Programiranje 1
Osnove programiranja kroz programski jezik C

Filip Marić i Predrag Janičić

Beograd, 2015.

Sadržaj

Predgovor

Ova knjiga pisana je kao udžbenik za predmet „Programiranje 1“ na smeru *Informatika* Matematičkog fakulteta u Beogradu. Nastala je na osnovu materijala za predavanja koja smo na ovom predmetu držali od 2006. godine.

U ovom predmetu i u ovoj knjizi, centralno mesto ima programski jezik C, ali predmet i knjiga nisu samo kurs ovog jezika, već pokušavaju da dâju šire osnove programiranja, ilustrovane kroz jedan konkretan jezik.

Na kraju poglavlja naveden je veći broj pitanja i zadataka koji mogu da služe za proveru znanja. Među ovim pitanjima su praktično sva pitanja koja su zadata na testovima i ispitima iz predmeta „Programiranje 1“ tokom prethodnih pet godina. Na kraju knjige navedena su rešenja zadataka. Odgovori na pitanja nisu navedeni i na kraju knjige jer su već sadržani u njoj. Na pitanja tipa „šta ispisuje naredni program“ nisu navođeni odgovori jer čitalac to može da proveri na svom računaru (i knjiga pokušava da ohrabri čitaoca da čitanje knjige kombinuje sa radom na računaru).

Autori

Deo I

Osnovni pojmovi računarstva i programiranja

Glava 1

Računarstvo i računarski sistemi

Računarstvo i informatika predstavljaju jednu od najatraktivnijih i najvažnijih oblasti današnjice. Život u savremenom društvu ne može se zamisliti bez korišćenja različitih računarskih sistema: stonih i prenosnih računara, tableta, pametnih telefona, ali i računara integrisanih u različite mašine (automobile, avione, industrijske mašine, itd). Definicija računarskog sistema je prilično široka. Može se reći da se danas pod digitalnim računarskim sistemom (računarom) podrazumeva mašina koja može da se programira da izvršava različite zadatke svođenjem na elementarne operacije nad brojevima. Brojevi se, u savremenim računarima, zapisuju u binarnom sistemu, kao nizovi nula i jedinica tj. binarnih cifara, tj. *bitova* (engl. bit, od *binary digit*). Koristeći n bitova, može se zapisati 2^n različitih vrednosti. Na primer, jedan *bajt* (B) označava osam bitova i može da reprezentuje 2^8, tj. 256 različitih vrednosti.[1]

Računarstvo se bavi izučavanjem računara, ali i opštije, izučavanjem teorije i prakse procesa računanja i primene računara u raznim oblastima nauke, tehnike i svakodnevnog života[2].

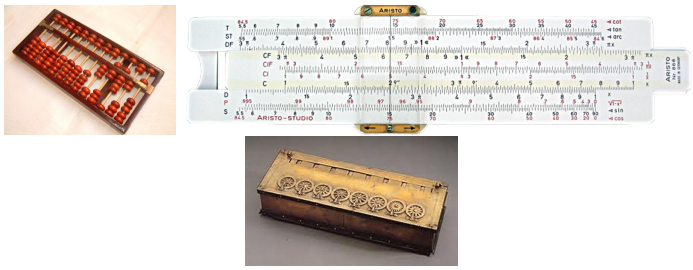
Računari u današnjem smislu nastali su polovinom XX veka, ali koreni računarstva su mnogo stariji od prvih računara. Vekovima su ljudi stvarali mehaničke i elektromehaničke naprave koje su mogle da rešavaju neke numeričke zadatke. Današnji računari su *programabilni*, tj. mogu da se isprogramiraju da vrše različite zadatke. Stoga je oblast *programiranja*, kojom se ova knjiga bavi, jedna od najznačajnijih oblasti računarstva. Za funkcionisanje modernih računara neophodni su i *hardver* i *softver*. Hardver (tehnički sistem računara) čine opipljive, fizičke komponente računara: procesor, memorija, matična ploča, hard disk, DVD uređaj, itd. Softver (programski sistem računara) čine računarski programi i prateći podaci koji određuju izračunavanja koja vrši računar. Računarstvo je danas veoma široka i dobro utemeljena naučna disciplina sa mnoštvom podoblasti.

[1] Količina podataka i kapacitet memorijskih komponenti savremenih računara obično se iskazuje u bajtovima ili izvedenim jedinicama. Obično se smatra da je jedan *kilobajt* (KB) jednak 1024 bajtova (mada neke organizacije podrazumevaju da je jedan KB jednak 1000 bajtova). Slično, jedan *megabajt* (MB) je jednak 1024 KB ili 1024^2 B, jedan *gigabajt* (GB) je jednak 1024 MB, a jedan *teraabajt* (TB) je jednak 1024 GB.

[2] Često se kaže da se računarstvo bavi računarima isto onoliko koliko se astronomija bavi teleskopima, a biologija mikroskopima. Računari nisu sami po sebi svrha i samo su sredstvo koje treba da pomogne u ostvarivanju različitih zadataka.

1.1 Rana istorija računarskih sistema

Programiranje u savremenom smislu postalo je praktično moguće tek krajem Drugog svetskog rata, ali je njegova istorija znatno starija. Prvi precizni postupci i sprave za rešavanje matematičkih problema postojali su još u vreme antičkih civilizacija. Na primer, kao pomoć pri izvođenju osnovnih matematičkih operacija korišćene su računaljke zvane *abakus*. U IX veku persijski matematičar *Al Horezmi*[3] precizno je opisao postupke računanja u indo-arapskom dekadnom brojevnom sistemu (koji i danas predstavlja najkorišćeniji brojevni sistem). U XIII veku *Leonardo Fibonači*[4] doneo je ovaj način zapisivanja brojeva iz Azije u Evropu i to je bio jedan od ključnih preduslova za razvoj matematike i tehničkih disciplina tokom renesanse. Otkriće logaritma omogućilo je svođenje množenja na sabiranje, dodatno olakšano raznovrsnim analognih spravama (npr. *klizni lenjir* — *šiber*)[5]. Prve mehaničke sprave koje su mogle da potpuno automatski izvode aritmetičke operacije i pomažu u rešavanju matematičkih zadataka su napravljene u XVII veku. *Blez Paskal*[6] konstruisao je 1642. godine mehaničke sprave, kasnije nazvane *Paskaline*, koje su služile za sabiranje i oduzimanje celih brojeva. *Gotfrid Lajbnic*[7] konstruisao je 1672. godine mašinu koja je mogla da izvršava sve četiri osnovne aritmetičke operacije (sabiranje, oduzimanje, množenje i deljenje) nad celim brojevima. Ova mašina bila je zasnovana na dekadnom brojevnom sistemu, ali Lajbnic je prvi predlagao i korišćenje binarnog brojevnog sistema u računanju.



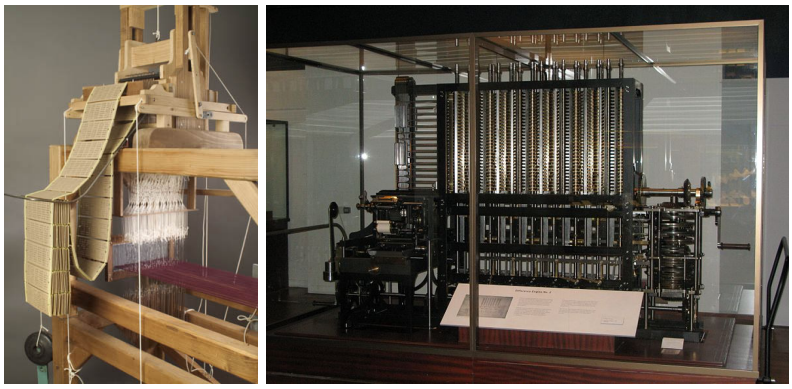
Slika 1.1: Abakus. Šiber. Paskalina

Mehanički uređaji. *Žozef Mari Žakard*[8] konstruisao je 1801. godine prvu programabilnu mašinu — mehanički tkački razboj koji je koristio bušene kartice kao svojevrsne programe za generisanje kompleksnih šara na tkanini. Svaka rupa na kartici određivala je jedan pokret mašine, a svaki red na kartici odgovarao je jednom redu šare.

U prvoj polovini XIX veka, *Čarls Bebidž*[9] dizajnirao je, mada ne i realizovao, prve programabilne računske mašine. Godine 1822. započeo je rad na

[3] Muhammad ibn Musa al-Khwarizmi (780–850), persijski matematičar.
[4] Leonardo Pisano Fibonacci, (1170–1250), italijanski matematičar iz Pize.
[5] Zanimljivo je da su klizni lenjiri nošeni na pet Apolo misija, uključujući i onu na Mesec, kako bi astronautima pomagali u potrebnim izračunavanjima.
[6] Blaise Pascal (1623–1662), francuski filozof, matematičar i fizičar. U njegovu čast jedan programski jezik nosi ime *PASCAL*.
[7] Gottfried Wilhelm Leibniz (1646–1716), nemački filozof i matematičar.
[8] Joseph Marie Jacquard (1752–1834), francuski trgovac.
[9] Charles Babbage (1791–1871), engleski matematičar, filozof i pronalazač.

diferencijskoj mašini koja je trebalo da računa vrednosti polinomijalnih funkcija (i eliminiše česte ljudske greške u tom poslu) u cilju izrade što preciznijih logaritamskih tablica. Ime je dobila zbog toga što je koristila tzv. metod konačnih razlika kako bi bila eliminisana potreba za množenjem i deljenjem. Mašina je trebalo da ima oko 25000 delova i da se pokreće ručno, ali nije nikada završena[10]. Ubrzo nakon što je rad na prvom projektu utihnuo bez rezultata, Bebidž je započeo rad na novoj mašini nazvanoj *analitička mašina*. Osnovna razlika u odnosu na sve prethodne mašine, koje su imale svoje specifične namene, bila je u tome što je analitička mašina zamišljena kao računska mašina opšte namene koja može da se *programira* (programima zapisanim na bušenim karticama, sličnim Žakardovim karticama). Program zapisan na karticama kontrolisao bi mehanički računar (pokretan parnom mašinom) i omogućavao sekvencijalno izvršavanje naredbi, grananje i skokove, slično programima za savremene računare. Osnovni delovi računara trebalo je da budu mlin (engl. mill) i skladište (engl. store), koji po svojoj funkcionalnosti sasvim odgovaraju procesoru i memoriji današnjih računara. *Ada Bajron*[11] zajedno sa Bebidžem napisala je prve programe za analitičku mašinu i, da je mašina uspešno konstruisana, njeni programi bi mogli da računaju određene složene nizove brojeva (takozvane Bernulijeve brojeve). Zbog ovoga se ona smatra prvim programerom u istoriji (i njoj u čast jedan programski jezik nosi ime *Ada*). Ona je bila i prva koja je uvidela da se računske mašine mogu upotrebiti i za nematematičke namene, čime je na neki način anticipirala današnje namene digitalnih računara.



Slika 1.2: Žakardov razboj. Bebidžova diferencijska mašina.

Elektromehanički uređaji. Elektromehanički uređaji za računanje koristili su se od sredine XIX veka do vremena Drugog svetskog rata.

Jedan od prvih je uređaj za čitanje bušenih kartica koji je konstruisao *Herman Holerit*[12]. Ovaj uređaj korišćen je 1890. za obradu rezultata popisa sta-

[10] Dosledno sledeći Bebidžev dizajn, 1991. godine (u naučno-popularne svrhe) uspešno je konstruisana diferencijska mašina koja radi besprekorno. Nešto kasnije, konstruisan je i „štampač“ koji je Bebidž dizajnirao za diferencijsku mašinu, tj. štamparska presa povezana sa parnom mašinom koja je štampala izračunate vrednosti.

[11] Augusta Ada King (rođ. Byron), Countess of Lovelace, (1815–1852), engleska matematičarka. U njenu čast nazvan je programski jezik ADA.

[12] Herman Hollerith (1860–1929), američki pronalazač.

novništva u SAD. Naime, obrada rezultata popisa iz 1880. godine trajala je više od 7 godina, a zbog naglog porasta broja stanovnika procenjeno je da bi obrada rezultata iz 1890. godine trajala više od 10 godina, što je bilo neprihvatljivo mnogo. Holerit je sproveo ideju da se podaci prilikom popisa zapisuju na mašinski čitljivom medijumu (na bušenim karticama), a da se kasnije obrađuju njegovom mašinom. Koristeći ovaj pristup obrada rezultata popisa uspešno je završena za godinu dana. Od Holeritove male kompanije kasnije je nastala čuvena kompanija *IBM*.

Godine 1941, *Konrad Cuze*[13] konstruisao je 22-bitni uređaj za računanje *Z3* koji je imao izvesne mogućnosti programiranja (podržane su bile petlje, ali ne i uslovni skokovi), te se često smatra i prvim realizovanim programabilnim računarom[14]. Cuzeove mašine tokom Drugog svetskog rata naišle su samo na ograničene primene. Cuzeova kompanija proizvela je oko 250 različitih tipova računara do kraja šezdesetih godina, kada je postala deo kompanije *Simens* (nem. Siemens).

U okviru saradnje kompanije IBM i univerziteta Harvard, tim *Hauarda Aikena*[15] završio je 1944. godine uređaj *Harvard Mark I*. Ovaj uređaj čitao je instrukcije sa bušene papirne trake, imao je preko 760000 delova, dužinu 17m, visinu 2.4m i masu 4.5t. Mark I mogao je da pohrani u memoriji (korišćenjem elektromehaničkih prekidača) 72 broja od po 23 dekadne cifre. Sabiranje i oduzimanje dva broja trajalo je trećinu, množenje šest, a deljenje petnaest sekundi.

Slika 1.3: Holeritova mašina. Harvard Mark I. ENIAC (proces reprogramiranja).

Elektronski računari. Elektronski računari koriste se od kraja 1930-ih do danas.

[13] Konrad Zuse (1910–1995), nemački inženjer.
[14] Uređaju Z3 prethoddili su jednostavniji uređaji Z1 i Z2, izgrađeni 1938. i 1940. godine.
[15] Howard Hathaway Aiken (1900–1973).

Jedan od prvih elektronskih računara *ABC* (specijalne namene — rešavanje sistema linearnih jednačina) napravili su 1939. godine *Atanasov*[16] i *Beri*[17]. Mašina je prva koristila binarni brojevni sistem i električne kondenzatore (engl. capacitor) za skladištenje bitova — sistem koji se u svojim savremenim varijantama koristi i danas u okviru tzv. DRAM memorije. Mašina nije bila programabilna.

Krajem Drugog svetskog rada, u Engleskoj, u *Blečli parku* (engl. Bletchley Park) u kojem je radio i *Alan Tjuring*[18], konstruisan je računar *Kolos (engl. Colossus)* namenjen dešifrovanju nemačkih poruka. Računar je omogućio razbijanje nemačke šifre zasnovane na mašini *Enigma*, zahvaljujući čemu su saveznici bili u stanju da prate komunikaciju nemačke podmorničke flote, što je značajno uticalo na ishod Drugog svetskog rata.

U periodu između 1943. i 1946. godine od strane američke vojske i tima univerziteta u Pensilvaniji koji su predvodili *Džon Mokli*[19] i *Džej Ekert*[20] konstruisan je prvi elektronski računar opšte namene — *ENIAC („Electronic Numerical Integrator and Calculator“).* Imao je 1700 vakuumskih cevi, dužinu 30m i masu 30t. Računske operacije izvršavao je hiljadu puta brže od elektromehaničkih uređaja. Osnovna svrha bila mu je jedna specijalna namena — računanje trajektorije projektila. Bilo je moguće da se mašina preprogramira i za druge zadatke ali to je zahtevalo intervencije na preklopnicima i kablovima koje su mogle da traju danima.

1.2 Računari fon Nojmanove arhitekture

Rane mašine za računanje nisu bile programabilne već su radile po unapred fiksiranom programu, određenom samom konstrukcijom mašine. Takva arhitektura se i danas koristi kod nekih jednostavnih mašina, na primer, kod kalkulatora („digitrona“). Da bi izvršavali nove zadatke, rani elektronski računari nisu programirani u današnjem smislu te reči, već su suštinski redizajnirani. Tako su, na primer, operaterima bile potrebne nedelje da bi prespojili kablove u okviru kompleksnog sistema ENIAC i tako ga instruisali da izvršava novi zadatak.

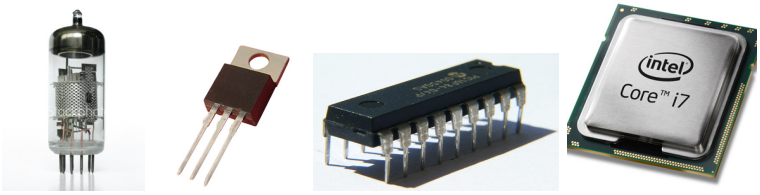
Potpuna konceptualna promena došla je kasnih 1940-ih, sa pojavom računara koji programe na osnovu kojih rade čuvaju u memoriji zajedno sa podacima — *računara sa skladištenim programima (engl. stored program computers).* U okviru ovih računara, postoji jasna podela na hardver i softver. Iako ideje za ovaj koncept datiraju još od Čarlsa Bebidža i njegove analitičke mašine i nastavljaju se kroz radove Tjuringa, Cuzea, Ekerta, Moklija, za rodonačelnika ovakve arhitekture računara smatra se *Džon fon Nojman*[21]. Fon Nojman se u ulozi konsultanta priključio timu Ekerta i Moučlija i 1945. godine je u svom izveštaju *EDVAC („Electronic Discrete Variable Automatic Computer“)* opisao arhitekturu koja se i danas koristi u najvećem broju savremenih računara i u kojoj se programi mogu učitavati isto kao i podaci koji se obrađuju. Računar EDVAC, naslednik računara ENIAC, koristio je binarni zapis brojeva,

[16] John Vincent Atanasoff (1903–1995).
[17] Clifford Edward Berry (1918–1963).
[18] Alan Turing (1912–1954), britanski matematičar.
[19] John William Mauchly (1907–1980).
[20] J. Presper Eckert (1919–1995).
[21] John Von Neumann (1903–1957), američki matematičar.

u memoriju je mogao da upiše hiljadu 44-bitnih podataka i bio je jedan od prvih računara koji su mogli da učitaju programe u memoriju. Iako je dizajn računara EDVAC bio prvi opis fon Nojmanove arhitekture, pre 1951. godine - kada je EDVAC pušten u rad, već je nekoliko računara slične arhitekture bilo konstruisano i funkcionalno njega (Mark 1 i EDSAC - 1949. godine i MESM u tadašnjem SSSR, 1950. godine).

Osnovni elementi fon Nojmanove arhitekture računara su *procesor* tj. *CPU (engl. Central Processing Unit)* i *glavna memorija*, koji su međusobno povezani. Ostale komponente računara (npr. ulazno-izlazne jedinice, spoljašnje memorije, ...) smatraju se pomoćnim i povezuju se na jezgro računara koje čine procesor i glavna memorija. Sva obrada podataka vrši se u procesoru. U memoriju se skladište podaci koji se obrađuju, ali i programi, predstavljeni nizom elementarnih instrukcija (kojima se procesoru zadaje koju akciju ili operaciju da izvrši). I podaci i programi se zapisuju obično kao binarni sadržaj i nema nikakve suštinske razlike između zapisa programa i zapisa podataka. Tokom rada, podaci i programi se prenose između procesora i memorije. S obzirom na to da i skoro svi današnji računari imaju fon Nojmanovu arhitekturu, način funkcionisanja ovakvih računara biće opisan detaljnije u poglavlju o savremenim računarskim sistemima.

Moderni programabilni računari se, po pitanju tehnologije koju su koristili, mogu grupisati u četiri generacije, sve zasnovane na fon Nojmanovoj arhitekturi.

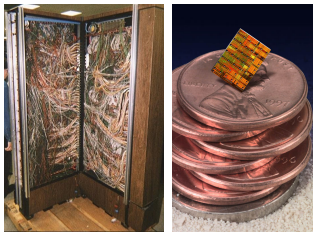


Slika 1.4: Osnovni gradivni elementi korišćeni u četiri generacije računara: vakuumska cev, tranzistor, integrisano kolo i mikroprocesor

I generacija računara (od kraja 1930-ih do kraja 1950-ih) koristila je *vakuumske cevi* kao logička kola i *magnetne doboše* (a delom i magnetne trake) za memoriju. Za programiranje su korišćeni mašinski jezik i asembler a glavne primene su bile vojne i naučne. Računari su uglavnom bili unikatni (tj. za većinu nije postojala serijska proizvodnja). Prvi realizovani računari fon Nojmanove arhitekture bili su *Mančesterska „Beba“ (engl. Manchester „Baby“)* — eksperimentalna mašina, razvijena 1949. na Univerzitetu u Mančesteru, na kojoj je testirana tehnologija vakuumskih cevi i njen naslednik *Mančesterski Mark 1 (engl. Manchester Mark 1)*, *EDSAC* — razvijen 1949. na Univerzitetu u Kembridžu, MESM razvijen 1950. na Kijevskom elektrotehničkom institutu i EDVAC koji je prvi dizajniran, ali napravljen tek 1951. na Univerzitetu u Pensilvaniji. Tvorci računara EDVAC, počeli su 1951. godine proizvodnju prvog komercijalnog računara *UNIVAC – UNIVersal Automatic Computer* koji je prodat u, za to doba neverovatnih, 46 primeraka.

II generacija računara (od kraja 1950-ih do polovine 1960-ih) koristila je *tranzistore* umesto vakuumskih cevi. Iako je tranzistor otkriven još 1947. godine, tek sredinom pedesetih počinje da se koristi umesto vakuumskih cevi kao osnovna elektronska komponenta u okviru računara. Tranzistori su izgrađeni od tzv. *poluprovodničkih elemenata* (obično silicijuma ili germanijuma). U poređenju sa vakuumskih cevima, tranzistori su manji, zahtevaju manje energije te se manje i greju. Tranzistori su unapredili ne samo procesore i memoriju već i spoljašnje uređaje. Počeli su da se široko koriste magnetni diskovi i trake, započelo je umrežavanja računara i čak korišćenje računara u zabavne svrhe (implementirana je prva računarska igra *Spacewar* za računar *PDP-1*). U ovo vreme razvijeni su i prvi jezici višeg nivoa (FORTRAN, LISP, ALGOL, COBOL). U to vreme kompanija IBM dominirala je tržištem — samo računar *IBM 1401*, prodat u više od deset hiljada primeraka, pokrivao je oko trećinu tada postojećeg tržišta.

III generacija računara (od polovine 1960-ih do sredine 1970-ih) bila je zasnovana na *integrisanim kolima* smeštenim na silicijumskim *(mikro)čipovima*. Prvi računar koji je koristio ovu tehnologiju bio je IBM 360, napravljen 1964. godine.

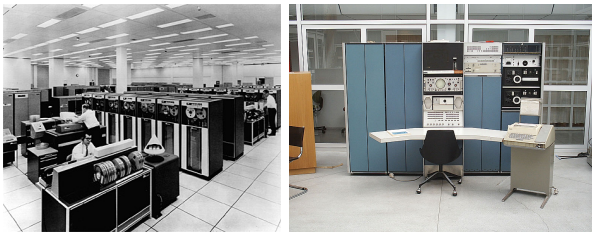


Slika 1.5: Integrisana kola dovela su do minijaturizacije i kompleksni žičani spojevi su mogli biti realizovani na izuzetno maloj površini.

Nova tehnologija omogućila je poslovnu primenu računara u mnogim oblastima. U ovoj eri dominirali su *mejnfrejm (engl. mainframe)* računari koji su bili izrazito moćni za to doba, čija se brzina merila milionima instrukcija u sekundi (engl. MIPS; na primer, neki podmodeli računara IBM 360 imali su brzinu od skoro 1 MIPS) i koji su imali mogućnost skladištenja i obrade velike količine podataka te su korišćeni od strane vlada i velikih korporacija za popise, statističke obrade i slično. Kod računara ove generacije uveden je sistem *deljenja vremena (engl. timesharing)* koji dragoceno procesorsko vreme raspodeljuje i daje na uslugu različitim korisnicima koji istovremeno rade na računaru i komuniciraju sa njim putem specijalizovanih *terminala*. U ovo vreme uvedeni su prvi standardi za jezike višeg nivoa (npr. ANSI FORTRAN). Korišćeni su različiti operativni sistemi, uglavnom razvijeni u okviru kompanije IBM. Sa udelom od 90%, kompanija IBM je imala apsolutnu dominaciju na tržištu ovih računara.

Pored mejnfrejm računara, u ovom periodu široko su korišćeni i *mini računari (engl. minicomputers)* koji se mogu smatrati prvim oblikom ličnih (personalnih) računara. Procesor je, uglavnom, bio na raspolaganju isključivo jednom korisniku. Obično su bili veličine ormana i retko su ih posedovali pojedinci (te

se ne smatraju kućnim računarima). Tržištem ovih računara dominirala je kompanija *DEC – Digital Equipment Corporation* sa svojim serijama računara poput *PDP-8* i *VAX*. Za ove računare, obično se vezuje operativni sistem *Unix* i programski jezik *C* razvijeni u *Belovim laboratorijama (engl. Bell Laboratories)*, a često i *hakerska*[22] kultura nastala na univerzitetu *MIT (engl. Massachusetts Institute of Technology)*.



Slika 1.6: Mejnfrejm računar: IBM 7094. Mini računar: DEC PDP 7

IV generacija računara (od ranih 1970-ih) zasnovana je na visoko integrisanim kolima kod kojih je na hiljade kola smešeno na jedan silikonski čip. U kompaniji Intel 1971. godine napravljen je prvi mikroprocesor *Intel 4004* — celokupna centralna procesorska jedinica bila je smeštena na jednom čipu. Iako prvobitno namenjena za ugradnju u kalkulatore, ova tehnologija omogućila je razvoj brzih a malih računara pogodnih za kućnu upotrebu.

Časopis *Popular electronics* nudio je 1975. godine čitaocima mogućnost naručivanja delova za sklapanje mikroračunara *MITS Altair 8800* zasnovanog na procesoru *Intel 8080* (nasledniku procesora *Intel 4004*). Interesovanje među onima koji su se elektronikom bavili iz hobija bio je izuzetno pozitivan i samo u prvom mesecu prodato je nekoliko hiljada ovih „uradi-sâm“ računara. Smatra se da je Altair 8800 bio inicijalna kapisla za „revoluciju mikroračunara“ koja je usledila narednih godina. Altair se vezuje i za nastanak kompanije *Microsoft* — danas jedne od dominantnih kompanija u oblasti proizvodnje softvera. Naime, prvi proizvod kompanije *Microsoft* bio je interpretator za programski jezik *BASIC* za Altair 8800.

Nakon Altaira pojavljuje se još nekoliko računarskih kompleta na sklapanje. Prvi mikroračunar koji je prodavan već sklopljen bio je *Apple*, na čijim temeljima je nastala istoimena kompanija, danas jedan od lidera na tržištu računarske opreme.

Kućni računari koristili su se sve više — uglavnom od strane entuzijasta — za jednostavnije obrade podataka, učenje programiranja i igranje računarskih igara. Kompanija *Commodore* je 1977. godine predstavila svoj računarom *Commodore PET* koji je zabeležio veliki uspeh. *Commodore 64*, jedan od najuspešnijih računara za kućnu upotrebu, pojavio se 1982. godine. Iz iste kompanije je i serija *Amiga* računara sa kraja 1980-ih i početka 1990-ih. Pored kompanije *Commodore*, značajni proizvođači računara toga doba bili su i

[22] Termin *haker* se obično koristi za osobe koje neovlašćeno pristupaju računarskim sistemima, ali *hakeraj* kao programerska podkultura podrazumeva anti-autoritaran pristup razvoju softvera, obično povezan sa pokretom za slobodan softver. U oba slučaja, hakeri su pojedinici koji na inovativan način modifikuju postojeće hardverske i softverske sisteme.

Sinclair (sa veoma popularnim modelom *ZX Spectrum*), *Atari*, *Amstrad*, itd. Kućni računari ove ere bili su obično jeftini, imali su skromne karakteristike i najčešće koristili kasetofone i televizijske ekrane kao ulazno-izlazne uređaje.

Slika 1.7: Prvi mikroprocesor: Intel 4004. Naslovna strana časopisa *„Popular electronics“* sa Altair 8800. Commodore 64. IBM PC 5150.

Najznačajnija računarska kompanija toga doba — IBM — uključila se na tržište kućnih računara 1981. godine, modelom *IBM PC 5150*, poznatijem jednostavno kao *IBM PC* ili *PC* (*engl. Personal computer*). Zasnovan na Intelovom procesoru *Intel 8088*, ovaj računar veoma brzo je zauzeo tržište računara za ličnu poslovnu upotrebu (obrada teksta, tabelarna izračunavanja, ...). Prateći veliki uspeh IBM PC računara, pojavio se određen broj *klonova* — računara koji nisu proizvedeni u okviru kompanije IBM, ali koji su kompatibilni sa IBM PC računarima. PC arhitektura vremenom je postala standard za kućne računare. Sredinom 1980-ih, pojavom naprednijih grafičkih (VGA) i zvučnih (SoundBlaster) kartica, IBM PC i njegovi klonovi stekli su mogućnost naprednih multimedijalnih aplikacija i vremenom su sa tržišta istisli sve ostale proizvođače. I naslednici originalnog IBM PC računara (IBM PC/XT, IBM PC/AT, ...) bili su zasnovani na Intelovim procesorima, pre svega na *x86* seriji (Intel 80286, 80386, 80486) i zatim na seriji *Intel Pentium* procesora. Operativni sistem koji se tradicionalno vezuju uz PC računare dolaze iz kompanije *Microsoft* — prvo *MS DOS*, a zatim *MS Windows*. PC arhitektura podržava i korišćenje drugih operativnih sistema (na primer, GNU/Linux).

Jedini veliki konkurent IBM PC arhitekturi koji se sve vreme održao na tržištu (pre svega u SAD) je serija računara *Macintosh* kompanije *Apple*. *Macintosh*, koji se pojavio 1984., je prvi komercijalni kućni računar sa grafičkim korisničkim interfejsom i mišem. Operativni sistem koji se i danas koristi na Apple računarima je *Mac OS*.

Iako su prva povezivanja udaljenih računara izvršena još krajem 1960-ih godina, pojavom *interneta (engl. internet)* i *veba (engl. World Wide Web — WWW)*, većina računara postaje međusobno povezana sredinom 1990-ih godina. Danas se veliki obim poslovanja izvršava u internet okruženju, a domen korišćenja računara je veoma širok. Došlo je do svojevrsne informatičke revolucije koja je promenila savremeno društvo i svakodnevni život. Na primer,

tokom prve decenije XXI veka došlo je do pojave *društvenih mreža (engl. social networks)* koje postepeno preuzimaju ulogu osnovnog medijuma za komunikaciju.

Slika 1.8: Stoni računar. Prenosni računar: IBM ThinkPad. Tablet: Apple Ipad 2. Pametni telefon: Samsung Galaxy S2.

Tržištem današnjih računara dominiraju računari zasnovani na *PC* arhitekturi i *Apple Mac* računari. Pored stonih (engl. desktop) računara popularni su i prenosni (engl. notebook ili laptop) računari. U najnovije vreme, javlja se trend *tehnološke konvergencije* koja podrazumeva stapanje različitih uređaja u jedinstvene celine, kao što su *tableti (engl. tablet)* i *pametni telefoni (engl. smartphone)*. Operativni sistemi koji se danas uglavnom koriste na ovim uređajima su *IOS* kompanije Apple, kao i *Android* kompanije Google.

Pored ličnih računara u IV generaciji se i dalje koriste mejnfrejm računari (na primer, IBM Z serija) i superračunari (zasnovani na hiljadama procesora). Na primer, kineski superračunar Tianhe-2 radi brzinom od preko 30 petaflopsa (dok prosečni lični računar radi brzinom reda 10 gigaflopsa).[23]

1.3 Oblasti savremenog računarstva

Savremeno računarstvo ima mnogo podoblasti, kako praktičnih, tako i teorijskih. Zbog njihove isprepletenosti nije jednostavno sve te oblasti sistematizovati i klasifikovati. U nastavku je dat spisak nekih od oblasti savremenog računarstva (u skladu sa klasifikacijom američke asocijacije ACM — *Association for Computing Machinery*, jedne od najvećih i najuticajnijih računarskih zajednica):

- *Algoritmika* (procesi izračunavanja i njihova složenost);

[23] Flops je mera računarskih performansi, posebno pogodna za izračunavanja nad brojevima u pokretnom zarezu (i pogodnija nego generička mera koja se odnosi na broj instrukcija u sekundi). Broj flopsa govori koliko operacija nad brojevima u pokretnom zarezu može da izvrši računar u jednoj sekundi. Brzina današnjih računara se obično izražava u gigaflopsima (10^9 flopsa), teraflopsima (10^12 flopsa) i petaflopsima (10^15 flopsa).

- *Strukture podataka* (reprezentovanje i obrada podataka);
- *Programski jezici* (dizajn i analiza svojstava formalnih jezika za opisivanje algoritama);
- *Programiranje* (proces zapisivanja algoritama u nekom programskom jeziku);
- *Softversko inženjerstvo* (proces dizajniranja, razvoja i testiranja programa);
- *Prevođenje programskih jezika* (efikasno prevođenje jezika, obično na mašinski jezik);
- *Operativni sistemi* (sistemi za upravljanje računarom i programima);
- *Mrežno računarstvo* (algoritmi i protokoli za komunikaciju između računara);
- *Primene* (dizajn i razvoj softvera za svakodnevnu upotrebu);
- *Istraživanje podataka* (pronalaženje relevantnih informacija u velikim skupovima podataka);
- *Veštačka inteligencija* (rešavanje problema u kojima se javlja kombinatorna eksplozija);
- *Robotika* (algoritmi za kontrolu ponašanja robota);
- *Računarska grafika* (analiza i sinteza slika i animacija);
- *Kriptografija* (algoritmi za zaštitu privatnosti podataka);
- *Teorijsko računarstvo* (teorijske osnove izračunavanja, računarska matematika, verifikacija softvera, itd).

1.4 Hardver savremenih računara

Hardver čine opipljive, fizičke komponente računara. Iako je u osnovi savremenih računarskih sistema i dalje fon Nojmanova mašina (procesor i memorija), oni se danas ne mogu zamisliti bez niza hardverskih komponenti koje olakšavaju rad sa računarom.

Iako na prvi pogled deluje da se jedan uobičajeni stoni računar sastoji od kućišta, monitora, tastature i miša, ova podela je veoma površna, podložna promenama (već kod prenosnih računara, stvari izgledaju znatno drugačije) i nikako ne ilustruje koncepte bitne za funkcionisanje računara. Mnogo značajnija je podela na osnovu koje računar čine:

- *centralna procesorska jedinica* (engl. central processing unit, CPU), koja obrađuje podatke;
- *glavna memorija* (engl. main memory), u kojoj se istovremeno čuvaju i podaci koji se obrađuju i trenutno pokrenuti programi (takođe zapisani binarno, u obliku podataka);

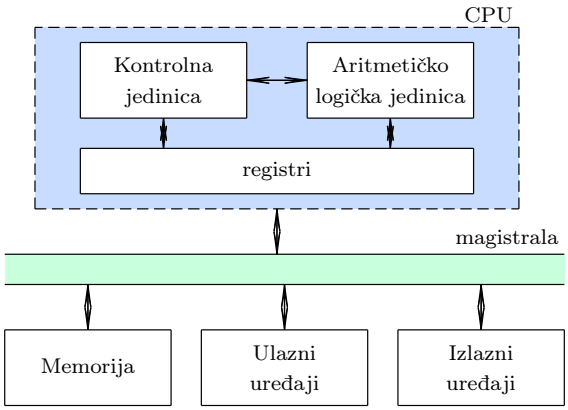
- različiti *periferijski uređaji* ili *ulazno-izlazne jedinice* (engl. peripherals, input-output devices, IO devices), kao što su miševi, tastature, ekrani, štampači, diskovi, a koje služe za komunikaciju korisnika sa sistemom i za trajno skladištenje podataka i programa.

Sve nabrojane komponente međusobno su povezane i podaci se tokom rada računara prenose od jedne do druge. Veza između komponenti uspostavlja se hardverskim sklopovima koji se nazivaju *magistrale* (engl. bus). Magistrala obuhvata provodnike koji povezuju uređaje, ali i čipove koji kontrolišu protok podataka. Svi periferijski uređaji se sa memorijom, procesorom i magistralama povezuju hardverskim sklopovima koji se nazivaju *kontroleri*. *Matična ploča* (engl. motherboard) je štampana ploča na koju se priključuju procesor, memorijski čipovi i svi periferijski uređaji. Na njoj se nalaze čipovi magistrale, a danas i mnogi kontroleri periferijskih uređaja. Osnovu hardvera savremenih računara, dakle, čine sledeće komponente:

Procesori - Centralna procesorska jedinica (tj. procesor), jedna od dve centralne komponente svakog računarskog sistema fon Nojmanove arhitekture, danas je obično realizovana na pojedinačnom čipu – mikroprocesoru. Procesor se sastoji od *kontrolne jedinice (engl. Control Unit)* koja upravlja njegovim radom i *aritmetičko-logičke jedinice (engl. Arithmetic Logic Unit)* koja je zadužena za izvođenje aritmetičkih operacija (sabiranje, oduzimanje, množenje, poređenje, ...) i logičkih operacija (konjunkcija, negacija, ...) nad brojevima. Procesor sadrži i određeni, manji broj, *registara* koji privremeno mogu da čuvaju podatke. Registri su obično fiksirane širine (8 bitova, 16 bitova, 32 bita, 64 bita). Komunikacija sa memorijom se ranije vršila isključivo preko specijalizovanog registra koji se nazivao akumulator. Aritmetičko logička jedinica sprovodi operacije nad podacima koji su smešteni u registrima i rezultate ponovo smešta u registre. Kontrolna jedinica procesora čita instrukciju po instrukciju programa zapisanog u memoriji i na osnovu njih određuje sledeću akciju sistema (na primer, izvrši prenos podataka iz procesora na određenu memorijsku adresu, izvrši određenu aritmetičku operaciju nad sadržajem u registrima procesora, uporedi sadržaje dva registra i ukoliko su jednaki izvrši instrukciju koja se nalazi na zadatoj memorijskoj adresi i slično). Brzina procesora meri se u milionima operacija u sekundi (engl. Million Instructions Per Second, MIPS) tj. pošto su operacije u pokretnom zarezu najzahtevnije, u broju operacija u pokretnom zarezu u sekundi (engl. FLoating Point Operations per Second, FLOPS). Današnji standardni procesori rade oko 10 GFLOPS (deset milijardi operacija u pokretnom zarezu po sekundi).

Važne karakteristike procesora danas su broj jezgara (obično 1, 2 ili 4), širina reči (obično 32 bita ili 64 bita) i radni takt (obično nekoliko gigaherca (GHz)) — veći radni takt obično omogućava izvršavanje većeg broja operacija u jedinici vremena.

Memorijska hijerarhija - Druga centralna komponenta fon Nojmanove arhitekture je *glavna memorija* u koju se skladište podaci i programi. Memorija je linearno uređeni niz registara (najčešće bajtova), pri čemu svaki registar ima svoju adresu. Kako se kod ove memorije sadržaju može pristupati u slučajnom redosledu (bez unapred fiksiranog redosleda), ova



Slika 1.9: Shema računara fon Nojmanove arhitekture

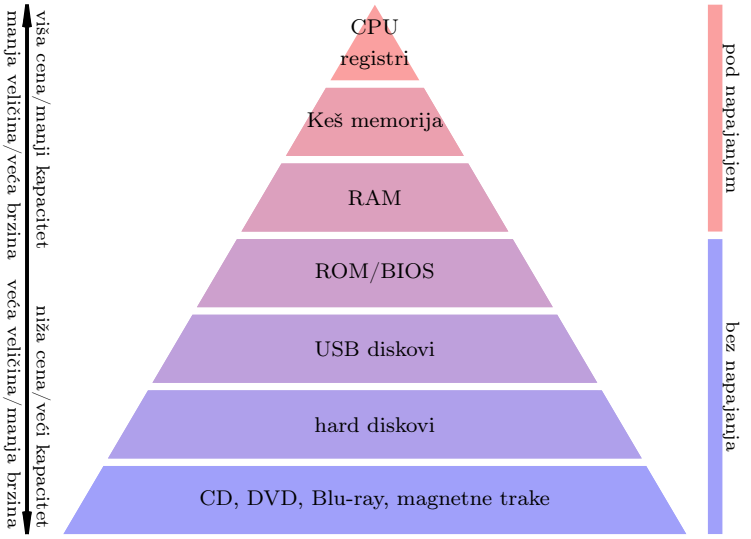
memorija se često naziva i *memorija sa slobodnim pristupom* (engl. random access memory, RAM). Osnovni parametri memorija su *kapacitet* (danas obično meren gigabajtima (GB)), *kašnjenje* koje izražava vreme potrebno da se memorija pripremi za čitanje odnosno upis podataka (danas obično mereno u nanosekundama (ns)), kao i *protok* koji izražava količinu podataka koji se prenose po jedinici merenja (danas obično mereno u GBps).

U savremenim računarskim sistemima, uz glavnu memoriju uspostavlja se čitava *hijerarhija memorija* koje služe da unaprede funkcionisanje sistema. Memorije neposredno vezane za procesor koje se koriste isključivo dok je računar uključen nazivaju se *unutrašnje memorije*, dok se memorije koje se koriste za skladištenje podataka u trenucima kada računar nije uključen nazivaju *spoljne memorije*. Procesor obično nema načina da direktno koristi podatke koji se nalaze u spoljnim memorijama (jer su one znatno sporije od unutrašnjih), već se pre upotrebe svi podaci prebacuju iz spoljnih u unutrašnju memoriju.

Memorijska hijerarhija predstavlja se piramidom. Od njenog vrha ka dnu opadaju kvalitet i brzina memorija, ali zato se smanjuje i cena, pa se kapacitet povećava.

Registri procesora predstavljaju najbržu memoriju jer se sve aritmetičke i logičke operacije izvode upravo nad podacima koji se nalaze u njima.

Keš (engl. cache) je mala količina brze memorije (nekoliko hiljada puta manjeg kapaciteta od glavne memorije; obično nekoliko megabajta) koja se postavlja između procesora i glavne memorije u cilju ubrzanja rada računara. Keš se uvodi jer su savremeni procesori postali znatno brži od glavnih memorija. Pre pristupa glavnoj memoriji procesor uvek prvo pristupa kešu. Ako traženi podatak tamo postoji, u pitanju je tzv. pogodak keša (engl. cache hit) i podatak se dostavlja procesoru. Ako se podatak ne nalazi u kešu, u pitanju je tzv. promašaj keša (engl. cache miss) i podatake se iz glavne memorije prenosi u keš zajedno sa određenim brojem podataka koji za njim slede (glavni faktor brzine glavne memorije



Slika 1.10: Memorijska hijerarhija

je njeno kašnjenje i praktično je svejedno da li se prenosi jedan ili više podataka jer je vreme prenosa malog broja bajtova mnogo manje od vremena kašnjenja). Motivacija ovog pristupa je u tome što programi često pravilno pristupaju podacima (obično redom kojim su podaci smešteni u memoriji), pa je velika verovatnoća da će se naredni traženi podaci i instrukcije naći u keš-memoriji.

Glavna memorija čuva sve podatke i programe koje procesor izvršava. Mali deo glavne memorije čini ROM (engl. read only memory) – nepromenljiva memorija koja sadrži osnovne programe koji služe za kontrolu određenih komponenata računara (na primer, osnovni ulazno-izlazni sistem BIOS). Znatno veći deo glavne memorije čini RAM – privremena promenljiva memorija sa slobodnim pristupom. Terminološki, podela glavne memorije na ROM i RAM nije najpogodnija jer ove vrste memorije nisu suštinski različite — nepromenljivi deo (ROM) je takođe memorija sa slobodnim pristupom (RAM). Da bi RAM memorija bila što brža, izrađuje se uvek od poluprovodničkih (elektronskih) elemenata. Danas se uglavnom realizuje kao sinhrona dinamička memorija (SDRAM). To znači da se prenos podataka između procesora i memorije vrši u intervalima određenim otkucajima sistemskog sata (često se u jednom otkucaju izvrši nekoliko prenosa). Dinamička memorija je znatno jeftinija i jednostavnija, ali zato sporija od statičke memorije od koje se obično gradi keš.

Spoljne memorije čuvaju podatke trajno, i kada računar ostane bez električnog napajanja. Kao centralna spoljna skladišta podataka uglavnom se koriste *hard diskovi (engl. hard disk)* koji čuvaju podatke korišćenjem magnetne tehnologije, a u novije vreme se sve više koriste i *SSD uređaji (engl. solid state drive)* koji čuvaju podatke korišćenjem elektronskih

tzv. fleš memorija (engl. flash memory). Kao prenosne spoljne memorije koriste se uglavnom *USB fleš-memorije* (izrađene u sličnoj tehnologiji kao i SSD) i *optički diskovi* (CD, DVD, Blu-ray).

Ulazni uređaji. Osnovni ulazni uređaji današnjih računara su *tastature* i *miševi*. Prenosni računari imaju ugrađ enu tastaturu, a umesto miša može se koristiti tzv. *tačped* (engl. touchpad). Tastature i miševi se sa računarom povezuju ili kablom (preko PS/2 ili USB priključaka) ili bežično (najčešće korišćenjem BlueTooth veze). Ovo su uglavnom standardizovani uređaji i nema velikih razlika među njima. *Skeneri* sliku sa papira prenose u računar. Princip rada je sličan digitalnom fotografisanju, ali prilagođen slikanju papira.

Izlazni uređaji. Osnovni izlazni uređaji savremenih računara su monitori. Danas dominiraju monitori tankog i ravnog ekrana (engl. flat panel display), zasnovani obično na tehnologiji tečnih kristala (engl. liquid crystal display, LCD) koji su osvetljeni pozadinskim LED osvetljenjem. Ipak, još uvek su ponegde u upotrebi i monitori sa katodnom cevi (engl. cathode ray tube, CRT). Grafički kontroleri koji služe za kontrolu slike koja se prikazuje na monitoru ili projektoru danas su obično integrisani na matičnoj ploči, a ponekad su i na istom čipu sa samim procesorom (engl. Accelerated Processing Unit, APU).

Što se tehnologije štampe tiče, danas su najzastupljeniji laserski štampači i inkdžet štampači (engl. inkjet). Laserski štampači su češće crno-beli, dok su ink-džet štampači obično u boji. Sve su dostupniji i 3D štampači.

1.5 Softver savremenih računara

Softver čine računarski programi i prateći podaci koji određuju izračunavanja koje vrši računar. Na prvim računarima moglo je da se programira samo na *mašinski zavisnim programskim jezicima* — na jezicima specifičnim za konkretnu mašinu na kojoj program treba da se izvršava. Polovinom 1950-ih nastali su prvi *jezici višeg nivoa* i oni su drastično olakšali programiranje. Danas se programi obično pišu u *višim programskim jezicima* a zatim prevode na *mašinski jezik* — jezik razumljiv računaru. Bez obzira na to kako je nastao, da bi mogao da se izvrši na računaru, program mora da budu smešten u memoriju u obliku binarno zapisanih podataka (tj. u obliku niza nula i jedinica) koji opisuju instrukcije koje su neposredno podržane arhitekturom računara. U ovom poglavlju biće prikazani osnovni principa rada računara kroz nekoliko jednostavnih primera programa.

1.5.1 Primeri opisa izračunavanja

Program specifikuje koje operacije treba izvršiti kako bi se rešio neki zadatak. Principi rada programa mogu se ilustrovati na primeru nekoliko jednostavnih izračunavanja i instrukcija koje ih opisuju. Ovi opisi izračunavanja dati su u vidu prirodno-jezičkog opisa ali direktno odgovaraju i programima na višim programskim jezicima.

Kao prvi primer, razmotrimo izračunavanje vrednosti $2x + 3$ za datu vrednost x. U programiranju (slično kao i u matematici) podaci se predstavljaju

promenljivama. Međutim, promenljive u programiranju (za razliku od matematike) vremenom mogu da menjaju svoju vrednost (tada kažemo da im se *dodeljuje nova vrednost*). U programiranju, svakoj promenljivoj pridruženo je (jedno, fiksirano) mesto u memoriji i tokom izvršavanja programa promenljiva može da menja svoju vrednost, tj. sadržaj dodeljenog memorijskog prostora. Ako je promenljiva čija je vrednost ulazni parametar označena sa x, a promenljiva čija je vrednost rezultat izračunavanja označena sa y, onda se pomenuto izračunavanje može opisati sledećim jednostavnim opisom.

```
y := 2*x + 3
```

Simbol `*` označava množenje, `+` sabiranje, a `:=` označava da se promenljivoj sa njene leve strane dodeljuje vrednost izraza sa desne strane.

Kao naredni primer, razmotrimo određivanje većeg od dva data broja. Računari (tj. njihovi procesori) obično imaju instrukcije za poređenje brojeva, ali određivanje vrednosti većeg broja zahteva nekoliko koraka. Pretpostavimo da promenljive x i y sadrže dve brojevne vrednosti, a da promenljiva m treba da dobije vrednost veće od njih. Ovo izračunavanje može da se izrazi sledećim opisom.

```
ako je x >= y onda
    m := x
inače
    m := y
```

Kao malo komplikovaniji primer razmotrimo stepenovanje. Procesori skoro uvek podržavaju instrukcije kojima se izračunava zbir i proizvod dva cela broja, ali stepenovanje obično nije podržano kao elementarna operacija. Složenije operacije se mogu ostvariti korišćenjem jednostavnijih. Na primer, n-ti stepen broja x (tj. vrednost x^n) moguće je izračunati uzastopnom primenom množenja: ako se krene od broja 1 i n puta sa pomnoži brojem x, rezultat će biti x^n. Da bi moglo da se osigura da će množenje biti izvršeno tačno n puta, koristi se brojačka promenljiva i koja na početku dobija vrednost 0, a zatim se, prilikom svakog množenja, uvećava sve dok ne dostigne vrednost n. Ovaj postupak možemo predstaviti sledećim opisom.

```
s := 1, i := 0
dok je i < n radi sledeće:
    s := s·x, i := i+1
```

Kada se ovaj postupak primeni na vrednosti $x = 3$ i $n = 2$, izvodi se naredni niz koraka.

`s := 1,`	`i := 0,`	pošto je `i(=0)` manje od `n(=2)` vrše se dalje operacije
`s := s·x = 1·3 = 3,`	`i := i+1 = 0+1 = 1,`	pošto je `i(=1)` manje od `n(=2)` vrše se dalje operacije
`s := s·x = 3·3 = 9,`	`i := i+1 = 1+1 = 2,`	pošto `i(=2)` nije manje od `n(=2)`, ne vrše se dalje operacije.

1.5.2 Mašinski programi

Mašinski programi su neposredno vezani za procesor računara na kojem se koriste — procesor je konstruisan tako da može da izvršava određene elementarne naredbe. Ipak, razvoj najvećeg broja procesora usmeren je tako da se isti mašinski programi mogu koristiti na čitavim familijama procesora.

Primitivne instrukcije koje podržava procesor su veoma malobrojne i jednostavne (na primer, postoje samo instrukcije za sabiranje dva broja, konjunkcija bitova, instrukcija skoka i slično) i nije lako kompleksne i apstraktne algoritme izraziti korišćenjem tog uskog skupa elementarnih instrukcija. Ipak, svi zadaci koje računari izvršavaju svode se na ove primitivne instrukcije.

Asemblerski jezici. Asemblerski (ili simbolički) jezici su jezici koji su veoma bliski mašinskom jeziku računara, ali se, umesto korišćenja binarnog sadržaja za zapisivanje instrukcija koriste (mnemotehničke, lako pamtljive) simboličke oznake instrukcija (tj. programi se unose kao tekst). Ovim se, tehnički, olakšava unos programa i programiranje (programer ne mora da direktno manipuliše binarnim sadržajem), pri čemu su sve mane mašinski zavisnog programiranja i dalje prisutne. Kako bi ovako napisan program mogao da se izvršava, neophodno je izvršiti njegovo prevođenje na mašinski jezik (tj. zapisati instrukcije binarnom azbukom) i uneti na odgovarajuće mesto u memoriji. Ovo prevođenje je jednostavno i jednoznačno i vrše ga jezički procesori koji se nazivaju *asembleri*.

Sva izračunavanja u primerima iz poglavlja 1.5.1 su opisana neformalno, kao uputstva čoveku a ne računaru. Da bi se ovako opisana izračunavanja mogla sprovesti na nekom računaru fon Nojmanove arhitekture neophodno je opisati ih preciznije. Svaka elementarna operacija koju procesor može da izvrši u okviru programa zadaje se *procesorskom instrukcijom* — svaka instrukcija instruiše procesor da izvrši određenu operaciju. Svaki procesor podržava unapred fiksiran, konačan *skup instrukcija* (engl. instruction set). Svaki program računara predstavljen je nizom instrukcija i skladišti se u memoriji računara. Naravno, računari se razlikuju (na primer, po tome koliko registara u procesoru imaju, koje instrukcije može da izvrši njihova aritmetičko-logička jedinica, koliko memorije postoji na računaru, itd). Međutim, da bi se objasnili osnovni principi rada računara nije neophodno razmatrati neki konkretan računar, već se može razmatrati neki hipotetički računar. Pretpostavimo da procesor sadrži tri registra označena sa `ax`, `bx` i `cx` i još nekoliko izdvojenih bitova (tzv. zastavica). Dalje, pretpostavimo da procesor može da izvršava naredne *aritmetičke instrukcije* (zapisane ovde u asemblerskom obliku):

- Instrukcija `add ax, bx` označava operaciju sabiranja vrednosti brojeva koji se nalaze u registrima `ax` i `bx`, pri čemu se rezultat sabiranja smešta u registar `ax`. Operacija `add` može se primeniti na bilo koja dva registra.
- Instrukcija `mul ax, bx` označava operaciju množenja vrednosti brojeva koji se nalaze u registrima `ax` i `bx`, pri čemu se rezultat množenja smešta u registar `ax`. Operacija `mul` može se primeniti na bilo koja dva registra.
- Instrukcija `cmp ax, bx` označava operaciju poređenja vrednosti brojeva koji se nalaze u registrima `ax` i `bx` i rezultat pamti postavljanjem zastavice u procesoru. Operacija `cmp` se može primeniti na bilo koja dva registra.

Program računara je niz instrukcija koje se obično izvršavaju redom, jedna za drugom. Međutim, pošto se javlja potreba da se neke instrukcije ponove veći broj puta ili da se određene instrukcije preskoče, uvode se *instrukcije skoka*. Kako bi se moglo specifikovati na koju instrukciju se vrši skok, uvode se *labele* – označena mesta u programu. Pretpostavimo da naš procesor može da izvršava sledeće dve vrste skokova (bezuslovne i uslovne):

- Instrukcija `jmp label`, gde je `label` neka labela u programu, označava bezuslovni skok koji uzrokuje nastavak izvršavanja programa od mesta u programu označenog navedenom labelom.
- Uslovni skokovi prouzrokuju nastavak izvršavanja programa od instrukcije označene navedenom labelom, ali samo ako je neki uslov ispunjen. Ukoliko uslov nije ispunjen, izvršava se naredna instrukcija. U nastavku će se razmatrati samo instrukcija `jge label`, koja uzrokuje uslovni skok na mesto označeno labelom `label` ukoliko je vrednost prethodnog poređenja brojeva bila *veće ili jednako*.

Tokom izvršavanja programa, podaci se nalaze u memoriji i u registrima procesora. S obzirom na to da procesor sve operacije može da izvrši isključivo nad podacima koji se nalaze u njegovim registrima, svaki procesor podržava i *instrukcije prenosa podataka* između memorije i registara procesora (kao i između samih registara). Pretpostavimo da naš procesor podržava sledeću instrukciju ove vrste.

- Instrukcija `mov` označava operaciju prenosa podataka i ima dva parametra — prvi određuje gde se podaci prenose, a drugi koji određuje koji se podaci prenose. Parametar može biti ime registra (što označava da se pristupa podacima u određenom registru), broj u zagradama (što označava da se pristupa podacima u memoriji i to na adresi određenoj brojem u zagradama) ili samo broj (što označava da je podatak baš taj navedeni broj). Na primer, instrukcija `mov ax bx` označava da se sadržaj registra `bx` prepisuje u registar `ax`, instrukcija `mov ax, [10]` označava da se sadržaj iz memorije sa adrese `10` prepisuje u registar `ax`, instrukcija `mov ax, 1` označava da se u registar ax upisuje vrednost `1`, dok instrukcija označava `mov [10], ax` da se sadržaj registra `ax` upisuje u memoriju na adresu `10`.

Sa ovakvim procesorom na raspolaganju, izračunavanje vrednosti $2x + 3$ može se ostvariti na sledeći način. Pretpostavimo da se ulazni podatak (broj x) nalazi u glavnoj memoriji i to na adresi `10`, a da rezultat y treba smestiti na adresu `11` (ovo su sasvim proizvoljno odabrane adrese). Izračunavanje se onda može opisati sledećim programom (nizom instrukcija).

```
mov ax, [10]
mov bx, 2
mul ax, bx
mov bx, 3
add ax, bx
mov [11], ax
```

Instrukcija `mov ax, [10]` prepisuje vrednost promenljive x (iz memorije sa adrese `10`) u registar `ax`. Instrukcija `mov bx, 2` upisuje vrednost `2` u registar `bx`. Nakon instrukcije `mul ax, bx` vrši se množenje i registar `ax` sadrži vrednost $2x$. Instrukcija `mov bx, 3` upisuje vrednost `3` u registar `bx`, nakon instrukcije `add ax, bx` se vrši sabiranje i u registru `ax` se nalazi tražena vrednost $2x + 3$. Na kraju se ta vrednost instrukcijom `mov [11], ax` upisuje u memoriju na adresu `11`.

Određivanje većeg od dva broja može se ostvariti na sledeći način. Pretpostavimo da se ulazni podaci nalaze u glavnoj memoriji i to broj x na adresi `10`, broj y na adresi `11`, dok rezultat m treba smestiti na adresu `12`. Program (niz instrukcija) kojima može da se odredi maksimum je sledeći:

```
mov ax, [10]
mov bx, [11]
cmp ax, bx
jge vecix
mov [12], bx
jmp kraj
vecix:
mov[12], ax
kraj:
```

Nakon prenosa vrednosti oba broja u registre procesora (instrukcijama `mov ax, [10]` i `mov bx, [11]`), vrši se njihovo poređenje (instrukcija `cmp ax, bx`). Ukoliko je broj x veći od ili jednak broju y prelazi se na mesto označeno labelom `vecix` (instrukcijom `jge vecix`) i na mesto rezultata upisuje se vrednost promenljive x (instrukcijom `mov[12], ax`). Ukoliko uslov skoka `jge` nije ispunjen (ako x nije veće ili jednako y), na mesto rezultata upisuje se vrednost promenljive y (instrukcijom `mov [12], bx`) i bezuslovno se skače na kraj programa (instrukcijom `jmp kraj`) (kako bi se preskočilo izvršavanje instrukcije koja na mesto rezultata upisuje vrednost promenljive x).

Izračunavanje stepena može se ostvariti na sledeći način. Pretpostavimo da se ulazni podaci nalaze u glavnoj memoriji i to broj x na adresi `10`, a broj n na adresi `11`, i da konačan rezultat treba da bude smešten u memoriju i to na adresu `12`. Pretpostavimo da će pomoćne promenljive s i i koje se koriste u postupku biti smeštene sve vreme u procesoru, i to promenljiva s u registru `ax`, a promenljiva i u registru `bx`. Pošto postoji još samo jedan registar (`cx`), u njega će naizmenično biti smeštane vrednosti promenljivih n i x, kao i konstanta 1 koja se sabira sa promenljivom i. Niz instrukcija kojim opisani hipotetički računar može da izračuna stepen je sledeći:

```
  mov ax, 1
  mov bx, 0
petlja:
  mov cx, [11]
  cmp bx, cx
  jge kraj
  mov cx, [10]
  mul ax, cx
  mov cx, 1
  add bx, cx
```

```
  jmp petlja
kraj:
  mov [12], ax
```

Ilustrujmo izvršavanje ovog programa na izračunavanju vrednosti 3^2. Inicijalna konfiguracija je takva da se na adresi `10` u memoriji nalazi vrednost $x = 3$, na adresi `11` vrednost $n = 2$. Početna konfiguracija (tj. vrednosti memorijskih lokacija i registara) može da se predstavi na sledeći način:

```
10: 3    ax: ?
11: 2    bx: ?
12: ?    cx: ?
```

Nakon izvršavanja prve dve instrukcije (`mov ax, 1` i `mov bx, 0`), postavlja se vrednost registara `ax` i `bx` i prelazi se u sledeću konfiguraciju:

```
10: 3    ax: 1
11: 2    bx: 0
12: ?    cx: ?
```

Sledeća instrukcija (`mov cx, [11]`) kopira vrednost `2` sa adrese `11` u registar `cx`:

```
10: 3    ax: 1
11: 2    bx: 0
12: ?    cx: 2
```

Vrši se poređenje sa registrom `bx` (`cmp bx, cx`) i kako uslov skoka (`jge kraj`) nije ispunjen (vrednost `0` u `bx` nije veća ili jednaka od vrednosti `2` u `cx`), nastavlja se dalje. Nakon kopiranja vrednosti `3` sa adrese `10` u registar `cx` (instrukcijom `mov cx, [10]`), vrši se množenje vrednosti u registrima `ax` i `cx` (instrukcijom `mul ax, cx`) i dolazi se u sledeću konfiguraciju:

```
10: 3    ax: 3
11: 2    bx: 0
12: ?    cx: 3
```

Nakon toga, u `cx` se upisuje `1` (instrukcijom `mov cx, 1`) i vrši se sabiranje vrednosti registara `bx` i `cx` (instrukcijom `add bx, cx`) čime se vrednost u registru `bx` uvećava za `1`.

```
10: 3    ax: 3
11: 2    bx: 1
12: ?    cx: 1
```

Bezuslovni skok (`jmp petlja`) ponovo vraća kontrolu na početak petlje, nakon čega se u `cx` opet prepisuje vrednost `2` sa adrese `11` (`mov cx, [11]`). Vrši se poređenje sa registrom `bx` (`cmp bx, cx`) i kako uslov skoka (`jge kraj`) nije ispunjen (vrednost `1` u `bx` nije veća ili jednaka vrednosti `2` u `cx`), nastavlja se dalje. Nakon još jednog množenja i sabiranja dolazi se do konfiguracije:

```
10: 3    ax: 9
11: 2    bx: 2
12: ?    cx: 1
```

Bezuslovni skok ponovo vraća kontrolu na početak petlje, nakon čega se u `cx` opet prepisuje vrednost `2` sa adrese `11`. Vrši se poređenje sa registrom `bx`, no, ovaj put je uslov skoka ispunjen (vrednost `2` u `bx` je veća ili jednaka vrednosti `2` u `cx`) i skače se na mesto označeno labelom `kraj`, gde se poslednjom instrukcijom (`mov [12], ax`) konačna vrednost iz registra `ax` kopira u memoriju na dogovorenu adresu `12`, čime se stiže u završnu konfiguraciju:

```
10: 3     ax: 9
11: 2     bx: 2
12: 9     cx: 1
```

Mašinski jezik. Fon Nojmanova arhitektura podrazumeva da se i sam program (niz instrukcija) nalazi u glavnoj memoriji prilikom njegovog izvršavanja. Potrebno je svaki program (poput tri navedena) predstaviti nizom nula i jedinica, na način „razumljiv“ procesoru — na mašinskom jeziku. Na primer, moguće je da su binarni kôdovi za instrukcije uvedeni na sledeći način:

`mov`	`001`
`add`	`010`
`mul`	`011`
`cmp`	`100`
`jge`	`101`
`jmp`	`110`

Takođe, pošto neke instrukcije primaju podatke različite vrste (neposredno navedeni brojevi, registri, apsolutne memorijske adrese), uvedeni su posebni kôdovi za svaki od različitih vidova adresiranja. Na primer:

neposredno	`00`
registarsko	`01`
apsolutno	`10`

Pretpostavimo da registar `ax` ima oznaku `00`, registar `bx` ima oznaku `01`, a registar `cx` oznaku `10`. Pretpostavimo i da su sve adrese osmobitne. Pod navedenim pretpostavkama, instrukcija `mov [10], ax` se, u ovom hipotetičkom mašinskom jeziku, može kodirati kao `001 10 01 00010000 00`. Kôd `001` dolazi od instrukcije `mov`, zatim slede `10` i `01` koji ukazuju da prvi argument predstavlja memorijsku adresu, a drugi oznaku registra, za čim sledi memorijska adresa $(10)_{16}$ binarno kodirana sa `00010000` i na kraju oznaka `00` registra `ax`. Na sličan način, celokupan prikazani mašinski kôd navedenog asemblerskog programa koji izračunava $2x + 3$ je moguće binarno kodirati kao:

```
001 01 10 00 00010000   // mov ax, [10]
001 01 00 01 00000010   // mov bx, 2
011 00 01               // mul ax, bx
001 01 00 01 00000011   // mov bx, 3
010 00 01               // add ax, bx
001 10 01 00010001 00   // mov [11], ax
```

Između prikazanog asemblerskog i mašinskog programa postoji veoma direktna i jednoznačna korespondencija (u oba smera) tj. na osnovu datog mašinskog kôda moguće je jednoznačno rekonstruisati asemblerski kôd.

Specifični hardver koji čini kontrolnu jedinicu procesora dekodira jednu po jednu instrukciju i izvršava akciju zadatu tom instrukcijom. Kod realnih procesora, broj instrukcija i načini adresiranja su mnogo bogatiji a prilikom pisanja

programa potrebno je uzeti u obzir mnoge aspekte na koje se u navedenim jednostavnim primerima nije obraćala pažnja. Ipak, mašinske i asemblerske instrukcije stvarnih procesora veoma su slične navedenim hipotetičkim instrukcijama.

1.5.3 Klasifikacija savremenog softvera

Računarski programi veoma su složeni. Hardver računara sačinjen je od elektronskih kola koja mogu da izvrše samo elementarne operacije i, da bi računar mogao da obavi i najjednostavniji zadatak zanimljiv korisniku, neophodno je da se taj zadatak razloži na mnoštvo elementarnih operacija. Napredak računara ne bi bio moguć ako bi programeri morali svaki program da opisuju i razlažu do krajnjeg nivoa elementarnih instrukcija. Zato je poželjno da programeri naredbe računaru mogu zadavati na što apstraktnijem nivou. Računarski sistemi i softver se grade slojevito i svaki naredni sloj oslanja se na funkcionalnost koju mu nudi sloj ispod njega. U skladu sa tim, softver savremenih računara se obično deli na *aplikativni* i *sistemski*. Osnovni zadatak sistemskog softvera je da posreduje između hardvera i aplikativnog softvera koji krajnji korisnici koriste. Granica između sistemskog i aplikativnog softvera nije kruta i postoje programi za koje se može smatrati da pripadaju obema grupama (na primer, editori teksta).

Aplikativni softver je softver koji krajnji korisnici računara direktno koriste u svojim svakodnevnim aktivnostima. To su pre svega pregledači Veba, zatim klijenti elektronske pošte, kancelarijski softver (programi za kucanje teksta, izradu slajd-prezentacija, tabelarna izračunavanja), video igre, multimedijalni softver (programi za reprodukciju i obradu slika, zvuka i video-sadržaja) itd.

Sistemski softver je softver čija je uloga da kontroliše hardver i pruža usluge aplikativnom softveru. Najznačajniji skup sistemskog softvera, danas prisutan na skoro svim računarima, čini *operativni sistem* (OS). Pored OS, sistemski softver sačinjavaju i različiti *uslužni programi*: editori teksta, alat za programiranje (prevodioci, dibageri, profajleri, integrisana okruženja) i slično.

Korisnici OS često identifikuju sa izgledom ekrana tj. sa programom koji koriste da bi pokrenuli svoje aplikacije i organizovali dokumente. Međutim, ovaj deo sistema koji se naziva *korisnički interfejs (engl. user interface – UI)* ili *školjka (engl. shell)* samo je tanak sloj na vrhu operativnog sistema i OS je mnogo više od onoga što krajnji korisnici vide. Najveći i najznačajni deo OS naziva se *jezgro (engl. kernel)*. Osim što kontroliše i apstrahuje hardver, operativni sistem tj. njegovo jezgro sinhronizuje rad više programa, raspoređuje procesorsko vreme i memoriju, brine o sistemu datoteka na spoljašnjim memorijama itd. Najznačanjni operativni sistemi danas su Microsoft Windows, sistemi zasnovani na Linux jezgru (na primer, Ubuntu, RedHat, Fedora, Suse) i Mac OS X.

OS upravlja svim resursima računara (procesorom, memorijom, periferijskim uređajima) i stavlja ih na raspolaganje aplikativnim programima. OS je u veoma tesnoj vezi sa hardverom računara i veliki deo zadataka se izrvršava uz direktnu podršku specijalizovanog hardvera namenjenog

isključivo izvršavanju OS. Nekada se hardver i operativni sistem smatraju jedinstvenom celinom i umesto podele na hardver i softver razmatra se podela na sistem (hardver i OS) i na aplikativni softver.

Programer ne bi trebalo da misli o konkretnim detaljima hardvera, tj. poželjno je da postoji određena *apstrakcija hardvera*. Na primer, mnogo je pogodnije ako programer umesto da mora da kaže „Neka se zavrti ploča diska, neka se glava pozicionira na određenu poziciju, neka se zatim tu upiše određeni bajt itd.“ može da kaže „Neka se u datu datoteku na disku upiše određeni tekst“. OS je taj koji se brine o svim detaljima, dok se programer (tačnije, aplikacije koje on isprogramira), kada god mu je potrebno obraća sistemu da mu tu uslugu pruži. Konkretni detalji hardvera poznati su u okviru operativnog sistema i komande koje programer zadaje izvršavaju se uzimajući u obzir ove specifičnosti. Operativni sistem, dakle, programeru pruža skup funkcija koje on može da koristi kako bi postigao željenu funkcionalnost hardvera, sakrivajući pritom konkretne hardverske detalje. Ovaj skup funkcija naziva se *programski interfejs za pisanje aplikacija*[24] (engl. *Aplication Programming Interface, API*). Funkcije se nazivaju i *sistemski pozivi* (jer se OS poziva da izvrši određeni zadatak). Programer nema mogućnost direktnog pristupa hardveru i jedini način da se pristupi hardveru je preko sistemskih poziva. Ovim se osigurava određena bezbednost celog sistema.

Postoji više nivoa na kojima se može realizovati neka funkcionalnost. Programer aplikacije je na vrhu hijerarhije i on može da koristi funkcionalnost koju mu pruža programski jezik koji koristi i *biblioteke* tog jezika. Izvršni programi često koriste funkcionalnost specijalne *rantajm biblioteke (engl. runtime library)* koja koristi funkcionalnost operativnog sistema (preko sistemskih poziva), a zatim operativni sistem koristi funkcionalnost samog hardvera.

Pitanja za vežbu

Pitanje 1.1. *Nabrojati osnovne periode u razvoju računara i navesti njihove osnovne karakteristike i predstavnike.*

Pitanje 1.2. *Ko je i u kom veku konstruisao prvu mehaničku spravu na kojoj je bilo moguće sabirati prirodne brojeve, a ko je i u kom veku konstruisao prvu mehaničku spravu na kojoj je bilo moguće sabirati i množiti prirodne brojeve?*

Pitanje 1.3. *Kojoj spravi koja se koristi u današnjem svetu najviše odgovaraju Paskalove i Lajbnicove sprave?*

Pitanje 1.4. *Kakva je veza između tkačkih razboja i računara s početka XIX veka?*

Pitanje 1.5. *Koji je značaj Čarlsa Bebidža za razvoj računarstva i programiranja? U kom veku je on dizajnirao svoje računske mašine? Kako se one zovu i koja od njih je trebalo da bude programabilna? Ko se smatra prvim programerom?*

[24] Ovaj termin se ne koristi samo u okviru operativnih sistema, već i u širem kontekstu, da označi skup funkcija kroz koji jedan programski sistem koristi drugi programski sistem.

Pitanje 1.6. *Na koji način je Herman Holerit doprineo izvršavanju popisa stanovnika u SAD 1890? Kako su bili čuvani podaci sa tog popisa? Koja čuvena kompanija je nastala iz kompanije koju je Holerit osnovao?*

Pitanje 1.7. *Kada su nastali prvi elektronski računari? Nabrojati nekoliko najznačajnijih.*

Pitanje 1.8. *Na koji način je programiran računar ENIAC, a na koji računar EDVAC?*

Pitanje 1.9. *Koje su osnovne komponente računara fon Nojmanove arhitekture? Šta se skladišti u memoriju računara fon Nojmanove arhitekture? Gde se vrši obrada podataka u okviru računara fon Nojmanove arhitekture? Od kada su računari zasnovani na fon Nojmanovoj arhitekturi?*

Pitanje 1.10. *Šta su to računari sa skladištenim programom? Šta je to hardver a šta softver?*

Pitanje 1.11. *Šta su procesorske instrukcije? Navesti nekoliko primera.*

Pitanje 1.12. *Koji su uobičajeni delovi procesora? Da li se u okviru samog procesora nalazi određena količina memorije za smeštanje podataka? Kako se ona naziva?*

Pitanje 1.13. *Ukratko opisati osnovne elektronske komponente svake generacije računara savremenih elektronskih računara? Šta su bile osnovne elektronske komponente prve generacije elektronskih računara? Od koje generacije računara se koriste mikroprocesori? Koji tipovi računara se koriste u okviru III generacije?*

Pitanje 1.14. *U kojoj deceniji dolazi do pojave računara za kućnu upotrebu? Koji je najprodavaniji model kompanije Commodore? Da li je IBM proizvodio računare za kućnu upotrebu? Koji komercijalni kućni računar prvi uvodi grafički korisnički interfejs i miša?*

Pitanje 1.15. *Koja serija Intelovih procesora je bila dominantna u PC računarima 1980-ih i 1990-ih godina?*

Pitanje 1.16. *Šta je to tehnološka konvergencija? Šta su to tableti, a šta „pametni telefoni“?*

Pitanje 1.17. *Koje su osnovne komponente savremenog računara? Šta je memorijska hijerarhija? Zašto se uvodi keš-memorija? Koje su danas najkorišćenije spoljne memorije?*

Pitanje 1.18. *U koju grupu jezika spadaju mašinski jezici i asemblerski jezici?*

Pitanje 1.19. *Da li je kôd na nekom mašinskom jeziku prenosiv sa jednog na sve druge računare? Da li asembler zavisi od mašine na kojoj se koristi?*

Pitanje 1.20. *Ukoliko je raspoloživ asemblerski kôd nekog programa, da li je moguće jednoznačno konstruisati odgovarajući mašinski kôd? Ukoliko je raspoloživ mašinski kôd nekog programa, da li je moguće jednoznačno konstruisati odgovarajući asemblerski kôd?*

Zadatak 1.1. *Na opisanom asemblerskom jeziku opisati izračunavanje vrednosti izraza* `x := x*y + y + 3`. *Generisati i mašinski kôd za napisani program.*

Zadatak 1.2. *Na opisanom asemblerskom jeziku opisati izračunavanje:*

```
ako je (x < 0)
   y := 3*x;
inace
   x := 3*y;
```

Zadatak 1.3. *Na opisanom asemblerskom jeziku opisati izračunavanje:*

```
dok je (x <= 0) radi
   x := x + 2*y + 3;
```

Zadatak 1.4. *Na opisanom asemblerskom jeziku opisati izračunavanje kojim se izračunava* $[\sqrt{x}]$, *pri čemu se* x *nalazi na adresi 100, a rezultat smešta na adresu 200.* √

Pitanje 1.21. *Koji su osnovni razlozi slojevite organizacije softvera? Šta je sistemski, a šta aplikativni softver?*

Pitanje 1.22. *Koji su osnovni zadaci operativnog sistema? Šta su sistemski pozivi? Koji su operativni sistemi danas najkorišćeniji?*

Glava 2

Reprezentacija podataka u računarima

Današnji računari su *digitalni*. To znači da su svi podaci koji su u njima zapisani, zapisani kao nizovi celih brojeva. Dekadni brojevni sistem koji ljudi koriste u svakodnevnom životu nije pogodan za zapis brojeva u računarima jer zahteva azbuku od 10 različitih simbola (cifara). Bilo da se radi o elektronskim, magnetnim ili optičkim komponentama, tehnologija izrade računara i medijuma za zapis podataka koristi elemente koji imaju dva diskretna stanja, što za zapis podataka daje azbuku od samo dva različita simbola. Tako, na primer, ukoliko između dve tačke postoji napon viši od određenog praga, onda se smatra da tom paru tačaka odgovara vrednost 1, a inače mu odgovara vrednost 0. Takođe, polje hard diska može biti ili namagnetisano što odgovara vrednosti 1 ili razmagnetisano što odgovara vrednosti 0. Slično, laserski zrak na površini kompakt diska „buši rupice“ kojim je određen zapis podataka pa polje koje nije izbušeno predstavlja vrednost 0, a ono koje jeste izbušesno predstavlja vrednost 1. U nastavku će biti pokazano da je azbuka od samo dva simbola dovoljna za zapisivanje svih vrsta brojeva, pa samim tim i za zapisivanje svih vrsta digitalnih podataka.

2.1 Analogni i digitalni podaci i digitalni računari

Kontinualna priroda signala. Većina podataka koje računari koriste nastaje zapisivanjem prirodnih signala. Najznačajniji primeri signala su zvuk i slika, ali se pod signalima podrazumevaju i ultrazvučni signali, EKG signali, zračenja različite vrste itd.

Signali koji nas okružuju u prirodi u većini slučajeva se prirodno mogu predstaviti neprekidnim funkcijama. Na primer, zvučni signal predstavlja promenu pritiska vazduha u zadatoj tački i to kao neprekidnu funkciju vremena. Slika se može opisati intenzitetom svetlosti određene boje (tj. određene talasne dužine) u datom vremenskom trenutku i to kao neprekidna funkcija prostora.

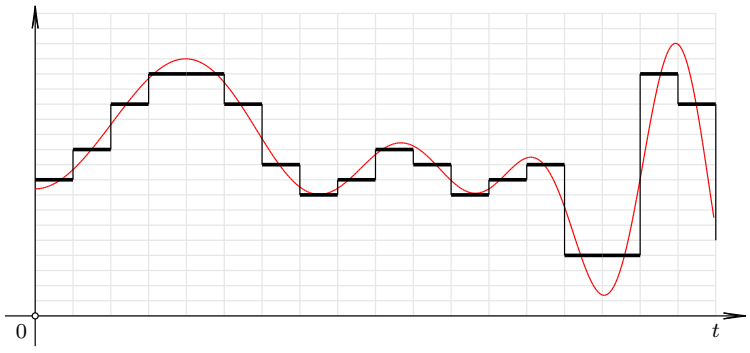
Analogni zapis. Osnovna tehnika koja se primenjuje kod analognog zapisa signala je da se kontinualne promene signala koji se zapisuje opišu kontinualnim promenama određenog svojstva medijuma na kojem se signal zapisuje. Tako,

na primer, promene pritiska vazduha koji predstavlja zvučni signal direktno odgovaraju promenama nivoa namagnetisanja na magnetnoj traci na kojoj se zvuk analogno zapisuje. Količina boje na papiru direktno odgovara intenzitetu svetlosti u vremenskom trenutku kada je fotografija bila snimljena. Dakle, analogni zapis uspostavlja *analogiju* između signala koji je zapisan i određenog svojstva medijuma na kome je signal zapisan.

Osnovna prednost analogne tehnologije je da je ona obično veoma jednostavna ukoliko se zadovoljimo relativno niskim kvalitetom (još su drevni narodi mogli da naprave nekakav zapis zvuka uz pomoć jednostavne igle prikačene na trepereću membranu).

Osnovni problem analogne tehnologije je što je izrazito teško na medijumu napraviti veran zapis signala koji se zapisuje i izrazito je teško napraviti dva identična zapisa istog signala. Takođe, problem predstavlja i inherentna nestalnost medijuma, njegova promenljivost tokom vremena i podložnost spoljašnjim uticajima. S obzirom na to da varijacije medijuma direktno dovode do varijacije zapisanog signala, vremenom neizbežno dolazi do pada kvaliteta analogno zapisanog signala. Obrada analogno zapisanih signala je obično veoma komplikovana i za svaku vrstu obrade signala, potrebno je da postoji uređaj koji je specijalizovan za tu vrste obrade.

Digitalni zapis. Osnovna tehnika koja se koristi kod digitalnog zapisa podataka je da se vrednost signala izmeri u određenim vremenskim trenucima ili određenim tačkama prostora i da se onda na medijumu zapišu izmerene vrednosti. Ovim je svaki digitalno zapisani signal predstavljen nizom brojeva koji se nazivaju *odbirci (engl. sample)*. Svaki od brojeva predstavlja vrednost signala u jednoj tački diskretizovanog domena. S obzirom na to da izmerene vrednosti takođe pripadaju kontinualnoj skali, neophodno je izvršiti i diskretizaciju kodomena, odnosno dopustiti zapisivanje samo određenog broja nivoa različitih vrednosti.



Slika 2.1: Digitalizacija zvučnog signala

Digitalni zapis predstavlja diskretnu aproksimaciju polaznog signala. Važno pitanje je koliko često je potrebno vršiti merenje, kako bi se polazni kontinualni signal mogao verno rekonstruisati. Odgovor daje tvrđenje o odabiranju (tzv. Najkvist-Šenonova teorema), koje kaže da je signal dovoljno meriti dva

puta češće od najviše frekvencije koja sa u njemu javlja. Na primer, pošto čovekovo uho čuje frekvencije do 20KHz, dovoljno je da frekvencija odabiranja (sempliranja) bude 40KHz. Dok je za analogne tehnologije za postizanje visokog kvaliteta zapisa potrebno imati medijume visokog kvaliteta, kvalitet reprodukcije digitalnog zapisa ne zavisi od toga kakav je kvalitet medija na kome su podaci zapisani, sve dok je medijum dovoljnog kvaliteta da se zapisani brojevi mogu razaznati. Dodatno, kvarljivost koja je inherentna za sve medije postaje nebitna. Na primer, papir vremenom žuti što uzrokuje pad kvaliteta analognih fotografija tokom vremena. Međutim, ukoliko bi papir sadržao zapis brojeva koji predstavljaju vrednosti boja u tačkama digitalno zapisane fotografije, činjenica da papir žuti ne bi predstavljala problem dok god se brojevi mogu razaznati.

Digitalni zapis omogućava kreiranje apsolutno identičnih kopija što dalje omogućava prenos podataka na daljinu. Na primer, ukoliko izvršimo fotokopiranje fotografije, napravljena fotokopija je daleko lošijeg kvaliteta od originala. Međutim, ukoliko umnožimo CD na kojem su zapisani brojevi koji čine zapis neke fotografije, kvalitet slike ostaje apsolutno isti. Ukoliko bi se dva CD-a pregledala pod mikroskopom, oni bi izgledali delimično različito, ali to ne predstavlja problem sve dok se brojevi koji su na njima zapisani mogu razaznati.

Obrada digitalno zapisanih podataka se svodi na matematičku manipulaciju brojevima i ne zahteva (za razliku od analognih podataka) korišćenje specijalizovanih mašina.

Osnovni problem implementacije digitalnog zapisa predstavlja činjenica da je neophodno imati veoma razvijenu tehnologiju da bi se uopšte stiglo do iole upotrebljivog zapisa. Na primer, izuzetno je komplikovano napraviti uređaj koji je u stanju da 40 hiljada puta izvrši merenje intenziteta zvuka. Jedna sekunda zvuka se predstavlja sa 40 hiljada brojeva, za čiji je zapis neophodna gotovo cela jedna sveska. Ovo je osnovni razlog zbog čega se digitalni zapis istorijski javio kasno. Kada se došlo do tehnološkog nivoa koji omogućava digitalni zapis, on je doneo mnoge prednosti u odnosu na analogni.

2.2 Zapis brojeva

Proces digitalizacije je proces reprezentovanja (raznovrsnih) podataka brojevima. Kako se svi podaci u računarima reprezentuju na taj način — brojevima, neophodno je precizno definisati zapisivanje različitih vrsta brojeva. Osnovu digitalnih računara, u skladu sa njihovom tehnološkom osnovom, predstavlja *binarni* brojevni sistem (sistem sa osnovom 2). U računarstvu se koriste i *heksadekadni* brojevni sistem (sistem sa osnovom 16) a i, nešto ređe, *oktalni* brojevni sistem (sistem sa osnovom 8), zbog toga što ovi sistemi omogućavaju jednostavnu konverziju između njih i binarnog sistema. Svi ovi brojevni sistemi, kao i drugi o kojima će biti reči u nastavku teksta, su *pozicioni*. U pozicionom brojnom sistemu, udeo cifre u celokupnoj vrednosti zapisanog broja zavisi od njene pozicije.

Treba naglasiti da je zapis broja samo konvencija a da su brojevi koji se zapisuju apsolutni i ne zavise od konkretnog zapisa. Tako, na primer, zbir dva prirodna broja je uvek jedan isti prirodni broj, bez obzira na to u kom sistemu su ova tri broja zapisana.

S obzirom na to da je svaki zapis broja u računaru ograničen, ne mogu biti zapisani svi celi brojevi. Ipak, za cele brojeve zapisive u računaru se obično govori samo *celi brojevi*, dok su ispravnija imena *označeni celi brojevi* (engl. signed integers) i *neoznačeni celi brojevi* (engl. unsigned integers), koji podrazumevaju konačan zapis. Ni svi realni brojevi (sa potencijalno beskonačnim decimalnim zapisom) ne mogu biti zapisani u računaru. Za zapis zapisivih realnih brojeva (koji su uvek racionalni) se koristi konvencija zapisa u pokretnom zarezu. Iako je jedino precizno ime za ove brojeve *brojevi u pokretnom zarezu* (engl. floating point numbers), često se koriste i imena *realni* ili *racionalni brojevi*. Zbog ograničenog zapisa brojeva, rezultati matematičkih operacija nad njima sprovedenih u računaru, neće uvek odgovarati rezultatima koji bi se dobili bez tih ograničenja (zbog takozvanih *prekoračenja*). Naglasimo još i da, za razliku od matematike gde se skup celih brojeva smatra podskupom skupa realnih brojeva, u računarstvu, zbog različitog načina zapisa, između ovih vrsta brojeva ne važi nikakva direktna veza.

2.2.1 Neoznačeni brojevi

Pod *neoznačenim brojevima* podrazumeva se neoznačeni zapis nenegativnih celih brojeva i znak se izostavlja iz zapisa.

Određivanje broja na osnovu datog zapisa. Pretpostavimo da je dat pozicioni brojevni sistem sa osnovom b, gde je b prirodan broj veći od 1. Niz cifara $(a_n\, a_{n-1}\, \ldots\, a_1\, a_0)_b$ predstavlja zapis[1] broja u osnovi b, pri čemu za svaku cifru a_i važi $0 \leq a_i < b$.

Vrednost broja zapisanog u osnovi b definiše se na sledeći način:

$$(a_n\, a_{n-1}\, \ldots\, a_1\, a_0)_b = \sum_{i=0}^{n} a_i \cdot b^i = a_n \cdot b^n + a_{n-1} \cdot b^{n-1} + \ldots a_1 \cdot b + a_0$$

Na primer:

$(101101)_2 = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2 + 1 = 32 + 8 + 4 + 1 = 45$,

$(3245)_8 = 3{\cdot}8^3 + 2{\cdot}8^2 + 4{\cdot}8 + 5 = 3{\cdot}512 + 2{\cdot}64 + 4{\cdot}8 + 5 = 1536 + 128 + 32 + 5 = 1701$.

Navedena definicija daje i postupak za određivanje vrednosti datog zapisa:

x `:=` 0
`za svako` i `od` 0 `do` n
    x `:=` $x + a_i \cdot b^i$

ili, malo modifikovano:

x `:=` a_0
`za svako` i `od` 1 `do` n
    x `:=` $x + a_i \cdot b^i$

[1] Ako u zapisu broja nije navedena osnova, podrazumeva se da je osnova 10.

Za izračunavanje vrednosti nekog $(n+1)$-tocifrenog zapisa drugim navedenim postupkom potrebno je n sabiranja i $n + (n-1) + \ldots + 1 = \frac{n(n+1)}{2}$ množenja. Zaista, da bi se izračunalo $a_n \cdot b^n$ potrebno je n množenja, da bi se izračunalo $a_{n-1} \cdot b^{n-1}$ potrebno je $n-1$ množenja, itd. Međutim, ovo izračunavanje može da se izvrši i efikasnije. Ukoliko se za izračunavanje člana b^i iskoristi već izračunata vrednost b^{i-1}, broj množenja se može svesti na $2n$. Ovaj način izračunavanja primenjen je u sledećem postupku:

x := a_0
B := 1
za svako i od 1 do n
    B := $B \cdot b$
    x := $x + a_i \cdot B$

Još efikasniji postupak izračunavanja se može dobiti korišćenjem *Hornerove sheme*:

$$(a_n\, a_{n-1}\, \ldots\, a_1\, a_0)_b = (\ldots((a_n \cdot b + a_{n-1}) \cdot b + a_{n-2}) \ldots + a_1) \cdot b + a_0$$

Korišćenjem ove sheme, dolazi se do sledećeg postupka za određivanje vrednosti broja zapisanog u nekoj brojevnoj osnovi:

x := 0
za svako i od n unazad do 0
    x := $x \cdot b + a_i$

Naredni primer ilustruje primenu Hornerovog postupka na zapis $(9876)_{10}$.

i		3	2	1	0
a_i		9	8	7	6
x	0	$0 \cdot 10 + 9 = 9$	$9 \cdot 10 + 8 = 98$	$98 \cdot 10 + 7 = 987$	$987 \cdot 10 + 6 = 9876$

Međurezultati dobijeni u ovom računu direktno odgovaraju prefiksima zapisa čija se vrednost određuje, a rezultat u poslednjoj koloni je traženi broj.

Navedeni postupak može se primeniti na proizvoljnu brojevnu osnovu. Sledeći primer ilustruje primenu postupka na zapis $(3245)_8$.

i		3	2	1	0
a_i		3	2	4	5
x	0	$0 \cdot 8 + 3 = 3$	$3 \cdot 8 + 2 = 26$	$26 \cdot 8 + 4 = 212$	$212 \cdot 8 + 5 = 1701$

Navedena tabela može se kraće zapisati na sledeći način:

	3	2	4	5
0	3	26	212	1701

Hornerov postupak je efikasniji u odnosu na početni postupak, jer je u svakom koraku dovoljno izvršiti samo jedno množenje i jedno sabiranje (ukupno $n+1$ sabiranja i $n+1$ množenja).

Određivanje zapisa datog broja. Za svaku cifru a_i u zapisu broja x u osnovi b važi da je $0 \leq a_i < b$. Dodatno, pri deljenju broja x osnovom b, ostatak je a_0 a celobrojni količnik je broj čiji je zapis $(a_n\, a_{n-1} \ldots a_1)_b$. Dakle, izračunavanjem celobrojnog količnika i ostatka pri deljenju sa b, određena je poslednja cifra broja x i broj koji se dobija uklanjanjem poslednje cifre iz zapisa. Ukoliko se isti postupak primeni na dobijeni količnik, dobija se postupak koji omogućava da se odrede sve cifre u zapisu broja x. Postupak se zaustavlja kada tekući količnik postane 0. Ako se izračunavanje ostatka pri deljenju označi sa `mod`, a celobrojnog količnika sa `div`, postupak kojim se određuje zapis broja x u datoj osnovi b se može formulisati na sledeći način:

i `:=` 0
`dok je` x `različito od` 0
    a_i `:=` x `mod` b
    x `:=` x `div` b
    i `:=` $i+1$

Na primer, $1701 = (3245)_8$ jer je $1701 = 212 \cdot 8 + 5 = (26 \cdot 8 + 4) \cdot 8 + 5 = ((3 \cdot 8 + 2) \cdot 8 + 4) \cdot 8 + 5 = (((0 \cdot 8 + 3) \cdot 8 + 2) \cdot 8 + 4) \cdot 8 + 5$. Ovaj postupak se može prikazati i tabelom:

i	0	1	2	3	
x	1701	1701 div 8 = 212	212 div 8 = 26	26 div 8 = 3	3 div 8 = 0
a_i	1701	1701 mod 8 = 5	212 mod 8 = 4	26 mod 8 = 2	3 mod 8 = 3

Prethodna tabela može se kraće zapisati na sledeći način:

1701	212	26	3	0
5	4	2	3	

Druga vrsta tabele sadrži celobrojne količnike, a treća ostatke pri deljenju sa osnovom b, tj. tražene cifre. Zapis broja se formira tako što se dobijene cifre čitaju unatrag.

Ovaj algoritam i Hornerov algoritam su međusobno simetrični u smislu da se svi međurezultati poklapaju.

Direktno prevođenje između heksadekadnog i binarnog sistema. Osnovni razlog korišćenja heksadekadnog sistema je mogućnost jednostavnog prevođenja brojeva između binarnog i heksadekadnog sistema. Pri tome, heksadekadni sistem omogućava da se binarni sadržaj memorije zapiše kompaktnije (uz korišćenje manjeg broja cifara). Prevođenje se može izvršiti tako što se grupišu četiri po četiri binarne cifre, krenuvši unazad, i svaka četvorka se zasebno prevede u odgovarajuću heksadekadnu cifru na osnovu sledeće tabele:

heksa	binarno	heksa	binarno	heksa	binarno	heksa	binarno
0	0000	4	0100	8	1000	C	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

Na primer, proizvoljni 32-bitni sadržaj može se zapisati korišćenjem osam heksadekadnih cifara:
$(1011\,0110\,0111\,1100\,0010\,1001\,1111\,0001)_2 = (B67C29F1)_{16}$

Zapisi fiksirane dužine U računarima se obično koristi fiksirani broj binarnih cifara (sačuvanih u pojedinačnim *bitovima*) za zapis svakog broja. Takve zapise označavamo sa $(\ldots)_b^n$, ako se koristi n cifara. Ukoliko je broj cifara potrebnih za zapis broja kraći od zadate dužine zapisa, onda se broj proširuje vodećim nulama. Na primer, $55 = (0011\,0111)_2^8$. Ograničavanjem broja cifara ograničava se i raspon brojeva koje je moguće zapisati (u binarnom sistemu) i to na raspon od 0 do $2^n - 1$. U sledećoj tabeli su dati rasponi za najčešće korišćene dužine zapisa:

broj bitova	raspon
8	od 0 do 255
16	od 0 do 65535
32	od 0 do 4294967295

2.2.2 Označeni brojevi

Označeni brojevi su celi brojevi čiji zapis uključuje i zapis znaka broja (+ ili −). S obzirom na to da savremeni računari koriste binarni brojevni sistem, biće razmatrani samo zapisi označenih brojeva u binarnom brojevnom sistemu. Postoji više načina zapisivanja označenih brojeva od kojih su najčešće u upotrebi *označena apsolutna vrednost* i *potpuni komplement*.

Označena apsolutna vrednost. Zapis broja se formira tako što se na prvu poziciju zapisa unapred fiksirane dužine n, upiše znak broja, a na preostalih $n - 1$ pozicija upiše zapis apsolutne vrednosti broja. Pošto se za zapis koriste samo dva simbola (0 i 1), konvencija je da se znak + zapisuje simbolom 0, a znak − simbolom 1. Ovim se postiže da pozitivni brojevi imaju identičan zapis kao da su u pitanju neoznačeni brojevi. Na primer, $+100 = (0\,1100100)_2^8$, $-100 = (1\,1100100)_2^8$.

Osnovni problem zapisa u obliku označene apsolutne vrednosti je činjenica da se osnovne aritmetičke operacije teško izvode ukoliko su brojevi zapisani na ovaj način.

Potpuni komplement. Zapis u potpunom komplementu (engl. two's complement) označenih brojeva zadovoljava sledeće uslove:

1. Nula i pozitivni brojevi se zapisuju na isti način kao da su u pitanju neoznačeni brojevi, pri čemu u njihovom zapisu prva cifra mora da bude 0.

2. Sabiranje se sprovodi na isti način kao da su u pitanju neoznačeni brojevi, pri čemu se prenos sa poslednje pozicije zanemaruje.

Na primer, broj +100 se u potpunom komplementu zapisuje kao $(0\,1100100)_2^8$. Nula se zapisuje kao $(0\,0000000)_2^8$. Zapis broja −100 u obliku $(\ldots)_2^8$ se može odrediti na sledeći način. Zbir brojeva −100 i +100 mora da bude 0.

	binarno	dekadno
	????????	-100
+	01100100	+100
~~1~~	00000000	0

Analizom traženog sabiranja cifru po cifru, počevši od poslednje, sledi da se -100 mora zapisati kao $(10011100)_2^8$. Do ovoga je moguće doći i na sledeći način. Ukoliko je poznat zapis broja x, zapis njemu suprotnog broja je moguće odrediti iz uslova da je $x + (-x) = (1\ 00\ldots00)_2^{n+1}$. Pošto je $(1\ 00\ldots00)_2^{n+1} = (11\ldots11)_2^n + 1$, zapis broja $(-x)$ je moguće odrediti tako što se izračuna $(11\ldots11)_2^n - x + 1$. Izračunavanje razlike $(11\ldots11)_2^n - x$ se svodi na *komplementiranje* svake pojedinačne cifre broja x. Tako se određivanje zapisa broja -100 može opisati na sledeći način:

	01100100	+100
	10011011	komplementiranje
+	1	
	10011100	

Kao što je traženo, zapisi svih pozitivnih brojeva i nule počinju cifrom 0 dok zapisi negativnih brojeva počinju sa 1.

Broj -2^{n-1} je jedini izuzetak u opštem postupku određivanja zapisa u potpunom komplementu. Zapis broja $(100\ldots00)_2^n$ je sam sebi komplementaran, a pošto počinje cifrom 1, uzima se da on predstavlja zapis najmanjeg zapisivog negativnog broja, tj. broja -2^{n-1}. Zahvaljujući ovoj konvenciji, u zapisu potpunog komplementa $(\ldots)_2^n$ moguće je zapisati brojeve od -2^{n-1} do $2^{n-1}-1$. U sledećoj tabeli su dati rasponi za najčešće korišćene dužine zapise u potpunom komplementu:

broj bita	raspon
8	od -128 do $+127$
16	od -32768 do $+32767$
32	od -2147483648 do $+2147483647$

Kao što je rečeno, za sabiranje brojeva zapisanih u potpunom komplementu može se koristiti opšti postupak za sabiranje neoznačenih brojeva (pri čemu se podrazumeva da može doći do prekoračenja, tj. da neke cifre rezultata ne mogu biti upisane u raspoloživ broj mesta). To ne važi samo za sabiranje, već i za oduzimanje i množenje. Ovo pogodno svojstvo važi i kada je jedan od argumenata broj -2^{n-1} (koji se je jedini izuzetak u opštem postupku određivanja zapisa). Sve ovo su važni razlozi za korišćenje potpunog komplementa u računarima.

2.2.3 Brojevi u pokretnom zarezu

Za zapis realnih brojeva se najčešće koristi tzv. zapis u *pokretnom zarezu*. Osnovna ideja je da se brojevi zapisuju u obliku

$$\pm\, m \cdot b^e,$$

pri čemu se onda zasebno zapisuju znak broja, mantisa m i eksponent e, pri čemu se osnova b podrazumeva (danas se obično koristi osnova $b = 2$, dok se nekada koristila i osnova $b = 16$). Zapisivanje brojeva u pokretnom zarezu propisano je standardom *IEEE 754* iz 1985. godine, međunarodne asocijacije *Institut inženjera elektrotehnike i elektronike*, IEEE (engl. Institute of Electrical and Electronics Engineers).

2.3 Zapis teksta

Međunarodna organizacija za standardizaciju, *ISO (engl. International Standard Organization)* definiše tekst (ili dokument) kao *„informaciju namenjenu ljudskom sporazumevanju koja može biti prikazana u dvodimenzionalnom obliku. Tekst se sastoji od grafičkih elemenata kao što su karakteri, geometrijski ili fotografski elementi ili njihove kombinacije, koji čine sadržaj dokumenta“*. Iako se tekst obično zamišlja kao dvodimenzioni objekat, u računarima se tekst predstavlja kao jednodimenzioni (linearni) niz karaktera koji pripadaju određenom unapred fiksiranom skupu karaktera. U zapisu teksta, koriste se specijalni karakteri koji označavaju prelazak u novi red, tabulator, kraj teksta i slično.

Osnovna ideja koja omogućava zapis teksta u računarima je da se svakom karakteru pridruži određen (neoznačeni) ceo broj (a koji se interno u računarima zapisuje binarno) i to na unapred dogovoreni način. Ovi brojevi se nazivaju *kôdovima karaktera* (engl. character codes). Tehnička ograničenja ranih računara kao i neravnomeran razvoj računarstva između različitih zemalja, doveli su do toga da postoji više različitih standardnih tabela koje dodeljuju numeričke kôdove karakterima. U zavisnosti od broja bitova potrebnih za kodiranje karaktera, razlikuju se 7-bitni kôdovi, 8-bitni kôdovi, 16-bitni kôdovi, 32-bitni kôdovi, kao i kodiranja promenljive dužine koja različitim karakterima dodeljuju kôdove različite dužine.

Postoji veoma jasna razlika između karaktera i njihove grafičke reprezentacije. Elementi pisanog teksta koji najčešće predstavljaju grafičke reprezentacije pojedinih karaktera nazivaju se *glifovi* (engl. glyph), a skupovi glifova nazivaju se *fontovi* (engl. font). Korespondencija između karaktera i glifova ne mora biti jednoznačna. Naime, softver koji prikazuje tekst može više karaktera predstaviti jednim glifom (to su takozvane *ligature*, kao na primer glif za karaktere „f“ i „i“: fi), dok jedan isti karakter može biti predstavljen različitim glifovima u zavisnosti od svoje pozicije u reči. Takođe, moguće je da određeni fontovi ne sadrže glifove za određene karaktere i u tom slučaju se tekst ne prikazuje na željeni način, bez obzira što je ispravno kodiran. Fontovi koji se obično instaliraju uz operativni sistem sadrže glifove za karaktere koji su popisani na takozvanoj *WGL4* listi (*Windows Glyph List 4*) koja sadrži uglavnom karaktere korišćene u evropskim jezicima, dok je za ispravan prikaz, na primer, kineskih karaktera, potrebno instalirati dodatne fontove. Specijalnim karakterima se najčešće ne pridružuju zasebni grafički likovi.

Englesko govorno područje. Tokom razvoja računarstva, broj karaktera koje je bilo poželjno kodirati je postajao sve veći. Pošto je računarstvo u ranim fazama bilo razvijano uglavnom u zemljama engleskog govornog područja, bilo je potrebno predstaviti sledeće karaktere:

- Mala slova engleskog alfabeta: `a, b, ... , z`
- Velika slova engleskog alfabeta: `A, B, ... , Z`
- Cifre `0, 1, ..., 9`
- Interpunkcijske znake: `, . : ; + * - _ ( ) [ ] { }` ...
- Specijalne znake: kraj reda, tabulator, ...

Standardne tabele kôdova ovih karaktera su se pojavile još tokom 1960-ih godina. Najrasprostranjenije od njih su:

- *EBCDIC* - IBM-ov standard, korišćen uglavnom na mainframe računarima, pogodan za bušene kartice.

- *ASCII* - standard iz koga se razvila većina danas korišćenih standarda za zapis karaktera.

ASCII. *ASCII (American Standard Code for Information Interchange)* je standard uspostavljen 1968. godine od strane *Američkog nacionalnog instituta za standarde, (engl. American National Standard Institute)* koji definiše sedmobitan zapis kôda svakog karaktera što daje mogućnost zapisivanja ukupno 128 različitih karaktera, pri čemu nekoliko kôdova ima dozvoljeno slobodno korišćenje. *ISO* takođe delimično definiše ASCII tablicu kao deo svog standarda *ISO 646 (US)*.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	NUL	STX	SOT	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Slika 2.2: ASCII tablica

Osnovne osobine ASCII standarda su:

- Prva 32 karaktera – od $(00)_{16}$ do $(1F)_{16}$ – su specijalni kontrolni karakteri.
- Ukupno 95 karaktera ima pridružene grafičke likove (engl. printable characters).
- Cifre 0-9 predstavljene su kôdovima $(30)_{16}$ do $(39)_{16}$, tako da se njihov ASCII zapis jednostavno dobija dodavanjem prefiksa 011 na njihov binarni zapis.
- Kôdovi velikih i malih slova se razlikuju u samo jednom bitu u binarnoj reprezentaciji. Na primer, A se kodira brojem $(41)_{16}$ odnosno $(100\,0001)_2$, dok se a kodira brojem $(61)_{16}$ odnosno $(110\,0001)_2$. Ovo omogućava da se konverzija veličine slova u oba smera može vršiti efikasno.
- Slova su poređana u *kolacionu sekvencu*, u skladu sa engleskim alfabetom.

Različiti operativni sistemi predviđaju različito kodiranje oznake za prelazak u novi red. Tako operativni sistem Windows podrazumeva da se prelazak u novi red kodira sa dva kontrolna karaktera i to CR (carriage return) predstavljen kôdom $(0D)_{16}$ i LF (line feed) predstavljen kodom $(0A)_{16}$, operativni sistem Unix i njegovi derivati (pre svega Linux) podrazumevaju da se koristi samo karakter LF, dok MacOS podrazumeva korišćenje samo karaktera CR.

Nacionalne varijante ASCII tablice i ISO 646. Tokom 1980-ih, *Jugoslovenski zavod za standarde* definisao je standard *YU-ASCII (YUSCII, JUS I.B1.002, JUS I.B1.003)* kao deo standarda ISO 646, tako što su kôdovi koji imaju slobodno korišćenje (a koji u ASCII tabeli uobičajeno kodiraju zagrade i određene interpunkcijske znakove) dodeljeni našim dijakriticima:

YUSCII	ASCII	kôd	YUSCII	ASCII	kôd
Ž	@	$(40)_{16}$	ž	`	$(60)_{16}$
Š	[	$(5B)_{16}$	š	{	$(7B)_{16}$
Đ	\	$(5C)_{16}$	đ	|	$(7C)_{16}$
Ć	]	$(5D)_{16}$	ć	}	$(7D)_{16}$
Č	^	$(5E)_{16}$	č	~	$(7E)_{16}$

Osnovne mane YUSCII kodiranja su to što ne poštuje abecedni poredak, kao i to da su neke zagrade i važni interpunkcijski znaci izostavljeni.

8-bitna proširenja ASCII tabele. Podaci se u računaru obično zapisuju bajt po bajt. S obzirom na to da je ASCII sedmobitni standard, ASCII karakteri se zapisuju tako što se njihov sedmobitni kôd proširi vodećom nulom. Ovo znači da jednobajtni zapisi u kojima je vodeća cifra 1, tj. raspon od $(80)_{16}$ do $(FF)_{16}$ nisu iskorišćeni. Međutim, ni ovih dodatnih 128 kôdova nije dovoljno da se kodiraju svi karakteri koji su potrebni za zapis tekstova na svim jezicima (ne samo na engleskom). Zbog toga je, umesto jedinstvene tabele koja bi proširivala ASCII na 256 karaktera, standardizovano nekoliko takvih tabela, pri čemu svaka od tabela sadrži karaktere potrebne za zapis određenog jezika odnosno određene grupe jezika. Praktičan problem je što postoji dvostruka standardizacija ovako kreiranih kôdnih strana i to od strane ISO (International Standard Organization) i od strane značajnih korporacija, pre svega kompanije *Microsoft*.

ISO je definisao familiju 8-bitnih kôdnih strana koje nose zajedničku oznaku *ISO/IEC 8859* (kôdovi od $(00)_{16}$ do $(1F)_{16}$, $(7F)_{16}$ i od $(80)_{16}$ do $(9F)_{16}$ ostali su nedefinisani ovim standardom, iako se često u praksi popunjavaju određenim kontrolnim karakterima):

ISO-8859-1	Latin 1	većina zapadno-evropskih jezika
ISO-8859-2	Latin 2	centralno i istočno-evropski jezici
ISO-8859-3	Latin 3	južno-evropski jezici
ISO-8859-4	Latin 4	severno-evropski jezici
ISO-8859-5	Latin/Cyrillic	ćirilica većine slovenskih jezika
ISO-8859-6	Latin/Arabic	najčešće korišćeni arapski
ISO-8859-7	Latin/Greek	moderni grčki alfabet
ISO-8859-8	Latin/Hebrew	moderni hebrejski alfabet

Microsoft je definisao familju 8-bitnih strana koje se označavaju kao *Windows-125x* (ove strane se nekada nazivaju i *ANSI*). Za srpski jezik, značajne su kôdne strane:

Windows-1250	centralno i istočno-evropski jezici
Windows-1251	ćirilica većine slovenskih jezika
Windows-1252	(često se neispravno naziva i ANSI) zapadno-evropski jezici

Unicode. Iako navedene kôdne strane omogućavaju kodiranje tekstova koji nisu na engleskom jeziku, nije moguće, na primer, u istom tekstu koristiti i ćirilicu i latinicu. Takođe, za azijske jezike nije dovoljno 256 mesta za zapis

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8																
9																
A	NBSP	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	SHY	®	¯
B	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	

Slika 2.3: ISO-8859-1 tablica

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8																
9																
A	NBSP	Ą	˘	Ł	¤	Ľ	Ś	§	¨	Š	Ş	Ť	Ź	SHY	Ž	Ż
B	°	ą	˛	ł	´	ľ	ś	ˇ	¸	š	ş	ť	ź	˝	ž	ż
C	Ŕ	Á	Â	Ă	Ä	Ĺ	Ć	Ç	Č	É	Ę	Ë	Ě	Í	Î	Ď
D	Đ	Ń	Ň	Ó	Ô	Ő	Ö	×	Ř	Ů	Ú	Ű	Ü	Ý	Ţ	ß
E	ŕ	á	â	ă	ä	ĺ	ć	ç	č	é	ę	ë	ě	í	î	ď
F	đ	ń	ň	ó	ô	ő	ö	÷	ř	ů	ú	ű	ü	ý	ţ	

Slika 2.4: ISO-8859-2 tablica

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	€		‚		„	…	†	‡		‰	Š	‹	Ś	Ť	Ž	Ź
9		‘	’	“	”	•	–	—		™	š	›	ś	ť	ž	ź
A		ˇ	˘	Ł	¤	Ą	¦	§	¨	©	Ş	«	¬		®	Ż
B	°	±	˛	ł	´	µ	¶	·	¸	ą	ş	»	Ľ	˝	ľ	ż
C	Ŕ	Á	Â	Ă	Ä	Ĺ	Ć	Ç	Č	É	Ę	Ë	Ě	Í	Î	Ď
D	Đ	Ń	Ň	Ó	Ô	Ő	Ö	×	Ř	Ů	Ú	Ű	Ü	Ý	Ţ	ß
E	ŕ	á	â	ă	ä	ĺ	ć	ç	č	é	ę	ë	ě	í	î	ď
F	đ	ń	ň	ó	ô	ő	ö	÷	ř	ů	ú	ű	ü	ý	ţ	

Slika 2.5: Windows-1250 tablica

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8																
9																
A	NBSP	Ё	Ђ	Ѓ	Є	S	I	Ї	J	Љ	Њ	Ћ	Ќ	SHY	Ў	Џ
B	А	Б	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О	П
C	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я
D	а	б	в	г	д	е	ж	з	и	й	к	л	м	н	о	п
E	р	с	т	у	ф	х	ц	ч	ш	щ	ъ	ы	ь	э	ю	я
F	№	ё	ђ	ѓ	є	s	i	ї	j	љ	њ	ћ	ќ	§	ў	

Slika 2.6: ISO-8859-5 tablica

svih karaktera. Pošto je kapacitet računara vremenom rastao, postepeno se krenulo sa standardizacijom skupova karaktera koji karaktere kodiraju sa više od jednog bajta. Kasnih 1980-ih, dve velike organizacije započele su standar-

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	Ђ		‚		„	…	†	‡		‰	Љ	‹	Њ	Ќ	Ћ	Џ
9		‘	’	“	”	•	–	—		™	љ	›	њ	ќ	ћ	џ
A		Ў	ў	Ј	¤	Ґ	¦	§	Ё	©	Є	«	¬		®	Ї
B	°	±	І	і	ґ	µ	¶	·	ё	№	є	»	ј	Ѕ	ѕ	ї
C	А	Б	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О	П
D	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я
E	а	б	в	г	д	е	ж	з	и	й	к	л	м	н	о	п
F	р	с	т	у	ф	х	ц	ч	ш	щ	ъ	ы	ь	э	ю	

Slika 2.7: Windows-1251 tablica

dizaciju tzv. univerzalnog skupa karaktera (engl. Universal Character Set — UCS). To su bili ISO, kroz standard *10646* i projekat *Unicode*, organizovan i finansiran uglavnom od strane američkih kompanija koje su se bavile proizvodnjom višejezičkog softvera (Xerox Parc, Apple, Sun Microsystems, Microsoft, ...).

ISO 10646 zamišljen je kao četvorobajtni standard. Prvih 65536 karaktera koristi se kao osnovni višejezični skup karaktera, dok je preostali prostor ostavljen kao proširenje za drevne jezike, naučnu notaciju i slično.

Unicode je za cilj imao da bude:

- Univerzalan (UNIversal) — sadrži sve savremene jezike sa pismom;
- jedinstven (UNIque) — bez dupliranja karaktera - kodiraju se pisma, a ne jezici;
- uniforman (UNIform) — svaki karakter sa istim brojem bitova.

Početna verzija Unicode standarda svakom karakteru dodeljuje dvobajtni kôd (tako da kôd svakog karaktera sadrži tačno 4 heksadekadne cifre). Dakle, moguće je dodeliti kôdove za ukupno $2^{16} = 65536$ različitih karaktera. S vremenom se shvatilo da dva bajta neće biti dovoljno za zapis svih karaktera koji se koriste na planeti, pa je odlučeno da se skup karaktera proširi i Unicode danas dodeljuje kôdove od $(000000)_{16}$ do $(10FFFF)_{16}$ podeljenih u 16 tzv. ravni, pri čemu svaka ravan sadrži 65536 karaktera. U najčešćoj upotrebi je *osnovna višejezična ravan (engl. basic multilingual plane)* koja sadrži većinu danas korišćenih karaktera (uključujući i CJK — Chinese, Japanese, Korean — karaktere koji se najčešće koriste) čiji su kôdovi između $(0000)_{16}$ i $(FFFF)_{16}$.

Vremenom su se pomenuta dva projekta UCS i Unicode združila i danas postoji izuzetno preklapanje između ova dva standarda.

U sledećoj tabeli je naveden raspored određenih grupa karaktera u osnovnoj višejezičkoj ravni:

0020-007E	ASCII printable
00A0-00FF	Latin-1
0100-017F	Latin extended A (osnovno proširenje latinice, sadrži sve naše dijakritike)
0180-027F	Latin extended B
...	
0370-03FF	grčki alfabet
0400-04FF	ćirilica
...	
2000-2FFF	specijalni karakteri
3000-3FFF	CJK (Chinese-Japanese-Korean) simboli
...	

Unicode standard u suštini predstavlja veliku tabelu koja svakom karakteru dodeljuje broj. Standardi koji opisuju kako se niske karaktera prevode u nizove bajtova se definišu dodatno.

UCS-2. ISO definiše UCS-2 standard koji svaki Unicode karakter osnovne višejezičke ravni jednostavno zapisuje sa odgovarajuća dva bajta.

UTF-8. Latinični tekstovi kodirani korišćenjem UCS-2 standarda sadrže veliki broj nula. Ne samo što te nule zauzimaju dosta prostora, već zbog njih softver koji je razvijen za rad sa dokumentima u ASCII formatu ne može da radi bez izmena nad dokumentima kodiranim korišćenjem UCS-2 standarda. *Unicode Transformation Format (UTF-8)* je algoritam koji svakom dvobajtnom Unicode karakteru dodeljuje određeni niz bajtova čija dužina varira od 1 do najviše 3. UTF je ASCII kompatibilan, što znači da se ASCII karakteri zapisuju pomoću jednog bajta, na standardni način. Konverzija se vrši na osnovu sledećih pravila:

raspon	binarno zapisan Unicode kôd	binarno zapisan UTF-8 kôd
0000-007F	00000000 0xxxxxxx	0xxxxxxx
0080-07FF	00000yyy yyxxxxxx	110yyyyy 10xxxxxx
0800-FFFF	zzzzyyyy yyxxxxxx	1110zzzz 10yyyyyy 10xxxxxx

Na primer, karakter A koji se nalazi u ASCII tabeli ima Unicode kôd $(0041)_{16} = (0000\,0000\ 0100\,0001)_2$, pa se na osnovu prvog reda prethodne tabele u UTF-8 kodiranju zapisuje kao $(01000001)_2 = (41)_{16}$. Karakter Š ima Unicode kôd $(0160)_{16} = (0000\,0001\ 0110\,0000)_2$. Na njega se primenjuje drugi red prethodne tabele i dobija se da je njegov UTF-8 zapis $(1100\,0101\ 1010\,0000)_2 = (C5A0)_{16}$. Opisani konverzioni algoritam omogućava da se čitanje samo početka jednog bajta odredi da li je u pitanju karakter zapisan korišćenjem jednog, dva ili tri bajta.

2.4 Zapis multimedijalnih sadržaja

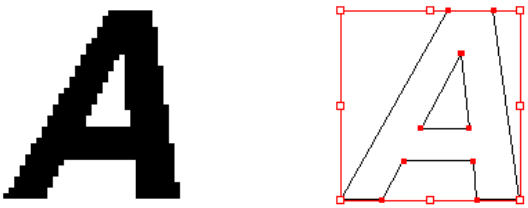
Računari imaju sve veću ulogu u većini oblasti svakodnevnog života. Od mašina koje su pre svega služile za izvođenje vojnih i naučnih izračunavanja, računari su postali i sredstvo za kućnu zabavu (gledanje filmova, slušanje muzike), izvor informacija (internet) i nezaobilazno sredstvo u komunikaciji (elektronska pošta (engl. e-mail), ćaskanje (engl. chat, instant messaging), video

konferencije, telefoniranje korišćenjem interneta (VoIP), ...). Nagli razvoj i proširivanje osnovne namene računara je posledica široke raspoloživosti velike količine multimedijalnih informacija (slika, zvuka, filmova, ...) koje su zapisane u digitalnom formatu. Sve ovo je posledica tehnološkog napretka koji je omogućio jednostavno i jeftino digitalizovanje signala, skladištenje velike količine digitalno zapisanih informacija, kao i njihov brz prenos i obradu.

2.4.1 Zapis slika

Slike se u računaru zapisuju koristeći *vektorski* zapis, *rasterski* zapis ili *kombinovani zapis*.

U vektorskom obliku, zapis slike sastoji se od konačnog broja geometrijskih figura (tačaka, linija, krivih, poligona), pri čemu se svaka figura predstavlja koncizno svojim koordinatama ili jednačinom u Dekartovoj ravni. Slike koje računari generišu često koriste vektorsku grafiku. Vektorski zapisane slike često zauzimaju manje prostora, dozvoljavaju uvećavanje (engl. zooming) bez gubitaka na kvalitetu prikaza i mogu se lakše preuređivati, s obzirom na to da se objekti mogu nezavisno jedan od drugoga pomerati, menjati, dodavati i uklanjati.



Slika 2.8: Primer slike u rasterskoj (levo) i vektorskoj (desno) grafici

U rasterskom zapisu, slika je predstavljena pravougaonom matricom komponenti koje se nazivaju pikseli (engl. pixel - PICture ELement). Svaki piksel je individualan i opisan jednom bojom. Raster nastaje kao rezultat digitalizacije slike. Rasterska grafika se još naziva i bitmapirana grafika. Uređaji za prikaz (monitori, projektori), kao i uređaji za digitalno snimanje slika (fotoaparati, skeneri) koriste rasterski zapis.

Modeli boja. Za predstavljanje crno-belih slika, dovoljno je boju predstaviti intenzitetom svetlosti. Različite količine svetlosti se diskretizuju u konačan broj nivoa osvetljenja čime se dobija određeni broj nijansi sive boje. Ovakav model boja se naziva *grayscale*. Ukoliko se za zapis informacije o količini svetlosti koristi 1 bajt, ukupan broj nijansi sive boje je 256. U slučaju da se slika predstavlja sa ukupno dve boje (na primer, crna i bela, kao kod skeniranog teksta nekog dokumenta) koristi se model pod nazivom *Duotone* i boja se tada predstavlja jednim bitom.

U RGB modelu boja, kombinovanjem crvene (R), zelene (G) i plave (B) komponente svetlosti reprezentuju se ostale boje. Tako se, na primer, kombinovanjem crvene i zelene boje reprezentuje žuta boja. Bela boja se reprezentuje maksimalnim vrednosti sve tri osnovne komponente, dok se crna boja repre-

zentuje minimalnim vrednostima osnovnih komponenti. U ovom modelu, zapis boje čini informacija o intenzitetu crvene, plave i zelene komponente. RGB model boja se koristi kod uređaja koji boje prikazuju kombinovanjem svetlosti (monitori, projektori, ...). Ukoliko se za informaciju o svakoj komponenti koristi po jedan bajt, ukupan broj bajtova za zapis informacije o boji je 3, te je moguće koristiti $2^{24} = 16777216$ različitih boja. Ovaj model se često naziva i *TrueColor* model boja.

Za razliku od aditivnog RGB modela boja, kod kojeg se bela boja dobija kombinovanjem svetlosti tri osnovne komponente, u štampi se koristi subtraktivni CMY (Cyan-Magenta-Yellow) model boje kod kojeg se boje dobijaju kombinovanjem obojenih pigmenata na belom papiru. Kako se potpuno crna teško dobija mešanjem drugih pigmenata, a i kako je njena upotreba obično daleko najčešća, obično se prilikom štampanja uz CMY pigmente koristi i crni pigment čime se dobija model CMYK.

Za obradu slika pogodni su HSL ili HSV (poznat i kao HSB) model boja. U ovom modelu, svaka boja se reprezentuje Hue (H) komponentom (koja predstavlja ton boje), Saturation (S) komponentom (koja predstavlja zasićenost boje odnosno njenu „jarkost“) i Lightness (L), Value (V) ili Brightness (B) komponentom (koja predstavlja osvetljenost).

Formati zapisa rasterskih slika. Rasterske slike su reprezentovane matricom piksela, pri čemu se za svaki piksel čuva informacija o njegovoj boji. Dimenzije ove matrice predstavljaju tzv. *apsolutnu rezoluciju* slike. Apsolutna rezolucija i model boja koji se koristi određuju broj bajtova potrebnih za zapis slike. Na primer, ukoliko je apsolutna rezolucija slike 800x600 piksela, pri čemu se koristi RGB model boje sa 3 bajta po pikselu, potrebno je ukupno $800 \cdot 600 \cdot 3B = 1440000B \approx 1.373MB$ za zapis slike. Kako bi se smanjila količina informacija potrebnih za zapis slike, koriste se tehnike kompresije i to *(i)* kompresije bez gubitka (engl. lossless), i *(ii)* kompresije sa gubitkom (engl. lossy). Najčešće korišćeni formati u kojima se koristi tehnike kompresije bez gubitka danas su GIF i PNG (koji se koriste za zapis dijagrama i sličnih računarski generisanih slika), dok je najčešće korišćeni format sa gubitkom JPEG (koji se obično koristi za fotografije).

2.4.2 Zapis zvuka

Zvučni talas predstavlja oscilaciju pritiska koja se prenosi kroz vazduh ili neki drugi medijum (tečnost, čvrsto telo). Digitalizacija zvuka se vrši merenjem i zapisivanjem vazdušnog pritiska u kratkim vremenskim intervalima. Osnovni parametri koji opisuju zvučni signal su njegova amplituda (koja odgovara „glasnoći“) i frekvencija (koja odgovara „visini“). Pošto ljudsko uho obično čuje raspon frekvencija od 20Hz do 20KHz, dovoljno je koristiti frekvenciju odabiranja 40KHz, tj. dovoljno je izvršiti odabiranje oko 40 000 puta u sekundi. Na primer, AudioCD standard koji se koristi prilikom snimanja običnih audio kompakt diskova, propisuje frekvenciju odabiranja 44.1KHz. Za postizanje još većeg kvaliteta, neki standardi (miniDV, DVD, digital TV) propisuju odabiranje na frekvenciji 48KHz. Ukoliko se snima ili prenosi samo ljudski govor (na primer, u mobilnoj telefoniji), frekvencije odabiranja mogu biti i znatno manje. Drugi važan parametar digitalizacije je broj bita kojim se zapisuje svaki

pojedinačni odbirak. Najčešće se koristi 2 bajta po odbirku (16 bita), čime se dobija mogućnost zapisa $2^{16} = 65536$ različitih nivoa amplitude.

Kako bi se dobio prostorni osećaj zvuka, primenjuje se tehnika višekanalnog snimanja zvuka. U ovom slučaju, svaki kanal se nezavisno snima posebnim mikrofonom i reprodukuje na posebnom zvučniku. *Stereo* zvuk podrazumeva snimanje zvuka sa dva kanala. *Surround* sistemi podrazumevaju snimanje sa više od dva kanala (od 3 pa čak i do 10), pri čemu se često jedan poseban kanal izdvaja za specijalno snimanje niskofrekvencijskih komponenti zvuka (tzv. bas). Najpoznatiji takvi sistemi su 5+1 gde se koristi 5 regularnih i jedan bas kanal.

Kao i slika, nekomprimovan zvuk zauzima puno prostora. Na primer, jedan minut stereo zvuka snimljenog u AudioCD formatu zauzima $2 \cdot 44100 \frac{sample}{sec} \cdot 60sec \cdot 2 \frac{B}{sample} = 10584000B \approx 10.1MB$. Zbog toga se koriste tehnike kompresije, od kojeg je danas najkorišćenija tehnika kompresije sa gubitkom MP3 (MPEG-1 Audio-Layer 3). MP3 kompresija se zasniva na tzv. psiho-akustici koja proučava koje je komponente moguće ukloniti iz zvučnog signala, a da pritom ljudsko uho ne oseti gubitak kvaliteta signala.

Pitanja i zadaci

Pitanje 2.1. *Kako su u digitalnom računaru zapisani svi podaci? Koliko se cifara obično koristi za njihov zapis?*

Pitanje 2.2. *Koje su osnovne prednosti digitalnog zapisa podataka u odnosu na analogni? Koji su osnovni problemi u korišćenju digitalnog zapisa podataka?*

Pitanje 2.3. *Šta je Hornerova shema? Opisati Hornerova shemu pseudokôdom.*

Pitanje 2.4. *Koje su prednosti zapisa u potpunom komplementu u odnosu na zapis u obliku označene apsolutne vrednosti?*

Pitanje 2.5. *Šta je to IEEE 754?*

Pitanje 2.6. *Šta je to glif, a šta font? Da li je jednoznačna veza između karaktera i glifova? Navesti primere.*

Pitanje 2.7. *Koliko bitova koristi ASCII standard? Šta čini prva 32 karaktera ASCII tabele? Kako se određuje binarni zapis karaktera koji predstavljaju cifre?*

Pitanje 2.8. *Navesti barem dve jednobajtne kôdne strane koje sadrže ćirilične karaktere.*

Pitanje 2.9. *Nabrojati bar tri kôdne sheme u kojima može da se zapiše reč* računarstvo.

Pitanje 2.10. *Koliko bitova koristi ASCII tabela karaktera, koliko YUSCII tabela, koliko ISO-8859-1, a koliko osnovna UNICODE ravan? Koliko različitih karaktera ima u ovim tabelama?*

Pitanje 2.11. *Koja kodiranja teksta je moguće koristiti ukoliko se u okviru istog dokumenta želi zapisivanje teksta koji sadrži jedan pasus na srpskom (pisan latinicom), jedan pasus na nemačkom i jedan pasus na ruskom (pisan ćirilicom)?*

Pitanje 2.12. *U čemu je razlika između Unicode i UTF-8 kodiranja?*

Pitanje 2.13. *Prilikom prikazivanja filma, neki program prikazuje titlove tipa* `"raèunari æe biti..."`. *Objasniti u čemu je problem.*

Pitanje 2.14. *Šta je to piksel? Šta je to sempl?*

Zadatak 2.1. *Prevesti naredne brojeve u dekadni brojevni sistem:*
(a) $(10111001)_2$ *(b)* $(3C4)_{16}$ *(c)* $(734)_8$
Zadatak uraditi klasičnim postupkom, a zatim i korišćenjem Hornerove sheme. √

Zadatak 2.2. *Zapisati dekadni broj* 254 *u osnovama* 2, 8 *i* 16. √

Zadatak 2.3. *(a)*
Registar ima sadržaj

10101011010010001111010101010110011010111010101011100010100100 11.

Zapisati ovaj broj u heksadekadnom sistemu.

(b) Registar ima sadržaj $A3BF461C89BE23D7$. *Zapisati ovaj sadržaj u binarnom sistemu.*

√

Zadatak 2.4. *Na Vebu se boje obično predstavljaju šestocifrenim heksadekadnim kôdovima u RGB sistemu: prve dve cifre odgovaraju količini crvene boje, naredne dve količini zelene i poslednje dve količini plave. Koja je količina RGB komponenti (na skali od 0-255) za boju* `#35A7DC`*? Kojim se kôdom predstavlja čista žuta boja ako se zna da se dobija mešanjem crvene i zelene? Kako izgledaju kôdovi za nijanse sive boje?* √

Zadatak 2.5. *U registru se zapisuju neoznačeni brojevi. Koji raspon brojeva može da se zapiše ukoliko je širina registra:*
(a) 4 *bita* *(b)* 8 *bita* *(c)* 16 *bita* *(d)* 24 *bita* *(e)* 32 *bita* √

Zadatak 2.6. *Ukoliko se koristi binarni zapis neoznačenih brojeva širine* 8 *bita, zapisati brojeve:*
(a) 12 *(b)* 123 *(c)* 255 *(d)* 300 √

Zadatak 2.7. *U registru se zapisuju brojevi u potpunom komplementu. Koji raspon brojeva može da se zapiše ukoliko je širina registra:*
(a) 4 *bita* *(b)* 8 *bita* *(c)* 16 *bita* *(d)* 24 *bita* *(e)* 32 *bita* √

Zadatak 2.8. *Odrediti zapis narednih brojeva u binarnom potpunom komplementu širine* 8 *bita:*
(a) 12 *(b)* -123 *(c)* 0 *(d)* -18 *(e)* -128 *(f)* 200 √

Zadatak 2.9. *Odrediti zapis brojeva -5 i 5 u potpunom komplementu dužine 6 bita. Odrediti, takođe u potpunom komplementu dužine 6 bita, zapis zbira i proizvoda ova dva broja.* √

Zadatak 2.10. *Ukoliko se zna da je korišćen binarni potpuni komplement širine 8 bita, koji su brojevi zapisani?*
(a) 11011010 *(b)* 01010011 *(c)* 10000000 *(d)* 11111111
(e) 01111111 *(f)* 00000000
Šta predstavljaju dati zapisi ukoliko se zna da je korišćen zapis neoznačenih brojeva? √

Zadatak 2.11. *Odrediti broj bitova neophodan za kodiranje 30 različitih karaktera.* √

Zadatak 2.12. *Znajući da je dekadni kôd za karakter A 65, navesti kodirani zapis reči* `FAKULTET` *ASCII kôdovima u heksadekadnom zapisu. Dekodirati sledeću reč zapisanu u ASCII kôdu heksadekadno: 44 49 53 4B 52 45 54 4E 45.* √

Zadatak 2.13. *Korišćenjem ASCII tablice odrediti kôdove kojima se zapisuje tekst:* `"Programiranje 1"`. *Kôdove zapisati heksadekadno, oktalno, dekadno i binarno. Šta je sa kodiranjem teksta* `Matematički fakultet`*?* √

Zadatak 2.14. *Za reči* `računarstvo`, `informatika`, `њир-lat`, `ћирчиђ` *navesti da li ih je moguće kodirati narednim metodima i, ako jeste, koliko bajtova zauzimaju:*
(a) ASCII (b) Windows-1250 (c) ISO-8859-5 (d) ISO-8859-2
(e) Unicode (UCS-2) (f) UTF-8
√

Zadatak 2.15. *Odrediti (heksadekadno predstavljene) kôdove kojima se zapisuje tekst* `kružić` *u UCS-2 i UTF-8 kodiranjima. Rezultat proveriti korišćenjem HEX editora.* √

Vežba 2.1. *HEX editori su programi koji omogućavaju direktno pregledanje, kreiranje i ažuriranje bajtova koji sačinjavaju sadržaja datoteka. Korišćenjem HEX editora pregledati sadržaj nekoliko datoteka različite vrste (tekstualnih, izvršnih programa, slika, zvučnih zapisa, video zapisa, . . .).*

Vežba 2.2. *Uz pomoć omiljenog editora teksta (ili nekog naprednijeg, ukoliko editor nema tražene mogućnosti) kreirati datoteku koja sadrži listu imena 10 vaših najomiljenijih filmova (pisano latinicom uz ispravno korišćenje dijakritika). Datoteka treba da bude kodirana kodiranjem:*
(a) Windows-1250 (b) ISO-8859-2 (c) Unicode (UCS-2) (d) UTF-8
Otvoriti zatim datoteku iz nekog pregledača Veba i proučiti šta se dešava kada se menja kodiranje koje pregledač koristi prilikom tumačenja datoteke (obično meni `View->Character encoding`*). Objasniti i unapred pokušati predvideti ishod (uz pomoć odgovarajućih tabela koje prikazuju kôdne rasporede).*

Vežba 2.3. *Za datu datoteku kodiranu UTF-8 kodiranjem, korišćenjem editora teksta ili nekog od specijalizovanih alata (na primer,* `iconv`*) rekodirati ovu datoteku u ISO-8859-2. Eksperimentisati i sa drugim kodiranjima.*

Vežba 2.4. *Korišćenjem nekog naprednijeg grafičkog programa (na primer, GIMP ili Adobe Photoshop) videti kako se boja* `#B58A34` *predstavlja u CMY i HSB modelima.*

Glava 3

Algoritmi i izračunljivost

Neformalno govoreći, *algoritam*[1] je precizan opis postupka za rešavanje nekog problema u konačnom broju koraka. Algoritmi postoje u svim ljudskim delatnostima. U matematici, na primer, postoje algoritmi za sabiranje i za množenje prirodnih brojeva. U računarstvu, postoje algoritmi za određivanje najmanjeg elementa niza, uređivanje elemenata po veličini i slično. Da bi moglo da se diskutuje o tome šta se može a šta ne može izračunati algoritamski, potrebno je najpre precizno definisati pojam algoritma.

3.1 Formalizacije pojma algoritma

Precizni postupci za rešavanje matematičkih problema postojali su u vreme starogrčkih matematičara (na primer, Euklidov algoritam za određivanje najvećeg zajedničkog delioca dva broja), pa i pre toga. Ipak, sve do početka dvadesetog veka nije se uviđala potreba za preciznim definisanjem pojma algoritma. Tada je, u jeku reforme i novog utemeljivanja matematike, postavljeno pitanje da li postoji algoritam kojim se (pojednostavljeno rečeno) mogu dokazati sve matematičke teoreme[2]. Da bi se ovaj problem uopšte razmatrao, bilo je neophodno najpre definisati (matematički precizno) šta je to algoritam. U tome, bilo je dovoljno razmatrati algoritme za izračunavanje funkcija čiji su i argumenti i rezultujuće vrednosti prirodni brojevi (a drugi matematički i nematematički algoritmi se, digitalizacijom, mogu svesti na taj slučaj). Formalno zasnovan pojam algoritma omogućio je da kasnije budu identifikovani problemi

[1] Reč „algoritam“ ima koren u imenu persijskog astronoma i matematičara Al-Horezmija (engl. Muhammad ibn Musa al-Khwarizmi). On je 825. godine napisao knjigu, u međuvremenu nesačuvanu u originalu, verovatno pod naslovom „O računanju sa indijskim brojevima“. Ona je u dvanaestom veku prevedena na latinski, bez naslova, ali se na nju obično pozivalo njenim početnim rečima „Algoritmi de numero Indorum“, što je trebalo da znači „Al-Horezmi o indijskim brojevima“ (pri čemu je ime autora latinizovano u „Algoritmi“). Međutim, većina čitalaca je reč „Algoritmi“ shvatala kao množinu od nove, nepoznate reči „algoritam“ koja se vremenom odomaćila sa značenjem „metod za izračunavanje“.

[2] Ovaj problem, poznat pod imenom „Entscheidungsproblem“, postavio je David Hilbert 1928. godine. Poznato je da je još Gotfrid Lajbnic u XVII veku, nakon što je napravio mašinu za računanje, verovao da će biti moguće napraviti mašinski postupak koji će, manipulisanjem simbolima, biti u stanju da dâ odgovor na sva matematička pitanja. Ipak, 1930-ih, rezultatima Čerča, Tjuringa i Gedela pokazano je da ovakav postupak ne može da postoji.

za koje postoje algoritmi koji ih rešavaju, kao i problemi za koje ne postoje algoritmi koji ih rešavaju.

Pitanjem formalizacije pojma algoritma 1920-ih i 1930-ih godina nezavisno se bavilo nekoliko istaknutih matematičara i uvedeno je nekoliko raznorodnih formalizama, tj. nekoliko raznorodnih sistema izračunavanja. Najznačajniji među njima su:

- *Tjuringove mašine* (Tjuring),
- *rekurzivne funkcije* (Gedel[3] i Klini[4]),
- *λ-račun* (Čerč[5]),
- *Postove mašine* (Post[6]),
- *Markovljevi algoritmi* (Markov[7]),
- *mašine sa beskonačno mnogo registara* (engl. Unlimited Register Machines — URM)[8].

Navedeni sistemi nastali su pre savremenih računara i većina njih podrazumeva određene apstraktne mašine. I u to vreme bilo je jasno da je opis postupka dovoljno precizan ukoliko je moguće njime instruisati neku mašinu koja će taj postupak izvršiti. U tom duhu, i *savremeni programski jezici* predstavljaju precizne opise algoritama i moguće ih je pridružiti gore navedenoj listi. Značajna razlika je u tome što nabrojani formalizmi teže da budu što jednostavniji tj. da koriste što manji broj operacija i što jednostavije modele mašina, dok savremeni programski jezici teže da budu što udobniji za programiranje te uključuju veliki broj operacija (koje, sa teorijskog stanovišta, nisu neophodne jer se mogu definisati preko malog broja osnovnih operacija).

Blok dijagrami (kaže se i *dijagrami toka*, tj. *tokovnici*) mogu se smatrati poluformalnim načinom opisa algoritama. Oni neće biti razmatrani u teorijskom kontekstu ali će biti korišćeni u rešenjima nekih zadataka, radi lakšeg razumevanja.

Za sistem izračunavanja koji je dovoljno bogat da može da simulira Tjuringovu mašinu i tako da izvrši sva izračunavanja koja može da izvrši Tjuringova mašina kaže se da je *Tjuring potpun* (engl. Turing complete). Za sistem izračunavanja koji izračunava identičnu klasu funkcija kao Tjuringova mašina kažemo da je *Tjuring ekvivalentan* (engl. Turing equivalent).

Ako se neka funkcija može izračunati u nekom od navednih formalizama, onda kažemo da je ona *izračunljiva* u tom formalizmu. Iako su navedni formalizmi međusobno veoma različiti, može se dokazati da su klase izračunljivih funkcija identične za sve njih, tj. svi oni formalizuju isti pojam algoritma i izračunljivosti. Drugim rečina, svi navedeni formalizmi su Tjuring ekvivalentni. Zbog toga se, umesto pojmova poput, na primer, Tjuring-izračunljiva ili URM-izračunljiva funkcija (i sličnih) može koristiti samo termin *izračunljiva funkcija*.

[3] Kurt Gödel (1906-1978), austrijsko-američki matematičar.
[4] Stephen Kleene (1909-1994), američki matematičar.
[5] Alonzo Church (1903-1995), američki matematičar.
[6] Emil L. Post (1897-1954), američki matematičar.
[7] Andrej Andrejevič Markov (1856-1922), ruski matematičar.
[8] Postoji više srodnih formalizama koji se nazivaju URM. Prvi opisi mogu da se nađu u radovima Šeperdsona i Sturdžisa (engl. Sheperdson and Sturgis), dok je opis koji će biti dat u nastavku preuzet od Katlanda (engl. Nigel Cutland).

Svi navedeni formalizmi za izračunavanje, podrazumevaju u nekom smislu, pojednostavljeno rečeno, beskonaču raspoloživu memoriju. Zbog toga savremeni računari (koji raspolažu konačnom memorijom) opremljeni savremenim programskim jezicima nisu Tjuring potpuni, u strogom smislu. S druge starne, svako izračunavanje koje se može opisati na nekom modernom programskom jeziku, može se opisati i kao program za Tjuringovu mašinu ili neki drugi ekvivalentan formalizam.

3.2 Čerč-Tjuringova teza

Veoma važno pitanje je koliko formalne definicije algoritama uspevaju da pokriju naš intuitivni pojam algoritma, tj. da li je zaista moguće efektivno izvršiti sva izračunavanja definisana nekom od formalizacija izračunljivosti i, obratno, da li sva izračunavanja koja intuitivno umemo da izvršimo zaista mogu da budu opisana korišćenjem bilo kog od precizno definisanih formalizama izračunavanja. Intuitivno je jasno da je odgovor na prvo pitanje potvrdan (jer čovek može da simulira rad jednostavnih mašina za izračunavanje), ali oko drugog pitanja postoji doza rezerve. *Čerč-Tjuringova teza*[9] tvrdi da je odgovor na oba navedena pitanja potvrdan.

Čerč-Tjuringova teza: *Klasa intuitivno izračunljivih funkcija identična je sa klasom formalno izračunljivih funkcija.*

Ovo tvrđenje je hipoteza, a ne teorema i ne može biti formalno dokazana. Naime, ono govori o intuitivnom pojmu algoritma, čija svojstva ne mogu biti formalno, matematički ispitana. Ipak, u korist ove teze govori činjenica da je dokazano da su sve navedene formalne definicije algoritama međusobno ekvivalentne, kao i da do sada nije pronađen nijedan primer intuitivno, efektivno izvodivog postupka koji nije moguće formalizovati u okviru nabrojanih formalnih sistema izračunavanja.

3.3 ur mašine

U daljem tekstu, pojam izračunljivosti biće uveden i izučavan na bazi UR *mašina (*URM*)*. Iako su po skupu izračunljivih funkcija nabrojani formalizacije pojma algoritma jednake, oni se međusobno razlikuju po svom duhu, a UR mašina je verovatno najbliža savremenom stilu programiranja: UR mašina ima ograničen (mali) skup instrukcija, programe i memoriju.

Kao i drugi formalizmi izračunavanja, UR mašine su apstrakcije sine koje formalizuju pojam algoritma, predstavljajući matematičku idealizaciju računara. UR mašine su dizajnirane tako da koriste samo mali broj operacija. Očigledno je da nije neophodno da formalizam za izračunavanje kao osnovnu operaciju ima, na primer, stepenovanje prirodnih brojeva, jer se ono može svesti na množenje. Slično, nije neophodno da formalizam za izračunavanje kao primitivnu operaciju ima množenje prirodnih brojeva, jer se ono može svesti na sabiranje. Čak ni sabiranje nije neophodno, jer se može svesti na operaciju pronalaženja sledbenika (uvećavanja za vrednost 1, tj. inkrementiranja). Zaista, definicija

[9] Ovu tezu, svaki za svoj formalizam, formulisali su nezavisno Čerč i Tjuring.

prirodnih brojeva podrazumeva postojanje nule i operacije sledbenika[10] i sve operacije nad prirodnim brojevima se svode na te dve elementarne. Pored njih, UR mašine koriste još samo dve instrukcije. Sva ta četiri tipa instrukcija brojeve čitaju i upisuju ih na polja ili *registre* beskonačne trake koja služi kao prostor za čuvanje podataka, tj. kao memorija mašine. Registri trake označeni su sa $R_1, R_2, R_3, \ldots$. Svaki od njih u svakom trenutku sadrži neki prirodan broj. Stanje registara u nekom trenutku zovemo *konfiguracija*. Sadržaj k-tog registra (registra R_k) označava se sa r_k, kao što je to ilustrovano na sledećoj slici:

R_1	R_2	R_3	$\ldots$
r_1	r_2	r_3	$\ldots$

Spisak i kratak opis URM instrukcija (naredbi) dati su u tabeli 3.1.[11] U tabeli, na primer, $R_m \leftarrow 0$ označava da se vrednost 0 upisuje u registar R_m, a na primer $r_m := r_m + 1$ označava da se sadržaj registra R_m (vrednost r_m) uvećava za jedan i upisuje u registar R_m.

oznaka	naziv	efekat
$Z(m)$	nula–instrukcija	$R_m \leftarrow 0$ (tj. $r_m := 0$)
$S(m)$	instrukcija sledbenik	$R_m \leftarrow r_m + 1$ (tj. $r_m := r_m + 1$)
$T(m,n)$	instrukcija prenosa	$R_n \leftarrow r_m$ (tj. $r_n := r_m$)
$J(m,n,p)$	instrukcija skoka	ako je $r_m = r_n$, idi na p-tu; inače idi na sledeću instrukciju

Tabela 3.1: Tabela URM instrukcija

URM program P je konačan numerisan niz URM instrukcija. Instrukcije se izršavaju redom (počevši od prve), osim u slučaju instrukcije skoka. Izvršavanje programa se zaustavlja onda kada ne postoji instrukcija koju treba izvršiti (kada se dođe do kraja programa ili kada se naiđe na skok na instrukciju koja ne postoji u numerisanom nizu instrukcija).

Početnu konfiguraciju čini niz prirodnih brojeva $a_1, a_2, \ldots$ koji su upisani u registre $R_1, R_2, \ldots$. Ako je funkcija koju treba izračunati $f(x_1, x_2, \ldots, x_n)$, onda se podrazumeva da su vrednosti $x_1, x_2, \ldots, x_n$ redom smeštene u prvih n registara iza kojih sledi niz nula. Podrazumeva se i da, na kraju rada programa, rezultat treba da bude smešten u prvi registar.

Ako URM program P za početnu konfiguraciju $a_1, a_2, \ldots, a_n$ ne staje sa radom, onda pišemo $P(a_1, a_2, \ldots, a_n)\uparrow$. Inače, ako program staje sa radom i u prvom registru je, kao rezultat, vrednost b, onda pišemo $P(a_1, a_2, \ldots, a_n)\downarrow b$.

Kažemo da URM program izračunava funkciju $f : \mathbb{N}^n \to \mathbb{N}$ ako za svaku n-torku argumenata $a_1, a_2, \ldots, a_n$ za koju je funkcija f definisana i važi $f(a_1, a_2, \ldots, a_n) = b$ istovremeno važi i $P(a_1, a_2, \ldots, a_n)\downarrow b$. Funkcija je URM-*izračunljiva* ako postoji URM program koji je izračunava.

[10] Skup prirodnih brojeva se definiše induktivno, kao najmanji skup koji sadrži nulu i zatvoren je u odnosu na operaciju pronalaženja sledbenika.

[11] Oznake URM instrukcija potiču od naziva ovih instrukcija na engleskom jeziku (*zero instruction, succesor instruction, transfer instruction* i *jump instruction*).

Primer 3.1. *Neka je funkcija f definisana na sledeći način: $f(x, y) = x + y$. Vrednost funkcije f može se izračunati za sve vrednosti argumenata x i y* UR *mašinom. Ideja algoritma za izračunavanje vrednosti $x + y$ je da se vrednosti x doda vrednost 1, y puta, jer važi:*

$$x + y = x + \underbrace{1 + 1 + \ldots + 1}_{y}$$

Odgovarajući URM *program podrazumeva sledeću početnu konfiguraciju:*

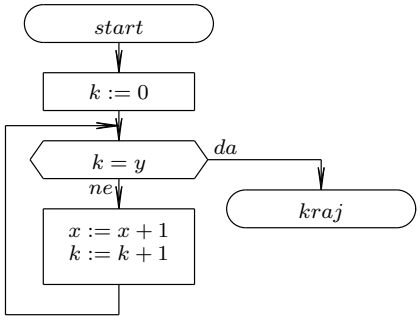
R_1	R_2	R_3	$\ldots$
x	y	0	$\ldots$

i sledeću konfiguraciju u toku rada programa:

R_1	R_2	R_3	$\ldots$
$x + k$	y	k	$\ldots$

gde je $k \in \{0, 1, \ldots, y\}$.

Algoritam se može zapisati u vidu dijagrama toka i u vidu URM *programa kao u nastavku:*



1. $J(3, 2, 100)$
2. $S(1)$
3. $S(3)$
4. $J(1, 1, 1)$

Naglasimo da u dijagramu toka postoji dodela $k := 0$, ali u URM programu ne postoji odgovarajuća instrukcija. To je zbog toga što se, u skladu sa definicijom UR mašina podrazumeva da svi registri (nakon onih koje sadrže argumente programa) sadrže nule. Isto važi i za URM programe koji slede.

Prekid rada programa je realizovan skokom na nepostojeću instrukciju 100[12]. Bezuslovni skok je realizovan naredbom oblika $J(1, 1, \ldots)$ — poređenje registra sa samim sobom uvek garantuje jednakost te se skok vrši uvek.

Primer 3.2. *Neka je funkcija f definisana na sledeći način:*

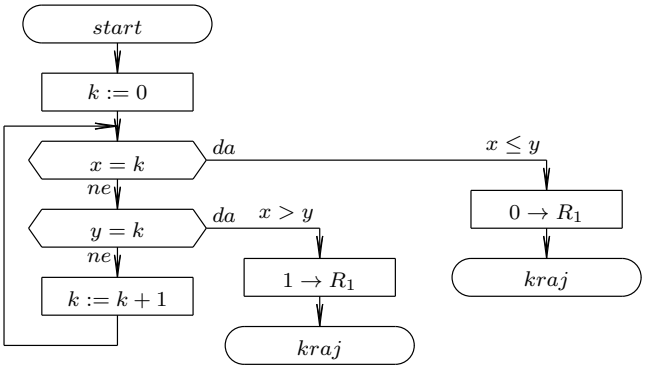
$$f(x, y) = \begin{cases} 0 & , \text{ ako } x \leq y \\ 1 & , \text{ inače} \end{cases}$$

[12]Broj 100 je odabran proizvoljno kao broj sigurno veći od broja instrukcija u programu. I u programima koji slede, uniformnosti radi, za prekid programa će se takođe koristiti skok na instrukciju 100.

URM *program koji je računa koristi sledeću konfiguraciju u toku rada:*

R_1	R_2	R_3	...
x	y	k	...

gde k dobija redom vrednosti $0, 1, 2 \ldots$ sve dok ne dostigne vrednost x ili vrednost y. Prva dostignuta vrednost je broj ne veći od onog drugog. U skladu sa tim zaključkom i definicijom funkcije f, izračunata vrednost je 0 ili 1.



1.	$J(1,3,5)$	$x = k?$
2.	$J(2,3,7)$	$y = k?$
3.	$S(3)$	$k := k + 1$
4.	$J(1,1,1)$	
5.	$Z(1)$	$0 \to R_1$
6.	$J(1,1,100)$	*kraj*
7.	$Z(1)$	
8.	$S(1)$	$1 \to R_1$

Primer 3.3. *Napisati* URM *program koji izračunava funkciju*

$$f(x) = [\sqrt{x}]$$

Predloženi algoritam se zasniva na sledećoj osobini:

$$n = [\sqrt{x}] \Leftrightarrow n^2 \leq x < (n+1)^2$$

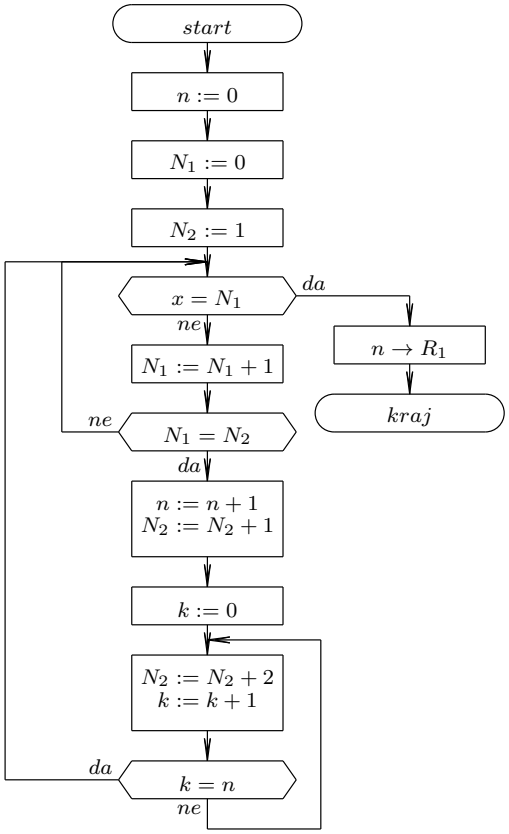
Odgovarajući URM *program podrazumeva sledeću početnu konfiguraciju:*

R_1	R_2	...
x	0	...

i sledeću konfiguraciju u toku svog rada:

R_1	R_2	R_3	R_4	R_5	...
x	n	$N_1 = n^2$	$N_2 = (n+1)^2$	k	...

Osnovna ideja algoritma je uvećavanje broja n za 1 i odgovarajućih vrednosti N_1 i N_2, sve dok se broj x ne nađe između njih. Nakon što se N_1 i N_2 postave na vrednosti kvadrata dva uzastopna broja n^2 i $(n+1)^2$, broj N_1 se postepeno inkrementira i proverava se da li je jednak broju x. Ukoliko se ne naiđe na x, a N_1 dostigne N_2, tada se n inkrementira i tada oba broja N_1 i N_2 imaju vrednost n^2 (naravno, za uvećano n). Tada je potrebno N_2 postaviti na vrednost $(n+1)^2$, tj. potrebno je uvećati N_2 za $(n+1)^2 - n^2 = 2n + 1$. Ovo se postiže tako što se N_2 najpre uveća za 1, a zatim n puta uveća za 2. Nakon toga se ceo postupak ponavlja.



1.	$Z(2)$	$n := 0$
2.	$Z(3)$	$N_1 := 0$
3.	$S(4)$	$N_2 := 1$
4.	$J(1,3,16)$	$x = N_1?$
5.	$S(3)$	$N_1 := N_1 + 1$
6.	$J(3,4,8)$	$N_1 = N_2?$
7.	$J(1,1,4)$	
8.	$S(2)$	$n := n + 1$
9.	$S(4)$	$N_2 := N_2 + 1$
10.	$Z(5)$	$k := 0$
11.	$S(4)$	$N_2 := N_2 + 1$
12.	$S(4)$	$N_2 := N_2 + 1$
13.	$S(5)$	$k := k + 1$
14.	$J(5,2,4)$	$k = n?$
15.	$J(1,1,11)$	
16.	$T(2,1)$	$n \to R_1$

Primer 3.4. *Neka je funkcija f definisana na sledeći način:*

$$f(x) = \begin{cases} 0 & , \textit{ako je } x = 0 \\ nedefinisano & , \textit{inače} \end{cases}$$

Nedefinisanost funkcije se postiže time što se napravi program koji se ne zaustavlja, izuzev ako je vrednost prvog argumenta jednaka 0:

1. $J(1, 2, 100)$
2. $J(1, 1, 1)$

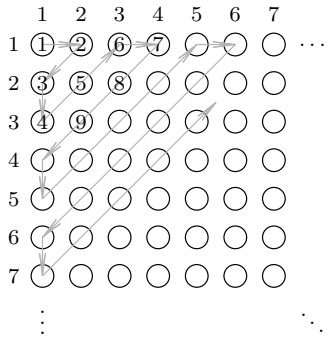
3.4 Enumeracija urm programa

Kardinalnost. Za dva skupa kaže se da *imaju istu kardinalnost* ako i samo ako je između njih moguće uspostaviti bijektivno preslikavanje. Za skupove koji imaju istu kardinalnost kao skup prirodnih brojeva kaže se da su *prebrojivi*. Dakle, neki skup je prebrojiv ako i samo ako je njegove elemente moguće poređati u niz (niz odgovara bijekciji sa skupom prirodnih brojeva). Za skupove koji su ili konačni ili prebrojivi, kaže se da su *najviše prebrojivi*. Georg Kantor[13] je prvi pokazao da skup realnih brojeva ima veću kardinalnost od skupa prirodnih brojeva, tj. da skup realnih brojeva nije prebrojiv.

Primer 3.5. *Skupovi parnih i neparnih brojeva imaju istu kardinalnost jer je između njih moguće uspostaviti bijektivno preslikavanje $f(n) = n + 1$. Ono što je na prvi pogled iznenađujuće je da su oba ova skupa prebrojiva, odnosno imaju istu kardinalnost kao i skup prirodnih brojeva, iako su njegovi pravi podskupovi (npr. $f(n) = 2 \cdot n$ uspostavlja bijekciju između skupa prirodnih i skupa parnih brojeva).*

Lema 3.1. *Uređenih parova prirodnih brojeva ima prebrojivo mnogo.*

Dokaz: Razmotrimo beskonačnu tabelu elemenata koja odgovara svim uređenim parovima prirodnih brojeva. Moguće je izvršiti „cik-cak“ obilazak ove tabele, polazeći od gornjeg levog ugla kao što je to pokazano na slici 3.1.



Slika 3.1: Cik-cak nabrajanje

Prilikom obilaska, svakom elementu je moguće dodeliti jedinstven redni broj, čime je uspostavljena tražena bijekcija sa skupom prirodnih brojeva. □

[13] Georg Cantor, 1845–1918, nemački matematičar.

Korišćenjem sličnih ideja, mogu se dokazati i sledeća tvrđenja.

Lema 3.2. *Dekartov proizvod konačno mnogo prebrojivih skupova je prebrojiv.*

Lema 3.3. *Najviše prebrojiva unija prebrojivih skupova je prebrojiv skup.*

Enumeracija URM programa. Važno pitanje je koliko ima URM programa i koliko ima različitih URM programa (tj. onih koji izračunavaju različite funkcije). Očigledno je da ih ima beskonačno, a naredna tvrđenja govore da ih ima prebrojivo mnogo.

Lema 3.4. *Postoji prebrojivo mnogo različitih* URM *instrukcija.*

Dokaz: Instrukcija tipa Z, trivijalno, ima prebrojivo mnogo (Z instrukcije se mogu poređati u niz na sledeći način: $Z(1), Z(2), \ldots$). Slično je i sa instrukcijama tipa S. T instrukcije bijektivno odgovaraju parovima prirodnih brojeva, pa ih (na osnovu leme 3.2) ima prebrojivo mnogo. Slično, J instrukcije odgovaraju bijektivno trojkama prirodnih brojeva, pa ih (na osnovu leme 3.2) ima prebrojivo mnogo. Skup svih instrukcija je unija ova četiri prebrojiva skupa, pa je na osnovu leme 3.3 i ovaj skup prebrojiv. □

Teorema 3.1. *Različitih* URM *programa ima prebrojivo mnogo*

Dokaz: Skup svih URM programa se može predstaviti kao unija skupa programa koji imaju jednu instrukciju, skupa programa koji imaju dve instrukcije i tako dalje. Svaki od ovih skupova je prebrojiv (na osnovu leme 3.2) kao konačan Dekartov stepen prebrojivog skupa instrukcija. Dakle, skup svih URM programa je prebrojiv (na osnovu leme 3.3) kao prebrojiva unija prebrojivih skupova. □

Na osnovu teoreme 3.1, moguće je uspostaviti bijekciju između skupa svih URM programa i skupa prirodnih brojeva. Drugim rečima, može se definisati pravilo koje svakom URM programu dodeljuje jedinstven prirodan broj i koje svakom prirodnom broju dodeljuje jedinstven URM program. Zbog toga, za fiksirano dodeljivanje brojeva programima, možemo da govorimo o prvom, drugom, trećem, ..., stotom URM programu itd.

3.5 Neizračunljivost i neodlučivost

Razmotrimo narednih nekoliko problema.

1. Neka su data dva konačna skupa reči. Pitanje je da li je moguće nadovezati nekoliko reči prvog skupa i, nezavisno, nekoliko reči drugog skupa tako da se dobije ista reč. Na primer, za skupove $\{a, ab, bba\}$ i $\{baa, aa, bb\}$, jedno rešenje je $bba \cdot ab \cdot bba \cdot a = bb \cdot aa \cdot bb \cdot baa$. Za skupove $\{ab, bba\}$ i $\{aa, bb\}$ rešenje ne postoji, jer se nadovezivanjem reči prvog

skupa uvek dobija reč čija su poslednja dva slova različita, dok se nadovezivanjem reči drugog skupa uvek dobija reč čija su poslednja dva slova ista. Zadatak je utvrditi da li postoji opšti algoritam koji za proizvoljna dva zadata skupa reči određuje da li tražena nadovezivanja postoje.[14]

2. Neka su date Diofantske jednačine $p(x_1, \ldots, x_n) = 0$, gde je p polinom sa celobrojnim koeficijentima. Zadatak je utvrditi da li postoji opšti algoritam kojim se određuje da li proizvoljna zadata diofantska jednačina ima racionalnih rešenja.[15]

3. Zadatak je utvrditi da li postoji opšti algoritam koji proverava da li se proizvoljni zadati program P zaustavlja za date ulazne parametre.[16]

4. Zadatak je utvrditi da li postoji opšti algoritam koji za proizvoljni zadati skup aksioma i zadato tvrđenje proverava da li je tvrđnje posledica aksioma.[17]

Za sva četiri navedena problema pokazano je da su *algoritamski nerešivi* ili *neodlučivi* (tj. ne postoji izračunljiva funkcija koja ih rešava). Ovo ne znači da nije moguće rešiti pojedine instance problema[18], već samo da ne postoji jedinstven, opšti postupak koji bi mogao da reši proizvoljnu instancu problema. Pored nabrojanih, ima još mnogo važnih neodlučivih problema

U nastavku će precizno, korišćenjem formalizma UR mašina, biti opisan treći od navedenih problema, tj. *halting problem*, izuzetno važan za teorijsko računarstvo.

Problem ispitivanja zaustavljanja programa (halting problem). Pitanje zaustavljanja računarskih programa je jedno od najznačajnijih pitanja računarstva i programiranja. Često je veoma važno da li se neki program zaustavlja za neku ulaznu vrednost, da li se zaustavlja za ijednu ulaznu vrednost i slično. Za mnoge konkretne programe i za mnoge konkretne ulazne vrednosti, na ovo pitanje može se odgovoriti. No, nije očigledno da li postoji opšti postupak kojim se za proizvoljni dati program i proizvoljne vrednosti ulaznih argumenata može proveriti da li se program zaustavlja ako se pokrene sa tim argumentima.

Iako se problem ispitivanja zaustavljanja može razmatrati i za programe u savremenim programskim jezicima, u nastavku ćemo razmotriti formulaciju halting problema za URM programe:

Da li postoji URM *program koji na ulazu dobija drugi* URM *program P i neki broj x i ispituje da li se program P zaustavlja za ulazni parametar x?*

[14] Ovaj problem se naziva *Post's correspondence problem*, jer ga je postavio i rešio Emil Post 1946. godine.

[15] Ovaj problem je 10. Hilbertov problem izložen 1900. godine kao deo skupa problema koje „matematičari XIX veka ostavljaju u amanet matematičarima XX veka". Problem je rešio Matijaševič 1970-ih godina.

[16] Ovaj problem rešio je Alan Tjuring 1936. godine.

[17] Ovaj, već pomenut, problem, formulisao je David Hilbert 1928. godine, a nešto kasnije rešenje je proisteklo iz rezultata Čerča, Tjuringa i Gedela.

[18] Instanca ili *primerak problema* je jedan konkretan zadatak koji ima isti oblik kao i opšti problem. Na primer, za prvi u navedenom spisku problema, jedna instanca je zadatak ispitivanja da li je moguće nadovezati nekoliko reči skupa $\{ab, bba\}$ i, nezavisno, nekoliko reči skupa $\{aa, bb\}$ tako da se dobije ista reč.

Problem prethodne formulacije je činjenica da traženi URM program mora na ulazu da prihvati kao svoj argument drugi URM program, što je naizgled nemoguće, s obzirom na to da URM programi kao argumente mogu da imaju samo prirodne brojeve. Ipak, ovo se jednostavno razrešava zahvaljujući tome što je svakom URM programu P moguće dodeliti jedinstveni prirodan broj n koji ga identifikuje, i obratno, svakom broju n može se dodeliti program P_n (kao što je opisano u poglavlju 3.4). Imajući ovo u vidu, dolazi se do teoreme o halting problemu za URM.

Teorema 3.2 (Neizračunljivost halting problema)**.** *Neka je funkcija h definisana na sledeći način:*

$$h(x,y) = \begin{cases} 1, & \textit{ako se program } P_x \textit{ zaustavlja za ulaz } y \\ 0, & \textit{inače.} \end{cases}$$

Ne postoji program koji izračunava funkciju h, tj. ne postoji program koji za zadate vrednosti x i y može da proveri da li se program P_x zaustavlja za ulazni argument y.

Dokaz: Pretpostavimo da postoji program H koji izračunava funkciju h. Onda se jednostavno može konstruisati i program H' sa jednim argumentom x, koji vraća rezultat 1 (tj. upisuje ga u prvi registar) ako se program P_x zaustavlja za ulaz x, a rezultat 0 ako se program P_x ne zaustavlja za ulaz x. Dalje, postoji i program Q (dobijen kombinovanjem programa H' i programa iz primera 3.4) koji za argument x vraća rezultat 0 ako se P_x ne zaustavlja za x (tj. ako je $h(x,x) = 0$), a izvršava beskonačnu petlju ako se P_x zaustavlja za x (tj. ako je $h(x,x) = 1$). Za program Q važi:

$$Q(x) \downarrow 0 \text{ ako je } P_x(x) \uparrow$$
$$Q(x) \uparrow \text{ ako je } P_x(x) \downarrow$$

Ako postoji takav program Q, onda se i on nalazi u nizu svih programa tj. postoji redni broj k koji ga jedinstveno identifikuje, pa važi:

$$P_k(x) \downarrow 0 \text{ ako je } P_x(x) \uparrow$$
$$P_k(x) \uparrow \text{ ako je } P_x(x) \downarrow$$

No, ako je x jednako upravo k, pokazuje se da je definicija ponašanja programa Q (tj. programa P_k) kontradiktorna: program Q (tj. program P_k) za ulaznu vrednost k vraća 0 ako se P_k ne zaustavlja za k, a izvršava beskonačnu petlju ako se P_k zaustavlja za k:

$$P_k(k) \downarrow 0 \text{ ako je } P_k(k) \uparrow$$
$$P_k(k) \uparrow \text{ ako je } P_k(k) \downarrow$$

Dakle, polazna pretpostavka je bila pogrešna i ne postoji program H, tj. funkcija h nije izračunljiva. Pošto funkcija h, karakteristična funkcija halting problema, nije izračunljiva, halting problem nije odlučiv. □

Funkcija koja odgovara halting problemu je jedna od najznačajnih funkcija iz skupa prirodnih brojeva u skup prirodnih brojeva koje ne mogu biti izračunate, ali takvih, neizračunljivih funkcija, ima još mnogo. Može se dokazati da funkcija jedne promenljive koje za ulazni prirodni broj vraćaju isključivo 0 ili 1 (tj. funkcija iz N u $\{0,1\}$) ima neprebrojivo mnogo, dok programa ima samo prebrojivo mnogo. Iz toga direktno sledi da ima neprebrojivo mnogo funkcija koje nisu izračunljive.

3.6 Vremenska i prostorna složenost izračunavanja

Prvo pitanje koje se postavlja kada je potrebno izračunati neku funkciju (tj. napisati neki program) je da li je ta funkcija izračunljiva (tj. da li uopšte postoji neki program koji je izračunava). Ukoliko takav program postoji, sledeće pitanje je koliko izvršavanje tog program zahteva vremena i prostora (memorije). U kontekstu URM programa, pitanje je koliko za neki URM program treba izvršiti pojedinačnih instrukcija (uključujući ponavljanja) i koliko registara se koristi. U URM programu navedenom u primeru 3.1 postoje ukupno četiri instrukcije, ali se one mogu ponavljati (za neke ulazne vrednosti). Jednostavno se može izračunati da se prva instrukcija u nizu izvršava $y + 1$ puta, a preostale tri po y puta. Dakle, ukupan broj izvršenih instrukcija za ulazne vrednosti x i y je jednak $4y + 1$. Ako se razmatra samo takozvani red algoritma, onda se zanemaruju konstante kojima se množe i sabiraju vrednosti ulaznih argumenata, pa je vremenska složenost u ovom primeru linearna po drugom argumentu, argumentu y (i to se zapisuje $O(y)$). Bez obzira na vrednosti ulaznih argumenata, program koristi tri registra, pa je njegova prostorna složenost konstantna (i to se zapisuje $O(1)$). Najčešće se složenost algoritma određuje tako da ukazuje na to koliko on može utrošiti vremena i prostora u najgorem slučaju. Ponekad je moguće izračunati i prosečnu složenost algoritma — prosečnu za sve moguće vrednosti argumenata. O prostornoj i vremenoskoj složenosti algoritama biće više reči u drugom tomu ove knjige.

Pitanja i zadaci za vežbu

Pitanje 3.1. *Po kome je termin* algoritam *dobio ime?*

Pitanje 3.2. *Šta je to algoritam (formalno i neformalno)? Navesti nekoliko formalizma za opisivanje algoritama. Kakva je veza između formalnog i neformalnog pojma algoritma. Šta tvrdi Čerč-Tjuringova teza? Da li se ona može dokazati?*

Pitanje 3.3. *Da li postoji algoritam koji opisuje neku funkciju iz skupa prirodnih brojeva u skup prirodnih brojeva i koji može da se isprogramira u programskom jeziku C i izvrši na savremenom računaru, a ne može na Tjuringovoj mašini?*

Pitanje 3.4. *Da li je svaka* URM *izračunljiva funkcija intuitivno izračunljiva? Da li je svaka intuitivno izračunljiva funkcija* URM *izračunljiva?*

Pitanje 3.5. *U čemu je ključna razlika između* URM *mašine i bilo kog stvarnog računara?*

Pitanje 3.6. *Opisati efekat* URM *naredbe* $J(m, n, p)$.

Pitanje 3.7. *Da li se nekim* URM *programom može izračunati hiljadita cifra broja* 2^{1000} *?*

Pitanje 3.8. *Da li postoji* URM *program koji izračunava broj* $\sqrt{2}$*? Da li postoji* URM *program koji izračunava n-tu decimalnu cifru broja* $\sqrt{2}$*, gde je n zadati prirodan broj?*

Pitanje 3.9. *Da li se nekim* URM *programom može izračunati hiljadita decimalna cifra broja* π *?*

Pitanje 3.10. *Koliko ima racionalnih brojeva? Koliko ima kompleksnih brojeva? Koliko ima različitih programa za Tjuringovu mašinu? Koliko ima različitih programa u programskom jeziku C?*

Pitanje 3.11. *Koliko elemenata ima unija konačno mnogo konačnih skupova? Koliko elemenata ima unija prebrojivo mnogo konačnih skupova? Koliko elemenata ima unija konačno mnogo prebrojivih skupova? Koliko elemenata ima unija prebrojivo mnogo prebrojivih skupova?*

Pitanje 3.12. *Koliko ima različitih* URM *programa? Kakva je kardinalnost skupa* URM *programa u odnosu na kardinalnost skupa prirodnih brojeva? Kakva je kardinalnost skupa* URM *programa u odnosu na kardinalnost skupa realnih brojeva? Kakva je kardinalnost skupa* URM *programa u odnosu na kardinalnost skupa programa na jeziku C?*

Pitanje 3.13. *Da li se svakom* URM *programu može pridružiti jedinstven prirodan broj (različit za svaki program)? Da li se svakom prirodnom broju može pridružiti jedinstven* URM *program (različit za svaki broj)?*

Pitanje 3.14. *Da li se svakom* URM *programu može pridružiti jedinstven realan broj (različit za svaki program)? Da li se svakom realnom broju može pridružiti jedinstven* URM *program (različit za svaki broj)?*

Pitanje 3.15. *Kako se naziva problem ispitivanja zaustavljanja programa? Kako glasi* halting *problem? Da li je on odlučiv ili nije? Ko je to dokazao?*

Pitanje 3.16.

1. *Da li postoji algoritam koji za drugi zadati* URM *program utvrđuje da li se zaustavlja ili ne?*
2. *Da li postoji algoritam koji za drugi zadati* URM *utvrđuje da li se zaustavlja posle 100 koraka?*
3. *Da li je moguće napisati* URM *program koji za drugi zadati* URM *program proverava da li radi beskonačno?*
4. *Da li je moguće napisati* URM *program koji za drugi zadati* URM *program proverava da li vraća vrednost 1?*
5. *Da li je moguće napisati* URM *program kojim se ispituje da li data izračunljiva funkcija (ona za koju postoji* URM *program)* f *zadovoljava da je* $f(0) = 0$*?*

6. *Da li je moguće napisati* URM *program koji za drugi zadati* URM *program ispituje da li izračunava vrednost* 2012 *i zašto?*

Pitanje 3.17. *Na primeru korena uporedite URM sa savremenim asemberlskim jezicima. Da li URM ima neke prednosti?*

Zadatak 3.1. *Napisati* URM *program koji izračunava funkciju* $f(x,y) = xy$.
√

Zadatak 3.2. *Napisati* URM *program koji izračunava funkciju* $f(x) = 2^x$.

Zadatak 3.3. *Napisati* URM *program koji izračunava funkciju* $f(x,y) = x^y$.

Zadatak 3.4. *Napisati* URM *program koji izračunava funkciju:*

$$f(x,y) = \begin{cases} 0 & , \textit{ako } x \geq y \\ 1 & , \textit{inače} \end{cases}$$

Zadatak 3.5. *Napisati* URM *program koji izračunava funkciju*

$$f(x,y) = \begin{cases} x - y & , \textit{ako } x \geq y \\ 0 & , \textit{inače} \end{cases}$$

√

Zadatak 3.6. *Napisati* URM *program koji izračunava funkciju:*

$$f(x) = \begin{cases} x/3 & , \textit{ako } 3|x \\ nedefinisano & , \textit{inače} \end{cases}$$

Zadatak 3.7. *Napisati* URM *program koji izračunava funkciju* $f(x) = x!$.

Zadatak 3.8. *Napisati* URM *program koji izračunava funkciju* $f(x) = \left[\frac{2x}{3}\right]$.

Zadatak 3.9. *Napisati* URM *program koji broj* 1331 *smešta u prvi registar.*

Zadatak 3.10. *Napisati* URM *program koji izračunava funkciju* $f(x) = 1000 \cdot x$.

Zadatak 3.11. *Napisati* URM *program koji izračunava funkciju* $f(x,y) = 2x + y$.

Zadatak 3.12. *Napisati* URM *program koji izračunava funkciju* $f(x,y) = min(x,y)$, *odnosno:*

$$f(x,y) = \begin{cases} x & , \textit{ako } x \leq y \\ y & , \textit{inače} \end{cases}$$

Zadatak 3.13. *Napisati* URM *program koji izračunava funkciju* $f(x,y) = 2^{(x+y)}$

Zadatak 3.14. *Napisati* URM *program koji izračunava funkciju*

$$f(x,y) = \begin{cases} 1 & , \textit{ako } x|y \\ 0 & , \textit{inače} \end{cases}$$

Zadatak 3.15. *Napisati* URM *program koji izračunava funkciju*

$$f(x,y) = \begin{cases} \left[\frac{y}{x}\right] & , \textit{ako } x \neq 0 \\ nedefinisano & , \textit{inače} \end{cases}$$

Zadatak 3.16. *Napisati* URM *program koji izračunavaju sledeću funkciju:*

$$f(x,y) = \begin{cases} 2x & ,x < y \\ x - y & ,x \geq y \end{cases}$$

Zadatak 3.17. *Napisati* URM *program koji izračunava funkciju:*

$$f(x,y) = \begin{cases} x/3 & ,3|x \\ y^2 & ,inace \end{cases}$$

Zadatak 3.18. *Napisati* URM *program koji izracunava funkciju* $f(x,y,z) = x + y + z$

Zadatak 3.19. *Napisati* URM *program koji izracunava funkciju* $f(x,y,z) = min(x,y,z)$.

Zadatak 3.20. *Napisati* URM *program koji izračunava funkciju*

$$f(x,y,z) = \begin{cases} \left[\frac{y}{3}\right] & , \text{ ako } 2|z \\ x+1 & , \text{ inače} \end{cases}$$

Zadatak 3.21. *Napisati* URM *program koji izračunava funkciju*

$$f(x,y,z) = \begin{cases} 1, & \text{ako je } x+y > z \\ 2, & \text{inače} \end{cases}$$

Glava 4

Viši programski jezici

Razvoj programskih jezika, u bliskoj je vezi sa razvojem računara tj. sa razvojem hardvera. Programiranje u današnjem smislu nastalo je sa pojavom računara fon Nojmanovog tipa čiji se rad kontroliše programima koji su smešteni u memoriji, zajedno sa podacima nad kojim operišu. Na prvim računarima tog tipa moglo je da se programira samo na mašinski zavisnim programskim jezicima, a od polovine 1950-ih nastali su jezici višeg nivoa koji su drastično olakšali programiranje.

Prvi programski jezici zahtevali su od programera da bude upoznat sa najfinijim detaljima računara koji se programira. Problemi ovakvog načina programiranja su višestruki. Naime, ukoliko je želeo da programira na novom računaru, programer je morao da izuči sve detalje njegove arhitekture (na primer, skup instrukcija procesora, broj registara, organizaciju memorije). Programi napisani za jedan računar mogli su da se izvršavaju isključivo na istim takvim računarima i prenosivost programa nije bila moguća.

Viši programski jezici namenjeni su ljudima a ne mašinama i sakrivaju detalje konkretnih računara od programera. Specijalizovani programi (tzv. *jezički procesori*, *programski prevodioci*, *kompilatori* ili *interpretatori*) na osnovu specifikacije zadate na višem (apstraktnijem) nivou mogu automatski da proizvedu mašinski kôd za specifičan računar na kojem se programi izvršavaju. Ovim se omogućava prenosivost programa (pri čemu, da bi se program mogao izvršavati na nekom konkretnom računaru, potrebno je da postoji procesor višeg programskog jezika baš za taj računar). Takođe, znatno se povećava nivo apstrakcije prilikom procesa programiranja što u mnogome olakšava ovaj proces.

Razvojni ciklus programa u većini savremenih viših programskih jezika teče na sledeći način. Danas je, nakon faze planiranja, prva faza u razvoju programa njegovo *pisanje* tj. unošenje programa na višem programskom jeziku (tzv. *izvorni program* ili *izvorni kôd* – engl. source code), što se radi pomoću nekog editora teksta. Naredna faza je njegovo *prevođenje*, kada se na osnovu izvornog programa na višem programskom jeziku dobija prevedeni kôd na asemblerskom odnosno mašinskom jeziku (tzv. *objektni kôd* – engl. object code), što se radi pomoću nekog programskog prevodioca. U fazi *povezivanja* više objektnih programa povezuje se sa objektnim kôdom iz standardne biblioteke u jedinstvenu celinu (tzv. izvršni program – engl. executable program). Povezivanje vrši specijalizovan program *povezilac* (engl. linker). Nakon povezivanja,

kreiran je program u *izvršnom obliku* i on možse da se *izvršava*. Nabrojane faze se obično ponavljaju, vrši se dopuna programa, ispravljanje grešaka, itd.

4.1 Kratki pregled istorije programskih jezika

Prvim višim programskim jezikom najčešće se smatra jezik *FORTRAN*, nastao u periodu 1953-1957 godine. Njegov glavni autor je Džon Bakus[1], koji je implementirao i prvi interpretator i prvi kompilator za ovaj jezik. Ime FORTRAN dolazi od *The IBM Mathematical FORmula TRANslating System*, što ukazuje na to da je osnovna motivacija bila da se u okviru naučno-tehničkih primena omogući unošenje matematičkih formula, dok je sistem taj koji bi unete matematičke formule prevodio u niz instrukcija koje računar može da izvršava. Neposredno nakon pojave Fortrana, Džon Mekarti[2] je dizajnirao programski jezik *LISP* (*LISt Processing*), zasnovan na λ-računu. LISP je uveo funkcionalno programiranje i dugo bio najpopularniji jezik u oblasti veštačke inteligencije. Krajem 1950-ih godina, nastao je i jezik COBOL (*COmmon Business-Oriented Language*), čije su osnovne primene u oblasti poslovanja. Zanimljivo je da puno starih COBOL programa i danas uspešno radi u velikim poslovnim sistemima kod nas i u svetu.

Rani programski jezici, nastali pod uticajem asemblerskih jezika, intenzivno koriste skokove u programima (tzv. GOTO naredbu) što dovodi do programa koje je teško razumeti i održavati. Kao odgovor na softversku krizu 1970-ih godina (period kada zbog loše prakse programiranja softver nije mogao da dostigne mogućnosti hardvera), nastala je praksa *strukturnog programiranja* u kojoj se insistira na disciplinovanom pristupu programiranju, bez nekontrolisanih skokova i uz korišćenje samo malog broja naredbi za kontrolu toka programa.

Krajem 1950-ih godina započet je razvoj programskog jezika *ALGOL 60* koji je uneo mnoge koncepte prisutne skoro u svim današnjim programskim jezicima. Tokom 1970-ih pojavio se jezik *C* koji i dalje predstavlja osnovni jezik sistemskog programiranja. Tokom 1970-ih pojavio se i jezik *Pascal* koji je vrlo elegantan jezik čiji je cilj bio da ohrabri strukturno programiranje i koji se zbog ovoga (u svojim unapređenim oblicima) i danas ponegde koristi u nastavi programiranja.

Kao odgovor na još jednu softversku krizu prelazi se na korišćenje tzv. objektno-orijentisanih jezika koji olakšavaju izradu velikih programa i podelu posla u velikim programerskim timovima. Tokom 1980-ih se pojavljuje jezik *C++* koji nadograđuje jezik C objektno-orijentisanim konceptima, a tokom 1990-ih, pod uticajem interneta, i jezik *Java*, čija je jedna od osnovnih ideja prenosivost izvršnog kôda između heterogenih računarskih sistema. Microsoft, je krajem 1990-ih započeo razvoj jezika *C#* koji se danas često koristi za programiranje Windows aplikacija.

Pojavom veba postaju značajni skript jezici, kao što su *PHP*, *JavaScript*, *Python*, *Ruby* itd.

[1] John Backus (1924–2007), američki naučnik iz oblasti računarstva, dobitnik Tjuringove nagrade.

[2] John McCarthy (1927–2011), američki naučnik iz oblasti računarstva. Dobitnik Tjuringove nagrade za svoj rad na polju veštačke inteligencije.

4.2 Klasifikacije programskih jezika

Po načinu programiranja, programski jezici se klasifikuju u *programske paradigme*.

Najkorišćeniji programski jezici danas spadaju u grupu *imperativnih* programskih jezika. Kako u ovu grupu spada i programski jezik C, u nastavku će biti najviše reči upravo o ovakvim jezicima. U ovim jezicima stanje programa karakterišu *promenljive* kojima se predstavljaju podaci i *naredbe* kojima se vrše određene transformacije promenljivih. Pored imperativne, značajne programske paradigme su i *objektno-orijentisana*, *funkcionalna*, *logička*. U savremenim jezicima mešaju se karakteristike različitih programskih paradigmi tako da je podela sve manje striktna.

Većina programskih jezika danas je *proceduralna* što znači da je zadatak programera da opiše način (proceduru) kojim se dolazi do rešenja problema. Nasuprot njima, *deklarativni* programski jezici od programera zahtevaju da precizno opiše problem, dok se mehanizam programskog jezika onda bavi pronalaženjem rešenja problema. Iako uspešni u ograničenim domenima, deklarativni jezici još nisu dovoljno efikasni da bi bili dominantno korišćeni.

4.3 Leksika, sintaksa, semantika programskih jezika

Kako bi bilo moguće pisanje i prevođenje programa u odgovarajuće programe na mašinskom jeziku nekog konkretnog računara, neophodno je precizno definisati šta su ispravni programi nekog programskog jezika, kao i precizno definisati koja izračunavanja odgovaraju naredbama programskog jezika. Pitanjima ispravnosti programa bavi se *sintaksa programskih jezika* (i njena podoblast *leksika programskih jezika*). Leksika se bavi opisivanjem osnovnih gradivnih elemenata jezika, a sintaksa načinima za kombinovanje tih osnovnih elemenata u ispravne jezičke konstrukcije. Pitanjem značenja programa bavi se *semantika programskih jezika*. Leksika, sintaksa i semantika se izučavaju ne samo za programske jezike, već i za druge veštačke jezike, ali i za prirodne jezike.

Leksika. Osnovni leksički elementi prirodnih jezika su reči, pri čemu se razlikuje nekoliko različitih vrsta reči (imenice, glagoli, pridevi, ...) i reči imaju različite oblike (padeži, vremena, ...). Zadatak leksičke analize prirodnog jezika je da identifikuje reči u rečenici i svrsta ih u odgovarajuće kategorije. Slično važi i za programske jezike. Programi se računaru zadaju predstavljeni nizom karaktera. Pojedinačni karakteri se grupišu u nedeljive celine koje predstavljaju osnovne leksičke elemente, koji bi bili analogni rečima govornog jezika. Na primer, leksika jezika UR mašina razlikuje rezervisane reči (Z, S, J, T) i brojevne konstante. Razmotrimo naredni fragment kôda u jeziku C:

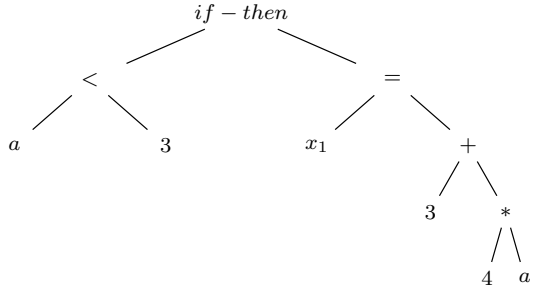
```
if (a < 3)
   x1 = 3+4*a;
```

U ovom kôdu, razlikuju se sledeće *lekseme* (reči) i njima pridruženi *tokeni* (kategorije).

`if`	ključna reč
`(`	zagrada
`a`	identifikator
`<`	operator
`3`	celobrojni literal
`)`	zagrada
`x1`	identifikator
`=`	operator
`3`	celobrojni literal
`+`	operator
`4`	celobrojni literal
`*`	operator
`a`	celobrojni literal
`;`	punktuator

Leksikom programa obično se bavi deo programskog prevodioca koji se naziva *leksički analizator*.

Sintaksa. Sintaksa prirodnih jezika definiše načine na koji pojedinačne reči mogu da kreiraju ispravne rečenice jezika. Slično je i sa programskim jezicima, gde se umesto ispravnih rečenica razmatraju ispravni programi. Na primer, sintaksa jezika UR mašina definiše ispravne programe kao nizove instrukcija oblika: $Z(\text{broj})$, $S(\text{broj})$, $J(\text{broj}, \text{broj}, \text{broj})$ i $T(\text{broj}, \text{broj})$. Sintaksa definiše formalne relacije između elemenata jezika, time pružajući strukturne opise ispravnih niski jezika. Sintaksa se bavi samo formom i strukturom jezika bez bilo kakvih razmatranja u vezi sa njihovim značenjem. Sintaksička struktura rečenica ili programa se može predstaviti u obliku stabla. Prikazani fragment kôda je u jeziku C sintaksički ispravan i njegova sintaksička struktura se može predstaviti na sledeći način:



Dakle, taj fragment kôda predstavlja `if-then` naredbu (iskaz) kojoj je uslov dat izrazom poređenja vrednosti promenljive `a` i konstante `3`, a telo predstavlja naredba dodele promenljivoj `x` vrednosti izraza koji predstavlja zbir konstante `3` i proizvoda konstante `4` i vrednosti promenljive `a`.

Sintaksom programa obično se bavi deo programskog prevodioca koji se naziva *sintaksički analizator*.

Semantika. Semantika pridružuje značenje sintaksički ispravnim niskama jezika. Za prirodne jezike, semantika pridružuje ispravnim rečenicama neke specifične objekte, misli i osećanja. Za programske jezike, semantika za dati program opisuje koje je izračunavanje opisano tim programom.

Tako se, na primer, naredbi `if (a < 3) x1 = 3+4*a;` jezika C može pridružiti sledeće značenje: „Ako je tekuća vrednost promenljive `a` manja od 3, tada promenljiva `x1` treba da dobije vrednost zbira broja 3 i četvorostruke tekuće vrednosti promenljive `a`“.

Postoje sintaksički ispravne rečenice prirodnog jezika kojima nije moguće dodeliti ispravno značenje, tj. za njih nije definisana semantika, na primer: „Bezbojne zelene ideje besno spavaju“ ili „Pera je oženjeni neženja“. Slično je i sa programskim jezicima. Neki aspekti semantičke korektnosti programa se mogu proveriti tokom prevođenja programa (na primer, da su sve promenljive koje se koriste u izrazima definisane i da su odgovarajućeg tipa), dok se neki aspekti mogu proveriti tek u fazi izvršavanja programa (na primer, da ne dolazi do deljenja nulom). Na osnovu toga, razlikuje se statička i dinamička semantika. Naredni C kôd nema jasno definisano dinamičko značenje, pa u fazi izvršavanja dolazi do greške (iako se prevođenje na mašinski jezik uspešno izvršava).

```
int x = 0;
int y = 1/x;
```

Dok većina savremenih jezika ima precizno i formalno definisanu leksiku i sintaksu, formalna definicija semantike postoji samo za neke programske jezike[3]. U ostalim slučajevima, semantika programskog jezika se opisuje neformalno, opisima zadatim korišćenjem prirodnog jezika. Čest je slučaj da neki aspekti semantike ostaju nedefinisani standardom jezika i prepušta se implementacijama kompilatora da samostalno odrede potpunu semantiku. Tako, na primer, programski jezik C ne definiše kojim se redom vrši izračunavanje vrednosti izraza, što u nekim slučajevima može da dovede do različitih rezultata istog programa prilikom prevođenja i izvršavanja na različitim sistemima (na primer, nije definisano da li se za izračunavanje vrednosti izraza `f() + g()` najpre poziva funkcija `f` ili funkcija `g`).

4.4 Pragmatika programskih jezika

Pragmatika jezika govori o izražajnosti jezika i o odnosu različitih načina za iskazivanje istih stvari. Pragmatika prirodnih jezika se odnosi na psihološke i sociološke aspekte kao što su korisnost, opseg primena i efekti na korisnike. Isti prirodni jezik se koristi drugačije, na primer, u pisanju tehničkih uputstava, a drugačije u pisanju pesama. Pragmatika programskih jezika uključuje pitanja kao što su lakoća programiranja, efikasnost u primenama i metodologija programiranja. Pragmatika je ključni predmet interesovanja onih koji dizajniraju i implementiraju programske jezike, ali i svih koji ih koriste. U pitanja pragmatike može da spada i izbor načina na koji napisati neki program, u zavisnosti od toga da li je, u konkretnom slučaju, važno da je program kratak ili da je jednostavan za razumevanje ili da je efikasan.

Pragmatika jezika se, između ostalog, bavi i sledećim pitanjima dizajna programskih jezika.

[3] Leksika se obično opisuje *regularnim izrazima*, sintaksa *kontekstno slobodnim gramatikama*, dok se semantika formalno zadaje ili aksiomatski (npr. u obliku Horove logike) ili u vidu denotacione semantike ili u vidu operacione semantike.

Promenljive i tipovi podataka. U fon Nojmanovom modelu, podaci se smeštaju u memoriju računara (najčešće predstavljeni nizom bitova). Međutim, programski jezici uobičajeno, kroz koncept *promenljivih*, nude programerima apstraktniji pogled na podatke. Promenljive omogućavaju programeru da imenuje podatke i da im pristupa na osnovu imena, a ne na osnovu memorijskih adresa (kao što je to slučaj kod asemblerskih jezika). Svakoj promenljivoj dodeljen je određen broj bajtova u memoriji (ili, eventualno, u registrima procesora) kojima se predstavljaju odgovarajući podaci. Pravila *životnog veka* (engl. lifetime) određuju u kom delu faze izvršavanja programa je promenljivoj dodeljen memorijski prostor (promenljivu je moguće koristiti samo tada). Nekada je moguće na različitim mestima u programu koristiti različite promenljive istog imena i pravila *dosega indentifikatora* (engl. scope) određuju deo programa u kome se uvedeno ime može koristiti za imenovanje određenog objekta (najčešće promenljive).

Organizovanje podataka u *tipove* omogućava da programer ne mora da razmišlja o podacima na nivou njihove interne (binarne) reprezentacije, već da o podacima može razmišljati znatno apstraktnije, dok se detalji interne reprezentacije prepuštaju jezičkom procesoru. Najčešći tipovi podataka direktno podržani programskim jezicima su celi brojevi (0, 1, 2, -1, -2, ...), brojevi u pokretnom zarezu (1.0, 3.14, ...), karakteri (`a`, `b`, `0`, `,`, `!`, ...), niske (`"zdravo"`), ... Pored ovoga, programski jezici obično nude i mogućnost korišćenja složenih tipova (na primer, nizovi, strukture tj. slogovi koji mogu da objedinjavaju nekoliko promenjivih istog ili različitog tipa).

Svaki tip podataka karakteriše:

- vrsta podataka koje opisuje (na primer, celi brojevi),
- skup operacija koje se mogu primeniti nad podacima tog tipa (na primer, sabiranje, oduzimanje, množenje, ...),
- način reprezentacije i detalji implementacije (na primer, zapis u obliku binarnog potpunog komplementa širine 8 bita, odakle sledi opseg vrednosti od -128 do 127).

Naredni fragment C kôda obezbeđuje da su promenljive `x`, `y` i `z` celobrojnog tipa (tipa `int`), a da `z` nakon izvršavanja ovog fragmenta ima vrednost jednaku zbiru promenljivih `x` i `y`.

```
int x, y;
...
int z = x + y;
```

Prilikom prevođenja i kreiranja mašinskog kôda, prevodilac, vođen tipom promenljivih `x`, `y` i `z`, dodeljuje određeni broj bitova u memoriji ovim promenljivim (tj. rezerviše određene memorijske lokacije isključivo za smeštanje vrednosti ovih promenljivih) i generiše kôd (procesorske instrukcije) kojim se tokom izvršavanja programa sabiraju vrednosti smeštene na dodeljenim memorijskim lokacijama. Pri tom se vodi računa o načinu reprezentacije koji se koristi za zapis. S obzirom da je naglašeno da su promenljive `x`, `y` i `z`, celobrojnog tipa, prevodilac će najverovatnije podrazumevati zapis u potpunom komplementu i

operacija `+` će se prevesti u mašinsku instrukciju kojom se sabiraju brojevi zapisani u potpunom komplementu (koja je obično direktno podržana u svakom procesoru). U zavisnosti od tipa operanada, ista operacija se ponekad prevodi na različite načine. Tako, da su promenljive bile tipa `float`, najverovatnije bi se podrazumevao zapis u obliku pokretnog zareza i operacija `+` bi se prevela u mašinsku instrukciju kojom se sabiraju brojevi zapisani u pokretnom zarezu. Time, tipovi podataka iz jezika višeg nivoa u odgovarajućem mašinskom kôdu postoje samo implicitno, najčešće samo kroz mašinske instrukcije nastale nakon prevođenja.

Tokom prevođenja programa se razrešavaju i pitanja *konverzija tipova*. Na primer, u C kôdu `int x = 3.1;` promenljiva `x` je deklarisana kao celobrojna, pa se za nju odvaja određeni broj bitova u memoriji i ta memorija se inicijalizuje binarnim zapisom broja `3`. Naime, realnu vrednost `3.1` nije moguće zapisati kao celobrojnu promenljivu, pa se vrši konverzija tipa i zaokruživanje vrednosti.

Statički tipizirani jezici (uključujući C), zahtevaju da programer definiše tip svake promenljive koja se koristi u programu i da se taj tip ne menja tokom izvršavanja programa. Međutim, neki statički tipizirani programski jezici ne zahtevaju od programera definisanje tipova, već se tipovi automatski zaključuju iz teksta programa. S druge strane, u *dinamički tipiziranim jezicima* ista promenljiva može da sadrži podatke različitog tipa tokom raznih faza izvršavanja programa. U nekim slučajevima dopušteno je vršiti operacije nad promenljivima različitog tipa, pri čemu se tip implicitno konvertuje. Na primer, jezik JavaScript ne zahteva definisanje tipa promenljivih i dopušta kôd poput `a = 1; b = "2"; a = a + b;`.

Kontrola toka izvršavanja. Osnovi gradivni element imperativnih programa su *naredbe*. Osnovna naredba je *naredba dodele* kojom se vrednost neke promenljive postavlja na vrednost nekog *izraza* definisanog nad konstantama i definisanim promenljivim. Na primer, `x1 = 3 + 4*a;`.

Naredbe se u programu često nižu i izvršavaju sekvencijalno, jedna nakon druge. Međutim, kako bi se postigla veća izražajnost, programski jezici uvode specijalne vrste naredbi kojima se vrši kontrola toka izvršavanja programa (tzv. kontrolne strukture). Ovim se postiže da se u zavisnosti od tekućih vrednosti promenljivih neke naredbe uopšte ne izvršavaju, neke izvršavaju više puta i slično. Najčešće korišćene kontrolne strukture su granajuće naredbe (`if-then-else`), petlje (`for`, `while`, `do-while`, `repeat-until`) i naredbe skoka (`goto`). Za naredbu skoka (`goto`) je pokazano da nije neophodna, tj. svaki program se može zameniti programom koji daje iste rezultate a pri tome koristi samo sekvencijalno nizanje naredbi, naredbu izbora (`if-then-else`) i jednu vrstu petlje (na primer, `do-while`).[4] Ovaj važan teorijski rezultat ima svoju punu praktičnu primenu i u današnjem, strukturnom, programiranju naredba skoka se gotovo uopšte ne koristi jer dovodi do nerazumljivih (tzv. špageti) programa.

Potprogrami. Većina programskih jezika pruža mogućnost definisanja neke vrste potprograma, ali nazivi potprograma se razlikuju (najčešće se koriste termini *funkcije*, *procedure*, *sabrutine* ili *metode*). Potprogrami izoluju određena

[4] Ovo tvrđenje je poznato kao *teorema o strukturnom programiranju*, Korado Bema (nem. Corrado Böhm) i Đuzepea Jakopinija (it. Giuseppe Jacopini) iz 1966.

izračunavanja koja se kasnije mogu pozivati tj. koristiti na više različitih mesta u različitim kontekstima. Tako je, na primer, u jeziku C moguće definisati funkciju kojom se izračunava najveći zajednički delilac (NZD) dva broja, a kasnije tu funkciju iskoristiti na nekoliko mesta kako bi se izračunao NZD različitih parova brojeva.

```
int nzd(int a, int b) { ... }
...
x = nzd(1234, 5678);
y = nzd(8765, 4321);
```

Primetimo da su potprogrami obično parametrizovani tj. mogu da primaju ulazne *parametre* (kaže se i *argumente*). Postoje različiti načini prenosa parametara u potprograme. Na primer, u nekim slučajevima (tzv. *prenos po vrednosti*) funkcija dobija svoju kopiju parametra navedenog u pozivu i sve vreme barata kopijom, ostavljajući originalni parametar nepromenjen. U nekim slučajevima (tzv. *prenos po adresi*), parametar se ne kopira već se u funkciju prenosi samo memorijska adresa na kojoj se parametar nalazi i funkcija sve vreme barata originalnim parametrom.

Potprogrami imaju i mogućnost vraćanja vrednosti izračunavanja pozivaocu. Neki jezici (npr. Pascal) suštinski razlikuju funkcije koje izračunavaju (i kaže se *vraćaju*) neku vrednost i procedure čiji je zadatak samo da proizvedu određeni sporedni efekat (npr. da ispišu nešto na ekran ili da promene vrednost promenljive).

Modularnost. Modularnost podrazumeva razbijanje većeg programa na nezavisne celine. Celine, koje sadrže definicije srodnih podataka i funkcija, obično se nazivaju *biblioteke* (engl. library) i smeštaju se u posebne datoteke. Ovim se postiže lakše održavanje kompleksnih sistema, kao i mogućnost višestruke upotrebe pojedinih modula u okviru različitih programa. Celine se obično zasebno prevode i kasnije povezuju u jedinstven program.

Programski jezici obično imaju svoje *standardne biblioteke* (engl. standard library) koje sadrže funkcije često potrebne programeru (npr. funkciju za izračunavanje dužine niske karaktera). Često se kôd funkcija standardne biblioteke statički povezuju sa programom (tj. uključuje se u izvršni kôd programa).

Osim standardne biblioteke, programske jezike često karakteriše i *rantajm biblioteka* (engl. runtime library) koja predstavlja sponu između kompilatora (tj. izvršnih programa koje on generiše) i operativnog sistema. Funkcije rantajm biblioteke se ne uključuju u izvršni kôd programa već se dinamički pozivaju tokom izvršavanja programa (i zato, da bi program mogao da se izvrši na nekom operativnom sistemu, neophodno je da na njemu bude instalirana rantajm biblioteka za taj programski jezik).

Upravljanje memorijom. Neki programski jezici zahtevaju od programera da eksplicitno rukuje memorijom tj. da od sistema zahteva memoriju u trenutku kada je ona potrebna i da tu memoriju eksplicitno oslobađa, tj. vraća sistemu kada ona programu više nije potrebna. Drugi programski jezici oslobađaju programera ove dužnosti time što koriste tzv. *sakupljače otpada* (engl. garbage collector) čiji je zadatak da detektuju memoriju koju program ne koristi i da je oslobađaju. Iako je programiranje u jezicima sa sakljupljačima otpada

jednostavnije, programi su obično sporiji (jer određeno vreme odlazi na rad sakupljača otpada).

U cilju obezbeđivanja pogodnog načina da se hardverom upravlja, neki programski jezici dopuštaju programeru da pristupi proizvoljnoj memorijskoj adresi (npr. u jeziku C, to se može raditi korišćenjem pokazivača), čime se dobija veća sloboda, ali i mnogo veća mogućnost pravljenja grešaka. Sa druge strane, neki programski jezici imaju za cilj skrivanje svojstava hardvera od programera, štite memoriju od direktnog pristupa programera i dopuštaju korišćenje podataka samo u okviru memorije zauzete promenljivim programa.

4.4.1 Jezički procesori

Jezički procesori (ili programski prevodioci) su programi čija je uloga da analiziraju leksičku, sintaksičku i (donekle) semantičku ispravnost programa višeg programskog jezika i da na osnovu ispravnog ulaznog programa višeg programskog jezika generišu kôd na mašinskom jeziku (koji odgovara polaznom programu, tj. vrši izračunavanje koje je opisano na višem programskom jeziku). Kako bi konstrukcija jezičkih procesora uopšte bila moguća, potrebno je imati precizan opis kako leksike i sintakse, tako i semantike višeg programskog jezika.

U zavisnosti od toga da li se ceo program analizira i transformiše u mašinski kôd pre nego što može da se izvrši, ili se analiza i izvršavanje programa obavljaju naizmenično deo po deo programa (na primer, naredba po naredba), jezički procesori se dele u dve grupe: *kompilatore* i *interpretatore*.

Kompilatori. Kompilatori (ili kompajleri) su programski prevodioci kod kojih su faza prevođenja i faza izvršavanja programa potpuno razdvojene. Nakon analize *izvornog kôda* programa višeg programskog jezika, kompilatori generišu *izvršni (mašinski) kôd* i dodatno ga optimizuju, a zatim čuvaju u vidu *izvršnih (binarnih) datoteka*. Jednom sačuvani mašinski kôd moguće je izvršavati neograničen broj puta, bez potrebe za ponovnim prevođenjem. Krajnjim korisnicima nije neophodno dostavljati izvorni kôd programa na višem programskom jeziku, već je dovoljno distribuirati izvršni mašinski kôd[5]. Jedan od problema u radu sa kompilatorima je da se prevođenjem gubi svaka veza između izvršnog i izvornog kôda programa. Svaka (i najmanja) izmena u izvornom kôdu programa zahteva ponovno prevođenje programa ili njegovih delova. S druge strane, kompilirani programi su obično veoma efikasni.

Interpretatori. Interpretatori su programski prevodioci kod kojih su faza prevođenja i faza izvršavanja programa isprepletane. Interpretatori analiziraju deo po deo (najčešće naredbu po naredbu) izvornog kôda programa i odmah nakon analize vrše i njegovo izvršavanje. Rezultat prevođenja se ne smešta u izvršne datoteke, već je prilikom svakog izvršavanja neophodno iznova vršiti analizu izvornog kôda. Zbog ovoga, programi koji se interpretiraju se obično izvršavaju znatno sporije nego u slučaju kompilacije. S druge strane, razvojni ciklus programa je često kraći ukoliko

[5]Ipak, ne samo u akademskom okruženju, smatra se da je veoma poželjno da se uz izvršni distribuira i izvorni kôd programa (tzv. *softver otvorenog kôda, engl. open source)* kako bi korisnici mogli da vrše modifikacije i prilagođavanja programa za svoje potrebe.

se koriste interpretatori. Naime, prilikom malih izmena programa nije potrebno iznova vršiti analizu celokupnog kôda.

Za neke programske jezike postoje i interpretatori i kompilatori. Intepretator se tada koristi u fazi razvoja programa da bi omogućio interakciju korisnika sa programom, dok se u fazi eksploatacije kompletno razvijenog i istestiranog programa koristi kompilator koji proizvodi program sa efikasnim izvršavanjem.

Danas se često primenjuje i tehnika kombinovanja kompilatora i interpretatora. Naime, kôd sa višeg programskog jezika se prvo kompilira u neki precizno definisan međujezik niskog nivoa (ovo je obično jezik neke apstraktne virtuelne mašine), a zatim se vrši interpretacija ovog međujezika i njegovo izvršavanje na konkretnom računaru. Ovaj pristup primenjen je kod programskog jezika Java, kod .Net jezika (C#, VB), itd.

Pitanja i zadaci za vežbu

Pitanje 4.1. *Ukoliko je raspoloživ kôd nekog programa na nekom višem programskom jeziku (na primer, na jeziku C), da li je moguće jednoznačno konstruisati odgovarajući mašinski kôd? Ukoliko je raspoloživ mašinski kôd nekog programa, da li je moguće jednoznačno konstruisati odgovarajući kôd na jeziku C?*

Pitanje 4.2. *Za koji programski jezik su izgrađeni prvi interpretator i kompilator i ko je bio njihov autor? Koji su najkorišćeniji programski jezici 1960-ih, koji 1970-ih, koji 1980-ih i 1990-ih, a koji danas? Šta je to strukturno, a šta objektno-orijentisano programiranje?*

Pitanje 4.3. *Koje su najznačajnije programske paradigme? U koju paradigmu spada programski jezik C? Šta su to proceduralni, a šta su to deklarativni jezici?*

Pitanje 4.4. *Šta je rezultat leksičke analize programa? Šta leksički analizator dobija kao svoj ulaz, a šta vraća kao rezultat svog rada? Šta je zadatak sintaksičke analize programa? Šta sintaksički analizator dobija kao svoj ulaz, a šta vraća kao rezultat svog rada?*

Pitanje 4.5. *Navesti primer leksički ispravnog, ali sintaksički neispravnog dela programa. Navesti primer sintaksički ispravnog, ali semantički neispravnog dela programa.*

Pitanje 4.6. *Šta karakteriše svaki tip podataka? Da li su tipovi prisutni i u prevedenom, mašinskom kôdu? Šta je to konverzija tipova? Šta znači da je programski jezik statički, a šta znači da je dinamički tipiziran?*

Pitanje 4.7. *Koje su osnovne naredbe u programskim jezicima? Šta je to GOTO naredba i po čemu je ona specifična?*

Pitanje 4.8. *Čemu služe potprogrami? Koji su najčešće vrste potprograma? Čemu služe parametri potprograma, a čemu služi povratna vrednost? Koji su najčešći načini prenosa parametara?*

Pitanje 4.9. *Šta podrazumeva modularnost programa? Čemu služe biblioteke? Šta je standardna, a šta rantajm biblioteka?*

Pitanje 4.10. *Šta su prednosti, a šta mane mogućnosti pristupa proizvoljnoj memoriji iz programa? Šta su sakupljači otpada?*

Pitanje 4.11. *Šta je kompilator, a šta interpretator? Koje su osnovne prednosti, a koje su mane korišćenja kompilatora u odnosu na korišćenje interpretatora?*

Pitanje 4.12. *Ako se koristi kompilator, da li je za izvršavanje programa korisniku neophodno dostaviti izvorni kôd programa? Ako se koristi interpretator, da li je za izvršavanje programa korisniku neophodno dostaviti izvorni kôd programa?*

Deo II

Jezik C

Glava 5

Osnovno o programskom jeziku C

Programski jezik C je programski jezik opšte namene koji je 1972. godine razvio Denis Riči[1] u Belovim telefonskim laboratorijama (engl. Bell Telephone Laboratories) u SAD. Ime C dolazi od činjenice da je jezik nastao kao naslednik jezika B (a koji je bio pojednostavljena verzija jezika BCPL). C je jezik koji je bio namenjen prevashodno pisanju sistemskog softvera i to u okviru operativnog sistema Unix. Međutim, vremenom je počeo da se koristi i za pisanje aplikativnog softvera na velikom broju drugih platformi. C je danas prisutan na širokom spektru platformi — od mikrokontrolera do superračunara. Jezik C je uticao i na razvoj drugih programskih jezika (najznačajniji od njih je jezik C++ koji se može smatrati proširenjem jezika C).

Jezik C spada u grupu imperativnih (proceduralnih) programskih jezika. Kako je izvorno bio namenjen za sistemsko programiranje, programerima nudi prilično direktan pristup memoriji i konstrukcije jezika su tako osmišljene da se jednostavno prevode na mašinski jezik. Zahvaljuju tome, u C-u se relativno jednostavno kreiraju programi koji su ranije uglavnom pisani na asemblerskom jeziku. Jezik je kreiran u minimalističkom duhu — ima mali broj ključnih reči, a dodatna funkcionalnost programerima se nudi uglavnom kroz korišćenje (standardizovanih) bibliotečkih funkcija (na primer, ne postoje naredbe jezika kojima bi se učitavali podaci sa tastature računara ili ispisivali na ekran, već se ovi zadaci izvršavaju pozivanjem funkcija iz standardne biblioteke).

U razvoju jezika C se od samog početka insistiralo na standardizaciji i prenosivosti kôda (tzv. portabilnosti), zahvaljujući čemu se isti programi na C-u mogu koristiti (tj. prevoditi) na različitim platformama.

5.1 Standardizacija jezika

Postoji nekoliko značajnih neformalnih i formalnih standarda jezika C:

K&R C. Brajan Kerningen[2] i Denis Riči objavili su 1978. godine prvo izdanje knjige *Programski jezik C* (*The C Programming Language)*. Ova knjiga, među programerima obično poznata kao *K&R* ili kao *white book*, godinama je služila kao neformalna specifikacija jezika. Čak i posle pojave

[1] Dennis Ritchie (1941–2011), američki informatičar, dobitnik Tjuringove nagrade 1983. godine.

[2] Brian Kerninghan (1942–), kanadsko-američki informatičar.

novih standarda, K&R je dugo vremena služio kao „najmanji zajednički imenilac“ koji je korišćen kada je bilo potrebno postići visok stepen prenosivosti, jer su konstrukte opisane u prvoj verziji K&R knjige uglavnom podržavali svi C prevodioci.

ANSI C i ISO C. Tokom 1980-ih godina, C se proširio na veliki broj heterogenih platformi i time se javila potreba za zvaničnom standardizacijom jezika. Godine 1989. američki nacionalni institut za standardizaciju (ANSI) objavio je standard pod zvaničnim nazivom *ANSI X3.159-1989 „Programming Language C“*. Ova verzija jezika C se obično jednostavnije naziva *ANSI C* ili *C89*. Godine 1990. Međunarodna organizacija za standardizaciju (ISO) usvojila je ovaj dokument (uz sitnije izmene) pod oznakom *ISO/IEC 9899:1990* čime je briga o standardizaciji jezika C prešla od američkog pod međunarodno okrilje. Ova verzija jezika se ponekad naziva i *C90*, pa *C89* i *C90* predstavljaju isti jezik. Ovaj jezik predstavlja nadskup K&R jezika C i uključuje mnoge konstrukte do tada nezvanično podržane od strane velikog broja prevodilaca.

C99. Godine 1999. usvojen je novi standard jezika C *ISO/IEC 9899:1999*, poznat pod kraćim imenom *C99*. Ovaj standard uveo je sitne izmene u odnosu na prethodni i uglavnom proširio jezik novim konstruktima, u velikoj meri inspirisanim modernim programskim jezicima kakav je C++.

C11 (nekada C1X). Godine 2007. započet je rad na novom standardu jezika C koji je završen 2011. godine i objavljen kao *ISO/IEC 9899:2011*. Ovaj, sada tekući standard, ozvaničio je neka svojstva već podržana od strane popularnih kompilatora i precizno opisao memorijski model radi jasnije i bolje podrške tzv. višenitnim izračunavanjima (kada se nekoliko izračunavanja u programu odvija paralelno).

U nastavku teksta uglavnom će biti razmatran ANSI C, pri čemu će biti uvedeni i neki konstrukti jezika C99 i C11 (uz jasnu naznaku da se radi o dopunama).

Pitanja za vežbu

Pitanje 5.1.1. *Kada je nastao programski jezik C? Ko je njegov autor? Za koji operativni sistem se vezuje nastanak programskog jezika C?*

Pitanje 5.1.2. *U koju grupu jezika spada C? Za šta se najviše koristi?*

Pitanje 5.1.3. *Koja je najpoznatija knjiga o jeziku C? Nabrojati sve zvanične standarde jezika C.*

5.2 Prvi programi

Jezik C deli mnoga svojstva sa drugim programskim jezicima, ali ima i svoje specifičnosti. U nastavku teksta jezik C biće prezentovan postupno, koncept po koncept, često iz opšte perspektive programiranja i programskih jezika. Na početku, biće navedeno i prodiskutovano nekoliko jednostavnijih programa koji ilustruju osnovne pojmove programskog jezika C.

Program „Zdravo¡‘

Prikaz jezika C biće započet programom koji na standardni izlaz ispisuje poruku `Zdravo!`. Neki delovi programa će biti objašnjeni samo površno, a u kasnijem tekstu će biti dat njihov detaljni opis.

Program 5.1.

```
#include <stdio.h>

int main() {
   printf("Zdravo!\n"); /* ispisuje tekst */
   return 0;
}
```

Navedeni program sastoji se iz definicije jedne funkcije i ona se zove `main` (od engleskog *main* — glavna, glavno). Program može da sadrži više funkcija, ali obavezno mora da sadrži funkciju koja se zove `main` i izvršavanje programa uvek počinje od te funkcije. Prazne zagrade iza njenog imena ukazuju na to da se eventualni argumenti ove funkcije ne koriste. Reč `int` pre imena funkcije označava da ova funkcija, kao rezultat, vraća celobrojnu vrednost (engl. integer), tj. vrednost tipa `int`.

Naredbe funkcije `main` su grupisane između simbola `{` i `}` (koji označavaju početak i kraj tela funkcije). Obe naredbe funkcije završavaju se simbolom `;`.

Kao što je ranije pomenuto, programi obično imaju više funkcija i funkcija `main` poziva te druge funkcije za obavljanje raznovrsnih podzadataka. Funkcije koje se pozivaju mogu da budu korisnički definisane (tj. napisane od strane istog programera koji piše program) ili bibliotečke (tj. napisane od strane nekog drugog tima programera). Određene funkcije čine takozvanu *standardnu biblioteku* programskog jezika C i njihov kôd se obično isporučuje uz sam kompilator. Prva naredba u navedenoj funkciji `main` je poziv standardne bibliotečke funkcije `printf` koja ispisuje tekst na takozvani *standardni izlaz* (obično ekran). Tekst koji treba ispisati se zadaje između para znakova `"`, koji se ne ispisuju. U ovom slučaju, tekst koji se ispisuje je `Zdravo!`. Ni znakovi `\n` se ne ispisuju, nego obezbeđuju prelazak u novi red. Kako bi C prevodilac umeo da generiše kôd poziva funkcije `printf`, potrebno je da zna tip njene povratne vrednosti i tipove njenih argumenata. Ovi podaci o funkciji `printf`, njen svojevrsni opis, takozvana *deklaracija*, nalaze se u *datoteci zaglavlja* `stdio.h` (ova datoteka čini deo standardne biblioteke zadužen za ulazno-izlaznu komunikaciju i deo je instalacije prevodioca za C). Prva linija programa, kojom se prevodiocu saopštava da je neophodno da uključi sadržaj datoteke `stdio.h` je *pretprocesorska direktiva*. To nije naredba C jezika, već instrukcija C pretprocesoru koji predstavlja nultu fazu kompilacije i koji pre kompilacije program priprema tako što umesto prve linije upisuje celokupan sadržaj datoteke `stdio.h`.

Naredba `return 0;` prekida izvršavanje funkcije `main` i, kao njen rezultat, vraća vrednost `0`. Vrednost funkcije vraća se u okruženje iz kojeg je ona pozvana, u ovom slučaju u okruženje iz kojeg je pozvan sâm program. Uobičajena konvencija je da se iz funkcije `main` vraća vrednost 0 da ukaže na to da je izvršavanje proteklo bez problema, a ne-nula vrednost da ukaže na problem ili grešku koja se javila tokom izvršavanja programa.

Tekst naveden između simbola `/*` i simbola `*/` predstavlja komentare. Oni su korisni samo programerima, doprinose čitljivosti i razumljivosti samog programa. Njih C prevodilac ignoriše i oni ne utiču na izvršnu verziju programa.

Primetimo da je svaka naredba u prikazanom programu pisana u zasebnom redu, pri čemu su neki redovi uvučeni u odnosu na druge. Naglasimo da sa stanovišta C prevodioca ovo nije neophodno (u ekstremnom slučaju, dopušteno bi bilo da je ceo kôd osim prve linije naveden u istoj liniji). Ipak, smatra se da je „nazubljivanje“ (engl. indentation) kôda u skladu sa njegovom sintaksičkom strukturom nezaobilazna praksa. Naglasimo i da postoje različiti stilovi nazubljivanja (na primer, da li otvorena vitičasta zagrada počinje u istom ili sledećem redu). Međutim, obično se smatra da nije bitno koji se stil koristi dok god se koristi na uniforman način.

Ukoliko se, na primer, koristi GCC prevodilac (što je čest slučaj pod operativnim sistemom Linux), *izvorni program*, poput navedenog programa (unet u nekom editoru teksta i sačuvan u datoteci), može se prevesti u *izvršni program* tako što se u komandoj liniji navodi:

```
gcc -o ime_izvrsnog_koda ime_izvornog_koda
```

Na primer, ako je izvorni kôd sačuvan u datoteci `zdravo.c`, a željeno ime izvršnog programa je `zdravo`, potrebno je uneti tekst:

```
gcc -o zdravo zdravo.c
```

U tom slučaju, izvršni program pokreće se sa `./zdravo`. Ako se ime izvršnog programa izostavi, podrazumeva se ime `a.out`[3].

Nakon pokretanja programa dobija se sledeći izlaz.

```
Zdravo!
```

Tokom prevođenja, prevodilac može da detektuje *greške* (engl. error) u izvornom programu. Tada prevodilac ne generiše izvršni program nego izveštava programera o vrsti tih grešaka i brojevima linija u kojima se nalaze. Programer, na osnovu tog izveštaja, treba da ispravi greške u svom programu i ponovi proces prevođenja. Pored grešaka, prevodilac može da prijavi i *upozorenja* (engl. warning) za mesta u programu koja ukazuju na potencijalne propuste u programu. Ako u prevođenju ne postoje greške već samo upozorenja, generiše se izvršni program, ali on se možda neće ponašati na željeni način, te treba biti oprezan sa njegovim korišćenjem. Više reči o greškama i upozorenjima biće u poglavlju 9.2.5.

Program koji ispisuje kvadrat unetog celog broja

Prethodni program nije uzimao u obzir nikakve podatke koje bi uneo korisnik, već je prilikom svakog pokretanja davao isti izlaz. Naredni program očekuje od korisnika da unese jedan ceo broj i onda ispisuje kvadrat tog broja.

Program 5.2.

```
#include <stdio.h>

int main() {
   int a;
```

[3] Razlog za ovo je istorijski – `a.out` je skraćeno za `assembler output` jer je izvršni program obično bio izlaz nakon faze asembliranja.

```
   printf("Unesite ceo broj: ");
   scanf("%i", &a);
   printf("Kvadrat unetog broja je: %i", a*a);
   return 0;
}
```

Prva linija funkcije `main` je takozvana *deklaracija promenljive*. Ovom deklaracijom se uvodi promenljiva `a` celobrojnog tipa — tipa `int`. Naredna naredba (poziv funkcije `printf`) na standardni izlaz ispisuje tekst `Unesite ceo broj:`, a naredba nakon nje (poziv funkcije `scanf`) omogućava učitavanje vrednosti neke promenljive sa *standardnog ulaza* (obično tastature). U okviru poziva funkcije `scanf` zadaje se format u kojem će podatak biti pročitan — u ovom slučaju treba pročitati podatak tipa `int` i format se u tom slučaju zapisuje kao `%i`. Nakon formata, zapisuje se ime promenljive u koju treba upisati pročitanu vrednost. Simbol `&` govori da će promenljiva `a` biti promenjena u funkciji `scanf`, tj. da će pročitani broj biti smešten na adresu promenljive `a`. Korišćenjem funkcije `printf`, pored fiksnog teksta, može se ispisati i vrednost neke promenljive ili izraza. U formatu ispisa, na mestu u tekstu gde treba ispisati vrednost izraza zapisuje se format tog ispisa — u ovom slučaju `%i`[4], jer će biti ispisana celobrojna vrednost. Nakon niske koja opisuje format, navodi se izraz koji treba ispisati — u ovom slučaju vrednost `a*a`, tj. kvadrat broja koji je korisnik uneo. Nakon prevođenja i pokretanja programa, korisnik unosi broj, a zatim se ispisuje njegov kvadrat. Na primer,

```
Unesite ceo broj: 5
Kvadrat unetog broja je: 25
```

Program koji izračunava rastojanje između tačaka

Sledeći primer prikazuje program koji računa rastojanje između dve tačke u dekartovskoj ravni. Osnovna razlika u odnosu na prethodne programe je što se koriste razlomljeni brojevi, tj. brojevi u pokretnom zarezu. Dakle, u ovom programu, umesto promenljivih tipa `int` koriste se promenljive tipa `double`, dok se prilikom učitavanja i ispisa ovakvih vrednost za format koristi niska `%lf` umesto niske `%i`. Dodatno, za računanje kvadratnog korena koristi se funkcija `sqrt` deklarisana u zaglavlju `math.h`.

Program 5.3.

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

int main() {
  double x1, y1, x2, y2;
  printf("Unesi koordinate prve tacke: ");
  scanf("%lf%lf", &x1, &y1);
  printf("Unesi koordinate druge tacke: ");
  scanf("%lf%lf", &x2, &y2);
  printf("Rastojanje je: %lf\n",
```

[4]Umesto `%i`, potpuno ravnopravno moguće je koristiti i `%d`. `%i` dolazi od toga što je podatak tipa `int`, a `%d` jer je u dekadnom brojevnom sistemu.

```
     sqrt((x2 - x1)*(x2 - x1) + (y2 - y1)*(y2 - y1)));
  return 0;
}
```

Nakon unosa podataka, traženo rastojanje se računa primenom Pitagorine teoreme $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. Pošto u jeziku C ne postoji operator stepenovanja, kvadriranje se sprovodi množenjem.

Za prevođenje programa koji koriste matematičke funkcije, prilikom korišćenja GCC prevodioca potrebno je navesti argument `-lm` (kao poslednji argument). Na primer,

```
gcc -o rastojanje rastojanje.c -lm
```

Nakon uspešnog prevođenja i pokretanja programa korisnik unosi podatke i ispisuje se rezultat. Na primer,

```
Unesi koordinate prve tacke: 0.0 0.0
Unesi koordinate druge tacke: 1.0 1.0
Rastojanje je: 1.414214
```

Program koji ispituje da li je uneti broj paran

Naredni program ispituje da li je uneti broj paran.

Program 5.4.

```
#include <stdio.h>

int main() {
  int a;
  printf("Unesi broj: ");
  scanf("%d", &a);
  if (a % 2 == 0)
     printf("Broj %d je paran\n", a);
  else
     printf("Broj %d je neparan\n", a);
  return 0;
}
```

Ceo broj je paran ako i samo ako je ostatak pri deljenju sa dva jednak nuli. Ostatak pri deljenju se izračunava operatorom `%`, dok se poređenje jednakosti vrši operatorom `==`. Operator ispitanja jednakosti (`==`) i operator dodele (`=`) se suštinski razlikuju. Naredba `if` je *naredba grananja* kojom se u zavisnosti od toga da li je navedeni uslov (u ovom slučaju `a % 2 == 0`) ispunjen usmerava tok programa. Uslov se uvek navodi u zagradama `()` iza kojih ne sledi simbol `;`. U slučaju da je uslov ispunjen, izvršava se prva naredba navedena nakon uslova (takozvana `then` grana), dok se u slučaju da uslov nije ispunjen izvršava, ukoliko postoji, naredba navedena nakon reči `else` (takozvana `else` grana). I `then` i `else` grana mogu da sadrže samo jednu naredbu ili više naredbi. Ukoliko sadrže više naredbi, onda te naredbe čine *blok* čiji se početak i kraj moraju označiti zagradama `{` i `}`. Grana `else` ne mora da postoji.

Program koji ispisuje tablicu kvadrata i korena

Naredni program ispisuje kvadrate i korene svih celih brojeva od 1 do 100.

Program 5.5.

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

#define N 100

int main() {
  int i;
  for (i = 1; i <= N; i++)
      printf("%3d %5d %7.4f\n", i, i*i, sqrt(i));
  return 0;
}
```

Novina u ovom programu, u odnosu na prethodne, je *petlja* `for` koja služi da se određene naredbe ponove više puta (obično uz različite vrednosti promenljivih). Petlja se izvršava tako što se prvo izvrši inicijalizacija promenljive `i` na vrednost 1 (zahvaljujući kôdu `i=1`), u svakom koraku petlje se vrednost promenljive `i` uvećava za 1 (zahvaljujući kôdu `i++`) i sve to se ponavlja dok vrednost promenljive `i` ne postane `N` (zahvaljujući uslovu `i <= N`). Pre prevođenja programa vrednost `N` menja se brojem 100 i to zahvaljujući tzv. pretprocesorskoj direktivi `#define N 100` koja se često koristi za definisanje konstantnih parametara u programima (efekat je isti kao da je u petlji stajalo `for (i = 1; i <= 100; i++)`, međutim, gornja granica je jasno izdvojena na početku programa i lakše se može uočiti i promeniti. O petljama biće više reči u glavi 7, a o pretprocesoru biće više reči u poglavlju 9.2.1.

Za svaku vrednost promenljive `i` između 1 i 100 izvršava se telo petlje – naredba `printf`. U okviru format niske se, umesto `%d`, koristi `%3d`, čime se postiže da se broj uvek ispisuje u polju širine 3 karaktera. Slično se, korišćenjem `%7.4f`, postiže da se vrednost korena ispisuje na 4 decimale u polju ukupne širine 7 karaktera.

Program koji ispituje zbir prvih prirodnih brojeva.

Naredni program ispisuje prvi prirodni broj za koji je zbir svih prirodnih brojeva do tog broja veći od 100.

Program 5.6.

```
#include <stdio.h>

#define N 100

int main() {
  int i = 1;
  int s = 1;
  while(s <= N) {
    i++;
```

```
    s = s+i;
  }
  printf("%d\n", i);
  return 0;
}
```

Novina u ovom programu, u odnosu na prethodne, je *petlja* `while` koja se izvršava sve dok je ispunjen zadati uslov. Deklaracija `int i = 1;` je *deklaracija sa inicijalizacijom* – njom se promenljvoj `i` dodeljuje inicijalna vrednost `1`. Kao i u prethodnom programu, fleksibilnosti radi, umesto fiksne vrednost `100` u uslovu petlje koristi se simboličko ime `N` uvedeno pretprocesorskom direktivom.

Pitanja i zadaci za vežbu

Pitanje 5.2.1. *Funkciju kojeg imena mora da sadrži svaki C program?*

Pitanje 5.2.2. *Šta su to pretprocesorske direktive? Da li je* `#include<stdio.h>` *pretprocesorska direktiva, naredba C-a ili poziv funkcije iz standardne biblioteke?*

Pitanje 5.2.3. *Šta su to datoteke zaglavlja?*

Pitanje 5.2.4. *Šta je standardna biblioteka programskog jezika?*

Pitanje 5.2.5. *Kojom naredbom se vraća rezultat funkcije u okruženje iz kojeg je ona pozvana?*

Zadatak 5.2.1. *Napisati program koji za uneti poluprečnik r izračunava obim i površinu kruga (za broj π koristiti konstantu* `M_PI` *iz zaglavlja* `math.h`*).* √

Zadatak 5.2.2. *Napisati program koji za unete koordinate temena trougla izračunava njegov obim i površinu.* √

Zadatak 5.2.3. *Napisati program koji za unete dve dužine stranica trougla a i b ($a > 0$, $b > 0$) i ugla između njih γ ($\gamma > 0$ je zadat u stepenima) izračunava dužinu treće stranice c i veličine dva preostala ugla α i β. Napomena: koristiti sinusnu i/ili kosinusnu teoremu i funkcije* `sin()` *i* `cos()` *iz zaglavlja* `math.h`*.* √

Zadatak 5.2.4. *Napisati program koji za unetu brzinu u $\frac{km}{h}$ ispisuje odgovarajuću brzinu u $\frac{m}{s}$.* √

Zadatak 5.2.5. *Napisati program koji za unetu početnu brzinu v_0, ubrzanje a i vreme t izračunava trenutnu brzinu v i pređeni put s tela koje se kreće ravnomerno ubrzano.* √

Zadatak 5.2.6. *Napisati program koji za unetih 9 brojeva $a_1, a_2, \ldots, a_9$ izračunava determinantu:*

$$\begin{vmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{vmatrix}$$

√

Zadatak 5.2.7. *Napisati program koji ispisuje maksimum tri uneta broja.* √

Zadatak 5.2.8. *Napisati program koji učitava cele brojeve a i b i zatim ispisuje sve kubove brojeva između njih.* √

Glava 6

Predstavljanje podataka i operacije nad njima

U ovoj glavi biće opisani naredni pojmovi:

Promenljive i konstante, koje su osnovni oblici podataka kojima se operiše u programu.

Tipovi promenljivih, koji određuju vrstu podataka koje promenljive mogu da sadrže, način reprezentacije i skup vrednosti koje mogu imati, kao i skup operacija koje se sa njima mogu primeniti.

Deklaracije, koje uvode spisak promenljivih koje će se koristiti, određuju kog su tipa i, eventualno, koje su im početne vrednosti.

Operatori, odgovaraju operacijama koje su definisane nad podacima određene vrste.

Izrazi, koji kombinuju promenljive i konstante (korišćenjem operatora), dajući nove vrednosti.

6.1 Promenljive i imena promenljivih

Promenljive su osnovni objekti koji se koriste u programima. Promenljiva u svakom trenutku svog postojanja ima vrednost kojoj se može pristupiti — koja se može pročitati i koristiti, ali i koja se (ukoliko nije traženo drugacije) može menjati.

Imena promenljivih (ali i funkcija, struktura, itd.) određena su *identifikatorima*. U prethodnim programima korišćene su promenljive čija su imena `a`, `i`, `x1`, `x2` itd. Generalno, identifikator može da sadrži slova i cifre, kao i simbol `_` (koji je pogodan za duga imena), ali identifikator ne može počinjati cifrom. Dodatno, ključne reči jezika C (na primer, `if`, `for`, `while`) ne mogu se koristiti kao identifikatori.

U identifikatorima, velika i mala slova se razlikuju. Na primer, promenljive sa imenima `a` i `A` se tretiraju kao dve različite promenljive. Česta praksa je da malim slovima počinju imena promenljivih i funkcija, a velikim imena simboličkih konstanti, vrednosti koje se ne menjaju u toku programa.

Imena promenljivih i funkcija, u principu, treba da oslikavaju njihovo značenje i ulogu u programu, ali za promenljive kao što su indeksi u petljama se obično koriste kratka, jednoslovna imena (na primer `i`). Ukoliko ime promenljive sadrži više reči, onda se te reči, radi bolje čitljivosti, razdvajaju simbolom `_` (na primer, `broj_studenata`) ili početnim velikim slovima (na primer, `brojStudenata`) — ovo drugo je takozvana kamilja notacija (CamelCase). Postoje različite konvencije za imenovanje promenljivih. U nekim konvencijama, kao što je *mađarska notacija*, početna slova imena promenljivih predstavljaju kratku oznaka tipa te promenljive (na primer, `iBrojStudenata`).[1]

Iako je dozvoljeno, ne preporučuje se korišćenje identifikatora koji počinju simbolom `_`, jer se oni obično koriste za sistemske funkcije i promenljive.

ANSI C standard (C89) garantuje da se barem 31 početnih znakova imena promenljive smatra značajnom, dok standard C99 povećava taj broj na 63. Ukoliko dve promenljive imaju više od 63 početna znaka ista, onda se ne garantuje da će te dve promenljive biti razlikovane.[2]

6.2 Deklaracije

Sve promenljive moraju biti deklarisane pre korišćenja. Deklaracija[3] sadrži tip i listu od jedne ili više promenljivih tog tipa, razdvojenih zarezima.

```
int broj;  /* deklaracija celog broja        */
int a, b;  /* deklaracija vise celih brojeva */
```

U opštem slučaju nije propisano koju vrednost ima promenljiva neposredno nakon što je deklarisana[4]. Prilikom deklaracije može se izvršiti početna inicijalizacija. Moguće je kombinovati deklaracije sa i bez inicijalizacije.

```
int vrednost = 5;    /* deklaracija sa inicijalizacijom */
int a = 3, b, c = 5; /* deklaracije sa inicijalizacijom */
```

Kvalifikator `const` (dostupan u novijim standardima jezika C) može biti dodeljen deklaraciji promenljive da bi naznačio i obezbedio da se njena vrednost neće menjati, na primer:

```
/* ovu promenljivu nije moguce menjati */
const double GRAVITY = 9.81;
```

Deklaracije promenljivih mogu se navoditi na različitim mestima u programu. Najčešće se navode na početku funkcije (u dosada navedenim primerima to je bila funkcija `main`). Ukoliko je promenljiva deklarisana u nekoj funkciji, onda kažemo da je ona *lokalna* za tu funkciju i druge funkcije ne mogu da je koriste. Različite funkcije mogu imati lokalne promenljive istog imena. Promenljive deklarisane van svih funkcija su *globalne* i mogu se koristiti u više

[1] Postoje argumenti protiv korišćenja takve notacije u jezicima u kojima kompilatori vrše proveru korektnosti tipova.

[2] Za takozvane spoljašnje promenljive ove vrednosti su 6 (C89) odnosno 31 (C99).

[3] Deklaracije promenljivih najčešće su ujedno i definicije. Odnos između deklaracija i definicija promenljivih u jeziku C veoma je suptilan i biće razmotren u glavi 9.

[4] Samo u nekim specijalnim slučajevima (koji će biti diskutovani u daljem tekstu), podrazumevana vrednost je 0.

funkcija. Vidljivost tj. oblast važenja identifikatora (i njima uvedenih promenljivih) određena je pravilima *dosega identifikatora* o čemu će više reči biti u poglavlju 9.2.2.

6.3 Osnovni tipovi podataka

Kao i u većini drugih programskih jezika, u jeziku C podaci su organizovani u *tipove*. To omogućava da se u programima ne radi samo nad pojedinačnim bitovima i bajtovima (tj. nad nulama i jedinicama), već i nad skupovima bitova koji su, pogodnosti radi, organizovani u složenije podatke (na primer, cele brojeve ili brojeve u pokretnom zarezu). Jedan tip karakteriše: vrsta podataka koje opisuje, način reprezentacije, skup operacija koje se mogu primeniti nad podacima tog tipa, kao i broj bitova koji se koriste za reprezentaciju (odakle sledi opseg mogućih vrednosti).

6.3.1 Tip `int`

U jeziku C, cele brojeve opisuje tip `int` (od engleskog *integer*, *ceo broj*). Podrazumeva se da su vrednosti ovog tipa označene i reprezentuju se najčešće koristeći potpuni komplement. Nad podacima ovog tipa mogu se koristiti aritmetičke operacije (na primer, `+`, `-`, `*`, `/`, `%`), relacije (na primer, `<`, `>=`) i druge. Standardom nije propisano koliko bitova koriste podaci tipa `int`, ali je propisano da se koristi najmanje šesnaest bita (tj. dva bajta). Veličina tipa `int` je obično prilagođena konkretnoj mašini, tj. njenom mikroprocesoru. Na današnjim računarima, podaci tipa `int` obično zauzimaju 32 ili 64 bita , tj. 4 ili 8 bajtova[5].

Tipu `int` mogu biti pridruženi kvalifikatori `short`, `long` i (od standarda C99) `long long`. Ovi kvalifikatori uvode cele brojeve potencijalno različitih dužina u odnosu na `int`. Nije propisano koliko tipovi `short int`, `long int` i `long long int` zauzimaju bitova, ali propisano je da `short` zauzima barem dva bajta, da `int` zauzima barem onoliko bajtova koliko i `short int`, da `long int` zauzima barem onoliko bajtova koliko `int` i da zauzima barem četiri bajta, a da `long long int` zauzima barem onoliko koliko zauzima `long int`. Ime tipa `short int` može se kraće zapisati `short`, ime tipa `long int` može se kraće zapisati `long`, a ime tipa `long long int` može se kraće zapisati sa `long long`.

Bilo kojem od tipova `short`, `int`, `long` i `long long` mogu biti pridruženi kvalifikatori `signed` ili `unsigned`. Kvalifikator `unsigned` označava da se broj tretira kao neoznačen i da se sva izračunavanja izvršavaju po modulu 2^n, gde je n broj bitova koji tip koristi. Kvalifikator `signed` označava da se broj tretira kao označen i za njegovu reprezentaciju se najčešće koristi zapis u potpunom komplementu. Ukoliko uz tip `short`, `int`, `long` ili `long long` nije naveden ni kvalifikator `signed` ni kvalifikator `unsigned`, podrazumeva se da je vrednost označen broj.

Podaci o opsegu ovih (i drugih tipova) za konkretan računar i C prevodilac sadržani su u standardnoj datoteci zaglavlja `<limits.h>`. Pregled danas najčešćih vrednosti (u tridesetdvobitnom okruženju) dat je u tabeli 6.1.

[5] Kažemo da veličina podataka zavisi od *sistema*, pri čemu se pod sistemom podrazumeva i hardver računara i operativni sistem na kojem će se program izvršavati.

	označeni (signed)	neoznačeni (unsigned)
karakteri (char)	1B = 8b [-2^7, 2^7-1] = [-128, 127]	1B = 8b [0, 2^8-1] = [0, 255]
kratki (short int)	2B = 16b [-32K, 32K-1] = [-2^{15}, 2^{15}-1] = [-32768, 32767]	2B = 16b [0, 64K-1] = [0, 2^{16}-1] = [0, 65535]
dugi (long int)	4B = 32b [-2G, 2G-1] = [-2^{31}, 2^{31}-1] = [-2147483648,2147483647]	4B = 32b [0, 4G-1] = [0, 2^{32}-1] = [0, 4294967295]
veoma dugi (long long int) od C99	8B = 64b [-2^{63}, 2^{63}-1] = [-9.2·10^{18}, 9.2·10^{18}]	8B = 64b [0, 2^{64}-1] = [0, 1.84·10^{19}]

Slika 6.1: Najčešći opseg tipova (važi i na x86 + gcc platformi).

Prilikom štampanja i učitavanja vrednosti celobrojnih tipova, u pozivu funkcije `printf` ili `scanf`, potrebno je navesti i različite formate (npr. `%u` za `unsigned`, `%l` za `long`), o čemu će više reči biti u glavi 12.

Konačan opseg tipova treba uvek imati u vidu jer iz ovog razloga neke matematičke operacije neće dati očekivane vrednosti. Na primer, na trideset-dvobitnom sistemu, naredni program štampa negativan rezultat.

Program 6.1.

```
#include <stdio.h>

int main() {
  int a = 2000000000, b = 2000000000;
  printf("Zbir brojeva %d i %d je: %d\n", a + b);
  return 0;
}
```

```
  Zbir brojeva 2000000000 i 2000000000 je: -294967296
```

Da su promenljive bile deklarisane kao promenljive tipa `unsigned int` i štampane korišćenjem formata `%u`, rezultat bi bio ispravan i u smislu neograničenih celih brojeva.

Program 6.2.

```
#include <stdio.h>

int main() {
  unsigned int a = 2000000000, b = 2000000000;
  printf("Zbir brojeva %u i %u je: %u\n", a, b, a + b);
  return 0;
}
```

```
Zbir brojeva 2000000000 i 2000000000 je: 4000000000
```

6.3.2 Tip `char`

Male cele brojeve opisuje tip `char` (od engleskog *character — karakter, simbol, znak*). I nad podacima ovog tipa mogu se primenjivati aritmetičke operacije i relacije. Podatak tipa `char` zauzima tačno jedan bajt. Standard ne propisuje da li se podatak tipa `char` smatra označenim ili neoznačenim brojem (na nekom sistemu se može smatrati označenim, a na nekom drugom se može smatrati neoznačenim). Na tip `char` mogu se primeniti kvalifikatori `unsigned` i `signed`. Kvalifikator `unsigned` obezbeđuje da se vrednost tretira kao neoznačen broj, skup njegovih mogućih vrednosti je interval od 0 do 255 i sva izračunavanja nad podacima ovog tipa se izvršavaju po modulu $2^8 = 256$. Kvalifikator `signed` obezbeđuje da se vrednost tretira kao označen broj i za njegovu reprezentaciju se najčešće koristi zapis u potpunom komplementu, a skup njegovih mogućih vrednosti je interval od -128 do 127.

Iako je `char` opšti brojevni tip podataka i može da se koristi za predstavljanje vrednosti malih celih brojeva, u jeziku C se ovaj tip obično koristi za brojeve koji predstavljaju kôdove karaktera (otuda i ime `char`). Standard ne propisuje koje kodiranje se koristi, ali na savremenim PC računarima i C implementacijama, najčešće se koristi ASCII kodiranje karaktera. U tom slučaju, karakteri su u potpunosti identifikovani svojim ASCII kôdovima i obratno. Bez obzira da li je u pitanju označeni ili neoznačeni karakter, opseg dozvoljava da se sačuva svaki ASCII kôd (podsetimo se, ASCII kodovi su sedmobitni i imaju vrednosti između 0 i 127).

Brojevna vrednost promenljive `c` tipa `char` može se ispisati sa `printf("%d", c)`, a znakovna sa `printf("%c", c)`. Formati koji se koriste za ispisivanje i učitavanje vrednosti ovog i drugih tipova navedeni su u glavi 12.

Standardna biblioteka sadrži mnoge funkcije (i makroe) za rad sa karakterskim tipom i one su deklarisane u datoteci zaglavlja `<ctype.h>`. Najkorišćenije su `isalpha`, `isdigit` i slične kojima se proverava da li je karakter slovo abecede ili je cifra, kao i `toupper`, `tolower` kojima se malo slovo prevodi u veliko i obratno. Više detalja o standardnoj biblioteci može se naći u glavi 11.

6.3.3 Tipovi `float`, `double` i `long double`

Realne brojeve ili, preciznije, brojeve u pokretnom zarezu opisuju tipovi `float`, `double` i (od standarda C99) `long double`. Tip `float` opisuje brojeve u pokretnom zarezu osnovne tačnosti, tip `double` opisuje brojeve u pokretnom zarezu dvostruke tačnosti, a tip `long double` brojeve u pokretnom zarezu proširene tačnosti. Nije propisano koliko ovi tipovi zauzimaju bitova, ali propisano je da `double` zauzima barem onoliko bajtova koliko i `float`, a da `long double` zauzima barem onoliko bajtova koliko `double`. Podaci o opsegu i detaljima ovih (i drugih tipova) za konkretan računar i C prevodilac sadržani su u standardnoj datoteci zaglavlja `<float.h>`.

I nad podacima ovih tipova mogu se koristiti uobičajene aritmetičke operacije (osim operacije računanja ostatka pri deljenju `%`) i relacije.

Brojevi u pokretnom zarezu u savremenim računarima se najčešće zapisuju u skladu sa IEEE754 standardom. Ovaj standard uključuje i mogućnost zapisa

specijalnih vrednosti (na primer, $+\infty$, $-\infty$, NaN) i one se ravnopravno koriste prilikom izvođenja aritmetičkih operacija. Na primer, vrednost izraza 1.0/0.0 je $+\infty$, za razliku od celobrojnog izraza 1/0 čije izračunavanje dovodi do greške prilikom izvršavanja programa (engl. division by zero error). Standard ne definiše ponašanje u slučaju celobrojnog deljenja nulom a obično se za takvo deljenje u kôdu prijavljuje samo upozorenje (ukoliko se do tog deljenja dođe u fazi izvršavanja, prijavljuje se greška i prekida se izvršavanje programa).

Najveći broj funkcija iz C standardne biblioteke (pre svega matematičke funkcije definisane u zaglavlju `<math.h>`) koriste tip podataka `double`. Tip `float` se u programima koristi uglavnom zbog uštede memorije ili vremena na računarima na kojima je izvođenje operacija u dvostrukoj tačnosti veoma skupo (u današnje vreme, međutim, većina računara podržava efikasnu manipulaciju brojevima zapisanim u dvostrukoj tačnosti).

I prilikom štampanja i učitavanja vrednosti ovih tipova, u pozivu funkcije `printf` ili `scanf`, potrebno je navesti različite formate (npr. `%f` za `float` i `double`[6]), o čemu će više reči biti u glavi 12.

6.3.4 Logički tip podataka

Iako mnogi programski jezici imaju poseban tip za predstavljanje (logičkih) istinitosnih vrednosti, programski jezik C sve do standarda C99 nije imao ovakav tip podataka već je korišćena konvencija da se logički tip predstavlja preko brojevnih (najčešće celobrojnih) vrednosti tako što se smatra da vrednost 0 ima istinitosnu vrednost *netačno*, a sve vrednosti različite od 0 imaju istinitosnu vrednost *tačno*. Ipak, C99 standardizuje tip podataka `bool` i konstante `true` koja označava *tačno* i `false` koja označava *netačno*.

6.4 Konstante i konstantni izrazi

Svaki izraz koji se pojavljuje u programu je ili promenljiva ili konstanta ili poziv funkcije ili složen izraz. Konstante su fiksne vrednosti kao, na primer, `0`, `2`, `2007`, `3.5`, `1.4e2` ili `'a'`. Ista vrednost se ponekad može predstaviti različitim konstantama. Za sve konstante i za sve izraze, pravilima jezika jednoznačno su određeni njihovi tipovi. Poznavanje tipova konstanti i izraza je važno jer od tih tipova može zavisiti vrednost složenog izraza u kojem figuriše konstanta ili neki podizraz. Od tipova konstanti i izraza zavisi i koje operacije je moguće primeniti nad njima. Tipovi konstanti i izraza su neophodni i kako bi se znalo koliko memorijskog prostora treba rezervisati za neke međurezultate u fazi izvršavanja.

6.4.1 Celobrojne konstante

Celobrojna konstanta u tekstu programa, kao što je `123` ili `-456`, je tipa `int`. Velike celobrojne konstante koje ne mogu biti reprezentovane tipom `int`, a mogu u tip `long` su tipa `long`. Ako ne mogu da budu reprezentovane ni tipom `long`, a mogu tipom `unsigned long`, onda su tipa `unsigned long`. Dakle,

[6] Što se tiče učitavanja podataka tipa `double` od standarda C99 dopušteno je i navođenje formata `%lf`.

tačan tip dekadne celobrojne konstante ne može da se odredi ako se ne znaju detalji sistema. Na primer, na većini sistema poziv

```
printf("%d %d", 2147483647, 2147483648);
```

ispisuje

```
2147483647 -2147483648
```

jer se `2147483647` tumači kao konstanta tipa `int`, dok se `2147483648` tumači kao konstanta tipa `unsigned long`, što ne odgovara formatu `%d`.

Ukoliko se želi da se neka celobrojna konstanta tretira kao da ima tip `unsigned`, onda se na njenom kraju zapisuje slovo `u` ili `U`. Tip takve konstante bié `unsigned long`, ako ona može da bude reprezentovana tim tipom, a `unsigned int` inače. Ukoliko se želi eksplicitno naglasiti da se neka celobrojna konstanta tretira kao da je tipa `long`, onda se na njenom kraju zapisuje slovo `l` ili `L`. Na primer, `12345l` je tipa `long`, `12345` je tipa `int`, a `12345ul` je `unsigned long`.

Osim u dekadnom, celobrojne konstante mogu biti zapisane i u oktalnom i u heksadekadnom sistemu. Zapis konstante u oktalnom sistemu počinje cifrom `0`, a zapis konstante u heksadekadnom sistemu počinje simbolima `0x` ili `0X`. Na primer, broj `31` se može u programu zapisati na sledeće načine: `31` (dekadni zapis), `037` (oktalni zapis), `0x1f` (heksadekadni zapis). I oktalne i heksadekadne konstante mogu da budu označene slovima `U` ili `u` tj. `l` i `L` na kraju svog zapisa.

Negativne konstante ne postoje, ali se efekat može postići izrazima gde se ispred konstante navodi unarni operator `-`. Slično, može se navesti i operator plus, ali to nema efekta (npr. `+123` je isto kao i `123`).

6.4.2 Konstante u pokretnom zarezu

Konstante realnih brojeva ili, preciznije, konstantni brojevi u pokretnom zarezu sadrže decimalnu tačku (na primer, `123.4`) ili eksponent (`1e-2`) ili i jedno i drugo. Vrednosti ispred i iza decimalne tačke mogu biti izostavljene (ali ne istovremeno). Na primer, ispravne konstante su i `.4` ili `5.` (ali ne i `.`). Brojevi su označeni i konstante mogu počinjati znakom `-` ili znakom `+` (koji ne proizvodi nikakav efekat). Tip svih ovih konstanti je `double`, osim ako na kraju zapisa imaju slovo `f` ili `F` kada je njihov tip `float` (npr. `1.23f`). Slova `L` i `l` na kraju zapisa označavaju da je tip vrednosti `long double`.

6.4.3 Karakterske konstante

Iako se tip `char` koristi i za predstavljanje malih celih brojeva, on se prevashodno koristi za predstavljanje kôdova karaktera (najčešće ASCII kôdova). Direktno specifikovanje karaktera korišćenjem numeričkih kôdova nije preporučljivo. Umesto toga, preporučuje se korišćenje karakterskih konstanti. Karakterske konstante u programskom jeziku C se navode između `"` navodnika. Vrednost date konstante je numerička vrednost datog karaktera u korišćenoj karakterskoj tabeli (na primer, ASCII). Na primer, u ASCII kodiranju, karakterska konstanta `'0'` predstavlja vrednost 48 (koja nema veze sa numeričkom vrednošću 0), `'A'` je karakterska konstanta čija je vrednost u ASCII kôdu 65,

`'a'` je karakterska konstanta čija je vrednost u ASCII kôdu 97, što je ilustrovano sledećim primerom.

```
char c = 'a';
char c = 97; /* ekvivalentno prethodnom (na ASCII masinama),
                ali se ne preporucuje zbog toga sto smanjuje
                citljivost i prenosivost programa */
```

Karakterske konstante su tipa `int`, ali se najčešće dodeljuju promenljivama tipa `char` i tada se njihova vrednost konvertuje (više o konverzijama tipova prilikom dodele biće rečeno u poglavlju 6.6). Standard dopušta i navođenje više karaktera između navodnika (npr. `'ab'`), ali vrednost ovakve konstante nije precizno definisana standardom (stoga ćemo takve konstante izbegavati u programima).

Specijalni karakteri se mogu navesti korišćenjem specijalnih sekvenci karaktera koje počinju karakterom \ (engl. escape sequences). Jezik C razlikuje sledeće specijalne sekvence:

`\a`	alert (bell) character
`\b`	backspace
`\f`	formfeed
`\n`	newline
`\r`	carriage return
`\t`	horizontal tab
`\v`	vertical tab
`\\`	backslash
`\?`	question mark
`\'`	single quote
`\"`	double quote
`\ooo` (npr. `\012`)	octal number
`\xhh` (npr. `\x12`)	hexadecimal number

Karakterska konstanta `'\0'` predstavlja karakter čija je vrednost nula. Ovaj karakter ima specijalnu ulogu u programskom jeziku C jer se koristi za označavanje kraja niske karaktera (o čemu će više biti reči u nastavku). Iako je numerička vrednost ovog karaktera baš 0, često se u programima piše `'\0'` umesto 0 kako bi se istakla karakterska priroda ovog izraza.

Pošto se karakterske konstante identifikuju sa njihovim numeričkim vrednostima, one predstavljaju podatke tipa `int` i mogu ravnopravno da učestvuju u aritmetičkim izrazima (o izrazima će više biti rečeno u nastavku ove glave). Na primer, na ASCII sistemima, izraz `'0' <= c && c <= '9'` proverava da li karakterska promenljiva `c` sadrži ASCII kôd neke cifre, dok se izrazom `c - '0'` dobija numerička vrednost cifre čiji je ASCII kôd sadržan u promenljivoj `c`.

6.4.4 Konstantni izrazi

Konstantni izraz je izraz koji sadrži samo konstante (na primer, `4 + 3*5`). Takvi izrazi mogu se izračunati u fazi prevođenja i mogu se koristiti na svakom mestu na kojem se može pojaviti konstanta. Tip konstantnog izraza zavisi od tipova operanada (više o tome biće rečeno u poglavlju 6.6.4)./

6.5 Operatori i izrazi

Izrazi se u programskom jeziku C grade od konstanti i promenjivih primenom širokog spektra operatora. Osim od konstanti i promenljivih, elementarni izrazi se mogu dobiti i kao rezultat poziva funkcija, operacija pristupa elementima nizova, struktura i slično.

Operatorima su predstavljene osnovne operacije i relacije koje se mogu vršiti nad podacima osnovnih tipova u jeziku C. Za celobrojne tipove, te operacije uključuju aritmetičke, relacijske i logičke operacije. Podržane su i operacije koje se primenjuju nad pojedinačnim bitovima celobrojnih vrednosti.

Operatori se dele na osnovu svoje *arnosti* tj. broja operanada na koje se primenjuju. *Unarni* operatori deluju samo na jedan operand i mogu biti *prefiksni* kada se navode pre operanda i *postfiksni* kada se navode nakon svog operanda. *Binarni* operatori imaju dva operanda i obično su infiksni tj. navode se između svojih operanda. U jeziku C postoji jedan *ternarni* operator koji se primenjuje na tri operanda.

Vrednost izraza određena je vrednošću elementarnih podizraza (konstanti, promenljivih) i pravilima pridruženim operatorima. Neka pravila izračunavanja vrednosti izraza nisu propisana standardom i ostavljena je sloboda implementatorima prevodilaca da u potpunosti preciziraju semantiku na način koji dovodi do efikasnije implementacije. Tako, na primer, standardi ne definišu kojim se redom izračunavaju operandi operatora `+` pre njihovog sabiranja, pa nije propisano koja od funkcija `f` i `g` će biti prva pozvana prilikom izračunavanja izraza `f() + g()`.

6.5.1 Prioritet i asoscijativnost operatora

Izrazi mogu da obuhvataju više operatora i zagrade `(` i `)` se koriste da bi odredile kojim redosledom ih treba primenjivati. Ipak, slično kao i u matematici, postoje konvencije koje omogućavaju izostavljanje zagrada. Jedna od osnovnih takvih konvencija je *prioritet operatora* koji definiše kojim redosledom će se dva različita operatora primenjivati kada se nađu u istom, nezagrađenom izrazu. Na primer, kako je i očekivano, vrednost konstantnog izraza `3 + 4 * 5` biće `23`, jer operator `*` ima prioritet u odnosu na operator `+`. Semantika jezika C uglavnom poštuje prioritet koji važi u matematici, ali, s obzirom na mnogo veći broj operatora uvodi i dodatna pravila. Prioritet operatora dat je u tabeli navedenoj u dodatku A. Navedimo neke osnovne principe u definisanju prioriteta:

1. Unarni operatori imaju veći prioritet u odnosu na binarne.
2. Postfiksni unarni operatori imaju veći prioritet u odnosu na prefiksne unarne operatore.
3. Aritmetički operatori imaju prioritet u odnosu na relacijske koji imaju prioritet u odnosu na logičke operatore.
4. Operatori dodele imaju veoma nizak prioritet.

Druga važna konvencija je *asocijativnost operatora* koja definiše kojim redosledom će se izračunavati dva ista operatora ili operatora istog prioriteta kada se nađu uzastopno u istom, nezagrađenom izrazu. Obično se razlikuju

leva asocijativnost, kada se izraz izračunava sleva na desno i *desna asocijativnost*, kada se izraz izračunava zdesna na levo. Neki jezici razlikuju i *neasocijativnost*, kada je zabranjeno da se isti operator dva puta uzastopno ponovi u istom izrazu. Matematički, neki operatori su *asocijativni*, što znači da vrednost izraza ne zavisi od redosleda primene uzastopno ponovljenog operatora. Na primer, u matematici uvek važi $x + (y + z) = (x + y) + z$. Zbog načina reprezentacije podataka u računaru mnogi operatori (uključujući i `+`) nisu uvek asocijativni. Na primer, izraz `(INT_MIN + INT_MAX) + 1` ima vrednost `0` bez nastanka prekoračenja, dok izraz `INT_MIN + (INT_MAX + 1)` ima vrednost 0, ali dolazi do prekoračenja[7]. Ovakvo ponašanje je posledica činjenice da je u jeziku C sabiranje celih brojeva zapravo sabiranje po modulu. Operator `-` ima levu asocijativnost. Na primer, vrednost izraza `1 - 2 - 3` je `-4` (a ne `2`, što bi bio slučaj da ima desnu asocijativnost, iz čega je jasno da `-` nije asocijativan operator). Većina operatora ima levu asocijativnost (najznačajniji izuzeci su prefiksni unarni operatori i operatori dodele). Asocijativnost operatora data je u tabeli u dodatku A.

6.5.2 Operator dodele

Operatorom dodele se neka vrednost pridružuje datoj promenljivoj. Operator dodele se zapisuje `=`. Na primer,

```
broj_studenata = 80;
broj_grupa     =  2;
```

U dodeljivanju vrednosti, sa leve strane operatora dodele može da se nalazi promenljiva, a biće diskutovano kasnije, i element niza ili memorijska lokacija. Ti objekti, objekti kojima može biti dodeljena vrednost nazivaju se *l-vrednosti* (od engleskog *l-value* ili *left-value*).

Operator dodele se može koristiti za bilo koji tip l-vrednosti, ali samo ako je desna strana odgovarajućeg tipa (istog ili takvog da se njegova vrednost može konvertovati u tip l-vrednosti o čemu će više reči biti u poglavlju 6.6).

Tip izraza dodele je tip leva strane, a vrednost izraza dodele je vrednost koja će biti dodeljena levoj strani. Promena vrednosti objekta na levoj stranica je *bočni (sporedni) efekat* (engl. side effect) do kojeg dolazi prilikom izračunavanja vrednosti izraza. Na primer, izvršavanje naredbe `broj_studenata = 80;` svodi se na izračunavanje izraza `broj_studenata = 80`. Tip ovog izraza je tip promenljive `broj_studenata = 80`, vrednost je jednaka `80`, a prilikom ovog izračunavanja menja se vrednost promenljive `broj_studenata`. Ovakvo ponašanje može se iskoristiti i za višestruko dodeljivanje. Na primer, nakon sledeće naredbe, sve tri promenljive `x`, `y` i `z` imaće vrednost `0`:

```
x = y = z = 0;
```

6.5.3 Aritmetički operatori

Nad operandima brojevnih tipova mogu se primeniti sledeći aritmetički operatori:

[7] `INT_MIN` i `INT_MAX` su simbolička imena čije su vrednosti uvedene direktivom `#define` u datoteci zaglavlja `limits.h`. Njima odgovaraju vrednosti najmanje i najveće vrednosti tipa `int` na konkretnom sistemu.

`+` binarni operator sabiranja;

`-` binarni operator oduzimanja;

`*` binarni operator množenja;

`/` binarni operator (celobrojnog) deljenja;

`%` binarni operator ostatka pri deljenju;

`-` unarni operator promene znaka;

`+` unarni operator.

Operator `%` moguće je primeniti isključivo nad operandima celobrojnog tipa.

Operator deljenja označava različite operacije u zavisnosti od tipa svojih operanada.[8] Kada se operator deljenja primenjuje na dve celobrojne vrednosti primenjuje se celobrojno deljenje (tj. rezultat je celi deo količnika). Na primer, izraz `9/5` ima vrednost `1`. U ostalim slučajevima primenjuje se deljenje realnih brojeva (preciznije, deljenje brojeva u pokretnom zarezu). Na primer, izraz `9.0/5.0` ima vrednost `1.8` (jer se koristi deljenje brojeva u pokretnom zarezu). U slučaju da je jedan od operanada ceo broj, a drugi broj u pokretnom zarezu, vrši se implicitna konverzija (promocija) celobrojnog operanda u broj u pokretnom zarezu i primenjuje se deljenje brojeva u pokretnom zarezu (više reči o konverzijama biće u poglavlju 6.6).

Prefiksni unarni operatori `+` i `-` imaju desnu asocijativnost i viši prioritet od svih binarnih operatora. Operatori `*`, `/` i `%` imaju isti prioritet, viši od prioriteta binarnih operatora `+` i `-`. Svi navedeni binarni operatori imaju levu asocijativnost.

Inkrementiranja i dekrementiranje. Operator inkrementiranja (uvećavanja za 1) zapisuje se sa `++`, a operator dekrementiranja (umanjivanja za 1) zapisuje se sa `--`:

`++` (prefiksno i postfiksno) inkrementiranje;

`--` (prefiksno i postfiksno) dekrementiranje.

Oba operatora mogu se primeniti nad celim brojevima i brojevima u pokretnom zarezu. Inkrementiranje i dekrementiranje se mogu primenjivati samo nad l-vrednostima (najčešće su to promenljive ili elementi nizova). Tako, na primer, izraz `5++` nije ispravan. Oba operatora su unarna (imaju po jedan operand) i mogu se upotrebiti u prefiksnom (na primer, `++n`) ili postfiksnom obliku (na primer, `n++`). Razlika između ova dva oblika je u tome što `++n` uvećava vrednost promenljive `n` pre nego što je ona upotrebljena u širem izrazu, a `n++` je uvećava nakon što je upotrebljena. Preciznije, vrednost izraza `n++` je stara vrednost promenljive `n`, a vrednost izraza `++n` je nova vrednost promenljive `n`, pri čemu se u oba slučaja, prilikom izračunavanja vrednosti izraza, kao sporedni efekat, uvećava vrednost promenljive `n`. Na primer, ako promenljiva `n` ima vrednost `5`, onda

[8] Na hardveru su operacije nad celim brojevima i brojevima u pokretnom zarezu implementirane nezavisno i u izvršnom programu koristi se jedna od njih, izabrana u fazi prevođenja u zavisnosti od tipova operanada. Informacije o tipovima iz izvornog programa su na ovaj, ali i na druge slične načine, upotrebljene tokom prevođenja i one se ne čuvaju u izvršnom programu.

```
x = n++;
```

dodeljuje promenljivoj `x` vrednost `5`, a

```
x = ++n;
```

dodeljuje promenljivoj `x` vrednost `6`. Promenljiva `n` u oba slučaja dobija vrednost `6`. Slično, kôd

```
int a = 3, b = 3, x = a++, y = ++b;
printf("a = %d, b = %d, x = %d, y = %d\n);
```

ispisuje

```
a = 4, b = 4, x = 3, y = 4
```

Ukoliko ne postoji širi kontekst, tj. ako inkrementiranje čini čitavu naredbu, vrednost izraza se i ne koristi i onda nema razlike između naredbe `n++;` i `++n;`. Na primer,

```
int a = 3, b = 3;
a++; ++b;
printf("a = %d, b = %d\n", a, b);
```

prethodni kôd ispisuje

```
a = 4, b = 4
```

Sâm trenutak u kojem se vrši izvršavanje sporednog efekta precizno je definisan standardom i određen tzv. *sekvencionim tačkama* (engl. sequence point) jezika u kojima se garantuje da je efekat izvršen. Tako je, na primer, između svake dve susedne promenljive koje se deklarišu (mesto označenom simbolom `,`) sekvenciona tačka, kao i kraj naredbe i deklaracije (mesto označeno sa `;`), dok operator `+` to često nije[9]. Na primer, naredni kôd

```
int a = 3, x = a++, y = a++;
int b = 3, z = b++ + b++;
printf("a = %d, x = %d, y = %d,\n", a, x, y);
printf("b = %d, z = %d\n", b, z);
```

ispisuje

```
a = 5, x = 3, y = 4,
b = 5, z = 6
```

Zaista, `x` prima originalnu vrednost promenljive `a` (vrednost `3`). Pošto je zarez nakon inicijalizacije promenljive `x` označava mesto gde je sekvenciona tačka, prvo uvećanje promenljive `a` vrši se na tom mestu tako da `y` prima uvećanu vrednost promenljive `a` (vrednost `4`). Drugo uvećanje promenljive `a` (na vrednost `5` vrši se na kraju deklaracije sa inicijalizacijama, tj. na mestu označenom sa `;`). U drugom slučaju promenljiva `z` sabira dva puta originalnu vrednost promenljive `b` (vrednost `3`), jer `+` ne označava sekvencionu tačku, a promenljiva

[9] Precizan spisak svih sekvencionih tačaka može se naći u standardu jezika.

`b` se uvećava i to dva puta na kraju deklaracije sa inicijalizacijama (na mestu obeleženom sa `;`).

Primetimo da semantika operatora inkrementiranja može biti veoma komplikovana i stoga se korišćenje slozenijih izraza sa ovim operatorima, koji se oslanjaju na fine detalje semantike, ne savetuje.

6.5.4 Relacioni i logički operatori

Relacioni operatori. Nad celim brojevima i brojevima u pokretnom zarezu mogu se koristiti sledeći binarni relacioni operatori:

`==` jednako;

`!=` različito.

`>` veće;

`>=` veće ili jednako;

`<` manje;

`<=` manje ili jednako;

Relacioni operatori poretka `<`, `<=`, `>` i `>=` imaju isti prioritet i to viši od operatora jednakosti `==` i različitosti `!=` i svi imaju levu asocijativnost. Rezultat relacionog operatora primenjenog nad dva broja je vrednost 0 (koja odgovara istinitosnoj vrednosti *netačno*) ili vrednost 1 (koja odgovara istinitosnoj vrednosti *netačno*). Na primer, izraz `3 > 5` ima vrednost 0, a izraz `7 < 5 != 1` je isto što i `(7 < 5) != 1` i ima vrednost 1 jer izraz `7 < 5` ima vrednost 0, što je različito od 1. Ako promenljiva `x` ima vrednost 2, izraz `3 < x < 5` ima vrednost 1 (*tačno*) što je različito od možda očekivane vrednosti 0 (*netačno*) jer 2 nije između 3 i 5. Naime, izraz se izračunava sleva na desno, podizraz `3 < x` ima vrednost 0, a zatim izraz `0 < 5` ima vrednost 1. Kako u ovakvim slučajevima kompilator ne prijavljuje grešku, a u fazi izvršavanja se dobija rezultat neočekivan za početnike, ovo je opasna greška programera koji su navikli na uobičajenu matematičku notaciju. Dakle, proveru da li je neka vrednost između dve zadate neophodno je vršiti uz primenu logičkog operatora `&&`.

Binarni relacioni operatori imaju niži prioritet od binarnih aritmetičkih operatora.

Operator `==` koji ispituje da li su neke dve vrednosti jednake i operator dodele `=` različiti su operatori i imaju potpuno drugačiju semantiku. Njihovo nehotično mešanje čest je uzrok grešaka u C programima.

Logički operatori. Logički operatori primenjuju se nad brojevnim vrednostima i imaju tip rezultata `int`. Brojevnim vrednostima pridružene su logičke ili istinitosne vrednosti na sledeći način: ukoliko je broj jednak 0, onda je njegova logička vrednost 0 (*netačno*), a inače je njegova logička vrednost 1 (*tačno*). Iako su sve vrednosti operanada koje su različite od 0 dopuštene i tumače se kao *tačno*, rezultat izračunavanja *tačno* nije proizvoljna vrednost različita od 0, već isključivo vrednost 1.

Postoje sledeći logički operatori:

`!` logička negacija — *ne*;

`&&` logička konjunkcija — *i*;

`||` logička disjunkcija — *ili*.

Operator `&&` ima viši prioritet u odnosu na operator `||`, a oba su levo asocijativna. Binarni logički operatori imaju niži prioritet u odnosu na binarne relacijske i logičke operatore. Operator `!`, kao unarni operator, ima viši prioritet u odnosu na bilo koji binarni operator i desno je asocijativan. Na primer,

- vrednost izraza `5 && 4.3` jednaka je `1`;
- vrednost izraza `10.2 || 0` jednaka je `1`;
- vrednost izraza `0 && 5` jednaka je `0`;
- vrednost izraza `!1` jednaka je `0`;
- vrednost izraza `!9.2` jednaka je `0`;
- vrednost izraza `!0` jednaka je `1`;
- vrednost izraza `!(2>3)` jednaka je `1`;
- irazom `3 < x && x < 5` proverava se da li je vrednost promenljive `x` između `3` i `5`;
- izraz `a > b || b > c && b > d` ekvivalentan je izrazu
  `(a>b) || ((b>c) && (b>d));`
- izrazom `g % 4 == 0 && g % 100 != 0 || g % 400 == 0` proverava se da li je godina `g` prestupna.

Lenjo izračunavanje. U izračunavanju vrednosti logičkih izraza koristi se strategija *lenjog izračunavanja* (engl. lazy evaluation). Osnovna karakteristika ove strategije je izračunavanje samo onog što je neophodno. Na primer, prilikom izračunavanja vrednosti izraza

```
2<1 && a++
```

biće izračunato da je vrednost podizraza `(2<1)` jednaka `0`, pa je i vrednost celog izraza (zbog svojstva logičkog *i*) jednaka `0`. Zato nema potrebe izračunavati vrednost podizraza `a++`, te će vrednost promenljive `a` ostati nepromenjena nakon izračunavanja vrednosti navedenog izraza. S druge strane, nakon izračunavanja vrednosti izraza

```
a++ && 2<1
```

vrednost promenljive `a` će biti promenjena (uvećana za 1). U izrazima u kojima se javlja logičko *i*, ukoliko je vrednost prvog operanda jednaka 1, onda se izračunava i vrednost drugog operanda.

U izrazu u kojem se javlja logičko *ili*, ukoliko je vrednost prvog operanda jednaka 1, onda se ne izračunava vrednost drugog operanda. Ukoliko je vrednost prvog operanda jednaka 0, onda se izračunava i vrednost drugog operanda. Na primer, nakon

```
1<2 || a++_
```

se ne menja vrednost promenljive `a`, a nakon

```
2<1 || a++
```

se menja (uvećava za 1) vrednost promenljive `a`.

6.5.5 Bitovski operatori

C podržava naredne operatore za rad nad pojedinačnim bitovima, koji se mogu primenjivati samo na celobrojne argumente:

`~` bitovska negacija;

`&` bitovska konjunkcija;

`|` bitovska disjunkcija;

`^` bitovska ekskluzivna disjunkcija;

`«` pomeranje (šiftovanje) bitova ulevo;

`»` pomeranje (šiftovanje) bitova udesno.

Kao unarni operator, operator `~` ima najveći prioritet i desno je asocijativan. Prioritet operatora pomeranja je najveći od svih binarnih bitovskih operatora — nalazi se između prioriteta aritmetičkih i relacijskih operatora. Ostali bitovski operatori imaju prioritet između relacijskih i logičkih operatora i to `&` ima veći prioritet od `^` koji ima veći prioritet od `|`. Ovi operatori imaju levu asocijativnost.

`&` – **bitovsko *i*** — primenom ovog operatora vrši se konjunkcija pojedinačnih bitova dva argumenta (*i*-ti bit rezultata predstavlja konjunkciju *i*-tih bitova argumenata). Na primer, ukoliko su promenljive `x1` i `x2` tipa `unsigned char` i ukoliko je vrednost promenljive `x1` jednaka 74, a promenljive `x2` jednaka 87, vrednost izraza `x1 & x2` jednaka je 66. Naime, broj 74 se binarno zapisuje kao `01001010`, broj 87 kao `01010111`, i konjunkcija njihovih pojedinačnih bitova daje `01000010`, što predstavlja zapis broja 66. S obzirom na to da se prevođ enje u binarni sistem efikasnije sprovodi iz heksadekadnog sistema nego iz dekadnog, prilikom korišćenja bitovskih operatora konstante se obično zapisuju heksadekadno. Tako bi, u prethodnom primeru, broj `x1` imao vrednost zapisanu kao `0x4A`, a broj `x2` bi imao vrednost zapisanu kao `0x57`.

`|` – **bitovsko *ili*** — primenom ovog operatora vrši se (obična) disjunkcija pojedinačnih bitova dva navedena argumenta. Za brojeve iz tekućeg primera, vrednost izraza `x1 | x2` je `01011111`, tj. 95, tj. `0x5F`.

`^` – **bitovsko *ekskluzivno ili*** — primenom ovog operatora vrši se ekskluzivna disjunkcija pojedinačnih bitova dva argumenta. Za brojeve iz tekućeg primera, vrednost izraza `x1 ^ x2` je `00011101`, tj. 29, tj. `0x1D`.

`~` – **jedinični komplement** — primenom ovog operatora vrši se komplementiranje (invertovanje) svakog bita argumenta. Na primer, vrednost izraza `~x1` u tekućem primeru je `10110101`, tj. `B5`, tj. 181.

`«` – **levo pomeranje (šiftovanje)** — primenom ovog operatora bitovi prvog argumenta se pomeraju u levo za broj pozicija naveden kao drugi argument. Početni bitovi prvog argumenta se zanemaruju, dok se na završna mesta rezultata upisuju nule. Levo pomeranje broja za jednu poziciju odgovara množenju sa dva. Na primer, ukoliko promenljiva `x` ima tip `unsigned char` i vrednost `0x95`, tj. `10010101`, vrednost izraza `x << 1` je `00101010`, tj. `0x2A`.

» – desno pomeranje (šiftovanje) — primenom ovog operatora bitovi prvog argumenta se pomeraju u desno za broj pozicija naveden kao drugi argument. Krajnji (desni) bitovi prvog argumenta se zanemarju, a što se tiče početnih (levih) bitova rezultata, postoji mogućnost da se oni popunjavaju uvek nulama (tzv. *logičko pomeranje*) ili da se oni popune vodećim bitom (koji predstavlja znak) prvog argumenta (tzv. *aritmetičko pomeranje*). Osnovna motivacija aritmetičkog pomeranja je da desno pomeranje broja za jednu poziciju odgovara celobrojnom deljenju sa dva. Koje pomeranje će se vršiti zavisi od tipa prvog argumenta. Ukoliko je argument neoznačen, početni bitovi rezultata će biti postavljeni na nulu, bez obzira na vrednost vodećeg bita prvog argumenta. Ukoliko je argument označen, početni bitovi rezultata će biti postavljeni na vrednost vodećeg bita prvog argumenta. Na primer, ukoliko promenljiva `x` ima tip `signed char` i vrednost `0x95`, tj. `10010101`, vrednost izraza `x >> 1` je `11001010`, tj. `0xCA`. Ukoliko je tip promenljive `x` `unsigned char`, tada je vrednost izraza `x >> 1` broj `01001010`, tj. `0x4A`.

Bitovske operatore ne treba mešati sa logičkim operatorima. Na primer, vrednost izraza `1 && 2` je `1` (tačno i tačno je tačno), dok je vrednost izraza `1 & 2` jednaka `0` (`000...001 & 000..010 = 000...000`).

Više reči o upotrebi bitovskih operatora kao i primeri programa biće dati u drugom tomu ove knjige.

6.5.6 Složeni operatori dodele

Dodela koja uključuje aritmetički operator `i = i + 2;` može se zapisati kraće i kao `i += 2;`. Slično, naredba `x = x * (y+1);` ima isto dejstvo kao i `x *= y+1;`. Za većinu binarnih operatora postoje odgovarajući složeni operatori dodele:

`+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `«=`, `»=`

Operatori dodele imaju niži prioritet od svih ostalih operatora i desnu asocijativnost.

Izračunavanje vrednosti izraza

```
izraz1 op= izraz2
```

obično ima isto dejstvo kao i izračunavanje vrednosti izraza

```
izraz1 = izraz1 op izraz2
```

gde je `op` jedan od nabrojanih operatora. Međutim, postoje slučajevi kada navedena dva izraza imaju različite vrednosti[10], te treba biti obazriv sa korišćenjem složenih operatora dodele.

Kraći zapis uz korišćenje složenih operatora dodele obično daje i nešto efikasniji kôd (mada savremeni kompilatori obično umeju i sami da prepoznaju situacije u kojima se izraz može prevesti isto kao da je zapisan u kraćem obliku).

[10] Na primer, prilikom izračunavanja izraza `a[i++] += 1` (`a[i]` označava `i`-ti element niza `a`), promenljiva i inkrementira se samo jednom, dok se u slučaju `a[i++] = a[i++] + 1` promenljiva i inkrementira dva puta. Slično, prilikom izračunavanja izraza `a[f()] += 1`, funkcija `f()` se poziva samo jednom, dok se u slučaju `a[f()] = a[f()] + 1` funkcija `f()` poziva dva puta, što može da proizvede različiti efekat ukoliko funkcija `f()` u dva različita poziva vraća različite vrednosti. Funkcije i nizovi su detaljnije opisani u narednim glavama.

Naredni primer ilustruje primenu složenog operatora dodele (obratiti pažnju na rezultat operacija koji zavisi od tipa operanda).

Program 6.3.

```
#include <stdio.h>

int main() {
    unsigned char c = 254;
    c += 1; printf("c = %d\n", c);
    c += 1; printf("c = %d\n", c);
    return 0;
}
```

Izlaz programa:

```
c = 255
c = 0
```

6.5.7 Operator uslova

Ternarni operator uslova `?:` se koristi u sledećem opštem obliku:

```
izraz1 ? izraz2 : izraz3;
```

Prioritet ovog operatora je niži u odnosu na sve binarne operatore osim dodela i operatora `,`.

Izraz `izraz1` se izračunava prvi. Ako on ima ne-nula vrednost (tj. ako ima istinitosnu vrednost *tačno*), onda se izračunava vrednost izraza `izraz2` i to je vrednost čitavog uslovnog izraza. Inače se izračunava vrednost `izraz3` i to je vrednost čitavog uslovnog izraza. Na primer, izračunavanjem izraza

```
m = (a > b) ? a : b
```

vrednost promenljive `m` postavlja se na maksimum vrednosti promenljivih `a` i `b`, dok se izračunavanjem izraza

```
m = (a < 0) ? -a : a
```

vrednost promenljive `m` postavlja na apsolutnu vrednost promenljive `a`.

Opisana semantika ternarnog operatora takođe je lenja. Na primer, nakon izvršavanja kôda

```
n = 0;
x = (2 > 3) ? n++ : 9;
```

promenljiva `x` imaće vrednost `9`, a promenljiva `n` će zadržati vrednost `0` jer se drugi izraz neće izračunavati.

6.5.8 Operator zarez

Binarni operator zarez (`,`) je operator kao i svaki drugi (najnižeg je prioriteta od svih operatora u C-u) i prilikom izračunavanja vrednosti izraza izgrađenog njegovom primenom, izračunavaju se oba operanda, pri čemu se vrednost celokupnog izraza definiše kao vrednost desnog operanda (ta vrednost se često zanemaruje). Ovaj operator se često koristi samo da spoji dva izraza u jedinstveni (kako bi se takav složeni izraz mogao upotrebiti na mestima gde sintaksa zahteva navođenje jednog izraza). To je najčešće u inicijalizaciji i koraku `for` petlje (o čemu će više reči biti u poglavlju 7.4). Još jedna od čestih upotreba je i u kôdu oblika

```
x = 3, y = 5; /* ekivalentno bi bilo i x = 3; y = 5; */
```

Treba razlikovati operator `,` od zareza koji razdvajaju promenljive prilikom deklaracije i poziva funkcije (koji nisu operatori). Česta greška u korišćenju ovog operatora je pri pokušaju pristupa višedimenzionom nizu (više o nizovima rečeno je u poglavlju 6.7.4). Na primer `A[1, 2]` potpuno je ekvivalentno sa `A[2]`, što je pogrešno ako je niz `A` dvodimenzioni. Ispravan način pristupa elementu na poziciji $(1, 2)$ je `A[1][2]`.

6.5.9 Operator sizeof

Veličinu u bajtovima koju zauzima neki tip ili neka promenljiva moguće je odrediti korišćenjem operatora `sizeof`. Tako, `sizeof(int)` predstavlja veličinu tipa `int` i na tridesetidvobitnim sistemima vrednost ovog izraza je najčešće 4.

Vrednost koju vraća operator `sizeof` ima tip `size_t`. Ovo je neoznačen celobrojni tip, koji obično služi za reprezentovanje veličine objekata u memoriji. Tip `size_t` ne mora nužno da bude jednak tipu `unsigned int` (standard C99 zahteva samo da vrednosti ovog tipa imaju barem dva bajta).

Pitanja i zadaci za vežbu

Pitanje 6.5.1.

1. *Koji su osnovni oblici podataka kojima se operiše u programu?*
2. *Šta sve karakteriše jedan tip podataka?*
3. *Da li se promenljivoj može menjati tip u toku izvršavanja programa?*
4. *Šta je to inicijalizacija promenljive? U opštem slučaju, ukoliko celobrojna promenljiva nije inicijalizovana, koja je njena početna vrednost?*
5. *Šta je uloga kvalifikatora* `const`*?*
6. *Kako se grade izrazi?*
7. *Nabrojati aritmetičke, relacijske operatore, logičke i bitovske operatore.*

Pitanje 6.5.2.

1. *Da li ime promenljive može počinjati simbolom* `_`*? Da li se ovaj simbol može koristiti u okviru imena?*

2. *Da li ime promenljive može počinjati cifrom? Da li se cifre mogu koristiti u okviru imena?*
3. *Koliko početnih karaktera u imenu promenljive u jeziku C se garantovano smatra bitnim?*

Pitanje 6.5.3.

1. *Deklarisati označenu karaktersku promenljivu pod imenom* `c` *i inicijalizovati je tako da sadrži kôd karaktera* `Y`.
2. *Deklarisati neoznačenu karaktersku promenljivu* `x`*, tako da ima inicijalnu vrednost jednaku kôdu karaktera* `;`.
3. *Deklarisati promenljivu koja je tipa neoznačeni dugi ceo broj i inicijalizovati tako da sadrži heksadekadnu vrednost* `FFFFFFFF`.

Pitanje 6.5.4.

1. *Navesti barem jedan tipa u jeziku C za koji je standardom definisano koliko bajtova zauzima.*
2. *Koja ograničenja važi za dužine u bajtovima tipova* `short int`*,* `int`*,* `long int`*?*
3. *Ako je opseg tipa* `int` *oko četiri milijarde, koliko je* `sizeof(int)`*?*
4. *Ukoliko je tip promenljive* `int`*, da li se podrazumeva da je njena vrednost označena ili neoznačena ili to standard ne propisuje?*
5. *U kojoj datoteci zaglavlja se nalaze podaci o opsezima celobrojevnih tipova za konkretnu implementaciju?*
6. *Na 32-bitnom sistemu, koliko bajtova zauzima podatak tipa:*

   (a) `char` *(b)* `int` *(c)* `short int` *(d)* `unsigned long`
7. *Koliko bajtova zauzima podatak tipa* `char`*? Ukoliko nije navedeno, da li se podrazumeva da je podatak ovog tipa označen ili neoznačen?*
8. *Koliko bajtova zauzima podatak tipa* `signed char`*? Koja je najmanja a koja najveća vrednost koju može da ima?*
9. *Koliko bajtova zauzima podatak tipa* `unsigned char`*? Koja je najmanja a koja najveća vrednost koju može da ima?*
10. *Koja ograničenja važi za dužine u bajtovima tipova* `float` *i* `double`*?*
11. *U kojoj datoteci zaglavlja se nalaze podaci o opsezima tipova broja u pokretnom zarezu za konkretnu implementaciju?*

Pitanje 6.5.5.

1. *Koja je razlika između konstanti* `3.4` *i* `3.4f`*? A između* `123` *i* `0123`*?*
2. *Kojeg su tipa i koje brojeve predstavljaju sledeće konstante?*

   (a) `1234` *(b)* `'.'` *(c)* `6423ul` *(d)* `12.3e-2` *(e)* `3.74e+2f` *(f)* `0x47` *(g)* `0543` *(h)* `3u` *(i)* `'0'`
3. *Kolika je vrednost konstantnog izraza* `0x20 + 020 + '2' - '0'` *?*

4. *Da li C omogućava direktan zapis binarnih konstanti? Koji je najkraći način da se celobrojna promenljiva inicijalizuje na binarni broj* `10111010110101001110011`0*?*

Pitanje 6.5.6. *Ako se na sistemu koristi ASCII tabela karaktera, šta ispisuje sledeći C kôd:*

```
char c1 = 'a'; char c2 = 97;
printf("%c %d %d %c", c1, c1, c2, c2);
```

Pitanje 6.5.7.

1. *Šta je to asocijativnost, a šta prioritet operatora?*
2. *Navesti neki operator jezika C koji ima levu i neki operator koji ima desnu asocijativnost.*
3. *Kakav je odnos prioriteta unarnih i binarnih operatora?*
4. *Kakav je odnos prioriteta logičkih, aritmetičkih i relacijskih operatora?*
5. *Kakav je odnos prioriteta operatora dodele i ternarnog operatora?*
6. *Da li dodele spadaju u grupu nisko ili visoko prioritetnih operatora? Koji operator jezika C ima najniži prioritet?*

Pitanje 6.5.8. *Imajući u vidu asocijativnost C operatora, šta je rezultat izvršavanja kôda:*

```
double a = 2.0, b = 4.0, c = -2.0;
printf("%f\n", (-b + sqrt(b*b - 4*a*c)) / 2.0 * a);
```

Kako popraviti kôd tako da ispravno izračunava rešenje kvadratne jednačine?

Pitanje 6.5.9. *Postaviti zagrade u naredne izraze u skladu sa podrazumevanim prioritetom i asocijativnosti operatora (na primer,* `3+4*5+7` *—* `(3+(4*5))+7`*).*
(a) `a = b = 4` *(b)* `a = 3 == 5` *(c)* `c = 3 == 5 + 7 <= 4`
(d) `3 - 4 / 2 < 3 && 4 + 5 * 6 <= 3 % 7 * 2`
(e) `a = b < c ? 3 * 5 : 4 < 7, 2` *(f)* `a = b + c && d != e`

Pitanje 6.5.10. *Koja je vrednost narednih izraza:*
(a) `3 / 4` *(b)* `3.0 / 4` *(c)* `14 % 3` *(d)* `3 >= 7`
(e) `3 && 7` *(f)* `3 || 0` *(g)* `a = 4` *(h)* `(3 < 4) ? 1 : 2`
(k) `3 < 5 < 5` *(i)* `3 < 7 < 5` *(j)* `6%3 || (6/3 + 5)`
(j) `3 < 5 ? 6 + 2 : 8 + 3`

Pitanje 6.5.11. *Da li su definisana deljenja* `1/0` *i* `1.0/0.0`*?*

Pitanje 6.5.12. *Da li su ispravni sledeći izrazi i ako jesu šta je njihova vrednost:*
(a) `2++` *(b)* `a++` *(c)* `2**` *(d)* `a**` *(e)* `2>>2` *(f)* `a>>2` *(g)* `a &&= 0` *(e)* `a ||= -7`

Pitanje 6.5.13.

1. *Ako pre naredbe* `a=b++;` *promenljiva* `a` *ima vrednost* `1`*, a promenljiva* `b` *ima vrednost* `3`*, koje vrednosti imaju nakon navedene naredbe?*
2. *Ako pre naredbe* `a+=b;` *promenljiva* `a` *ima vrednost* `1`*, a promenljiva* `b` *ima vrednost* `3`*, koje vrednosti imaju nakon navedene naredbe?*

3. *Kolika je vrednost promenljivih* `a`, `b`, `x` *i* `y` *nakon izvršavanja narednog kôda?*

```
int a = 1, b = 1, x, y;
x = a++; y = ++b;
```

4. *Koje vrednosti imaju promenljive* `a` *i* `b` *nakon izvršavanja kôda*
`int a=5; b = (a++ - 3) + (a++ - 3);`*?*

Pitanje 6.5.14.

1. *Šta znači to da se operatori* `&&`, `||` *i* `?:` *izračunavaju lenjo? Navesti primere.*
2. *Šta ispisuje naredni kôd?*

```
int a = 3, b = 0, c, d;
c = a % 2 || b++;
d = a % 3 || b++;
printf("%d %d %d %d", a, b, c, d);
```

3. *Koje vrednosti imaju promenljive* `a` *i* `b` *nakon izvršavanja narednog kôda?*
`int a=3; b = (a++ - 3) && (a++ - 3);`*?*
4. *Koju vrednost imaju promenljive* `a`, `b`, `c` *nakon izvršavanja narednog kôda?*
`int a, b=2, c; c = b++; a = c>b && c++;`
5. *Koje vrednosti imaju promenljive* `a` *i* `b` *nakon izvršenja narednog kôda?*
`unsigned char a, b=255; a = ++b && b++;`

Pitanje 6.5.15.

1. *Napisati izraz koji je tačan ako je godina* `g` *prestupna.*
2. *Napisati izraz čija je vrednost jednaka manjem od brojeva* `a` *i* `b`.
3. *Napisati izraz čija je vrednost jednaka apsolutnoj vrednosti broja* `a`.
4. *Napisati izraz koji je, na ASCII sistemu, tačan ako je karakter* `c` *malo slovo.*
5. *Napisati izraz koji je tačan ako je karakter* `c` *samoglasnik.*
6. *Napisati izraz čija je vrednost jednaka vrednosti implikacije* $\mathtt{p} \Rightarrow \mathtt{q}$.
7. *Napisati izraz koji ako je* `c` *malo slovo ima vrednost odgovarajućeg velikog slova, a inače ima vrednost* `c`.

Zadatak 6.5.1. *Napisati program koji za zadati sat, minut i sekund izračunava koliko je vremena (sati, minuta i sekundi) ostalo do ponoći. Program treba i da izvrši proveru da li su uneti podaci korektni.* √

Zadatak 6.5.2. *Napisati program koji za uneta tri pozitivna broja u pokretnom zarezu ispituje da li postoji trougao čije su dužine stranica upravo ti brojevi — uslov provere napisati u okviru jednog izraza.* √

Zadatak 6.5.3. *Napisati program koji izračunava zbir cifara unetog četvorocifrenog broja.* √

Zadatak 6.5.4. *Napisati program koji razmenjuje poslednje dve cifre unetog prirodnog broja. Na primer, za uneti broj* 1234 *program treba da ispiše* 1243.

Napisati program koji ciklično u levo rotira poslednje tri cifre unetog prirodnog broja. Na primer, za uneti broj 12345 *program treba da ispiše* 12453.
√

Zadatak 6.5.5. *Napisati program koji za uneti par prirodnih brojeva izračunava koji je po redu taj par u cik-cak nabrajanju datom na slici 3.1. Napisati i program koji za dati redni broj u cik-cak nabrajanju određuje odgovarajući par prirodnih brojeva.* √

Zadatak 6.5.6. *Napisati program koji za unete koeficijente A i B izračunava rešenje jednačine $Ax+B=0$. Program treba da prijavi ukoliko jednačina nema rešenja ili ima više od jednog rešenja.* √

Zadatak 6.5.7. *Napisati program koji za unete koeficijente A_1, B_1, C_1, A_2, B_2 i C_2 izračunava rešenje sistema jednačina $A_1x + B_1y = C_1$, $A_2x + B_2y = C_2$. Program treba da prijavi ukoliko sistem nema rešenja ili ima više od jednog rešenja.* √

Zadatak 6.5.8. *Napisati program koji za unete koeficijente a, b i c ($a \neq 0$) izračunava (do na mašinsku tačnost) i ispisuje realna rešenja kvadratne jednačine $ax^2 + bx + c = 0$.* √

Zadatak 6.5.9. *Napisati program koji ispisuje sve delioce unetog neoznačenog celog broja (na primer, za unos* `24` *treženi delioci su* `1, 2, 3, 4, 6, 8, 12` *i* `24`*).* √

Zadatak 6.5.10. *Napisati program učitava karaktere sa standardnog ulaza sve dok se ne unese tačka (karakter* `.`*), zapeta (karakter* `,`*) ili kraj datoteke (*`EOF`*) i prepisuje ih na standardni izlaz menjajući svako malo slovo velikim. (Napomena: za učitavanje karaktera koristiti funkciju* `getchar`*, a za ispis funkciju* `putchar`*)* √

6.6 Konverzije tipova

Konverzija tipova predstavlja pretvaranje vrednosti jednog tipa u vrednost drugog tipa. Jezik C je veoma fleksibilan po pitanju konverzije tipova i, u mnogim situacijama, dopušta korišćenje vrednosti jednog tipa tamo gde se očekuje vrednost drugog tipa (ponekad se kaže da je C *slabo tipiziran jezik*). Iako su konverzije često neophodne kako bi se osiguralo korektno funkcionisanje programa i omogućilo mešanje podataka različitih tipova u okviru istog programa, konverzije mogu dovesti i do gubitka podataka ili njihove loše interpretacije. Jezik C je statički tipiziran, pa, iako se prilikom konverzija konvertuju vrednosti izraza, promenljive sve vreme ostaju da budu onog tipa koji im je pridružen deklaracijom.

6.6.1 Promocija i democija

Postoje eksplicitne konverzije (koje se izvršavaju na zahtev programera) i implicitne konverzije (koje se izvršavaju automatski kada je to potrebno). Neke konverzije je moguće izvesti bez gubitka informacija, dok se u nekim slučajevima prilikom konverzije vrši izmena same vrednosti podatka.

Promocija (ili *napredovanje*) predstavlja konverziju vrednosti „manjeg tipa“ u vrednost „većeg tipa“ (na primer, `int` u `float` ili `float` u `double`) u kom slučaju najčešće ne dolazi do gubitka informacije.

```
float a = 4;  /* 4 je int pa se implicitno promovise u float */
```

Naglasimo da do gubitka informacija ipak može da dođe. Na primer, kôd

```
float f = 16777217;
printf("%f\n", f);
```

obično (tj. na velikom broju konkretnih sistema) ispisuje

```
16777216.00000
```

jer vrednost `16777217.0` ne može da se zapiše u okviru tipa `float` ako on koristi četiri bajta. Naime, tada tip `float` za zapis mantise koristi tri bajta, tj. dvadeset četiri binarne cifre i može da reprezentuje sve pozitivne celobrojne vrednosti do 2^{24} ali samo neke veće od 2^{24}. Broj 16777217 jednak je $2^{24} + 1$ i binarno se zapisuje kao 100000000000000000000001, pa prilikom konverzije u `float` (dužine četiri bajta) dolazi do gubitka informacija.

Democija (ili *nazadovanje*) predstavlja konverziju vrednosti većeg tipa u vrednost manjeg tipa (na primer, `long` u `short`, `double` u `int`). Prilikom democije, moguće je da dođe do gubitka informacije (u slučaju da se polazna vrednost ne može predstaviti u okviru novog tipa). U takvim slučajevima, kompilator obično izdaje upozorenje, ali ipak vrši konverziju.

```
int b = 7.0f; /* 7.0f je float pa se vrsi democija u 7 */
int c = 7.7f; /* 7.7f je float i vrsi se democija u 7,
                 pri cemu se gubi informacija */
unsigned char d = 256; /* d dobija vrednost 0 */
```

Prilikom konverzije iz brojeva u pokretnom zarezu u celobrojne tipove podataka i obratno potrebno je potpuno izmeniti interni zapis podataka (na primer, iz IEEE754 zapisa brojeva u pokretnom zarezu u zapis u obliku potpunog komplementa). Ovo se najčešće vrši uz „odsecanje decimala“, a ne zaokruživanjem na najbliži ceo broj (tako je `7.7f` konvertovano u `7`, a ne u `8`). Prilikom konverzija celobrojnih tipova istog internog zapisa različite širine, vrši se odsecanje vodećih bitova zapisa (u slučaju democija) ili proširivanje zapisa dodavanjem vodećih bitova (u slučaju promocija). U slučaju da je polazni tip neoznačen, promocija se vrši dodavanjem vodećih nula (engl. zero extension). U slučaju da je polazni tip označen, promocija se vrši proširivanjem vodećeg bita (engl. *sign extension*) i na taj način se zadržava vrednost i pozitivnih i negativnih brojeva zapisanih u potpunom komplementu.

Navedeni primer `int c = 7.7f;` ilustruje i zašto vrednost izraza dodele nije jednaka desnoj strani (u ovom slučaju `7.7f`), nego vrednosti koja je dodeljena objektu na levoj strani operatora dodele (u ovom slučaju `7`).

Česta implicitna konverzija je konverzija tipa `char` u `int`. S obzirom na to da standard ne definiše da li je tip `char` označen ili neoznačen, rezultat konverzije se razlikuje od implementacije do implementacije (ukoliko je karakter neoznačen, prilikom proširivanja dopisuju mu se vodeće nule, a ukoliko je označen, dopisuje se vrednost njegovog vodećeg bita), što može dovesti do problema prilikom prenošenja programa sa jedne na drugu platformu. Ipak, standard garantuje da će svi karakteri koji mogu da se štampaju (engl. printable characters) uvek imati pozitivne kôdove (i vodeći bit 0), tako da oni ne predstavljaju problem prilikom konverzija (proširivanje se tada uvek vrši nulama). Ukoliko se tip `char` koristi za smeštanje nečeg drugog osim karaktera koji mogu da se štampaju, preporučuje se eksplicitno navođenje kvalifikatora `signed` ili `unsigned` kako bi se programi izvršavali identično na svim računarima.

Sve brojevne vrednosti (i cele i u pokretnom zarezu) prevode se u istinitosne vrednosti tako što se 0 prevodi u 0, a sve ne-nula vrednosti se prevode u 1.

6.6.2 Eksplicitne konverzije

Operator eksplicitne konverzije tipa ili operator *kastovanja* (engl. type cast operator) se navodi tako što se ime rezultujućeg tipa navodi u malim zagradama ispred izraza koji se konvertuje:

```
(tip)izraz
```

Operator kastovanja je prefiksni, unaran operator i ima viši prioritet od svih binarnih operatora. U slučaju primene operatora kastovanja na promenljivu, vrednost izraza je vrednost promenljive konvertovana u traženi tip, a vrednost same promenljive se ne menja (i, naravno, ne menja se njen tip).

Eksplicitna konverzija može biti neophodna u različitim situacijama. Na primer, ukoliko se na celobrojne operande želi primena deljenja brojeva u pokretnom zarezu:

```
int a = 13, b = 4;
printf("%d\t", a/b);
printf("%f\n", (double)a/(double)b);
```

```
3       3.250000
```

Prilikom prvog poziva funkcije `printf` upotrebljen je format `%d`, zbog toga što je izraz `a/b` celobrojnog tipa – navođenje `%f` umesto `%d`, naravno, ne utiče na njegov tip i rezultat i ne bi imalo smisla.

U nastavku će biti prikazano da je, u navedenom primeru, bilo dovoljno konvertovati i samo jedan od operanada u tip `double` i u tom slučaju bi se drugi operand implicitno konvertovao.

6.6.3 Konverzije pri dodelama

Prilikom primene operatora dodele vrši se implicitna konverzija vrednosti desne strane dodele u tip leve strane dodele, pri čemu se za tip dodele uzima tip leve strane. Na primer,

```
int a;
double b = (a = 3.5);
```

U navedenom primeru, prilikom dodele promenljivoj `a`, vrši se democija `double` konstante 3.5 u vrednost 3 tipa `int`, a zatim se, prilikom dodele promenljivoj `b`, vrši promocija te vrednosti u `double` vrednost 3.0.

6.6.4 Implicitne konverzije u aritmetičkim izrazima

Prilikom primene nekih operatora vrše se implicitne konverzije (uglavnom promocije) koje obezbeđuju da operandi postanu istog tipa pogodnog za primenu operacija. Ove konverzije se nazivaju *uobičajene aritmetičke konverzije* (engl. *usual arithmetic conversions*). Ove konverziju se, na primer, primenjuju prilikom primene aritmetičkih (`+`, `-`, `*`, `/`) i relacijskih binarnih operatora (`<`, `>`, `<=`, `>=`, `==`, `!=`), prilikom primene uslovnog operatora `?:`.

Aritmetički operatori se ne primenjuju na „male“ tipove tj. na podatke tipa `char` i `short` (zbog toga što je u tim slučajevima verovatno da će doći do prekoračenja tj. da rezultat neće moći da se zapiše u okviru malog tipa), već se pre primene operatora mali tipovi promovišu u tip `int`. Ovo se naziva *celobrojna promocija* (engl. integer promotion)[11].

```
unsigned char cresult, c1, c2, c3;
c1 = 100;
c2 = 3;
c3 = 4;
cresult = c1 * c2 / c3;
```

U navedenom primeru, vrši se promocija promenljivih `c1`, `c2` i `c3` u `int` pre izvođenja operacija, a zatim se vrši democija rezultata prilikom upisa u promenljivu `cresult`. Kada se promocija ne bi vršila, množenje `c1 * c2` bi dovelo do prekoračenja jer se vrednost 300 ne može predstaviti u okviru tipa `char` i rezultat ne bi bio korektan (tj. ne bi bila dobijena vrednost $(100 \cdot 3)/4$). Pošto se promocija vrši, biće dobijen korektan rezultat.

Generalno, prilikom primene aritmetičkih operacija primenjuju se sledeća pravila konverzije:

1. Ako je bar jedan od operanada tipa `long double` i drugi se promoviše u `long double`;
2. inače, ako je bar jedan od operanada tipa `double` i drugi se promoviše u `double`;
3. inače, ako je bar jedan od operanada tipa `float` i drugi se promoviše u `float`;
4. inače, svi operandi tipa `char` i `short` se promovišu u `int`.
5. ako je jedan od operanada tipa `long long` i drugi se prevodi u `long long`;
6. inače, ako je jedan od operanada tipa `long` i drugi se prevodi u `long`.

U slučaju korišćenja neoznačenih operanada (tj. mešanja označenih i neoznačenih operanada), pravila konverzije su nešto komplikovanija:

[11] Celobrojna promocija ima jednostavno opravdanje ako se razmotri prevedeni mašinski program. Naime, tip `int` odgovara širini registara u procesoru tako da se konverzija malih vrednosti vrši jednostavnim upisivanjem u registre. Procesori obično nemaju mašinske instrukcije za rad sa kraćim tipovima podataka i operacije nad kraćim tipovima podataka (slučaj u kojem se ne bi vršila celobrojna promocija) zapravo bi bilo teže ostvariti.

1. Ako je neoznačeni operand širi[12] od označenog, označeni operand se konvertuje u neoznačeni širi tip;
2. inače, ako je tip označenog operanda takav da može da predstavi sve vrednosti neoznačenog tipa, tada se neoznačeni tip prevodi u širi, označeni.
3. inače, oba operanda se konvertuju u neoznačeni tip koji odgovara označenom tipu.

Problemi najčešće nastaju prilikom poređenja vrednosti različitih tipova. Na primer, ako `int` zauzima 16 bitova, a `long` 32 bita, tada je `-1l < 1u`. Zaista, u ovom primeru porede se vrednosti tipa `signed long` i `unsigned int`. Pošto `signed long` može da predstavi sve vrednosti tipa `unsigned int`, vrši se konverzija oba operanda u `signed long` i vrši se poređenje. S druge strane, važi da je `-1l > 1ul`. U ovom slučaju porede se vrednosti tipa `signed long` i `unsigned long`. Pošto tip `signed long` ne može da predstavi sve vrednosti tipa `unsigned long`, oba operanda se konvertuju u `unsigned long`. Tada se `-1l` konvertuje u `ULONG_MAX` (najveći neoznačen broj koji se može zapisati u 32 bita), dok `1ul` ostaje neizmenjen i vrši se poređenje.

Ukoliko se želi postići uobičajeni poredak brojeva, bez obzira na širinu tipova na prenosiv način, poređenje je poželjno izvršiti na sledeći način:

```
signed   si = /* neka vrednost */;
unsigned ui = /* neka vrednost */;

/*  if (si < ui) - ne daje uvek korektan rezultat */
if (si < 0 || (unsigned)si < ui) {
  ...
}
```

Pitanja i zadaci za vežbu

Pitanje 6.6.1. *Šta su to implicitne, a šta eksplicitne konverzije? Na koji način se vrednost karakterske promenljive* `c` *se može eksplicitno konvertovati u celobrojnu vrednost?*

Pitanje 6.6.2. *Šta su to promocije, a šta democije? Koje od narednih konverzija su promocije, a koje democije: (a) int u short, (b) char u float, (c) double u long, (d) long u int?*

Pitanje 6.6.3. *Navesti pravila koja se primenjuju prilikom primene aritmetičkih operatora.*

Pitanje 6.6.4. *Koje vrednosti imaju promenljive* `e` *i* `f` *nakon kôda*
`unsigned char c = 255, d = 1, e = c + d; int f = c + d;`*?*
Koje vrednosti imaju promenljive `e`*,* `g` *i* `f` *nakon kôda*
`unsigned char c = 255, d = 1; signed char e = c;`
`char f = c + d; int g = c + d;`*?*

[12] Preciznije, standard uvodi pojam konverzionog ranga (koji raste od `char` ka `long long`) i u ovoj situaciji se razmatra konverzioni rang.

Pitanje 6.6.5. *Ako aritmetički operator ima dva argumenta, a nijedan od njih nije ni tipa* `long double`*, ni tipa* `double`*, ni tipa* `float`*, u koji tip se implicitno konvertuju argumenti tipa* `char` *i* `short int`*?*

Pitanje 6.6.6.

1. *Koju vrednost ima promenljiva* `x` *tipa* `float` *nakon naredbe* `x = (x = 3/2);`*?*
2. *Ako promenljiva* `c` *ima tip* `char`*, koji tip će ona imati nakon dodele* `c = 1.5 * c;`*?*
3. *Koju vrednost ima promenljiva* `x` *tipa* `float` *nakon naredbe* `x = 3/2 + (double)3/2 + 3.0/2;`*?*
4. *Ako je promenljiva* `f` *tipa* `float`*, a promenljiva* `a` *tipa* `int`*, koju vrednost ima* `f` *nakon naredbe:* `f = (a = 1/2 < 0.5)/2;`*?*

Pitanje 6.6.7. *Koje vrednosti imaju promenljive* `i` *i* `c` *nakon naredbi*

```
char c, c1 = 130, c2 = 2; int i; i = (c1*c2)/4; c = (c1*c2)/4;
```

Pitanje 6.6.8. *Da li nakon naredbe* `tip c1, c2, c = (c1*c2)/c2;` *promenljiva* `c` *nužno ima vrednost promenljive* `c1` *ako je*

- `tip` *tip* `char`*,*
- `tip` *tip* `int`*,*
- `tip` *tip* `float`*?*

6.7 Nizovi i niske

Često je u programima potrebno korišćenje velikog broja srodnih promenljivih. Umesto velikog broja pojedinačno deklarisanih promenljivih, moguće je koristiti *nizove*. Obrada elemenata nizova se onda obično vrši na uniforman način, korišćenjem petlji.

6.7.1 Primer korišćenja nizova

Razmotrimo, kao ilustraciju, program koji učitava 10 brojeva sa standardnog ulaza, a zatim ih ispisuje u obratnom poretku. Bez nizova, neophodno je koristiti deset promenljivih. Da se u zadatku tražilo ispisivanje unatrag 1000 brojeva, program bi bio izrazito mukotrpan za pisanje i nepregledan.

Program 6.4.

```
#include <stdio.h>

int main() {
  int b0, b1, b2, b3, b4, b5, b6, b7, b8, b9;
  scanf("%d%d%d%d%d%d%d%d%d%d",
        &b0, &b1, &b2, &b3, &b4, &b5, &b6, &b7, &b8, &b9);
  printf("%d %d %d %d %d %d %d %d %d %d",
        b9, b8, b7, b6, b5, b4, b3, b2, b1, b0);
}
```

```
1 2 3 4 5 6 7 8 9 10
10 9 8 7 6 5 4 3 2 1
```

Umesto 10 različitih srodnih promenljivih, moguće je upotrebiti niz i time znatno uprostiti prethodni program.

Program 6.5.

```
#include <stdio.h>

int main() {
  int b[10], i;
  for (i = 0; i < 10; i++)
    scanf("%d", &b[i]);
  for (i = 9; i >= 0; i--)
    printf("%d ", b[i]);
  return 0;
}
```

6.7.2 Deklaracija niza

Nizovi u programskom jeziku C mogu se deklarisati na sledeći način:

```
tip ime-niza[broj-elemenata];
```

Na primer, deklaracija

```
int a[10];
```

uvodi niz `a` od 10 celih brojeva. Prvi element niza ima indeks 0, pa su elementi niza:

```
a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7], a[8], a[9]
```

Prilikom deklaracije može se izvršiti i inicijalizacija:

```
int a[5] = { 1, 2, 3, 4, 5 };
```

Nakon ove deklaracije, sadržaj niza `a` jednak je:

1	2	3	4	5

Veličina memorijskog prostora potrebnog za niz određuje se u fazi prevođenja programa, pa broj elemenata niza (koji se navodi u deklaraciji) mora biti konstantan izraz. U narednom primeru, deklaracija niza `a` je ispravna, dok su deklaracije nizova `b` i `c` neispravne[13]:

```
int x;
char a[100+10];
int b[];
float c[x];
```

[13]Standardi nakon C99 uvode pojam niza promenljive dužine (engl. variable length array, VLA). U tom svetlu, deklaracija niza `c` bila bi ispravna. Ipak, u ovoj knjizi, VLA neće biti razmatrani.

Ako se koristi inicijalizacija, moguće je navesti veću dimenziju od broja navedenih elemenata (kada se inicijalizuju samo početni elementi deklarisanog niza). Neispravno je navođenje manje dimenzije od broja elemenata inicijalizatora.

Dimenziju niza je moguće izostaviti samo ako je prilikom deklaracije izvršena i inicijalizacija niza i tada se dimenzija određuje na osnovu broja elemenata u inicijalizatoru. Na primer, nakon deklaracije

```
int b[] = { 1, 2, 3 };
```

sadržaj niza `b` jednak je:

1	2	3

Ukoliko dimenzija niza nije zadata eksplicitno već je implicitno određena inicijalizacijom, broj elemenata može se izračunati na sledeći način:

```
sizeof(ime niza)/sizeof(tip elementa niza)
```

Kada se pristupa elementu niza, indeks može da bude proizvoljan izraz celobrojne vrednosti, na primer:

```
a[2*2+1]
a[i+2]
```

U fazi prevođenja (pa ni u fazi izvršavanja[14]) ne vrši se nikakva provera da li je indeks pristupa nizu u njegovim granicama i moguće je bez ikakve prijave greške ili upozorenja od strane prevodioca pristupati i lokaciji koji se nalazi van opsega deklarisanog niza (na primer, moguće je koristiti element `a[13]`, pa čak element `a[-1]` u prethodnom primeru). Ovo najčešće dovodi do fatalnih grešaka prilikom izvršavanja programa. S druge strane, ovakva (jednostavna) politika upravljanja nizovima omogućava veću efikasnost programa.

Pojedinačni elementi nizova su l-vrednosti (tj. moguće im je dodeljivati vrednosti). Međutim, nizovi nisu l-vrednosti i nije im moguće dodeljivati vrednosti niti ih menjati. To ilustruje sledeći primer:

```
int a[3] = {5, 3, 7};
int b[3];
b = a;      /* Neispravno - nizovi se ne mogu dodeljivati. */
a++;        /* Neispravno - nizovi se ne mogu menjati. */
```

6.7.3 Niske

Posebno mesto u programskom jeziku C zauzimaju nizovi koji sadrže karaktere — *niske karaktera* ili kraće *niske* (engl. *strings*). Niske se navode između dvostrukih navodnika `""` (na primer, `"ja sam niska"`). Specijalni karakteri se u niskama navode korišćenjem specijalnih sekvenci (na primer, `"prvi red\ndrugi red"`). Niske su interno reprezentovane kao nizovi karaktera na čiji desni kraj se dopisuje karakter `'\0'`, tzv. završna, terminalna nula

[14] U fazi izvršavanja, operativni sistem obično proverava da li se pokušava upis van memorije koja je dodeljena programu i ako je to slučaj, obično nasilno prekida izvršavanje programa (npr. uz poruku `segmentation fault`). O ovakvim greškama biće više reči u poglavlju 9.3.5.

(engl. *null terminator*). Zbog toga, niske u C-u se nazivaju *niske završene nulom* (engl. *null terminated strings*)[15]. Posledica ovoga je da ne postoji ograničenje za dužinu niske, ali je neophodno proći kroz celu nisku kako bi se odredila njena dužina.

Niske mogu da se koriste i prilikom inicijalizacije nizova karaktera.

```
char s1[] = {'Z', 'd', 'r', 'a', 'v', 'o'};
char s2[] = {'Z', 'd', 'r', 'a', 'v', 'o', '\0'};
char s3[] = "Zdravo";
```

Nakon navedenih deklaracija, sadržaj niza `s1` jednak je:

0	1	2	3	4	5
'Z'	'd'	'r'	'a'	'v'	'o'

a sadržaj nizova `s2` i `s3` jednak je:

0	1	2	3	4	5	6
'Z'	'd'	'r'	'a'	'v'	'o'	'\0'

Niz `s1` sadrži 6 karaktera (i zauzima 6 bajtova). Deklaracije za `s2` i `s3` su ekvivalentne i ovi nizovi sadrže po 7 karaktera (i zauzimaju po 7 bajtova). Prvi niz karaktera nije terminisan nulom i ne može se smatrati ispravnom niskom. Potrebno je jasno razlikovati karakterske konstante (na primer, `'x'`) koje predstavljaju pojedinačne karaktere i niske (na primer, `"x"` — koja sadrži dva karaktera, `'x'` i `'\0'`).

Ukoliko se u programu niske nalaze neposredno jedna uz drugu, one se automatski spajaju. Na primer:

```
printf("Zdravo, " "svima");
```

je ekvivaletno sa

```
printf("Zdravo, svima");
```

Standardna biblioteka definiše veliki broj funkcija za rad sa niskama. Prototipovi ovih funkcija se nalaze u zaglavlju `string.h`. Na primer, funkcija `strlen` služi za izračunavanje dužine niske. Pri računanju dužine niske, ne računa se završna nula. Ilustracije radi, jedan od načina da se neposredno, bez pozivanja neke funkcije (naravno, uvek je preporučljivo pozvati bibliotečku funkciju ukoliko je dostupna), izračuna dužina niske `s` dat je sledećim kôdom, nakon čijeg izvršavanja promenljiva `i` sadrži traženu dužinu:

```
int i = 0;
while (s[i] != '\0')
  i++;
```

Kôd koji obrađuju niske obično ih obrađuje karakter po karakter i obično sadrži petlju oblika:

```
while (s[i] != '\0')
  ...
```

ili oblika:

[15] Neki programski jezici koriste drugačije konvencije za interni zapis niski. Na primer, u Pascal-u se pre samog sadržaja niske zapisuju broj koji predstavlja njenu dužinu (tzv. P-niske).

```
for (i = 0; s[i] != '\0'; i++)
   ...
```

Kako završna nula ima numeričku vrednost 0 (što predstavlja istinitosnu vrednost *netačno*), poređenje u prethodnoj petlji može da se izostavi:

```
for (i = 0; s[i]; i++)
   ...
```

Izuzetno neefikasan kôd dobija se pozivanjem funkcije `strlen` za izračunavanje dužine niske u svakom koraku iteracije:

```
for (i = 0; i < strlen(s); i++)
   ...
```

Još jedna funkcija standardne biblioteke za rad sa niskama je funkcija `strcpy` koja, kada se pozove sa `strcpy(dest, src)`, vrši kopiranje niske `src` u nisku `dest`, pretpostavljajući da je niska kojoj se dodeljuje vrednost dovoljno velika da primi nisku koja se dodeljuje. Ukoliko ovo nije ispunjeno, vrši se promena sadržaja memorijskih lokacija koje su van dimenzija niza što može dovesti do fatalnih grešaka prilikom izvršavanja programa. Ilustracije radi, navedimo kako se kopiranje niski može izvesti neposredno. U nisku `dest` se prebacuje karakter po karakter niske `src` sve dok dodeljeni karakter ne bude završna nula:

```
int i = 0;
while ((dest[i] = src[i]) != '\0')
   i++;
```

Poređenje različitosti sa terminalnom nulom se može izostaviti, a petlja se, može preformulisati kao `for` petlja:

```
int i;
for (i = 0; dest[i] = src[i]; i++)
    ;
```

6.7.4 Višedimenzioni nizovi

Pored jednodimenzionih mogu se koristiti i višedimenzioni nizovi, koji se deklarišu na sledeći opšti način:

```
tip ime-niza[broj-elemenata]...[broj-elemenata];
```

Dvodimenzioni nizovi (matrice) tumače se kao jednodimenzioni nizovi čiji su elementi nizovi. Zato se elementima dvodimenzionalnog niza pristupa sa:

```
ime-niza[vrsta][kolona]
```

a ne sa `ime-niza[vrsta, kolona]` (greške do kojih može doći zbog pokušaja ovakvog pristupa već su napomenute u kontekstu operatora zarez (`,`), u poglavlju 6.5).

Elementi se u memoriji smeštaju po vrstama pa se, kada se elementima pristupa u redosledu po kojem su smešteni u memoriji, najbrže menja poslednji indeks. Niz se može inicijalizovati navođenjem liste inicijalizatora u vitičastim

zagradama; pošto su elementi opet nizovi, svaki od njih se opet navodi u okviru vitičastih zagrada (mada je unutrašnje vitičaste zagrade moguće i izostaviti). Razmotrimo, kao primer, jedan dvodimenzioni niz:

```
char a[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

Kao i u slučaju jednodimenzionih nizova, ako je naveden inicijalizator, vrednost prvog indeksa moguće je i izostaviti (jer se on u fazi kompilacije može odrediti na osnovu broja inicijalizatora):

```
char a[][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

U memoriji su elementi poređani na sledeći način: `a[0][0]`, `a[0][1]`, `a[0][2]`, `a[1][0]`, `a[1][1]`, `a[1][2]`, tj. vrednosti elemenata niza poređane su na sledeći način: `1`, `2`, `3`, `4`, `5`, `6`. U ovom primeru, element `a[x][y]` je k-ti po redu, pri čemu je k jednako `3*x+y`. Pozicija elementa višedimenzionog niza može se slično izračunati i slučaju nizova sa tri i više dimenzija.[16]

Razmotrimo, kao dodatni primer, dvodimenzioni niz koji sadrži broj dana za svaki mesec, pri čemu su u prvoj vrsti vrednosti za obične, a u drugoj vrsti za prestupne godine:

```
char broj_dana[][13] = {
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};
```

Za skladištenje malih prirodnih brojeva koristi se tip `char`, a u nultu kolonu su upisane nule kako bi se podaci za mesec `m` nalazili upravo u koloni `m` (tj. kako bi se mesecima pristupalo korišćenjem indeksa 1-12, umesto sa indeksa 0-11). Niz `broj_dana` je moguće koristiti kako bi se, na primer, promenljiva `bd` postavila na broj dana za mesec `mesec` i godinu `godina`:

```
int prestupna = (godina % 4 == 0 && godina % 100 != 0) ||
                godina % 400 == 0;
char bd = broj_dana[prestupna][mesec];
```

Vrednost logičkog izraza je uvek nula (netačno) ili jedan (tačno), pa se vrednost `prestupna` može koristiti kao indeks pri pristupanju nizu.

[16] Nekada programeri ovu tehniku izračunavanja pozicija eksplicitno koriste da matricu smeste u jednodimenzioni niz, ali, pošto jezik dopušta korišćenje višedimenzionih nizova, za ovim nema potrebe.

Pitanja i zadaci za vežbu

Pitanje 6.7.1.

1. *Napisati deklaraciju sa inicijalizacijom niza* `a` *tipa* `int` *koji za elemente ima neparne brojeve manje od 10.*
2. *Navesti primer inicijalizacije niza tipa* `char` *tako da mu sadržaj bude* `zdravo` *i to: (i) navođenjem pojedinačnih karaktera između vitičastih zagrada i (ii) konstantom niskom.*

Pitanje 6.7.2.

1. *Ukoliko je niz* `a` *deklarisan kao* `float a[10];`*, koja je vrednost izraza* `sizeof(a)`*?*
2. *Ako je niz* `a` *tipa* `float` *inicijalizovan u okviru deklaracije i njegova dimenzija nije navedena, kako se ona može izračunati?*
3. *Kada je u deklaraciji niza dozvoljeno izostaviti njegovu dimenziju?*

Pitanje 6.7.3. *Precrtati sve neispravne linije narednog kôda.*

```
int a[];
int b[] = {1, 2, 3};
int c[5] = {1, 2, 3};
int d[2] = {1, 2, 3};
c = b;
b++;
```

Pitanje 6.7.4. *Deklarisati i inicijalizovati dvodimenzioni niz koji sadrži matricu*

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}.$$

Deklarisati i inicijalizovati dvodimenzioni niz koji sadrži matricu

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}.$$

Zadatak 6.7.1. *Napisati program koji unosi prirodan broj n ($n < 1000$), a zatim i n elemenata niza celih brojeva, nakon čega:*

1. *ispisuje unete brojeve unatrag;*
2. *ispisuje najveći element unetog niza;*
3. *ispisuje zbir elemenata unetog niza.*

√

Zadatak 6.7.2. *Sa standardnog ulaza se učitava tekst sve dok se ne unese tačka (tj. karakter .). Napisati program koji u unetom tekstu prebrojava pojavljivanje svake od cifara (broj pojavljivanja smestiti u niz od 10 elemenata).* √

Zadatak 6.7.3. *Napisati program koji za uneti datum u obliku (dan, mesec, godina) određuje koji je to po redu dan u godini. Proveriti i koretknost unetog datuma. U obzir uzeti i prestupne godine. Koristiti niz koji sadrži broj dana za svaki mesec i uporediti sa rešenjem zadatka bez korišćenja nizova.* √

Zadatak 6.7.4. *Paskalov trougao sadrži binomne koeficijente* $\binom{n}{k}$. *Napisati program koji ispisuje prvih m redova trougla* ($m < 100$). *Na primer, za* $m = 6$ *ispisuje se:*

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
```

(primetite da su ivice trougla 1 tj. da važi $\binom{n}{0} = \binom{n}{n} = 1$, *kao da se svaki unutrašnji član trougla može dobiti kao zbir odgovarajuća dva člana prethodne vrste, tj. da važi da je* $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$*).* √

Zadatak 6.7.5. *Napisati program koji sa standardnog ulaza unosi prvo dimenziju matrice* ($n < 10$) *pa zatim elemente matrice. Na primer,*

```
4
1 2 3 4
5 6 7 8
9 1 2 3
4 5 6 7
```

Program izračunava i ispisuje sumu elemenata glavne dijagonale i sumu elemenata iznad sporedne dijagonale matrice. Suma elemenata glavne dijagonale matrice navedene u primeru je $1 + 6 + 2 + 7 = 16$, *a iznad sporedne dijagonale je* $1 + 2 + 3 + 5 + 6 + 9 = 26$ √

Zadatak 6.7.6. *Napisati program koji sa standardnog ulaza unosi prvo dimenziju matrice* ($n < 100$) *pa zatim elemente matrice i zatim proverava da li je matrica donje-trougaona. Matrica je donje-trougaona ako se u gornjem trouglu (iznad glavne dijagonale, ne uključujući je) nalaze sve nule. Na primer:*

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 5 & 6 & 0 & 0 \\ 9 & 1 & 2 & 0 \\ 4 & 5 & 6 & 7 \end{pmatrix}$$

√

6.8 Korisnički definisani tipovi

Programski jezik C ima svega nekoliko ugrađenih osnovnih tipova. Već nizovi i niske predstavljaju složene tipove podataka. Osim nizova, postoji još nekoliko načina izgradnje složenih tipova podataka. Promenljive se mogu organizovati u *strukture* (tj. *slogove*), pogodne za specifične potrebe. Na taj način se

povezane vrednosti (ne nužno istog tipa) tretiraju kao jedna celina i za razliku od nizova gde se pristup pojedinačnim vrednostima vrši na osnovu brojevnog indeksa, pristup pojedinačnim vrednostima vrši se na osnovu imena polja strukture. Pored struktura, mogu se koristiti *unije*, koje su slične strukturama, ali kod kojih se jedan isti memorijski prostor koristi za više promenljivih. Mogu se koristiti i nabrojivi tipovi, sa konačnim skupom vrednosti. Pored definisanja novih tipova, već definisanim tipovima se može pridružiti i novo ime.

6.8.1 Strukture

Osnovni tipovi jezika C često nisu dovoljni za pogodno opisivanje svih podataka u programu. Ukoliko je neki podatak složene prirode tj. sastoji se od više delova, ti njegovi pojedinačni delovi mogu se čuvati nezavisno (u zasebnim promenljivama), ali to često vodi programima koji su nejasni i teški za održavanje. Umesto toga, pogodnije je koristiti *strukture*. Za razliku od nizova koji objedinjuju jednu ili više promenljivh istog tipa, struktura objedinjuje jednu ili više promenljivih, ne nužno istih tipova. Definisanjem strukture uvodi se novi tip podataka i nakon toga mogu da se koriste promenljive tog novog tipa, na isti način kao i za druge tipove.

Korišćenje struktura biće ilustrovano na primeru razlomaka. U jeziku C ne postoji tip koji opisuje razlomke, ali može se definisati struktura koja opisuje razlomke. Razlomak može da bude opisan parom koji čine brojilac i imenilac, na primer, celobrojnog tipa. Brojilac (svakog) razlomka zvaće se `brojilac`, a imenilac (svakog) razlomka zvaće se `imenilac`. Struktura `razlomak` može se definisati na sledeći način:

```
struct razlomak {
    int brojilac;
    int imenilac;
};
```

Ključna reč `struct` započinje definiciju strukture. Nakon nje, navodi se ime strukture, a zatim, između vitičastih zagrada, opis njenih *članova* (ili *polja*). Imena članova strukture se ne mogu koristiti kao samostalne promenljive, one postoje samo kao deo složenijeg objekta. Prethodnom definicijom strukture uveden je samo novi tip pod imenom `struct razlomak`, ali ne i promenljive tog tipa.

Strukture mogu sadržati promenljive proizvoljnog tipa. Na primer, moguće je definisati strukturu koja sadrži i niz.

```
struct student {
  char ime[50];
  float prosek;
};
```

Strukture mogu biti ugnježdene, tj. članovi struktura mogu biti druge strukture. Na primer:

```
struct dvojni_razlomak {
    struct razlomak gore;
    struct razlomak dole;
};
```

Definicija strukture uvodi novi tip i nakon nje se ovaj tip može koristiti kao i bilo koji drugi. Definicija strukture se obično navodi van svih funkcija. Ukoliko je navedena u okviru funkcije, onda se može koristiti samo u okviru te funkcije. Prilikom deklarisanja promenljivih ovog tipa, kao deo imena tipa, neophodno je korišćenje ključne reči `struct`, na primer:

```
struct razlomak a, b, c;
```

Definicijom strukture je opisano da se razlomci sastoje od brojioca i imenioca, dok se navedenom deklaracijom uvode tri konkretna razlomka koja se nazivaju `a`, `b` i `c`.

Moguća je i deklaracija sa inicijalizacijom, pri čemu se inicijalne vrednosti za članove strukture navode između vitičastih zagrada:

```
struct razlomak a = { 1, 2 };
```

Redosled navođenja inicijalizatora odgovara redosledu navođenja članova strukture. Dakle, navedenom deklaracijom je uveden razlomak `a` čiji je brojilac 1, a imenilac 2.

Definisanje strukture i deklarisanje i inicijalizacija promenljivih može se uraditi istovremeno (otuda i neuobičajeni simbol `;` nakon zatvorene vitičaste zagrade prilikom definisanja strukture):

```
struct razlomak {
  int brojilac;
  int imenilac;
} a = {1, 2}, b, c;
```

Članu strukture se pristupa preko imena promenljive (čiji je tip struktura) iza kojeg se navodi tačka a onda ime člana, na primer:

```
a.imenilac
```

Na primer, vrednost promenljive `a` tipa `struct razlomak` može biti ispisan na sledeći način:

```
printf("%d/%d", a.brojilac, a.imenilac);
```

Naglasimo da je operator `.`, iako binaran, operator najvišeg prioriteta (istog nivoa kao male zagrade i unarni postfiksni operatori).

Ne postoji konflikt između imena polja strukture i istoimenih promenljivih, pa je naredni kôd korektan.

```
  int brojilac = 5, imenilac = 3;
  a.brojilac = brojilac; a.imenilac = imenilac;
```

Od ranije prikazanih operacija, nad promenljivama tipa strukture dozvoljene su operacije dodele a nisu dozvoljeni aritmetički i relacioni operatori.

Nizovi struktura

Često postoji povezana skupina složenih podataka. Umesto da se oni čuvaju u nezavisnim nizovima (što bi vodilo programima teškim za održavanje) bolje je koristiti nizove struktura. Na primer, ako je potrebno imati podatke o imenima i broju dana meseci u godini, moguće je te podatke čuvati u nizu sa brojevima dana i u (nezavisnom) nizu imena meseci. Bolje je, međutim, opisati strukturu mesec koja sadrži broj dana i ime:

```
struct opis_meseca {
    char ime[10];
    int broj_dana;
};
```

i koristiti niz ovakvih struktura:

```
struct opis_meseca meseci[13];
```

(deklarisan je niz dužine 13 da bi se meseci mogli referisati po svojim rednim brojevima, pri čemu se početni element niza ne koristi).

Moguća je i deklaracija sa inicijalizacijom (u kojoj nije neophodno navođenje broja elemenata niza)[17]:

```
struct opis_meseca meseci[] = {
  { "",0 },
  { "januar",31 },
  { "februar",28 },
  { "mart",31 },
  ...
  { "decembar",31 }
};
```

U navednoj inicijalizaciji unutrašnje vitičaste zagrade je moguće izostaviti:

```
struct opis_meseca meseci[] = {
  "",0,
  "januar",31,
  "februar",28,
  "mart",31,
  ...
  "decembar",31
};
```

Nakon navedene deklaracije, ime prvog meseca u godini se može dobiti sa `meseci[1].ime`, njegov broj dana sa `meseci[1].broj_dana` itd.

Kao i obično, broj elemenata ovako inicijalizovanog niza može se izračunati na sledeći način:

```
sizeof(meseci)/sizeof(struct opis_meseca)
```

[17] U ovom primeru se zanemaruje činjenica da februar može imati i 29 dana.

6.8.2 Unije

Unije su donekle slične strukturama. One objedinjuju nekoliko promenljivih koje ne moraju imati isti tip. No, za razliku od struktura kod kojih se sva polja u strukturi istovremeno popunjavaju i sva se mogu istovremeno koristiti, kod unije se u jednom trenutku može koristiti samo jedno polje. Veličina strukture odgovara zbiru veličina njenih polja, dok veličina unije odgovara veličini njenog najvećeg polja. Osnovna svrha unija je ušteda memorije.

Skoro sva sintaksička pravila koja se odnose na strukutre se prenose i na unije (jedina razlika je da se umesto ključne reči `struct` koristi ključna reč `union`).

Naredna unija omogućava korisniku da izabere da li će vreme da pamti kao ceo broj sekundi ili kao razlomljen broj (koji obuhvata i manje delove sekunde). Oba polja unije neće se moći koristiti istovremeno.

```
union vreme {
  int obicno;
  float precizno;
};
```

Nakon definisanja tipa unije, moguće je definisati i promenljive ovog tipa, na primer:

```
union vreme a;
a.obicno = 17;
```

Unije se često koriste i kao članovi struktura. Neka se, na primer, u programu čuvaju i obrađuju informacije o studetima i zaposlenima na nekom fakultetu. Za svakoga se čuva ime, prezime i matični broj, za zaposlene se još čuva i koeficijent za platu, dok se za studente čuva broj indeksa:

```
struct akademac {
   char ime_i_prezime[50];
   char jmbg[14];
   char vrsta;
   union {
     double plata;
     char indeks[7];
   } dodatno;
};
```

U ovom slučaju poslednji član strukture (`dodatno`) je unijskog tipa (sâm tip unije nije imenovan). Član strukture `vrsta` tipa `char` sadrži informaciju o tome da li se radi o zaposlenom (na primer, vrednost `z`) ili o studentu (na primer, vrednost `s`). Promenljive `plata` i `indeks` dele zajednički memorijski prostor i podrazumeva se da se u jednom trenutku koristi samo jedan podatak u uniji. Na primer:

```
int main() {
  struct akademac pera = {"Pera Peric", "0101970810001", 'z'};
  pera.dodatno.plata = 56789.5;
  printf("%f\n", pera.dodatno.plata);
```

```
  struct akademac ana = {"Ana Anic", "1212992750501", 's'};
  strcpy(ana.dodatno.indeks, "12/123");
  printf("%s\n", ana.dodatno.indeks);
}
```

Pokušaj promene polja `pera.dodatno.indeks` bi narušio podatak o koeficijentu plate, dok bi pokušaj promene polja `ana.dodatno.koeficijent` narušio podatak o broju indeksa.

Navedimo, kao ilustraciju, kako se korišćenjem unije može otkriti binarni zapis broja u pokretnom zarezu:

```
union { float x; unsigned char s[4]; } u;
u.x = 1.234f;
printf("%x%x%x%x", u.s[0], u.s[1], u.s[2], u.s[3]);
```

6.8.3 Polja bitova

Još jedan od načina uštede memorije u C programima su *polja bitova* (engl. bit fields). Naime, najmanji celobrojni tip podataka je `char` koji zauzima jedan bajt, a za predstavljanje neke vrste podataka dovoljan je manji broj bitova. Na primer, zamislimo da želimo da predstavimo grafičke karakteristike pravougaonika u nekom programu za crtanje. Ako je dopušteno samo osnovnih 8 boja (crvena, plava, zelena, cijan, magenta, žuta, bela i crna) za predstavljanje boje dovoljno je 3 bita. Vrsta okvira (pun, isprekidan, tačkast) može da se predstavi sa 2 bita. Na kraju, da li pravougaonik treba ili ne treba popunjavati može se kodirati jednim bitom. Ovo može da se iskoristiti za definisanje sledećeg polja bitova.

```
struct osobine_pravougaonika {
  unsigned char popunjen           : 1;
  unsigned char boja               : 3;
  unsigned char vrsta_okvira       : 2;
};
```

Iza svake promenljive, naveden je broj bitova koji je potrebno odvojiti za tu promenljivu. Veličina ovako definisanog polja bitova (vrednost izraza `sizeof(struct osobine_pravougaonika)`) je samo 1 bajt (iako je potrebno samo 6 bitova, svaki podatak mora da zauzima ceo broj bajtova, tako da je odvojen 1 bit više nego sto je potrebno). Da je u pitanju bila obična struktura, zauzimala bi 3 bajta.

Polje bitova se nadalje koristi kao obična struktura. Na primer:

```
...
#define ZELENA  02
...
#define PUN 00

struct osobine_pravougaonika op;
op.popunjen = 0;
op.boja = ZELENA;
op.vrsta_okvira = PUN;
```

6.8.4 Nabrojivi tipovi (`enum`)

U nekim slučajevima korisno je definisati tip podataka koji ima mali skup dopuštenih vrednosti. Ovakvi tipovi se nazivaju *nabrojivi tipovi*. U jeziku C nabrojivi tipove se definišu korišćenjem ključne reči `enum`. Na primer:

```
enum znak_karte {
  KARO,
  PIK,
  HERC,
  TREF
};
```

Nakon navedene definicije, u programu se mogu koristiti imena `KARO`, `PIK`, `HERC`, `TREF`, umesto nekih konkretnih konstantnih brojeva, što popravlja čitljivost programa. Pri tome, obično nije važno koje su konkretne vrednosti pridružene imenima `KARO`, `PIK`, `HERC`, `TREF`, već je dovoljno znati da su one sigurno međusobno različite i celobrojne. U navedenom primeru, `KARO` ima vrednost 0, `PIK` vrednost 1, `HERC` vrednost 2 i `TREF` vrednost 3. Moguće je i eksplicitno navođenje celobrojnih vrednosti. Na primer:

```
enum znak_karte {
  KARO = 1,
  PIK  = 2,
  HERC = 4,
  TREF = 8
};
```

Vrednosti nabrojivih tipova nisu promenljive i njima se ne može menjati vrednost. S druge strane, promenljiva može imati tip koji je nabrojiv tip i koristiti se na uobičajene načine.

Sličan efekat (uvođenja imena sa pridruženim celobrojnim vrednostima) se može postići i pretprocesorskom direktivom `#define` (videti poglavlje 9.2.1), ali u tom slučaju ta imena ne čine jedan tip (kao u slučaju da se koristi `enum`). Grupisanje u tip je pogodno zbog provera koje se vrše u fazi prevođenja.

Slično kao i kod struktura i unija, uz definiciju tipa moguće je odmah deklarisati i promenljive. Promenljive se mogu i naknadno definisati (uz obavezno ponovno korišćenje ključne reči `enum`). Na primer,

```
enum znak_karte znak;
```

Nabrojivi tipovi mogu učestvovati u izgradnji drugih složenih tipova.

```
struct karta {
  unsigned char broj;
  enum znak_karte znak;
} mala_dvojka = {2, TREF};
```

Nabrojivi tipovi se često koriste da zamene konkretne brojeve u programu, na primer, povratne vrednosti funkcija. Mnogo je bolje, u smislu čitljivosti programa, ukoliko funkcije vraćaju (različite) vrednosti koje su opisane nabrojivim tipom (i imenima koja odgovaraju pojedinim povratnim vrednostim) nego konkretne brojeve. Tako, na primer, tip povratne vrednosti neke funkcije može da bude nabrojiv tip definisan na sledeći način:

```
enum return_type {
  OK,
  FileError,
  MemoryError,
  TimeOut
};
```

6.8.5 Typedef

U jeziku C moguće je kreirati nova imena postojećih tipova koristeći ključnu reč `typedef`. Na primer, deklaracija

```
typedef int Length;
```

uvodi ime `Length` kao sinonim za tip `int`. Ime tipa `Length` se onda može koristiti u deklaracijama, eksplicitnim konverzijama i slično, na isti način kao što se koristi ime `int`:

```
Length len, maxlen;
```

Novo ime tipa se navodi kao poslednje, na poziciji na kojoj se u deklaracijama obično navodi ime promenljive, a ne neposredno nakon ključne reči `typedef`. Obično se novouvedena imena tipova pišu velikim početnim slovima kako bi se istakla.

Deklaracijom `typedef` se ne kreira novi tip već se samo uvodi novo ime za potojeći tip. Staro ime za taj tip se može koristiti i dalje.

Veoma često korišćenje `typedef` deklaracija je u kombinaciji sa strukturama kako bi se izbeglo pisanje ključne reči `struct` pri svakom korišćenju imenu strukturnog tipa. Na primer:

```
struct point {
  int x, y;
};
typedef struct point Point;

Point a, b;
```

Definicija strukture i pridruživanje novog imena se mogu uraditi istovremeno. Na primer:

```
typedef struct point {
  int x, y;
} Point;

Point a, b;
```

Deklaracija `typedef` je slična pretprocesorskoj direktivi `#define` (videti poglavlje 9.2.1), s tim što nju obrađuje kompilator (a ne pretprocesor) i može da da rezultat i u slučajevima u kojima jednostavne pretprocesorske tekstualne zamene to ne mogu. Na primer:

```
typedef int (*PFI)(char *, char *);
```

uvodi ime `PFI`, za tip „pokazivač na funkciju koja prima dva `char *` argumenta i vraća `int`“ (više reči o pokazivačima na funkcije će biti u glavi 10) i koje se može koristiti na primer kao:

```
PFI strcmp, numcmp;
```

Osim estetskih razloga, postoje dve osnovne svrhe za korišćenje ključne reči `typedef` i imenovanje tipova. Prvi je parametrizovanje tipova u programu kako bi se dobilo na njegovoj prenosivosti. Ukoliko se `typedef` koristi za uvođenje novih imena za tipove koji su mašinski zavisni, u slučaju da se program prenosi na drugu mašinu, potrebno je promeniti samo `typedef` deklaracije. Jedan od primera je korišćenje `typedef` za imenovanje celobrojnog tipa, i zatim odabir `short`, `int` i `long` u zavisnosti od mašine. Druga svrha upotrebe `typedef` je poboljšanje čitljivosti programa – tip sa imenom `Point` se jednostavnije razumeti nego dugo ime strukture.

Pitanja i zadaci za vežbu

Pitanje 6.8.1.

1. *Definisati strukturu* `complex` *koja ima dva člana tipa* `double`.
2. *Definisati strukturu* `student` *kojom se predstavljaju podaci o studentu (ime, prezime, JMBG, prosečna ocena).*

Pitanje 6.8.2. *Da li je nad vrednostima koje su istog tipa strukture dozvoljeno koristiti operator dodele i koje je njegovo dejstvo?*

Pitanje 6.8.3. *Navesti primer inicijalizacije niza struktura tipa* `struct datum { unsigned dan, mesec, godina; }` *na današnji i sutrašnji datum.*

Pitanje 6.8.4. *Data je struktura:*

```
struct tacka {
    int a, b;
    char naziv[5];
}
```

Ova struktura opisuje tačku sa koordinatama `(a,b)` *u ravni kojoj je dodeljeno ime* `naziv`. *Za dve promenljive tipa* `struct tacka`, *napisati naredbe koje kopiraju sadržaj prve promenljive u drugu promenljivu.*

Pitanje 6.8.5. *Kakva je veza između nabrojivog (*`enum`*) i celobrojnog tipa u C-u?*

Pitanje 6.8.6.

1. *Na koji način se postojećem tipu* `t` *može dodeliti novo ime* `NoviTip`*?*
2. *Uvesti novo ime* `real` *za tip* `double`.
3. *Definisati novi tip* `novitip` *koji odgovara strukturi koji ima jedan član tipa* `int` *i jedan član tipa* `float`.

Zadatak 6.8.1. *Napisati program koji učitava 10 kompleksnih brojeva i među njima određuje broj najvećeg modula. Definisati i koristiti strukturu* `complex`. √

Zadatak 6.8.2. *Napraviti program koji učitava redni broj dana u nedelji (broji se od ponedeljka) i ispisuje da li je radni dan ili vikend (definisati i koristiti nabrojivi tip podataka).* √

Glava 7

Naredbe i kontrola toka

Osnovni elementi kojima se opisuju izračunavanja u C programima su *naredbe*. Naredbe za kontrolu toka omogućavaju različite načine izvršavanja programa, u zavisnosti od vrednosti promenljivih. Naredbe za kontrolu toka uključuju naredbe grananja i petlje. Iako u jeziku C postoji i naredba skoka (`goto`), ona neće biti opisivana, jer često dovodi do loše strukturiranih programa, nečitljivih i teških za održavanje a, dodatno, svaki program koji koristi naredbu `goto` može se napisati i bez nje. Na osnovu teoreme o strukturnom programiranju, od naredbi za kontrolu toka dovoljna je naredba grananja (tj. naredba `if`) i jedna vrsta petlje (na primer, petlja `while`), ali se u programima često koriste i druge naredbe za kontrolu toka radi bolje čitljivosti kôda. Iako mogu da postoje opšte preporuke za pisanje kôda jednostavnog za razumevanje i održavanje, izbor naredbi za kontrolu toka u konkretnim situacijama je najčešće stvar afiniteta programera.

7.1 Naredba izraza

Osnovni oblik naredbe koji se javlja u C programima je takozvana naredba izraza (ova vrsta naredbi obuhvata i naredbu dodele i naredbu poziva funkcije). Naime, svaki izraz završen karakterom `;` je naredba. Naredba izraza se izvršava tako što se izračuna vrednost izraza i izračunata vrednost se potom zanemaruje. Ovo ima smisla ukoliko se u okviru izraza koriste operatori koji imaju bočne efekte (na primer, `=`, `++`, `+=` itd.) ili ukoliko se pozivaju funkcije. Naredba dodele, kao i naredbe koje odgovaraju pozivima funkcija sintaksički su obuhvaćene naredbom izraza. Naredni primer ilustruje nekoliko vrsta naredbi:

```
3 + 4*5;
n = 3;
c++;
f();
```

Iako su sve četiri navedene naredbe ispravne i sve predstavljaju naredbe izraza, prva naredba ne proizvodi nikakav efekat i nema semantičko opravdanje pa se retko sreće u stvarnim programima. Četvrtka naredba predstavlja poziv funkcije `f` (pri čemu se, ako je ima, povratna vrednost funkcije zanemaruje).

Zanimljiva je i *prazna naredba* koja se obeležava samo znakom `;` (na primer, telo `for` petlje može sadržati samo praznu naredbu).

7.2 Složene naredbe i blokovi

U nekim slučajevima potrebno je više različitih naredbi tretirati kao jednu jedinstvenu naredbu. Vitičaste zagrade `{` i `}` se koriste da grupišu naredbe u složene naredbe ili blokove i takvi blokovi se mogu koristiti na svim mestima gde se mogu koristiti i pojedinačne naredbe. Iza zatvorene vitičaste zagrade ne piše se znak `;`. Vitičaste zagrade koje okružuju više naredbi neke petlje su jedan primer njihove upotrebe, ali one se, za grupisanje naredbi, koriste i u drugim kontekstima. Dopušteno je i da blok sadrži samo jednu naredbu, ali tada nema potrebe koristiti blok (izuzetak je definicija funkcije koja ne može da bude pojedinačna naredba već mora biti blok). Svaki blok, na početku može da sadrži (moguće praznu) listu deklaracija promeljivih (koje se mogu koristiti samo u tom bloku). Vidljivost tj. oblast važenja imena promenljivih određena je pravilima *dosega* (o čemu će više reči biti u poglavlju 9.2.2). Nakon deklaracija, navode se naredbe (elementarne ili složene, tj. novi blokovi). Postoje različite konvencije za nazubljivanje vitičastih zagrada prilikom unosa izvornog kôda.

7.3 Naredbe grananja

Naredbe grananja (ili naredbe uslova), na osnovu vrednosti nekog izraza, određuju naredbu (ili grupu naredbi) koja će biti izvršena.

7.3.1 Naredba `if-else`

Naredba uslova `if` ima sledeći opšti oblik:

```
if (izraz)
  naredba1
else
  naredba2
```

Naredba `naredba1` i `naredba2` su ili pojedinačne naredbe (kada se završavaju simbolom `;`) ili blokovi naredbi zapisani između vitičastih zagrada (iza kojih se ne piše simbol `;`).

Deo naredbe `else` je opcioni, tj. može da postoji samo `if` grana. Izraz `izraz` predstavlja logički uslov i najčešće je u pitanju celobrojni izraz (ali može biti i izraz čija je vrednost broj u pokretnom zarezu) za koji se smatra, kao i uvek, da je tačan (tj. da je uslov ispunjen) ako ima ne-nula vrednost, a inače se smatra da je netačan.

Na primer, nakon naredbe

```
if (5 > 7)
  a = 1;
else
  a = 2;
```

promenljiva `a` će imati vrednost `2`, a nakon naredbe

```
if (7)
  a = 1;
else
  a = 2;
```

promenljiva `a` će imati vrednost `1`.

Kako se ispituje istinitosna vrednost izraza koji je naveden kao uslov, ponekad je moguće taj uslov zapisati kraće. Na primer, `if (n != 0)` je ekvivalentno sa `if (n)`.

S obzirom na to da je dodela operator, naredni C kôd je sintaksički ispravan, ali verovatno je semantički pogrešan (tj. ne opisuje ono što je bila namera programera):

```
a = 3;
if (a = 0)
   printf("a je nula\n");
else
   printf("a nije nula\n");
```

Naime, efekat ovog kôda je da postavlja vrednost promenljive `a` na nulu (a ne da ispita da li je `a` jednako `0`), a zatim ispisuje tekst `a nije nula`, jer je vrednost izraza `a = 0` nula, što se smatra netačnim. Zamena operatora `==` operatorom `=` u naredbi `if` je česta greška.

if-else višeznačnost. Naredbe koje se izvršavaju uslovno mogu da sadrže nove naredbe uslova, tj. može biti više ugnježdenih `if` naredbi. Ukoliko vitičastim zagradama nije obezbeđeno drugačije, `else` se odnosi na poslednji prethodeći neuparen `if`. Ukoliko se želi drugačije ponašanje, neophodno je navesti vitičaste zagrade. U narednom primeru, `else` se odnosi na drugo a ne na prvo `if` (iako nazubljivanje sugeriše drugačije):

```
if (izraz1)
  if (izraz2)
    naredba1
else
  naredba2
```

U narednom primeru, `else` se odnosi na prvo a ne na drugo `if` :

```
if (izraz1) {
  if (izraz2)
    naredba1
} else
  naredba2
```

7.3.2 Konstrukcija `else-if`

Za višestruke odluke često se koristi konstrukcija sledećeg oblika:

```
if (izraz1)
  naredba1
else if (izraz2)
  naredba2
else if (izraz3)
  naredba3
else
  naredba4
```

U ovako konstruisanoj naredbi, uslovi se ispituju jedan za drugim. Kada je jedan uslov ispunjen, onda se izvršava naredba koja mu je pridružena i time se završava izvršavanje čitave naredbe. Naredba `naredba4` u gore navedenom primeru se izvršava ako nije ispunjen nijedan od uslova `izraz1`, `izraz2`, `izraz3`. Naredni primer ilustruje ovaj tip uslovnog grananja.

```
if (a > 0)
  printf("A je veci od nule\n");
else if (a < 0)
  printf("A je manji od nule\n");
else /* if (a == 0) */
  printf("A je nula\n");
```

7.3.3 Naredba `if-else` i operator uslova

Naredna naredba

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

određuje i smešta u promenljivu `x` veću od vrednosti `a` i `b`. Naredba ovakvog oblika se može zapisati kraće korišćenjem ternarnog operatora uslova `?:` (koji je opisan u poglavlju 6.5.7), na sledeći način:

```
x = (a > b) ? a : b;
```

7.3.4 Naredba `switch`

Naredba `switch` se koristi za višestruko odlučivanje i ima sledeći opšti oblik:

```
switch (izraz) {
    case konstantan_izraz1: naredbe1
    case konstantan_izraz2: naredbe2
    ...
    default:                naredbe_n
}
```

Naredbe koje treba izvršiti označene su *slučajevima* (engl. *case*) za različite moguće pojedinačne vrednosti zadatog izraza `izraz`. Svakom slučaju pridružen je konstantni celobrojni izraz. Ukoliko zadati izraz `izraz` ima vrednost konstantnog izraza navedenog u nekom slučaju, onda se izvršavanje nastavlja od prve naredbe pridružene tom slučaju, pa se nastavlja i sa izvršavanjem naredbi koje odgovaraju sledećim slučajevima iako izraz nije imao njihovu vrednost, sve dok se ne naiđe na kraj ili naredbu `break`. Na slučaj `default` se prelazi ako vrednost izraza `izraz` nije navedena ni uz jedan slučaj. Slučaj `default` je opcioni i ukoliko nije naveden, a nijedan postojeći slučaj nije ispunjen, onda se ne izvršava nijedna naredba u okviru bloka `switch`. Slučajevi mogu biti navedeni u proizvoljnom poretku (uključujući i slučaj `default`), ali različiti poretci mogu da daju različito ponašanje programa. Iako to standard ne zahteva, slučaj `default` se gotovo uvek navodi kao poslednji slučaj. I ukoliko

slučaj `default` nije naveden kao poslednji, ako vrednost izraza `izraz` nije navedena ni uz jedan drugi slučaj, prelazi se na izvršavanje naredbi od naredbe pridružene slučaju `default`.

U okviru naredbe `switch` često se koristi naredba `break`. Kada se naiđe na naredbu `break`, napušta se naredba `switch`. Najčešće se naredbe pridužene svakom slučaju završavaju naredbom `break` (čak i nakon poslednje navedenog slučaja, što je najčešće `default`). Time se ne menja ponašanje programa, ali se obezbeđuje da poredak slučajeva ne utiče na izvršavanje programa, te je takav kôd jednostavniji za održavanje.

Izostavljanje naredbe `break`, tj. previđanje činjenice da se, ukoliko nema naredbi `break`, nastavlja sa izvršavanjem naredbi narednih slučajeva, često dovodi do grešaka u programu. S druge strane, izostavljanje naredbe `break` može biti pogodno (i opravdano) za pokrivanje više različitih slučajeva jednom naredbom (ili blokom naredbi).

U narednom primeru proverava se da li je uneti broj deljiv sa tri, korišć cenjem naredbe `switch`.

Program 7.1.

```
#include <stdio.h>

int main() {
  int n;
  scanf("%i",&n);

  switch (n % 3) {
    case 1:
    case 2:
             printf("Uneti broj nije deljiv sa 3");
             break;
    default: printf("Uneti broj je deljiv sa 3");
  }
  return 0;
}
```

U navedenom primeru, bilo da je vrednost izraza `n % 3` jednaka `1` ili `2`, biće ispisan tekst `Uneti broj nije deljiv sa 3`, a inače će biti ispisan tekst `Uneti broj je deljiv sa 3`. Da u nije navedena naredba `break`, onda bi u slučaju da je vrednost izraza `n % 3` jednaka `1` (ili `2`), nakon teksta `Uneti broj nije deljiv sa 3`, bio ispisan i tekst `Uneti broj je deljiv sa 3` (jer bi bilo nastavljeno izvršavanje svih naredbi za sve naredne slučajeve).

7.4 Petlje

Petlje (ili repetitivne naredbe) uzrokuju da se određena naredba (ili grupa naredbi) izvršava više puta (sve dok je neki logički uslov ispunjen).

7.4.1 Petlja `while`

Petlja `while` ima sledeći opšti oblik:

```
while(izraz)
   naredba
```

U petlji `while` ispituje se vrednost izraza `izraz` i ako ona ima istinitosnu vrednost *tačno* (tj. ako je vrednost izraza različita od nule), izvršava se `naredba` (što je ili pojedinačna naredba ili blok naredbi). Zatim se uslov `izraz` iznova proverava i sve se ponavlja dok mu istinosna vrednost ne postane *netačno* (tj. dok vrednost ne postane jednaka nuli). Tada se izlazi iz petlje i nastavlja sa izvršavanjem prve sledeće naredbe u programu.

Ukoliko iza `while` sledi samo jedna naredba, onda, kao i oblično, nema potrebe za vitičastim zagradama. Na primer:

```
while (i < j)
  i++;
```

Sledeća `while` petlja se izvršava beskonačno:

```
while (1)
  i++;
```

7.4.2 Petlja `for`

Petlja `for` ima sledeći opšti oblik:

```
for (izraz1; izraz2; izraz3)
  naredba
```

Komponente `izraz1`, `izraz2` i `izraz3` su izrazi. Obično su `izraz1` i `izraz3` izrazi dodele ili inkrementiranja, a `izraz2` je relacioni izraz. Izraz `izraz1` se obično naziva *inicijalizacija* i koristi se za postavljanje početnih vrednosti promenljivih, izraz `izraz2` je *uslov* izlaska iz petlje, a `izraz3` je *korak* i njime se menjaju vrednosti relevantnih promenljivih. Naredba `naredba` naziva se *telo* petlje.

Inicijalizacija (izraz `izraz1`) izračunava se samo jednom, na početku izvršavanja petlje. Petlja se izvršava sve dok uslov (izraz `izraz2`) ima ne-nula vrednost (tj. sve dok mu je istinitosna vrednost *tačno*), a korak (izraz `izraz3`) izračunava se na kraju svakog prolaska kroz petlju. Redosled izvršavanja je, dakle, oblika: *inicijalizacija*, *uslov*, *telo*, *korak*, *uslov*, *telo*, *korak*, ..., *uslov*, *telo*, *korak*, *uslov*, pri čemu je *uslov* ispunjen svaki, osim poslednji put. Dakle, gore navedena opšta forma petlje `for` ekvivalentna je konstrukciji koja koristi petlju `while`:

```
izraz1;
while (izraz2) {
  naredba
  izraz3;
}
```

Petlja `for` se obično koristi kada je potrebno izvršiti jednostavno početno dodeljivanje vrednosti promenljivama i jednostavno ih menjati sve dok je ispunjen zadati uslov (pri čemu su i početno dodeljivanje i uslov i izmene lako vidljivi u definiciji petlje). To ilustruje sledeća tipična forma `for` petlje:

```
for(i = 0; i < n; i++)
   ...
```

Bilo koji od izraza `izraz1`, `izraz2`, `izraz3` može biti izostavljen, ali simboli `;` i tada moraju biti navedeni. Ukoliko je izostavljen izraz `izraz2`, smatra se da je njegova istinitosna vrednost *tačno*. Na primer, sledeća `for` petlja se izvršava beskonačno (ako u bloku naredbi koji ovde nije naveden nema neke naredbe koja prekida izvršavanje, na primer, `break` ili `return`):

```
for (;;)
  ...
```

Ako je potrebno da neki od izraza `izraz1`, `izraz2`, `izraz3` objedini više izraza, može se koristiti operator `,`.

```
for (i = 0, j = 10; i < j; i++, j--)
   printf("i=%d, j=%d\t", i, j);
```

Prethodni kôd ispisuje

```
i=0, j=10   i=1, j=9   i=2, j=8   i=3, j=7   i=4, j=6
```

Sledeći program, koji ispusuje tablicu množenja, ilustruje dvostruku `for` petlje:

Program 7.2.

```
#include<stdio.h>

int main() {
  int i, j, n=3;
  for(i = 1; i <= n; i++) {
    for(j = 1; j <= n; j++)
      printf("%i * %i = %i\t", i, j, i*j);
    printf("\n");
  }
  return 0;
}
```

```
1 * 1 = 1       1 * 2 = 2       1 * 3 = 3
2 * 1 = 2       2 * 2 = 4       2 * 3 = 6
3 * 1 = 3       3 * 2 = 6       3 * 3 = 9
```

7.4.3 Petlja `do-while`

Petlja `do-while` ima sledeći opšti oblik:

```
do {
  naredbe
} while(izraz)
```

Telo (blok naredbi `naredbe`) naveden između vitičastih zagrada se izvršava i onda se izračunava uslov (izraz `izraz`). Ako je on tačan, telo se izvršava ponovo i to se nastavlja sve dok izraz `izraz` nema vrednost nula (tj. sve dok njegova istinitosna vrednost ne postane *netačno*).

Za razliku od petlje `while`, naredbe u bloku ove petlje se uvek izvršavaju barem jednom.

Naredni program ispisuje cifre koje se koriste u zapisu unetog neoznačenog broja, zdesna na levo.

Program 7.3.

```
#include <stdio.h>
int main() {
  unsigned n;
  printf("Unesi broj: ");
  scanf("%u", &n);
  do {
    printf("%u ", n % 10);
    n /= 10;
  } while (n > 0);
  return 0;
}
```

```
  Unesi broj: 1234
  4 3 2 1
```

Primetimo da, u slučaju da je korišćena `while`, a ne `do-while` petlja, za broj `0` ne bi bila ispisana ni jedna cifra.

7.4.4 Naredbe `break` i `continue`

U nekim situacijama pogodno je napustiti petlju ne zbog toga što nije ispunjen uslov petlje, već iz nekog drugog razloga. To je moguće postići naredbom `break` kojom se izlazi iz tekuće petlje (ili naredbe `switch`)[1] Na primer:

```
for(i = 1; i < n; i++) {
  if(i > 10)
    break;
  ...
}
```

Korišćenjem naredbe `break` se narušava strukturiranost kôda i to može da oteža njegovu analizu (na primer, analizu ispravnosti ili analizu složenosti). U nekim situacijama, korišćenje naredbe `break` može da poboljša čitljivost, ali kôd koji koristi naredbu `break` uvek se može napisati i bez nje. U datom primeru, odgovarajući alternativni kôd je, na primer:

```
for(i = 1; i<n && i<=10; i++)
  ...
```

[1] Naredbom `break` ne izlazi se iz bloka naredbe grananja.

Naredbom `continue` se prelazi na sledeću iteraciju u petlji. Na primer,

```
for(i = 0; i < n; i++) {
  if (i % 10 == 0)
    continue;  /* preskoci brojeve deljive sa 10 */
  ...
}
```

Slično kao za naredbu `break`, korišćenjem naredbe `continue` se narušava strukturiranost kôda, ali može da se poboljša čitljivost. Kôd koji koristi naredbu `continue` uvek se može napisati i bez nje. U datom primeru, odgovarajući alternativni kôd je, na primer:

```
for(i = 0; i < n; i++)
  if (i % 10 != 0)  /* samo za brojeve koji nisu deljivi sa 10 */
    ...
```

U slučaju ugnježdenih petlji, naredbe `break` i `continue` imaju dejstvo samo na unutrašnju petlju. Tako, na primer, fragment

```
  for (i = 0; i < 3; i++)
      for (j = 0; j < 3; j++) {
          if (i + j > 2) break;
          printf("%d %d   ", i, j);
      }
```

ispisuje

```
0 0  0 1  0 2  1 0  1 1  2 0
```

Pitanja i zadaci za vežbu

Pitanje 7.1.

1. *Koju vrednost ima promenljiva* `x` *nakon izvršavanja kôda*

   ```
   if (!x)
     x += 2;
     x *= 2;
   ```

   ako je pre navedenog kôda imala vrednost `0`*, a koju ako je imala vrednost* `5`*?*

2. *Koju vrednost ima promenljiva* `x` *nakon izvršavanja kôda*

   ```
   int x = 0;
   if (x > 3);
      x++;
   ```

Pitanje 7.2. *Navesti primer naredbe u kojoj se javlja* `if-else` *višeznačnost.*

Pitanje 7.3. *Kako se, u okviru naredbe* `switch`*, označava slučaj koji pokriva sve slučajeve koji nisu već pokriveni?*

Pitanje 7.4. *Ako su promenljive* `i` *i* `j` *tipa* `int` *odrediti njihovu vrednost nakon naredbi:*

1.
```
i=2; j=5;
switch(j/i)
{
case 1:  i++;
         break;
case 2:  i += ++j;
default: j += ++i;
}
```

2.
```
int i=2, j=4;
switch (j%3) {
  case 0:  i=j++;
  case 1:  j++;++i;
  case 2:  j=i++;
  default:
}
```

Pitanje 7.5. *Šta ispisuje prikom izvršavanja narednih kôdova?*

1.
```
int i = 2; int j = 2;
while(i + 2*j <= 15) {
   i++;
   if (i % 3 == 0)
      j--;
   else {
      i+=2; j++;
   }
   printf("%d %d\n", i, j);
}
```

2.
```
for (i = 0; i < 5; i++)
    for (j = 0; j < 5; j++) {
       if ((i + j) % 3 == 0) break;
       printf("%d %d\n", i, j);
    }
```

3.
```
for (i = 0; i < 5; i++)
    for (j = 0; j < 5; j++) {
       if ((i + j) % 3 == 0) continue;
       printf("%d %d\n", i, j);
    }
```

4.
```
for (i = 1, j = 10; i < j; i++, j--)
     for (k = i; k <= j; k++)
          printf("%d ", k);
```

Pitanje 7.6. *1. Ukoliko se u naredbi* `for (e1; e2; e3) ...` *izostavi izraz* `e2`*, koja će biti njegova podrazumevana vrednost?*

2. *Da li je u naredbi* `for(e1; e2; e3) ...`*, moguće da su izrazi* `e1` *i* `e3` *sačinjeni od više nezavisnih podizraza i ako jeste, kojim operatorom su povezani ti podizrazi?*

Pitanje 7.7. *1. Izraziti sledeći kôd (i) petljom* `while` *i (ii) petljom* `for`

```
do {
  naredbe
} while(izraz)
```

2. *Izraziti sledeći kôd (i) petljom* `while` *i (ii) petljom* `do-while`

```
for (izraz1; izraz2; izraz3)
    naredba
```

3. *Izraziti sledeći kôd (i) petljom* `for` *i (ii) petljom* `do-while`

```
while (izraz)
    naredba
```

Pitanje 7.8. *1. Transformisati naredni kôd tako da ne sadrži* `break`*:*

```
while(A) {
    if(B) break;
    C;
}
```

2. *Transformisati naredni kôd tako da ne sadrži* `continue`*:*

```
for(;A;) {
    if(B) continue;
    C;
}
```

3. *Transformisati naredni kôd i zapisati ga u obliku* `while` *petlje tako da ne sadrži* `break`*:*

```
for(i = 1;;i++) if (i++ == 3) break; else printf("%d", i);
for(i = 1;;) if (++i == 3) break; else printf("%d", i);
for(i = 10;;) if (--i == 3) break; else printf("%d", i);
for(i = 1; i<5;) if (++i == 5) break; else printf("%d", i);
```

4. *Transformisati naredni kôd i zapisati ga u obliku* `for` *petlje sa sva tri izraza neprazna i bez korišćenja naredbe* `break`*.*

```
for(i = 10;;) if (--i == 3) break; else printf("%d", i);
for(i = 1;;) if (i++ == 3) break; else printf("%d", i);
```

Pitanje 7.9. *Da li se naredne petlje zaustavljaju:*

```
unsigned char c; for(c=10; c<9; c++) { ... }
signed char c; for(c=10; c<9; c++) { ... }
signed char c; for(c=0; c<128; c++) { ... }
unsigned char c; for(c=0; c<128; c++) { ... }
char c; for(c=0; c<128; c++) { ... }
```

Zadatak 7.1. *Napisati program koji ispisuje sve neparne brojeve manje od unetog neoznačenog celog broja* `n`. √

Zadatak 7.2. *Napisati program koji izračunava i ispisuje vrednost funkcije* $\sin(x)$ *u 100 tačaka intervala* $[0, 2\pi]$ *na jednakim rastojanjima.* √

Zadatak 7.3. *Napisati program koji učitava realan broj* x *i neoznačeni ceo broj* n *i izračunava* x^n. √

Zadatak 7.4. *Napisati programe koji ispisuje naredne dijagrame, pri čemu se dimenzija* `n` *unosi. Svi primeri su za* $n = 4$*:*

```
a) **** b) ****   c) *     d) ****   e)    *   f) * * * *  g)     *
   ****    ***       **        ***        **       * * *         * *
   ****    **        ***        **       ***        * *         * * *
   ****    *         ****        *      ****         *         * * * *
```

√

Zadatak 7.5. *Napisati program koji određuje sumu svih delilaca broja (koristiti činjenicu da se delioci javljaju u paru).* √

Zadatak 7.6. *Napisati program koji određuje da li je uneti neoznačeni ceo broj prost.* √

Zadatak 7.7. *Napisati program koji ispisuje sve proste činioce unetog neoznačenog celog broja (na primer, za unos* `24` *traženi prosti činioci su* `2, 2, 2, 3`*).* √

Zadatak 7.8. *Napisati program koji učitava cele brojeve sve dok se ne unese* 0*, a onda ispisuje:*

1. *broj unetih brojeva;*
2. *zbir unetih brojeva;*
3. *proizvod unetih brojeva;*
4. *minimum unetih brojeva;*
5. *maksimum unetih brojeva;*
6. *aritmetičku sredinu unetih brojeva (*$\frac{x_1+\ldots+x_n}{n}$*);*
7. *geometrijsku sredinu unetih brojeva (*$\sqrt[n]{x_1 \cdot \ldots \cdot x_n}$*);*
8. *harmonijsku sredinu unetih brojeva (*$\frac{n}{\frac{1}{x_1}+\ldots+\frac{1}{x_n}}$*);*

√

Zadatak 7.9. *Napisati program koji učitava cele brojeve sve dok se ne unese* 0 *i izračunava dužinu najduže serije uzastopnih jednakih brojeva.* √

Zadatak 7.10. *Napisati program koji za uneti neoznačeni ceo broj n izračunava ceo deo njegovo korena $\lfloor\sqrt{n}\rfloor$ (koristiti algoritam opisan u primeru 3.3).* √

Zadatak 7.11. *Korišćenjem činjenice da niz definisan sa*

$$x_0 = 1, \qquad x_{n+1} = x_n - \frac{x_n^2 - a}{2 \cdot x_n}$$

teži ka $\sqrt{a}$, napisati program koji (bez korišćenja funkcije `sqrt`*) procenjuje vrednost $\sqrt{a}$. Iterativni postupak zaustaviti kada je $|x_{n+1} - x_n| < 0.0001$.* √

Zadatak 7.12. *Fibonačijev niz brojeva $1, 1, 2, 3, 5, 8, 13, 21, \ldots$ je definisan uslovima*

$$f_0 = 1, \quad f_1 = 1, \quad f_{n+2} = f_{n+1} + f_n.$$

Napisati program koji ispisuje prvih k članova ovog niza (koristiti ideju da se u svakom trenutku pamte, u dve promenljive, vrednosti dva prethodna elementa niza). √

Zadatak 7.13. *Napisati program koji ispisuje sve cifre unetog neoznačenog celog broja* `n`*, počevši od cifre jedinica. Napisati i program koji izračunava sumu cifara unetog neoznačenog celog broja* `n`*.* √

Zadatak 7.14. *Napisati program koji „nadovezuje“ dva uneta neoznačena cela broja (na primer, za unos* `123` *i* `456` *program ispisuje* `123456`*). Ako je jedan od brojeva 0, rezultat treba da bude jednak drugom broju. Pretpostaviti da rezultat staje u opseg tipa* `unsigned`*.* √

Zadatak 7.15. *Napisati program koji izračunava broj dobijen obrtanjem cifara unetog celog broja (na primer, za unos* `1234` *program ispisuje broj* `4321`*).* √

Zadatak 7.16. *Napisati program koji izbacuje sve neparne cifre iz zapisa unetog neoznačenog celog broja (na primer, za uneti broj* 123456 *program ispisuje* 246*).* √

Zadatak 7.17. *Napisati program koji umeće datu cifru c na datu poziciju p neoznačenog celog broja n (pozicije se broje od 0 sa desna). Na primer, za unos $c = 5$, $p = 2$, $n = 1234$, program ispisuje* 12534*.* √

Zadatak 7.18.

1. *Napisati program koji razmenjuje prvu i poslednju cifru unetog neoznačenog celog broja (na primer, za unos* 1234 *program ispisuje* 4231*).*
2. *Napisati program koji ciklično pomera cifre unetog neoznačenog celog broja u levo (na primer, za unos* 1234 *program ispisuje* 2341*).*
3. *Napisati program koji ciklično pomera cifre unetog neoznačenog celog broja u desno (na primer, za unos* 1234 *program ispisuje* 4123*).*

√

Zadatak 7.19. *Napisati program koji za zadati dan, mesec i godinu ispituje da li je uneti datum korektan (uzeti u obzir i prestupne godine). Koristiti* `case` *naredbu.* √

Zadatak 7.20. *Napisati program koji učitava neoznačene cele brojeve sve dok se ne unese 0, a na standardni prepisuje sve one brojeve koji su:*

1. *manji od minimuma prethodno unetih brojeva (prvi broj se uvek ispisuje);*
2. *veći od aritmetičke sredine prethodno unetih brojeva.*

√

Zadatak 7.21. *Napisati program koji ispisuje prvih n (najviše 100000) elemenata niza koji se dobija tako što se kreće od* 0*, a zatim prethodni niz ponovi pri čemu se svaki broj uveća za* 1*, tj.* `0112122312232334...` √

Zadatak 7.22. *Sa standardnog ulaza se unose dva niza brojeva (svaki sadrži najviše 100 elemenata). Napisati program koji određuje njihov presek, uniju i razliku (redosled prikaza elemenata nije bitan, i u ulaznim nizovima se elementi ne ponavljaju).* √

Zadatak 7.23. *Napisati program koji proverava da li je data niska palindrom (čita se isto sleva i zdesna, na primer,* `anavolimilovana`*). Prilagoditi program tako da se razmaci zanemaruju i ne pravi se razlika između malih i velikih slova (na primer,* `Ana voli Milovana` *je palindrom).* √

Zadatak 7.24. *Napisati program koji obrće i ispisuje datu nisku.* √

Zadatak 7.25. *Napisati program koji sa standardnog ulaza unosi prvo dimenziju matrice ($n < 100$) pa zatim elemente matrice i zatim ispisuje sve dijagonale matrice paralelne sa sporednom dijagonalom. Na primer, unos:*

```
3 1 2 3 4 5 6 7 8 9
```

opisuje matricu

$$\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$$

za koju program ispisuje

```
1
2 4
3 5 7
6 8
9
```

√

Zadatak 7.26. *Napisati program koji sa standardnog ulaza učitava prvo dimenzije matrice ($n < 100$ i $m < 100$) a zatim redom i elemente matrice. Nakon toga ispisuje indekse (i i j) onih elemenata matrice koji su jednaki zbiru svih svojih susednih elemenata (pod susednim elementima podrazumevamo okolnih* 8 *polja matrice ako postoje). Na primer, za matricu:*

```
1 1 2 1 3
0 8 1 9 0
1 1 1 0 0
0 3 0 2 2
```

polja na pozicijama `[1][1]`*,* `[3][1]`*, i* `[3][4]` *zadovoljavaju traženi uslov.* √

Glava 8

Funkcije

Svaki C program sačinjen je od funkcija. Funkcija `main` mora da postoji i, pojednostavljeno rečeno, izvršavanje programa uvek počinje izvršavanjem ove funkcije[1]. Iz funkcije `main` (ali i drugih) pozivaju se druge funkcije, bilo bibliotečke (poput funkcije `printf`), bilo korisnički definisane (kakve će biti opisane u nastavku)[2]. Druge funkcije koriste se kako bi kôd bio kraći, čitljiviji, modularniji, šire upotrebljiv itd. U funkciju se obično izdvaja neko izračunavanje, neka obrada koja predstavlja celinu za sebe i koristi se više puta u programu. Tako se dobija ne samo kraći i jednostavniji kôd, već i kvalitetniji u smislu da je neko izračunavanje skriveno u funkciji i ona može da se koristi čak i ako se ne zna kako tačno je ona implementirana, već je dovoljno znati šta radi, tj. kakav je rezultat njenog rada za zadate argumente.

8.1 Primeri definisanja i pozivanja funkcije

Naredni programi ilustruju jednostavne primere korišćenja funkcija. Pojmovi koji su u okviru ovog primera samo ukratko objašnjeni biće detaljnije objašnjeni u nastavku teksta.

Program 8.1.

```
#include <stdio.h>

int kvadrat(int n);

int main() {
    printf("Kvadrat broja %i je %i\n", 5, kvadrat(5));
    printf("Kvadrat broja %i je %i\n", 9, kvadrat(9));
    return 0;
}
```

[1] U glavi 9 biće objašnjeno da se određeni delovi izvršnog programa, obično sistemski pozivi i određene inicijalizacije, izvršavaju i pre poziva funkcije `main`, kao i da se određeni delovi programa izvršavaju i nakon njenog završetka.

[2] Iz funkcija programa može se pozivati i funkcija `main`, ali to obično nema mnogo smisla i generalno se ne preporučuje.

```
int kvadrat(int n) {
    return n*n;
}
```

Linija `int kvadrat(int n);` *deklariše* funkciju `kvadrat` koja će biti *definisana* kasnije u kôdu[3]. Iz deklaracije se vidi da funkcija `kvadrat` ima jedan parametar tipa `int` i vraća rezultat tipa `int`. U funkciji `main`, u okviru poziva funkcije `printf` *poziva* se funkcija `kvadrat` za vrednosti `5` i `9` (tipa `int`) i ispisuje se rezultat, tipa `int`, koji ona vraća. Svaki poziv funkcije je izraz (koji može ravnopravno dalje učestvovati u širim izrazima) čiji je tip povratni tip funkcije, a vrednost povratna vrednost funkcije. Na mestu poziva, tok izvršavanja programa prelazi na početak funkcije koja je pozvana, a kada se završi izvršavanje funkcije tok izvršavanja programa vraća se na mesto poziva. Definicija funkcije `kvadrat` je jednostavna: ona kao rezultat, vraća kvadrat celobrojne vrednosti `n` čija se vrednost postavlja na osnovu vrednosti koja je prosleđena prilikom poziva.

U narednom programu, funkcija `stepen` izračunava celobrojni stepen realne promenljive.

Program 8.2.

```
#include <stdio.h>

int max = 10;

float stepen(float x, unsigned n);

int main() {
  unsigned i;
  for(i = 0; i < max; i++)
     printf("%d %f %f\n", i, stepen(2.0f,i), stepen(3.0f,i));
   return 0;
}

float stepen(float x, unsigned n) {
   unsigned i;
   float s = 1.0f;
   for(i = 1; i<=n; i++)
     s *= x;
   return s;
}
```

Primetimo da je promenljiva `i` deklarisana u funkciji `main`, druga promenljiva `i` deklarisana je u funkciji `stepen`, a promenljiva `max` deklarisana je van svih funkcija. Promenljive deklarisana u funkcijama nazivamo obično *lokalne promenljive* i njih je moguće koristiti samo u okviru funkcije u kojoj su definisane, dok promenljive deklarisana van svih funkcija nazivamo obično *globalne promenljive* i one su zajedničke za više funkcija. O ovoj temi biće više reči u poglavlju 9.2.2.

[3] Za razliku od funkcija gde se deklaracije i definicije prilično jednostavno razlikuju, kod promenljivih je ovu razliku mnogo teže napraviti. Odnos između deklaracija i definicija promenljivih u jeziku C biće diskutovan u glavi 9.

8.2 Definicija funkcije

Definicija funkcije ima sledeći opšti oblik:

```
tip ime_funkcije(niz_deklaracija_parametara) {
   deklaracije
   naredbe
}
```

Imena funkcija su identifikatori i za njih važe potpuno ista pravila kao i za imena promenljivih. Radi čitljivosti kôda, poželjno je da ime funkcije oslikava ono šta ona radi.

Definicije funkcija mogu se navesti u proizvoljnom poretku i mogu se nalaziti u jednoj ili u više datoteka. U drugom slučaju, potrebno je instruirati prevodilac da obradi više izvornih datoteka i da napravi jednu izvršnu verziju (videti poglavlje 9.2.6).

Postojanje dve definicije funkcije istog imena u jednom programu dovodi do greške tokom prevođenja ili povezivanja (čak i ako su liste parametara različite).

8.3 Parametri funkcije

Funkcija može imati parametre koje obrađuje i oni se navode u okviru definicije, iza imena funkcije i između zagrada. Termini *parametar funkcije* i *argument funkcije* se ponekad koriste kao sinonimi. Ipak, pravilno je termin *parametar funkcije* koristiti za promenljivu koja čini deklaraciju funkcije, a termin *argument funkcije* za izraz naveden u pozivu funkcije na mestu parametra funkcije. Ponekad se argumenti funkcija nazivaju i *stvarni argumenti*, a parametri funkcija *formalni argumenti*. U primeru iz poglavlja 8.1, `n` je parametar funkcije `int kvadrat(int n);`, a `5` i `9` su njeni argumenti u pozivima `kvadrat(5)` i `kvadrat(9)`.

Parametri funkcije mogu se u telu funkcije koristiti kao lokalne promenljive te funkcije a koje imaju početnu vrednost određenu vrednostima argumenata u pozivu funkcije.

Kao i imena promenljivih, imena parametara treba da oslikavaju njihovo značenje i ulogu u programu. Pre imena svakog parametra neophodno je navesti njegov tip. Ukoliko funkcija nema parametara, onda se između zagrada ne navodi ništa ili se navodi ključna reč `void`. Ukoliko se, umesto ključne reči `void`, između zagrada ne navede ništa, prevodilac u pozivima funkcije ne proverava da li je ona zaista pozvana bez argumenata.

Promenljive koje su deklarisane kao parametri funkcije lokalne su za tu funkciju i njih ne mogu da koriste druge funkcije. Štaviše, bilo koja druga funkcija može da koristi isto ime za neki svoj parametar ili za neku svoju lokalnu promenljivu.

Funkcija `main` može biti bez parametara ili može imati dva parametra unapred određenog tipa (videti poglavlje 12.4).

Prilikom poziva funkcije, vrši se *prenos argumenata*, što će biti opisano u zasebnom poglavlju 8.7.

8.4 Povratna vrednost funkcije

Funkcija može da vraća rezultat i `tip` označava tip vrednosti koja se vraća kao rezultat. Funkcija rezultat vraća naredbom `return r;` gde je `r` izraz zadatog tipa ili tipa koji se može konvertovati u taj tip. Naredba `return r;` ne samo da vraća vrednost `r` kao rezultat rada funkcije, nego i prekida njeno izvršavanje. Ako funkcija koja treba da vrati vrednost ne sadrži `return` naredbu, kompilator može da prijavi upozorenje, a u fazi izvršavanja rezultat poziva će biti nedefinisana vrednost (što je obično pogrešno). Ako funkcija ne treba da vraća rezultat, onda se kao tip povratne vrednosti navodi specijalan tip `void` i tada naredba `return` nema argumenata (tj. navodi se `return;`). Štaviše, u tom slučaju nije ni neophodno navoditi naredbu `return` iza poslednje naredbe u funkciji.

Funkcija koja je pozvala neku drugu funkciju može da ignoriše, tj. da ne koristi vrednost koju je ova vratila.

Iako je sintaksički ispravno i drugačije, funkcija `main` uvek treba da ima `int` kao tip povratne vrednosti (jer okruženje iz kojeg je program pozvan uvek kao povratnu vrednost očekuje tip `int`).

8.5 Deklaracija funkcije

U navedenim primerima, prvo je navedena *deklaracija* (ili *prototip*) funkcije, a tek kasnije njena definicija. U prvom primeru, deklaracija je

```
int kvadrat(int n);
```

a definicija

```
int kvadrat(int n) {
    return n*n;
}
```

Deklaracija ukazuje prevodiocu da će u programu biti korišćena funkcija sa određenim tipom povratne vrednosti i parametrima određenog tipa. Zahvaljujući tome, kada prevodilac, u okviru funkcije `main`, naiđe na poziv funkcije `kvadrat`, može da proveri da li je njen poziv ispravan (čak iako je definicija funkcije nepoznata). Pošto prototip služi samo za proveravanje tipova u pozivima, nije neophodno navoditi imena parametara, već je dovoljno navesti njihove tipove. U navedenom primeru, dakle, prototip je mogao da glasi i

```
int kvadrat(int);
```

Definicija funkcija mora da bude u skladu sa navedenim prototipom, tj. moraju da se podudaraju tipovi povratne vrednosti i tipovi parametara.

Nije neophodno za svaku funkciju navoditi najpre njen prototip, pa onda definiciju. Ukoliko je navedena definicija funkcije, onda se ona može koristiti u nastavku programa i bez navođenja prototipa (jer prevodilac iz definicije funkcije može da utvrdi sve relevantne tipove). Međutim, kada postoji više funkcija u programu i kada postoje njihove međuzavisnosti, može biti veoma teško (i to nepotrebno teško) poređati njihove definicije na način koji omogućava prevođenje (sa proverom tipova argumenata). Zato je praksa da se na početku

programa navode prototipovi funkcija, čime poredak njihovih definicija postaje irelevantan.

Postojanje dve deklaracije iste funkcije u okviru jednog programa je dozvoljeno, ali postojanje dve deklaracije funkcije istog imena, a različitih lista parametara dovodi do greške tokom prevođenja.

Ukoliko se ni definicija ni deklaracije funkcije ne navedu pre prvog poziva te funkcije, prevodilac pretpostavlja da funkcija vraća vrednost tipa `int`, i ne vrši nikakvu provera ispravnosti argumenata u pozivima funkcije. Ovakvo ponašanje je predviđeno jedino zbog kompatibilnosti sa starim verzijama jezika C, te je opšta preporuka da se svakako pre prvog poziva funkcije navede njena deklaracija ili definicija.

8.6 Konverzije tipova argumenata funkcije

Prilikom poziva funkcije, ukoliko je poznata njena deklaracija, vrši se implicitna konverzija tipova argumenata u tipove parametara. Slično, prilikom vraćanja vrednosti funkcije (putem `return` naredbe) vrši se konverzija vrednosti koja se vraća u tip povratne vrednosti funkcije.

U sledećem primeru, prilikom poziva funkcije `f` vrši se konverzija `double` vrednosti u celobrojnu vrednost i program ispisuje `3`:

Program 8.3.

```
#include <stdio.h>

void f(int a) {
  printf("%d\n", a);
}

int main() {
  f(3.5);
}
```

Ukoliko deklaracija funkcije nije navedena pre njenog poziva, u fazi prevođenja ne vrši se nikakva provera ispravnosti tipova argumenata i povratne vrednosti a u situacijama gde se poziva funkcija vrše se podrazumevane promocije argumenata: one obuhvataju celobrojne promocije (tipovi `char` i `short` se promovišu u `int`), i promociju tipa `float` u tip `double`. U ovom slučaju (da deklaracija nije navedena pre poziva funkcije), ukoliko je definicija funkcije ipak navedena negde u programu, zbog kompilacije koja je sprovedena na opisani način, izvršni program može da bude generisan, ali njegovo ponašanje može da bude neočekivano. Ako je u navedenom primeru definicija funkcije `f` navedena nakon definicije funkcije `main`, prilikom prevođenja funkcije `main` neće biti poznat tip funkcije `f` i kao parametar u okviru poziva biće preneta vrednost `3.5` zapisana u pokretnom zarezu (kao vrednost tipa `double`). S druge strane, funkcija `f` (koja je kompilirana nakon funkcije `main`) prima parametar tipa `int` i dobijenu vrednost `3.5` tumačiće kao ceo broj zapisan u potpunom komplementu. Zato bi tako modifikovani program ispisao vrednost `0` (a ne `3.5`, kao u prvom slučaju).

Prototipovi funkcija iz standardne biblioteke dati su u datotekama zaglavlja i da bi se one mogle bezbedno koristiti dovoljno je samo uključiti odgovarajuće

zaglavlje. Međutim, neki prevodioci (uključujući GCC) poznaju prototipove funkcija standardne biblioteke, čak i kada zaglavlje nije uključeno. Tako, ako se u GCC-u ne uključi potrebno zaglavlje, dobija se upozorenje, ali ne i greška jer su prototipovi standardnih funkcija unapred poznati. Ipak, ovakav kôd treba izbegavati i zaglavlja bi uvek trebalo eksplicitno uključiti (tj. pre svakog poziva funkcije trebalo bi osigurati da kompilator poznaje njen prototip).

8.7 Prenos argumenata

Na mestu u programu gde se poziva neka funkcija kao njen argument se može navesti promenljiva, ali i bilo koji izraz istog tipa (ili izraz čija vrednost može da se konvertuje u taj tip). Na primer, funkcija `kvadrat` iz primera iz poglavlja 8.1 može biti pozvana sa `kvadrat(5)`, ali i sa `kvadrat(2+3)`.

Argumenti funkcija se uvek prenose *po vrednosti*. To znači da se vrednost koji je poslata kao argument funkcije *kopira* kada počne izvršavanje funkcije i onda funkcija radi samo sa tom kopijom, ne menjajući original. Na primer, ako je funkcija `kvadrat` deklarisana sa `int kvadrat(int n)` i ako je pozvana sa `kvadrat(a)`, gde je `a` neka promenljiva, ta promenljiva će nakon izvršenja funkcije `kvadrat` ostati nepromenjena, ma kako da je funkcija `kvadrat` definisana. Naime, kada počne izvršavanje funkcije `kvadrat`, vrednost promenljive `a` biće iskopirana u lokalnu promenljivu `n` koja je navedena kao parametar funkcije i funkcija će koristiti samo tu kopiju u svom radu. Prenos argumenata ilustruje i funkcija `swap` definisana na sledeći način:

Program 8.4.

```
#include <stdio.h>

void swap(int a, int b) {
   int temp = a;
   a = b;
   b = temp;
}

int main() {
   int x = 3, y = 5;
   swap(x, y);
   printf("x = %d, y = %d\n", x, y);
}
```

U funkciji `swap` argumenti `a` i `b` zamenjuju vrednosti, ali ako je funkcija pozvana iz neke druge funkcije sa `swap(x, y)`, onda će vrednosti promenljivih `x` i `y` ostati nepromenjene nakon ovog poziva.

```
x = 3, y = 5
```

Moguće je i da se ime parametra funkcije poklapa sa imenom promenljive koja je prosleđena kao stvarni argument. Na primer:

Program 8.5.

```
#include <stdio.h>

void f(int a) {
  a = 3;
  printf("f: a = %d\n", a);
}

int main() {
   int a = 5;
   f(a);
   printf("main: a = %d\n", a);
}
```

I u ovom slučaju radi se o dve različite promenljive (promenljiva u pozvanoj funkciji je kopija promenljive iz funkcije u kojoj se poziv nalazi).

```
f: a = 3
main: a = 5
```

Ukoliko je potrebno promeniti neku promenljivu unutar funkcije, onda se kao argument ne šalje vrednost te promenljive nego njena adresa (i ta adresa se onda prenosi po vrednosti). O tome će biti reči u glavi 10. Alternativno, moguće je da funkcija samo izračuna vrednost i vrati je kao rezultat, a da se izmena promenljive izvrši u funkciji pozivaocu, naredbom dodele.

Prenos nizova vrši se na specifičan način, o čemu će biti reči u narednom poglavlju.

8.8 Nizovi i funkcije

Niz se ne može preneti kao argument funkcije. Umesto toga, moguće je kao argument navesti ime niza. Tokom kompilacije, imenu niza pridružena je informacija o adresi početka niza, o tipu elemenata niza i o broju elemenata niza. Kada se ime niza navede kao argument funkcije, onda do te funkcije (čiji je tip odgovarajućeg parametra pokazivački tip, o čemu će biti više reči u glavi 10) stiže informacija o adresi početka niza i o tipu elemenata (ali ne i o imenu niza niti o broju elemenata niza). Pošto funkcija koja je pozvana dobija informaciju o adresi početka originalnog niza, ona može da neposredno menja njegove elemente (i takve izmene će biti sačuvane nakon izvršenja funkcije), što je drugačije u odnosu na sve ostale tipove podataka. Prenos adrese početka niza vrši se — kao i uvek — po vrednosti. U pozvanoj funkciji, adresa početka niza navedenog u pozivu se kopira i može se menjati (tj. ona je l-vrednost), pri čemu te izmene ne utiču na njenu originalnu vrednost (adresu početka niza). S druge strane, kao što je ranije rečeno, originalna adresa početka niza nije l-vrednost i ne može se menjati.

Funkcija koja prima niz može biti deklarisana na neki od narednih načina:

```
tip ime_fje(tip ime_niza[dimenzija]);
tip ime_fje(tip ime_niza[]);
```

S obzirom na to da se u funkciju prenosi samo adresa početka niza, a ne i dimenzija niza, prvi oblik deklaracije nema puno smisla tako se znatno ređe koristi.

Ukoliko se ime dvodimenzionog niza koristi kao argument u pozivu funkcije, deklaracija parametra u funkciji mora da uključi broj kolona; broj vrsta je nebitan, jer se i u ovom slučaju prenosi samo adresa (tj. pokazivač na niz vrsta, pri čemu je svaka vrsta niz). U slučaju niza sa više od dve dimenzije, samo se prva dimenzija može izostaviti (dok je sve naredne dimenzije neophodno navesti).

```
tip ime_fje(tip ime_niza[bv][bk]);
tip ime_fje(tip ime_niza[][bk]);
```

Naredni primer ilustruje činjenicu da se u funkciju ne prenosi ceo niz. Ukoliko se program pokrene na mašini na kojoj su tip `int` i adresni tip reprezentovani sa 4 bajta) program ispisuje, u okviru funkcije `main`, broj 20 (5 elemenata tipa `int` čija je veličina 4 bajta)[4] i, u okviru funkcije `f`, broj 4 (veličina adrese koja je preneta u funkciju). Dakle, funkcija `f` nema informaciju o broju elemenata niza `a`.

Program 8.6.

```
#include <stdio.h>

void f(int a[]) {
  printf("f: %d\n", sizeof(a));
}

int main() {
  int a[] = {1, 2, 3, 4, 5};
  printf("main: %d\n", sizeof(a));
  f(a);
  return 0;
}
```

```
  main: 20
  f: 4
```

Prilikom prenosa niza (tj. adrese njegovog početka) u funkciju, pored imena niza, korisnik obično treba da eksplicitno prosledi i broj elemenata niza kao dodatni argument (kako bi pozvana funkcija imala tu informaciju).

Povratni tip funkcije ne može da bude niz. Funkcija ne može da kreira niz koji bi bio vraćen kao rezultat, a rezultat može da vrati popunjavanjem niza koji joj je prosleđen kao argument.

U narednom programu, funkcija `ucitaj_broj` ne uspeva da učita i promeni vrednost broja `x`, dok funkcija `ucitaj_niz` ispravno unosi i menja elemente niza `y` (jer joj je poznata adresa početka niza `y`).

Program 8.7.

```
#include <stdio.h>

void ucitaj_broj(int a) {
```

[4] `sizeof` je operator, a ne funkcija, te prilikom poziva `sizeof(a)` nema prenosa argumenata u kojem bi bila izgubljena informacija o broju elemenata niza `a`.

```
  printf("Ucitaj broj: ");
  scanf("%d", &a);
}

void ucitaj_niz(int a[], int n) {
  int i;
  printf("Ucitaj niz: ");
  for (i = 0; i < n; i++)
     scanf("%d", &a[i]);
}

int main() {
  int x = 0;
  int y[3] = {0, 0, 0};
  ucitaj_broj(x);
  ucitaj_niz(y, 3);
  printf("x = %d\n", x);
  printf("y = %d %d %d\n", y[0], y[1], y[2]);
}
```

```
  Unesi broj: 5
  Unesi niz: 1 2 3
  x = 0
  y = 1, 2, 3
```

Prenos niski u funkciju se vrši kao i prenos bilo kog drugog niza. Jedina razlika je što nije neophodno funkciji prosleđivati dužinu niske, jer se ona može odrediti i u samoj funkciji zahvaljujući postojanju terminalne nule.[5] Na primer, funkcija `strchr` stanadardne biblioteke proverava da li data niska sadrži dati karakter. U nastavku je navedena jedna njena moguća implementacija, zasnovana na takozvanoj linearnoj pretrazi niza. Ova implementacija pronalazi prvu poziciju karaktera `c` u niski `s` a vraća `-1` ukoliko `s` ne sadrzi `c`, što je nešto drugačije ponašenja u odnosu na implementaciju iz standardne biblioteke.

```
int strchr(const char s[], char c) {
  int i;
  for (i = 0; s[i] != '\0'; i++)
     if (s[i] == c)
        return i;

  return -1;
}
```

U nekim slučajevima želimo da sprečimo da funkcija izmeni niz (ili nisku) koji joj je prosleđen i tada je uz niz potrebno navesti da je konstantan (ključnom rečju `const`), što je i urađeno i u slučaju funkcije `strchr`. Ključna reč `const` mogla bi se navesti i uz parametar `c`, ali to bi (kako se karakter prenosi po vrednosti) značilo samo da ovaj parametar ne može da se menja unutar tela funkcije.

[5] Naravno, dužina niske ne određuje dužinu niza karaktera u koji je smeštena, već samo broj karaktera do završne nule.

8.9 Korisnički definisani tipovi i funkcije

Parametri funkcija mogu biti i strukture i drugi korisnički definisani tipovi. Pored toga, funkcije kao tip povratne vrednosti mogu imati tip strukture. Prenos argumenta se i u ovom slučaju vrši po vrednosti.

Funkcija `kreiraj_razlomak` od dva cela broja kreira i vraća objekat tipa `struct razlomak`:

```
struct razlomak kreiraj_razlomak(int brojilac, int imenilac)
{
    struct razlomak rezultat;
    rezultat.brojilac = brojilac;
    rezultat.imenilac = imenilac;
    return rezultat;
}
```

Navedni primer pokazuje i da ne postoji konflikt između imena parametara i istoimenih članova strukture. Naime, imena članova strukture su uvek vezana za ime promenljive (u ovom primeru `rezultat`).

Sledeći primer ilustruje funkcije sa parametrima i povratnim vrednostima koji su tipa strukture:

```
struct razlomak saberi_razlomke(struct razlomak a,
                                struct razlomak b)
{
    struct razlomak c;
    c.brojilac = a.brojilac*b.imenilac + a.imenilac*b.brojilac;
    c.imenilac = a.imenilac*b.imenilac;
    return c;
}
```

Na sličan način mogu se implementirati i druge operacije nad razlomcima, kao množenje razlomaka, skraćivanje razlomka, poređenje razlomaka, itd.

Kao i drugi tipovi argumenata, strukture se u funkcije prenose po vrednosti. Tako, naredna funkcija ne može da promeni razlomak koji je upotrebljen kao argument.

```
void skrati(struct razlomak r) {
  int n = nzd(r.brojilac, r.imenilac);
  r.brojilac /= n; r.imenilac /= n;
}
```

Naime, nakon poziva

```
  struct razlomak r = {2, 4};
  skrati(r);
```

razlomak `r` ostaje jednak 2/4 (u funkciji se menja kopija originalne strukture nastala prilikom prenosa po vrednosti).

8.10 Rekurzija

Funkcije mogu da pozivaju druge funkcije. Funkcija može da pozove i sama sebe (u tom slučaju argumenti u pozivima su obično različiti). Na primer, funkcija

```
double stepen(double x, unsigned n) {
  if (n == 0)
    return 1;
  else
    return x*stepen(x, n-1);
}
```

izračunava vrednost x^n. Na primer, `stepen(2.0, 3)` = `2.0*stepen(2.0, 2)` = `4.0*stepen(2.0, 1)` = `8.0*stepen(2.0, 0)` = `8.0*1` = `8.0`.

Rekurzivna rešenja često su vremenski i memorijski zahtevna (jer često zauzimaju mnogo prostora na steku poziva), ali zato često daju kraći i razumljiviji izvorni kôd. Rekurzija je veoma važna programerska tehnika i o njenoj upotrebi će više reči biti u drugom delu ove knjige.

Pitanja i zadaci za vežbu

Pitanje 8.10.1. *Šta je to deklaracija funkcije, a šta je to definicija funkcije? U kojim slučajevima se koriste deklaracije?*

Pitanje 8.10.2. *Šta je to parametar, a šta argument funkcije?*

Pitanje 8.10.3. *Koliko najmanje parametara može da ima funkcija? Kako izgleda deklaracija funkcije koja nema parametara? Koja je razlika između deklaracija* `void f(void);` *i* `void f();`*?*

Pitanje 8.10.4. *Šta se navodi kao tip povratne vrednosti za funkciju koja ne vraća rezultat? Da li takva funkcija mora da sadrži naredbu* `return`*? Da li takva funkcija može da sadrži naredbu* `return`*? Šta se navodi kao argument* `return` *naredbe?*

Pitanje 8.10.5. *Kako se prenose argumenti funkcija u C-u? Da li se u nekoj situaciji prenos argumenata ne vrši po vrednosti?*

Pitanje 8.10.6. *Šta ispisuju sledeći programi?*

1.
```
int f(int x) {
    x = x+2;
    return x+4;
}

int main() {
    int x = 1, y = 2;
    x = f(x+y);
    printf("%d\n", x);
}
```

2.
```
void f(int n,int k) {
  n = 3 * k++;
}

int main() {
  int n = 4, k = 5;
  f(n, k);
  printf("%d %d\n", n, k);
}
```

3.
```
void f(int x) {
  x += (++x) + (x++);
}
int main() {
  int x = 2;
  f(x); f(x);
  printf("%d\n", x);
}
```

Pitanje 8.10.7.

1. *Koje su informacije od navedenih, tokom kompilacije, pridružene imenu niza: (i) adresa početka niza; (ii) tip elemenata niza; (iii) broj elemenata niza; (iv) adresa kraja niza?*

2. *Koje se od ovih informacija čuvaju u memoriji tokom izvršavanja programa?*

3. *Šta se sve od navedenog prenosi kada se ime niza navede kao argument funkcije: (i) adresa početka niza; (ii) elementi niza; (iii) podatak o broju elemenata niza; (iv) adresa kraja niza.*

Pitanje 8.10.8. *Šta ispisuje naredni program:*

```
void f(int a[]) {
  int i;
  for(i = 0; i < sizeof(a)/sizeof(int); i++)
    a[i] = a[i] + 1;
}

int main() {
  int i;
  int a[] = {1, 2, 3, 4};
  f(a);
  for(i = 0; i < sizeof(a)/sizeof(int); i++)
    printf("%d ", a[i]);
}
```

Pitanje 8.10.9.

1. *Koja funkcija standardne biblioteke se koristi za izračunavanje dužine niske i u kojoj datoteci zaglavlja je ona deklarisana?*

2. *Čemu je jednako* `sizeof("abc")`*, a čemu* `strlen("abc")`*?*

3. *Navesti neku implementaciju funkcije* `strlen`*.*

Pitanje 8.10.10. *Navesti liniju kôda koja obezbeđuje da se prilikom poziva* `void` *funkcije* `f` *celobrojni argument 2 implicitno prevede u* `double` *vrednost.*

```
int main() {
   f(2);
   return 0;
}
```

Pitanje 8.10.11. *Kako se vrši prenos unija u funkciju?*

Zadatak 8.10.1. *Napisati funkciju koja proverava da li je uneti pozitivan ceo broj prost i program koji je testira. Uporediti sa rešenjem koje sadrži samo funkciju* `main`*.* √

Zadatak 8.10.2. *Napisati program koji za tri tačke Dekartove ravni zadate parovima svojih koordinata (tipa* `double`*) izračunava površinu trougla koji obrazuju (koristiti Heronov obrazac, na osnovu kojeg je površina trougla jednaka* $\sqrt{p(p-a)(p-b)(p-c)}$*, gde su* a*,* b *i* c *dužine stranica, a* p *njegov poluobim; napisati i koristiti funkciju koja računa rastojanje između dve tačke Dekartove ravni).* √

Zadatak 8.10.3. *Za prirodan broj se kaže da je* savršen *ako je jednak zbiru svih svojih pravih delioca (koji uključuju broj* 1*, ali ne i sam taj broj). Ispisati sve savršene brojeve manje od 10000.* √

Zadatak 8.10.4. *Napisati funkciju koja poredi dva razlomka i vraća 1 ako je prvi veći, 0 ako su jednaki i -1 ako je prvi manji.* √

Zadatak 8.10.5. *Napisati funkciju (i program koji je testira) koja:*

1. *proverava da li dati niz sadrži dati broj;*
2. *pronalazi indeks prve pozicije na kojoj se u nizu nalazi dati broj (*`-1` *ako niz ne sadrži broj).*
3. *pronalazi indeks poslednje pozicije na kojoj se u nizu nalazi dati broj (*`-1` *ako niz ne sadrži broj).*
4. *izračunava zbir svih elemenata datog niza brojeva;*
5. *izračunava prosek (aritmetičku sredinu) svih elemenata datog niza brojeva;*
6. *izračunava najmanji element datog elemenata niza brojeva;*
7. *određuje poziciju najvećeg elementa u nizu brojeva (u slučaju više pojavljivanja najvećeg elementa, vratiti najmanju poziciju);*
8. *proverava da li je dati niz brojeva uređen neopadajuće.*

√

Zadatak 8.10.6. *Napisati funkciju (i program koji je testira) koja:*

1. *izbacuje poslednji element niza;*

2. *izbacuje prvi element niza (napisati varijantu u kojoj je bitno očuvanje redosleda elemenata i varijantu u kojoj nije bitno očuvanje redosleda);*
3. *izbacuje element sa date pozicije* `k`*;*
4. *ubacuje element na kraj niza;*
5. *ubacuje element na početak niza;*
6. *ubacuje dati element* `x` *na datu poziciju* `k`*;*
7. *izbacuje sva pojavljivanja datog elementa* `x` *iz niza.*

Napomena: funkcija kao argument prima niz i broj njegovih trenutno popunjenih elemenata, a vraća broj popunjenih elemenata nakon izvođenja zahtevane operacije. √

Zadatak 8.10.7. *Napisati funkciju (i program koji je testira) koja:*

1. *određuje dužinu najduže serije jednakih uzastopnih elemenata u datom nizu brojeva;*
2. *određuje dužinu najvećeg neopadajućeg podniza datog niza celih brojeva;*
3. *određuje da li se jedan niz javlja kao podniz uzastopnih elemenata drugog;*
4. *određuje da li se jedan niza javlja kao podniz elemenata drugog (elementi ne moraju da budu uzastopni, ali se redosled pojavljivanja poštuje);*
5. *obrće dati niz brojeva;*
6. *rotira sve elemente datog niza brojeva za* `k` *pozicija ulevo;*
7. *rotira sve elemente datog niza brojeva za* `k` *pozicija udesno;*
8. *izbacuje višestruka pojavljivanja elemenata iz datog niza brojeva (napisati varijantu u kojoj se zadržava prvo pojavljivanje i varijantu u kojoj se zadržava poslednje pojavljivanje).*
9. *spaja dva niza brojeva koji su sortirani neopadajući u treći niz brojeva koji je sortiran neopadajući.*

√

Zadatak 8.10.8. *Napisati funkciju koja poredi dve niske leksikografski (kao u rečniku, u skladu sa kôdnim rasporedom sistema). Funkcija vraća negativan rezultat ukoliko prva niska leksikografski prethodi drugoj, nulu u slučaju da su niske jednake i pozitivan rezultat ukoliko druga niska leksikografski prethodi prvoj.* √

Zadatak 8.10.9. *Napisati funkciju koja za dve niske proverava da li je:*

1. *prva podniz druge (karakteri prve niske se ne moraju nalaziti na susednim pozicijama u drugoj).*
2. *prva podniska druge (karakteri prve niske se moraju nalaziti na susednim pozicijama u drugomq). Ako jeste funkcija treba da vrati poziciju prve niske u drugoj, a ako nije, onda* -1*. Napisati funkciju koja proverava da li jedna zadata niska sadrži drugu zadatu nisku.*

√

Zadatak 8.10.10. *Napisati funkciju koja za dve niske proverava da li je jedan permutacija druge (niska je permutacija druge ako se od nje dobija samo premeštanjem njegovih karaktera - bez ikakvog brisanja ili dodavanja karaktera). Na primer, funkcija odredi da niske* `"abc"` *i* `"cba"` *jesu, a da niske* `"aab"` *i* `"ab"` *nisu permutacija.* √

Zadatak 8.10.11. *Za datu kvadratnu matricu kažemo da je* magični kvadrat *ako je suma elemenata u svakoj koloni i svakoj vrsti jednaka. Napisati program koji sa standardnog ulaza učitava prirodni broj* n *(*$n < 10$*) i zatim elemente kvadratne matrice, proverava da li je ona* magični kvadrat *i ispisuje odgovarajuću poruku na standardni izlaz. Koristiti pomoćne funkcije za izračunavanje zbira elemenata vrsta, kolona i dijagonala matrice.*

Primer, matrica:

$$\begin{pmatrix} 7 & 12 & 1 & 14 \\ 2 & 13 & 8 & 11 \\ 16 & 3 & 10 & 5 \\ 9 & 6 & 15 & 4 \end{pmatrix}$$

je magični kvadrat. √

Zadatak 8.10.12. *Definisati strukturu kojom se može predstaviti matrica dimenzija najviše* 100×100 *(struktura sadrži dvodimenzioni brojeva tipa* `float` *i dimenzije).*

1. *Napisati funkciju koja učitava matricu sa standardnog ulaza.*
2. *Napisati funkciju koja ispisuje matricu na standardni izlaz.*
3. *Napisati funkciju koja sabira dve matrice istih dimenzija.*
4. *Napisati funkciju koja množi dve matrice odgovarajućih dimenzija.*

√

Zadatak 8.10.13. *Napisati funkcije koji vrše unos, ispis, sabiranje i množenje* velikih prirodnih brojeva *(za koje nije dovoljna veličina tipa* `unsigned long`*). Cifre velikog broja smeštati u niz i pretpostaviti da brojevi nemaju više od* 1000 *cifara.* √

Glava 9

Organizacija izvornog i izvršnog programa

U uvodnim poglavljima već je naglašeno da se *izvorni program* piše najčešće na višim programskim jezicima i predstavlja sredstvo komunikacije između programera i računara, ali i između programera sâmih. Specijalizovani programi, prevodioci (kompilatori), prevode izvorni u *izvršni program*, napisan na mašinskom jeziku računara, tj. na jeziku neposredno podržanom arhitekturom mikroprocesora.

Postoji mnoštvo pisanih pravila (tj. pravila koja proističu iz standarda jezika koji se koristi) i nepisanih pravila (razne konvencije i običaji) za organizovanje kôda izvornog programa. Ta pravila olakšavaju programiranje i omogućuju programeru kreiranje razumljivih programa koji se efikasno mogu prevoditi i izvršavati. Svi programi u prethodnim poglavljima bili su jednostavni, imali su svega nekoliko funkcija i nije bilo potrebno previše obraćati pažnju na pomenuta pravila. Međutim, sa usložnjavanjem programa, poštovanje kodeksa postaje ključno jer u protivnom pisanje i održavanje kôda postaje veoma komplikovano[1]. Takođe, postoje pravila za organizovanje kôda izvršnog programa koja omogućavaju njegovo efikasno izvršavanje na određ enom računaru, u sklopu određenog operativnog sistema, što podrazumeva i nesmetano izvršavanje istovremeno sa drugim programima. Zato pravila organizacije kôda izvršnog programa zavise i od višeg programskog jezika koji se koristi, ali i od hardvera, mašinskog jezika, kao i operativnog sistema računara na kome će se program izvršavati.

I u fazi prevođenja i u fazi izvršavanja mogu se javiti greške i one moraju biti ispravljene da bi program funkcionisao ispravno. Greške tokom izvršavanja mogu biti takve da program ne može dalje da nastavi sa izvršavanjem (npr. ako program pokuša da pristupi delu memorije koji mu nije dodeljen, operativni sistem nasilno prekida njegovo izvršavanje), ali mogu da budu takve da se program nesmetano izvršava, ali da su rezultati koje prikazuje pogrešni. Dakle, to što ne postoje greške u fazi prevoď enja i to što se program nesmetano izvršava, još uvek ne znači da je program ispravan, tj. da zadovoljava svoju specifikaciju i daje tačne rezultate za sve vrednosti ulaznih parametara. Ispravnost pro-

[1] Postoji izreka da dobre od loših programera više razlikuje disciplina pri programiranju nego pamet.

grama u tom, dubljem smislu zahteva formalnu analizu i ispitivanje te vrste ispravnosti najčešće je van moći automatskih alata. Ipak, sredstva za prijavljivanje grešaka tokom prevođenja i izvršavanja programa znatno olakšavaju proces programiranja i često ukazuju i na suštinske propuste koji narušavaju ispravnost programa.

9.1 Od izvornog do izvršnog programa

Iako proces prevođenja jednostavnih programa početnicima izgleda kao jedan, nedeljiv proces, on se sastoji od više faza i podfaza (a od kojih se svaka i sama sastoji od više koraka). U slučaju jezika C, ti koraci su obično *pretprocesiranje* (engl. preprocessing), *kompilacija* (engl. compilation) i *povezivanje* (engl. linking). Tokom svih faza prevođenja, može biti otkrivena i prijavljena greška u programu. U izveštajima o greškama obično se ne navodi eksplicitno u kojoj fazi je greška detektovana (mada se ta informacija može rekonstruisati na osnovu vrste greške).

Pretprocesiranje. Faza pretprocesiranja je pripremna faza kompilacije. Ona omogućava da se izvrše neke jednostavne transformacije izvornog teksta programa pre nego što on bude prosleđen kompilatoru — kompilator, dakle, ne obrađuje tekst programa koji je programer napisao, već samo tekst koji je nastao njegovim pretprocesiranjem. Jedan od najvažnijih zadataka pretprocesora je da omogući da se izvorni kôd pogodno organizuje u više ulaznih datoteka. Pretprocesor izvorni kôd iz različitih datoteka objedinjava u tzv. *jedinice prevođenja* i prosleđuje ih kompilatoru. Na primer, u `.c` datoteke koje sadrže izvorni kôd uključuju se *datoteke zaglavlja* (engl. header files) `.h` koje sadrže deklaracije promenljivih, funkcija i tipova podataka tako da svaka jedinica prevođenja sadrži zajednički tekst nekoliko `.h` (na primer, već pomenuta zaglavlja standardne biblioteke) i jedne `.c` datoteke.

Program koji vrši pretprocesiranje naziva se *pretprocesor* (engl. preprocessor). Rezultat rada pretprocesora, može se dobiti korišćenje GCC prevodioca navođenjem opcije `-E` (na primer, `gcc -E program.c`).

Više reči o fazi pretprocesiranja biće u poglavlju 9.2.1.

Kompilacija. Kompilator kompilira (tj. prevodi) svaku jedinicu prevođenja zasebno, sprovođenjem sledećih faza (videti poglavlje 4.3):

- *leksička analiza* — izdvajanje *leksema*, osnovnih jezičkih elemenata;
- *sintaksička analiza* — kreiranje sintaksnog stabla;
- *semantička analiza* — provera semantike i transformacija kreiranog stabla;
- *generisanje međukôda* — generisanje kôda na jeziku interne reprezentacije;
- *optimizacija međukôda* — optimizovanje generisanog kôda;
- *generisanje kôda na mašinskom jeziku* — prevođenje optimizovanog kôda u objektne module.

Kompilacijom se od svake jedinice prevođenja gradi zasebni *objektni modul* (engl. object module)[2]. Objektni moduli sadrže programe (mašinski kôd funkcija) i podatke (memorijski prostor rezervisan za promenljive). Iako su u mašinskom obliku, objektni moduli se ne mogu izvršavati.

Na primer, rezultat rada GCC kompilatora može se dobiti navođenjem opcije `-c` (na primer, `gcc -c program.c`). Ovim se dobija datoteka `program.o` – objektni modul koji je na Linux sistemu u ELF formatu (engl. Executable and Linkable Format), formatu u kome je i izvršna datoteka, pri čemu objektni modul nije izvršan (kaže se da je relokatibilan) i mora se povezati da bi se dobila izvršna datoteka.

Mnoge dodatne opcije utiču na rezultat kompilacije. Na primer, ukoliko se navede opcija `gcc -O3`, vrši se najopsežnija optimizacija kôda što uzrokuje da se rezultujući kôd (često znatno) brže izvršava. Ukoliko se navede opcija `-Wall` navode se sva upozorenja na moguće greške tokom kompilacije.

Povezivanje. Povezivanje je proces kreiranja jedinstvene izvršne datoteke od jednog ili više objektnih modula koji su nastali ili kompilacijom izvornog kôda programa ili su objektni moduli koji sadrže mašinski kôd i podatke standardne ili neke nestandardne biblioteke[3]. Program koji vrši povezivanje zove se *povezivač* ili *linker* (engl. linker).

Nakon kompilacije, u objektnim modulima adrese mašinskog kôda funkcija i adrese nekih promenljivih nisu razrešene (može se smatrati da su prazne jer su umesto stvarnih postavljene specijalne, fiktivne adrese – često vrednost 0) i tek tokom povezivanja vrši se njihovo korektno razrešavanje (zamena ispravnim vrednostima). Zato je faza povezivanja neophodna, čak iako jedan objektni modul sadrži sve promenljive i funkcije koje se koriste u programu (što najčešće nije slučaj jer se koriste funkcije standardne biblioteke čiji se izvršni kôd ne nalazi u objektnim modulima nastalim kompilacijom izvornog kôda).

Kao što je već rečeno, pored objektnih modula nastalih kompilacijom izvornog kôda koje je programer napisao, izvršnom programu se može dodati unapred pripremljen mašinski kôd nekih biblioteka. Taj kôd je mogao nastati prevođenjem sa jezika C, ali to ne mora biti slučaj. Na primer, neke biblioteke su pisane u asembleru, a neke su nastale prevođenjem sa drugih viših programskih jezika. Mehanizam povezivanja, dakle, dopušta i kombinovanje kôda napisanog na različitim programskim jezicima. Mehanizam povezivanja bitno skraćuje trajanje kompilacije jer se kompilira samo tanak sloj korisničkog izvornog kôda dok se kompleksan kôd biblioteka koristi unapred kompiliran. Takve biblioteke uvek su praćene datotekama zaglavlja koje služe kompilatoru da proveri i ispravno prevede kôd u kojem se poziva na funkcionalnost biblioteke.

Najčešći primer korišćenja unapred pripremljenog mašinskog kôda sa pratećim datotekama zaglavlja je korišćenje standardne biblioteke (pregled sadržaja standardne biblioteke biće dat u glavi 11). Kôd standardne biblioteke je obično dostavljen uz prevodilac i dat je samo u obliku mašinskog, a ne izvornog kôda (na primer, obično nije moguće videti izvorni kôd funkcije `printf`). Deo standardne biblioteke su datoteke zaglavlja koje sadrže samo potrebne de-

[2] Objektni moduli na sistemu Linux najčešće imaju ekstenziju `.o`, a na sistemu Windows `.obj`.

[3] Biblioteke na sistemu Linux najčešće imaju ekstenziju `.a` (jer su arhive koje sadrže nekoliko `.o` datoteka), dok na sistemu Windows najčešće imaju ekstenziju `.lib`.

klaracije (na primer, deklaracija funkcije `printf` može se pronaći u zaglavlju `<stdio.h>`).

Pored *statičkog povezivanja*, koje se vrši nakon kompilacije, postoji i *dinamičko povezivanje*, koje se vrši tokom izvršavanja programa (zapravo na njegovom početku). Naime, statičko povezivanje u izvršnu datoteku umeće mašinski kôd svih bibliotečkih funkcija. Kako bi se smanjila veličina izvršnih datoteka, mašinski kôd nekih često korišćenih funkcija se ne uključuje u izvršnu datoteku programa već postoji u tzv. *bibliotekama koje se dinamički povezuju* (engl. dynamic link library)[4]. Izvršne datoteke sadrže nerazrešene pozive funkcija koje se dinamički povezuju i tek prilikom pokretanja programa, dinamičke biblioteke se zajedno sa programom učitavaju u memoriju računara i adrese poziva bibliotečkih funkcija se razrešavaju. Zato, prilikom izvršavanja programa biblioteke sa dinamičkim povezivanjem moraju biti prisutne na sistemu. Na primer, funkcije standardne biblioteke intenzivno pozivaju funkcije rantajm biblioteke i razrešavanje takvih poziva vrši se dinamički, u fazi izvršavanja programa. Još jedna prednost dinamičkog povezivanja bibliotečkih funkcija je da se nove verzije biblioteka (često sa ispravljenim sitnijim greškama) mogu koristiti bez potrebe za izmenom kôda koji te biblioteke koristi.

Proces povezivanja obično se automatski pokreće odmah nakon kompilacije. Na primer, ako se koristi GCC prevodilac, prevođenje programa od jedne datoteke i povezivanje sa objektnim modulima standardne biblioteke se postiže na sledeći način:

```
gcc -o program program.c
```

Ovim se dobija izvršni program `program` (u ELF formatu).

U procesu povezivanja, automatski se uključuju potrebni objektni moduli standardne biblioteke (na operativnom sistemu Linux ovaj modul se obično zove `libc` ili slično). Dodatne biblioteke mogu se uključiti navođenjem parametra `-l`. Na primer,

```
gcc -o program -lm program.c
```

u proces povezivanja uključuje i standardnu biblioteku `libm` koja nije podrazumevano obuhvaćena povezivanjem.

Odvojena kompilacija i povezivanje. Korak kompilacije i korak povezivanja moguće je razdvojiti tj. moguće je prvo prevesti datoteku sa izvornim programom i eksplicitno napraviti objektni modul (u ovom slučaju bi se zvao `program.o`), a tek zatim ovako napravljeni objektni modul povezati (uključujući i objektni kôd standardne biblioteke) i dobiti izvršni program. U slučaju da se koristi GCC, to se postiže sa:

```
gcc -c program.c
gcc -o program program.o
```

Kao što je već rečeno, parametar `-c` govori GCC prevodiocu da ne treba da vrši povezivanje već samo kompilaciju i da rezultat eksplicitno sačuva kao objektni

[4] Ekstenzija ovih datoteka u sistemu Linux je obično `.so` (shared object), dok je u sistemu Windows `.dll`.

modul u datoteci `program.o`. Drugi poziv vrši povezivanje tog objektnog modula i daje izvršni program `program`.

Program se može izgraditi i iz više jedinica prevođenja. Na primer, ako je izvorni kôd programa razdvojen u dve datoteke `program1.c` i `program2.c`, izvršni program moguće je dobiti komandom:

```
gcc -o program program1.c program2.c
```

Naravno, svaka datoteka se odvojeno prevodi i tek onda se vrši povezivanje. Dakle, efekat je isti kao da je pozvano

```
gcc -c program1.c
gcc -c program2.c
gcc -o program program1.o program2.o
```

Razdvajanjem kompilacije i povezivanja u dve faze omogućeno je da se moduli koji se ne menjaju ne prevode iznova svaki put kada je potrebno napraviti novu verziju izvršnog program (kada se promeni neka izvorna datoteka). Na primer, ako je promena izvršena samo u datoteci `program1.c`, nije potrebno ponovo prevoditi datoteku `program2.c`.

Program `make`. Program `make` (prvu verziju napisao je Sjuart Feldman, 1977. godine) olakšava generisanje izvršnih programa od izvornih datoteka. Iako postoje varijante za razne platforme, program `make` najčešće se koristi na operativnom sistemu Linux. Alternativa (ili nadogradnja) programa `make` su savremena *integrisana razvojna okruženja* (engl. integrated development environment, IDE) u kojima programer na interaktivni način specifikuje proces kompilacije.

Prilikom korišćenja programa `make`, korisnik u datoteci koja obično ima ime `Makefile` specifikuje način dobijanja novih datoteka od starih. Specifikuju se zavisnosti između datoteka i akcije koje se izvršavaju ukoliko su te zavisnosti narušene (linije koje sadrže akcije moraju da počnu tabulatorom, a ne razmacima). Nakon pokretanja programa `make`, ukoliko se desi da je neka zavisnost narušena, izvršava se odgovarajuća akcija i rekurzivno se nastavlja sa proverom zavisnosti, sve dok se sve zavisnosti ne ispune. Na primer, datoteka `Makefile` sledećeg sadržaja

```
program: program1.o program2.o biblio.o
	gcc -o program program1.o program2.o biblio.o

program1.o : program1.c biblio.h
	gcc -c program1.c

program2.o : program2.c biblio.h
	gcc -c program2.c
```

kaže da datoteka `program` zavisi od datoteka `program1.o`, `program2.o` i `biblio.o`. Ukoliko je zavisnost narušena (što znači, ukoliko je bilo koja od datoteka `program1.o`, `program2.o` ili `biblio.o` novijeg datuma od datoteke `program`), izvršava se povezivanje. Slično, ako je datoteka `program1.c` ili `biblio.h` novijeg datuma o `program1.o` koji od njih zavisi (verovatno tako što datoteka

`program1.c` uključuje zaglavlje `biblio.h`), vrši se njena kompilacija. Analogno je i za `program2`.

Program `make` ima još mnogo opcija ali one neće ovde biti opisivane.

Izvršavanje programa. Nakon uspešnog prevođenja može da sledi faza izvršavanja programa.

Izvršni programi smešteni su u datotekama na disku i pre pokretanja učitavaju se u glavnu memoriju. Za ovo je zadužen program koji se naziva *učitavač* ili *louder* (engl. loader) koji je obično integrisan u operativni sistem. Njegov zadatak je da proveri da li korisnik ima pravo da izvrši program, da prenese program u glavnu memoriju, da prekopira argumente komandne linije (o kojima će više biti reči u poglavlju 12.4) u odgovarajući deo memorije programa koji se pokreće, da inicijalizuju određene registre u procesoru i na kraju da pozovu početnu funkciju programa. Nasuprot očekivanju, to nije funkcija `main`, već funkcija `_start` koja poziva funkciju `main`, ali pre toga i nakon toga vrši dodatne pripremne i završne operacije sa programom, korišćenjem usluga rantajm biblioteke tj. operativnog sistema.

Pre konačne distribucije korisnicima, program se obično intenzivno testira tj. izvršava za različite vrednosti ulaznih parametara. Ukoliko se otkrije da postoji greška tokom izvršavanja (tzv. *bag* od engl. bug), vrši se pronalaženje uzroka greške u izvornom programu (tzv. *debagovanje*) i njeno ispravljanje. U pronalaženju greške mogu pomoći programi koji se nazivaju *debageri* i koji omogućavaju izvršavanje programa korak po korak (ili pauziranje njegovog izvršavanja u nekim karakterističnim tačkama), uz prikaz međuvrednosti promenljivih. Da bi program mogao da bude analiziran primenom debagera, on mora biti preveden na poseban način, tako da izvršna verzija sadrži i informacije o izvornom kôdu. Na primer, za prevodilac GCC potrebno je koristiti opciju `-g`. Takva verzija izvršnog programa zove se *debag* verzija, a verzija koja se isporučuje krajnjem korisniku zove se *riliz* (engl. release).

Pitanja i zadaci za vežbu

Pitanje 9.1.1. *Navesti i objasniti osnovne faze na putu od izvornog do izvršnog programa.*

Pitanje 9.1.2. *U kom obliku se isporučuje standardna C biblioteka?*

Pitanje 9.1.3. *Kako se korišćenjem GCC prevodioca može izvršiti samo faza pretprocesiranja kôda i prikazati rezultat? Kako se može izvršiti samo faza kompilacije, bez povezivanja objektnih modula?*

Pitanje 9.1.4. *Šta su jedinice prevođenja? Šta su datoteke zaglavlja? U kojoj fazi se od više datoteka grade jedinice prevođenja?*

Pitanje 9.1.5. *Šta su objektni moduli? Šta sadrže? Da li se mogu izvršavati? Kojim procesom nastaju objektni moduli? Kojim procesom se od objektnih modula dobija izvršni program?*

Pitanje 9.1.6. *Da li svi objektni moduli koji se uključuju u kreiranje izvršnog programa moraju nastati kompilacijom sa programskog jezika C? Da li je moguće kombinovati različite programske jezike u izgradnji izvršnih programa?*

Pitanje 9.1.7. *Šta znači da u objektnim modulima pre povezivanja adrese nisu korektno razrešene?*

Pitanje 9.1.8. *U kom formatu su objektni moduli i izvršne datoteke na Linux sistemima?*

Pitanje 9.1.9. *Šta je statičko, a šta dinamičko povezivanje? Koje su prednosti dinamičkog u odnosu na statičko povezivanje?*

Pitanje 9.1.10. *U kojim fazama prevođenja programa se vrši prijavljivanje grešaka?*

Pitanje 9.1.11. *Kako se korišćenjem GCC prevodioca vrši odvojena kompilacija i povezivanje programa u datotekama* `p1.c` *i* `p2.c`.

Pitanje 9.1.12. *Šta je i čemu služi program* `make`*?*

Pitanje 9.1.13. *Kako se zove alat koji omogućava da se program izvršava korak-po-korak i da se prate vrednosti promenljivih?*

9.2 Organizacija izvornog programa

Viši programski jezici namenjeni su prvenstveno čoveku (a ne računaru). Obično je daleko lakše neki algoritam pretočiti u program na višem programskom jeziku nego u program na mašinskom jeziku. Isto važi i za razumevanje programa koje je napisao neko drugi. Ipak, pisanje, razumevanje i održavanje veoma dugih programa može da predstavlja veliki izazov za programera, čak i kada je program napisan na višem programskom jeziku. Kako bi se olakšalo pisanje, razumevanje i održavanje programa, za jezik C (kao i za svaki drugi viši programski jezik), programeru su na raspolaganju mnoga sredstva koja omogućavaju da program bude kraći, pregledniji, da se efikasnije kompilira, da se brže izvršava itd.

Osnovni koraci u organizovanju složenijih programa obično su podela složenih zadataka na jednostavnije poslove i njihovo izdvajanje u zasebne funkcije (tzv. *funkcionalna dekompozicija*), definisanje odgovarajućih tipova podataka i organizovanje podataka definisanjem odgovarajućih promenljivih. Funkcije u programu trebalo bi da budu što nezavisnije i što slabije međusobno uslovljene (loše je ako je za razumevanje rada neke funkcije potrebno potpuno razumeti tačno kako radi neka druga funkcija). Veliki programi se organizuju u više datoteka tj. *modula* koji sadrže podatke i funkcije koji objedinjavaju određenu funkcionalnost (npr. definiciju tipa podataka za predstavljanje kompleksnih brojeva i funkcije za izvođenje operacija nad tim tipom ima smisla grupisati u zaseban modul za rad sa kompleksnim brojevima). Mnogi moduli mogu biti korišćeni u različitim programima i tada se obično grade kao biblioteke.

Opisani pristup programiranju omogućavaju i olakšavaju različiti mehanizmi jezika C. Na primer, pretprocesor, između ostalog, olakšava korišćenje standardne, ali i korisnički definisanih biblioteka. Pretprocesor sa linkerom omogućava da se program podeli na više datoteka, tako da se te datoteke mogu spojiti u jedinstven izvršni program. *Doseg identifikatora* određuje da li se neka imena mogu koristiti u čitavim jedinicama prevođenja ili samo u njihovim manjim delovima (najčešće funkcijama ili još užim blokovima). Postojanje promenljivih čija je upotreba ograničena na samo određene uske delove

izvornog kôda olakšava razumevanje programa i smanjuje mogućnost grešaka i smanjuje međusobnu zavisnost između raznih delova programa. *Životni vek promenljivih* određuje da li je neka promenljiva dostupna tokom čitavog izvršavanja programa ili samo tokom nekog njegovog dela. Postojanje promenljivih koje traju samo pojedinačno izvršavanje neke funkcije znatno štedi memoriju, dok postojanje promenljivih koje traju čitavo izvršavanje programa omogućava da se preko njih vrši komunikacija između različitih funkcija modula. *Povezanost identifikatora* u vezi je sa deljenjem podataka između različitih jedinica prevođenja i daje mogućnost korišćenja zajedničkih promenljivih i funkcija u različitim jedinicama prevođenja (modulima), ali i mogućnost sakrivanja nekih promenljivih tako da im se ne može pristupiti iz drugih jedinica prevođ enja. Odnos između dosega, životnog veka i povezanosti je veoma suptilan i sva ova tri aspekta objekata određuju se na osnovu mesta i načina deklarisanja tj. definisanja objekata, ali i primenom kvalifikatora `auto`, `register`, `static` i `external` o čemu će biti više reči u nastavku ovog poglavlja.

Sva ova sredstva donose potencijalna olakšanja u proces programiranja, ali sva stvaraju i mogućnosti raznih grešaka ukoliko se ne koriste na ispravan način.

9.2.1 Pretprocesor

Kada se od izvornog programa napisanog na jeziku C proizvodi program na mašinskom jeziku (izvršni program), pre samog prevodioca poziva se C pretprocesor. Pretprocesiranje, dakle, predstavlja samo pripremnu fazu, pre kompilacije. Suštinski, pretprocesor vrši samo jednostavne operacije nad tekstualnim sadržajem programa i ne koristi nikakvo znanje o jeziku C. Pretprocesor ne analizira značenje naredbi napisanih u jeziku C već samo *pretprocesorske direktive* (engl. preprocessing directive) (one se ne smatraju delom jezika C) na osnovu kojih vrši određene transformacije teksta izvornog programa. Neke od operacija pretprocesora su zamena komentara belinama ili spajanje linija razdvojenih u izvornom programu simbolom \. Dve najčešće korišćene pretprocesorske direktive su `#include` (za uključivanje sadržaja neke druge datoteke) i `#define` koja zamenjuje neki tekst, *makro*, drugim tekstom. Pretprocesor omogućava i definisanje makroa sa argumentima, kao i uslovno prevođenje (određeni delovi izvornog programa se prevode samo ukoliko su ispunjeni zadati uslovi). Kako bi mogao da izvrši svoje zadatke, tokom izvršavanja pretprocesor izvršava određenu jednostavniju leksičku analizu (tokenizaciju) na koju se nadovezuje kasnija faza leksičke analize, koja se vrši tokom kompilacije.

Uključivanje datoteka zaglavlja

Veliki programi obično su podeljeni u više datoteka, radi preglednosti i lakšeg održavanja. Često je potrebno da se iste promenljive, funkcije i korisnički definisani tipovi koriste u više datoteka i neophodno je da kompilator prilikom prevođenja svake od njih poznaje deklaracije[5] promenljivih, funkcija i tipova

[5] Podsetimo, deklaracije sadrže informacije koje su potrebne kompilatoru da prevede određenu datoteku i tako generiše objektni modul (npr. prototip funkcije je deklaracija), dok definicije sadrže mnogo više informacija – informacije koje su dovoljne da nakon faze povezivanja nastane izvršni program (npr. ceo kôd funkcije predstavlja njenu definiciju). Kompilator ne mora da poznaje definiciju funkcije da bi generisao njen poziv, ali linker mora, da bi mogao da taj poziv razreši.

podataka koji se u njoj koriste. Čak i kada programer piše samo jednu datoteku, program po pravilu koristi i funkcionalnost koju mu pruža standardna biblioteka i kako bi ona mogla da se koristi, neophodno je da kompilator tokom kompilacije zna deklaracije standardnih funkcija (npr. funkcije `printf`), promenljivih i tipova. Jedno rešenje bilo bi da programer u svakoj datoteci, na početku navede sve potrebne deklaracije, ali ponavljanje istih deklaracija na više mesta u izvornom kôdu je mukotrpan posao i otvara prostor za greške. Bolje rešenje (koja se obično i koristi) je da se deklaracije izdvajaju u zasebne datoteke koje se onda uključuju gde god je potrebno. Takve datoteke zovu se *datoteke zaglavlja* (engl. header files). Osnovni primer datoteka zaglavlja su zaglavlja standardne biblioteke. Primer kreiranja korisnički definisanih datoteka zaglavlja biće naveden u poglavlju 9.2.6. U datoteku koja se prevodi, sadržaj neke druge datoteke uključuje se direktivom `#include`. Linija oblika:

```
#include "ime_datoteke"
```

i linija oblika

```
#include <ime_datoteke>
```

zamenjuju se sadržajem datoteke `ime_datoteke`. U prvom slučaju, datoteka koja se uključuje traži se u okviru posebnog skupa direktorijuma *include path* (koja se većini kompilatora zadaje korišćenjem opcije `-I`) i koja obično podrazumevano sadrži direktorijum u kojem se nalazi datoteka u koju se vrši uključivanje. Ukoliko je ime, kao u drugom slučaju, navedeno između znakova `<` i `>`, datoteka se traži u sistemskom *include* direktorijumu u kojem se nalaze standardne datoteke zaglavlja, čija lokacija zavisi od sistema i od C prevodioca koji se koristi.

Ukoliko se promeni uključena datoteka, sve datoteke koje zavise od nje moraju biti iznova prevedene. Datoteka koja se uključuje i sama može sadržati direktive `#include`.

Pošto, zahvaljujući `#include` direktivi, do kompilatora stiže program koji ne postoji fizički u jednoj datoteci, već je kombinacija teksta koji se nalazi u različitim datotekama, kaže se da se program sastoji od različitih *jedinica prevođenja* (a ne od više datoteka). Jedinice prevođenja imaju mnogo direktniju vezu sa objektnim modulima, povezivanjem i izvršnim programom od pojedinačnih fizičkih datoteka koje programer kreira.

Makro zamene

Pretprocesorska direktiva `#define` omogućava zamenjivanje niza karaktera u datoteci, *makroa* (engl. macro) drugim nizom karaktera pre samog prevođenja. Njen opšti oblik je:

```
#define originalni_tekst novi_tekst
```

U najjednostavnijem obliku, ova direktiva koristi se za zadavanje vrednosti nekom simboličkom imenu, na primer:

```
#define MAX_LEN 80
```

Ovakva definicija se koristi kako bi se izbeglo navođenje iste konstantne vrednosti na puno mesta u programu. Umesto toga, koristi se simboličko ime koje se može lako promeniti — izmenom na samo jednom mestu. U navedenom primeru, `MAX_LEN` je samo simboličko ime i nikako ga ne treba mešati sa promenljivom (čak ni sa promenljivom koja je `const`). Naime, za ime `MAX_LEN` u memoriji se ne rezerviše prostor tokom izvršavanja programa, već se svako njeno pojavljivanje, pre samog prevođenja zamenjuju zadatom vrednošću (u navedenom primeru — vrednošću 80). Tako `MAX_LEN` nije vidljiva ni kompilatoru, ni linkeru, niti u bilo kom obliku postoji u izvršnom programu.

Makro može biti bilo koji identifikator, pa čak i ključna reč jezika C. Tekst zamene (`novi_tekst` u navedenom opštem obliku) je tekst do kraja reda, a moguće je da se prostire na više redova ako se navede simbol \ na kraju svakog reda koji se nastavlja. Direktive zamene mogu da koriste direktive koje im prethode.

Zamene se ne vrše u konstantnim niskama niti drugim simboličkim imenima[6]. Na primer, gore navedena direktiva neće uticati na naredbu `printf( "MAX_LEN is 80");` niti na simboličko ime `MAX_LEN_VAL`.

Moguće je defisati i pravila zamene sa argumentima od kojih zavisi tekst zamene. Na primer, sledeća definicija

```
#define  max(A, B)  ((A) > (B) ? (A) : (B))
```

definiše tekst zamene za `max(A, B)` koji zavisi od argumenata. Ukoliko je u nastavku datoteke, u nekoj naredbi navedeno `max(2, 3)`, to će, pre prevođenja, biti zamenjeno sa `((2) > (3) ? (2) : (3))`. Slično, tekst `max(x+2, 3*y)` biće zamenjen tekstom `((x+2) > (3*y) ? (x+2) : (3*y))`. Tekst `max(2, 3)` i slični ne predstavljaju poziv funkcije i nema nikakvog prenosa argumenata kao kod pozivanja funkcija. Postoje i druge razlike u odnosu na poziv funkcije. Na primer, ukoliko je negde u programu navedeno `max(a++, b++)`, na osnovu date definicije, ovaj tekst biće zamenjen tekstom

`((a++) > (b++) ? (a++) : (b++))`,

što će dovesti do toga da se veća od vrednosti `a` i `b` inkrementira dva puta (što možda nije planirano).

Važno je voditi računa i o zagradama u tekstu zamene, kako bi bio očuvan poredak primene operacija. Na primer, ukoliko se definicija

```
#define kvadrat(x)  x*x
```

primeni na `kvadrat(a+2)`, tekst zamene će biti `a+2*a+2`, a ne `(a+2)*(a+2)`, kao što je verovatno željeno i zbog toga bi trebalo koristiti:

```
#define kvadrat(x)  (x)*(x)
```

Navedena definicija, međutim, i dalje ne daje ponašanje koje je verovatno željeno. Naime, kada se makro primeni na izraz `a/kvadrat(b)`, on će biti zamenjen izrazom `a/(b)*(b)`, što je ekvivalentno sa `(a/b)*b` (a ne sa sa `a/(b*b)`). Zbog toga je bolja definicija:

```
#define kvadrat(x)  ((x)*(x))
```

[6] Zato je neophodno da pretprocesor tokom svog rada izvrši jednostavniju leksičku analizu.

Tekst zamene može da sadrži čitave blokove sa deklaracijama, kao u sledećem primeru:

```
#define swap(t, x, y) { t z; z=x; x=y; y=z; }
```

koji definiše zamenjivanje vrednosti dve promenljive tipa `t`. Celobrojnim promenljivama `a` i `b` se, zahvaljujući ovom makrou, mogu razmeniti vrednosti sa `swap(int, a, b)`.

Ukoliko se u direktivi `#define`, u novom tekstu, ispred imena parametra navede simbol `#`, kombinacija će biti zamenjena vrednošću parametra navedenog između dvostrukih navodnika. Ovo može da se kombinuje sa nadovezivanjem niski. Na primer, naredni makro može da posluži za otkrivanje grešaka:

```
#define  dprint(expr)   printf(#expr " = %g\n", expr)
```

Tako će tekst:

```
dprint(x/y);
```

unutar nekog programa biti zamenjen tekstom:

```
printf("x/y" " = %g\n", x/y);
```

Pošto se niske automatski nadovezuju, efekat ove naredbe je isti kao efekat naredbe:

```
printf("x/y = %g\n", x/y);
```

Česta je potreba da se prilikom pretprocesiranja dve niske nadovežu kako bi se izgradio složeni identifikator. Ovo se postiže petprocesorskim operatorom `##`. Na primer,

```
#define dodaj_u_niz(ime, element)  \
   niz_##ime[brojac_##ime++] = element
```

Poziv

```
dodaj_u_niz(a, 3);
```

se tada zamenjuje naredbom

```
niz_a[brojac_a++] = 3;
```

Dejstvo primene direktive zamene je od mesta na kojem se nalazi do kraja datoteke ili do reda oblika:

```
#undef originalni_tekst
```

Kao što je rečeno, makro zamene i funkcije se, iako mogu izgledati slično, suštinski razlikuju. Kod makro zamena nema provere tipova argumenata niti implicitnih konverzija što može da dovodi do grešaka u kompilaciji ili do neočekivanog rada programa. Argument makroa u zamenjenoj verziji može biti naveden više puta što može da utiče na izvršavanje programa, a izostavljanje zagrada u tekstu zamene može da utiče na redosled izračunavanja operacija. S druge strane, kod poziva makroa nema prenosa argumenata te se oni izvršavaju brže nego odgovarajuće funkcije. Ipak, moderni kompilatori kratke funkcije

obično *inlajnuju* (umeću čitavo telo funkcije na mesto poziva, starajući se o argumentima) i tada nema gubitaka u vremenskoj ili prostornoj efikasnosti u odnosu na korišćenje makroa. Sve u svemu, makro zamene sa argumentima mogu imaju i dobre i loše strane i treba ih koristiti oprezno.

Mnoge standardne „funkcije“ za ulaz i izlaz iz standardne C biblioteke su zapravo makroi (na primer, `getchar` koji koristi funkciju `getc`, `printf` koji koristi funkciju `fprintf`, itd.).

Uslovno prevođenje

Pretprocesorskim direktivama je moguće isključiti delove kôda iz procesa prevođenja, u zavisnosti od vrednosti uslova koji se računa u fazi pretprocesiranja. Direktiva `#if` izračunava vrednost konstantnog celobrojnog izraza (koji može, na primer, da sadrži konstante i simbolička imena definisana direktivom `#define`). Ukoliko je dobijena vrednost jednaka nuli, onda se ignorišu (ne prevode) sve linije programa do direktive `#endif` ili do direktive `#else` ili do direktive `#elif` koja ima značenje kao `else-if`. U okviru argumenta direktive `#if`, može se koristiti izraz `defined(ime)` koji ima vrednost 1 ako je simboličko ime `ime` definisano nekom prethodnom direktivom, a 0 inače. Kraći zapis za `#if defined(ime)` je `#ifdef ime`, a kraći zapis za `#if !defined(ime)` je `#ifndef ime`. Na primer, sledeći program:

Program 9.1.

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

int main()
{
  #ifdef SRPSKI
    printf("Zdravo, svete");
  #else
    printf("Hello, world");
  #endif
  return 0;
}
```

ispisuje tekst na srpskom jeziku, a izostavljanjem direktive iz prvog reda ispisivao bi tekst na engleskom jeziku.[7] Ovo grananje se suštinski razlikuje od grananja zasnovanog na C naredbi `if`. Naime, u navedenom primeru prevodi se samo jedna od dve verzije programa i dobijeni izvršni program nema nikakvu informaciju o drugoj verziji. Za razliku od toga, kada se koristi grananje koje koristi C jezik (a ne pretprocesor), program sadrži kôd za sve moguće grane.

Pitanja i zadaci za vežbu

Pitanje 9.2.1.

1. *Čemu služi direktiva* `#include`*?*

[7] Za lokalizaciju programa bolje je koristiti specijalizovane alate — npr. GNU gettext.

2. *Gde se traži datoteka* `filename` *ako se uključi pretprocesorskom direktivom* `#include "filename"`*, a gde ako se uključi direktivom* `#include <filename>`*? Kada se obično koristi direktiva* `#include "filename"`*, a kada direktiva* `#include <filename>` *?*

Pitanje 9.2.2.

1. *Dokle važi dejstvo pretprocesorske direktive* `#define NUM_LINES 1000`*?*
2. *Kako se poništava dejstvo pretprocesorske direktive*

   `#define NUM_LINES 1000`*?*
3. *Kako se od neke tačke u programu poništava dejstvo pretprocesorske direktive* `#define x(y) y*y+y`*?*

Pitanje 9.2.3. *Kako se može proveriti da li je neko simboličko ime definisano u trenutnoj jedinici prevođenja?*

Pitanje 9.2.4. *Objasniti razliku između makroa i funkcija. U kojoj fazi se vrši razrešavanje poziva makroa, a u kojoj poziva funkcija? Objasniti šta znači da kod makroa nema prenosa argumenata, a kod funkcija ima.*

Pitanje 9.2.5. *Kojoj operaciji u tekstualnom editoru je sličan efekat direktive* `#define`*, a kojoj efekat direktive* `#include`*?*

Pitanje 9.2.6. *Kojim parametrom se GCC navodi da izvrši samo pretprocesiranje i prikaže rezultat faze pretprocesiranja?*

Pitanje 9.2.7. *Za svaki od programa koji slede, navesti šta je rezultat pretprocesiranja programa i šta se ispisuje kada se program prevede i izvrši.*

1. ```
   #include <stdio.h>
   #define x(y) y*y+y
   int main() { printf("%d", x(3 + 5)); /* ispis */ }
   ```
2. ```
   #include <stdio.h>
   #define A(x,y) (x > 2) ? x*y : x-y
   int main() { printf("%d %d", A(4,3+2), A(4,3*2); }
   ```
3. ```
   #include <stdio.h>
   #define A(x,y) (y>=0) ? x+y : x-y
   int main() {
      if (b<=0) c = A(a+b,b); else c = A(a-b,b);
      printf("%d\n", c);
   }
   ```
4. ```
   #include <stdio.h>
   #define proizvod(x,y) x*y
   int main() { printf("%d\n", proizvod(2*3+4, 2+4*5)); }
   ```
5. ```
   #include <stdio.h>
   #define A(x,y) (x > 2) ? x*y : x-y
   int main() { printf("%d %d\n", A(4,3+2), A(4,3*2)); }
   ```

Pitanje 9.2.8.
   ```

1. *Navesti primer poziva makroa* `kvadrat` *za kvadriranje koji pri definiciji* `#define kvadrat(x) x*x` *daje pogrešan rezultat dok pri definiciji* `#define kvadrat(x) ((x)*(x))` *daje ispravan rezultat.*
2. *Navesti primer poziva makroa* `dvostruko` *za koji pri definiciji* `#define dvostruko(x) (x)+(x)` *daje pogrešan rezultat dok pri definiciji* `#define dvostruko(x) ((x)+(x))` *daje ispravan rezultat.*

Pitanje 9.2.9. *Napisati makro za:*

1. *izračunavanje ostatka pri deljenju prvog argumenta drugim,*
2. *izračunavanje trećeg stepena nekog broja,*
3. *izračunavanje maksimuma dva broja,*
4. *izračunavanje maksimuma tri broja,*
5. *izračunavanje apsolutne vrednosti broja,*

9.2.2 Doseg identifikatora

Jedna od glavnih karakteristika dobrih programa je da se promenljive većinom definišu u funkcijama (ili čak nekim užim blokovima) i upotreba promenljive je uglavnom ograničena na funkciju (ili blok) u kojoj je deklarisana. Ovim se smanjuje zavisnost između funkcija i ponašanje funkcije određeno je samo njenim ulaznim parametrima, a ne nekim globalnim stanjem programa. Time se omogućava i da se analiza rada programa zasniva na analizi pojedinačnih funkcija, nezavisnoj od konteksta celog programa. Ipak, u nekim slučajevima prihvatljivo je da funkcije međusobno komuniciraju korišćenjem zajedničkih promenljivih. *Doseg*, tj. vidljivost identifikatora određena je načinom tj. mestom u izvornom kôdu na kojem su uvedeni.

Doseg identifikatora (engl. scope of identifiers) određuje deo teksta programa u kojem je moguće koristiti određeni identifikator i u kojem taj identifikator identifikuje određeni objekat (na primer, promenljivu ili funkciju). Svaki identifikator ima neki doseg. Jezik C spada u grupu jezika sa statičkim pravilima dosega što znači da se doseg svakog identifikatora može jednoznačno utvrditi analizom izvornog kôda (bez obzira na moguće tokove izvršavanja programa). U jeziku C postoje sledeće vrste dosega:

- *doseg nivoa datoteke* (engl. file level scope) koji podrazumeva da ime važi od tačke uvođenja do kraja datoteke;
- *doseg nivoa bloka* (engl. block level scope) koji podrazumeva da ime važi od tačke uvođenja do kraja bloka u kojem je uvedeno;
- *doseg nivoa funkcije* (engl. function level scope) koji podrazumeva da ime važi u celoj funkciji u kojoj je uvedeno; ovaj doseg imaju jedino labele koje se koriste uz `goto` naredbu;
- *doseg nivoa prototipa funkcije* (engl. function prototype scope) koji podrazumeva da ime važi u okviru prototipa (deklaracije) funkcije; ovaj doseg imaju samo imena parametara u okviru prototipova funkcije; on omogućava da se u prototipovima funkcija navode i imena (a ne samo tipovi)

argumenata, što nije obavezno, ali može da olakša razumevanje i dokumentovanje kôda.

Najznačajni nivoi dosega su doseg nivoa datoteke i doseg nivoa bloka. Identifikatori koji imaju doseg nivoa datoteke najčešće se nazivaju *spoljašnji* ili *globalni*, dok se identifikatori koji imaju ostale nivoe dosega (najčešće doseg nivoa bloka) nazivaju *unutrašnji* ili *lokalni*.[8] Na osnovu diskusije sa početka ovog poglavlja, jasno je da je poželjno koristiti identifikatore promenljivih lokalnog dosega kada god je to moguće. S druge strane, funkcije su obično globalne. Imajući u vidu podelu na globalne i lokalne objekte, C program se često definiše kao skup globalnih objekata (promenljivih, funkcija, tipova podataka itd.).

U ranim verzijama jezika C nije bilo moguće definisati funkciju u okviru definicije druge funkcije, tj. funkciju sa dosegom nivoa bloka, pa nije bilo *lokalnih funkcija*, dok je od standarda C99 i takva mogućnost predviđena. Doseg globalnih funkcija je doseg nivoa datoteke i proteže se od mesta deklaracije pa do kraja datoteke.

U nastavku su dati primeri različitih dosega:

```
int a;
/* a je globalna promenljiva - doseg nivoa datoteke */

/* f je globalna funkcija - doseg nivoa datoteke */
void f(int c) {
  /* c je lokalna promenljiva -
     doseg nivoa bloka (tela funkcije f) */
  int d;
  /* d je lokalna promenljiva -
     doseg nivoa bloka (tela funkcije f) */

  void g() { printf("zdravo"); }
  /* g je lokalna funkcija -
     doseg nivoa bloka (tela funkcije f) */

  for (d = 0; d < 3; d++) {
     int e;
     /* e je lokalna promenljiva -
        doseg nivoa bloka (tela petlje) */
     ...
  }
  kraj:
  /* labela kraj - doseg nivoa funkcije */
}


/* h je globalna funkcija - doseg nivoa datoteke */
void h(int b); /* b - doseg nivoa prototipa funkcije */
```

[8] Iako tekst standarda koristi termine spoljašnji i unutrašnji, u nastavku teksta će u većini biti korišćeni termini globalni i lokalni, kako bi se izbegla zabuna sa spoljašnjom i unutrašnjom povezanošću objekata.

Jezik C dopušta tzv. konflikt identifikatora tj. moguće je da postoji više identifikatora istog imena. Ako su njihovi dosezi jedan u okviru drugog, tada identifikator u užoj oblasti dosega sakriva identifikator u široj oblasti dosega. Na primer, u narednom programu, promenljiva `a` inicijalizovana na vrednost `5` sakriva promenljivu `a` inicijalizovanu na vrednost `3`.

```
void f() {
   int a = 3, i;
   for (i = 0; i < 4; i++) {
     int a = 5;
     printf("%d ", a);
   }
}
```

```
5 5 5 5
```

Ovim je omogućeno da prilikom uvođenja novih imena programer ne mora da brine da li je takvo ime već upotrebljeno u širem kontekstu.

9.2.3 Životni vek objekata i kvalifikatori `static` i `auto`

U određenoj vezi sa dosegom, ali ipak nezavisno od njega je pitanje trajanja objekata (pre svega promenljivih). *Životni vek (engl. storage duration, lifetime)* promenljive je deo vremena izvršavanja programa u kojem se garantuje da je za tu promenljivu rezervisan deo memorije i da se ta promenljiva može koristiti. U jeziku C postoje sledeće vrste životnog veka:

- *statički (engl. static)* životni vek koji znači da je objekat dostupan tokom celog izvršavanja programa;
- *automatski (engl. automatic)* životni vek koji najčešće imaju promenljive koje se automatski stvaraju i uklanjaju prilikom pozivanja funkcija;
- *dinamički (engl. dynamic)* životni vek koji imaju promenljive koje se alociraju i dealociraju na eksplicitan zahtev programera (videti poglavlje 10.9).

Životni vek nekog objekta se određuje na osnovu pozicije u kôdu na kojoj je objekat uveden i na osnovu eksplicitnog korišćenja nekog od kvalifikatora `auto` (automatski životni vek) ili `static` (statički životni vek).

Lokalne automatske promenljive

Najčešće korišćene lokalne promenljive su promenljive deklarisane u okviru bloka. Rekli smo da je njihov doseg nivoa bloka, one su dostupne od tačke deklaracije do kraja bloka i nije ih moguće koristiti van bloka u kojem su deklarisane. Ne postoji nikakva veza između promenljivih istog imena deklarisanih u različitim blokovima.

Lokalne promenljive podrazumevano su *automatskog* životnog veka (sem ako je na njih primenjen kvalifikator `static` ili kvalifikator `extern` — o čemu će biti reči u narednim poglavljima). Iako je moguće i eksplicitno okarakterisati životni vek korišćenjem kvalifikatora `auto`, to se obično ne radi jer se za ovakve

promenljive automatski životni vek podrazumeva. Početna vrednost lokalnih automatskih promenljivih nije određena i zavisi od ranijeg sadržaja memorijske lokacije koja je pridružena promenljivoj, pa se smatra nasumičnom.

Automatske promenljive „postoje“ samo tokom izvršavanja funkcije u kojoj su deklarisane i prostor za njih je rezervisan u stek okviru te funkcije (videti poglavlje 9.3.3). Ne postoji veza između promenljive jednog imena u različitim aktivnim instancama jedne funkcije (jer svaka instanca funkcije ima svoj stek okvir i u njemu prostor za promenljivu tog imena). Dakle, ukoliko se u jednu funkciju uđe rekurzivno, kreira se prostor za novu promenljivu, potpuno nezavisan od prostora za prethodnu promenljivu istog imena.

Formalni parametri funkcija, tj. promenljive koje prihvataju argumente funkcije imaju isti status kao i lokalne automatske promenljive.

Na lokalne automatske promenljive i parametre funkcija moguće je primeniti i kvalifikator `register` čime se kompilatoru sugeriše da se ove promenljive čuvaju u registrima procesora, a ne u memoriji (o organizaciji izvršnog programa i organizaciji memorije tokom izvršavanja programa biće više reči u poglavlju 9.3.3). Međutim, s obzirom na to da današnji kompilatori tokom optimizacije kôda veoma dobro mogu da odrede koje promenljive ima smisla čuvati u registrima, ovaj kvalifikator se sve ređe koristi.

Globalne statičke promenljive i funkcije

Globalne promenljive deklarisane su van svih funkcija i njihov doseg je nivoa datoteke tj. mogu se koristiti od tačke uvođenja, u svim funkcijama do kraja datoteke. Životni vek ovih promenljivih uvek je *statički*, tj. prostor za ove promenljive rezervisan je tokom celog izvršavanja programa: prostor za njih se rezerviše na početku izvršavanja programa i oslobađa onda kada se završi izvršavanje programa. Prostor za ove promenljive obezbeđuje se u segmentu podataka (videti poglavlje 9.3.3). Ovakve promenljive se podrazumevano inicijalizuju na vrednost 0 (ukoliko se ne izvrši eksplicitna inicijalizacija).

S obzirom na to da ovakve promenljive postoje sve vreme izvršavanja programa i na to da može da ih koristi više funkcija, one mogu da zamene prenos podataka između funkcija. Međutim, to treba činiti samo sa dobrim razlogom (na primer, ako najveći broj funkcija treba da koristi neku zajedničku promenljivu) jer, inače, program može postati nečitljiv i težak za održavanje.

Globalne promenljive uvek imaju statički vek. Na njih je moguće primeniti kvalifikator `static`, međutim, on ne služi da bi se naglasio statički životni vek (to se podrazumeva), već naglašava da one imaju unutrašnju povezanost (videti poglavlje 9.2.4).

Lokalne statičke promenljive

U nekim slučajevima poželjno je čuvati informaciju između različitih poziva funkcije (npr. potrebno je brojati koliko puta je pozvana neka funkcija). Jedno rešenje bilo bi uvođenje globalne promenljive, statičkog životnog veka, međutim, zbog globalnog dosega tu promenljivu bilo bi moguće koristiti (i promeniti) i iz drugih funkcija, što je nepoželjno. Zato je poželjna mogućnost definisanja promenljivih koje bi bile statičkog životnog veka (kako bi čuvale vrednost tokom čitavog izvršavanja programa), ali lokalnog dosega (kako bi se mogle koristiti i menjati samo u jednoj funkciji).

U deklaraciji lokalne promenljive može se primeniti kvalifikator `static` i u tom slučaju ona ima statički životni vek — kreira se na početku izvršavanja programa i oslobađa prilikom završetka rada programa. Tako modifikovana promenljiva ne čuva se u stek okviru svoje funkcije, već u segmentu podataka (videti poglavlje 9.3.3). Ukoliko se vrednost statičke lokalne promenljive promeni tokom izvršavanja funkcije, ta vrednost ostaje sačuvana i za sledeći poziv te funkcije. Ukoliko inicijalna vrednost statičke promenljive nije navedena, podrazumeva se vrednost `0`. Statičke promenljive se inicijalizuju samo jednom, konstantnim izrazom, na početku izvršavanja programa tj. prilikom njegovog učitavanja u memoriju[9]. Doseg ovih promenljivih i dalje je nivoa bloka tj. promenljive su i dalje lokalne, što daje željene osobine.

Naredni program ilustruje kako se promenljiva `a` u funkciji `f` iznova kreira tokom svakog poziva, dok promenljiva `a` u funkciji `g` zadržava svoju vrednost tokom raznih poziva.

Program 9.2.

```
#include <stdio.h>

void f() {
  int a = 0;
  printf("f: %d ", a);
  a = a + 1;
}

void g() {
  static int a = 0;
  printf("g: %d ", a);
  a = a + 1;
}

int main() {
  f(); f();
  g(); g();
  return 0;
}
```

Kada se prevede i pokrene, prethodni program ispisuje:

```
f: 0 f: 0 g: 0 g: 1
```

9.2.4 Povezanost identifikatora i kvalifikatori `static` i `extern`

Pravila dosega daju mogućnost progameru da odredi da li želi da je neko ime vidljivo samo u jednoj ili u više funkcija definisanih u jednoj jedinici prevođenja. Međutim, kada se program sastoji od više jedinica prevođ enja, doseg nije dovoljan i potrebna je malo finija kontrola. Poželjno je da postoji mogućnost definisanja: (i) objekata (promenljivih ili funkcija) koji se mogu koristiti samo u pojedinačnim funkcijama, (ii) objekata koji se mogu koristiti u više funkcija

[9] Ovakvo ponašanje je različito u odnosu na jezik C++ gde se inicijalizacija vrši prilikom prvog ulaska u blok u kojem je ovakva promenljiva definisana.

neke jedinice prevođenja, ali ne i van te jedinice prevođenja i (iii) objekata koji se mogu koristiti u svim funkcijama u celom programu, moguće i u različitim jedinicama prevođenja.

Identifikator koji je deklarisan u različitim dosezima ili u istom dosegu više puta može da označava isti objekat ili različite objekte. Šta će biti slučaj, određuje se na osnovu *povezanosti identifikatora* (engl. linkage of identifiers). Povezanost identifikatora tesno je vezana za fazu povezivanja programa i najčešće se koristi da odredi međusobni odnos objekata u različitim jedinicama prevođ enja. Između ostalog, ona daje odgovor na pitanje da li je i kako moguće objekte definisane u jednoj jedinici prevođenja koristiti u nekoj drugoj jedinici prevođenja. Na primer, da li je dopušteno da se u dve različite jedinice prevođenja koristi identifikator `a` za neku globalnu celobrojnu promenljivu i da li taj identifikator `a` označava jednu istu ili dve različite promenljive.

Jezik C razlikuje identifikatore:

- *bez povezanosti* (engl. no linkage),
- identifikatore sa *spoljašnjom povezanošću* (engl. external linkage) i
- identifikatore sa *unutrašnjom povezanošću* (engl. internal linkage).

Ne treba mešati spoljašnju i unutrašnju povezanost sa spoljašnjim i unutrašnjim dosegom (otuda i korišćenje alternativnih termina globalni i lokalni za doseg) — povezanost (i unutrašnja i spoljašnja) se odnosi isključivo na globalne objekte (tj. objekte spoljašnjeg dosega), dok su lokalni objekti (tj. objekti unutrašnjeg dosega) najčešće bez povezanosti (osim ako im se povezanost ne promeni kvalifikatorom `extern` o čemu će biti reči u nastavku). Unutrašnja i spoljašnja povezanost identifikatora ima smisla samo kod objekata sa statičkim životnim vekom, dok su objekti automatskog životnog veka bez povezanosti.

Kako bi se povezanost potpuno razumela, potrebno je skrenuti pažnju na još jedan značajan aspekt jezika C i razjasniti šta se podrazumeva pod *deklaracijom*, a šta pod *definicijom* objekta. Podsetimo se, definicija objekta prouzrokuje da kompilator rezerviše određeni memorijski prostor za objekat, dok deklaracija kompilatoru daje samo informaciju o tipu objekta, dok se prilikom izgradnje izvršnog programa zahteva da pored deklaracija negde postoji i definicija. Objekti mogu da imaju veći broj (istovetnih) deklaracija, ali mogu da imaju samo jednu definiciju. U slučaju funkcija, pitanje šta je deklaracija, a šta je definicija jasno je na osnovu toga da li prototip prati i telo funkcije. Slično, svaka deklaracija lokalne promenljive koja je do sada prikazane bila je ujedno i njena definicija. Međutim, to je nešto drugačije kod globalnih promenljivih. Naime, ako deklaraciju prati inicijalizacija, ona se uvek ujedno smatra i definicijom. Ali, ako je globalna promenljiva uvedena bez inicijalizacije (npr. ako postoji globalna deklaracija `int a;`), nije jasno da li je u pitanju deklaracija ili definicija, već se smatra *uslovnom definicijom* (engl. tentative definition). Ako postoji prava definicija iste promenljive (na primer, deklaracija sa inicijalizacijom), onda se uslovna definicija smatra samo deklaracijom. Međutim, ako prave definicije nema, onda se uslovna definicija smatra definicijom (uz inicijalnu vrednost 0), tj. ponašanje je isto kao da pored uslovne definicije postoji prava definicija sa inicijalizatorom 0. Takođe, ako se pri deklaraciji promenljive (bilo lokalne, bilo globalne) navede kvalifikator `extern` (o kome će biti reči u nastavku) ta deklaracija obavezno prestaje da bude definicija (i tada nije dopušteno navoditi inicijalnu vrednost).

Identifikatori bez povezanosti

Identifikatori bez povezanosti nisu vidljivi prilikom procesa povezivanja i mogu se potpuno nezavisno ponavljati u različitim funkcijama, bilo da su one navedene u jednoj ili u više jedinica prevođenja. Svaka deklaracija identifikatora bez povezanosti ujedno je i njegova definicija i ona određuje jedinstveni nezavisni objekat. Bez povezanosti su najčešće lokalne promenljive (bez obzira da li su automatskog ili statičkog životnog veka), parametri funkcija, korisnički definisani tipovi, labele itd. Na primer, ako su funkcije `f`, funkcije `g1` i `g2` i struktura `i` definisane u različitim jedinicama prevođenja, sva pojavljivanja imena `i` i sva pojavljivanja imena `j` označavaju različite, potpuno nezavisne promenljive i tipove.

```
int f() {          int g1(int i) {          struct i { int a; }
   int i;             static int j;
   ...             }
}                  int g2() {
                      static int i, j;
                      ...
                   }
```

Spoljašnja povezanost i kvalifikator `extern`

Spoljašnja povezanost identifikatora omogućava da se isti objekat koristi u više jedinica prevođenja. Sve deklaracije identifikatora sa spoljašnjom povezanošću u skupu jedinica prevođenja određuju jedan isti objekat (tj. sve pojave ovakvog identifikatora u različitim jedinicama prevođenja odnose se na jedan isti objekat), dok u celom programu mora da postoji tačno jedna definicija tog objekta. Tako se jedan identifikator koristi za isti objekat u okviru više jedinica prevođenja.

Kvalifikator `extern` najčešće se koristi isključivo kod programa koji se sastoje od više jedinica prevođenja i služi da naglasi da neki identifikator ima spoljašnju povezanost. Jedino deklaracije mogu biti okvalifikovane kvalifikatorom `extern`. Samim tim, nakon njegovog navođenja nije moguće navoditi inicijalizaciju promenljivih niti telo funkcije (što ima smisla samo prilikom definisanja). Slično, za niz u `extern` deklaraciji nije potrebno navoditi dimenziju. Pošto `extern` deklaracije nisu definicije, njima se ne rezerviše nikakva memorija u programu već se samo naglašava da se očekuje da negde (najčešće u drugim jedinicama prevođenja) postoji definicija objekta koji je se `extern` deklaracijom deklariše (i time uvodi u odgovarajući doseg).

Ipak, postoje slučajevi kada se spoljašnja povezanost podrazumeva i nije neophodno eksplicitno upotrebiti `extern` deklaraciju. Sve globalne funkcije jezika C podrazumevano imaju spoljašnju povezanost (osim ako na njih nije primenjen kvalifikator `static` kada je povezanost unutrašnja) i zato se prilikom njihove deklaracije retko kad eksplicitno navodi kvalifikator `extern`. Slično kao kod funkcija, podrazumevana povezanost globalnih promenljivih je spoljašnja (osim ako na njih nije primenjen kvalifikator `static` kada je povezanost unutrašnja). Ranije je već rečeno da deklaracije globalnih promenljivih (oblika npr. `int x;`), ako nisu praćene inicijalizacijom, predstavljaju samo „uslovne“ definicije, tj. ako negde postoji prava definicija objekta sa tim imenom, one nisu definicije već samo deklaracije. U tom svetlu, kvalifikator `extern` nije

neophodno navesti uz takve deklaracije. Ipak, za razliku od funkcija, njegova upotreba se savetuje kako bi se eksplicitno naglasilo da se želi samo deklaracija (a ne i definicija) promenljive i kako odgovor na pitanje da li je nešto deklaracija ili definicija ne bi zavisio od ostatka izvornog teksta programa.

Kod lokalnih promenljivih, svaka deklaracija ujedno je i definicija i ovakvi objekti su po pravilu bez povezanosti (čak i ako je na njih primenjen kvalifikator `static`). Zato je kod lokalnih promenljivih neophodno koristiti kvalifikator `extern` kada se želi naglasiti da je u pitanju deklaracija (a ne definicija) promenljive i da je ta promenljiva definisana na nekom drugom mestu (tj. da promenljiva ima spoljašnju povezanost). Ovom, donekle neobičnom konstrukcijom, postiže se da globalna promenljiva (sa spoljašnjom povezanošću) u nekoj drugoj jedinici prevođenja, u tekućoj jedinici prevođenja ima samo lokalni doseg.

Razmotrimo naredni primer koji sadrži dve jedinice prevođenja.

Program 9.3.

```
#include <stdio.h>
int a;
int b = 3;

void f() {
  printf("a=%d\n", a);
}
```

```
#include <stdio.h>
extern int a;
void f();

int g() {
  extern int b;
  printf("b=%d\n", b);
  f();
}

int main() {
  a = 1;
  /* b = 2; */
  g();
  return 0;
}
```

```
b=3
a=0
```

U prvoj jedinici prevođenja postoje globalne promenljive `a` i `b` kao i globalna funkcija `f`. Sve one podrazumevano imaju statički životni vek i spoljašnju povezanost. Linija `int b = 3;` sadrži inicijalizaciju tako da je jasno da je u pitanju definicija. U prvom trenutku nije jasno da li je `int a;` deklaracija ili definicija i u pitanju je samo uslovna definicija, međutim, pošto ne postoji druga definicija promenljive `a`, ova linija se ipak tumači kao definicija, uz podrazumevanu inicijalnu vrednost 0. Druga jedinica prevođenja sadrži dve deklaracije: deklaraciju promenljive `a` i funkcije `f`. U slučaju promenljive eksplicitno je upotrebljen kvalifikator `extern` čime je naglašeno da je u pitanju promenljiva sa spoljašnjom povezanošću i da se želi pristup promenljivoj `a` iz prve jedinice prevođenja. Čak i da kvalifikator `extern` nije naveden, program bi radio identično. U slučaju funkcije `f` nije bilo neophodno navoditi `extern` jer je jasno da je u pitanju deklaracija. Identifikator `b` je uveden u lokalni doseg funkcije `g` uz kvalifikator `extern` čime je naglašeno da se misli na promenljivu `b` definisanu

u prvoj jedinici prevođenja. U funkciji `main` postavlja se vrednost promenljive `a` na 1. Pokušaj postavljanja vrednosti promenljive `b` (pod komentarom) ne bi uspeo jer identifikator `b` nije u dosegu funkcije `main`.

Unutrašnja povezanost i kvalifikator `static`

Globalni objekat ima unutrašnju povezanost ako se kvalifikuje kvalifikatorom `static`.[10] Unutrašnja povezanost povezuje sva pojavljivanja istog identifikatora, na nivou tačno jedne jedinice prevođenja. Ukoliko se u istoj jedinici prevođenja, u istom dosegu naiđe na više deklaracija identifikatora sa unutrašnjom povezanošću sve one se odnose na isti objekat i za taj objekat mora postojati tačno jedna definicija u toj jedinici prevođenja. S druge strane, kvalifikator `static` onemogućava korišćenje promenljive ili funkcije u drugim datotekama koje čine program. Povezivanje, naravno, neće uspeti ni ukoliko je za neku promenljivu ili funkciju deklarisanu sa `static` u drugoj datoteci navedena deklaracija na koju je primenjen kvalifikator `extern`. Ukoliko neka druga jedinica prevođenja sadrži deklaraciju istog identifikatora neophodno je da postoji definicija odgovarajućeg objekta, različitog od objekta iz prve jedinice prevođenja.

Osnovna uloga unutrašnje povezanosti obično je u tome da neki deo kôda učini zatvorenim, u smislu da je nemoguće menjanje nekih njegovih globalnih promenljivih ili pozivanje nekih njegovih funkcija iz drugih datoteka. Time se taj deo kôda „enkapsulira“ i iz drugih jedinica prevođenja mu se može pristupiti samo kroz preostale tačke komunikacije (funkcije i globalne promenljive sa spoljašnjom povezanošću). Cilj programera, dakle, nije da onemogući druge programere da koriste deo njegovog kôda, već da im omogući jasan interfejs čijim korišćenjem se onemogućuju nehotične greške. Ukoliko je u jednoj jedinici prevođenja deklarisana globalna promenljiva ili funkcija sa unutrašnjom povezanošću, i u drugim jedinicama prevođenja se može deklarisati globalna promenljiva ili funkcija istog imena, ali će se ona odnositi na drugi, nezavisan objekat.

Kao primer, razmotrimo naredne dve jedinice prevođenja

Program 9.4.

```
#include <stdio.h>
static int a = 3;
int b = 5;

static int f() {
  printf("f: a=%d, b=%d\n",
          a, b);
}

int g() {
  f();
}
```

```
#include <stdio.h>

int g(); /* int f(); */
int a, b;

int main() {
   printf("main: a=%d, b=%d\n",
   a, b);
   g(); /* f() */
   return 0;
}
```

[10] Kvalifikator `static` u programskom jeziku C ima potpuno drugu ulogu i dejstvo kada se primenjuje na lokalne promenljive: u tom slučaju, ovaj kvalifikator označava da promenljiva ima statički životni vek (videti poglavlje 9.2.3).

Program ispisuje

```
main: a=0, b=5
f: a=3, b=5
```

U prvoj jedinici prevođenja promenljiva `a` i funkcija `f` imaju unutrašnju povezanost (dok promenljiva `b` i funkcija `g` imaju spoljašnju povezanost (o kojoj će biti reči u nastavku poglavlja). U drugoj jedinici prevođenja promenljive `a, b` i funkcije `g` i `main` imaju spoljašnju povezanost. Ime `a` u dve različite jedinice prevođenja odnosi se na dve različite promenljive. Promenljiva `a`, definisana u prvoj jedinici prevođenja, ključnom rečju `static` sakrivena je i nije joj moguće pristupiti iz ostalih jedinica prevođenja. Slično je i sa funkcijom `f` (uklanjanje komentara u drugoj jedinici prevođenja dovelo bi do greške prilikom povezivanja jer je definicija funkcije `f` skrivena u prvoj jedinici prevođenja). Sa druge strane, sva pojavljivanja imena `b` odnose se na istu promenljivu i sva pojavljivanja imena `g` odnose se na istu funkciju.

Pošto u prvoj jedinici prevođenja postoji definicija promenljive `b`, u drugoj jedinici prevođenja `int b;` je samo njena deklaracija. Pošto je definicija promenljive `a` iz prve jedinice prevođenja skrivena, prilikom povezivanja druge jedinice prevođenja nije dostupna definicija promenljive `a`, onda se `int a;` u drugoj jedinici prevođenja smatra njenom definicijom (uz podrazumevanu vrednost 0).

9.2.5 Greške u fazi prevođenja i povezivanja

U procesu generisanja izvršnog programa od izvornog programa, greške (engl. error) mogu biti otkrivene u nekoliko faza. Poruke o otkrivenim greškama se obično usmeravaju na *standardni izlaz za greške*, `stderr` (videti poglavlje 12.1). Čak i u veoma kratkim programima, za svaku od ovih faza može da postoji greška koja tada može biti otkrivena. Razmotrimo jednostavan (veštački, bez konkretne svrhe) program naveden u nastavku i greške do kojih može doći malim izmenama:

Program 9.5.

```
#include<stdio.h>
int main() {
  int a = 9;
  if (a == 9)
     printf("9");
  return 0;
}
```

Pretprocesiranje. Na primer, ukoliko je, umesto pretprocesorske direktive `#include`, u programu navedeno `#Include`, biće prijavljena greška (jer ne postoji direktiva `#Include`):

```
primer.c:1:2: error: invalid preprocessing directive #Include
```

Kompilacija

- Leksička analiza: Na primer, ukoliko u programu piše `int a = 09;` umesto `int a = 9;`, biće detektovana neispravna oktalna konstanta `09`, tj. neispravna leksema `09` i biće prijavljena greška:

  ```
  primer.c:4:9: error: invalid digit "9" in octal constant
  ```

- Sintaksička analiza: Na primer, ukoliko, umesto `if (a == 9) ...`, u programu postoji naredba `if a == 9 ...`, zbog izostavljenih zagrada biće prijavljena greška:

  ```
  primer.c:5:6: error: expected ‘(’ before ‘a’
  ```

- Semantička analiza: Na primer, ukoliko u programu, umesto `if (a == 9) ...`, postoji naredba `if ("a" * 9) ...`, tokom provere tipova biće prijavljena greška (jer operator `*` nije moguće primenjivati nad niskama):

  ```
  primer.c:5:8: error: invalid operands to binary *
                       (have ‘char *’ and ‘int’)
  ```

Povezivanje. Na primer, ukoliko u programu, umesto `printf(...);`, postoji naredba `print(...);`, biće prijavljena greška (jer nije raspoloživa definicija funkcije `print`):

```
primer.c:(.text+0x20): undefined reference to ‘print’
collect2: ld returned 1 exit status
```

Sve greške u izvornom programu koje mogu biti otkrivene u celokupnom procesu prevođenja bivaju otkrivene u fazi leksičke, sintaksičke ili semantičke analize, ili u fazi linkovanja. Dakle, tokom generisanja i optimizacije među kôda, kao i tokom generisanja kôda ne može biti prijavljenih grešaka (barem kada je prevodilac ispravan). Dobro je poznavati faze u prevođenju i vrste grešaka koje u tim fazama mogu biti otkrivene, jer se sa tim znanjem prijavljene greške lakše otklanjaju.

Ukoliko prevodilac naiđe na grešku ili greške u izvornom kôdu, to prijavljuje (navodi vrstu greške, liniju i kolonu programa u kojoj je greška) i ne generiše izvršni program. Izvršni program se, naravno, ne generiše ni u slučaju ako je greška otkrivena u fazi povezivanja. Programer tada treba da otkloni detektovane greške i ponovi proces prevođenja.

Pored prijave grešaka, kompilator može da izda i *upozorenja* (engl. warning) koja ukazuju na potencijalne propuste u programu, ali se i pored njih izvršni program može generisati. Na primer, ukoliko navedeni program, umesto naredbe `if (a == 9) ...` sadrži naredbu `if (a = 9) ...`, prevodilac može izdati upozorenje:

```
primer.c:4: warning: suggest parentheses around assignment used as truth value
```

Ukoliko navedeni program, umesto naredbe `if (a == 9) ...` sadrži naredbu `if (9 == 9) ...`, prevodilac može izdati upozorenje:

```
primer.c:3: warning: unused variable ‘a’
```

Ukoliko navedeni program, umesto naredbe `if (a == 9) ...` sadrži naredbu `if (a / 0) ...`, prevodilac može izdati upozorenje:

```
primer.c:4: warning: division by zero
```

Ukoliko se u fazi izvršavanja naiđe na celobrojno deljenje nulom (kao što je to slučaj sa navedenim primerom), na većini sistema će doći do greške u fazi izvršavanja (videti poglavlje 9.3.5). Ipak, standard proglašava ponašanje programa u slučaju deljenja nulom nedefinisanim, pa navedeni kôd dovodi u fazi prevođenja samo do upozorenja, a ne do greške.

Upozorenja, čak i kada je uspešno generisan izvršni kôd, ne treba ignorisati i dobra praksa može da bude da se izvorni kôd modifikuje sve dok ima ijednog upozorenja.

Svaki prevodilac predviđa više vrsta upozorenja i te vrste zavise od konkretnog prevodioca. Na primer, u prevodiocu GCC, sve vrste upozorenja se omogućavaju opcijom `-Wall`.

9.2.6 Primer organizacije programa u više datoteka

Kao primer programa sastavljenog iz više datoteka i više jedinica prevođenja, razmotrimo definisanje jednostavne korisničke biblioteke za rad sa pozitivnim razlomcima. Biblioteka je projektovana tako da prikaže što više koncepata opisanih u ovoj glavi (u realnoj situaciji neka rešenja možda bi bila drugačija). Biblioteka pruža definiciju tipa za reprezentovanje razlomaka (`RAZLOMAK`), funkcije za kreiranje razlomka na osnovu brojioca i imenioca, za skraćivanje razlomka i za sabiranje razlomaka. Primena ovih funkcija može dovesti do dva (u ovom primeru) ishoda: da je operacija izvršena uspešno i da se prilikom primene operacije javlja neispravan razlomak (onaj kojem je imenilac 0). Ovim ishodima odgovaraju članovi `OK` i `IMENILAC_0` nabrojivog tipa `RAZLOMAK_GRESKA`. Kako im nisu eksplicitno pridružene vrednosti – ovim imenima će implicitno biti pridružene vrednosti `0` i `1`. Tekstualni opis mogućih ishoda čuva se u nizu `razlomak_poruka`. Ishod rada funkcije biće čuvan u globalnoj promenljivoj `razlomak_greska`.

Datoteka `razlomak.h` je javni interfejs biblioteke i sadrži deklaracije funkcija i promenljivih koje su na raspolaganju njenim korisnicima.

```
#ifndef __RAZLOMAK_H_
#define __RAZLOMAK_H_

typedef struct razlomak {
  unsigned brojilac;
  unsigned imenilac;
} RAZLOMAK;

typedef enum {OK, IMENILAC_0} RAZLOMAK_GRESKA;
extern RAZLOMAK_GRESKA razlomak_greska;
extern char* razlomak_poruka[];

RAZLOMAK napravi_razlomak(unsigned brojilac, unsigned imenilac);
RAZLOMAK skrati_razlomak(RAZLOMAK r);
RAZLOMAK saberi_razlomke(RAZLOMAK r1, RAZLOMAK r2);
```

```
#endif
```

Kao što je već rečeno, u deklaracijama funkcija nije bilo neophodno navoditi kvalifikator `extern` (jer funkcije podrazumevano imaju spoljašnju povezanost), dok je on upotrebljen kod promenljive `razlomak_greska` i niza `razlomak_poruka` kako bi se nedvosmisleno naglasilo da su u pitanju deklaracije, a ne definicije.

Datoteka zaglavlja `razlomak.h` počinje i završava se pretprocesorskim direktivama. Generalno, u datoteci zaglavlja, pored prototipova funkcija, mogu da se nalaze i neke definicije (najčešće korisnički definisanih tipova, a ponekad čak i funkcija i promenljivih). U slučaju da se datoteka zaglavlja uključi više puta u neku jedinicu prevođenja (što se često događa preko posrednog uključivanja), desilo bi se da neka jedinica prevođenja sadrži višestruko ponavljanje nekih definicija što bi dovelo do greške prilikom kompilacije. Ovaj problem se može rešiti korišćenjem pretprocesorskih direktiva i uslovnim prevođenjem. Kako bi se sprečilo višestruko uključivanje, svaka datoteka zaglavlja u tom slučaju treba da ima sledeći opšti oblik:

```
#ifndef IME
#define IME

...

#endif
```

gde je tekst `IME` karakterističan za tu datoteku (na primer — njeno ime obogaćeno nekim specijalnim simbolima). Prilikom prvog uključivajna ove datoteke, definiše se simboličko ime `IME` i zbog toga naredna uključivanja (iz iste datoteke) ignorišu celokupan njen sadržaj. Ovaj mehanizam olakšava održavanje datoteka zaglavlja i primenjen je i u navedenom primeru.

Implementacija biblioteke sadržana je u datoteci `razlomak.c`.

```
#include "razlomak.h"

char* razlomak_poruka[] = {
  "Operacija uspesno sprovedena",
  "Greska: imenilac je nula!"
};

static const RAZLOMAK NULA = {0, 0};
RAZLOMAK_GRESKA razlomak_greska;

static int nzd(unsigned a, unsigned b) {
    return (b == 0) ? a : nzd(b, a % b);
}

RAZLOMAK skrati_razlomak(RAZLOMAK r) {
  if (r.imenilac == 0) {
    razlomak_greska = IMENILAC_0; return NULA;
  }
```

```
  unsigned n = nzd(r.brojilac, r.imenilac);
  r.brojilac /= n;
  r.imenilac /= n;
  razlomak_greska = OK;
  return r;
}

RAZLOMAK napravi_razlomak(unsigned brojilac, unsigned imenilac) {
  RAZLOMAK rez;
  rez.brojilac = brojilac;
  rez.imenilac = imenilac;
  if (imenilac == 0) {
    razlomak_greska = IMENILAC_0; return NULA;
  }
 return skrati_razlomak(rez);
}

RAZLOMAK saberi_razlomke(RAZLOMAK r1, RAZLOMAK r2) {
  if (r1.imenilac == 0 || r2.imenilac == 0) {
    razlomak_greska = IMENILAC_0; return NULA;
  }
  return napravi_razlomak(
     r1.brojilac*r2.imenilac + r1.imenilac*r2.brojilac,
     r1.imenilac*r2.imenilac);
}
```

Prvi korak predstavlja uključivanje datoteke zaglavlja `razlomak.h`. Iako ovaj korak nije uvek neophodan (ukoliko se pre svakog poziva funkcije javlja njena definicija), svakako se preporučuje pre definicija uključiti deklaracije kako bi se tokom kompilacije proverilo da li su deklaracije i definicije uparene.

Datoteka sadrži definiciju globalnog niza `razlomak_poruka` i definiciju globalne promenljive `razlomak_poruka`. Elementi niza `razlomak_poruka` inicijalizovani su kao pokazivači na konstantne niske. Imenima `OK` i `IMENILAC_0` odgovaraju vrednosti `0` i `1`, pa pokazivač `razlomak_poruka[OK]` ukazuje na konstantnu nisku `"Operacija uspesno sprovedena"` a `razlomak_poruka[IMENILAC_0]` na konstantnu nisku `"Greska: imenilac je nula!"`. Datoteka sadrži konstantu `NULA` koja se koristi kao specijalna povratna vrednost u slučaju greške, kao i pomoćnu funkciju `nzd` za pronalaženje najvećeg zajedničkog delioca dva broja. S obzirom na to da se ne predviđa da ih korisnik biblioteke direktno koristi, one su sakrivene postavljanjem unutrašnjeg povezivanja korišćenjem kvalifikatora `static`.

Program koji demonstrira upotrebu biblioteke dat je u zasebnoj datoteci `program.c`.

Program 9.6.

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

int main() {
  unsigned a, b;
```

```
  RAZLOMAK r1, r2, r;

  scanf("%u%u", &a, &b);
  r1 = napravi_razlomak(a, b);
  if (razlomak_greska != OK) {
    printf("%s\n", razlomak_poruka[razlomak_greska]);
    return 1;
  }

  scanf("%u%u", &a, &b);
  r2 = napravi_razlomak(a, b);
  if (razlomak_greska != OK) {
    printf("%s\n", razlomak_poruka[razlomak_greska]);
    return 1;
  }

  r = saberi_razlomke(r1, r2);
  if (razlomak_greska != OK)
    printf("%s\n", razlomak_poruka[razlomak_greska]);

  printf("Rezultat je: %u/%u\n", r.brojilac, r.imenilac);
  return 0;
}
```

Program učitava dva razlomka, sabira ih i ispisuje rezultat, proveravajući u pojedinim koracima vrednost promenljive `razlomak_greska`. U slučaju da je došlo do greške, opis greške se ispisuje na standardni izlaz. Datoteka `razlomak.h` nesmetano je uključena u obe jedinice prevođenja.

Odgovarajuća `Makefile` datoteka može da bude:

```
program : program.o razlomak.o
     gcc -o program program.o razlomak.o

program.o : program.c razlomak.h
     gcc -c -O3 -Wall program.c

razlomak.o : razlomak.c razlomak.h
     gcc -c -O3 -Wall razlomak.c
```

Pitanja i zadaci za vežbu

Pitanje 9.2.10. *Koji program pravi od ulaznih datoteka jedinice prevođenja?*

Pitanje 9.2.11. *Prilikom generisanja izvršne verzije programa napisanog na jeziku C, koja faza prethodi kompilaciji, a koja sledi nakon faze kompilacije?*

Pitanje 9.2.12. *Kako se zove faza prevođenja programa u kojoj se od objektnih modula kreira izvršni program? Koji program sprovodi tu fazu?*

Pitanje 9.2.13. *Kako se korišćenjem GCC kompilatora može zasebno prevesti datoteka* `prva.c`*, zatim zasebno prevesti datoteka* `druga.c` *i na kraju povezati sve u program pod imenom* `program`*?*

Pitanje 9.2.14. *Kako se obično obezbeđuje da nema višestrukog uključivanja neke datoteke?*

Pitanje 9.2.15. *Koji sve nivoi dosega identifikatora postoje u jeziku C? Koja su dva najčešće korišćena? Kog nivoa je doseg lokalnih, a kog nivoa doseg globalnih promenljivih?*

Pitanje 9.2.16. *Nabrojati tri vrste identifikatora prema vrsti povezivanja.*

Pitanje 9.2.17. *Koju povezanost imaju lokalne promenljive?*

Pitanje 9.2.18.

1. *Koji kvalifikator se koristi da bi se onemogućilo da se globalna promenljiva koristi u nekoj drugoj jedinici prevođenja? Koje povezivanje se u tom slučju primenjuje?*
2. *Koji kvalifikator se koristi da bi se globalna promenljiva deklarisana u drugoj jedinici prevođenja mogla da koristi u drugoj? Koje povezivanje se u tom slučju primenjuje?*

Pitanje 9.2.19. *Ako je za neku spoljašnju promenljivu navedeno da je* `static` *a za istu tu promenljivu u nekoj drugoj datoteci navedeno da je* `extern`*, u kojoj fazi će biti prijavljena greška?*

Pitanje 9.2.20. *Da li je naredna deklaracija ispravna i koliko bajtova je njome alocirano:*

1. `int a[] = {1, 2, 3};`
2. `extern int a[];`
3. `extern int a[] = {1, 2, 3};`

Pitanje 9.2.21. *Koje vrste životnog veka postoje u jeziku C? Koji životni vek imaju lokalne promenljive (ako na njih nije primenjen kvalifikator* `static`*)? Koji životni vek imaju globalne promenljive?*

Pitanje 9.2.22. *Ukoliko je za neku lokalnu promenljivu naveden kvalifikator* `static`*, šta će biti sa njenom vrednošću nakon kraja izvršavanja funkcije u kojoj je deklarisana?*

Pitanje 9.2.23. *Navesti primer funkcije* `f` *u okviru koje je deklarisana lokalna statička promenljiva* `a` *i lokalna automatska promenljiva* `b`*. Koja je podrazumevana vrednost promenljive* `a`*, a koja promenljive* `b`*?*

Pitanje 9.2.24. *Šta ispisuju naredni programi:*

```
int i = 2;
int f() { static int i; return ++i; }
int g() { return ++i; }
int main() {
  int i;
  for(i=0; i<3; i++)
    printf("%d", f()+g());
}
```

```
int i=2;
void f() { static int i; i++; printf("%d",i); }
void g() { i++; printf("%d",i); }
void h() { int i = 0; i++; printf("%d",i); }
int main() {
  f(); g(); h(); f(); g(); h();
}
```

Pitanje 9.2.25. *Ukoliko je na globalnu promenljivu primenjen kvalifikator* `static`*, kakva joj je povezanost, šta je njen doseg i koliki joj je životni vek?*

9.3 Organizacija izvršnog programa

Ukoliko je od izvornog programa uspešno generisana izvršna verzija, ta, generisana verzija može biti izvršena. Zahtev za izvršavanje se izdaje operativnom sistemu (npr. navođenjem imena programa u komandnoj liniji). Program koji treba da bude izvršen najpre se učitava sa spoljne memorije u radnu memoriju računara. Operativni sistem mu stavlja određenu memoriju na raspolaganje, vrši se dinamičko povezivanje poziva rutina niskog nivoa rantajm bibliotekom, vrše se potrebne inicijalizacije i onda izvršavanje može da počne. U fazi izvršavanja moguće je da dođe do grešaka koje nije bilo moguće detektovati u fazi prevođenja i povezivanja.

Izvršni program generisan za jedan operativni sistem može se izvršavati samo na tom sistemu. Moguće je da izvršni program koji je generisan za jedan operativni sistem ne može da se izvršava na računaru koji koristi taj operativni sistem, ali ima arhitekturu koja ne odgovara generisanoj aplikaciji (na primer, 64-bitna aplikacija ne može da se izvršava na 32-bitnom sistemu).

9.3.1 Izvršno okruženje — operativni sistem i rantajm biblioteka

Kao što je ranije rečeno, izvršni program koji je dobijen prevođenjem i povezivanjem ne može da se izvršava autonomno. Naime, taj program sadrži pozive rutina niskog nivoa koje nisu ugrađene u program (jer bi inače izvršni program bio mnogo veći). Umesto toga, te rutine su raspoložive u vidu *rantajm biblioteke* (engl. runtime library). Funkcije rantajm biblioteke obično su realizovane kao niz zahteva operativnom sistemu (najčešće zahteva koji se odnose na ulaz/izlaz i na memoriju). Kada se program učita u memoriju, vrši se takozvano *dinamičko povezivanje* (engl. dynamic linking) i pozivi funkcija i promenljivih iz izvršnog programa se povezuju sa funkcijama i promenljivama iz rantajm biblioteke koja je učitana u memoriju u vreme izvršavanja.

Svojstva rantajm biblioteke zavise od kompilatora i operativnog sistema i u najvećem delu nisu specifikovana standardom. Štaviše, ni granica koja definiše šta je implementirano u vidu funkcija standardne biblioteke a šta u vidu funkcija rantajm biblioteke nije specifikovana niti kruta i zavisi od konkretnog kompilatora. Funkcije rantajm biblioteke su raspoložive programima generisanim od strane kompilatora ali obično nisu neposredno raspoložive aplikativnom programeru. Ako se izvršava više programa kompiliranih istim kompilatorom, svi oni koriste funkcije iste rantajm biblioteke.

Opisani pristup primenjuje se, ne samo na C, već i na druge programske jezike. Biblioteka za C ima i svoje specifičnosti. U operativnom sistemu Linux,

funkcije C rantajm biblioteke smatraju se i delom operativnog sistema i moraju da budu podržane. Nasuprot tome, operativni sistem Windows ne sadrži nužno podršku za C bibliteku i ona se isporučuje uz kompilatore. Zbog toga se i delovi rantajm biblioteke mogu statički ugraditi u izvršni program (da se ne bi zavisilo od njenog postaojanja na ciljnom sistemu).

Osnovni skup rantajm funkcija koju jezik C koristi je sadržan u modulu `crt0` (od nulta faza za „C Run-Time“). Zadatak ovog modula je da pripremi izvršavanje programa i obavlja sledeće zadatke: priprema standardnih ulazno/izlaznih tokova, inicijalizacija segmenata memorije, priprema argumenata funkcije `main`, poziv funkcije `main` i slično.

9.3.2 Vrste i organizacija podataka

Pojam tipova u višem programskom jeziku kao što je C namenjen je čoveku, a ne računaru. Prevođnjem se sve konstrukcije jezika višeg nivoa prevode na jednostavne konstrukcije neposredno podržane od strane procesora. Isto se dešava i sa tipovima: svi tipovi koji se pojavljuju u programu biće svedeni na osnovne tipove neposredno podržane od strane procesora — obično samo na cele brojeve i brojeve u pokretnom zarezu. Dakle, u izvornom programu, promenljivama je pridružen tip, ali u izvršnom programu nema eksplicitnih informacija o tipovima. Sve informacije o tipovima se razrešavaju u fazi prevođenja i u mašinskom kôdu se koriste instrukcije za konkretne vrste operanada. Na primer, u izvornom programu, operator + se može primenjivati nad raznim tipovima podataka, dok u mašinskom jeziku postoje zasebne instrukcije za sabiranje celih brojeva i za sabiranje brojeva u pokretnom zarezu.

Veličina osnovnih vrsta podataka u bajtovima u izvršnom programu zavisi od opcija prevođenja i treba da bude prilagođena sistemu na kojem će se izvršavati.[11] Faktori koje treba uzeti u obzir su hardver, ali i operativni sistem računara na kojem će se program izvršavati. Centralni hardver je danas obično ili 32-bitni ili 64-bitni tj. obično registri procesora, magistrale i podaci tipa `int` imaju veličinu 32 bita ili 64 bita. I softver (i aplikativni i sistemski) danas je najčešće ili 32-bitni ili 64-bitni. Na 64-bitnom hardveru može da se izvršava i 32-bitni softver, ali ne i obratno. Pod 32-bitnim softverom se najčešće podrazumeva softver koji koristi 32-bitni adresni prostor tj. može da adresira 2^{32} bajtova memorijskog prostora (analogno važi i za 64-bitni). Dakle, i operativni sistemi i aplikacije su danas najčešće 32-bitni ili 64-bitni. S obzirom na to da se aplikativni programi izvršavaju pod okriljem operativnog sistema, oni moraju biti prilagođeni operativnom sistemu na kojem će se izvršavati. Na 64-bitnom operativnom sistemu mogu da se izvršavaju i 32-bitni i 64-bitni programi, a na 32-bitnom operativnom sistemu mogu da se izvršavaju 32-bitni ali ne i 64-bitni programi.

Na većini 32-bitnih sistema konkretni tipovi podataka imaju brojeve bitova kao što je navedeno u tabeli 6.1 u glavi 6). Prilikom kompilacije za 64-bitne sisteme, postoji izbor koji tipove podataka će zauzeti 32, a koji 64 bita. Postoji nekoliko konvencija koje su imenovane tako da se iz imena vidi koji tipovi podataka zauzimaju 64 bita: ILP64 (integer, long, pointer), LP64 (long, pointer),

[11] Sistem na kome se vrši prevođenje ne mora biti isti kao onaj na kome će se vršiti izvršavanje i neki prevodioci dopuštaju da se u zavisnosti od opcija prevođenja kreiraju aplikacije koje su namenjene izvršavanju na sasvim drugačijoj arhitekturi računara pod drugim operativnim sistemom.

LLP64 (long long, pointer). Savremeni Linux sistemi i GCC obično koriste LP64 sistem, dok Windows koristi LLP64 sistem.

tip	32	ILP64	LP64	LLP64
`char`	8	8	8	8
`short`	16	16	16	16
`int`	32	64	32	32
`long`	32	64	64	32
`long long`	64	64	64	64
pointer	32	64	64	64

Za precizno specifikovanje broja bitova mogu se koristiti tipovi iz zaglavlja `<stdint.h>`.

Kada se koristi kompilator GCC, podrazumevani broj bitova odgovara sistemu na kojem se vrši prevođenje. Ovo se može promeniti opcijama `-m32` i `-m64` kojima se može zadati da se generisani kôd aplikacije bude 32-bitni ili 64-bitni (bez obzira da li se prevođenje vrši na 32-bitnom ili 64-bitnom sistemu).

9.3.3 Organizacija memorije

Način organizovanja i korišćenja memorije u fazi izvršavanja programa može se razlikovati od jednog do drugog operativnog sistema. Tekst u nastavku odnosi se, ako to nije drugačije naglašeno, na širok spektar platformi, pa su, zbog toga, načinjena i neka pojednostavljivanja.

Kada se izvršni program učita u radnu memoriju računara, biva mu dodeljena određena memorija i započinje njegovo izvršavanje. Dodeljena memorija organizovana je u nekoliko delova koje zovemo *segmenti* ili *zone*:

- *segment kôda (engl. code segment ili text segment)*;
- *segment podataka (engl. data segment)*;
- *stek segment (engl. stack segment)*;
- *hip segment (engl. heap segment)*.

U nastavku će biti opisana prva tri, dok će o hip segmentu biti više reči u poglavlju 10.9, posvećenom dinamičkoj alokaciji memorije.

Segmentacija je u određenoj vezi sa životnim vekom promenljivih (o kome je bilo reči u poglavlju 9.2.3): promenljive statičkog životnog veka se obično čuvaju u segmentu podataka, promenljive automatskog životnog veka se obično čuvaju u stek segmentu, dok se promenljive dinamičkog životnog veka obično čuvaju u hip segmentu.

Segment kôda

Fon Nojmanova arhitektura računara predviđa da se u memoriji čuvaju podaci i programi. Dok su ostala tri segmenta predviđena za čuvanje podataka, u segmentu kôda se nalazi sâm izvršni kôd programa — njegov mašinski kôd koji uključuje mašinski kôd svih funkcija programa (uključujući kôd svih korišćenih funkcija koje su povezane statički). Na nekim operativnim sistemima, ukoliko je pokrenuto više instanci istog programa, onda sve te instance dele isti prostor za izvršni kôd, tj. u memoriji postoji samo jedan primerak kôda. U

tom slučaju, za svaku instancu se, naravno, zasebno čuva informacija o tome do koje naredbe je stiglo izračunavanje.

Segment podataka

U segmentu podataka čuvaju se određene vrste promenljivih koje su zajedničke za ceo program (one koje imaju statički životni vek, najčešće globalne promenljive), kao i konstantni podaci (najčešće konstantne niske). Ukoliko se istovremeno izvršava više instanci istog programa, svaka instanca ima svoj zaseban segment podataka. Na primer, u programu

Program 9.7.

```
#include <stdio.h>
int a;
int main() {
  int b;
  static double c;
  printf("Zdravo\n");
  return 0;
}
```

u segmentu podataka će se nalaziti promenljive `a` i `c`, kao i konstantna niska `"Zdravo\n"` (sa završnom nulom, a bez navodnika). Promenljiva `b` je lokalna automatska i ona će se čuvati u segmentu steka. Ukoliko se ista konstantna niska javlja na više mesta u programu, standard ne definiše da li će za nju postojati jedna ili više kopija u segmentu podataka.

Stek segment

U stek segmentu (koji se naziva i *programski stek poziva (engl. call stack)*) čuvaju se svi podaci koji karakterišu izvršavanje funkcija. Podaci koji odgovaraju jednoj funkciji (ili, preciznije, jednoj instance jedne funkcije — jer, na primer, rekurzivna funkcija može da poziva samu sebe i da tako u jednom trenutku bude aktivno više njenih instanci) organizovani su u takozvani *stek okvir (engl. stack frame)*. Stek okvir jedne instance funkcije obično, između ostalog, sadrži:

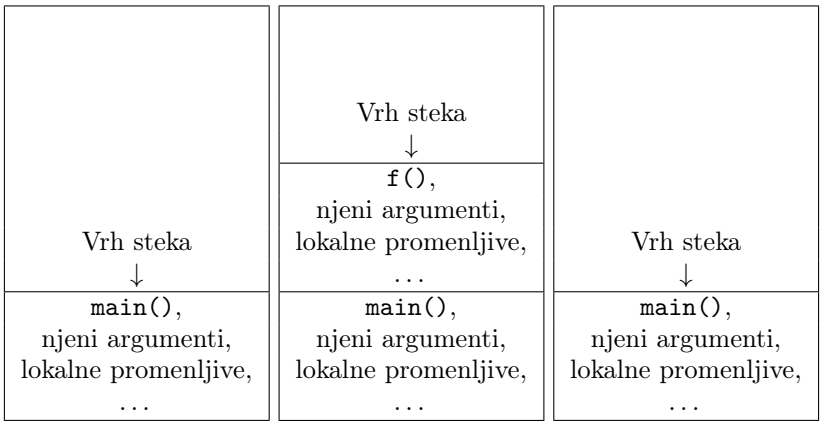
- argumente funkcije;
- lokalne promenljive (promenljive deklarisane unutar funkcije);
- međurezultate izračunavanja;
- adresu povratka (koja ukazuje na to odakle treba nastaviti izvršavanje programa nakon povratka iz funkcije);
- adresu stek okvira funkcije pozivaoca.

Stek poziva je struktura tipa *LIFO* ("last in - first out")[12]. To znači da se stek okvir može dodati samo na vrh steka i da se sa steka može ukloniti

[12]Ime *stack* je zajedničko ime za strukture podataka koje su okarakterisane ovim načinom pristupa.

samo okvir koji je na vrhu. Stek okvir za instancu funkcije se kreira onda kada funkcija treba da se izvrši i taj stek okvir se oslobađa (preciznije, smatra se nepostojećim) onda kada se završi izvršavanje funkcije.

Ako izvršavanje programa počinje izvršavanjem funkcije `main`, prvi stek okvir se kreira za ovu funkciju. Ako funkcija `main` poziva neku funkciju `f`, na vrhu steka, iznad stek okvira funkcije `main`, kreira se novi stek okvir za ovu funkciju (ilustrovano na slici 9.1). Ukoliko funkcija `f` poziva neku treću funkciju, onda će za nju biti kreiran stek okvir na novom vrhu steka. Kada se završi izvršavanje funkcije `f`, onda se vrh steka vraća na prethodno stanje i prostor koji je zauzimao stek okvir za `f` se smatra slobodnim (iako on neće biti zaista obrisan).



Slika 9.1: Organizacija steka i ilustracija izvršavanja funkcije. Levo: tokom izvršavanja funkcije `main()`. Sredina: tokom izvršavanja funkcije `f()` neposredno pozvane iz funkcije `main()`. Desno: nakon povratka iz funkcije `f()` nazad u funkciju `main()`.

Veličina stek segmenta obično je ograničena. Zbog toga je poželjno izbegavati smeštanje jako velikih podataka na segment steka. Na primer, sasvim je moguće da u programu u nastavku, sa leve strane, niz `a` neće biti uspešno alociran i doći će do greške prilikom izvršavanja programa, dok će u programu sa desne strane niz biti smešten u segment podataka i sve će teći očekivano. Predefinisana veličina steka C prevodioca se može promeniti zadavanjem odgovarajuće opcije.

```
int main() {
    int a[1000000];
    ...
}
```

```
int a[1000000];
int main() {
    ...
}
```

Opisana organizacija steka omogućava jednostavan mehanizam međusobnog pozivanja funkcija, kao i rekurzivnih poziva.

Prenos parametara u funkciju preko steka. Ilustrujmo prenos parametara u funkciju na sledećem jednostavnom primeru.

Program 9.8.

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

int main() {
  int a[] = {1, 2, 3};
  int s;
  s = f(a, sizeof(a)/sizeof(int));
  printf("s = %d, a = {%d, %d, %d}\n",
          s, a[0], a[1], a[2]);
  return 0;
}
```

U funkciji `main` deklarisane su dve lokalne promenljive: niz `a` i ceo broj `s`. Na početku izvršavanja funkcije `main` na programskom steku nalazi se samo jedan stek okvir sledećeg oblika[13]:

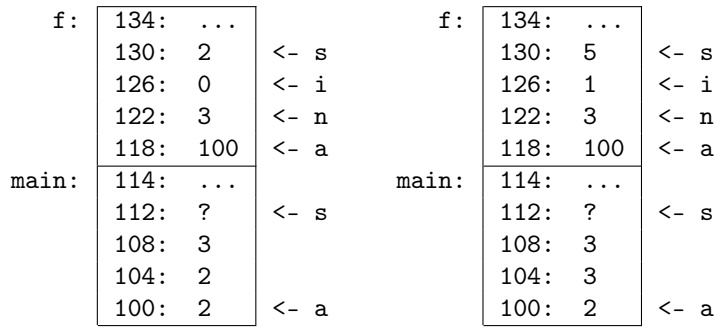
```
main:  114:  ...
       112:  ?     <- s
       108:  3
       104:  2
       100:  1     <- a
```

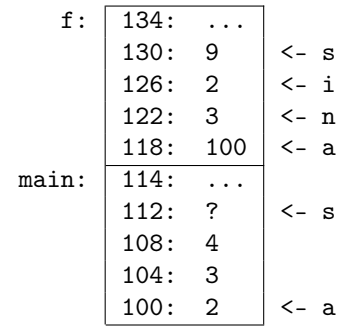
Nakon poziva funkcije `f`, iznad stek okvira funkcije `main` postavlja se stek okvir funkcije `f`. Ona ima četiri lokalne promenljive (od čega dve potiču od argumenta). Argumenti se inicijalizuju na vrednosti koje su prosleđene prilikom poziva. Primetimo da je umesto niza preneta samo adresa njegovog početka, dok je vrednost parametra `n` postavljena na `3` jer je to vrednost izraza `sizeof(a)/sizeof(int)` u funkciji `main` (iako se sintaksički ovaj izraz nalazi u okviru poziva funkcije `f`, on nema nikakve veze sa funkcijom i izračunava se u funkciji `main`).

```
   f:  134:  ...
       130:  0     <- s
       126:  ?     <- i
       122:  3     <- n
       118:  100   <- a
main:  114:  ...
       112:  ?     <- s
       108:  3
       104:  2
       100:  1     <- a
```

Nakon izvršavanja koraka petlje, stanje steka je sledeće.

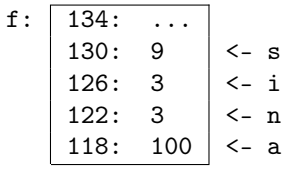
[13] U tekstu se pretpostavlja da se sve lokalne promenljive čuvaju na steku, mada u realnoj situaciji kompilator može da odluči da se neke promenljive čuvaju u registrima procesora.



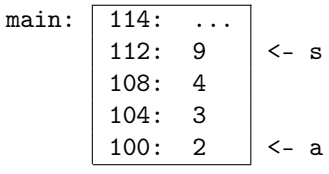


Primetimo da originalna vrednost niza `a` u funkciji `main` menja tokom izvršavanja funkcije `f`, zato što se niz ne kopira već se u funkciju prenosi adresa njegovog početka.

Kada funkcija `f` završi svoje izvršavanje, njen stek okvir je:



i tada je potrebno vratiti vrednost `s` pozivaocu (u ovom slučaju funkciji `main`). Ovo se najčešće postiže tako što se vrednost promenljive `s` upiše u registar `ax`, odakle je `main` preuzima i naredbom dodele upisuje u svoju promenljivu `s`. Kada funkcija `f` završi svoje izvršavanje, stanje steka je:



Nakon toga dolazi do poziva funkcije `printf`. Na stek okvir funkcije `main` postavlja se njen stek okvir i prosleđuju se argumenti. Prvi argument je adresa konstantne niske (koja se nalazi negde u segmentu podataka), a ostali su kopije četiri navedene vrednosti iz funkcije `main`. Nakon ispisa, uklanja se stek okvir funkcije `printf`, a zatim i funkcije `main`, pošto se došlo do njenog kraja.

Implementacija rekurzije. Navedeno je da je rekurzija situacija u kojoj jedna funkcija poziva sebe samu direktno ili indirektno. Razmotrimo, kao primer, funkciju koja rekurzivno izračunava faktorijel:[14]

Program 9.9.

```
#include <stdio.h>

int faktorijel(int n) {
  if (n <= 0)
    return 1;
  else
    return n*faktorijel(n-1);
}

int main() {
   int n;
   while(scanf("%d",&n) == 1)
      printf("%d! = %d\n", n, faktorijel(n));
   return 0;
}
```

Ukoliko je funkcija `faktorijel` pozvana za argument `5`, onda će na steku poziva da se formira isto toliko stek okvira, za pet nezavisnih instanci funkcije. U svakom stek okviru je drugačija vrednost argumenta `n`. No, iako u jednom trenutku ima 5 aktivnih instanci funkcije `faktorijel`, postoji i koristi se samo jedan primerak izvršnog kôda ove funkcije (u kôd segmentu), a svaki stek okvir pamti za svoju instancu dokle je stiglo izvršavanje funkcije, tj. koja je naredba u kôd segmentu tekuća.

9.3.4 Primer organizacije objektnih modula i izvršnog programa

U ovom poglavlju ćemo pokušati da proces kompilacije i povezivanja ogolimo do krajnjih granica tako što ćemo pokazati kako je moguće videti mašinski kôd tj. sadržaj objektnih modula i izvršnog programa nastalih kompilacijom sledećeg jednostavnog programa[15]. Detalji ovog primera svakako u velikoj meri prevazilaze osnove programiranja u programskom jeziku C, ali zainteresovanom čitaocu mogu da posluže da prilično demistifikuje proces prevođenja i izvršavanja C programa.

Program 9.10.

```
#include <stdio.h>
int n = 10, sum;

int main() {
   int i;
   for (i = 0; i < n; i++)
```

[14] Vrednost faktorijela se, naravno, može izračunati i iterativno, bez korišćenja rekurzije.

[15] Primer podrazumeva korišćenje GCC prevodioca na 32-bitnom Linux sistemu. Već na 64-bitnim Linux sistemima stvari izgledaju donekle drugačije.

```
        sum += i;
   printf("sum = %d\n", sum);
   return 0;
}
```

Čitav izvorni program sadržan je u jednoj jedinici prevođenja, tako da je za dobijanje izvršnog kôda potrebno prevesti tu jedinicu i povezati nastali objektni modul sa potrebnim delovima (objektnim modulima) unapred kompilirane standardne biblioteke.

Nakon kompilacije sa `gcc -c program.c`, sadržaj dobijenog objektnog modula `program.o` može se ispitati korišćenjem pomoćnih programa poput `nm` ili `objdump`.

Na primer, `nm --format sysv program.o` daje rezultat

```
Name      Value   Class   Type      Size    Section
main    |00000000|   T   | FUNC   |00000050| .text
n       |00000000|   D   | OBJECT |00000004| .data
printf  |        |   U   | NOTYPE |        | *UND*
sum     |00000004|   C   | OBJECT |00000004| *COM*
```

Tabela pokazuje da u objektnom modulu postoje četiri imenovana simbola. Simbol `main` je funkcija, nalazi se u segmentu kôda (na šta ukazuju sekcija `.text` i klasa `T`) i to na njegovom samom početku (na šta ukazuje vrednost `00000000`) i mašinske instrukcije dobijene njegovim prevođenjem zauzimaju $(50)_{16}$ bajtova. Simbol `n` nalazi se u segmentu podataka (na šta ukazuje sekcija `.data`), na njegovom samom početku (na šta ukazuje vrednost `00000000`) i inicijalizovan je (na šta ukazuje klasa `D`). Simbol `printf` je simbol čija je definicija nepoznata (na šta ukazuje oznaka sekcije `*UND*`, i klase `U`). Simbol `sum` se nalazi u zoni neinicijalizovanih podataka (na šta ukazuje klasa `C` i sekcija `*COM*`) — naime, u ranijim razmatranjima smatrali smo da se neinicijalizovani podaci nalaze u istom segmentu podataka kao i inicijalizovani, što je tačno, ali dublji uvid pokazuje da se oni cuvaju u posebnom delu segmenta podataka (često se ovaj segment naziva `.bss` segment). S obzirom na to da ovaj segment podrazumevano treba da bude ispunjen nulama, za razliku od inicijalizovanih podataka koji moraju biti upisani u izvršnu datoteku, kod neinicijalizovanih podataka, umesto da se nule upisuju u izvršnu datoteku, u njoj se samo naglašava njegova veličina. Prilikom učitavanja programa u glavnu memoriju, pre pozivanja funkcije `main` vrši se inicijalizacija ovog segmenta na nulu. Primetimo i da ime lokalne automatske promenljive `i` uopšte nije vidljivo u objektnom modulu (što se i moglo očekivati, jer je ova promenljiva bez povezanosti).

Mašinski kôd funkcije `main` može da se vidi, na primer, komandom `objdump -d program.o`.

```
00000000 <main>:
   0: 55                    push   ebp
   1: 89 e5                 mov    ebp,esp
   3: 83 e4 f0              and    esp,0xfffffff0
   6: 83 ec 20              sub    esp,0x20
   9: c7 44 24 1c 00 00 00  mov    DWORD PTR [esp+0x1c],0x0
  10: 00
  11: eb 16                 jmp    29 <main+0x29>
  13: 8b 15 00 00 00 00     mov    edx,DWORD PTR ds:0x0
```

```
 19: 8b 44 24 1c           mov    eax,DWORD PTR [esp+0x1c]
 1d: 01 d0                 add    eax,edx
 1f: a3 00 00 00 00        mov    ds:0x0,eax
 24: 83 44 24 1c 01        add    DWORD PTR [esp+0x1c],0x1
 29: a1 00 00 00 00        mov    eax,ds:0x0
 2e: 39 44 24 1c           cmp    DWORD PTR [esp+0x1c],eax
 32: 7c df                 jl     13 <main+0x13>
 34: a1 00 00 00 00        mov    eax,ds:0x0
 39: 89 44 24 04           mov    DWORD PTR [esp+0x4],eax
 3d: c7 04 24 00 00 00 00  mov    DWORD PTR [esp],0x0
 44: e8 fc ff ff ff        call   45 <main+0x45>
 49: b8 00 00 00 00        mov    eax,0x0
 4e: c9                    leave
 4f: c3                    ret
```

Adrese u generisanom kôdu su relativne (npr. adrese skokova su sve date u odnosu na početak funkcije `main`) i neke od njih će nakon povezivanja i tokom izvršavanja biti promenjene. Na primer, u instrukciji `29: mov eax,ds:0x0` vrši se kopiranje određene vrednosti iz memorije u registar `eax`. Trenutni kôd govori da je to sadržaj sa adrese 0 (u odnosu na početak segmenta podataka `ds`), ali u izvršnom programu to neće ostati tako. Zadatak linkera je da na ovom mestu adresu 0 zameni adresom promenljive `n` (jer je zadatak ove instrukcije upravo da vrednost promenljive `n` prepiše u registar `eax`). Slično tome, na mesto poziva funkcije `printf` u mašinskom kôdu generisan je poziv `44: call 45 <main+0x45>` koji nije ispravan i nema smisla jer ukazuje upravo na adresu argumenta tog poziva (koji je na 44). Zadatak linkera je da ovaj poziv korektno razreši. Spisak stavki koje linker treba da razreši mogu da se vide komandom `objdump -r program.o` (tzv. relokacije) i sam kôd nema nikakvo smisleno značenje dok se u obzir ne uzme tabela relokacije i dok se sve navedene stavke ne razreše (što je zadatak linkera).

```
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000015 R_386_32          sum
00000020 R_386_32          sum
0000002a R_386_32          n
00000035 R_386_32          sum
00000040 R_386_32          .rodata
00000045 R_386_PC32        printf


RELOCATION RECORDS FOR [.eh_frame]:
OFFSET   TYPE              VALUE
00000020 R_386_PC32        .text
```

Iz ovoga se vidi se da je jedan od zadataka linkera da korektnu adresu mašinskog kôda funkcije `printf` umetne na adresu `45` u segment kôda (`.text`) čime će poziv funkcije `printf` postati ispravan. Slično, potrebno je razrešiti i adresu promenljive `n` koja se koristi na mestu `2a` u mašniskom kôdu (za koju je, kao što smo opisali, trenutno pretpostavljeno da se nalazi na adresi `0` u segmentu podataka, što ne mora biti slučaj u konačnom kôdu), adresu promenljive `sum` i to na tri mesta na kojima joj se pristupa (`15`, `20` i `35`) i adresu početka segmenta `.rodata` (engl. read only data) u kome se nalazi konstantna niska `sum = %d\n`, za koji se trenutno pretpostavlja da je 0, a koja se koristi na

mestu `40` u mašinskom kôdu. Takođe, potrebno je da linker postavi i pravu adresu segmenta kôda (`.text`) u zaglavlju izvršne datoteke (`.eh_frame`).

Iako nije od presudnog značaja za programiranje u C-u, analiza prethodnog kôda može biti zanimljiva i pokazuje kako se pojednostavljeno opisani mehanizmi funkcionisanja izvršnog okruženja, realno odvijaju u praksi. Prevedeni mašinski kôd podrazumeva da programski stek u memoriji raste od viših ka nižim adresama (engl. downward growing). Registar `esp` sadrži uvek adresu vrha steka, dok registar `ebp` sadrži adresu početka stek okvira tekuće funkcije (koja je potrebna pri završetku funkcije kako bi stek okvir mogao da se ukloni). Prve četiri instrukcije predstavljaju tzv. *prolog* tekuće funkcije i služe da pripreme novi stek okvir (čuva se adresa početka starog stek okvira i to opet na steku, adresa početka novog stek okvira postaje adresa tekućeg vrha steka, a vrh steka se podiže (ka manjim adresama) kako bi se napravio prostor za čuvanje lokalnih promenljivih i sličnih podataka na steku). Prevedeni kôd podrazumeva da se promenljiva `i` nalazi na steku i to odmah iznad sačuvane vrednosti početka prvobitnog stek okvira (to je adresa `esp+0x1c` nakon pomeranja vrha steka). Naredba u liniji `9` je inicijalizacija petlje na vrednost 0. Nakon nje, skače se na proveru uslova u liniji `29`. Prvo se kopira vrednost promenljive `n` u registar `eax`, a zatim se vrši poređenje vrednosti promenljive `i` (smeštene u steku na adresi `esp+0x1c`) sa registrom `eax`. Ukoliko je `i` manje od `n`, vrši se povratak na telo petlje (instrukcije od linije `13` do `23`) i korak (instrukcija `24`). Telo petlje kopira vrednost promenljive `sum` (sa adrese koja će biti promenjena) u registar `edx`, kopira vrednost promenljive `i` (sa adrese `esp+0x1c`) u registar `eax`, sabira dva registra i rezultat iz registra `eax` kopira nazad na adresu promenljive `sum`. Korak uvećava vrednost promenljive `i` (na adresi `esp+0x1c`) za jedan. Po završetku petlje, kada uslov petlje više nije ispunjen prelazi se na poziv funkcije `printf` (instrukcije od adrese `34` do `48`). Poziv funkcije teče tako što se na stek (u dva mesta ispod njegovog vrha) upisuju argumenti poziva (u obratnom poretku – vrednost promenljive `sum` koja se kopira preko registra `x`), i zatim adresa format niske (iako je u trenutnom kôdu za obe adrese upisana `0` to će biti promenjeno tokom povezivanja). Poziv `call` iznad ova 3 argumenata postavlja adresu povratka i prenosi kontrolu izvršavanja na odgovarajuću adresu kôda (koju će biti ispravno postavljena tokom povezivanja). Funkcija `main` vraća povratnu vrednost tako što je u liniji `49` upisuje u registar `eax`. Instrukcije `leave` i `ret` predstavljaju *epilog* funkcije `main`. Instrukcija `leave` skida stek okvir tako što u registar `esp` upisuje vrednost koja se nalazi u `ebp`, i zatim restauriše vrednost početka prethodnog stek okvira tako što je skida sa steka (jer se pretpostavlja da je u prologu funkcije ova vrednost postavljena na stek). Instrukcija `ret` zatim tok izvršavanja prenosi na adresu povratka (koja se sada nalazi na vrhu steka).

Dakle, u fazi prevođenja se svaki poziv spoljašnje funkcije i svako referisanje na spoljašnje promenljive i konstante ostavlja nerazrešenim i od linkera se očekuje da ovakve reference razreši. Ako se u datoteci sa ulaznim programom poziva funkcija ili koristi promenljiva koja nije u njoj definisana (već samo deklarisana, kao što je slučaj sa funkcijom `printf` u prethodnom primeru), kreiranje objektnog modula biće uspešno, ali samo od tog objektnog modula nije moguće kreirati izvršni program (jer postoje pozivi funkcije čiji mašinski kôd nije poznat usled nedostatka definicije ili postoje promenljive za koju nije odvojena memorija, tako da referisanje na njih u objekntim modulima nisu ispravna). U fazi povezivanja, linker razrešava pozive funkcija i adrese promen-

ljivih i konstanti tražeći u svim navedenim objektnim modulima (uključujući i objektne module standardne biblioteke) odgovarajuću definiciju (mašniski kôd) takve funkcije tj. definiciju (dodeljenu adresu) takve promenljive.

Ukoliko su svi takvi zahtevi uspešni, linker za sve pozive funkcija upisuje adrese konkretne funkcije koje treba koristiti, za sva korišćenja promenljivih ukazuje na odgovarajuće konkretne promenljive i povezuje sve objektne module u jedan izvršni program.

Kôd dobijen povezivanjem obično i dalje nije u stanju da se autonomno izvršava. Naime, funkcije standardne biblioteke su implementirane tako da koriste određen broj rutina niskog nivoa, ali se sve te rutine ne ugrađuju u izvršni kôd jer bi on tako bio ogroman. Umesto da se te rutine ugrađ uju u izvršni kôd programa, one čine tzv. rantajm biblioteku i pozivaju se dinamički, tek u fazi izvršavanja (videti poglavlje 9.3.1).

Pretpostavimo da je prethodno opisani objektni modul povezan na sledeći način: `gcc -o program program.o`, dajući izvršni program `program`. Ako se on pregleda (npr. komandom `nm`), može da se vidi da postoji znatno više imenovanih simbola (npr. `_start`, `printf@@GLIBC_2.0` itd.). Ako se pogleda mašinski kôd funkcije `main` (npr. komandom `objdump`), može da se vidi da su sporne adrese uspešno razrešene. Međutim, na mestu poziva funkcije `printf` poziva se funkcija `printf@plt`, a koja onda poziva nedefinisanu funkciju `printf@@GLIBC_2.0`. Naime, ova funkcija implementirana je u rantajm biblioteci i njena adresa biće povezana dinamički, na početku izvršavanja programa.

9.3.5 Greške u fazi izvršavanja

U fazi pisanja izvornog i generisanja izvršnog programa od izvornog nije moguće predvideti neke greške koje se mogu javiti u fazi izvršavanja programa. Osim što je moguće da daje pogrešan rezultat (čemu je najčešći razlog pogrešan algoritam ili njegova implementacija), moguće je i da uspešno kompiliran program prouzrokuje greške tokom rada koje uzrokuju prekid njegovog izvršavanja. Te greške mogu biti posledica greške programera koji nije predvideo neku moguću situaciju a mogu biti i greške koje zavise od okruženja u kojem se program izvršava.

Jedna od najčešćih grešaka u fazi izvršavanja je nedozvoljeni pristup memoriji. Ona se, na primer, javlja ako se pokuša pristup memoriji koja nije dodeljena programu ili ako se pokuša izmena memorije koja je dodeljena programu, ali samo za čitanje (engl. read only). Operativni sistem detektuje ovakve situacije i signalizira, što najčešće prouzrokuje prekid programa uz poruku

```
Segmentation fault
```

Najčešći uzrok nastanka ovakve greške je pristup neadekvatno alociranom bloku memorije (npr. ako je niz definisan sa `int a[10];`, a pokuša se pristup elementu `a[1000]` ili se pokuša pristup nealociranoj bloku memorije za koji je predviđena dinamička alokacija o čemu će biti reči u glavi 10), dereferenciranje `NULL` pokazivača (o čemu će biti reči u glavi 10), ili prepunjenje steka usled velikog broja funkcijskih poziva (nastalih najčešće neadekvatno obrađenim izlaskom iz rekurzije ili definisanjem prevelikih lokalnih automatskih promenljivih). Odgovornost za nastanak ovakve greške obično je na progameru.

Ukoliko tokom izvršavanja programa dođe do (celobrojnog) deljenja nulom, biće prijavljena greška:

```
Floating point exception
```

i program će prekinuti rad. Neposredna odgovornost za ovu grešku pripada autoru programa jer je previdео da može doći do deljenja nulom i nije u programu predvideo akciju koja to sprečava.

U fazi izvršavanja može doći i do grešaka u komunikaciji sa periferijama (na primer, pokušaj čitanja iz nepostojeće datoteke na disku) itd.

Pitanja i zadaci za vežbu

Pitanje 9.3.1. *Opisati ukratko memorijske segmente koji se dodeljuju svakom programu koji se izvršava. Šta se, tokom izvršavanja programa, čuva: u stek segmentu, u hip segmentu, u segmentu podataka?*

Pitanje 9.3.2. *U kom segmentu memorije se tokom izvršavanja programa čuvaju: lokalne promenljive;*

Pitanje 9.3.3. *Da li se memorijski prostor za neku promenljivu može nalaziti na programskom steku?*

Pitanje 9.3.4. *Navesti barem tri vrste podataka koje se čuvaju u svakom stek okviru.*

Pitanje 9.3.5. *Kako se u prenose argumenti funkcija? Objasniti kako se taj proces odvija u memoriji računara?*

Pitanje 9.3.6. *Ukoliko se na programskom steku istovremeno nađe više instanci neke funkcije* `f`*, kakva mora biti ta funkcija* `f`*?*

Pitanje 9.3.7. *Da li se za svaku instancu rekurzivne funkcije stvara novi stek okvir? Da li se za svaku instancu rekurzivne funkcije stvara nova kopija funkcije u kôd segmentu?*

Pitanje 9.3.8. *Da li se tokom izvršavanja programa menja sadržaj: (a) segmenta kôda, (b) segmenta podataka, (c) stek segmenta?*

Pitanje 9.3.9. *U kom segmentu memorije ima izmena prilikom pozivanja funkcija? Ukoliko se umesto funkcija koriste makroi, da li će u tom segmentu memorije biti izmena u fazi izvršavanja?*

Pitanje 9.3.10. *U kom segmentu memorije se čuva izvršni mašinski kôd programa? U kojim segmentima memorije se čuvaju podaci? Kakva je veza između životnog veka promenljivih i segmenata memorije u kojima se čuvaju? U kom segmentu memorije se čuvaju automatske promenljive, u kom statičke promenljive, a u kom dinamičke?*

Pitanje 9.3.11. *U kom segmentu memorije se čuvaju argumenti funkcija i kakav im je životni vek? U kom segmentu memorije se čuvaju mašinske instrukcije prevedenog C programa?*

Pitanje 9.3.12. *Za svaku promenljivu u narednom kôdu reći koji joj je nivo dosega, kakav joj je životni vek u kom memorijskom segmentu je smeštena i kakva joj je inicijalna vrednost.*

```
int a;
int f() {
    double b; static float c;
}
```

Glava 10

Pokazivači i dinamička alokacija memorije

U jeziku C, pokazivači imaju veoma značajnu ulogu i teško je napisati iole kompleksniji program bez upotrebe pokazivača. Dodatne mogućnosti u radu sa pokazivačima daje dinamička alokacija memorije. Programer, u radu sa pokazivačima, ima veliku slobodu i širok spektar mogućnosti. To otvara i veliki prostor za efikasno programiranje ali i za greške.

10.1 Pokazivači i adrese

Memorija računara organizovana je u niz uzastopnih bajtova. Uzastopni bajtovi mogu se tretirati kao jedinstven podatak. Na primer, dva (ili četiri, u zavisnosti od sistema) uzastopna bajta mogu se tretirati kao jedinstven podatak celobrojnog tipa. *Pokazivači* (engl. pointer) predstavljaju tip podataka u C-u čije su vrednosti memorijske adrese. Na 16-bitnim sistemima adrese zauzimaju dva bajta, na 32-bitnim sistemima četiri, na 64-bitnim osam, i slično. Iako su, suštinski, pokazivačke vrednosti (adrese) celi brojevi, pokazivački tipovi se razlikuju od celobrojnih i ne mogu se mešati sa njima. Jezik C razlikuje više pokazivačkih tipova. Za svaki tip podataka (i osnovni i korisnički) postoji odgovarajući pokazivački tip. Pokazivači implicitno čuvaju informaciju o tipu onoga na šta ukazuju[1]. Ipak, informacija o tipu pokazivača (kao i za sve osnovne tipove) ne postoji tokom izvršavanja programa — tu informaciju iskoristi kompilator tokom kompilacije i u mašinskom kôdu koristi instrukcije koje odgovaraju konkretnim tipovima. Dakle, tokom izvršavanja programa, pokazivačka promenljiva zauzima samo onoliko bajtova koliko zauzima podatak o adresi.

Tip pokazivača koji ukazuje na podatak tipa `int` zapisuje se `int *`. Slično važi i za druge tipove. Prilikom deklaracije, nije bitno da li postoji razmak između zvezdice i tipa ili zvezdice i identifikatora i kako god da je napisano, zvezdica se vezuje uz identifikator. U narednom primeru, `p1`, `p2` i `p3` su pokazivači koji ukazuju na `int` dok je `p4` tipa `int`:

[1] Jedino pokazivač tipa `void*` nema informaciju o tipu podataka na koji ukazuje, i o tom tipu će biti reči u nastavku.

```
int *p1;
int* p2;
int* p3, p4;
```

Pokazivačka promenljiva može da sadrži adrese promenljivih ili elemenata niza i za to se koristi operator `&`. Unarni operator `&`, *operator referenciranja* ili *adresni operator* vraća adresu svog operanda. On može biti primenjen samo na promenljive i elemente niza, a ne i na izraze ili konstante. Na primer, ukoliko je naredni kôd deo neke funkcije

```
int a=10, *p;
p = &a;
```

onda se prilikom izvršavanja te funkcije, za promenljive `a` i `p` rezerviše prostor u njenom stek okviru. U prostor za `a` se upisuje vrednost `10`, a u prostor za `p` adresa promenljive `a`. Za promenljivu `p` tada kažemo da „pokazuje“ na `a`.

Unarni operator `*` (koji zovemo „operator dereferenciranja“[2]) se primenjuje na pokazivačku promenljivu i vraća sadržaj lokacije na koju ta promenljiva pokazuje, vodeći računa o tipu. Dereferencirani pokazivač može biti l-vrednost i u tom slučaju izmene dereferenciranog pokazivača utiču neposredno na prostor na koji se ukazuje. Na primer, nakon kôda

```
int a=10, *p;
p = &a;
*p = 5;
```

promenljiva `p` ukazuje na `a`, a u lokaciju na koju ukazuje `p` je upisana vrednost `5`. Time je i vrednost promenljive `a` postala jednaka `5`.

Dereferencirani pokazivač nekog tipa, može se pojaviti u bilo kom kontekstu u kojem se može pojaviti podatak tog tipa. Na primer, u datom primeru ispravna bi bila i naredba `*p = *p+a+3;`.

Kao što je već rečeno, pokazivački i celobrojni tipovi su različiti. Tako je, na primer, ako je data deklaracija `int *pa, a;`, naredni kôd neispravan: `pa = a;`. Takođe, neispravno je i `a = pa;`, kao i `pa = 1234`. Moguće je koristiti eksplicitne konverzije (na primer `a = (int)pa;` ili `pa = (int*)a;`), ali ove mogućnosti treba veoma oprezno koristiti i isključivo sa jasnim ciljem (koji ne može da se pogodno ostvari drugačije). Jedina celobrojna vrednost koja se može koristiti i kao pokazivačka vrednost je vrednost `0` — moguće je dodeliti nulu pokazivačkoj promenljivoj i porediti pokazivač sa nulom. Uvek se podrazumeva da pokazivač koji ima vrednost `0` ne može da pokazuje ni na šta smisleno (tj. adresa `0` ne smatra se mogućom). Pokazivač koji ima vrednost `0` nije moguće dereferencirati, tj. pokušaj dereferenciranja dovodi do greške tokom izvršavanja programa (najčešće „segmentation fault“). Umesto konstante `0` obično se koristi simbolička konstanta `NULL`, definisana u zaglavlju `<stdio.h>`, kao jasniji pokazatelj da je u pitanju specijalna pokazivačka vrednost.

U nekim slučajevima, poželjno je imati mogućnost „opšteg“ pokazivača, tj. pokazivača koji može da ukazuje na promenljive različitih tipova. Za to se koristi tip `void*`. Izraze ovog tipa je moguće eksplicitno konvertovati u bilo koji konkretni pokazivački tip, a i ukoliko je potrebno, a nije upotrebljena

[2] Simbol `*` koristi se i za označavanje pokazivačkih tipova i za operator dereferenciranja i treba razlikovati ove njegove dve različite uloge.

eksplicitna konverzija biće primenjena implicitna konverzija prilikom dodele. Naravno, nije moguće vršiti dereferenciranje pokazivača tipa `void*` jer nije moguće odrediti tip takvog izraza kao ni broj bajtova u memoriji koji predstavljaju njegovu vrednost. Pre dereferenciranja, neophodno je konvertovati vrednost ovog pokazivačkog tipa u neki konkretan pokazivački tip.

I na pokazivače se može primeniti ključna reč `const`. Tako `const int* p` označava pokazivač na konstantnu promenljivu tipa `int`, `int *const p` označava konstantni pokazivač na promenljivu tipa `int`, dok `const int *const p` označava konstantni pokazivač na konstantnu promenljivu tipa `int`. U prvom slučaju moguće je menjati pokazivač, ali ne i ono na šta on pokazuje, u drugom je moguće menjati ono na šta ukazuje pokazivač, ali ne i sam pokazivač, dok u trećem slučaju nije moguće menjati ni jedno ni drugo. Vrednost tipa `cost T` može biti dodeljena promenljivoj tipa `T`, a promemljivoj tipa `cost T` ne može biti dodeljena vrednost — pokušaj menjanja vrednosti konstantne promenljive (kao i svakog drugog konstantog sadržaja) dovodi do greške u fazi prevođenja.

Pitanja i zadaci za vežbu

Pitanje 10.1.1. *Koje su informacije pridružene pokazivaču? Koje od ovih informacija se u fazi izvršavanja čuvaju u memoriji dodeljenoj pokazivačkoj promenljivoj?*

Pitanje 10.1.2. *Koliko bajtova zauzima podatak tipa* `unsigned char`*? Koliko bajtova zauzima podatak tipa* `unsigned char*`*?*

Pitanje 10.1.3. *Ako je veličina koju zauzima element nekog tipa* `t` *8 bajtova, koliko onda bajtova zauzima pokazivač na vrednost tipa* `t`*: (a) 4; (b) 8; (c) 16; (d) zavisi od sistema?*

Pitanje 10.1.4. *Ako je promenljiva tipa* `double *` *na konkretnom sistemu zauzima 4 bajta, koliko bajtova na istom sistemu zauzima promenljiva tipa* `unsigned char *`*?*

Pitanje 10.1.5. *Na šta se može primeniti operator referenciranja? Na šta se može primeniti operator dereferenciranja? Kako se označavaju ovi operatori? Šta je njihovo dejstvo? Kakav im je prioritet?*

Pitanje 10.1.6. *Ako je promenljiva* `p` *pokazivačkog tipa, da li je dozvoljeno koristiti izraz* `&p`*? Da li se umesto vrednosti* `p` *može pisati i* `*(&p)`*? Da li se umesto vrednosti* `p` *može pisati i* `&(*p)`*?*

Pitanje 10.1.7. *Kako se označava generički pokazivački tip? Za šta se on koristi? Da li se pokazivač ovog tipa može referencirati? Zašto?*

Pitanje 10.1.8. *Ako je promenjliva tipa* `int*`*, da li joj se može dodeliti celobrojna vrednost? Ako je promenjliva tipa* `double*`*, da li joj se može dodeliti celobrojna vrednost?*

Pitanje 10.1.9. *Kog tipa je promenljiva* `a`*, a kog tipa promenljiva* `b` *nakon deklaracije* `short* a, b;`*? Koju vrednost ima promenljiva* `b` *nakon izvršavanja naredbi* `b = 2; a = &b; *a = 3; b++;`*?*

10.2 Pokazivači i argumenti funkcija

U jeziku C argumenti funkcija se uvek prenose po vrednosti[3]. To znači da promenljiva koja je upotrebljena kao argument funkcije ostaje nepromenjena nakon poziva funkcije. Na primer, ako je definisana sledeća funkcija

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}
```

i ukoliko je ona pozvana iz neke druge funkcije na sledeći način: `swap(x, y)` (pri čemu su `x` i `y` promenljive tipa `int`), onda se kreira stek okvir za `swap`, obezbeđuje se prostor za argumente `a` i `b`, vrednosti `x` i `y` se kopiraju u taj prostor i funkcija se izvršava koristeći samo te kopije. Nakon povratka iz funkcije, vrednosti `x` i `y` ostaju nepromenjene.

Ukoliko se želi da funkcija `swap` zaista zameni vrednosti argumentima, onda ona mora biti definisana drugačije:

```
void swap(int *pa, int *pb) {
  int tmp = *pa;
  *pa = *pb;
  *pb = tmp;
}
```

Zbog drugačijeg tipa argumenata, funkcija `swap` više ne može biti pozvana na isti način kao ranije — `swap(x, y)`. Njeni argumenti nisu tipa `int`, već su pokazivačkog tipa `int*`. Zato se kao argumenti ne mogu koristiti vrednosti `x` i `y` već njihove adrese — `&x` i `&y`. Nakon poziva `swap(&x, &y)`, kreira se stek okvir za `swap`, obezbeđuje se prostor za argumente pokazivačkog tipa `pa` i `pb` i zatim se vrednosti `&x` i `&y` kopiraju u taj prostor. Time se prenos argumenata vrši po vrednosti, kao i uvek. Funkcija se izvršava koristeći samo kopije pokazivača, ali zahvaljujući njima ona zaista pristupa promenljivama `x` i `y` i razmenjuje im vrednosti. Nakon povratka iz funkcije, vrednosti `x` i `y` su razmenjene. Na ovaj način, prenošenjem adrese promenljive, moguće je njeno menjanje u okviru funkcije. Isti taj mehanizam koristi se i u funkciji `scanf` koja je već korišćena.

Prenos pokazivača može da se iskoristi i da funkcija vrati više različitih vrednosti (preko argumenata). Na primer, naredna funkcija računa količnik i ostatak pri deljenju dva cela broja (ne koristeći pritom operatore `/` i `%`):

Program 10.1.

```
#include <stdio.h>

void deljenje(unsigned a, unsigned b,
              unsigned* pk, unsigned* po) {
  *pk = 0; *po = a;
  while (*po >= b) {
```

[3]Kao što je rečeno, u slučaju nizova, po vrednosti se ne prenosi čitav niz, već samo adresa njegovog početka.

```
    (*pk)++;
    *po -= b;
  }
}

int main() {
  unsigned k, o;
  deljenje(14, 3, &k, &o);
  printf("%d %d\n", k, o);
  return 0;
}
```

Kao i sve promenljive, kvalifikatorom `const` mogu biti označeni parametri funkcije čime se obezbeđuje da neki parametar ili sadržaj na koji ukazuje neki parametar neće biti menjan u okviru funkcije. I na tip povratne vrednosti funkcije može se primeniti kvalifikator `const`.

Pitanja i zadaci za vežbu

Pitanje 10.2.1. *Kako se prenose argumenti pokazivačkog tipa? Šta to znači? Objasniti na primeru.*

Pitanje 10.2.2. *Šta ispisuje naredni program?*

1. ```
   int a = 1, b = 2;
   void f(int* p) {
      p = &b;
   }
   int main() {
      int *p = &a
      f(p);
      printf("%d\n", *p);
   }
   ```

2. ```
   #include<stdio.h>
   void f(int* p) { p++; p += 2; p--; p = p + 1; }

   int main() {
      int a[] = {1, 2, 3, 4, 5}, *p = a, *q = a;
      f(p); printf("%d\n", *p);
      q++; q += 2; q--; q = q + 1; printf("%d\n", *q);
   }
   ```

Pitanje 10.2.3. *Ukoliko je potrebno da funkcija vrati više od jedne vrednosti, kako se to može postići?*

Pitanje 10.2.4. *Napisati funkciju* `f` *sa povratnim tipom* `void` *koja ima samo jedan argument tipa* `int` *i vraća ga udvostručenog.*

Napisati funkciju `f` *sa povratnim tipom* `void` *koja ima tačno jedan argument, a koja prosleđenu celobrojnu vrednost vraća sa promenjenim znakom.*

Napisati funkciju sa tipom povratne vrednosti `void` *koja služi za izračunavanje zbira i razlike svoja prva dva argumenta.*

Zadatak 10.1. *Napisati funkciju koja za uneti broj sekundi ($0 \leq s < 86400$) proteklih od prethodne ponoći izračunava trenutno vreme tj. broj sati, minuta i sekundi.* √

10.3 Pokazivači i nizovi

Postoji čvrsta veza između pokazivača i nizova. Operacije nad nizovima mogu se iskazati i u terminima korišćenja pokazivača i u daljem tekstu često nećemo praviti razliku između dva načina za pristupanje elementima niza.

Deklaracija `int a[10];` deklariše niz od 10 elemenata tipa `int`. Izraz `sizeof(a)` imaće istu vrednost kao izraz `10*sizeof(int)`. Početni element niza je `a[0]`, a deseti element je `a[9]` i oni su u memoriji poređani uzastopno. U fazi kompilacije, imenu niza `a` pridružena je informacija o adresi početnog elementa niza, o tipu elemenata niza, kao i o broju elemenata niza. Poslednje dve informacije koriste se samo tokom kompilacije i ne zauzimaju memorijski prostor u fazi izvršavanja.

Za razliku od pokazivača koji jesu l-vrednosti, imena nizova (u navedenom primeru ime `a`) to nisu (jer `a` uvek ukazuje na isti prostor koji je rezervisan za elemente niza). Vrednost `a` nije pokazivačkog tipa, ali mu je vrlo bliska. Izuzev u slučaju kada je u pitanju argument `sizeof` operatora ili `&` operatora, ime niza tipa `T` konvertuje se u pokazivač na `T`. Dakle, ime jednodimenzionog niza (na primer, `a` za deklaraciju `int a[10]`) biće, sem ako je navedeno kao argument `sizeof` ili `&` operatora, implicitno konvertovano u pokazivač (tipa `int*`) na prvi element niza. Vrednosti `a` odgovara pokazivač na prvi element niza, vrednosti `a+1` odgovara pokazivač na drugi element niza, itd. Dakle, umesto `&a[i]` može se pisati `a+i`, a umesto `a[i]` može se pisati `*(a+i)` (o tim izrazima se koristi tzv. pokazivačka aritmetika o kojoj će biti reči u narednom poglavlju). Naravno, kao i obično, nema provere granica niza, pa je dozvoljeno pisati (tj. prolazi fazu prevođenja) i `a[100]`, `*(a+100)`, `a[-100]`, `*(a-100)`, iako je veličina niza samo 10. U fazi izvršavanja, pristupanje ovim lokacijama može da promeni vrednost podataka koji se nalaze na tim lokacijama ili da dovede do prekida rada programa zbog pristupanja memoriji koja mu nije dodeljena.

Kao što se elementima niza može pristupati korišćenjem pokazivačke aritmetike, ako je `p` pokazivač nekog tipa (npr. `int* p`) na njega može biti primenjen nizovski indeksni pristup (na primer, `p[3]`). Vrednost takvog izraza određuje se tako da se poklapa sa odgovarajućim elementom niza koji bi počinjao na adresi `p` (bez obzira što `p` nije niz nego pokazivač). Na primer, ako bi na sistemu na kojem je `sizeof(int)` jednako 4 pokazivač `p` ukazivao na adresu `100` na kojoj počinje niz celih brojeva, tada bi `p[3]` bio ceo broj koji se nalazi smešten u memoriji počevši adrese `112`. Isti broj bio bi dobijen i izrazom `*(p+3)` u kome se koristi pokazivačka aritmetika.

Dakle, bez obzira da li je `x` pokazivač ili niz, `x[n]` isto je što i `*(x+n)`, tj. `x+n` isto je što i `&x[n]`. Ovako tesna veza pokazivača i nizova daje puno prednosti (na primer, iako funkcije ne primaju niz već samo pokazivač na prvi element, implementacija tih funkcija može to da zanemari i sve vreme da koristi indeksni pristup kao da je u pitanju niz, a ne pokazivač, što je korišćeno u ranijim poglavljima). Međutim, ovo je i čest izvor grešaka, najčešće prilikom alokacije memorije. Na primer,

```
int a[10];
```

```
int *b;
a[3] = 5; b[3] = 8;
```

niz `a` je ispravno deklarisan i moguće je pristupiti njegovom elementu na poziciji `3`, dok u slučaju pokazivača `b` nije izvršena alokacija elemenata na koje se pokazuje i, iako se pristup `b[3]` ispravno prevodi, on bi najverovatnije doveo do greške prilikom izvršavanja programa. Zato, svaki put kada se koristi operator indeksnog pristupa, potrebno je osigurati ili proveriti da je memorija alocirana na odgovarajući način (jer kompilator ne vrši takve provere).

Kao što je rečeno, izraz `a` ima vrednost adrese početnog elementa i tip `int [10]` koji se, po potrebi, može konvertovati u `int *`. Izraz `&a` ima istu vrednost (tj. sadrži istu adresu) kao i `a`, ali drugačiji tip — tip `int (*)[10]` — to je tip pokazivača na niz od 10 elemenata koji su tipa `int`. Taj tip ima u narednom primeru promenljiva `c`:

```
char a[10];
char *b[10];
char (*c)[10];
```

U navedenom primeru, promenljiva `a` je niz od 10 elemenata tipa `char` i zauzima 10 bajtova. Promenljiva `b` je niz `10` pokazivača na `char` i zauzima prostor potreban za smeštanje 10 pokazivača. Promenljiva `c` je (jedan) pokazivač na niz od `10` elemenata tipa `char` i zauzima onoliko prostora koliko i bilo koji drugi pokazivač. Vrednost se promenljivoj `c` može pridružiti, na primer, naredbom `c=&a;`. Tada je `(*c)[0]` isto što i `a[0]`. Ukoliko je promenljiva `c` deklarisana, na primer, sa `char (*c)[5];`, u dodeli `c=&a;` će biti izvršena implicitna konverzija i ova naredba će proći kompilaciju (uz upozorenje da je izvršena konverzija neusaglašenih tipova), ali to bi trebalo izbegavati.

Kao što je rečeno u poglavlju 8.7, niz se ne može preneti kao argument funkcije. Umesto toga, kao argument funkcije se može navesti ime niza i time se prenosi samo pokazivač koji ukazuje na početak niza. Funkcija koja prihvata takav argument za njegov tip može da ima `char a[]` ili `char *a`. Ovim se u funkciju kao argument prenosi (kao i uvek — po vrednosti) samo adresa početka niza, ali ne i informacija o dužini niza.

Kako se kao argument nikada ne prenosi čitav niz, već samo adresa početka, moguće je umesto adrese početka proslediti i pokazivač na bilo koji element niza, kao i bilo koji drugi pokazivač odgovarajućeg tipa. Na primer, ukoliko je `a` ime niza i ako funkcija `f` ima prototip `int f(int x[]);` (ili — ekvivalentno — `int f(int *x);`), onda se funkcija može pozivati i za početak niza (sa `f(a)`) ili za pokazivač na neki drugi element niza (na primer, `f(&a[2])` ili — ekvivalentno — `f(a+2)`). Naravno, ni u jednom od ovih slučajeva funkcija `f` nema informaciju o tome koliko elemenata niza ima nakon prosleđene adrese.

Pitanja i zadaci za vežbu

Pitanje 10.3.1. *Ako je u okviru funkcije deklarisan niz sa* `char a[10]`*; na koji deo memorije ukazuje* `a+9`*?*

Pitanje 10.3.2. *Ako je niz deklarisan sa* `int a[10];`*, šta je vrednost izraza* `a`*? Šta je vrednost izraza* `a + 3`*? Šta je vrednost izraza* `*(a+3)`*?*

Pitanje 10.3.3. *Da li se komanda* `ip = &a[0]` *može zameniti komandom (odgovoriti za svaku pojedinačno) (a)* `ip = a[0]`*; (b)* `ip = a`*; (c)* `ip = *a`*; (d)* `ip = *a[0]`*?*

Pitanje 10.3.4. *Da li je vrednost* `a[i]` *ista što i (odgovoriti za svaku pojedinačno) (a)* `a[0]+i`*; (b)* `a+i`*; (c)* `*(a+i)`*; (d)* `&(a+i)`*?*

Pitanje 10.3.5. *Ukoliko je niz deklarisan na sledeći način:* `float* x[10]`*, kako se može dobiti adresa drugog elementa niza?*

Pitanje 10.3.6. *Neka je niz* `a` *deklarisan sa* `int a[10]`*, i neka je* `p` *pokazivač tipa* `int*`*. Da li je naredba* `a = p;` *ispravna? Da li je naredba* `p = a;` *ispravna? Koja je vrednost izraza* `sizeof(p)` *nakon naredbe* `p = a;`*, ako* `int*` *zauzima četiri bajta?*

Pitanje 10.3.7. *Niz je deklarisan sa* `int a[10];`*. Da li je* `a+3` *ispravan izraz? Da li mu je vrednost ista kao i vrednost* `a[3]`*,* `&a[3]`*,* `*a[3]` *ili* `a[10]`*? Da li je* `*(a+3)` *ispravan izraz? Da li mu je vrednost ista kao i vrednost* `a[3]`*,* `a[10]`*,* `*a[3]` *ili* `*a[10]`*?*

Pitanje 10.3.8. *Šta ispisuje naredni kôd:*

```
int a = 3, b[] = {8, 6, 4, 2}, *c = &a,  *d = b + 2; *c = d[-1];
printf("%d\n", a);
```

```
int a[] = {7, 8, 9}; int *p; p = a;
printf("%d %d\n", *(p+2), sizeof(p));
```

Pitanje 10.3.9. *Ako je kao argument funkcije navedeno ime niza, šta se sve prenosi u funkciju: (i) adresa početka niza; (ii) elementi niza; (iii) podatak o broju elemenata niza; (iv) adresa kraja niza?*

Pitanje 10.3.10. *Koji prototip je ekvivalentan prototipu* `int f(char s[])`*?*

Da li ima razlike između prototipova `void f(char *x);` *i* `void f(char x[]);` *?*

Da li ima razlike između deklaracija `char* x;` *i* `char x[];`*? Da li su obe ispravne?*

10.4 Pokazivačka aritmetika

U prethodnom poglavlju je rečeno da nakon deklaracije `int a[10];`, vrednosti `a` odgovara pokazivač na prvi element niza, vrednosti `a+1` odgovara pokazivač na drugi element niza, itd. Izraz `p+1` i slični uključuju izračunavanje koje koristi *pokazivačku aritmetiku* i koje se razlikuje od običnog izračunavanja. Naime, izraz `p+1` ne označava dodavanje vrednosti `1` na `p`, već dodavanje dužine jednog objekta tipa na koji ukazuje `p`. Na primer, ako `p` ukazuje na `int`, onda `p+1` i `p` mogu da se razlikuju za dva ili četiri — za onoliko koliko bajtova na tom sistemu zauzima podatak tipa `int` (tj. vrednosti `p+1` i `p` se razlikuju za `sizeof(int)` bajtova). Na primer, ako je `p` pokazivač na `int` koji sadrži adresu 100, na sistemu na kojem `int` zauzima 4 bajta, vrednost `p+3` će biti adresa $100 + 3 \cdot 4 = 112$. Analogno važi za druge tipove.

Od pokazivača je moguće oduzimati cele brojeve (na primer, `p-n`), pri čemu je značenje ovih izraza analogno značenju u slučaju sabiranja. Na pokazivače

je moguće primenjivati prefiksne i postfiksne operatore `++` i `--`, sa sličnom semantikom. Dva pokazivača je moguće oduzimati. I u tom slučaju se ne vrši prosto oduzimanje dve adrese, već se razmatra veličina tipa pokazivača, sa kojom se razlika deli. Dva pokazivača nije moguće sabirati.

Pored dereferenciranja, dodavanja i oduzimanja celih brojeva, i oduzimanja pokazivača, nad pokazivačima se (samo ako su istog tipa) mogu primenjivati i relacijski operatori (na primer, `p1 < p2`, `p1 == p2`, ...). Tako je, na primer, `p1 < p2` tačno akko `p1` ukazuje na raniji element (u smislu linearno uređene memorije) niza od pokazivača `p2`.

Unarni operatori `&` i `*` imaju viši prioritet nego binarni aritmetički operatori. Zato je značenje izraza `*p+1` zbir sadržaja lokacije na koju ukazuje `p` i vrednosti 1 (a ne sadržaj na adresi `p+1`). Unarni operatori `&`, `*`, i prefiksni `++` se primenjuju zdesna nalevo, pa naredba `++*p;` inkrementira sadržaj lokacije na koju ukazuje `p`. Postfiksni operator `++`, kao i svi unarni operatori koji se primenjuju sleva na desno, imaju viši prioritet od unarnih operatora koji se primenjuju zdesna nalevo, pa `*p++` vraća sadržaj na lokaciji `p`, dok se kao bočni efekat `p` inkrementira.

Vezu pokazivača i nizova i pokazivačku aritmetiku ilustruje naredni jednostavni primer.

Program 10.2.

```
#include <stdio.h>
int main() {
  int a[] = {8, 7, 6, 5, 4, 3, 2, 1};
  int *p1, *p2, *p3;
  /* Ispis elemenata niza */
  printf("%d %d %d\n", a[0], a[3], a[5]);

  /* Inicijalizacija pokazivaca -
     ime niza se konvertuje u pokazivac */
  p1 = a; p2 = &a[3]; p3 = a + 5;
  printf("%d %d %d\n", *p1, *p2, *p3);

  /* Pokazivaci se koristi u nizovskoj sintaksi */
  printf("%d %d %d\n", p1[1], p2[2], p3[-1]);

  /* Pokazivacka aritmetika */
  p1++; --p2; p3 += 2; /* a++ nije dozvoljeno */
  printf("%d %d %d %lu\n", *p1, *p2, *p3, p3 - p1);

  /* Odnos izmedju prioriteta * i ++ */
  *p1++; printf("%d ", *p1);
  *(p1++); printf("%d ", *p1);
  (*p1)++; printf("%d\n", *p1);
  return 0;
}
```

```
8 5 3
8 5 3
```

```
7 3 4
7 6 1 6
6 5 6
```

Pitanja i zadaci za vežbu

Pitanje 10.4.1. *Ako je* `p` *tipa* `int*`*, šta je efekat naredbe* `*p += 5;` *a šta efekat naredbe* `p += 5;`*?*

Pitanje 10.4.2. *Ako su promenljive* `p` *i* `q` *tipa* `double *`*, za koliko će se razlikovati vrednosti* `p` *i* `q` *nakon komande* `p = q+n` *(pri čemu je* `n` *celobrojna promenljiva)?*

Pitanje 10.4.3. *Ako je promenljiva* `p` *tipa* `double *`*, za koliko će se promeniti njena vrednost (a) nakon komande* `++*p;`*? (b) nakon komande* `++(*p);`*? (c) nakon komande* `*p++;`*? (d) nakon komande* `(*p)++;`*?*

Da li postoji razlika između komandi `(*p)++` *i* `*(p++)`*?*

Pitanje 10.4.4.

1. *Koje binarne operacije su dozvoljene nad vrednostima istog pokazivačkog tipa (npr. nad promenljivima* `p1` *i* `p2` *tipa* `int*`*)?*
2. *Koje binarne aritmetičke operacije se mogu primeniti nad pokazivačem i celim brojem (npr. nad promenljivom* `p` *tipa* `int*` *i promenljivom* `n` *tipa* `int`*)?*

Pitanje 10.4.5. *Ako su* `p1`*,* `p2` *pokazivači istog tipa, da li postoji razlika između vrednosti:* `(p1-p2)+d` *i* `p1-(p2-d)` *? Kog su tipa navedene vrednosti?*

Pitanje 10.4.6. *Ako su* `p1`*,* `p2`*,* `p3` *pokazivači istog tipa, da li je naredna konstrucija ispravna:*

(1) `p=NULL;`	*(2)* `p=10;`	*(3)* `p1=p1+0;`
(4) `p1=p1+10;`	*(5)* `p1=(p1+p1)+0;`	*(6)* `p1=(p1+p1)+10;`
(7) `p1=(p1-p1)+0;`	*(8)* `p1=(p1-p1)+10;`	*(9)* `p1=(p1-p2)+10;`
(10) `p1=p1-(p2+10);`	*(11)* `p1=p1+(p1-p2);`	*(12)* `p1=p1-(p1-p2);`
(13) `p1=p1+(p2-p3);`	*(14)* `p1=(p1+p2)-p3;`	*(15)* `p1=(p1+p2)+p3;`
(16) `p1=p1+(p2+p3);`		

Pitanje 10.4.7.

1. *Za koliko bajtova se menja vrednost pokazivača* `p` *tipa* `int *` *nakon naredbe* `p += 2;`*?*
2. *Ako je* `p` *tipa* `float*`*, za koliko se razlikuju vrednosti* `p+5` *i* `p`*?*
3. *Za koliko bajtova se menja vrednost pokazivača* `p` *tipa* `T *` *(gde je* `T` *neki konkretan tip), nakon naredbe* `p += 2;`*?*
4. *Ako je* `p` *pokazivač tipa* `T`*, čemu je jednako* `p + 3`*?*

Pitanje 10.4.8.

1. *Kog je tipa* `x`*, a kog tipa* `y` *nakon deklaracije* `int *x, *y;`*? Da li su dozvoljene operacije* `x+y` *i* `x-y`*?*

2. *Kog je tipa* `a`*, a kog tipa* `b` *nakon deklaracije* `int* a, b;`*? Da li su dozvoljene operacije* `a+b` *i* `a-b`*?*

Pitanje 10.4.9. *Da li nakon* `int n[] = {1, 2, 3, 4, 5}; int* p = n;`*, sledeće naredbe menjaju sadržaj niza* `n`*, vrednost pokazivača* `p`*, ili ni jedno ni drugo ili oba: (a)* `*(p++);` *(b)* `(*p)++;` *(c)* `*p++;`*?*

10.5 Pokazivači i niske

Konstantne niske (na primer `"informatika"`) tretiraju se isto kao nizovi karaktera. Karakteri su poređani redom na uzastopnim memorijskim lokacijama i iza njih se nalazi završna nula (`'\0'`) koja označava kraj niske. Konstantne niske se u memoriji uvek čuvaju u segmentu podataka. Kada se u pozivu funkcije navede konstantna niska (na primer, u pozivu `printf("%d", x);`), funkcija umesto niske prihvata kao argument zapravo adresu početka konstantne niske u segmentu podataka.

Razmotrimo, kao primer, deklaraciju `char *p = "informatika";`. Ukoliko je takva deklaracija navedena u okviru neke funkcije, onda se u njenom stek okviru rezerviše prostor samo za promenljivu `p` ali ne i za nisku `"informatika"` — ona se čuva u segmentu podataka i `p` ukazuje na nju. Situacija je slična ako je deklaracija sa inicijalizacijom `char *p = "informatika";` spoljašnja (tj. globalna): promenljiva `p` će se čuvati u segmentu podataka i ukazivati na nisku `"informatika"` — koja se čuva negde drugde u segmentu podataka. U oba slučaja, promenljiva `p` ukazuje na početak niske `"informatika"` i `p[0]` je jednako `'i'`, `p[1]` je jednako `'n'` i tako dalje. Pokušaj da se promeni sadržaj lokacije na koji ukazuje `p` (na primer, `p[0]='x';` ili `p[1]='x';`) prolazi fazu prevođenja, ali u fazi izvršavanja dovodi do nedefinisanog ponašanja (najčešće do greške u fazi izvršavanja i prekida izvršavanja programa). S druge strane, promena vrednosti samog pokazivača `p` (na primer, `p++;`) je u oba slučaja dozvoljena. Dakle, `p` jeste l-vrednost, ali, na primer, `p[0]` nije.

U gore navedenim primerima koristi se pokazivač `p` tipa `char*`. Značenje inicijalizacije je bitno drugačije ako se koristi niz. Na primer, deklaracijom sa inicijalizacijom `char a[] = "informatika";` kreira se niz `a` dužine `12` i prilikom inicijalizacije on se popunjava karakterima niske `"informatika"`. U ovoj situaciji, dozvoljeno je menjanje vrednosti `a[0]`, `a[1]`, ..., `a[9]`, ali nije dozvoljeno menjanje vrednosti `a`. Dakle, `a` nije l-vrednost, ali, na primer, `a[0]` jeste.

Razmotrimo kao ilustraciju rada sa pokazivačima na karaktere nekoliko mogućih implementacija funkcije `strlen` koja vraća dužinu zadate niske. Njen argument `s` je pokazivač na početak niske (koja se, kao i obično, može tretirati kao niz). Ukoliko se funkcija pozove sa `strlen(p)`, promenljiva `s` nakon poziva predstavlja kopiju tog pokazivača `p` i može se menjati bez uticaja na originalni pokazivač koji je prosleđen kao argument:

```
size_t strlen(const char *s) {
   size_t n;
   for (n = 0; *s != '\0'; s++)
     n++;
   return n;
}
```

Još jedna verzija ove funkcije koristi mogućnost oduzimanja pokazivača (umesto `*s != '\0'`, korišćen je kraći, a ekvivalentan izraz `*s`):

```
size_t strlen(const char *s) {
   const char* t = s;
   while(*s)
      s++;
   return s - t;
}
```

Zbog neposrednih veza između nizova i pokazivača, kao i načina na koji se obrađuju, navedena funkcija `strlen` može se pozvati za argument koji je konstantna niska (na primer, `strlen("informatika");`, za argument koji je ime niza (na primer, `strlen(a);` za niz deklarisan sa `char a[10];`), kao i za pokazivač tipa `char*` (na primer, `strlen(p);` za promenljivu `p` deklarisanu sa `char *p;`.

Kao drugi primer, razmotrimo nekoliko mogućih implementacija funkcije `strcpy` koja prihvata dva karakterska pokazivača (ili dve niske) i sadržaj druge kopira u prvu. Da bi jedna niska bila iskopirana u drugu nije, naravno, dovoljno prekopirati pokazivač, već je neophodno prekopirati svaki karakter druge niske pojedinačno. U narednom kôdu, oba pokazivača se povećavaju za po jedan (operatorom inkrementiranja), sve dok drugi pokazivač ne dođe do završne nule, tj. do kraja niske (kako se argumenti prenose po vrednosti, promenljivama `s` i `t` se može menjati vrednost, a da to ne utiče na originalne pokazivače ili nizove koji su prosleđeni kao argumenti):

```
void strcpy(char *s, const char *t) {
  while ((*s = *t) != '\0') {
     s++;
     t++;
  }
}
```

Inkrementiranje koje se koristi u navedenom kôdu, može se izvršiti (u postfiksnom obliku) i u okviru same `while` petlje:

```
void strcpy(char *s, const char *t) {
   while ((*s++ = *t++) != '\0');
}
```

Konačno, poređenje sa nulom može biti izostavljeno jer svaka ne-nula vrednost ima istinitosnu vrednost *tačno*:

```
void strcpy(char *s, const char *t) {
  while (*s++ = *t++);
}
```

U navedenoj implementaciji funkcije `strcpy` kvalifikatorom `const` obezbeđeno je da se sadržaj lokacija na koje ukazuje `t` neće menjati (u protivnom dolazi do greške u fazi prevođenja.

Pitanja i zadaci za vežbu

Pitanje 10.5.1. *U kom segmentu memorije se tokom izvršavanja programa čuvaju: konstantne niske?*

Pitanje 10.5.2. *Nakon koda*

```
void f() {
  char* a = "Zdravo";
  char b[] = "Zdravo";
  ...
}
```

u kom segmentu memorije je smeštena promenljiva `a`*? u kom je segmentu ono na šta pokazuje promenljiva* `a`*? U kom segmentu su smešteni elementi niza* `b`*? Da li se vrednosti* `a` *i* `b` *mogu menjati? Koliko bajtova zauzima* `a`*, a koliko* `p`*?*

Pitanje 10.5.3. *Koju vrednost ima promenljiva* `i` *nakon naredbi:*

```
int i;
char a[] = "informatika";
i = strlen(a+2) ? strlen(a+4) : strlen(a+6);
```

Pitanje 10.5.4. *Razmotriti funkciju:*

```
char *dan(int n) {
   static char *ime[] = {
     "neispravna vrednost", "ponedeljak", "utorak", "sreda",
     "cetvrtak", "petak","subota", "nedelja"
   };
   return (n < 1 || n > 7) ? ime[0] : ime[n];
}
```

Šta se postiže navedenim kvalifikatorom `static`*?*
Koliko bajtova zauzima niz `ime` *i u kom delu memorije?*
U kojem delu memorije bi se nalazio niz `ime` *da nije naveden kvalifikator* `static`*?*
Na koji deo memorije ukazuje `ime[0]`*?*

Zadatak 10.5.1. *Navesti jednu moguću implementaciju funkcije* `strlen`*. Navesti jednu moguću implementaciju funkcije* `strcpy`*. Navesti jednu moguću implementaciju funkcije* `strcmp`*. Navesti jednu moguću implementaciju funkcije* `strrev`*. Navesti jednu moguću implementaciju funkcije* `strchr`*. Navesti jednu moguću implementaciju funkcije* `strstr`*. Napomena: sve vreme koristiti pokazivačku sintaksu.* √

10.6 Nizovi pokazivača i višedimenzioni nizovi

Kao što je rečeno u poglavlju 10.3, izuzev u slučaju ako je u pitanju argument `sizeof` ili `&` operatora, ime niza tipa `T` se konvertuje u pokazivač na `T`. To važi i za višedimenzione nizove. Ime niza tipa `T a[d1][d2]...[dn]` se konvertuje u pokazivač na `n-1`-dimenzioni niz, tj. u pokazivač tipa `T (*)[d2]...[dn]`. Razmotrimo, na primer, deklaraciju dvodimenzionog niza:

```
int a[10][20];
```

Svaki od izraza `a[0]`, `a[1]`, `a[2]`, ... označava niz od `20` elemenata tipa `int`, te ima tip `int [20]` (koji se, po potrebi, implicitno konvertuje u tip `int*`). Dodatno, `a[0]` sadrži adresu elementa `a[0][0]` i, opštije, `a[i]` sadrži adresu elementa `a[i][0]`. Sem u slučaju kada je argument `sizeof` ili `&` operatora, vrednost `a` se konvertuje u vrednost tipa `int (*)[20]` — ovo je tip pokazivača na niz od 20 elemenata. Slično, izrazi `&a[0]`, `&a[1]`, `...` su tipa pokazivača na niz od 20 elemenata, tj. `int (*)[20]`. Izrazi `a` i `a[0]` imaju istu vrednost (adresu početnog elementa dvodimenzionog niza), ali različite tipove: prvi ima tip `int (*)[20]`, a drugi tip `int*`. Ovi tipovi ponašaju se u skladu sa opštim pravilima pokazivačke aritmetike. Tako je, na primer, `a+i` jednako `a[i]`, a `a[0]+i` je jednako `&a[0][i]`.

Razmotrimo razliku između dvodimenzionalnog niza i niza pokazivača. Ako su date deklaracije

```
int a[10][20];
int *b[10];
```

i `a[3][4]` i `b[3][4]` su sintaksički ispravna referisanja na pojedinačni `int`. Ali `a` je pravi dvodimenzioni niz: 200 lokacija za podatak tipa `int` je rezervisano i uobičajena računica `20 * v + k` se koristi da bi se pristupilo elementu `a[v][k]`. Za niz `b`, međutim, deklaracija alocira samo 10 pokazivača i ne inicijalizuje ih — inicijalizacija se mora izvršiti eksplicitno, bilo statički (navođenjem inicijalizatora) ili dinamički (tokom izvršavanja programa). Pod pretpostavkom da svaki element niza `b` zaista pokazuje na niz od 20 elemenata, u memoriji će biti 200 lokacija za podataka tipa `int` i još dodatno 10 lokacija za pokazivače. Ključna prednost niza pokazivača nad dvodimenzionalnim nizom je činjenica da vrste na koje pokazuju ovi pokazivači mogu biti različite dužine. Tako, svaki element niza `b` ne mora da pokazuje na 20-to elementni niz — neki mogu da pokazuju na 2-elementni niz, neki na 50-elementi niz, a neki mogu da budu `NULL` i da ne pokazuju nigde.

Razmotrimo primer niza koji treba da sadrži imena meseci. Jedno rešenje je zasnovano na dvodimenzionalnom nizu (u koji se, prilikom inicijalizacije upisuju imena meseci):

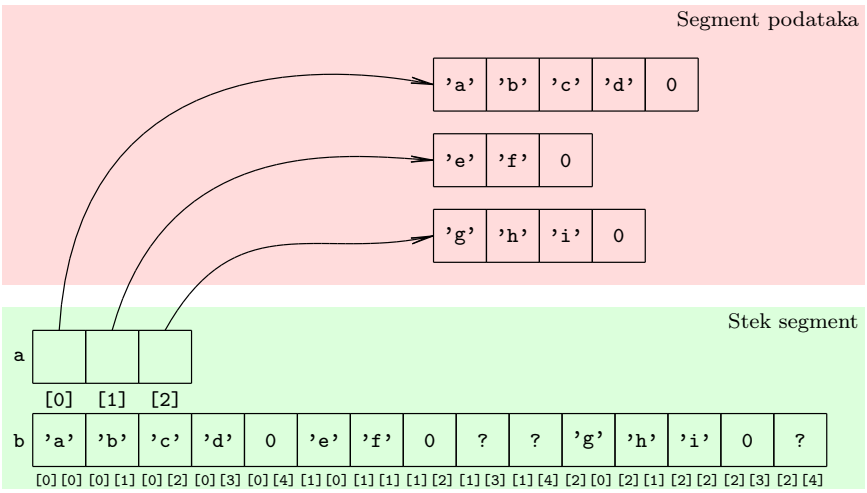
```
char meseci[][10] = {
   "Greska", "Januar", "Februar", "Mart", "April",
   "Maj", "Jun", "Jul", "Avgust", "Septembar",
   "Oktobar", "Novembar", "Decembar"};
```

Pošto meseci imaju imena različite dužine, bolje rešenje je napraviti niz pokazivača na karaktere i inicijalizovati ga da pokazuje na konstantne niske smeštene u segmentu podataka (primetimo da nije potrebno navesti broj elemenata niza pošto je izvršena inicijalizacija):

```
char *meseci[] = {
   "Greska", "Januar", "Februar", "Mart", "April",
   "Maj", "Jun", "Jul", "Avgust", "Septembar",
   "Oktobar", "Novembar", "Decembar"};
```

Slika 10.1 ilustruje sličan primer – memoriju koja odgovara nizovima `a` i `b` deklarisanim u okviru neke funkcije:

```
char* a[] = {"abcd", "ef", "ghi"};
char  b[][5] = {"abcd", "ef", "ghi"};
```



Slika 10.1: Niz pokazivača i dvodimenzioni niz u memoriji

Pitanja i zadaci za vežbu

Pitanje 10.6.1. *Navesti primer inicijalizacije dvodimenzionog niza brojeva tipa* `int`*. Navesti primer deklaracije trodimenzionog niza brojeva tipa* `float`*.*

Pitanje 10.6.2. *Da li se matrica* `a` *dimenzije* 3×3 *deklariše na sledeći način (odgovoriti za svaku pojedinačno): (a)* `a[3][3][3]`*; (b)* `a[3,3,3]`*; (c)* `a[3,3]`*; (d)* `a[3][3]`*; (e)* `a[3*3]`*?*

Pitanje 10.6.3. *Nakon deklaracije* `int a[10];` *kog tipa je vrednost* `a`*? Nakon deklaracije* `int b[10][20];` *kog tipa je vrednost* `b`*? Nakon deklaracije* `int b[10][20];` *kog tipa je vrednost* `b[0]`*?*

Pitanje 10.6.4. *Objasniti naredne dve deklaracije i u čemu se razlikuju:*

```
char *name[] = { "Illegal month", "Jan", "Feb", "Mar" };
char aname[][15] = { "Illegal month", "Jan", "Feb", "Mar" };
```

Pitanje 10.6.5. *Kako se deklariše trodimenzioni niz* `a` *dimenzija 10, 9, 8?*

Pitanje 10.6.6. *Koliko bajtova će biti rezervisano za naredne nizove:*

```
char *name[] = { "Illegal month", "Jan", "Feb", "Mar" };
char aname[][15] = { "Illegal month", "Jan", "Feb", "Mar" };
```

Pitanje 10.6.7.

1. *Da li je neposredno nakon deklaracije* `char a[3][3]`*, naredba* `a[0][0] = 'a';`
   (a) sintaksički ispravna? (b) semantički ispravna?

2. *Da li je neposredno nakon deklaracije* `char* a[3];`*, naredba* `a[0][0] = 'a';`
   (a) sintaksički ispravna? (b) semantički ispravna?

Pitanje 10.6.8. *Koliko bajtova zauzima i u kom delu memorije niz koji je u okviru neke funkcije deklarisan sa* `char* a[10];`

Pitanje 10.6.9. *Koja inicijalizacija dvodimenzionog niza je ispravna:*
(a) `int a[][2]=(1,2,3,4);` *(b)* `int a[2][]=(1,2,3,4);`
(c) `int a[][2]={1,2,3,4};` *(d)* `int a[2][]={1,2,3,4};`

Pitanje 10.6.10. *Koliko je, na 32-bitnim sistemima,* `sizeof(a)` *i* `sizeof(b)` *nakon deklaracija* `int* a[4]; int b[3][3];`*? Koliko je u opštem slučaju?*

Pitanje 10.6.11. *Navesti primer inicijalizacije niza pokazivača. Opisati efekat narednih deklaracija:* `int a[10][20]; int *b[10];`

Pitanje 10.6.12. *Pretpostavimo da se naredni kôd izvršava na 32-bitnom sistemu.*

```
char* a[] = {"Dobar dan!", "Zdravo, zdravo"};
char b[][15] = {"Dobar dan!", "Zdravo, zdravo"};
```

Čemu je jednako `sizeof(a)` *a čemu* `sizeof(b)`*?*

Pitanje 10.6.13. *Razmotriti naredni kôd:*

```
void f() {
  char* a = "Zdravo";
  char b[] = "Zdravo";
  ...
}
```

U kom segmentu memorije se nalazi promenljiva `a`*? U kom segmentu memorije je ono na šta pokazuje promenljiva* `a`*? U kom segmentu memorije je ono na šta pokazuje promenljiva* `b`*?*

Pitanje 10.6.14. *Koliko bajtova će na steku biti rezervisano za niz* `char *s[] = { "jedan", "dva", "tri" };` *koji je lokalna promenljiva?*

10.7 Pokazivači i strukture

Moguće je definisati i pokazivače na strukture. U slučaju da se članovima strukture pristupa preko pokazivača, umesto kombinacije operatora `*` i `.`, moguće je koristiti operator `->`. I operator `->` je operator najvišeg prioriteta. Na primer, umesto `(*pa).imenilac` može se pisati `pa->imenilac`:

```
struct razlomak *pa = &a;
printf("%d/%d", pa->brojilac, pa->imenilac);
```

Ukoliko se vrednost pokazivača `p` dodeli pokazivaču `q`, neće na adresu na koju ukazuje `q` biti iskopiran sadržaj na koji ukazuje `p`, nego će samo promenljiva `q` dobiti vrednost promenljive `p` i obe će ukazivati na prostor na koji je pre toga ukazivao pokazivač `p`. Ovo jednostavno i opšte pravilo važi i prilikom dodele između promenljivih tipa neke strukture. Neka su `a` i `b` promenljive istog

tipa strukture koja ima član pokazivačkog tipa `p`. Dodelom `a=b;` biće iskopirane vrednosti svih članova strukture iz `b` u `a`, uključujući i vrednost člana `p`, ali neće biti iskopiran sadržaj na koji je ukazivao pokazivač `b.p`. Umesto toga, nakon `a=b;` i `a.p` i `b.p` ukazivaće na istu adresu (na onu na koju je ukazivao `b.p`.

Strukture se kao argumenti u funkciju prenose po vrednosti. S obzirom na to da strukture često zauzimaju više prostora od elementarnih tipova podataka, čest je običaj da se umesto struktura u funkcije proslede njihove adrese, tj. pokazivači na strukture. Na primer,

```
void saberi_razlomke(struct razlomak *pa, struct razlomak *pb,
                     struct razlomak *pc)
{
    pc->brojilac = pa->brojilac*pb->imenilac +
                   pa->imenilac*pb->brojilac;
    pc->imenilac = pa->imenilac*pb->imenilac;
}
```

10.8 Pokazivači na funkcije

Funkcije se ne mogu direktno prosleđivati kao argumenti drugim funkcijama, vraćati kao rezultat funkcija i ne mogu se dodeljivati promenljivima. Ipak, ove operacije je moguće posredno izvršiti ukoliko se koriste pokazivači na funkcije. Razmotrimo nekoliko ilustrativnih primera.

Program 10.3.

```
#include <stdio.h>

void inc1(int a[], int n, int b[]) {
  int i;
  for (i = 0; i < n; i++)
     b[i] = a[i] + 1;
}

void mul2(int a[], int n, int b[]) {
  int i;
  for (i = 0; i < n; i++)
     b[i] = a[i] * 2;
}

void parni0(int a[], int n, int b[]) {
  int i;
  for (i = 0; i < n; i++)
     b[i] = a[i] % 2 == 0 ? 0 : a[i];
}

void ispisi(int a[], int n) {
  int i;
  for (i = 0; i < n; i++)
     printf("%d ", a[i]);
```

```
  putchar('\n');
}

#define N 8
int main() {
  int a[N] = {1, 2, 3, 4, 5, 6, 7, 8}, b[N];

  inc1(a, N, b);   ispisi(b, N);
  mul2(a, N, b);   ispisi(b, N);
  parni0(a, N, b); ispisi(b, N);

  return 0;
}
```

Sve funkcije u prethodnom programu kopiraju elemente niza `a` u niz `b`, prethodno ih transformišući na neki način. Moguće je izdvojiti ovaj zajednički postupak u zasebnu funkciju koja bi bila parametrizovana operacijom transformacije koja se primenjuje na elemente niza `a`:

```
#include <stdio.h>

void map(int a[], int n, int b[], int (*f) (int)) {
  int i;
  for (i = 0; i < n; i++)
     b[i] = (*f)(a[i]);
}

int inc1(int x) { return x + 1; }
int mul2(int x) { return 2 * x; }
int parni0(int x) { return x % 2 == 0 ? 0 : x; }

void ispisi(int a[], int n) {
  int i;
  for (i = 0; i < n; i++)
     printf("%d ", a[i]);
  putchar('\n');
}

#define N 8
int main() {
  int a[N] = {1, 2, 3, 4, 5, 6, 7, 8}, b[N];

  map(a, N, b, &inc1);   ispisi(b, N);
  map(a, N, b, &mul2);   ispisi(b, N);
  map(a, N, b, &parni0); ispisi(b, N);

  return 0;
}
```

Funkcija `map` ima poslednji argument tipa `int (*)(int)`, što označava pokazivač na funkciju koja prima jedan argument tipa `int` i vraća argument tipa

`int`.

Pokazivači na funkcije se razlikuju po tipu funkcije na koje ukazuju (po tipovima argumenata i tipu povratne vrednosti). Deklaracija promenljive tipa pokazivača na funkciju se vrši tako što se ime promenljive kojem prethodi karakter `*` navede u zagradama kojima prethodi tip povratne vrednosti funkcije, a za kojima sledi lista tipova parametara funkcije. Prisustvo zagrada je neophodno kako bi se napravila razlika između pokazivača na funkcije i samih funkcija. U primeru

```
double *a(double, int);
double (*b)(double, int);
```

promenljiva `a` označava funkciju koja vraća rezultat tipa `*double`, a prima argumente tipa `double` i `int`, dok promenljiva `b` označava pokazivač na funkciju koja vraća rezultat tipa `double`, a prima argumente tipa `double` i `int`.

Najčešće korišćene operacije sa pokazivačima na funkcije su, naravno, referenciranje (`&`) i dereferenciranje (`*`).[4]

Moguće je kreirati i nizove pokazivača na funkcije. Ovi nizovi se mogu i incijalizovati (na uobičajeni način). U primeru

```
int (*fje[3]) (int) = {&inc1, &mul2, &parni0};
```

`fje` predstavlja niz od 3 pokazivača na funkcije koje vraćaju `int`, i primaju argument tipa `int`. Funkcije čije se adrese nalaze u nizu se mogu direktno i pozvati. Na primer, naredni kôd ispisuje vrednost `4`:

```
printf("%d", (*fje[0])(3));
```

Pitanja i zadaci za vežbu

Pitanje 10.8.1. *Deklarisati funkciju* `f` *povratnog tipa* `int` *koja kao argument ima pokazivač na funkciju koja je povratnog tipa* `char` *i ima (samo jedan) argument tipa* `float`.

Pitanje 10.8.2. *Deklarisati funkciju* `f` *čiji poziv može da bude* `f(strcmp)`, *gde je* `strcmp` *funkcija deklarisana u* `string.h`.

Pitanje 10.8.3. *Ako se funkcija* `faktorijel` *sa prototipom* `int faktorijel(int n)` *može koristi kao (jedini) argument funkcije* `f` *tipa* `float`, *kako je deklarisana funkcija* `f`*?*

Pitanje 10.8.4. *Da li je ispravan sledeći prototip funkcije čiji je prvi od dva argumenta pokazivač na funkciju koja ima jedan argument:*

(a) `int f(int* g(char), int x);`
(b) `int f(void g(char), int x);`
(c) `int f(int (*g)(char), int x);`
(d) `int f(int (*g)(void), int x);`

[4] Iako kod nekih kompilatora oznake ovih operacija mogu da se izostave i da se koristi samo ime pokazivača (na primer, u prethodnom programu je bilo moguće navesti `map(a, N, b, inc1)` i `b[i] = f(a[i])`), ovo se ne preporučuje zbog prenosivosti programa.

Pitanje 10.8.5. *Da li je ispravan sledeći prototip funkcije:*
`int poredi(char *a, char *b, (*comp)(char *, char *));` *?*
Obrazložiti odgovor.

Pitanje 10.8.6. *Date su deklaracije:*

```
int *a1 (int* b1);
int (*a2) (int *b2);
```

Kog je tipa `a1`*, a kog* `a2`*?*

Pitanje 10.8.7. *Kog je tipa* `x` *deklarisano sa:*

1. `double (*x[3])(int);`
2. `int (*x) (double);`
3. `int *x (double);`
4. `double* (*x) (float*);`
5. `int (*f) (float*);`

Zadatak 10.2. *Napisati funkciju koja u nizu određuje najdužu seriju elemenata koji zadovoljavaju dato svojstvo. Svojstvo dostaviti kao parametar funkcije. Iskoristiti je da bi se našla najduža serija parnih kao i najduža serija pozitivnih elemenata niza.* √

10.9 Dinamička alokacija memorije

U većini realnih aplikacija, u trenutku pisanja programa nije moguće precizno predvideti memorijske zahteve programa. Naime, memorijski zahtevi zavise od interakcije sa korisnikom i tek u fazi izvršavanja programa korisnik svojim akcijama implicitno određuje potrebne memorijske zahteve (na primer, koliko elemenata nekog niza će biti korišćeno). U nekim slučajevima, moguće je predvideti neko gornje ograničenje, ali ni to nije uvek zadovoljavajuće. Ukoliko je ograničenje premalo, program nije u stanju da obrađuje veće ulaze, a ukoliko je preveliko, program zauzima više memorije nego što mu je stvarno potrebno. Rešenje ovih problema je *dinamička alokacija memorije* koja omogućava da program u toku svog rada, u fazi izvršavanja, zahteva (od operativnog sistema) određenu količinu memorije. U trenutku kada mu memorija koja je dinamički alocirana više nije potrebna, program može i dužan je da je oslobodi i tako je vrati operativnom sistemu na upravljanje. Alociranje i oslobađanje vrši se funkcijama iz standardne biblioteke i pozivima rantajm biblioteke. U fazi izvršavanja, vodi se evidencija i o raspoloživim i zauzetim blokovima memorije.

10.9.1 Funkcije standardne biblioteke za rad sa dinamičkom memorijom

Standardna biblioteka jezika C podržava dinamičko upravljanje memorijom kroz nekoliko funkcija (sve su deklarisane u zaglavlju `<stdlib.h>`). Prostor za dinamički alociranu memoriju nalazi se u segmentu memorije koji se zove *hip* (engl. heap).

Funkcija `malloc` ima sledeći prototip:

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

Ona alocira blok memorije (tj. niz uzastopnih bajtova) veličine `n` bajtova i vraća adresu alociranog bloka u vidu generičkog pokazivača (tipa `void*`). U slučaju da zahtev za memorijom nije moguće ispuniti (na primer, zahteva se više memorije nego što je na raspolaganju), ova funkcija vraća `NULL`. Memorija na koju funkcija `malloc` vrati pokazivač nije inicijalizovana i njen sadržaj je, u principu, nedefinisan (tj. zavisi od podataka koji su ranije bili čuvani u tom delu memorije).

Funkcija `malloc` očekuje argument tipa `size_t`. Ovo je nenegativni celobrojni tip za čije vrednosti je rezervisano najmanje dva bajta. Ovaj tip se razlikuje od tipa `unsigned int`, ali su međusobne konverzije moguće i, zapravo, trivijalne. Tip `size_t` može da se koristi za čuvanje bilo kog indeksa niza, a on je i tip povratne vrednosti operatora `sizeof`.

Funkcija `calloc` ima sledeći prototip:

```
void *calloc(size_t n, size_t size)
```

Ona vraća pokazivač na blok memorije veličine `n` objekata navedene veličine `size`. U slučaju za zahtev nije moguće ispuniti, vraća se `NULL`. Za razliku od `malloc`, alocirana memorija je inicijalizovana na nulu.

Dinamički objekti alocirani navedenim funkcijama su *neimenovani* i bitno su različiti od promenljivih. Ipak, dinamički alociranim blokovima se pristupa na sličan način kao nizovima.

Navedene funkcije su generičke i koriste se za dinamičku alokaciju memorije za podatke bilo kog tipa. Kako bi se dobijenoj memoriji moglo pristupati slično kao u slučaju nizova, potrebno je (poželjno eksplicitno) konvertovati dobijeni pokazivač tipa `void*` u neki konkretni pokazivački tip.

Nakon poziva funkcije `malloc()` ili `calloc()` obavezno treba proveriti povratnu vrednost kako bi se utvrdilo da li je alokacija uspela. Ukoliko alokacija ne uspe, pokušaj pristupa memoriji na koju ukazuje dobijeni pokazivač dovodi do dereferenciranja `NULL` pokazivača i greške. Ukoliko se utvrdi da je funkcija `malloc()` (ili `calloc()`) vratila vrednost `NULL`, može se prijaviti korisniku odgovarajuća poruka ili pokušati neki metod oporavka od greške. Dakle, najčešći scenario upotrebe funkcije `malloc` sledeći:

```
int* p = (int*) malloc(n*sizeof(int));
if (p == NULL)
  /* prijaviti gresku */
```

U navedenom primeru, nakon uspešne alokacije, u nastavku programa se `p` može koristiti kao (statički alociran) niz celih brojeva.

U trenutku kada dinamički alociran blok memorije više nije potreban, poželjno je osloboditi ga. To se postiže funkcijom `free`:

```
void free(void* p);
```

Poziv `free(p)` oslobađa memoriju na koju ukazuje pokazivač `p` (a ne memorijski prostor koji sadrži sâm pokazivač `p`), pri čemu je neophodno da `p` pokazuje na blok memorije koji je alociran pozivom funkcije `malloc` ili `calloc`. Opasna je greška pokušaj oslobađanja memorije koja nije alocirana na ovaj način. Takođe, ne sme se koristiti nešto što je već oslobođeno niti se sme dva puta

oslobađati ista memorija. Redosled oslobađanja memorije ne mora da odgovara redosledu alociranja.

Ukoliko neki dinamički alociran blok nije oslobođen ranije, on će biti oslobođen prilikom završetka rada programa, zajedno sa svom drugom memorijom koja je dodeljena programu. Ipak, ne treba se oslanjati na to i preporučeno je eksplicitno oslobađanje sve dinamički alocirane memorije pre kraja rada programa, a poželjno čim taj prostor nije potreban.

Upotrebu ovih funkcija ilustruje naredni primer u kojem se unosi i obrnuto ispisuje niz čiji broj elemenata nije unapred poznat (niti je poznato njegovo gornje ograničenje), već se unosi sa ulaza tokom izvršavanja programa.

Program 10.4.

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

int main() {
  int n, i, *a;

  /* Unos broja elemenata */
  scanf("%d", &n);
  /* Alocira se memorija */
  if ((a = (int*)malloc(n*sizeof(int))) == NULL) {
    printf("Greska prilikom alokacije memorije\n");
    return 1;
  }
  /* Unos elemenata */
  for (i = 0; i < n; i++) scanf("%d",&a[i]);
  /* Ispis elemenata u obrnutom poretku */
  for (i = n-1; i >= 0; i--) printf("%d ",a[i]);
  /* Oslobadjanje memorije */
  free(a);

  return 0;
}
```

U nekim slučajevima potrebno je promeniti veličinu već alociranog bloka memorije. To se postiže korišćenjem funkcije `realloc`, čiji je prototip:

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

Parametar `memblock` je pokazivač na prethodno alocirani blok memorije, a parametar `size` je nova veličina u bajtovima. Funkcija `realloc` vraća pokazivač tipa `void*` na realociran blok memorije, a `NULL` u slučaju da zahtev ne može biti ispunjen. Zahtev za smanjivanje veličine alociranog bloka memorije uvek uspeva. U slučaju da se zahteva povećanje veličine alociranog bloka memorije, pri čemu iza postojećeg bloka postoji dovoljno slobodnog prostora, taj prostor se jednostavno koristi za proširivanje. Međutim, ukoliko iza postojećeg bloka ne postoji dovoljno slobodnog prostora, onda se u memoriji traži drugo mesto dovoljno da prihvati prošireni blok i, ako se nađe, sadržaj postojećeg bloka se kopira na to novo mesto i zatim se stari blok memorije oslobađa. Ova operacija može biti vremenski zahtevna.

Upotreba funkcije `realloc` ilustrovana je programom koji učitava cele brojeve i smešta ih u memoriju, sve dok se ne unese `-1` za kraj. S obzirom na to da se broj elemenata ne zna unapred, a ne zna se ni gornje ograničenje, neophodno je postepeno povećavati skladišni prostor tokom rada programa. Kako česta realokacija može biti neefikasna, u narednom programu se izbegava realokacija prilikom unosa svakog sledećeg elementa, već se vrši nakon unošenja svakog desetog elementa. Naravno, ni ovo nije optimalna strategija — u praksi se obično koristi pristup da se na početku realokacije vrše relativno često, a onda sve ređe i ređe (na primer, svaki put se veličina niza dvostruko uveća).

Program 10.5.

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

#define KORAK 256
int main() {
    int* a = NULL;     /* Niz je u pocetku prazan */
    int duzina = 0;    /* broj popunjenih elemenata niza */
    int alocirano = 0; /* broj elemenata koji mogu biti smesteni */
    int i;

    do {
        printf("Unesi ceo broj (-1 za kraj): ");
        scanf("%d", &i);

        /* Ako nema vise slobodnih mesta, vrsi se prosirivanje */
        if (duzina == alocirano) {
            alocirano += KORAK;
            a = realloc(a, alocirano*sizeof(int));
            if (a == NULL) return 1;
        }
        a[duzina++] = i;
    } while (i != -1);

    /* Ispis elemenata */
    printf("Uneto je %d brojeva. Alocirano je ukupno %d bajtova\n",
        duzina, alocirano*sizeof(int));
    printf("Brojevi su : ");
    for (i = 0; i<duzina; i++)
        printf("%d ", a[i]);

    /* Oslobadjanje memorije */
    free(a);

    return 0;
}
```

Bez upotrebe funkcije `realloc` centralni blok navedene funkcije `main` bi mogao da izgleda ovako:

```
if (duzina == alocirano) {
   /* Kreira se novi niz */
   int* new_a;
   alocirano += KORAK;
   new_a = malloc(alocirano*sizeof(int));
   /* Kopira se sadrzaj starog niza u novi */
   for (i = 0; i < duzina; i++) new_a[i] = a[i];
   /* Oslobadja se stari niz */
   free(a);
   /* a ukazuje na novi niz */
   a = new_a;
}
```

U ovoj implementaciji, prilikom svake realokacije vrši se premeštanje memorije, tako da je ona neefikasnija od verzije sa `realloc`.

U gore navedenom primeru koristi se konstrukcija:

```
a = realloc(a, alocirano*sizeof(int));
```

U nekim slučajevima ova konstrukcija može da bude neadekvatna ili opasna. Naime, ukoliko zahtev za proširenje memorijskog bloka ne uspe, vraća se vrednost `NULL`, upisuje u promenljivu `a` i tako gubi jedina veza sa prethodno alociranim blokom (i on, na primer, ne može da bude oslobođen).

10.9.2 Greške u radu sa dinamičkom memorijom

Dinamički alocirana memorija nudi mnoge mogućnosti i dobra rešenja ali često je i uzrok problema. Neki od najčešćih problema opisani su u nastavku.

Curenje memorije. Jedna od najopasnijih grešaka u radu sa dinamički alociranom memorijom je tzv. *curenje memorije (engl. memory leak)*. Curenje memorije je situacija kada se u tekućem stanju programa izgubi informacija o lokaciji dinamički alociranog, a neoslobođenog bloka memorije. U tom slučaju, program više nema mogućnost da oslobodi taj blok memorije i on biva „zauvek“ (zapravo — do kraja izvršavanja programa) izgubljen (rezervisan za korišćenje od strane programa koji više nema načina da mu pristupi). Curenje memorije ilustruje naredni primer.

```
char* p;
p = (char*) malloc(1000);
....
p = (char*) malloc(5000);
```

Inicijalno je 1000 bajtova dinamički alocirano i adresa početka ovog bloka memorije smeštena je u pokazivačku promenljivu `p`. Kasnije je dinamički alocirano 5000 bajtova i adresa početka tog bloka memorije je opet smeštena u promenljivu `p`. Međutim, pošto originalnih 1000 bajtova nije oslobođeno korišćenjem funkcije `free`, a adresa početka tog bloka memorije je izgubljena promenom vrednosti pokazivačke promenljive `p`, tih 1000 bajtova biva nepovratno izgubljeno za program.

Naročito su opasna curenja memorije koja se događaju u okviru neke petlje. U takvim situacijama može da se gubi malo po malo memorije, ali tokom dugtrajnog izvršavanja programa, pa ukupna količina izgubljene memorije može

da bude veoma velika. Moguće je da u nekom trenutku program iscrpi svu raspoloživu memoriju i onda će njegovo izvršavanje biti prekinuto od strane operativnog sistema. Čak i da se to ne desi, moguće je da se iscrpi raspoloživi prostor u glavnoj memoriji i da se, zbog toga, sadržaj glavne memorije prebacuje na disk i obratno (tzv. swapping), što onda ekstremno usporava rad programa.

Curenje memorije je naročito opasno zbog toga što često ne biva odmah uočeno. Obično se tokom razvoja program testira kratkotrajno i na malim ulazima. Međutim, kada se program pusti u rad i kada počne da radi duži vremenski period (možda i bez prestanka) i da obrađuje veće količine ulaza, curenje memorije postaje vidljivo, čini program neupotrebljivim i može da uzrokuje velike štete.

Većina programa za otkrivanje grešaka (debagera) detektuje da u programu postoji curenje memorije, ali ne može da pomogne u lociranju odgovarajuće greške u kôdu. Postoje specijalizovani programi *profajleri za curenje memorije* (engl. memory leaks profilers) koji olakšavaju otkrivanje uzroka curenja memorije.

Pristup oslobođenoj memoriji. Nakon poziva `free(p)`, memorija na koju pokazuje pokazivač `p` se oslobađa i ona više ne bi trebalo da se koristi. Međutim, poziv `free(p)` ne menja sadržaj pokazivača `p`. Moguće je da naredni poziv funkcije `malloc` vrati blok memorije upravo na toj poziciji. Naravno, ovo ne mora da se desi i nije predvidljivo u kom će se trenutku desiti, tako da ukoliko programer nastavi da koristi memoriju na adresi `p`, moguće je da će greška proći neopaženo. Zbog toga, preporučuje se da se nakon poziva `free(p)`, odmah `p` postavi na `NULL`. Tako se osigurava da će svaki pokušaj pristupa oslobođenoj memoriji biti odmah prepoznat tokom izvršavanja programa i operativni sistem će zaustaviti izvršavanje programa sa porukom o grešci (najčešće `segmentation fault`).

Oslobađanje istog bloka više puta . Nakon poziva `free(p)`, svaki naredni poziv `free(p)` za istu vrednost pokazivača `p` prouzrokuje nedefinisano ponašanje programa i trebalo bi ga izbegavati. Takozvana *višestruka oslobađanja* mogu da dovedu do pada programa a poznato je da mogu da budu i izvor bezbednosnih problema.

Oslobađanje neispravnog pokazivača. Funkciji `free(p)` dopušteno je proslediti isključivo adresu vraćenu od strane funkcije `malloc`, `calloc` ili `realloc`. Čak i prosleđivanje pokazivača na lokaciju koja pripada alociranom bloku (a nije njegov početak) uzrokuje probleme. Na primer,

```
free(p+10); /* Oslobodi sve osim prvih 10 elemenata bloka */
```

neće osloboditi „sve osim prvih 10 elemenata bloka“ i sasvim je moguće da će dovesti do neprijatnih posledica, pa čak i do pada programa.

Prekoračenja i potkoračenja bafera. Nakon dinamičke alokacije, pristup memoriji je dozvoljen samo u okviru granica bloka koji je dobijen. Kao i u slučaju statički alociranih nizova, pristup elementima van granice može da

prouzrokuje ozbiljne probleme u radu programa. Upis van granica bloka najčešće je opasniji od čitanja. U slučaju dinamički alociranih blokova memorije, obično se nakon samog bloka smeštaju dodatne informacije potrebne alokatoru memorije kako bi uspešno vodio evidenciju koji delovi memorije su zauzeti, a koji slobodni. Zato, i malo prekoračenje granice bloka prilikom upisa može da promeni te dodatne informacije i da uzrokuje pad sistema za dinamičko upravljanje memorijom. Postoje i mnogi drugi slučajevi u kojima prekoračenje i potkoračenje bafera mogu da naruše ispravno izvršavanje programa ili dovedu do njegovog pada.

10.9.3 Fragmentisanje memorije

Čest je slučaj da ispravne aplikacije u kojima ne postoji curenje memorije (a koje često vrše dinamičku alokaciju i dealokacije memorije) tokom dugog rada pokazuju degradaciju u performansama i na kraju prekidaju svoj rad na nepredviđeni način. Uzrok ovome je najčešće *fragmentisanje memorije*. U slučaju fragmentisane memorije, u memoriji se često nalazi dosta slobodnog prostora, ali on je rascepkan na male, nepovezane parčiće. Razmotrimo naredni (minijaturizovan) primer. Ukoliko 0 označava slobodni bajt, a 1 zauzet, a memorija trenutno ima sadržaj `100101011000011101010110`, postoji ukupno 12 slobodnih bajtova. Međutim, pokušaj alokacije 5 bajtova ne može da uspe, jer u memoriji ne postoji prostor dovoljan za smeštanje 5 povezanih bajtova. S druge strane, memorija koja ima sadržaj `111111111111111100000000` ima samo 8 slobodnih bajtova, ali jeste u stanju da izvrši alokaciju 5 traženih bajtova. Memoriju na hipu dodeljenu programu nije moguće automatski reorganizovati u fazi izvršavanja, jer nije moguće u toj fazi ažurirati sve pokazivače koji ukazuju na objekte na hipu.

Postoji nekoliko pristupa za izbegavanje fragmentisanja memorije. Ukoliko je moguće, poželjno je izbegavati dinamičku alokaciju memorije. Naime, alociranje svih struktura podataka statički (u slučajevima kada je to moguće) dovodi do bržeg i predvidljivijeg rada programa (po cenu većeg utroška memorije). Alternativno, ukoliko se ipak koristi dinamička alokacija memorije, poželjno je memoriju alocirati u većim blokovima i umesto alokacije jednog objekta alocirati prostor za nekoliko njih odjednom (kao što je to, na primer, bilo rađeno u primeru datom prilikom opisa funkcije `realloc`). Na kraju, postoje tehnike efikasnijeg rukovanja memorijom (na primer, memory pooling), u kojima se programer manje oslanja na sistemsko rešenje, a više na svoje rešenje koje je prilagođenjeno specifičnim potrebama.

10.9.4 Hip i dinamički životni vek

U poglavlju 9.3.3, rečeno je da je memorija dodeljena programu organizovana u segment kôda, segment podataka, stek segment i hip segment. Hip segment predstavlja tzv. slobodnu memoriju iz koje se crpi memorija koja se dinamički alocira. Dakle, funkcije `malloc`, `calloc` ili `realloc`, ukoliko uspeju, vraćaju adresu u hip segmentu. Objekti koji su alocirani u slobodnom memorijskom prostoru nisu imenovani, već im se pristupa isključivo preko adresa. Pošto ovi objekti nisu imenovani, oni nemaju definisan doseg (a doseg pokazivača putem kojih se pristupa dinamičkim objektima podleže standardnim

pravilima). Svi objekti koji su dinamički alocirani imaju *dinamički životni vek*. Ovo znači da se memorija i alocira i oslobađa isključivo na eksplicitni zahtev i to tokom rada programa.

Hip segment obično počinje neposredno nakon segmenta podataka, a na suprotnom kraju memorije od stek segmenta. Obično se podaci na hipu slažu od manjih ka većim adresama (engl. upward growing), dok se na steku slažu od većih ka manjim adresama (eng;. downward growing). Ovo znači da se u trenutku kada se iscrpi memorijski prostor dodeljen programu, hip i stek potencijalno „sudaraju“, ali operativni sistemi obično to sprečavaju i tada obično dolazi do nasilnog prekida rada programa.

Pitanja i zadaci za vežbu

Pitanje 10.9.1.

Navesti prototip, opisati njeno ponašanje i primer korišćenja funkcije:

- `malloc`;
- `calloc`;
- `realloc`;
- `free`.

U kom zaglavlju su deklarisane ove funkcije?

Pitanje 10.9.2. *Fukcija* `realloc` *ima isti efekat kao funkcija* `free` *ako je:*

- *vrednost njenog prvog argumenta nula;*
- *vrednost njenog drugog argumenta nula;*
- *vrednost oba njena argumenta nula;*
- *vrednost oba njena argumenta različita od nule.*

Pitanje 10.9.3. *Kada je, nakon komandi*

```
char *p;
p = (char*)malloc(n);
```

komanda `strcpy(p,"test");` *bezbedna?*

Pitanje 10.9.4. *U kom segmentu memorije se tokom izvršavanja programa čuvaju dinamički alocirani blokovi memorije.*

Pitanje 10.9.5. *Kako se zove pojava kada se izgubi vrednost poslednjeg pokazivača na neki dinamički alocirani blok memorije? Zašto je ova pojava opasna?*

Pitanje 10.9.6. *Da li će doći do curenja memorije ako nakon komande*
`p = (int*)malloc(sizeof(int)*5)`
slede komanda/komande:

- `q = (int*)malloc(sizeof(int)*5);`

- `p = (int*)malloc(sizeof(int)*5);`
- `free(p); free(p);`
- `free(p+1);`

Pitanje 10.9.7. *Zaokružiti komandu u narednom kôdu koja dovodi do greške?*

```
int *f = malloc(4*sizeof(int));
int* g = f;
free(g);
free(f);
f=g;
```

Pitanje 10.9.8. *Koje su prednosti korišćenja dinamičke alokacije memorije (u odnosu na statičku i automatsku alokaciju)? Koji su nedostaci?*

Pitanje 10.9.9. *Šta je, nakon poziva* `free(p)`*,vrednost pokazivača* `p`*, a šta vrednost* `*p`*?*

Pitanje 10.9.10. *Koja naredba treba da se, prema dobroj praksi, izvrši nakon* `free(p);`*? Šta se time postiže?*

Pitanje 10.9.11. *Kako sistem upravlja dinamičkom memorijom? Zašto se ne sme dinamički alociran prostor osloboditi dva puta?*

Pitanje 10.9.12. *Opisati ukratko bar četiri česte greške koje se javljaju u vezi sa dinamičkom alokacijom memorije.*

Pitanje 10.9.13. *Da li je u nekoj situaciji dozvoljeno funkciji* `free` *proslediti pokazivač koji nije dobijen funkcijama* `malloc`*,* `calloc` *i* `realloc`*?*

Pitanje 10.9.14. *Kako se zove situacija u kojoj je memorija na hipu podeljena na mnoštvo malih i nepovezanih blokova delova?*

Pitanje 10.9.15. *Da li se fragmentisanje memorije može umanjiti ako se?*

- *ne koristi rekurzija;*
- *ne koristi memorija na hipu;*
- *koristi što manje lokalnih promenljivih;*
- *koristi što manje globalnih promenljivih.*

Zadatak 10.9.1. *Napisati program koji sa standardnog ulaza učitava broj n, a zatim i niz a od n celih brojeva, pa zatim i ceo broj x sa standardnog ulaza. Program treba da vrati indeks broja x u nizu a (ako se x nalazi u a) ili indeks elementa niza a koji je po apsolutnoj vrednosti najbliži broju x.* √

Zadatak 10.9.2. *Napisati program koji sa standardnog ulaza učitava cele brojeve dok ne učita nulu kao oznaku za kraj. Na standardni izlaz ispisati koji broj se pojavio najviše puta među tim brojevima. Na primer, ako se na ulazu pojave brojevi:* `2 5 12 4 5 2 3 12 15 5 6 6 5` *program treba da vrati broj* 5*. Zadatak rešiti korišćenjem niza koji se dinamički realokacira.* √

Zadatak 10.9.3. *Napisati program koji sa standardnog ulaza učitava prvo dimenzije matrice (n i m) a zatim redom i elemente matrice (ne postoje pretpostavke o dimenziji matrice), a zatim ispisuje matricu spiralno, krenuvši od gornjeg levog ugla.* √

Zadatak 10.9.4.

1. *Napisati funkciju* `int skalarni_proizvod(int* a,int* b, int n)` *koja računa skalarni proizvod vektora a i b dužine n. (Sklarni proizvod dva vektora* $a = (a_1, \ldots, a_n)$ *i* $b = (b_1, \ldots, b_n)$ *je suma* $S = a_1 \cdot b_1 + a_2 \cdot b_2 + \ldots + a_n \cdot b_n$*)*

2. *Napisati funkciju* `int ortonormirana(int** A, int n)` *kojom se proverava da li je zadata kvadratna matrica A dimenzije* $n \times n$ *ortonormirana. Za matricu ćemo reći da je ortonormirana ako je skalarni proizvod svakog para različitih vrsta jednak 0, a skalarni proizvod vrste sa samom sobom 1. Funkcija vraća 1 ukoliko je matrica ortonormirana, 0 u suprotnom.*

3. *Napisati glavni program u kome učitava dimenzija kvadratne matrice* `n`*, a zatim i elementi i pozivom funkcije se utvrđuje da li je matrica ortonormirana. Maksimalna dimenzija matrice nije unapred poznata.*

√

Glava 11

Pregled standardne biblioteke

Jezik C ima mali broj naredbi i mnoge funkcionalnosti obezbeđene su kroz funkcije *standardne biblioteke* (engl. standard library). Standardna biblioteka obezbeđuje funkcije, ali i makroe i definicije tipova za potrebe rada sa datotekama, niskama, alokacijom memorije i slično. Standardna C biblioteka nije jedna, konkretna biblioteka koja se koristi za povezivanje sa programima napisanom u jeziku C, već ona pre predstavlja standard koji mora da poštuje svaki C prevodilac. Kako je implementirana standardna biblioteka zavisi od konkretnog sistema.

Programski interfejs za pisanje aplikacija (API) za standardnu biblioteku dat je u vidu datoteka zaglavlja, od kojih svaka sadrži deklaracije funkcije, definicije tipova i makroa. Skupu petnaest datoteka zaglavlja propisanih standardom ANSI C (C89), dokument *Normative Addendum 1* (NA1) dodao je tri datoteke 1995. godine. Verzija C99 uvela je još šest datoteka zaglavlja, a verzija C11 još pet (od kojih tri mogu da ne budu podržane od implementacija, a da se te implementacije i dalje smatraju standardnim). U tabeli 11.1 dat je pregled svih dvadeset devet zaglavlja. Ove datoteke, zajedno sa izvornim kôdom programa bivaju uključene u proces kompilacije (korišćenjem pretprocesora i njegove direktive `#include`). Pored ovih datoteka, standardna biblioteka sadrži i definicije funkcija koje obično nisu raspoložive u izvornom, već samo u prevedenom, objektnom (mašinskom) obliku, spremnom za povezivanje sa objektnim modulima programa korisnika. Ovaj (veći) deo standardne biblioteke biva uključen u fazi povezivanja (i ne obrađuje se tokom kompilacije).

U daljem tekstu će biti ukratko opisane neke od funkcija deklarisanih u nekim od ovih zaglavlja. Najkompleksnije zaglavlje `stdio.h` biće opisano u narednoj glavi, posvećenoj ulazu i izlazu C programa.

11.1 Zaglavlje `string.h`

```
size_t strlen  (char* str);

char*  strcpy  (char* destination, char* source);
char*  strncpy (char* destination, char* source, size_t num);
char*  strcat  (char* destination, char* source);
char*  strncat (char* destination, char* source, size_t num);
```

Datoteka zaglavlja	Od	Deklaracije i definicije koje sadrži
`<assert.h>`	C89	Makro `assert`.
`<complex.h>`	C99	Funkcije za rad sa kompleksnim brojevima.
`<ctype.h>`	C89	Funkcije za klasifikovanje i konvertovanje karaktera, nezavisno od karakterske tabele koja se koristi (ASCII ili neke druge);
`<errno.h>`	C89	Podrška za rad sa kôdovima grešaka koje vraćaju bibliotečke funkcije.
`<fenv.h>`	C99	Funkcije za kontrolu okruženja za brojeve u pokretnom zarezu.
`<float.h>`	C89	Konstante koje specifikuju svojstva brojeva u pokretnom zarezu zavisna od implementacije.
`<inttypes.h>`	C99	Proširena podrška za celobrojne tipovi fiksne širine.
`<iso646.h>`	NA1	Makroi za opisivanje standardnih tokena.
`<limits.h>`	C89	Konstante koje specifikuju svojstva celih brojeva zavisna od implementacije.
`<locale.h>`	C89	Funkcije za lokalizaciju.
`<math.h>`	C89	Često korišće matematičke funkcije.
`<setjmp.h>`	C89	Makroi `setjmp` i `longjmp`.
`<signal.h>`	C89	Funkcije za upravljanje signalima.
`<stdalign.h>`	C11	Podrška sa ravnanje objekata.
`<stdarg.h>`	C89	Podrška za funkcije sa promenljivim brojem parametara
`<stdatomic.h>`	C11	Podrška za atomičke operacije nad podacima deljenim između niti.
`<stdbool.h>`	C99	Tip `bool`.
`<stddef.h>`	C89	Nekoliko raznorodnih tipova i konstanti (uključujući NULL i `size_t`).
`<stdint.h>`	C99	Podrška za celobrojne tipove fiksne širine.
`<stdio.h>`	C89	Deklaracije osnovnih ulazno/izlaznih funkcija;
`<stdlib.h>`	C89	Deklaracije funkcija za konverzije, za generisanje pseudoslučajnih brojeva, za dinamičku alokaciju memorije, prekid programa itd.
`<stdnoreturn.h>`	C11	Podrška za specifikovanje funkcija bez povratka
`<string.h>`	C89	Funkcije za obradu niski.
`<tgmath.h>`	C99	Matematičke funkcije generičkog tipa.
`<threads.h>`	C11	Funkcije za upravljanje nitima.
`<time.h>`	C89	Funkcije za rad sa datumima i vremenom
`<uchar.h>`	C11	Tipovi i deklaracije za rad sa Unicode karakterima.
`<wchar.h>`	NA1	Funkcije za rad sa niskama karaktera veće širine.
`<wctype.h>`	NA1	Funkcije za klasifikovanje i konvertovanje karaktera veće širine.

Tabela 11.1: Pregled zaglavlja standardne biblioteke

```
int    strcmp  (char* str1, char* str2);
int    strncmp (char* str1, char* str2, size_t num);

char*  strchr  (char* str, int character);
char*  strrchr (char* str, int character);
char*  strstr  (char* str1, char * str2);

size_t strspn  (char* str1, char* str2);
size_t strcspn (char* str1, char* str2);
char*  strpbrk (char* str1, char* str2);

char*  strtok  (char * str, char* delimiters);
```

`strlen` Funkcija `strlen` izračunava dužinu prosleđene niske karaktera (terminalna nula se ne računa).

`strcpy` Funkcija `strcpy` kopira drugu navedenu nisku u prvu (pretpostavljajući da u prvoj ima dovoljno mesta za sve karaktere druge, uključujući i terminalnu nulu). Funkcija `strncpy` takođe vrši kopiranje, ali se dodatnim parameterom `n` kontroliše najveći broj karaktera koji može biti kopiran (u slučaju da je niska koja se kopira kraća od broja `n` ostatak se popunjava nulama, a ukoliko je duža od broja `n` terminalna nula se ne dodaje automatski na kraj). Korišćenje funkcije `strncpy` smatra se bezbednijim od `strcpy` jer je moguće kontrolisati mogućnost prekoračenja bafera.

`strcat` Funkcija `strcat` nadovezuje drugu navedenu nisku na prvu, pretpostavljajući da prva niska ima dovoljno prostora da smesti i sve karaktere druge, uključujući i njenu terminalnu nulu (terminalna nula prve niske se briše). Funkcija `strncat` dodatnim parametrom `n` kontroliše najveći broj druge niske koji će biti nadovezan na prvu.

`strchr` Funkcija `strchr` proverava da li data niska sadrži dati karakter i vraća pokazivač na prvu poziciju na kojoj je karakter nađen ili `NULL` ako karakter nije nađen. Funkcija `strrchr` radi slično, ali vraća pokazivač na poslednje pojavljivanje karaktera.

`strpbrk` Funkcija `strpbrk` vraća pokazivač na prvo pojavljivanje nekog karaktera iz druge navedene niske u prvoj navedenoj niski ili `NULL` pokazivač ako takvog karaktera nema.

`strstr` Funkcija `strstr` proverava da li je druga navedena niska podniska prve. Ako jeste, vraća se pokazivač na prvo njeno pojavljivanje unutar prve niske, a ako nije, vraća se pokazivač `NULL`.

`strspn` Funkcija `strspn` vraća dužinu početnog dela prve navedene niske koji se sastoji samo od karaktera sadržanih u drugoj navedenoj niski. Slično, funkcija `strcspn` vraća dužinu početnog dela prve navedene niske koji se sastoji samo od karaktera koji nisu sadržani u drugoj navedenoj niski.

`strtok` Niz poziva funkcije `strtok` služe da podele nisku u tokene koji su niske uzastopnih karaktera razdeljene karakterima navedenim u drugoj niski.

U prvom pozivu funkcije kao prvi argument navodi se niska koja se deli, a u drugom navodi se `NULL`. Na primer, kôd

```
char str[] ="- Ovo, je jedna niska.", delims[] = " ,.-";
char *s = strtok (str, delims);
while (s != NULL) {
  printf ("%s\n", s);
  s = strtok (NULL, delims);
}
```

ispisuje

```
Ovo
je
jedna
niska
```

11.2 Zaglavlje `stdlib.h`

```
void *malloc(size_t n);
void *calloc(size_t n, size_t size);
void *realloc(void *memblock, size_t size);
void free(void* p);

int rand(void);
void srand(unsigned int);

int system(char* command);
void exit(int);
```

Već smo naveli da se u zaglavlju `stdlib.h` nalaze deklaracije funkcija `malloc`, `calloc`, `realloc` i `free` koje služe za dinamičku alokaciju i oslobađanje memorije. Pored njih najznačajnije funkcije ovog zaglavlja su i sledeće.

`rand` Funkcija `int rand(void);` generiše pseudo-slučajne cele brojeve u intervalu od 0 do vrednosti `RAND_MAX` (koja je takođe definisana u zaglavlju `<stdlib.h>`). Termin *pseudo-slučajni* se koristi da se naglasi da ovako dobijeni brojevi nisu zaista slučajni, već su dobijeni specifičnim izračunavanjima koja proizvode nizove brojeva nalik na slučajne. Funkcija `rand()` vraća sledeći pseudo-slučajan broj u svom nizu. Pseudo-slučajni brojevi brojevi u pokretnom zarezu iz intervala $[0, 1)$ mogu se dobiti na sledeći način:

```
((double) rand() / (RAND_MAX+1.0))
```

Funkcija `rand()` može biti jednostavno upotrebljena za generisanje pseudo-slučajnih brojeva u celobrojnom intervalu $[n, m]$:

```
n+rand()%(m-n+1)
```

Ipak, bolja svojstva (bolju raspodelu) imaju brojevi koji se dobijaju na sledeći način:

```
n+(m-n+1)*((double) rand() / (RAND_MAX+1.0))
```

`srand` Funkcija `rand` niz pseudo-slučajnih brojeva generiše uvek počev od iste podrazumevane vrednosti (koja je jednaka 1). Funkcija `void srand(unsigned);` postavlja vrednost koja se u sledećem pozivu funkcije `rand` (a time, indirektno, i u svim narednim) koristi u generisanju tog niza. Za bilo koju vrednost funkcije `srand`, čitav niz brojeva koji vraća funkcija `rand` je uvek isti. To je pogodno za ponavljanje (na primer, radi debagovanja) procesa u kojima se koriste pseudo-slučajni brojevi.

`system` Prototip funkcije `system` je `int system(char* command);` Pozivom `system(komanda)`, izvršava se navedena komanda `komanda` (potencijalno sa argumentima) kao sistemski poziv kojim se izvršava komanda operativnog sistema ili neki drugi program (u okviru tekućeg programa, a ne iz komandne linije). Nakon toga, nastavlja se izvršavanje programa.

Na primer, pozivom `system("date");` aktivira se program `date` koji ispisuje tekući datum i koji je često prisutan na različitim operativnim sistemima.

Ukoliko je argument funkcije `system` vrednost `NULL`, onda se samo proverava da li je raspoloživ komandni interpretator koji bi mogao da izvršava zadate komande. U ovom slučaju, funkcija `system` vraća ne-nulu, ako je komandni interpretator raspoloživ i nulu inače.

Ukoliko argument funkcije `system` nije vrednost `NULL`, onda funkcija vraća istu vrednost koju je vratio komandni interpretator (obično nulu ukoliko je komanda izvršena bez grešaka). Vrednost -1 vraća se u slučaju greške (na primer, komandni interpretator nije rasploživ, nema dovoljno memorije da se izvrši komanda, lista argumenata nije ispravna).

`exit` Funkcija `void exit(int);` zaustavlja rad programa (bez obzira iz koje funkcije je pozvana) i vraća svoj argument kao povratnu vrednost programa. Obično povratna vrednost nula označava uspešno izvršenje programa, a vrednost ne-nula ukazuje na neku grešku tokom izvršavanja. Često se koriste i simboličke konstante `EXIT_SUCCESS` za uspeh i `EXIT_FAILURE` za neuspeh. Naredba `exit(e);` u okviru funkcije `main` ekvivalentna je naredbi `return e;`. Funkcija `exit` automatski poziva `fclose` za svaku datoteku otvorenu u okviru programa (više o funkciji `fclose` biće rečeno u glavi 12).

11.3 Zaglavlje `ctype.h`

```
int isalpha(int c); int isdigit(int c);
int isalnum(int c); int isspace(int c);
int isupper(int c); int islower(int c);
int toupper(int c); int tolower(int c);
```

Zaglavlje `ctype.h` sadrži deklaracije nekoliko funkcija za ispitivanje i konvertovanje karaktera. Sve funkcije navedene u nastavku imaju argument tipa `int` i imaju `int` kao tip povratne vrednosti.

`isalpha(c)` vraća ne-nula vrednost ako je `c` slovo, nulu inače;

`isupper(c)` vraća ne-nula vrednost ako je slovo `c` veliko, nulu inače;

`islower(c)` vraća ne-nula vrednost ako je slovo `c` malo, nulu inače;

`isdigit(c)` vraća ne-nula vrednost ako je `c` cifra, nulu inače;

`isalnum(c)` vraća ne-nula vrednost ako je `c` slovo ili cifra, nulu inače;

`isspace(c)` vraća ne-nula vrednost ako je `c` belina (razmak, tab, novi red, itd), nulu inače;

`toupper(c)` vraća karakter `c` konvertovan u veliko slovo ili, ukoliko je to nemoguće — sâm karakter `c`;

`tolower(c)` vraća karakter `c` konvertovan u malo slovo ili, ukoliko je to nemoguće — sâm karakter `c`;

11.4 Zaglavlje `math.h`

```
double sin(double); double asin(double);
double cos(double); double acos(double);
double tan(double); double atan(double); double atan2(double);
double log(double); double log10(double); double log2(double);
double pow(double); double exp(double); double sqrt(double);
```

Zaglavlje `math.h` sadrži deklaracije više od dvadeset često korišćenih matematičkih funkcija. Svaka od njih ima jedan ili dva argumenta tipa `double` i ima `double` kao tip povratne vrednosti.

`sin(x)` vraća vrednost funkcije $\sin(x)$, smatra se da je x zadato u radijanima;

`cos(x)` vraća vrednost funkcije $\cos(x)$, smatra se da je x zadato u radijanima;

`atan2(y, x)` vraća vrednost koja odgovara uglu u odnosu na x-osu tačke x, y. Ugao je u radijanima iz intervala $(-\pi, \pi]$. Vrednost nije definisana za koordinatni početak.

`exp(x)` vraća vrednost e^x;

`log(x)` vraća vrednost $\ln x$ (mora da važi $x > 0$); logaritam $\log_a b$ (za $a, b > 0$, $a \neq 1$) može se izračunati kao `log(b)/log(a)`.

`log10(x)` vraća vrednost $\log_{10} x$ (mora da važi $x > 0$);

`log2(x)` vraća vrednost $\log_{12} x$ (mora da važi $x > 0$);

`pow(x,y)` vraća vrednost x^y;

`sqrt(x)` vraća vrednost $\sqrt{x}$ (mora da važi $x > 0$); vrednost $\sqrt[n]{x}$ može se izračunati kao `pow(x, 1/n)`.

`fabs(x)` vraća apsolutnu vrednost od x.

Pored funkcija u ovom zaglavlju su definisane mnoge važne matematičke konstante. Na primer, `M_PI` ima vrednost broja π, `M_E` broja e, `M_SQRT2` broja $\sqrt{2}$ itd.

11.5 Zaglavlje `assert.h`

```
void assert(int)
```

`assert` Funkcija `assert` koristi se u fazi razvoja programa da ukaže na moguće greške. Ukoliko je pri pozivu `assert(izraz)` vrednost celobrojnog izraza `izraz` jednaka nuli, onda će na standardni tok za greške (`stderr`) biti ispisana poruka nalik sledećoj:

```
Assertion failed: expression, file filename, line nnn
```

i biće prekinuto izvršavanje programa. Ukoliko je simboličko ime `NDEBUG` definisano pre nego što je uključeno zaglavlje `<assert.h>`, pozivi funkcije `assert` se ignorišu. Dakle, funkcija `assert` se obično koristi u toku razvoja programa, u takozvanoj *debug* verziji. Kada je program spreman za korišćenje, proizvodi se *release* verzija i tada se pozivi funkcije `assert` ignorišu (jer je tada obično definisano ime `NDEBUG`). U fazi razvoja programa, upozorenja koja generiše `assert` ukazuju na krupne propuste u programu, ukazuju da tekući podaci ne zadovoljavaju neki uslov koji je podrazumevan, te pomažu u ispravljanju tih propusta. Funkcija `assert` ne koristi se da ukaže na greške u fazi izvršavanja programa koje nisu logičke (na primer, neka datoteka ne može da se otvori) već samo na one logičke greške koje ne smeju da se dogode. U završnoj verziji programa, pozivi funkcije `assert` imaju i dokumentacionu ulogu — oni čitaocu programa ukazuju na uslove za koje je autor programa podrazumevao da moraju da važe u nekoj tački programa.

Pitanja i zadaci za vežbu

Pitanje 11.5.1. *U kojoj datoteci zaglavlja je deklarisana funkcija* `cos`*? Kako se u programu može dobiti vrednost konstanti π i e?*

Pitanje 11.5.2. *Navesti prototip i opisati dejstvo funkcije* `strcpy`*. Navesti i jednu moguću implementaciju.*

Pitanje 11.5.3. *U kom zaglavlju standardne C biblioteke je deklarisana funkcija* `rand`*? Kako se može, korišćenjem funkcije iz standardne biblioteke, dobiti pseudoslučajan ceo broj iz intervala [10,20]? Kako se može dobiti pseudoslučajan ceo broj između* `n` *i* `m` *(pri čemu važi* `n<m`*)?*

Pitanje 11.5.4. *Šta je efekat funkcije* `exit`*?*

Pitanje 11.5.5. *U kojim situacijama se koristi makro* `assert`*? Kakav je efekat makroa* `assert` *u release izvršnoj verziji?*

Glava 12

Ulaz i izlaz C programa

Jezik C je dizajniran kao mali jezik i ulazno/izlazne operacije nisu direktno podržane samim jezikom, već specijalizovanim funkcijama iz standardne biblioteke jezika (koja je prisutna u svakom C okruženju). Pošto su ove funkcije deo standarda jezika C, mogu se koristiti u programima uz garantovanu prenosivost programa između različitih sistema. Svaka izvorna datoteka u kojoj se koriste ulazno/izlazne funkcije, trebalo bi da uključi standardno zaglavlje `<stdio.h>`.

12.1 Standardni tokovi

Standardna biblioteka implementira jednostavan model tekstualnog ulaza i izlaza. Ulaz i izlaz se modeluju tzv. *tokovima* (engl. stream) podataka (obično pojedinačnih bajtova ili karaktera). *Standardni ulaz* obično čine podaci koji se unose sa tastature. Podaci koji se upućuju na *standardni izlaz* se obično prikazuju na ekranu. Pored standardnog izlaza, postoji i *standardni izlaz za greške* na koji se obično upućuju poruke o greškama nastalim tokom rada programa i koji se, takođe, obično prikazuje na ekranu.

U mnogim okruženjima, moguće je izvršiti preusmeravanje (redirekciju) standardnog ulaza tako da se, umesto sa tastature, karakteri čitaju iz neke datoteke. Na primer, ukoliko se program pokrene sa

```
./prog < infile
```

onda program `prog` čita karaktere iz datoteke `infile`, umesto sa tastature.

Takođe, mnoga okruženja omogućavaju da se izvrši preusmeravanje (redirekcija) standardnog izlaza u neku datoteku. Na primer, ukoliko se program pokrene sa

```
./prog > outfile
```

onda program `prog` upisuje karaktere u datoteku `outfile`, umesto na ekran.

Ukoliko bi se poruke o greškama štampale na standardni izlaz, zajedno sa ostalim rezultatima rada programa, onda, u slučaju preusmeravanja standardnog izlaza u datoteku, poruke o greškama korisniku ne bi bile prikazane na ekranu i korisnik ih ne bi video. Ovo je glavni razlog uvođenja standardnog izlaza za greške koji se obično prikazuje na ekranu. Moguće je izvršiti redirekciju i standardnog izlaza za greške u datoteku, na primer:

```
./prog 2> errorfile
```

12.1.1 Ulaz i izlaz pojedinačnih karaktera

Najjednostavniji mehanizam čitanja sa standardnog ulaza je čitanje jednog po jednog karaktera korišćenjem funkcije `getchar`:

```
int getchar(void);
```

Funkcija `getchar` vraća sledeći karakter sa ulaza, svaki put kada se pozove, ili `EOF` kada dođe do kraja toka. Simbolička konstanta `EOF` je definisana u zaglavlju `<stdio.h>`. Njena vrednost je obično -1, ali umesto ove konkretne vrednosti ipak treba koristiti ime `EOF`. Funkcija `getchar` (kao i još neke srodne funkcije) umesto tipa `char` koristi tip `int` koji je dovoljno širok da u njega mogu da se smeste kako ASCII vrednosti od 0 do 127, tako i specijalna vrednost `EOF`, tj. -1. Ako bi povratni tip funkcije `getchar` bio `char`, a povratna vrednost u nekom konkretnom slučaju jednaka `EOF` (tj. konstantna vrednost -1), na sistemima na kojima je tip `char` podrazumevano neoznačen, vrednost `EOF` bi se konvertovala u 255, pa bi poređenje `getchar() == EOF` bilo netačno (jer bi sa leve strane bila vrednost 255 koja bi pre poređenja sa `-1` bila promovisana u tip `int`).

Funkcija `getchar()` najčešće se realizuje tako što karaktere uzima iz privremenog bafera koji se puni čitanjem jedne po jedne linije ulaza. Dakle, u interaktivnom radu sa programom, `getchar()` neće imati efekta sve dok se ne unese prelazak u novi red ili oznaka za kraj datoteke.[1]

Najjednostavniji mehanizam pisanja na standardni izlaz je pisanje jednog po jednog karaktera korišćenjem funkcije `putchar`:

```
int putchar(int);
```

Funkcija `putchar(c)` štampa karakter `c` na standardni izlaz, a vraća karakter koji je ispisala ili `EOF` ukoliko je došlo do greške.

Kao primer rada sa pojedinačnim karakterima, razmotrimo program koji prepisuje standardni ulaz na standardni izlaz, pretvarajući pri tom velika u mala slova.

Program 12.1.

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

int main()
{
    int c;
    while ((c = getchar()) != EOF)
        putchar(tolower(c));
    return 0;
}
```

[1] U standardnoj biblioteci ne postoji funkcija koja čita samo jedan karakter, ne cekajuci kraj ulaza.

Funkcija `tolower` deklarisana je u zaglavlju `<ctype.h>` i prevodi karaktere velikih slova u karaktere malih slova, ne menjajući pri tom ostale karaktere.

Pomenimo da su „funkcije“ poput `getchar` i `putchar` iz `<stdio.h>` i `tolower` iz `<ctype.h>` često makroi (zasnovani na drugim funkcijama iz standardne biblioteke), čime se izbegava dodatno vreme i prostor potreban za realizaciju funkcijskog poziva.

12.1.2 Linijski ulaz i izlaz

Bibliotečka funkcija `gets`:

```
char* gets(char* s);
```

čita karaktere sa standardnog ulaza do kraja tekuće linije ili do kraja datoteke i karaktere smešta u nisku s. Oznaka kraja reda se ne smešta u nisku, a niska se automatski završava nulom. Ne vrši se nikakva provera da li niska `s` sadrži dovoljno prostora da prihvati pročitani sadržaj i ovo funkciju `gets` čini veoma opasnom za korišćenje. U slučaju da je ulaz uspešno pročitan, `gets` vraća `s`, a inače vraća `NULL` pokazivač.

Bibliotečka funkcija `fputs`:

```
int puts(const char* s);
```

ispisuje nisku na koju ukazuje `s` na standardni izlaz, dodajući pri tom oznaku za kraj reda. Završna nula se ne ispisuje. Funkcija `puts` vraća `EOF` u slučaju da je došlo do greške, a nenegativnu vrednost inače.

12.1.3 Formatirani izlaz — `printf`

Funkcija `printf` već je korišćena u prethodnom tekstu. Njen prototip je:[2]

```
int printf(const char *format, arg1, arg2, ...);
```

Funkcija `printf` prevodi vrednosti osnovnih tipova podataka u serije karaktera i usmerava ih na standardni izlaz. Funkcija vraća broj odštampanih karaktera. Slučajevi korišćenja ove funkcije iz prethodnih glava jesu najuobičajeniji, ali svakako nisu potpuni. Pozivi funkcija `putchar` i `printf` mogu biti isprepletani — izlaz se vrši u istom redosledu u kojem su ove funkcije pozvane.

Funkcija `printf` prevodi, formatira i štampa svoje argumente na standardni izlaz pod kontrolom date „format niske“. Format niska sadrži dve vrste objekata: obične karaktere, koji se doslovno prepisuju na standardni izlaz i specifikacije konverzija od kojih svaka uzrokuje konverziju i štampanje sledećeg uzastopnog argumenta funkcije `printf`. Svaka specifikacija konverzije počinje karakterom % i završava se karakterima konverzije. Između % i karaktera konverzije moguće je da se redom nađu:

- Znak minus (-), koji prouzrokuje levo poravnanje konvertovanog argumenta.

[2] Funkcija printf je jedna od funkcija iz standardne biblioteke koja ima *promenljiv broj argumenata* (tj. broj argumenata u pozivima ne mora uvek da bude isti). Programer može definisati svoje funkcije sa promenljivim brojem argumenata koristeći funkcije deklarisane u standardnom zaglavlju stdarg.h.

- Broj koji specifikuje najmanju širinu polja. Konvertovani argument se štampa u polju koje je barem ovoliko široko. Ukoliko je potrebno, polje se dopunjava razmacima sa leve (odnosno desne strane, ukoliko se traži levo poravnanje).
- Tačka `.`, koja odvaja širinu polja od preciznosti.
- Broj za preciznost, koji specifikuje najveći broj karaktera koje treba štampati iz niske u slučaju štampanja niske ili broj tačaka iza decimalne tačke u slučaju broja u pokretnom zarezu ili najmanji broj cifara u slučaju celog broja.
- Karakter `h`, ako ceo broj treba da se štampa kao `short`, ili `l` ako ceo broj treba da se štampa kao `long`.

Konverzioni karakteri

Konverzioni karakteri su prikazani u narednoj tabeli. Ukoliko se nakon % navede pogrešan konverzioni karakter, ponašanje je nedefinisano.

Karakter	Tip	Štampa se kao
`d,i`	`int`	dekadni broj
`o`	`int`	neoznačeni oktalni broj (bez vodećeg simbola 0)
`x,X`	`int`	neoznačeni heksadekadni broj (bez vodećih 0x ili 0X), korišćenjem abcdef ili ABCDEF za 10, ...,15
`u`	`int`	neoznačeni dekadni broj
`c`	`int`	pojedinačni karakter
`s`	`char *`	štampa karaktere niske do karaktera `'\0'` ili broja karaktera navedenog u okviru preciznosti.
`f`	`double`	[-]m.dddddd, gde je broj d-ova određen preciznošću (podrazumevano 6)
`e,E`	`double`	[-]m.dddddde+/-xx ili [-]m.ddddddE+/-xx, gde je broj d-ova određen preciznošću (podrazumevano 6)
`g,G`	`double`	koristi %e ili %E ako je eksponent manji od -4 ili veći ili jednak preciznosti; inače koristi %f. Završne nule i završna decimalna tačka se ne štampaju
`p`	`void *`	pokazivač (reprezntacija zavisna od implementacije)
%		nijedan argument se ne konvertuje; štampa %

Za navedene konverzione karaktere postoje i modifikatori dužine: `h` (za celobrojne tipove, za `short`), `l` (za celobrojne tipove, za `long`), `L` (za brojeve u pokretnom zarezu, za `long double`). Tako, na primer, sekvenca `hd` može se koristiti za ispisivanje podatka tipa `short int`, sekvenca `ld` može se koristiti za ispisivanje podatka tipa `long int`, a sekvenca `Lf` za ispisivanje podatka tipa `long double`.

Primetimo da za format `float` ne postoji zaseban konverzioni karakter. To je zbog toga što se svi argumenti tipa `float` implicitno konvertuju u tip `double` prilikom poziva funkcije `printf` (jer je `printf` funkcija sa promenljivim brojem argumenata kod kojih je ova promocija podrazumevana). Počevši od standarda C99 pored formata `%f` dopušteno je navoditi i `%lf` (zanimljivo, kompilator GCC to i zahteva za tip `double`).

Širina polja ili preciznost se mogu zadati i kao `*`, i tada se njihova vrednost određuje na osnovu vrednosti narednog argumenta (koji mora biti `int`). Na primer, da se odštampa najviše `max` karaktera iz niske `s`, može se navesti:

```
printf("%.*s", max, s);
```

Prilikom poziva funkcije `printf` na osnovu prvog argumenta (format niske) određuje se koliko još argumenata sledi i koji su njihovi tipovi. Ukoliko se ne navede odgovarajući broj argumenata ili se navedu argumenti neodgovarajućih tipova, dolazi do greške. Takođe, treba razlikovati naredna dva poziva:

```
   printf(s);         /* pogresno ako s sadrzi karakter % */
   printf("%s", s);   /* bezbedno */
```

Primer prikaza celih brojeva:

```
int    count = -9234;
printf("Celobrojni formati:\n"
       "\tDekadni: %d  Poravnat: %.6d  Neoznacen: %u\n",
       count, count, count);
printf("Broj %d prikazan kao:\n\tHex: %Xh C hex: 0x%x Oct: %o\n",
        count, count, count, count );
```

```
Celobrojni formati:
   Dekadni: -9234  Poravnat: -009234  Neoznacen: 4294958062
Broj -9234 prikazan kao:
   Hex: FFFFDBEEh C hex: 0xffffdbee Oct: 37777755756
```

```
/* Prikaz konstanti zapisanih u razlicitim osnovama. */
printf("Cifre 10 predstavljaju:\n");
printf("\tHex: %i  Octal: %i  Decimal: %i\n",
        0x10, 010, 10);
```

```
Cifre 10 predstavljaju:
   Hex: 16  Octal: 8  Decimal: 10
```

Primer prikaza karaktera:

```
   char  ch = 'h';
   printf("Karakteri u polju date sirine:\n%10c%c\n", ch, ch);
```

```
Karakteri u polju date sirine:
         hh
```

Primer prikaza brojeva u pokretnom zarezu:

```
double fp = 251.7366;
printf("Brojevi u pokretnom zarezu:\n\t%f %.2f %e %E\n", fp, fp, fp, fp);
```

```
Realni brojevi:
   251.736600 251.74 2.517366e+002 2.517366E+002
```

Primer prikaza niski:
Ovaj primer prikazuje navođenje širine polja, poravnanja i preciznosti prilikom štampanja niski. U narednoj tabeli dat je efekat različitih format niski na štampanje niske `"hello, world"` (12 karaktera). Dvotačke su stavljene radi boljeg ilustrovanja efekta raznih formata.

```
   :%s:          :hello, world:
   :%10s:        :hello, world:
   :%.10s:       :hello, wor:
   :%-10s:       :hello, world:
   :%.15s:       :hello, world:
   :%-15s:       :hello, world   :
   :%15.10s:     :     hello, wor:
   :%-15.10s:    :hello, wor     :
```

12.1.4 Formatirani ulaz — `scanf`

Funkcija `scanf` je ulazni analogon funkcije `printf`. Funkcija `scanf` čita karaktere sa standardnog ulaza, interpretira ih na osnovu specifikacije navedene format niskom i smešta rezultat na mesta određena ostalim argumentima:

```
int scanf(const char *format, ...)
```

Opis format niske dat je u nastavku. Ostali argumenti moraju biti pokazivači i određuju lokacije na koje se smešta konvertovani ulaz. Kao i u slučaju funkcije `printf`, ovde će biti dat samo prikaz najčešćih načina korišćenja ove funkcije. Funkcija `scanf` prestaje sa radom kada iscrpi svoju format nisku ili kada neki deo ulaza ne može da se uklopi u shemu navedenu format niskom. Funkcija vraća broj uspešno uklopljenih i dodeljenih izlaznih podataka. U slučaju kraja datoteke, funkcija vraća `EOF`. Ova vrednost je različita od vrednosti 0 koja se vraća u slučaju da tekući karakter sa ulaza ne može da se uklopi u prvu specifikaciju zadatu format niskom. Svaki naredni poziv funkcije `scanf` nastavlja tačno od mesta na ulazu na kojem je prethodni poziv stao.

Format niska sadrži specifikacije konverzija kojima se kontroliše konverzija teksta pročitanog sa ulaza. Format niska može da sadrži:

- Praznine ili tabulatore koji se ne zanemaruju.
- Obične karaktere (ne %), za koje se očekuje da se poklope sa sledećim ne-belinama sa standardnog ulaza.

- Specifikacije konverzija, koje se sastoje od karaktera %, opcionog karaktera `*` koji sprečava dodeljivanje, opcionog broja kojim se navodi maksimalna širina polja, kao i od konverzionog karaktera.

- Specifikacija konverzije određuje konverziju narednog ulaznog polja. Obično se rezultat ove konverzije upisuje u promenljivu na koju pokazuje naredni ulazni argument. Ipak, ako se navede karakter `*`, konvertovana vrednost se preskače i nigde ne dodeljuje. Ulazno polje se definiše kao niska karaktera koji nisu beline. Ono se prostire ili do narednog karaktera beline ili do širine polja, ako je ona eksplicitno zadana. Ovo znači da funkcija `scanf` može da čita ulaz i iz nekoliko različitih linija sa standardnog ulaza, jer se prelasci u novi red tretiraju kao beline[3].

- Konverzioni karakter upućuje na željenu interpretaciju ulaznog polja.

U narednoj tabeli navedeni su konverzioni karakteri.

Karakter	Tip	Ulazni podatak
d	`int*`	dekadni ceo broj
i	`int*`	ceo broj, može biti i oktalan (ako ima vodeću 0) ili heksadekadan (vodeći 0x ili 0X)
o	`int*`	oktalni ceo broj (sa ili bez vodeće nule)
u	`unsigned*`	neoznačeni dekadni broj
x	`int*`	heksadekadni broj (sa ili bez vodećih 0x ili 0X)
c	`char*`	karakter, naredni karakter sa ulaza se smešta na navedenu lokaciju; uobičajeno preskakanje belina se ne vrši u ovom slučaju; Za čitanje prvog ne-belog karaktera, koristi se `%1s`
s	`char*`	niska (bez navodnika), ukazuje na niz karaktera dovoljno dugačak za nisku uključujući i terminalnu `'\0'` koja se automatski dopisuje
e,f,g	`double*`	broj u pokretnom zarezu sa opcionim znakom, opcionom decimalnom tačkom i opcionim eksponentom
%		doslovno %; ne vrši se dodela

Slično kao kod funkcije `printf`, ispred karaktera konverzije `d`, `i`, `o`, `u`, i `x` moguće je navesti i karakter `h` koji ukazuje da se u listi argumenata očekuje pokazivač na `short`, karakter `l` koji ukazuje da se u listi argumenata očekuje pokazivač na `long` ili karakter `L` koji ukazuje da se u listi argumenata očekuje pokazivač na `long double`.

Kao što je već rečeno, argumenti funkcije `scanf` moraju biti pokazivači. Ubedljivo najčešći oblik pogrešnog korišćenja ove funkcije je:

```
scanf("%d", n);
```

umesto

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

[3]Pod belinama se podrazumevaju razmaci, tabulatori, prelasci u novi redovi (carriage return i/ili line feed), vertikalni tabulatori i formfeed.

Ova greška se obično ne otkriva tokom kompilacije.

Funkcija `scanf` ignoriše beline u okviru format niske. Takođe, preskaču se beline tokom čitanja ulaznih vrednosti. Pozivi funkcije `scanf` mogu biti pomešani sa pozivima drugih funkcija koje čitaju standardni ulaz. Naredni poziv bilo koje ulazne funkcije će pročitati prvi karakter koji nije pročitan tokom poziva funkcije `scanf`.

Primeri

U narednom primeru, korišćenjem funkcije `scanf` implementiran je jednostavni kalkulator:

Program 12.2.

```
#include <stdio.h>

int main()
{
    double sum, v;

    sum = 0.0;
    while (scanf("%f", &v) == 1)
        printf("\t%.2f\n", sum += v);
    return 0;
}
```

Datume sa ulaza koji su u formatu `25 Dec 1988`, moguće je čitati korišćenjem:

```
int day, year;
char monthname[20];

scanf("%d %s %d", &day, monthname, &year);
```

Ispred `monthname` nema potrebe navesti simbol `&`, jer je ime niza već pokazivač. Doslovni karakteri se mogu pojaviti u okviru format niske i u tom slučaju se oni očekuju i na ulazu. Tako, za čitanje datuma sa ulaza koji su u formatu `12/25/1988`, moguće je koristiti:

```
int day, month, year;
scanf("%d/%d/%d", &month, &day, &year);
```

12.2 Ulaz iz niske i izlaz u nisku

Funkcija `printf` vrši ispis formatiranog teksta na standardni izlaz. Standardna biblioteka jezika C definiše funkciju `sprintf` koja je veoma slična funkciji `printf`, ali rezultat njenog rada je popunjavanje niske karaktera formatiranim tekstom. Prototip funkcije je:

```
int sprintf(char *string, const char *format, arg1, arg2, ...);
```

Ova funkcija formatira argumente `arg1`, `arg2`, ... na osnovu format niske, a rezultat smešta u nisku karaktera prosleđenu kao prvi argument, podrazumevajući da je ona dovoljno velika da smesti rezultat.

Slično funkciji `sprintf` koja vrši ispis u nisku karaktera umesto na standardni izlaz, standardna biblioteka jezika C definiše i funkciju `sscanf` koja je analogna funkciji `scanf`, osim što ulaz čita iz date niske karaktera, umesto sa standardnog ulaza.

```
int sscanf(const char* input, char* format, ...)
```

Navedimo primer korišćenja ove funkcije. Kako bi se pročitao ulaz čiji format nije fiksiran, najčešće je dobro pročitati celu liniju sa ulaza, a zatim je analizirati korišćenjem `sscanf`. Da bi se pročitao datum sa ulaza u nekom od prethodno navedena formata, moguće je koristiti:

```
while (getline(line, sizeof(line)) > 0) {
    if (sscanf(line, "%d %s %d", &day, monthname, &year) == 3)
        printf("valid: %s\n", line); /* 25 Dec 1988 */
    else if (sscanf(line, "%d/%d/%d", &month, &day, &year) == 3)
        printf("valid: %s\n", line); /* 12/25/1988 */
    else
        printf("invalid: %s\n", line); /* pogresan oblik */
}
```

12.3 Ulaz iz datoteka i izlaz u datoteke

Do sada opisane funkcije za ulaz i izlaz su uglavnom čitale sa standardnog ulaza i pisale na standardni izlaz, koji su automatski definisani i od strane operativnog sistema i kojima program automatski ima pristup. Iako je, mehanizmima preusmeravanja, bilo moguće izvesti programski pristup lokalnim datotekama, mehanizam preusmeravanja je veoma ograničen jer ne daje mogućnost istovremenog čitanja (ili pisanja) datoteke i standardnog ulaza kao ni mogućnost istovremenog čitanja (ili pisanja) više datoteka. Zbog toga, jezik C nudi direktnu podršku za rad sa lokalnim datotekama, bez potrebe za korišćenjem usluga preusmeravanja operativnog sistema. Sve potrebne deklaracije i za rad sa datotekama nalaze u zaglavlju `<stdio.h>`.

12.3.1 Tekstualne i binarne datoteke

Iako se svaka datoteka može razmatrati kao niz bajtova, obično razlikujemo datoteke koje sadrže tekstualni sadržaj od datoteka koje sadrže binarni sadržaj. U zavisnosti od toga da li se u datoteci nalazi tekstualni ili binarni sadržaj, razlikuju se dva različita načina pristupa.

Prvi način je prilagođen tekstualnim datotekama čiji sadržaj u principu čine vidljivi karakteri, sa dodatkom oznake kraja reda i horizontalnog tabulatora. Ovakve datoteke se obično obrađuju liniju po liniju, što je karakteristično za tekst. Na primer, iz datoteke se čita linija po linija ili se u datoteku upisuje linija po linija. Očigledno, u ovom slučaju oznaka za kraj linije je veoma relevantna i značajna. Međutim, na različitim sistemima tekst se kodira na različite načine i na nekim sistemima kraj reda u tekstualnoj datoteci se zapisuje sa dva karaktera (na primer, sa `\r\n` na sistemima Windows), a na nekim sa samo

jednim karakterom (na primer, sa `\n` na sistemu Linux). Kako bi se ovakvi detalji sakrili od programera i kako programer ne bi morao da se stara o ovakvim specifičnostima, C jezik nudi *tekstualni mod* rada sa datotekama. Ukoliko se datoteka otvori u tekstualnom modu, prilikom svakog čitanja i upisa u datoteku vrši se konverzija iz podrazumevanog formata označavanja kraja reda u jedinstven karakter `\n`. Tako će na Windows sistemima dva karaktera `\r\n` na kraju reda biti pročitana samo kao `\n` i, obratno, kada se u datoteku upisuje `\n` biće upisana dva karaktera `\r\n`. Na ovaj način pojednostavljuje se i olakšava rad sa datotekama koje čine tekstualni podaci. Da bi interpretiranje sadržaja tekstualne datoteke bilo uvek ispravno treba izbegavati, na primer, pravljenje tekstualnih datoteka koja na kraju poslednjeg reda nemaju oznaku kraja reda, izbegavati razmake u linijama nakon oznake za kraj reda, izbegavati da tekstualne datoteke sadrže karaktere koji nisu vidljivi karakteri, oznake kraja reda i tabulatora i slično.

Drugi način pristupa datotekama prilagođen je datotekama čiji sadržaj ne predstavlja tekst i u kojima se mogu naći bajtovi svih vrednosti od 0 do 255 (na primer, `jpg` ili `zip` datoteke). Za obradu ovakvih datoteka, jezik C nudi *binarni mod* rada sa datotekama. U ovom slučaju nema nikakve konverzije i interpretiranja karaktera prilikom upisa i čitanja i svaki bajt koji postoji u datoteci se čita doslovno.

Razlika između tekstualnih i binarnih datoteka još je oštrija u jezicima koji podržavaju Unicode, jer se u tom slučaju više bajtova čita i konvertuje u jedan karakter.

12.3.2 Pristupanje datoteci

Kako bi se pristupilo datoteci, bilo za čitanje, bilo za pisanje, potrebno je izvršiti određenu vrstu povezivanja datoteke i programa. Za ovo se koristi bibliotečka funkcija `fopen`:

```
FILE* fopen(const char* filename, const char* mode);
```

Funkcija `fopen` dobija nisku koja sadrži ime datoteke (na primer, `datoteka.txt`) i uz pomoć usluga operativnog sistema (koje neće na ovom mestu biti detaljnije opisane) vraća pokazivač na strukturu koja predstavlja sponu između lokalne datoteke i programa i koja sadrži informacije o datoteci koje će biti korišćene prilikom svakog pristupa datoteci. Ove informacije mogu da uključe adresu početka *bafera* (*prihvatnik*, eng. buffer) kroz koji se vrši komunikacija sa datotekom, tekuću poziciju u okviru bafera, informaciju o tome da li se došlo do kraja datoteke, da li je došlo do greške i slično. Programer ne mora i ne bi trebalo da direktno koristi ove informacije, već je dovoljno da čuva pokazivač na strukturu `FILE` i da ga prosleđuju svim funkcijama koje treba da pristupaju datoteci. Ime `FILE` je ime tipa, a ne ime strukture; ovo ime je uvedeno korišćenjem `typedef` (videti poglavlje 6.8.5).

Prvi argument funkcije `fopen` je niska karaktera koja sadrži ime datoteke. Drugi argument je niska karaktera koja predstavlja način (mod) otvaranja datoteke i koja ukazuje na to kako će se datoteka koristiti. Dozvoljeni modovi uključuju čitanje (read, `"r"`), pisanje (write, `"w"`) i dopisivanje (append, `"a"`). Ako se datoteka koja ne postoji na sistemu otvara za pisanje ili dopisivanje, onda se ona kreira (ukoliko je to moguće). U slučaju da se postojeća datoteka

otvara za pisanje, njen stari sadržaj se briše, dok se u slučaju otvaranja za dopisivanje stari sadržaj zadržava, a novi sadržaj upisuje nakon njega. Ukoliko se pokušava čitanje datoteke koja ne postoji dobija se greška. Takođe, greška se javlja i u slučaju da se pokušava pristup datoteci za koju program nema odgovarajuće dozvole. U slučaju greške funkcija `fopen` vraća `NULL`. Modovi `r+`, `w+` i `a+` ukazuju da će rad sa datotekom podrazumevati i čitanje i pisanje (ili dopisivanje). U slučaju da se želi otvaranje binarne datoteke, na mod se dopisuje `"b"` (na primer, `rb`, `wb`, `a+b`, ...).

Kada se C program pokrene, operativni sistem otvara tri toka podataka (standardni ulaz, stadnardni izlaz i standardni izlaz za greške), kao da su datoteke i obezbeđuje pokazivače kojima im se može pristupati. Ti pokazivači se nazivaju:

```
FILE* stdin;
FILE* stdout;
FILE* stderr;
```

Funkcija

```
int fclose(FILE *fp)
```

prekida vezu između programa i datoteke koju je funkcija `fopen` ostvarila. Pozivom funkcije `fclose` prazne se baferi koji privremeno čuvaju sadržaj datoteka čime se obezbeđuje da sadržaj zaista bude upisan na disk. Takođe, kreirana struktura `FILE` nije više potrebna i uklanja se iz memorije. S obzirom na to da većina operativnih sistema ograničava broj datoteka koje mogu biti istovremeno otvorene, dobra je praksa zatvarati datoteke čim prestanu da budu potrebne. U slučaju da program normalno završi, funkcija `fclose` se poziva automatski za svaku otvorenu datoteku.

12.3.3 Ulaz i izlaz pojedinačnih karaktera

Nakon što se otvori datoteka, iz nje se čita ili se u nju piše sadržaj. To mogu biti i pojedinačni karakteri.

Funkcija `getc` vraća naredni karakter iz datoteke određene prosleđenim `FILE` pokazivačem ili `EOF` u slučaju kraja datoteke ili greške.

```
int getc(FILE *fp)
```

Slično, funkcija `putc` upisuje dati karakter u datoteku određenu prosleđenim `FILE` pokazivačem i vraća upisani karakter ili `EOF` u slučaju greške. Iako su greške prilikom izlaza retke, one se nekada javljaju (na primer, ako se prepuni hard disk), pa bi trebalo vršiti i ovakve provere.

```
int putc(int c, FILE *fp);
```

Slično kao i `getchar` i `putchar`, `getc` i `putc` mogu biti definisani kao makroi, a ne funkcije.

Naredni primer kopira sadržaj datoteke `ulazna.txt` u datoteku `izlazna.txt` čitajući i pišući pritom pojedinačne karaktere. Naglasimo da je ovo veoma neefikasan način kopiranja datoteka.

Program 12.3.

```
#include <stdio.h>

/* filecopy:  copy file ifp to file ofp */
void filecopy(FILE *ifp, FILE *ofp)
{
     int c;
     while ((c = getc(ifp)) != EOF)
         putc(c, ofp);
}

int main()
{
     FILE *ulaz, *izlaz;

     ulaz = fopen("ulazna.txt","r");
     if(ulaz == NULL)
         return -1;

     izlaz = fopen("izlazna.txt","w");
     if(izlaz == NULL)
         return -1;

     filecopy(ulaz,izlaz);

     fclose(ulaz);
     fclose(izlaz);
     return 0;
}
```

Funkcija `ungetc`:

```
int ungetc (int c, FILE *fp);
```

„vraća“ karakter u datoteku i vraća identifikator pozicije datoteke tako da naredni poziv operacije čitanja nad ovom datotekom vrati upravo vraćeni karakter. Ovaj karakter može, ali ne mora biti jednak poslednjem pročitanom karakteru datoteke. Iako ova funkcije utiče na naredne operacije čitanja datoteke, ona ne menja fizički sadržaj same datoteke (naime vraćeni karakteri se najčešće smeštaju u memorijski bafer odakle se dalje čitaju, a ne u datoteku na disku). Funkcija `ungetc` vraća `EOF` ukoliko je došlo do greške, a vraćeni karakter inače.

Naredni primer čita karaktere iz datoteke dok su u pitanju cifre i pri tom gradi dekadni ceo broj. Pošto poslednji karakter koji je pročitan nije cifra, on se (uslovno rečeno) vraća u tok kako ne bi bio izostavljen prilikom narednih čitanja.

```
#include <stdio.h>

int readint(FILE* fp) {
  int val = 0, c;
  while (isdigit(c = getc(fp)))
```

```
    val = 10*val + (c - '0');
  ungetc(c, fp);
}
```

12.3.4 Provera grešaka i kraja datoteke

Funkcija `feof` vraća vrednost *tačno* (ne-nula) ukoliko se došlo do kraja date datoteke.

```
int feof(FILE *fp)
```

Funkcija `ferror` vraća vrednost *tačno* (ne-nule) ukoliko se došlo do greške u radu sa datotekom.

```
int ferror(FILE *fp)
```

12.3.5 Linijski ulaz i izlaz

Standardna biblioteka definiše i funkcije za rad sa tekstualnim datotekama liniju po liniju.

Funkcija `fgets` čita jednu liniju iz datoteke.

```
char *fgets(char *line, int maxline, FILE *fp)
```

Funkcija `fgets` čita sledeću liniju iz datoteke (uključujući oznaku kraja reda) iz datoteke `fp` i rezultat smešta u nisku `line`. Može da bude pročitano najviše `maxline-1` karaktera. Rezultujuća niska završava se karakterom `'\0'`. U normalnom toku izvršavanja, funkcija `fgets` vraća pokazivač na početak linije, a u slučaju kraja datoteke ili greške vraća `NULL`.

Funkcija `fputs` ispisuje nisku (koja ne mora da sadrži oznaku kraja reda) u datoteku:

```
int fputs(const char *line, FILE *fp)
```

Ona vraća `EOF` u slučaju da dođe do greške, a neki nenegativan broj inače.

Za razliku od funkcija `fgets` i `fputs`, funkcija `gets` briše završni `'\n'`, a funkcija `puts` ga dodaje.

Implementacija bibliotečkih funkcija nije značajno različita od implementacije ostalih funkcija, što je ilustrovano implementacijama funkcija `fgets` i `fputs`, u obliku doslovno preuzetom iz jedne realne implementacije C biblioteke:

```
/* fgets:  get at most n chars from iop */
char *fgets(char *s, int n, FILE *iop)
{
    register int c;
    register char *cs;

    cs = s;
    while (--n > 0 && (c = getc(iop)) != EOF)
        if ((*cs++ = c) == '\n')
            break;
```

```
    *cs = '\0';
    return (c == EOF && cs == s) ? NULL : s;
}

/* fputs:  put string s on file iop */
int fputs(char *s, FILE *iop)
{
    register int c;

    while (c = *s++)
       putc(c, iop);
    return ferror(iop) ? EOF : 0;
}
```

Korišćenjem funkcije `fgets` jednostavno je napraviti funkciju koja čita liniju po liniju sa standardnog ulaza, vraćajući dužinu pročitane linije, odnosno nulu kada liniju nije moguće pročitati:

```
/* getline:  read a line, return length */
int getline(char *line, int max)
{
     if (fgets(line, max, stdin) == NULL)
        return 0;
     else
        return strlen(line);
}
```

12.3.6 Formatirani ulaz i izlaz

Za formatirani ulaz i izlaz mogu se koristiti funkcije `fscanf` i `fprintf`. One su identične funkcijama `scanf` i `printf`, osim što je prvi argument `FILE` pokazivač koji ukazuje na datoteku iz koje se čita, odnosno u koju se piše. Format niska je u ovom slučaju drugi argument.

```
int fscanf(FILE *fp, const char *format, ...)
int fprintf(FILE *fp, const char *format, ...)
```

12.3.7 Rad sa binarnim datotekama

Standardna biblioteka nudi funkcije za direktno čitanje i pisanje bajtova u binarne datoteke. Progameru su na raspolaganju i funkcije za pozicioniranje u okviru datoteka, pa se binarne datoteke mogu koristiti i kao neke vrste memorije sa slobodnim pristupom.

Funkcija `fread` se koristi za čitanje niza slogova iz binarne datoteke, a funkcija `fwrite` za pisanje niza slogova u binarnu datoteku.

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size,
              size_t nmemb, FILE *stream);
```

Za ove funkcije, prvi argument je adresa na koju se smeštaju pročitani slogovi iz datoteke, odnosno u kojoj su smešteni slogovi koji će biti upisani u datoteku. Drugi argument predstavlja veličina jednog sloga, treći broj slogova, a četvrti pokazivač povezan datoteke. Funkcije vraćaju broj uspešno pročitanih, odnosno upisanih slogova.

Funkcija `fseek` služi za pozicioniranje na mesto u datoteci sa koga će biti pročitan ili na koje će biti upisan sledeći podatak. Funkcija `ftell` vraća trenutnu poziciju u datoteci (u obliku pomeraja od početka izraženog u broju bajtova). Iako primena ovih funkcija nije striktno ograničena na binarne datoteke, one se najčešće koriste sa binarnim datotekama.

```
int fseek (FILE * stream, long int offset, int origin);
long int ftell (FILE * stream);
```

Prvi argument obe funkcije je pokazivač na datoteku. Funkcija `fseek` kao drugi argument dobija pomeraj izražen u broju bajtova, dok su za treći argument dozvoljene vrednosti `SEEK_SET` koja označava da se pomeraj računa u odnosu na početak datoteke, `SEEK_CUR` koji označava da se pomeraj računa u odnosu na tekuću poziciju i `SEEK_END` koji označava da se pomeraj računa u odnosu na kraj datoteke.

Sledeći primer ilustruje primenu ovih funkcija.

Program 12.4.

```
#include <stdio.h>

int main() {
  struct S {
    int broj;
    int kvadrat;
  } s;

  FILE* f;
  int i;

  if ((f = fopen("junk", "r+b")) == NULL) {
    fprintf(stderr, "Greska prilikom otvaranja datoteke");
    return 1;
  }

  for (i = 1; i <= 5; i++) {
     s.broj = i; s.kvadrat = i*i;
     fwrite(&s, sizeof(struct S), 1, f);
  }

  printf("Upisano je %ld bajtova\n", ftell(f));

  for (i = 5; i > 0; i--) {
    fseek(f, (i-1)*sizeof(struct S), SEEK_SET);
    fread(&s, sizeof(struct S), 1, f);
    printf("%3d %3d\n", s.broj, s.kvadrat);
```

```
  }

  fclose(f);
}
```

12.4 Argumenti komandne linije programa

Jedan način da se određeni podaci proslede programu je i da se navedu u komandnoj liniji prilikom njegovog pokretanja. Argumenti koji su tako navedeni prenose se programu kao argumenti funkcije `main`. Prvi argument (koji se obično naziva `argc`, od engleskog *argument count*) je broj argumenata komandne linije (uključujući i sâm naziv programa) navedenih prilikom pokretanja programa. Drugi argument (koji se obično naziva `argv`, od engleskog *argument vector*) je niz niski karaktera koje sadrže argumente — svaka niska direktno odgovara jednom argumentu. Nazivu programa odgovara niska `argv[0]`. Ako je `argc` tačno 1, onda nisu navedeni dodatni argumenti nakon imena programa. Dakle, `argv[0]` je ime programa, `argv[1]` do `argv[argc-1]` su tekstovi argumenata programa, a element `argv[argc]` sadrži vrednost `NULL`. Identifikatori `argc` i `argv` su proizvoljni i funkcija `main` može biti deklarisana i na sledeći način:

```
int main (int br_argumenata, char* argumenti[]);
```

Naredni jednostavan program štampa broj argumenata kao i sadržaj vektora argumenata.

Program 12.5.

```
#include <stdio.h>
int main(int argc, char* argv[]) {
    int i;
    printf("argc = %d\n", argc);

    /* Nulti argument uvek je ime programa (na primer, a.out)*/
    for (i = 0; i < argc; i++)
        printf("argv[%d] = %s\n", i, argv[i]);
    return 0;
}
```

Ukoliko se program prevede sa `gcc -o echoargs echoargs.c` i pozove sa `./echoargs -U zdravo svima "dobar dan"`, ispisaće

```
argc = 5
argv[0] = ./echoargs
argv[1] = -U
argv[2] = zdravo
argv[3] = svima
argv[4] = dobar dan
```

Jedna od mogućih upotreba argumenata komandne linije je da se programu navedu različite opcije. Običaj je da se opcije kodiraju pojedinačnim karakterima i da se navode iza karaktera `-`. Pri tome, često je predviđeno da se iza

jednog simbola - može navesti više karaktera koji određuju opcije. Na primer, pozivom:

```
./program -a -bcd 134 -ef zdravo
```

programu se zadaju opcije `a`, `b`, `c`, `d` `e` i `f` i argumenti `134` i `zdravo`.

Naredni program ispisuje sve opcije pronađene u komandnoj liniji:

Program 12.6.

```
#include <stdio.h>

int main(int argc, char* argv[]) {
    /* Za svaki argument komande linije, pocevsi od argv[1]
       (preskace se ime programa) */
    int i;
    for (i = 1; i < argc; i++) {
        /* Ukoliko i-ti argument pocinje crticom */
        if (argv[i][0] == '-') {
            /* Ispisuju se sva njegova slova od pozicije 1 */
            int j;
            for (j = 1; argv[i][j] != '\0'; j++)
                printf("Prisutna je opcija : %c\n", argv[i][j]);
        }
    }
    return 0;
}
```

Kraće, ali dosta kompleksnije rešenje se intenzivno zasniva na pokazivačkoj aritmetici i prioritetu operatora.

Program 12.7.

```
#include <stdio.h>

int main(int argc, char* argv[]) {
  char c;
  /* Dok jos ima argumenata i dok je na poziciji 0 crtica */
  while(--argc>0 && (*++argv)[0]=='-')
     /* Dok se ne dodje do kraja tekuce niske */
     while (c=*++argv[0])
       printf("Prisutna je opcija : %c\n",c);
  return 0;
}
```

Pitanja i zadaci za vežbu

Pitanje 12.1.

Šta je to `stderr`*?*

Pitanje 12.2. *U kojoj datoteci zaglavlja se nalazi deklaracija funkcije* `printf`*? Navesti njen prototip. Koju vrednost vraća ova funkcija?*

Pitanje 12.3. *U kojoj datoteci zaglavlja se nalazi deklaracija funkcije* `scanf`*? Navesti njen prototip. Koju vrednost vraća ova funkcija?*

Pitanje 12.4. *Šta, u format niski koja je argument funkcije* `printf`*, specifikuje opcioni broj koji se navodi iza opcione tačke za argument koji je niska (na primer:* `printf("%.2s", "zdravo");`*)?*

Šta specifikuje opcioni broj koji se navodi iza opcione tačke za argument koji realan broj (na primer: `printf("%.2f", 3.2453);`*)?*

Pitanje 12.5. *Šta je rezultat poziva funkcije* `printf("#%6.*f#", 2, 3.14159);`*?*

Šta je rezultat poziva funkcije `printf("%6.1f",1/7);`*?*

Pitanje 12.6. *Navesti prototip funkcije iz standardne biblioteke koja omogućava formatirani ispis u nisku. Navesti primer korišćenja. O čemu treba voditi računa prilikom pozivanja ove funkcije?*

Pitanje 12.7. *Na što jednostavniji način u nisku* `n` *(deklarisanu sa* `char n[8];`*) upisati vrednosti* `1/7` *i* `2/7` *sa po pet cifara iza decimalnog zareza i razdvojene razmakom.*

Pitanje 12.8. *Koju vrednost vraća* `sscanf("1 2","%i%i%i", &a, &b, &c);`

Pitanje 12.9. *Navesti prototip funkcije* `gets`*. Navesti prototip funkcije* `puts`*. Kakva je razlika između funkcija* `gets` *i* `fgets`*? Kakva je razlika između funkcija* `puts` *i* `fputs`*?*

Pitanje 12.10. *U čemu se razlikuju tekstualne i binarne datoteke?*

Pitanje 12.11. *Navesti prototip funkcije* `fopen`*. Koju vrednost vraća ova funkcija ukoliko se datoteka ne može otvoriti?*

Pitanje 12.12. *Navesti prototip standardne bibliotečke funkcije za zatvaranje datoteke. Koju vrednost ona vraća? Navesti bar dva razloga zbog kojih je preporučljivo zatvoriti datoteku čim se završi njeno korišćenje.*

Pitanje 12.13. *Može se desiti da nakon prekida rada programa u toku njegovog izvršavanja, datoteka u koju su u okviru prorama upisani neki podaci funkcijom* `fprintf` *ipak bude prazna. Zašto se to može desiti?*

Pitanje 12.14. *Navesti prototip funkcije kojom se proverava da li dostignut kraj datoteke sa datotečkim pokazivačem* `p`*?*

Pitanje 12.15. *Ukratko opisati funkcije za određivanje mesta u datoteci.*

Pitanje 12.16. *Čemu služe funkcije* `fseek` *i* `fsetpos`*? Da li one isto rade za tekstualne i binarne datoteke?*

Pitanje 12.17. *Šta može biti vrednost argumenta* `offset` *u pozivu funkcije* `fseek` *(*`int fseek(FILE *stream, long offset, int origin)`*) za tekstualnu datoteku?*

Pitanje 12.18. *Navesti prototip funkcije* `main` *sa argumentima.*

Pitanje 12.19. *Kako se, u okviru funkcije* `main`*, može ispisati ime programa koji se izvršava?*

Zadatak 12.4.1. *Napisati funkciju* `main` *koja izračunava i ispisuje zbir svih argumenata navedenih u komandnoj liniji (pretpostaviti da će svi argumenti biti celi brojevi koji se mogu zapisati u okviru tipa* `int`*).* √

Zadatak 12.4.2. *Napisati program koji iz datoteke, čije se ime zadaje kao prvi argument komandne linije, čita prvo dimenziju kvadratne matrice n, a zatim elemente matrice (pretpostavljamo da se u datoteci nalaze brojevi pravilno raspoređeni, odnosno da za dato n, sledi $n \times n$ elemenata matrice). Matricu dinamički alocirati. Nakon toga, na standardni izlaz ispisati redni broj kolone koja ima najveći zbir elemenata. Na primer, za datoteka sa sadržajem:*

```
3
1 2 3
7 3 4
5 3 1
```

program treba da ispiše redni broj 0 *(jer je suma elemenata u nultoj koloni* `1 + 7 + 5 = 13`*, u prvoj* `2 + 3 + 3 = 8`*, u drugoj* `3 + 4 + 1 = 8`*).* √

Zadatak 12.4.3. *Data je datoteka* `apsolventi.txt`*. U svakom redu datoteke nalaze se podaci o apsolventu:* ime *(ne veće od 20 karaktera),* prezime *(ne veće od 20 karaktera),* broj preostalih ispita*. Datoteka je dobro formatirana i broj redova datoteke nije poznat. Potrebno je učitati podatke iz datoteke, odrediti prosečan broj zaostalih ispita i potom ispisati imena i prezimena studenta koji imaju veći broj zaostalih ispita od prosečnog u datoteku čije ime je zadato kao argument komadne linije. NAPOMENA: koristiti strukturu*

```
typedef struct {
     char ime[20];
     char prezime[20];
     unsigned br_ispita;
} APSOLVENT;
```

√

Zadatak 12.4.4. *Napisati funkciju* `unsigned btoi(char s[], unsigned char b)` *koja određuje vrednost zapisa datog neoznačenog broja* `s` *u datoj osnovi* `b`*. Napisati funkciju* `void itob(unsigned n, unsigned char b, char s[])` *koja datu vrednost* `n` *zapisuje u datoj osnovi* `b` *i smešta rezultat u nisku* `s`*. Napisati zatim program koji čita liniju po liniju datoteke koja se zadaje kao prvi argument komandne linije i obrađuje ih sve dok ne naiđe na praznu liniju. Svaka linija sadrži jedan dekadni, oktalni ili heksadekadni broj (zapisan kako se zapisuju konstante u programskom jeziku C). Program za svaki uneti broj u datoteku koja se zadaje kao drugi argument komandne linije ispisuje njegov binarni zapis. Pretpostaviti da će svi uneti brojevi biti u opsegu tipa* `unsigned`*.* √

Dodatak A

Tabela prioriteta operatora

Operator	Opis	Asocijativnost

`()`
`[]`
`.`
`->`
`++ --`
	Poziv funkcije
	indeksni pristup elementu niza
	pristup članu strukture ili unije
	pristup članu strukture ili unije preko pokazivača
	postfiksni inkrement/dekrement
		sleva na desno

`++ --`
`+ -`
`! ~`
`(type)`
`*`
`&`
`sizeof`
	prefiksni inkrement/dekrement
	unarni plus/minus
	logička negacija/bitski komplement
	eksplicitna konverzija tipa (cast)
	dereferenciranje
	referenciranje (adresa)
	veličina u bajtovima
		zdesna na levo

`* / %`	množenje, deljenje, ostatak	sleva na desno
`+ -`	sabiranje i oduzimanje	sleva na desno
`<< >>`	bitsko pomeranje ulevo i udesno	
`< <=` `> >=`	relacije manje, manje jednako relacije veće, veće jednako	sleva na desno
`== !=`	relacija jednako, različito	sleva na desno
`&`	bitska konjunkcija	sleva na desno
`^`	bitska ekskluzivna disjunkcija	sleva na desno
`|`	bitska disjunkcija	sleva na desno
`&&`	logička konjunkcija	sleva na desno
`||`	logička disjunkcija	sleva na desno
`?:`	ternarni uslovni	zdesna na levo

`=`
`+= -=`
`*= /= %=`
`&= ^= |=`
`<<= >>=`
	dodele	zdesna na levo

`,`	spajanje izraza u jedan	sleva na desno

Dodatak B

Rešenja zadataka

Softver savremenih računara

Rešenje zadatka 1.4:
Rešenje se zasniva na algoritmu opisanom u primeru 3.3. Na raspolaganju imamo samo 3 registra. Neka `ax` čuva vrednost N_1, `bx` vrednost N_2, dok registar `cx` privremeno čuva ostale potrebne vrednosti. Neka se na memorijskoj lokaciji 200 čuva rezultujuća vrednost n.

```
mov [200], 0
mov ax, 0
mov bx, 1
petlja:
  cmp ax, bx
  je uvecanje
  mov cx, [100]
  cmp cx, ax
  je kraj
  add ax, 1
  jmp petlja
uvecanje:
  mov cx, [200]
  add cx, 1
  mov [200], cx
  add bx, cx
  add bx, cx
  jmp petlja
kraj:
```

Reprezentacija podataka u računarima

Rešenje zadatka 2.1:
(a) 185 (b) 964 (c) 476
Rešenje zadatka 2.2:
$(11111110)_2$, $(376)_8$, $(FE)_{16}$,
Rešenje zadatka 2.3:

Prevođenjem svake četvorke binarnih cifara zasebno, dobija se $(AB48F5566BAAE293)_{16}$. Prevođenjem svake heksadekadne cifre zasebno dobija se

$(1010\,0011\,1011\,1111\,0100\,0110\,0001\,1100\,1000\,1001\,1011\,1110\,0010\,0011\,1101\,0111)_2$

Rešenje zadatka 2.4:
(a) rgb(53, 167, 220) (b) #FFFF00 (c) Svaki par heksadekadnih cifara je isti (npr. #262626 ili #A0A0A0).

Rešenje zadatka 2.5:
Najmanji broj je 0 ($00\ldots00$), a najveći 2^n-1 ($11\ldots11$). Dakle, (a) od 0 do $2^4-1=15$, (b) od 0 do $2^8-1=255$, (c) od 0 do $2^{16}-1=65535=64K$, (d) od 0 do $2^{24}-1=16\,777\,216=16M$, (e) $2^{32}-1=4\,294\,967\,296=4G$.

Rešenje zadatka 2.6:
(a) 00001100, (b) 01111011, (c) 11111111, (d) ne može da se zapiše

Rešenje zadatka 2.7:
Najmanji broj je -2^{n-1} ($10\ldots00$), a najveći je $2^{n-1}-1$ ($011\ldots11$). Dakle, (a) od -8 do 7, (b) od -128 do 127, (c) -32768 do 32767, (d) od -8388608 do 8388607 i (e) od -2147483648 do 2147483647.

Rešenje zadatka 2.8:
(a) 00001100, (b) 10000101, (c) 00000000, (d) 11101110, (e) 10000000 (f) ne može da se zapiše

Rešenje zadatka 2.9:
Broj 5 se zapisuje kao 000101, broj $-$ kao 111011, zbir kao 000000, a proizvod kao 100111.

Rešenje zadatka 2.10:
(a) -38, (b) 83, (c) -128, (d) -1, (e) 127, (f) 0

Rešenje zadatka 2.11:
Sa n bitova moguće je kodirati ukupno 2^n različitih karaktera. Dakle, najmanji broj bitova je 5.

Rešenje zadatka 2.12:
46 41 4b 55 4c 54 45 54, DISKRETNE

Rešenje zadatka 2.13:
Heksadekadno: 22 50 72 6f 67 72 61 6d 69 72 61 6e 6a 65 20 31 22,

Binarno: 00100010 01010000 01110010 01101111 01100111 01110010 01100001 01001101 01101001 01110010 01100001 01101110 01101010 01100101 00100000 00110001 00100010

Oktalno: 042 120 162 157 147 162 141 115 151 162 141 156 152 145 040 061 042

Dekadno: 34 80 114 111 103 114 97 109 105 114 97 110 106 101 32 49 34

Rešenje zadatka 2.14:

	ASCII	Windows-1250	ISO-8859-5	ISO-8859-2	Unicode (UCS-2)	UTF-8
računarstvo	-	11	-	11	22	12
informatika	11	11	11	11	22	11
ћир-lat	-	-	7	-	14	10
čirчиh	-	-	-	-	12	10

Rešenje zadatka 2.15:
UCS-2: 006b 0072 0075 017e 0069 0107 *UTF-8*: 6b 72 75 c5be 69 c487

Izračunljivost

Rešenje zadatka 3.1:
Predloženi algoritam se zasniva na sledećoj osobini:

$$xy = \underbrace{x + \overbrace{(1 + \ldots + 1)}^{x} + \ldots + \overbrace{(1 + \ldots + 1)}^{x}}_{y}$$

Odgovarajući URM program podrazumeva sledeću početnu konfiguraciju:

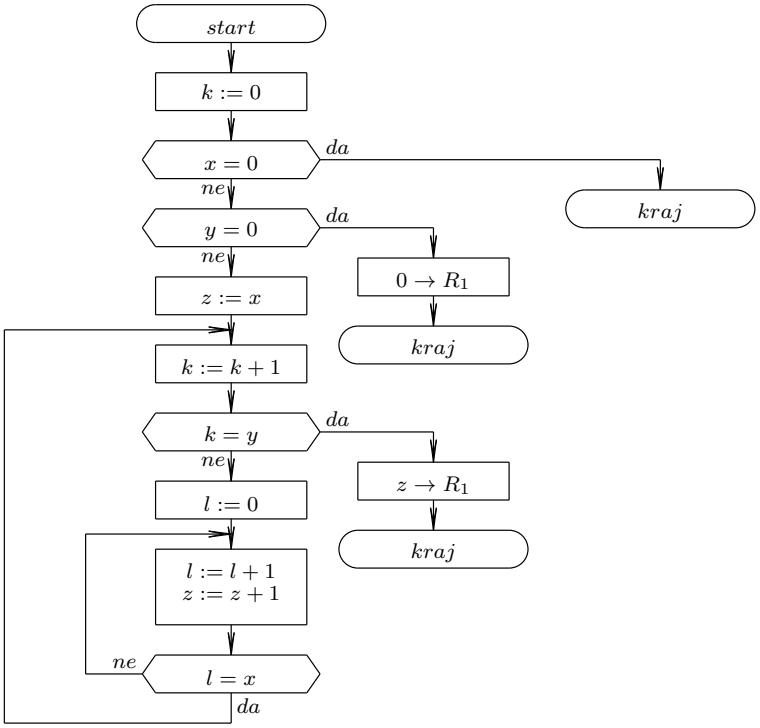
R_1	R_2	R_3	$\ldots$
x	y	0	$\ldots$

i sledeću radnu konfiguraciju:

R_1	R_2	R_3	R_4	R_5	$\ldots$
x	y	z	k	l	$\ldots$

gde k dobija redom vrednosti $0, 1, \ldots, y$, a za svaku od ovih vrednosti l dobija redom vrednosti $0, 1, \ldots, x$.

1.	$J(1, 10, 100)$	ako je $x = 0$, onda kraj
2.	$J(2, 10, 13)$	$y = 0$?
3.	$T(1, 3)$	$z := x$
4.	$S(4)$	$k := k + 1$
5.	$J(4, 2, 11)$	$k = y$?
6.	$Z(5)$	$l := 0$
7.	$S(5)$	$l := l + 1$
8.	$S(3)$	$z := z + 1$
9.	$J(5, 1, 4)$	
10.	$J(1, 1, 7))$	
11.	$T(3, 1)$	$z \to R_1$
12.	$J(1, 1, 100)$	
13.	$Z(1)$	$0 \to R_1$



Rešenje zadatka 3.5:
Predloženi algoritam se zasniva na sledećoj osobini:

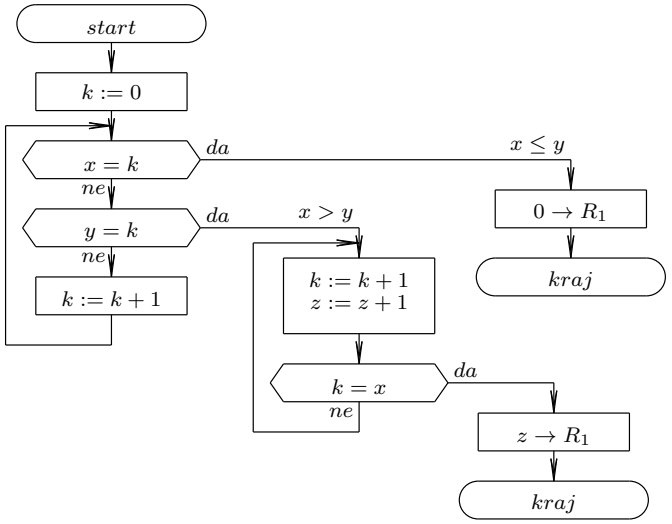
$$z = x - y \Leftrightarrow x = y + z$$

Odgovarajući URM program podrazumeva sledeću početnu konfiguraciju:

R_1	R_2	R_3	...
x	y	0	...

i sledeću radnu konfiguraciju:

R_1	R_2	R_3	R_4	...
x	y	k	z	...



1.	$J(1,3,5)$	$x = k?$
2.	$J(2,3,7)$	$y = k?$
3.	$S(3)$	$k := k + 1$
4.	$J(1,1,1)$	
5.	$Z(1)$	$0 \to R_1$
6.	$J(1,1,100)$	kraj
7.	$S(3)$	$k := k + 1$
8.	$S(4)$	$z := z + 1$
9.	$J(3,1,11)$	$k = x?$
10.	$J(1,1,7)$	
11.	$T(4,1)$	$z \to R_1$

Jezik C - uvod

Rešenje zadatka 5.2.1:

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

int main() {
   double r;
   printf("Unesi poluprecnik kruga: ");
   scanf("%lf", &r);
   assert(r > 0);
   printf("Obim kruga je: %lf\n", 2*r*M_PI);
   printf("Povrsina kruga je: %lf\n", r*r*M_PI);
   return 0;
}
```

Rešenje zadatka 5.2.2:

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

int main() {
   /* Koordinate tacaka A, B, C */
   double xa, ya, xb, yb, xc, yc;
   /* Duzine stranica BC, AC, AB */
   double a, b, c;
   /* Poluobim i povrsina trougla ABC */
   double s, P;

   printf("Unesi koordinate tacke A: \n");
   scanf("%lf%lf", &xa, &ya);
   printf("Unesi koordinate tacke B: \n");
   scanf("%lf%lf", &xb, &yb);
   printf("Unesi koordinate tacke C: \n");
   scanf("%lf%lf", &xc, &yc);
```

```
   /* Izracunavaju se duzine stranice trougla ABC i
      njegova povrsina Heronovim obrascem */
   a = sqrt((xb - xc)*(xb - xc) + (yb - yc)*(yb - yc));
   b = sqrt((xa - xc)*(xa - xc) + (ya - yc)*(ya - yc));
   c = sqrt((xa - xb)*(xa - xb) + (ya - yb)*(ya - yb));

   s = (a + b + c) / 2;
   P = sqrt(s*(s - a)*(s - b)*(s - c));

   printf("Povrsina trougla je: %lf\n", P);

   return 0;
}
```

Rešenje zadatka 5.2.3:

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

int main() {
   /* Duzine stranica trougla */
   double a, b, c;
   /* Velicine uglova trougla */
   double alpha, beta, gamma;

   printf("Unesi duzinu stranice a: ");
   scanf("%lf", &a); assert(a > 0);
   printf("Unesi duzinu stranice b: ");
   scanf("%lf", &b); assert(b > 0);
   printf("Unesi ugao gama (u stepenima): ");
   scanf("%lf", &gamma);

   /* Kosinusnom teoremom izracunavamo duzinu trece stranice i
      preostala dva ugla. Pri tom se vrsi konverzija stepena
      u radijane i obratno. */
   c = sqrt(a*a + b*b - 2*a*b*cos(gamma*M_PI/180.0));

   alpha = acos((b*b + c*c - a*a) / (2*b*c)) / M_PI * 180.0;
   beta = acos((a*a + c*c - b*b) / (2*a*c)) / M_PI * 180.0;

   printf("Duzina stranice c je: %lf\n", c);
   printf("Velicina ugla alfa (u stepenima) je: %lf\n", alpha);
   printf("Velicina ugla beta (u stepenima) je: %lf\n", beta);

   return 0;
}
```

Rešenje zadatka 5.2.4:

```
#include <stdio.h>

int main() {
  double kmh;
  printf("Unesi brzinu u km/h: ");
  scanf("%lf", &kmh);
  printf("Brzina u ms/s je: %lf\n", kmh * 1000.0 / 3600.0);
  return 0;
}
```

Rešenje zadatka 5.2.5:

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

int main() {
   /* Pocetna brzina u m/s, ubrzanje u m/s^2 i vreme u s*/
   double v0, a, t;

   printf("Unesi pocetnu brzinu (u m/s): ");
   scanf("%lf", &v0);
   printf("Unesi ubrzanje (u m/s^2): ");
   scanf("%lf", &a);
   printf("Unesi vreme: ");
   scanf("%lf", &t); assert(t >= 0);

   printf("Trenutna brzina (u m/s) je : %lf\n", v0+a*t);
   printf("Predjeni put (u m) je : %lf\n", v0*t+a*t*t/2);
   return 0;
}
```

Rešenje zadatka 5.2.6:

```
#include <stdio.h>

int main() {
  double a1, a2, a3, a4, a5, a6, a7, a8, a9;
  scanf("%lf%lf%lf", &a1, &a2, &a3);
  scanf("%lf%lf%lf", &a4, &a5, &a6);
  scanf("%lf%lf%lf", &a7, &a8, &a9);
  /* Determinantu ra\v cunamo primenom Sarusovog pravila:
     a1 a2 a3 a1 a2
     a4 a5 a6 a4 a5
     a7 a8 a9 a7 a8
     - glavne dijagonale pozitivno, sporedne negativno
   */
```

```
  printf("%lf\n", a1*a5*a9 + a2*a6*a7 + a3*a4*a8
                  - a3*a5*a7 - a1*a6*a8 - a2*a4*a9);
  return 0;
}
```

Rešenje zadatka 5.2.7:

```
#include <stdio.h>

int main() {
  int a1, a2, a3; /* brojevi */
  int m; /* maksimum brojeva */
  printf("Unesi tri broja: ");
  scanf("%d %d %d", &a1, &a2, &a3);
  m = a1;
  if (a2 > m)
    m = a2;
  if (a3 > m)
    m = a3;
  printf("Maksimum je: %d\n", m);
  return 0;
}
```

Rešenje zadatka 5.2.8:

```
#include <stdio.h>

int main() {
  int a, b, i;
  scanf("%d%d", &a, &b);
  for (i = a; i <= b; i++)
    printf("%d\n", i*i*i);
  return 0;
}
```

Jezik C - izrazi

Rešenje zadatka 6.5.1:

```
#include <stdio.h>

int main() {
  int h, m, s, h1, m1, s1;
  scanf("%d:%d:%d", &h, &m, &s);
  if (h < 0 || h > 23) {
    printf("Neispravno unet sat\n");
    return 1;
  }
```

```
  if (m < 0 || m > 59) {
    printf("Neispravno unet minut\n");
    return 2;
  }
  if (s < 0 || s > 59) {
    printf("Neispravno unet sekund\n");
    return 3;
  }

  s1 = 60 - s;
  m1 = 59 - m;
  h1 = 23 - h;
  if (s1 == 60) {
    s1 = 0;
    m1++;
  }
  if (m1 == 60) {
    m1 = 0;
    h1++;
  }

  printf("%d:%d:%d\n", h1, m1, s1);
  return 0;
}
```

Rešenje zadatka 6.5.2:

```
#include <stdio.h>

int main() {
  double x, y, z;
  scanf("%lf%lf%lf", &x, &y, &z);
  if (x > 0 && y > 0 && z > 0 &&
      x + y > z && x + z > y && y + z > x)
    printf("Trougao postoji\n");
  else
    printf("Trougao ne postoji\n");
  return 0;
}
```

Rešenje zadatka 6.5.3:

```
#include <stdio.h>

int main() {
   /* Broj cija se suma cifara racuna */
   int n;
   /* Cifre broja*/
   int c0, c1, c2, c3;
```

```
   printf("Unesi cetvorocifreni broj: ");
   scanf("%d", &n);
   if (n < 1000 || n >= 10000) {
      printf("Broj nije cetvorocifren\n");
      return 1;
   }

   c0 = n % 10;
   c1 = (n / 10) % 10;
   c2 = (n / 100) % 10;
   c3 = (n / 1000) % 10;

   printf("Suma cifara je: %d\n", c0 + c1 + c2 + c3);
   return 0;
}
```

Rešenje zadatka 6.5.4:

```
#include <stdio.h>

int main() {
  unsigned n, c0, c1; /* broj, poslednja i pretposlednja cifra */
  printf("Unesi broj: ");
  scanf("%d", &n);
  c0 = n % 10;
  c1 = (n / 10) % 10;
  printf("Zamenjene poslednje dve cifre: %u\n",
     (n / 100)*100 + c0*10 + c1);
  return 0;
}
```

Rešenje zadatka 6.5.5:

```
#include <stdio.h>

int main() {
  /*
  (0, 0) -> 1
  (0, 1) -> 2  (1, 0) -> 3
  (2, 0) -> 4  (1, 1) -> 5  (0, 2) -> 6
  (0, 3) -> 7  (1, 2) -> 8  (2, 1) -> 9  (3, 0) -> 10
  ...
  */
  unsigned x, y; /* koordinate */
  unsigned z;    /* pomocna promenljiva */
  unsigned n;    /* redni broj u cik-cak nabrajanju */
  printf("Unesi x i y koordinatu: ");
  scanf("%d%d", &x, &y);
```

```
  z = x + y;
  if (z % 2)
    n = z*(z + 1)/2 + x + 1;
  else
    n = z*(z + 1)/2 + y + 1;
  printf("Redni broj u cik-cak nabrajanju je: %u\n", n);
  return 0;
}
```

Rešenje zadatka 6.5.6:

```
#include <stdio.h>

int main() {
  double A, B;
  printf("Unesi koeficijente A i B jednacine A*X + B = 0: ");
  scanf("%lf%lf", &A, &B);
  if (A != 0)
    printf("Jednacina ima jedinstveno resenje: %lf\n", -B/A);
  else if (B == 0)
    printf("Svaki realan broj zadovoljava jednacinu\n");
  else
    printf("Jednacina nema resenja\n");
  return 0;
}
```

Rešenje zadatka 6.5.7:

```
#include <stdio.h>

int main() {
  double a1, b1, c1, a2, b2, c2; /* koeficijenti sistema */
  double Dx, Dy, D; /* determinante */
  scanf("%lf%lf%lf", &a1, &b1, &c1);
  scanf("%lf%lf%lf", &a2, &b2, &c2);
  /* Sistem resavamo primenom Kramerovog pravila */
  /* Determinanta sistema:
        a1 b1
        a2 b2
   */
  D = a1*b2 - b1*a2;
  /* Determinanta promenljive x:
        c1 b1
        c2 b2
   */
  Dx = c1*b2 - b1*c2;
  /* Determinanta promenljive y:
        a1 c1
        a2 c2
```

```
   */
  Dy = a1*c2 - c1*a2;
  if (D != 0)
    printf("x = %lf\ny = %lf\n", Dx / D, Dy / D);
  else if (Dx == 0 && Dy == 0)
    printf("Sistem ima beskonacno mnogo resenja\n");
  else
    printf("Sistem nema resenja\n");

  return 0;
}
```

Rešenje zadatka 6.5.8:

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

int main() {
  float a, b, c; /* koeficijenti */
  float D;       /* diskriminanta */
  printf("Unesi koeficijente a, b, c"
         " kvadratne jednacine ax^2 + bx + c = 0: ");
  scanf("%f%f%f", &a, &b, &c);
  assert(a != 0);
  D = b*b - 4*a*c;
  if (D > 0) {
    float sqrtD = sqrt(D);
    printf("Realna resenja su: %f i %f\n",
           (-b + sqrtD) / (2*a), (-b - sqrtD) / (2*a));
  } else if (D == 0) {
    printf("Jedinstveno realno resenje je: %f\n",
           -b/(2*a));
  } else {
    printf("Jednacina nema realnih resenja\n");
  }
  return 0;
}
```

Rešenje zadatka 6.5.9:

```
#include <stdio.h>

int main() {
  unsigned n; /* broj koji se ispituje */
  unsigned d; /* kandidat za delioca */
  scanf("%u", &n);
  for (d = 1; d <= n; d++)
    if (n % d == 0)
```

```
      printf("%u\n", d);
  return 0;
}
```

Rešenje zadatka 6.5.10:

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

int main() {
  int c;
  while ((c = getchar()) != '.' && c != ',' && c != EOF)
    putchar(toupper(c));
}
```

Jezik C - nizovi

Rešenje zadatka 6.7.1:

```
#include <stdio.h>

#define MAX 1000

int main() {
  int a[MAX];
  int n, i, max, zbir;
  scanf("%d", &n);
  if (n >= MAX) {
    printf("Previse elemenata\n");
    return 1;
  }
  /* Ucitavanje niza */
  for (i = 0; i < n; i++)
    scanf("%d", &a[i]);

  /* Ispisivanje niza unatrag */
  for (i = n - 1; i >= 0; i--)
    printf("%d ", a[i]);
  printf("\n");

  /* Odredjivanje i ispis zbira */
  zbir = 0;
  for (i = 0; i < n; i++)
    zbir += a[i];
  printf("zbir = %d\n", zbir);

  /* Odredjivanje i ispis maksimuma */
  max = a[0];
  for (i = 1; i < n; i++)
```

```
    if (a[i] > max)
      max = a[i];
  printf("max = %d\n", max);

  return 0;
}
```

Rešenje zadatka 6.7.2:

```
#include <stdio.h>

int main() {
  int b[10], i;

  for (i = 0; i < 10; i++)
    b[i] = 0;

  while ((c = getchar()) != '.')
    if ('0'<=c && c<='9')
       b[c - '0']++;

  for (i = 0; i < 10; i++)
    printf("%d ", b[i]);

  return 0;
}
```

Rešenje zadatka 6.7.3:

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

/* Pretprocesorska direktiva kojom se proverava da li je godina
   prestupna. Godina je prestupna ako je deljiva sa 4, a ne i sa 100
   ili je deljiva sa 400. */
#define prestupna(g) (((g) % 4 == 0 && (g) % 100 != 0) || ((g) % 400 == 0))

int main() {
  int d, m, g, b, M;
  /* Broj dana po mesecima. Niz je napravljen tako da se broj dana za
     mesec m moze ocitati sa br_dana[m].
   */
  int br_dana[] = {-1, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
  printf("Unesi datum u formatu d/m/g: ");
  assert(scanf("%d/%d/%d", &d, &m, &g) == 3);
  /* Provera ispravnosti datuma */
  if (g < 0) {
    printf("Pogresna godina\n");
    return 1;
```

```
  }
  if (m < 1 || m > 12) {
    printf("Pogresan mesec\n");
    return 1;
  }
  if (d < 1 || d > (m == 2 && prestupna(g) ? 29 : br_dana[m])) {
    printf("Pogresan dan\n");
    return 1;
  }

  b = 0;
  /* Broj dana u prethodnim mesecima */
  for (M = 1; M < m; M++)
    b += br_dana[M];
  /* Broj dana u tekucem mesecu */
  b += d;
  /* Eventualno dodati i 29. 2. */
  if (prestupna(g) && m > 2)
    b++;
  printf("%d\n", b);
}
```

Rešenje zadatka 6.7.4:

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

int main() {
  int a[100], n, k, m;
  /* Ucitavamo broj redova trougla */
  scanf("%d", &m);
  for (n = 0; n < m; n++) {
    /* Azuriramo tekuci red trougla */
    /* (n n) = 1 */
    a[n] = 1;
    for (k = n-1; k > 0; k--)
      /* (n k) = (n-1 k) + (n-1 k-1) */
      a[k] = a[k] + a[k-1];
    /* (n 0) = 1 */
    /* a[0] = 1; -- ovo vec vazi */

    /* Ispisujemo tekuci red */
    for (k = 0; k <= n; k++)
      printf("%d ", a[k]);
    printf("\n");
  }
}
```

Rešenje zadatka 6.7.5:

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

int main() {
  int n, A[10][10], i, j, s;
  /* ucitavamo matricu */
  scanf("%d", &n); assert(0 < n && n <= 10);
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      scanf("%d", &A[i][j]);

  /* sumiramo elemente glavne dijagonale */
  s = 0;
  for (i = 0; i < n; i++)
    s += A[i][i];
  printf("Suma elemenata glavne dijagonale je: %d\n", s);

  /* sumiramo elemente iznad sporedne dijagonale */
  s = 0;
  for (i = 0; i < n; i++)
    for (j = 0; i + j + 1 < n; j++)
      s += A[i][j];
  printf("Suma elemenata iznad sporedne dijagonale je: %d\n", s);

  return 0;
}
```

Rešenje zadatka 6.7.6:

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

int main() {
  int n, A[100][100], i, j, dt;
  /* ucitavamo matricu */
  scanf("%d", &n); assert(0 < n && n <= 100);
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      scanf("%d", &A[i][j]);
  /* proveravamo da li je matrica donjetrougaona */
  dt = 1;
  for (i = 0; i < n && dt; i++)
    for (j = i+1; j < n && dt; j++)
      if (A[i][j] != 0)
        dt = 0;
  printf("Matrica %s donjetrougaona\n", dt ? "je" : "nije");
  return 0;
}
```

Jezik C - korisnički definisani tipovi

Rešenje zadatka 6.8.1:

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

/* Struktura za reprezentovanje kompleksnog broja */
typedef struct complex {
  double Re, Im;
} COMPLEX;

int main() {
  /* Najveci broj */
  COMPLEX  max_z;
  /* Najveci moduo */
  double max_m = 0.0;

  int i;
  for (i = 0; i < 10; i++) {
    /* Tekuci broj */
    COMPLEX z;
    /* Moduo tekuceg broja */
    double z_m;
    /* Ucitavanje */
    scanf("%lf%lf", &z.Re, &z.Im);
    /* Racunanje modula */
    z_m = sqrt(z.Re*z.Re + z.Im*z.Im);
    /* Azuriranje najveceg */
    if (z_m > max_m) {
      max_z = z;
      max_m = z_m;
    }
  }
  printf("%lf + i*%lf\n", max_z.Re, max_z.Im);
  return 0;
}
```

Rešenje zadatka 6.8.2:

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

int main() {
  enum dani {PON = 1, UTO, SRE, CET, PET, SUB, NED};
  int dan;
  scanf("%d", &dan);
  assert(1 <= dan && dan <= 7);
  if (dan == SUB || dan == NED)
```

```
    printf("Vikend\n");
  else
    printf("Radni dan\n");
  return 0;
}
```

Jezik C - naredbe

Rešenje zadatka 7.1:

```
#include <stdio.h>

int main() {
  int n, i;
  printf("Unesi gornju granicu: ");
  scanf("%d", &n);
  for (i = 1; i < n; i += 2)
    printf("%d ", i);
  printf("\n");
  return 0;
}
```

Rešenje zadatka 7.2:

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

int main() {
  int N = 100;
  double l = 0.0, d = 2*M_PI;
  double h = (d - l) / (N - 1);
  double x;
  printf(" x      sin(x)\n");
  for (x = l; x <= d; x += h)
    printf("%4.2lf   %7.4lf\n", x, sin(x));
  return 0;
}
```

Rešenje zadatka 7.3:

```
#include <stdio.h>

int main() {
  double x, s;
  unsigned n, i;
  printf("Unesi broj x: ");
  scanf("%lf", &x);
  printf("Unesi izlozilac n: ");
```

```
  scanf("%d", &n);
  for (s = 1.0, i = 0; i < n; i++)
    s *= x;
  printf("x^n = %lf\n", s);
  return 0;
}
```

Rešenje zadatka 7.4:

```
#include <stdio.h>

#define N 4

int main() {
  int i, j;

  /* Deo (a): */

  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++)
      putchar('*');
    putchar('\n');
  }

  printf("--------------\n");

  /* Deo (b): */
  for (i = 0; i < N; i++) {
    for (j = 0; j < N - i; j++)
      putchar('*');
    putchar('\n');
  }

  printf("--------------\n");

  /* Deo (c): */
  for (i = 0; i < N; i++) {
    for (j = 0; j < i + 1; j++)
      putchar('*');
    putchar('\n');
  }

  printf("--------------\n");

  /* Deo (d): */
  for (i = 0; i < N; i++) {
    for (j = 0; j < i; j++)
      putchar(' ');
    for (j = 0; j < N - i; j++)
```

```
      putchar('*');
    putchar('\n');
  }

  printf("--------------\n");

  /* Deo (e): */
  for (i = 0; i < N; i++) {
    for (j = 0; j < N - i - 1; j++)
      putchar(' ');
    for (j = 0; j < i + 1; j++)
      putchar('*');
    putchar('\n');
  }

  printf("--------------\n");

  /* Deo (f): */
  for (i = 0; i < N; i++) {
    for (j = 0; j < i; j++)
      putchar(' ');
    for (j = 0; j < N - i - 1; j++) {
      putchar('*'); putchar(' ');
    }
    putchar('*');
    putchar('\n');
  }

  printf("--------------\n");

  /* Deo (g): */
  for (i = 0; i < N; i++) {
    for (j = 0; j < N - i - 1; j++)
      putchar(' ');
    for (j = 0; j < i; j++) {
      putchar('*'); putchar(' ');
    }
    putchar('*');
    putchar('\n');
  }

  return 0;

}
```

Rešenje zadatka 7.5:

```
#include <stdio.h>
/* Delioci se dodaju u parovima:  ako je n deljiv sa d, deljiv je i sa n/d.
```

```
   Pretraga se vrsi do korena iz n.
   Ako je n potpun kvadrat, koren se sabira samo jednom.
   Napomena: za jako velike vrednosti n, d*d moze da prekoraci.
*/
int main() {
  unsigned n; /* broj koji se ispituje */
  unsigned s; /* suma delioca */
  unsigned d; /* kandidat za delioca */
  scanf("%u", &n);
  for (s = 0, d = 1; d*d < n; d++)
    if (n % d == 0)
      s += d + n / d;
  if (d * d == n)
    s += d;
  printf("Suma: %u\n", s);
  return 0;
}
```

```
#include <stdio.h>

/* Suma delilaca se izracunava razlaganjem na proste cinioce.
   Koristi se cinjenica da je suma delilaca multiplikativna funkcija,
   tj. da je za uzajamno proste brojeve a i b sd(x*y) = sd(x) * sd(y).
   Takodje, za prost broj p, sd(d^n) = 1 + d + d^2 + ... + d^n. Zato,
   ako je n = p1^k1*...*pm^km razlaganje na proste cinioce, tada je
   sd(n) = (1+p1+...+p1^k1)*...*(1+pm+...+pm^km).
*/
int main() {
    unsigned n; /* broj koji se ispituje */
    unsigned s; /* suma delilaca */
    unsigned d; /* tekuci cinilac */
    scanf("%u", &n);
    for (s = 1, d = 2; d*d <= n; d++) {
      unsigned p = 1;
      while (n % d == 0) {
        /* u ovoj tacki d je sigurno prost */
        p = 1 + d*p;
        n = n / d;
      }
      s *= p;
    }
    if (n > 1) s *= (1 + n);
    printf("Suma: %u\n", s);
    return 0;
}
```

Rešenje zadatka 7.6:

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

```
/* Teorema: Ako broj ima pravog delioca koji je veci od korena iz n,
   onda ima delioca koji je manji od korena iz n.
   Dokaz: Ako je d <= sqrt(n) delilac broja n, onda je n/d >= sqrt(n) takodje
   delilac.
   Zato je dovoljno proveravati delioce samo od broja 2 do broja sqrt(n).
   Uslov d <= sqrt(n) menjamo sa d*d <= n, da bismo izbegli rad sa realnim
   brojevima (za jako velike vrednosti n, ovo moze da dovede do prekoracenja)
*/
int main() {
  unsigned n, d;
  int prost;
  scanf("%d", &n);
  for (prost = 1, d = 2; prost && d*d <= n; d++)
    if (n % d == 0)
      prost = 0;
  printf("Broj je %s\n", prost ? "prost" : "slozen");
  return 0;
}
```

Rešenje zadatka 7.7:

```
#include <stdio.h>

int main() {
  unsigned n, d;
  /* Ucitavamo broj */
  scanf("%u", &n);

  /* Prvi kandidat za prost cinioc je 2 */
  d = 2;
  /* Broj n delimo sa jednim po jednim njegovim ciniocem.  Postupak se
     zavrsava kada zavrsimo sa svim ciniocima i n svedemo na 1 */
  while (n > 1) {
    /* Broj je deljiv sa d - dakle, d je prost cinioc. Zaista, ako d
       ne bi bio prost, bio bi deljiv sa nekim svojim ciniocem d' koji
       je manji od njega. Time bi i n bio deljiv sa d'. No, kada se d'
       uvecavalo (a moralo je biti uvecano da bi se stiglo do d) n
       nije bio deljiv njime, sto je kontradikcija. */
    if (n % d == 0) {
      printf("%u\n", d);
      n /= d;
    } else
      /* Broj nije deljiv sa d, pa prelazimo na narednog kandidata */
      d++;
  }
  return 0;
}
```

Rešenje zadatka 7.8:

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

int main() {
  int a; /* broj koji se unosi */
  int n = 0; /* broj unetih brojeva */
  int s = 0; /* zbir unetih brojeva */
  int p = 1; /* proizvod unetih brojeva */
  int min = INT_MAX; /* minimum unetih brojeva */
  int max = INT_MIN; /* maksimum unetih brojeva */
  double sr = 0; /* zbir reciprocnih vrednosti */
  while (1) {
    scanf("%d", &a);
    if (a == 0) break;
    n++;
    s += a;
    p *= a;
    if (a < min) min = a;
    if (a > max) max = a;
    sr += 1.0/a;
  }
  if (n == 0) {
    printf("Nije unet nijedan broj\n");
    return 1;
  }
  printf("broj: %d\n", n);
  printf("zbir: %d\n", s);
  printf("proizvod: %d\n", p);
  printf("minimum: %d\n", min);
  printf("maksimum: %d\n", max);
  printf("aritmeticka sredina: %f\n", (double)s / (double)n);
  printf("geometrijska sredina: %f\n", pow(p, 1.0/n));
  printf("harmonijska sredina: %f\n", n / sr);

  return 0;
}
```

Rešenje zadatka 7.9:

```
#include <stdio.h>

int main() {
  int ts = 0; /* Tekuca serija */
  int ns = 0; /* Najduza serija */
  int pb, tb; /* Prethodni i tekuci broj */
```

```
  scanf("%d", &pb);
  if (pb != 0) {
    ts = ns = 1;
    while(1) {
      scanf("%d", &tb);
      if (tb == 0) break; /* Prekidamo kada je uneta 0 */

      /* Da li je tekuci broj jednak prethodnom? */
      if (tb == pb)
        /* Ako jeste, nastavlja se tekuca serija */
        ts++;
      else
        /* Inace, krenula je nova serija */
        ts = 1;

      /* Azuriranje najduze serije */
      if (ts > ns)
        ns = ts;

      /* Azuriranje prethodnog broja */
      pb = tb;
    }
  }
  /* Ispis rezultata */
  printf("%d\n", ns);
  return 0;
}
```

Rešenje zadatka 7.10:

```
#include <stdio.h>

int main() {
/* Izracunava se ceo deo korena unetog broja x.  Trazi se interval
   oblika [n^2, (n+1)^2] = [N1, N2] tako da mu x pripada i tada je n
   ceo deo korena iz x.
*/
  unsigned x;
  unsigned N1, N2, n;

  /* Ucitava se broj x */
  scanf("%d", &x);

  /* Krece se od intervala [0, 1] */
  n = 0; N1 = 0; N2 = 1;
  while (x != N1) {
    N1++;
    if (N1 == N2) {
      /* Prelazi se na sledeci interval */
```

```
      unsigned  l;
      /* Uvecava se malo n */
      n++;
      /* N2 je potrebno uvecati za 2*n+1 */
      N2++;
      l = 0;
      do {
        l++;
        N2 += 2;
      } while (l != n);
    }
  }
  /* Ispis rezultata */
  printf("%d\n", n);
}
```

Rešenje zadatka 7.11:

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

#define EPS 0.0001
int main() {
  double xp, x, a;
  printf("Unesite broj: ");
  scanf("%lf", &a);
  x = 1.0;
  do {
    xp = x;
    x = xp - (xp * xp - a) / (2.0 * xp);
    printf("%lf\n", x);
  } while (fabs(x - xp) >= EPS);
  printf("%lf\n", x);
  return 0;
}
```

Rešenje zadatka 7.12:

```
#include <stdio.h>

int main() {
  unsigned fpp = 1, fp = 1, k;
  scanf("%d", &k);
  if (k == 0) return 0;
  printf("%d\n", fpp);
  if (k == 1) return 0;
  printf("%d\n", fp);
  k -= 2;
  while (k > 0) {
```

```
    unsigned f = fpp + fp;
    printf("%d\n", f);
    fpp = fp; fp = f;
    k--;
  }
  return 0;
}
```

Rešenje zadatka 7.13:

```
#include <stdio.h>

int main() {
  unsigned n;
  printf("Unesi broj: ");
  scanf("%u", &n);
  do {
    /* Poslednja cifra broja */
    printf("%u\n", n % 10);
    /* Brisemo poslednju cifru broja */
    n /= 10;
  } while (n > 0);
  return 0;
}
```

```
#include <stdio.h>

int main() {
  unsigned n, suma = 0;
  printf("Unesi broj: ");
  scanf("%u", &n);
  do {
    suma += n % 10;
    n /= 10;
  } while (n > 0);
  printf("Suma cifara broja je: %u\n", suma);
  return 0;
}
```

Rešenje zadatka 7.14:

```
#include <stdio.h>

int main() {
  unsigned n1, n2, tmp;
  scanf("%d", &n1);
  scanf("%d", &n2);

  /* Da bi se izvrsilo nadovezivanje, n1 se mnozi sa 10^k,
```

```
     gde je k broj cifara broja n2 i zatim se dodaje n2 */

  /* posto ce nam n2 kasnije trebati, kopiramo ga u pomocnu promenljivu */
  tmp = n2;
  /* Dok ima cifara broja tmp */
  while (tmp > 0) {
    /* Mnozimo n1 sa 10 */
    n1 *= 10;
    /* Uklanjamo poslednju cifru broja tmp */
    tmp /= 10;
  }
  /* Uvecavamo n1 za n2 */
  n1 += n2;
  printf("%d\n", n1);
  return 0;
}
```

Rešenje zadatka 7.15:

```
#include <stdio.h>

int main() {
  unsigned n;     /* polazni broj */
  unsigned o = 0; /* obrnuti broj */
  scanf("%d", &n);
  do {
    /* poslednja cifra broja n se uklanja iz broja n i
       dodaje na kraj broja o */
    o = 10*o + n % 10;
    n /= 10;
  } while (n > 0);
  printf("%d\n", o);
  return 0;
}
```

Rešenje zadatka 7.16:

```
#include <stdio.h>

int main() {
  /* Broj koji se obradjuje */
  unsigned n;
  /* Rezultat i 10^k gde je k broj cifara rezultata (ovaj stepen je
     potreban zbog dodavanja cifara na pocetak rezutlata) */
  unsigned r = 0, s = 1;
  /* Ucitavamo broj */
  scanf("%u", &n);
  while (n > 0) {
    /* Uklanjamo mu poslednju cifru */
```

```
    unsigned c = n % 10;
    n /= 10;
    if (c % 2 == 0) {
      /* Ako je parna, dodajemo je kao prvu cifru rezultata */
      r = c * s + r;
      s *= 10;
    }
  }
  /* Ispis rezultata */
  printf("%u\n", r);
  return 0;
}
```

Rešenje zadatka 7.17:

```
#include <stdio.h>
int main() {
  unsigned n, s = 1;
  unsigned char p, c;
  scanf("%u%hhu%hhu", &n, &p, &c);

  /* s = 10^p */
  while (p > 0) {
    s *= 10;
    p--;
  }

  /* Broj se razdvaja na prvih p cifara (n/s) i poslednjih p cifara
     (n%s). Prve cifre se pomeraju za jedno mesto ulevo (n/s)*s*10,
     umece se cifra (c*s) i dodaju poslednje cifre (n%s). */
  printf("%d\n", (n/s)*s*10 + c*s + n%s);
  return 0;
}
```

Rešenje zadatka 7.18:

```
/* Razmena prve i poslednje cifre */
#include <stdio.h>

int main() {
  unsigned n, a, tmp, s, poc, sred, kraj;
  scanf("%u", &n);

  /* Jednocifreni brojevi se posebno obradjuju */
  if (n < 10) {
    printf("%u\n", n);
    return 0;
  }
```

```
  /* Odredjuje se s = 10^k gde je k+1 broj cifara broja n.  Koristi se
     pomocna promenljiva tmp kako bi se ocuvala vrednost n.
   */
  tmp = n; s = 1;
  while (tmp >= 10) {
    s *= 10;
    tmp /= 10;
  }

  /* Prva cifra */
  poc = n / s;
  /* Cifre izmedju prve i poslednje */
  sred = (n % s) / 10;
  /* Poslednja cifra */
  kraj = n % 10;

  /* Gradi se i stampa rezultat */
  printf("%u\n", s*kraj + 10 * sred + poc);

  return 0;

}

/* Rotiranje ulevo */
#include <stdio.h>

int main() {
  unsigned n, tmp, s;
  scanf("%u", &n);

  /* Odredjuje se s = 10^k gde je k+1 broj cifara broja n.  Koristi se
     pomocna promenljiva tmp kako bi se ocuvala vrednost n.
   */
  tmp = n; s = 1;
  while (tmp >= 10) {
    s *= 10;
    tmp /= 10;
  }

  /* Gradi se i ispisuje rezultat tako sto se poslednja cifra broja n
     stavlja ispred ostalih cifara broja n. */
  printf("%u\n", s * (n % 10) + n / 10);

  return 0;
}

/* Rotiranje udesno */
#include <stdio.h>

int main() {
```

```
  unsigned n, tmp, s;
  scanf("%u", &n);

  /* Odredjuje se s = 10^k gde je k+1 broj cifara broja n.  Koristi se
     pomocna promenljiva tmp kako bi se ocuvala vrednost n.
   */
  tmp = n; s = 1;
  while (tmp >= 10) {
    s *= 10;
    tmp /= 10;
  }

  /* Gradi se i ispisuje rezultat tako sto se prva cifra broja n
     stavlja iza ostalih cifara broja n. */
  printf("%u\n", n % s * 10 + n / s);

  return 0;
}
```

Rešenje zadatka 7.19:

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

int main() {
  int d, m, g, dm;

  /* Ucitavamo datum i proveravamo da je ispravno ucitan */
  assert(scanf("%d/%d/%d", &d, &m, &g) == 3);

  /* Vrsimo analizu meseca */
  switch(m) {
    /* Meseci sa 31 danom */
  case 1: case 3: case 5: case 7: case 8: case 10: case 12:
    dm = 31;
    break;
    /* Meseci sa 30 dana */
  case 4: case 6: case 9: case 11:
    dm = 30;
    break;
    /* Februar */
  case 2:
    /* Proverava se da li je godina prestupna */
    dm = (g % 4 == 0 && g % 100 != 0 || g % 400 == 0) ? 29 : 28;
    break;
  default:
    printf("Pogresan mesec\n");
    return 1;
  }
```

```
  /* Provera dana */
  if (d < 1 || d > dm) {
    printf("Pogresan dan\n");
    return 1;
  }
  /* Provera godine */
  if (g < 0) {
    printf("Pogresna godina\n");
    return 1;
  }
  printf("Uneti datum je korektan\n");
  return 0;
}
```

Rešenje zadatka 7.20:

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

int main() {
  int x; /* tekuci broj */
  int m; /* najmanji prethodno unet broj */
  int n; /* broj unetih brojeva */
  int s; /* suma unetih brojeva */

  /* Prvi uneti broj mora biti specificno obradjen jer nema
     prethodnih brojeva kako bi se odredio njihov minimum. Po uslovu
     zadatka, uvek ga ispisujemo.
  */
  scanf("%d", &x);
  printf("manji od minimuma: %d\n", x);
  m = x;
  /* Za sredinu prethodnih se uzima da je 0.0 */
  if (x > 0.0)
    printf("Veci od proseka prethodnih: %d\n", x);
  /* Azuriramo prosek prethodnih */
  s = x; n = 1;

  /* Obradjujemo ostale brojeve dok se ne pojavi 0 */
  while (x != 0) {
    scanf("%d", &x);
    if (x < m) {
      printf("Manji od minimuma: %d\n", x);
      /* Azuriramo vrednost minimuma */
      m = x;
    }
    if (x > s / n)
      printf("Veci od proseka prethodnih: %d\n", x);
```

```
    /* Azuriramo prosek prethodnih */
    s += x; n++;
  }

  return 0;
}
```

Rešenje zadatka 7.21:

```
#include <stdio.h>

int main() {
  unsigned a[1000], n, i, j;

  scanf("%d", &n);

  /* Gradimo niz sve do pozicije n */
  a[0] = 0;
  for (i = 1; i < n; i = j)
    /* Kopiramo prefiks uvecavajuci elemente za 1, vodeci racuna da ne
       predjemo granicu */
    for (j = i; j < i+i && j < n; j++)
      a[j] = a[j - i] + 1;

  /* Stampamo niz */
  for (i = 0; i < n; i++)
    printf("%d ", a[i]);
  printf("\n");

  return 0;
}
```

Rešenje zadatka 7.22:

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

int main() {
  int a[100], b[100]; /* Ulazni nizovi */
  int na, nb; /* Njihov broj elemenata */
  int p[100], u[200], r[100]; /* Presek, unija, razlika */
  int np, nu, nr; /* Njihov broj elemenata */
  int i, j, k; /* Pomocne promenljive */

  /* Ucitavamo niz a proveravajuci da se uneti elementi ne ponavljaju */
  printf("Unesi broj elemenata niza a: ");
  scanf("%d", &na);
  for (i = 0; i < na; i++) {
    scanf("%d", &a[i]);
```

```
  for (j = 0; j < i; j++)
    assert(a[i] != a[j]);
}

/* Ucitavamo niz b proveravajuci da se uneti elementi ne ponavljaju */
printf("Unesi broj elemenata niza b: ");
scanf("%d", &nb);
for (i = 0; i < nb; i++) {
  scanf("%d", &b[i]);
  for (j = 0; j < i; j++)
    assert(b[i] != b[j]);
}

/* ------------------------------------------------------ */
/* presek - u niz p kopiramo sve elemente niza a koji se javljaju u
   nizu b */
np = 0;
for (i = 0; i < na; i++) {
  for (j = 0; j < nb; j++) {
    /* Ako je element a[i] pronadjen u nizu b dodajemo ga u presek p
       i prekidamo pretragu */
    if (a[i] == b[j]) {
      p[np++] = a[i];
      break;
    }
  }
}
/* Ispisujemo elemente preseka */
printf("Presek: ");
for (i = 0; i < np; i++)
  printf("%d ", p[i]);
printf("\n");


/* ------------------------------------------------------ */
/* unija - u niz u kopiramo sve elemente niza a i za njima sve
   elemente niza b koji se ne javljaju u a */
/* Kopiramo niz a u niz u */
nu = 0;
for (i = 0; i < na; i++)
  u[nu++] = a[i];
for (i = 0; i < nb; i++) {
  /* Trazimo b[i] u nizu a */
  for (j = 0; j < na; j++)
    if (b[i] == a[j])
      break;
  /* Ako je petlja stigla do kraja, znaci da niz a ne sadrzi b[i] pa
     ga dodajemo u uniju u */
  if (j == na)
    u[nu++] = b[i];
```

```
  }
  /* Ispisujemo elemente unije */
  printf("Unija: ");
  for (i = 0; i < nu; i++)
    printf("%d ", u[i]);
  printf("\n");


  /* ------------------------------------------------------ */
  /* razlika - u niz r kopiramo sve elemente niza a koji se ne
     javljaju u nizu b */
  nr = 0;
  for (i = 0; i < na; i++) {
    /* Trazimo a[i] u nizu b */
    for (j = 0; j < nb; j++)
      if (a[i] == b[j])
        break;
    /* Ako je petlja stigla do kraja, znaci da niz b ne sadrzi a[i] pa
       ga dodajemo u razliku r */
    if (j == nb)
      r[nr++] = a[i];
  }
  /* Ispisujemo elemente razlike */
  printf("Razlika: ");
  for (i = 0; i < nr; i++)
    printf("%d ", r[i]);
  printf("\n");

  return 0;
}
```

Rešenje zadatka 7.23:

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

int main() {
  /* Rec koja se proverava */
  char s[] = "anavolimilovana";
  /* Bulovska promenljiva koja cuva rezultat */
  int palindrom = 1;

  /* Obilazimo nisku paralelno sa dva kraja sve dok se pozicije ne
     mimoidju ili dok ne naidjemo na prvu razliku */
  int i, j;
  for (i = 0, j = strlen(s) - 1; i < j && palindrom; i++, j--)
    if (s[i] != s[j])
      palindrom = 0;
```

```
  /* Ispisujemo rezultat */
  if (palindrom)
    printf("Rec %s je palindrom\n", s);
  else
    printf("Rec %s nije palindrom\n", s);
}
```

Rešenje zadatka 7.24:

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

int main() {
  /* Niska koja se obrce */
  char s[] = "Zdravo";

  /* Nisku obilazimo paralelno sa dva kraja sve dok se pozicije ne
     susretnu, razmenjujuci karaktere */
  int i, j;
  for (i = 0, j = strlen(s)-1; i<j; i++, j--) {
    int tmp = s[i];
    s[i] = s[j];
    s[j] = tmp;
  }

  /* Ispisujemo obrnutu nisku */
  printf("%s\n", s);
}
```

Rešenje zadatka 7.25:

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

#define MAX 100

int main() {
  unsigned n, i, j, k;
  int m[MAX][MAX];

  /* Ucitavanje matrice */
  scanf("%u", &n);
  assert(n < MAX);
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      scanf("%d", &m[i][j]);

  /* Ispisujemo dijagonale - na svakoj je i+j konstantno */
  for (k = 0; k <= 2*n-2; k++) {
```

```
    for (i = 0; i <= k; i++)
      if (i < n && k-i < n)
        printf("%d ", m[i][k-i]);
    putchar('\n');
  }

  return 0;
}
```

Rešenje zadatka 7.26:

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

#define MAX 100
int main() {
  unsigned m, n, i, j;
  int a[MAX][MAX];

  /* Ucitavamo matricu */
  scanf("%d%d", &m, &n);
  assert(m < MAX && n < MAX);
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
      scanf("%d", &a[i][j]);

  /* Proveravamo sva polja */
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++) {
      /* Odredjujemo zbir susednih elemenata polja (i, j) */
      int k, l;
      int s = 0;
      for (k = -1; k <= 1; k++)
        for (l = -1; l <= 1; l++) {
          if (k == 0 && l == 0)
            continue;
          if (0 <= i+k && i+k < m && 0 <= j+l && j+l < n)
            s += a[i+k][j+l];
        }

      /* Proveravamo da li je zbir jednak vrednosti na polju (i, j) */
      if (s == a[i][j])
        printf("%u %u: %d\n", i, j, a[i][j]);
    }

  return 0;
}
```

Jezik C - funkcije

Rešenje zadatka 8.10.1:

```
#include <stdio.h>

int prost(unsigned n) {
  unsigned d;
  /* Proveravaju se delioci do korena i prekida se ako se
     pronadje delioc */
  for (d = 2; d * d <= n; d++)
    if (n % d == 0)
      return 0;
  /* Ako delioc nije pronadjen, broj je prost */
  return 1;
}

int main() {
  /* Ucitava se broj i proverava da li je prost */
  unsigned n;
  scanf("%u", &n);
  printf(prost(n) ? "Prost\n" : "Slozen\n");
}
```

Rešenje zadatka 8.10.2:

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

double rastojanje(double x1, double y1, double x2, double y2) {
  double dx = x2 - x1, dy = y2 - y1;
  return sqrt(dx*dx + dy*dy);
}

int main() {
  double xa, ya, xb, yb, xc, yc;
  scanf("%lf%lf", &xa, &ya);
  scanf("%lf%lf", &xb, &yb);
  scanf("%lf%lf", &xc, &yc);
  double a = rastojanje(xb, yb, xc, yc);
  double b = rastojanje(xa, ya, xc, yc);
  double c = rastojanje(xa, ya, xb, yb);
  double s = (a + b + c) / 2.0;
  double P = sqrt(s * (s - a) * (s - b) * (s - c));
  printf("%lf\n", P);
  return 0;
}
```

Rešenje zadatka 8.10.3:

```
#include <stdio.h>

unsigned suma_delilaca(unsigned n) {
  unsigned s = 1, d;
  for (d = 2; d*d < n; d++)
    if (n % d == 0)
      s += d + n / d;
  if (d * d == n)
    s += d;
  return s;
}

int savrsen(unsigned n) {
  return n == suma_delilaca(n);
}

int main() {
  unsigned n;
  for (n = 1; n <= 10000; n++)
    if (savrsen(n))
      printf("%u\n", n);
  return 0;
}
```

Rešenje zadatka 8.10.4:

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

/* Struktura razlomak */
struct razlomak {
  int brojilac, imenilac;
};

/* a/b < c/d je ekvivalentno sa
   a*d < c*b ako je a*d > 0, tj. sa
   a*d > c*b ako je a*d < 0 */
int poredi_razlomke(struct razlomak a, struct razlomak b) {
  assert(a.imenilac != 0 && b.imenilac != 0);
  if(a.imenilac*b.imenilac>0) {
    if(a.brojilac*b.imenilac > a.imenilac*b.brojilac)
      return 1;
    else if(a.brojilac*b.imenilac == a.imenilac*b.brojilac)
      return 0;
    else
      return -1;
  } else {
    if(a.brojilac*b.imenilac < a.imenilac*b.brojilac)
      return 1;
```

```
    else if(a.brojilac*b.imenilac == a.imenilac*b.brojilac)
      return 0;
    else
      return -1;
  }
}

/* Ucitavaju se dva razlomka i porede se */
int main() {
  struct razlomak a, b;
  scanf("%d%d", &a.brojilac, &a.imenilac); assert(a.imenilac != 0);
  scanf("%d%d", &b.brojilac, &b.imenilac); assert(b.imenilac != 0);
  printf("%d\n", poredi_razlomke(a, b));
}
```

Rešenje zadatka 8.10.5:

```
#include <stdio.h>

int sadrzi(int a[], int n, int x) {
  int i;
  for (i = 0; i < n; i++)
    if (a[i] == x)
      return 1;
  return 0;
}

int prva_poz(int a[], int n, int x) {
  int i;
  for (i = 0; i < n; i++)
    if (a[i] == x)
      return i;
  return -1;
}

int poslednja_poz(int a[], int n, int x) {
  int i;
  for (i = n - 1; i >= 0; i--)
    if (a[i] == x)
      return i;
  return -1;
}

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

```
}

double prosek(int a[], int n) {
  int i;
  int s = 0;
  for (i = 0; i < n; i++)
    s += a[i];
  return (double)s / (double)n;
}

/* Pretpostavlja se da je niz neprazan */
int min(int a[], int n) {
  int i, m;
  m = a[0];
  for (i = 1; i < n; i++)
    if (a[i] < m)
      m = a[i];
  return m;
}

/* Pretpostavlja se da je niz neprazan */
int max_poz(int a[], int n) {
  int i, mp;
  mp = 0;
  for (i = 1; i < n; i++)
    if (a[i] > a[mp])
      mp = i;
  return mp;
}

int sortiran(int a[], int n) {
  int i;
  for (i = 0; i + 1 < n; i++)
    if (a[i] > a[i+1])
      return 0;
  return 1;
}

int main() {
  int a[] = {3, 2, 5, 4, 1, 3, 8, 7, 5, 6};
  int n = sizeof(a) / sizeof(int);
  printf("Sadrzi: %d\n", sadrzi(a, n, 3));
  printf("Prva pozicija: %d\n", prva_poz(a, n, 3));
  printf("Poslednja pozicija: %d\n", poslednja_poz(a, n, 3));
  printf("Suma: %d\n", suma(a, n));
  printf("Prosek: %lf\n", prosek(a, n));
  printf("Minimum: %d\n", min(a, n));
  printf("Pozicija maksimuma: %d\n", max_poz(a, n));
  printf("Sortiran: %d\n", sortiran(a, n));
  return 0;
```

```
}
```

Rešenje zadatka 8.10.6:

```
#include <stdio.h>

int izbaci_poslednji(int a[], int n) {
  return n - 1;
}

/* Cuva se redosled elemenata niza */
int izbaci_prvi_1(int a[], int n) {
  int i;
  for (i = 0; i + 1 < n; i++)
    a[i] = a[i + 1];
  return n - 1;
}

/* Ne cuva se redosled elemenata niza. */
int izbaci_prvi_2(int a[], int n) {
  int i;
  a[0] = a[n - 1];
  return n - 1;
}

/* Cuva se redosled elemenata niza. */
int izbaci_kti(int a[],  int n, int k) {
  int i;
  for (i = k; i + 1 < n; i++)
    a[i] = a[i + 1];
  return n - 1;
}

/* Izbacivanje prvog se moze svesti na izbacivanje k-tog. */
int izbaci_prvi_3(int a[], int n) {
  return izbaci_kti(a, n, 0);
}

/* Pretpostavlja se da ima dovoljno prostora za ubacivanje. */
int ubaci_na_kraj(int a[], int n, int x) {
  a[n] = x;
  return n + 1;
}

/* Pretpostavlja se da ima dovljno prostora za ubacivanje.
   Cuva se redosled elemenata niza.  */
int ubaci_na_pocetak_1(int a[], int n, int x) {
  int i;
  for (i = n; i > 0; i--)
```

```
    a[i] = a[i-1];
  a[0] = x;
  return n + 1;
}

/* Pretpostavlja se da ima dovljno prostora za ubacivanje.
   Ne cuva se redosled elemenata niza.  */
int ubaci_na_pocetak_2(int a[], int n, int x) {
  int i;
  a[n] = a[0];
  a[0] = x;
  return n + 1;
}

/* Pretpostavlja se da ima dovljno prostora za ubacivanje.
   Cuva se redosled elemenata niza.  */
int ubaci_na_poz_k(int a[], int n, int k, int x) {
  int i;
  for (i = n; i > k; i--)
    a[i] = a[i-1];
  a[k] = x;
  return n + 1;
}

/* Ubacivanje na pocetak se moze svesti na ubacivanje na poziciju k. */
int ubaci_na_pocetak_3(int a[], int n, int x) {
  return ubaci_na_poz_k(a, n, 0, x);
}

int izbaci_sve(int a[], int n, int x) {
  int i, j;
  for (j = 0, i = 0; i < n; i++)
    if (a[i] != x)
      a[j++] = a[i];
  return j;
}

void ispisi(int a[], int n) {
  int i;
  for (i = 0; i < n; i++)
    printf("%d ", a[i]);
  printf("\n");
}

int main() {
  int a[10] = {1, 2, 3};
  int n = 3;
  n = ubaci_na_pocetak_1(a, n, 0);
  ispisi(a, n);
  n = izbaci_poslednji(a, n);
```

```
  ispisi(a, n);
  n = ubaci_na_poz_k(a, n, 2, 4);
  ispisi(a, n);
  n = ubaci_na_kraj(a, n, 1);
  ispisi(a, n);
  n = izbaci_sve(a, n, 1);
  ispisi(a, n);
  return 0;
}
```

Rešenje zadatka 8.10.7:

```
#include <stdio.h>

/* Napomena: odredjen broj funkcija u ovom zadatku se moze
   implementirati i efikasnije, medjutim, uz prilicno komplikovaniji
   kod. */

void ispisi(int a[], int n) {
  int i;
  for (i = 0; i < n; i++)
    printf("%d ", a[i]);
  printf("\n");
}

int najduza_serija_jednakih(int a[], int n) {
  int i;
  /* Duzina tekuce serije je 0 ili 1 u zavisnosti od toga da li je niz
     prazan ili nije */
  int ts = n != 0;
  /* Najduza serija je tekuca serija */
  int ns = ts;
  for (i = 1; i < n; i++) {
    /* Ako je element jednak prethodnom */
    if (a[i] == a[i-1])
      /* Nastavlja se tekuca serija */
      ts++;
    else
      /* Inace, zapocinjemo novu seriju */
      ts = 1;
    /* Azuriramo vrednost najduze serije */
    if (ts > ns)
      ns = ts;
  }
  /* Vracamo duzinu najduze serije */
  return ns;
}

/* Ova funkcija se od prethodne razlikuje samo po uslovu poredjenja
```

```
   dva susedna elementa niza. Kasnije ce biti prikazano kako se to
   poredjenje moze parametrizovati. */
int najduza_serija_neopadajucih(int a[], int n) {
  int i;
  int ts = n != 0;
  int ns = ts;
  for (i = 1; i < n; i++) {
    if (a[i] >= a[i-1])
      ts++;
    else
      ts = 1;
    if (ts > ns)
      ns = ts;
  }
  return ns;
}

int podniz_uzastopnih(int a[], int n, int b[], int m) {
  int i, j;
  /* Za svako i takvo da od pozicije i do kraja niza a ima bar onoliko
     elemenata koliko i u nizu b*/
  for (i = 0; i + m - 1 < n; i++) {
    /* Proveravamo da li se niz b nalazi u nizu a pocevsi od
       pozicije i */
    for (j = 0; j < m; j++)
      if (a[i + j] != b[j])
        break;
    /* Nismo naisli na razliku, dakle, ceo niz b se nalazi u nizu a
       pocevsi od pozicije i */
    if (j == m)
      return 1;
  }
  /* Nismo pronasli b unutar a (inace bi funkcija bila prekinuta sa
     return 1) */
  return 0;
}

int podniz(int a[], int n, int b[], int m) {
  int i, j;
  /* Prolazimo kroz nisku a, trazeci slova niske b. */
  for (i = 0, j = 0; i < n && j < m; i++)
    /* Kada pronadjemo tekuce slovo niske b, prelazimo na sledece */
    if (a[i] == b[j])
      j++;
  /* Ako smo iscrpli sve karaktere niske b, onda jeste podniz, inace
     nije */
  return j == m;
}

void rotiraj_desno(int a[], int n, int k) {
```

```
  int i, j;
  /* k puta rotiramo za po jednu poziciju */
  for (j = 0; j < k; j++) {
    /* Upamtimo poslednji */
    int tmp = a[n-1];
    /* Sve elemente pomerimo za jedno mesto na desno */
    for (i = n-1; i > 0; i--)
      a[i] = a[i-1];
    /* Raniji poslednji stavimo na pocetak */
    a[0] = tmp;
  }
}

void rotiraj_levo(int a[], int n, int k) {
  int i, j;
  /* k puta rotiramo za po jednu poziciju */
  for (j = 0; j < k; j++) {
    /* upamtimo prvi */
    int tmp = a[0];
    /* Sve elemente pomerimo za jedno mesto u levo */
    for (i = 0; i + 1 < n; i++)
      a[i] = a[i+1];
    /* Raniji prvi postavimo na kraj */
    a[n - 1] = tmp;
  }
}

int izbaci_duplikate(int a[], int n) {
  int i, j, k;
  /* za svaki element niza a */
  for (i = 0; i < n; i++) {
    /* izbacujemo element a[i] iz ostatka niza a tako sto se elementi
       a[j] razliciti od a[i] prepisuju u niz a (ali na poziciju k
       koja moze biti i manja od j). */
    for (k = i + 1, j = i + 1; j < n; j++)
      if (a[j] != a[i])
        a[k++] = a[j];
    n = k;
  }
  return n;
}

/* Pretpostavlja se da u niz c moze da se smesti bar n + m elemenata */
int spoji(int a[], int n, int b[], int m, int c[]) {
  int i, j, k;
  /* Tekuca pozicija u prvom, drugom nizu i u rezultatu */
  i = 0, j = 0, k = 0;
  /* Dok postoje elementi i u jednom i u drugom nizu */
  while (i < n && j < m)
    /* U rezultat prepisujemo manji element */
```

```
    c[k++] = a[i] < b[j] ? a[i++] : b[j++];
  /* Ono sto je preostalo u nizovima prepisujemo u rezultat (jedna od
     naredne dve petlje je uvek prazna) */
  while (i < n)
    c[k++] = a[i++];
  while (j < m)
    c[k++] = b[j++];
  return k;
}

void obrni(int a[], int n) {
  int i, j;
  /* Prolazimo niz paralelno sa dva kraja dok se pozicije ne
     mimoidju */
  for (i = 0, j = n-1; i < j; i++, j--) {
    /* Vrsimo razmenu elemenata */
    int tmp = a[i];
    a[i] = a[j];
    a[j] = tmp;
  }
}

/* Testiramo napisane funkcije */
int main() {
  int a[] = {3, 4, 8, 8, 9, 12}, b[] = {10, 9, 7, 7, 7, 6, 5, 5, 2},
      c[] = {4, 8, 8, 9}, d[] = {3, 4, 9, 12}, e[20];
  int na = sizeof(a)/sizeof(int), nb = sizeof(b)/sizeof(int),
      nc = sizeof(c)/sizeof(int), nd = sizeof(d)/sizeof(int), ne;
  printf("%d\n", najduza_serija_jednakih(b, nb));
  printf("%d\n", najduza_serija_neopadajucih(b, nb));
  printf("%d\n", najduza_serija_neopadajucih(a, na));
  printf("%d\n", podniz_uzastopnih(a, na, c, nc));
  printf("%d\n", podniz_uzastopnih(a, na, d, nd));
  printf("%d\n", podniz(a, na, c, nc));
  printf("%d\n", podniz(a, na, d, nd));
  printf("%d\n", podniz(b, nb, d, nd));
  obrni(b, nb);
  ne = spoji(a, na, b, nb, e);
  ne = izbaci_duplikate(e, ne);
  ispisi(e, ne);
  rotiraj_levo(e, ne, 2);
  ispisi(e, ne);
  rotiraj_desno(e, ne, 5);
  ispisi(e, ne);
  return 0;
}
```

Rešenje zadatka 8.10.8:

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

```
/* Funkcija radi slicno funkciji strcmp iz string.h */
int poredi(char s1[], char s2[]) {
  /* Petlja tece sve dok ne naidjemo na prvi razliciti karakter */
  int i;
  for (i = 0; s[i]==t[i]; i++)
    if (s[i] == '\0') /* Naisli smo na kraj obe niske,
                      a nismo nasli razliku */
    return 0;

  /* s[i] i t[i] su prvi karakteri u kojima se niske razlikuju.
     Na osnovu njihovog odnosa, odredjuje se odnos stringova */
  return s[i] - t[i];
}

int main() {
  printf("%d\n", poredi("zdravo", "svima"));
  return 0;
}
```

Rešenje zadatka 8.10.9:

```
#include <stdio.h>

/* Proverava da li se niska sub javlja kao podniz niske str (redosled
   je bitan, ali karakteri ne moraju da se javljaju uzastopno.
   Funkcija vraca 1 ako jeste, a 0 ako nije podniz. */
int podniz(char str[], char sub[]) {
  int i, j = 0;
  /* Trazimo svako slovo iz niske sub */
  for (i = 0; sub[i] != '\0'; i++)
    /* Petlja traje dok ne nadjeno trazeno slovo */
    while (str[j] != sub[i]) {
      /* Ako smo usput stigli do kraja niske str, a nismo
         nasli trazeno slovo, onda nije podniz. */
      if (str[j] == '\0')
        return 0;
      j++;
    }
  /* Svako slovo iz sub smo pronasli, onda jeste pondiz */
  return 1;
}

/* Proverava da li niska str sadrzi nisku sub kao podnisku (karakteri
   moraju da se jave uzastopno). Funkcija radi isto slicno kao
   bibliotecka funkcija strstr, ali vraca poziciju na kojoj sub
   pocinje, odnosno -1 ukoliko ga nema. Postoje efikasniji algoritmi
   za re\v savanje ovog problema, od naivnog algoritma koji je ovde
   naveden. */
```

```
int podniska(char str[], char sub[]) {
  int i, j;
  /* Proveravamo da li sub pocinje na svakoj poziciji i */
  for (i = 0; str[i] != '\0'; i++)
  /* Poredimo sub sa str pocevsi od pozicije i
     sve dok ne naidjemo na razliku */
     for (j = 0; str[i+j] == sub[j]; j++)
        /* Nismo naisli na razliku a ispitali smo
           sve karaktere niske sub */
        if (sub[j+1]=='\0')
          return i;
  /* Nije nadjeno */
  return -1;
}

int main() {
  printf("%d\n", podniz("banana", "ann"));
  printf("%d\n", podniska("banana", "anan"));
  return 0;
}
```

Rešenje zadatka 8.10.10:

```
#include <stdio.h>

/* Dve niske su permutacija jedna druge akko imaju podjednak broj
   pojavljivanja svih karaktera. Pretpostavljamo da niske sadrze samo
   ASCII karaktere. */
int permutacija(char s[], char t[]) {
  /* Broj pojavljivanja svakog od 128 ASCII karaktera u niski s i niski t */
  int ns[128], nt[128], i;
  /* Inicijalizujemo brojace na 0 */
  for (i = 0; i < 128; i++)
    ns[i] = nt[i] = 0;
  /* Brojimo karaktere niske s */
  for (i = 0; s[i]; i++)
    ns[s[i]]++;
  /* Brojimo karaktere niske t */
  for (i = 0; t[i]; i++)
    nt[t[i]]++;

  /* Poredimo brojeve za svaki od 128 ASCII karaktera */
  for (i = 0; i < 128; i++)
    if (ns[i] != nt[i])
      return 0;
  /* Nismo naisli na razliku - dakle svi karakteri se javljaju isti
     broj puta */
  return 1;
}
```

```
/* Provera funkcije */
int main() {
  char s[] = "tom marvolo riddle ", t[] = "i am lord voldemort";
  printf("%d\n", permutacija(s, t));
  return 0;
}
```

Rešenje zadatka 8.10.11:

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

int zbir_vrste(int m[10][10], int n, int i) {
  int j;
  int zbir = 0;
  assert(n < 10 && i < 10);
  for (j = 0; j < n; j++)
    zbir += m[i][j];
  return zbir;
}

int zbir_kolone(int m[10][10], int n, int j) {
  int i;
  int zbir = 0;
  assert(n < 10 && j < 10);
  for (i = 0; i < n; i++)
    zbir += m[i][j];
  return zbir;
}

int zbir_glavne_dijagonale(int m[10][10], int n) {
  int i;
  int zbir = 0;
  assert(n < 10);
  for (i = 0; i < n; i++)
    zbir += m[i][i];
  return zbir;
}

int zbir_sporedne_dijagonale(int m[10][10], int n) {
  int i;
  int zbir = 0;
  assert(n < 10);
  for (i = 0; i < n; i++)
    zbir += m[i][n-i-1];
  return zbir;
}
```

```
int ucitaj(int m[10][10]) {
  int i, j, n;
  scanf("%d", &n); assert(1 <= n && n < 10);
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      scanf("%d", &m[i][j]);
  return n;
}

int magicni(int m[10][10], int n) {
  int i, j, zbir;
  zbir = zbir_vrste(m, n, 0);
  for (i = 1; i < n; i++)
    if (zbir_vrste(m, n, i) != zbir)
      return 0;
  for (j = 0; j < n; j++)
    if (zbir_kolone(m, n, j) != zbir)
      return 0;
  if (zbir_glavne_dijagonale(m, n) != zbir)
    return 0;
  if (zbir_sporedne_dijagonale(m, n) != zbir)
    return 0;
  return 1;
}

int main() {
  int n, m[10][10];
  n = ucitaj(m);
  printf("%d\n", magicni(m, n));
  return 0;
}
```

Rešenje zadatka 8.10.12:

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

#define MAX 100

typedef struct {
  float a[MAX][MAX];
  unsigned m, n;
} MATRICA;

/* Efikasnije bi bilo prenositi matricu kao argument,
   preko pokazivaca, ali to jos nije uvedeno. */
MATRICA ucitaj() {
  MATRICA m;
```

```
  unsigned i, j;
  scanf("%u%u", &m.m, &m.n);
  assert(m.n < MAX);
  for (i = 0; i < m.m; i++)
    for (j = 0; j < m.n; j++)
      scanf("%f", &m.a[i][j]);
  return m;
}

MATRICA saberi(MATRICA m1, MATRICA m2) {
  MATRICA m;
  unsigned i, j;
  assert(m1.m == m2.m && m1.n == m2.n);
  m.m = m1.m; m.n = m1.n;
  for (i = 0; i < m.n; i++)
    for (j = 0; j < m.n; j++)
      m.a[i][j] = m1.a[i][j] + m2.a[i][j];
  return m;
}

MATRICA pomnozi(MATRICA m1, MATRICA m2) {
  MATRICA m;
  unsigned i, j, k;
  assert(m2.m == m1.n);
  m.m = m1.m; m.n = m2.n;
  for (i = 0; i < m.m; i++)
    for (j = 0; j < m.n; j++) {
      m.a[i][j] = 0;
      for (k = 0; k < m1.n; k++)
        m.a[i][j] += m1.a[i][k]*m2.a[k][j];
    }
  return m;
}

void ispisi(MATRICA m) {
  unsigned i, j;
  for (i = 0; i < m.m; i++) {
    for (j = 0; j < m.n; j++)
      printf("%g ", m.a[i][j]);
    putchar('\n');
  }
}

int main() {
  MATRICA m1, m2, m;
  m1 = ucitaj(); m2 = ucitaj();
  if (m1.m == m2.m && m1.n == m2.n) {
    m = saberi(m1, m2);
    ispisi(m);
  }
```

```
  if (m1.n = m2.m) {
    m = pomnozi(m1, m2);
    ispisi(m);
  }
  return 0;
}
```

Rešenje zadatka ??:

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

#define MAX 100

/* Veliki broj je predstavljen nizom od n cifara. Cifre se zapisuju
   "naopako" kako bi se cifra uz 10^i nalazila na poziciji i */
typedef struct {
  unsigned char c[MAX];
  unsigned n;
} BROJ;

BROJ ucitaj() {
  BROJ rez;
  rez.n = 0;
  int c, i, j;
  while(isdigit(c = getchar())) {
    rez.c[rez.n++] = c - '0';
    assert(rez.n <= MAX);
  }
  /* Obrcemo zapis */
  for (i = 0, j = rez.n-1; i < j; i++, j--) {
    unsigned char tmp = rez.c[i];
    rez.c[i] = rez.c[j];
    rez.c[j] = tmp;
  }
  return rez;
}

void ispisi(BROJ b) {
  int i;
  for (i = b.n-1; i >= 0; i--)
    putchar('0' + b.c[i]);
}

BROJ saberi(BROJ a, BROJ b) {
  BROJ rez;
  unsigned i;
  /* Prenos */
  unsigned char p = 0;
```

```
  for (i = 0; i < a.n || i < b.n; i++) {
    /* Cifra prvog broja ili 0 ako su sve cifre iscrpljene */
    unsigned char c1 = i < a.n ? a.c[i] : 0;
    /* Cifra drugog broja ili 0 ako su sve cifre iscrpljene */
    unsigned char c2 = i < b.n ? b.c[i] : 0;
    /* Zbir cifara + prenos sa prethodne pozicije*/
    unsigned char c = c1 + c2 + p;
    /* Nova cifra */
    rez.c[i] = c % 10;
    /* Prenos na sledecu poziciju */
    p = c / 10;
  }
  /* Broj cifara rezultata */
  rez.n = i;
  /* Ako postoji prenos, treba dodati jos jednu cifru */
  if (p > 0) {
    assert(rez.n < MAX);
    rez.c[rez.n] = p;
    rez.n++;
  }
  return rez;
}

BROJ pomnozi(BROJ a, BROJ b) {
  BROJ rez; /* rezultat */
  unsigned i, j;
  unsigned char p; /* prenos */

  /* Maksimalni broj cifara rezultata */
  rez.n = a.n + b.n;
  assert(rez.n <= MAX);
  /* Inicijalizujemo sve cifre rezultata */
  for (i = 0; i < rez.n; i++)
    rez.c[i] = 0;
  /* Mnozimo brojeve ne vrseci nikakav prenos */
  for (i = 0; i < a.n; i++)
    for (j = 0; j < b.n; j++)
      rez.c[i+j] += a.c[i]*b.c[j];

  /* Normalizujemo rezultat */
  p = 0;
  for (i = 0; i < rez.n; i++) {
    unsigned char c = rez.c[i] + p;
    rez.c[i] = c % 10;
    p = c / 10;
  }
  /* Ako je prva cifra 0, smanjujemo broj cifara */
  if (rez.c[rez.n-1] == 0)
    rez.n--;
  return rez;
```

```
}

int main() {
  BROJ a = ucitaj(), b = ucitaj(), c;
  c = saberi(a, b);
  ispisi(c);
  putchar('\n');
  c = pomnozi(a, b);
  ispisi(c);
  putchar('\n');
}
```

Jezik C - pokazivači

Rešenje zadatka 10.1:

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

/* Funkcija vraca 3 vrednosti preko pokazivaca */
void od_ponoci(unsigned n, unsigned* h, unsigned *m, unsigned *s) {
  *h = n / 3600;
  n %= 3600;
  *m = n / 60;
  n %= 60;
  *s = n;
}

int main() {
  unsigned n, h, m, s;
  scanf("%d", &n); assert(n < 60*60*24);
  od_ponoci(n, &h, &m, &s);
  printf("%d:%d:%d\n", h, m, s);
  return 0;
}
```

Rešenje zadatka 10.2:

```
#include <stdio.h>

int paran(int x) {
  return x % 2 == 0;
}

int pozitivan(int x) {
  return x > 0;
}

/* Kriterijum se prosledjuje u obliku pokazivaca na funkciju */
```

```
int najduza_serija(int a[], int n, int (*f) (int)) {
  int i;
  /* Duzina tekuce serije je 0 ili 1 u zavisnosti od toga da li je niz
     prazan ili nije */
  int ts = 0;
  /* Najduza serija je tekuca serija */
  int ns = ts;
  for (i = 0; i < n; i++) {
    /* Ako element zadovoljava kriterijum */
    if ((*f)(a[i]))
      /* Nastavlja se tekuca serija */
      ts++;
    else
      /* Inace, prekidamo seriju */
      ts = 0;
    /* Azuriramo vrednost najduze serije */
    if (ts > ns)
      ns = ts;
  }
  /* Vracamo duzinu najduze serije */
  return ns;
}

/* Testiramo napisane funkcije */
int main() {
  int a[] = {1, 2, 4, -6, 5, 3, -3, 2, 5, 7, 9, 11},
      n = sizeof(a)/sizeof(int);
  printf("%d %d\n", najduza_serija(a, n, &paran),
                    najduza_serija(a, n, &pozitivan));
}
```

Rešenje zadatka 10.5.1:

```
#include <stdio.h>

size_t my_strlen(const char* s) {
  const char* t;
  for (t = s; *t; t++)
    ;
  return t - s;
}

size_t my_strcpy(char* dest, const char* src) {
  while(*dest++ = *src++)
    ;
}

int my_strcmp(char* s, char *t) {
  while (*s && *s == *t)
```

```
    s++, t++;
  return *s - *t;
}

void my_strrev(char* s) {
  char *t = s;
  /* Dovodimo s ispred terminalne nule */
  while(*s)
    s++;
  s--;
  /* Obrcemo karaktere */
  for (; t < s; t++, s--) {
    char tmp = *s; *s = *t; *t = tmp;
  }
}

const char* my_strchr(char x, const char* s) {
  while(*s) {
    if (*s == x) return s;
    s++;
  }
  return NULL;
}

const char* my_strstr(const char* str, const char* sub) {
    if (str == NULL || sub == NULL)
        return NULL;

    for ( ; *str; str++) {
        const char *s, *t;
        for (s = str, t = sub; *s && (*s == *t); s++, t++)
;
        if (*t == '\0')
            return str;
    }
    return NULL;
}

int main() {
  char s[] = "abc", t[] = "def", r[4];
  printf("%lu %lu\n", my_strlen(s), my_strlen(t));
  my_strcpy(r, s);
  printf("%s\n", r);
  my_strrev(s);
  printf("%s\n", s);
  printf("%d\n", my_strcmp(s, t));
  printf("%c\n", *my_strchr('x', "abcxyz"));
  printf("%s\n", my_strstr("abcdefghi", "def"));
  return 0;
}
```

Rešenje zadatka 10.9.1:

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

int main() {
  unsigned n, i, min;
  int *a, x;

  /* Ucitavamo niz */
  scanf("%u", &n);
  assert(n > 0);
  a = malloc(n*sizeof(int)); /* Dinamicka alokacija */
  assert(a != 0);
  for (i = 0; i < n; i++)
    scanf("%d", &a[i]);

  /* Ucitavamo broj koji se trazi */
  scanf("%d", &x);

  /* Trazimo element niza a najblizi broju x */
  min = 0;
  for (i = 1; i < n; i++)
    if (abs(a[i] - x) < abs(a[min] - x))
      min = i;

  /* Ispisujemo rezultat */
  printf("a[%d] = %d\n", min, a[min]);

  /* Oslobadjamo niz */
  free(a);
  return 0;
}
```

Rešenje zadatka 10.9.2:

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

#define KORAK 32

int main() {
  unsigned n = 0, aloc = 0, i, m, nm;
  int *a = NULL;

  /* Ucitavamo niz, dinamicki ga realocirajuci */
  do {
```

```
    if (aloc <= n) {
      int* a_novo;
      aloc += KORAK;
      a_novo = realloc(a, aloc*sizeof(int));
      assert(a_novo);
      a = a_novo;
    }
    scanf("%d", &a[n]);
  } while(a[n++] != 0);

  /* Trazenje najcesceg elementa - ovaj algoritam je neefikasan, ali
     se jednostavno implementira */
  /* Broj pojavljivanja najcesceg elementa */
  nm = 0;
  for (i = 0; i < n; i++) {
    /* Brojimo koliko puta se javlja element a[i] */
    int ni = 0, j;
    for (j = 0; j < n; j++)
      if (a[i] == a[j])
        ni++;
    /* Ako se javlja cesce od do tada najcesceg, azuriramo broj
       pojavljivanja najcesceg elementa i vrednost najcesceg */
    if (ni > nm) {
      nm = ni;
      m = a[i];
    }
  }

  /* Ispisujemo najcesci element i njegov broj pojavljivanja */
  printf("%d - %d\n", m, nm);

  /* Oslobadjamo niz */
  free(a);

  return 0;
}
```

Rešenje zadatka 10.9.3:

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
int main() {
  int n, i, j, lx, ux, ly, uy, **A;

  /* Ucitavamo dimenziju matrice */
  scanf("%d", &n);

  /* Dinamicki alociramo prostor */
```

```
  A = malloc(n*sizeof(int*));
  assert(A);
  for (i = 0; i < n; i++) {
    A[i] = malloc(n*sizeof(int));
    assert(A[i]);
  }

  /* Ucitavamo elemente matrice */
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      scanf("%d", &A[i][j]);

  /* Spiralno ispisujemo raspon [lx, ux] * [ly, uy] */
  lx = 0; ux = n-1; ly = 0; uy = n-1;
  while (lx <= ux && ly <= uy)  {
    for (i = lx; i <= ux; i++)
      printf("%d ", A[i][ly]);
    for (j = ly+1; j <= uy; j++)
      printf("%d ", A[ux][j]);
    for (i = ux-1; i >= lx; i--)
      printf("%d ", A[i][uy]);
    for (j = uy-1; j >= ly+1; j--)
      printf("%d ", A[lx][j]);
    lx++; ux--; ly++; uy--;
  }

  /* Oslobadjamo matricu */
  for (i = 0; i < n; i++)
    free(A[i]);
  free(A);

  return 0;
}
```

Rešenje zadatka 10.9.4:

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

int skalarni_proizvod(int *a, int *b, int n) {
  int suma = 0, i;
  for (i = 0; i < n; i++)
    suma += a[i]*b[i];
  return suma;
}

int ortonormirana(int** A, int n) {
  int i, j;
```

```
  for (i = 0; i < n; i++) {
    if (skalarni_proizvod(A[i], A[i], n) != 1)
      return 0;
    for (j = i+1; j < n; j++)
      if (skalarni_proizvod(A[i], A[j], n) != 0)
        return 0;
  }
  return 1;
}

int main() {
  int n, i, j, **A;

  /* Ucitavamo dimenziju matrice */
  scanf("%d", &n);

  /* Dinamicki alociramo prostor */
  A = malloc(n*sizeof(int*));
  assert(A);
  for (i = 0; i < n; i++) {
    A[i] = malloc(n*sizeof(int));
    assert(A[i]);
  }

  /* Ucitavamo elemente matrice */
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      scanf("%d", &A[i][j]);

  /* Ispitujemo da li je matrica ortonormirana */
  printf("%d\n", ortonormirana(A, n));

  /* Oslobadjamo matricu */
  for (i = 0; i < n; i++)
    free(A[i]);
  free(A);

  return 0;
}
```

Jezik C - ulaz/izlaz

Rešenje zadatka 12.4.1:

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

int main(int argc, char* argv[]) {
  int zbir = 0;
```

```
  for (i = 0; i < argc; i++)
    zbir += atoi(argv[i]);
  printf("%d\n", zbir);
  return 0;
}
```

Rešenje zadatka 12.4.2:

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

/* Racuna se zbir kolone j matrice A dimenzije n */
int zbir_kolone(int** A, unsigned n, unsigned j) {
  int i, zbir = 0;
  for (i = 0; i < n; i++)
    zbir += A[i][j];
  return zbir;
}

int main(int argc, char* argv[]) {
  unsigned n, i, j, max_j;
  int **A, max;
  FILE* dat;

  if (argc < 2) {
    fprintf(stderr, "Greska: nedostaje ime ulazne datoteke\n");
    return -1;
  }

  dat = fopen(argv[1], "r");
  if (dat == NULL) {
    fprintf(stderr,
            "Greska: ucitavanje datoteke %s nije uspelo\n",
            argv[1]);
    return -1;
  }

  /* Ucitavamo dimenziju matrice */
  assert(fscanf(dat, "%u", &n) == 1);
  assert(n >= 1);

  /* Dinamicki alociramo prostor */
  A = malloc(n*sizeof(int*));
  assert(A);
  for (i = 0; i < n; i++) {
    A[i] = malloc(n*sizeof(int));
    assert(A[i]);
  }
```

```
  /* Ucitavamo matricu */
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      assert(fscanf(dat, "%d", &A[i][j]) == 1);

  /* Odredjujemo maksimum */
  max_j = 0; max = zbir_kolone(A, n, 0);
  for (j = 1; j < n; j++) {
    int m = zbir_kolone(A, n, j);
    if (m > max) {
      max_j = j;
      max = m;
    }
  }

  /* Prijavljujemo rezultat */
  printf("Kolona: %u Zbir: %d\n", max_j, max);

  /* Oslobadjamo matricu */
  for (i = 0; i < n; i++)
    free(A[i]);
  free(A);

  /* Zatvaramo datoteku */
  fclose(dat);

  return 0;
}
```

Rešenje zadatka 12.4.3:

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

#define KORAK 64

typedef struct {
  char ime[20];
  char prezime[20];
  unsigned br_ispita;
} APSOLVENT;

int main() {
  FILE* dat;
  APSOLVENT* apsolventi = NULL, apsolvent;
  unsigned aloc = 0, n = 0, i, zbir;
  float prosek;
```

```
  dat = fopen("apsolventi.txt", "r");
  if (dat == NULL) {
    fprintf(stderr,
            "Greska pri otvaranju datoteke apsolventi.txt\n");
    return 1;
  }

  /* Ucitavanje apsolvenata uz dinamicku realokaciju niza */
  while(fscanf(dat, "%s%s%d",
               apsolvent.ime, apsolvent.prezime, &apsolvent.br_ispita)
        == 3) {
    if (n >= aloc) {
      aloc += KORAK;
      APSOLVENT* tmp = realloc(apsolventi, aloc*sizeof(APSOLVENT));
      assert(tmp);
      apsolventi = tmp;
    }
    apsolventi[n++] = apsolvent;
  }

  /* Zatvaramo datoteku */
  fclose(dat);

  /* Izracunavanje prosecnog broja ispita */
  zbir = 0;
  for (i = 0; i < n; i++)
    zbir += apsolventi[i].br_ispita;
  prosek = (float)zbir / n;

  /* Ispisujemo apsolvente sa natprosecnim brojem ispita */
  for (i = 0; i < n; i++)
    if (apsolventi[i].br_ispita > prosek)
      printf("%s %s: %d\n", apsolventi[i].ime,
                            apsolventi[i].prezime,
                            apsolventi[i].br_ispita);

  /* Oslobadjamo memoriju */
  free(apsolventi);

  return 0;
}
```

Rešenje zadatka 12.4.4:

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

```
/* Maksimalan broj cifara u zapisu */
#define MAX_DIGITS 100

/* Vrednost cifre predstavljene ASCII karakterom.
   Npr. '0'->0, ..., '9'->9,
        'a'->10, ..., 'f' -> 15,
        'A' -> 10, ..., 'F' -> 15,
        Za karakter koji nije alfanumericki vraca -1 */
signed char vrednost_cifre(unsigned char c) {
  if (isdigit(c)) /* Cifre */
    return c-'0';
  if (isalpha(c) && islower(c)) /* Mala slova */
    return c-'a' + 10;
  if (isalpha(c) && isupper(c)) /* Velika slova */
    return c-'A' + 10;
  return -1;
}

/* Cita zapis niske s kao broja u osnovi b. Citanje se vrsi dok se
   javljaju validne cifre u osnovi b. Koristi se Hornerova shema. */
unsigned btoi(char s[], unsigned char b) {
  unsigned rez = 0, i;
  for (i = 0; s[i]; i++) {
    signed char c = vrednost_cifre(s[i]);
    /* Karakter ne predstavlja validnu cifru u osnovi b */
    if (c == -1 || c >= b) break;
    rez = rez * b;
    rez += c;
  }
  return rez;
}

/* Odredjuje zapis cifre c kao ASCII karakter */
unsigned char zapis_cifre(unsigned c) {
  assert(c <= 36);
  if (0 <= c && c <= 9)
    return '0' + c;
  else if (10 <= c && c <= 36)
    return 'a' + c;
}

/* Zapisuje broj n u osnovi b.
   Funkcija pretpostavlja da niska s sadrzi dovoljno karaktera za
   zapis. */
void itob(unsigned n, unsigned char b, char s[]) {
  /* Formira se niz cifara */
  unsigned i = 0, j;
  do {
    s[i++] = zapis_cifre(n % b);
    n /= b;
```

```
  } while (n > 0);
  s[i] = '\0';
  /* Obrce se niz cifara */
  for (j = 0, --i; j < i; j++, i--) {
    char tmp = s[i];
    s[i] = s[j];
    s[j] = tmp;
  }
}

int main(int argc, char* argv[]) {
  /* Broj u osnovi b koji se ucitava (niska cifara i vrednost) */
  char s[MAX_DIGITS];
  unsigned n;
  /* Preveden broj u osnovu 2 */
  char bs[MAX_DIGITS];
  /* Datoteke iz kojih se ucitava i u koje se upisuje */
  FILE *ulaz, *izlaz;

  /* Otvaranje datoteka */
  if (argc < 3) {
    fprintf(stderr, "Upotreba: %s <ulaz> <izlaz>\n", argv[0]);
    return 1;
  }
  ulaz = fopen(argv[1], "r");
  if (ulaz == NULL) {
    fprintf(stderr, "Greska prilikom otvaranja %s\n", argv[1]);
    return 1;
  }
  izlaz = fopen(argv[2], "w");
  if (izlaz == NULL) {
    fprintf(stderr, "Greska prilikom otvaranja %s\n", argv[2]);
    return 1;
  }

  /* Citanje i obrada brojeva */
  while (fgets(s, MAX_DIGITS, ulaz) != NULL) {
    char *t = s;
    /* Ako je uneta prazna linija, prekida se postupak */
    if (*t == '\0' || *t == '\n')
      break;
    if (*t == '0') {
      t++;
      if (*t == 'x' || *t == 'X') {
        /* Heksadekadni broj pocinje sa 0x ili 0X */
        t++;
        n = btoi(t, 16);
        printf("%u\n", n);
        itob(n, 2, bs);
        fputs(bs, izlaz); fputc('\n', izlaz);
```

```
      } else {
        /* Oktalni broj pocinje sa 0 */
        n = btoi(t, 8);
        printf("%u\n", n);
        itob(n, 2, bs);
        fputs(bs, izlaz); fputc('\n', izlaz);
      }
    } else {
      /* Dekadni broj */
      n = btoi(t, 10);
      printf("%u\n", n);
      itob(n, 2, bs);
      fputs(bs, izlaz); fputc('\n', izlaz);
    }
  }

  /* Zatvaramo datoteke */
  fclose(ulaz);
  fclose(izlaz);
  return 0;
}
```