Programiranje I

Beleške za predavanja

Smer *Informatika*
Matematički fakultet, Beograd

Predrag Janičić i Filip Marić

2010.

Sadržaj

Deo I

Osnovni pojmovi računarstva i programiranja

Glava 1

Računarstvo i računarski sistemi

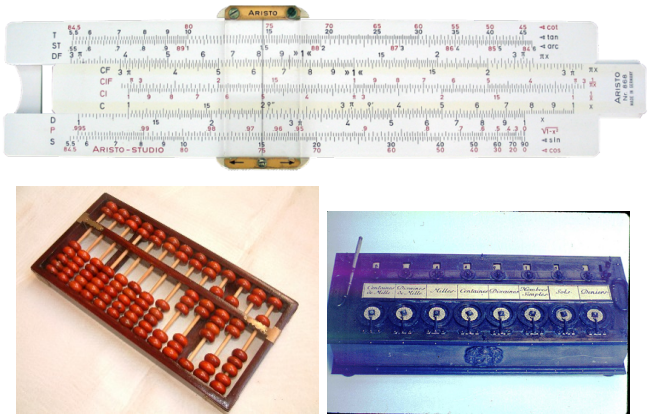
Koreni računarstva su mnogo stariji od prvih računara. Mnogo vekova unazad ljudi su stvarali naprave koje su olakšavale izračunavanja i mogle da rešavaju neke numeričke zadatke. Računari u današnjem smislu nastali su polovinom dvadesetog veka. Za funkcionisanje modernih računara neophodan je *softver* i *hardver*. Sofver čine računarski programi i prateći podaci koji određuju izračunavanja koje vrši računar. Hardver čine opipljive, fizičke komponente računara: procesor, memorija, matična ploča, hard disk, DVD uređaj, itd. Računarstvo je danas veoma široka i dobro utemeljena naučna disciplina sa mnoštvom podoblasti.

1.1 Rana istorija računarskih sistema

Programiranje je postalo praktično moguće tek krajem Drugog svetskog rata, ali ono svoje korene ima mnogo vekova ranije. Prvi precizni postupci za rešavanje matematičkih problema postojali su još u vreme Stare Grčke, pa i ranije, u starijim civilizacijama. Kao pomoć pri izvođenju osnovnih matematičkih operacija korišćene su računaljke (abakusi). U IX veku u Persiji matematičar *Al Horezmi (engl. Muhammad ibn Musa al-Khwarizmi, 780–850)* sastavlja rukopise u kojima su opisani precizni postupci računanja u indo-arapskom dekadnom brojevnom sistemu (koji i danas predstavlja osnovni brojevni sistem). U XIII veku *Leonardo Fibonači (lat. Leonardo Pisano Fibonacci, 1170–1250)* iz Pize donosi ovaj način zapisa brojeva u Evropu, što predstavlja jedan od glavnih preduslova za razvoj matematike i tehničkih disciplina tokom doba renesanse. Dolazi do otkrića logaritma uz pomoć kojeg se množenje može svesti na sabiranje što omogućava konstruisanje različitih analognih sprava koje su pomagale u računu (npr. klizni lenjir — šiber)[1]. Prve mehaničke sprave koja su mogle

[1] Zanimljivo je npr. da su klizni lenjiri nošeni na pet Apolo misija, uključujući i onu na mesec kako bi astronautima pomogli da izvrše potrebna računanja.

da potpuno automatski izvode aritmetičke operacije i pomažu u rešavanju matematičkih zadataka su napravljene kasnije. Francuski filozof, matematičar i fizičar *Blez Pascal (fr. Blaise Pascal, 1623–1662)* konstruisao je 1642. godine mehaničku spravu, kasnije nazvanu *Paskalina*, koja je služila za sabiranje i oduzimanje celih brojeva. U njegovu čast jedan programski jezik nosi ime *PASCAL*. *Gotfrid Lajbnic (nem. Gottfried Wilhelm Leibniz, 1646–1716)*, nemački filozof i matematičar, konstruisao je 1672. godine, mašinu koja je mogla da izvršava sve četiri osnovne aritmetičke operacije nad celim brojevima. Iako je mašina bila zasnovana na dekadnom brojevnom sistemu, Lajbinic je bio prvi koji je predlagao uvođenje binarnog brojevnog sistema.

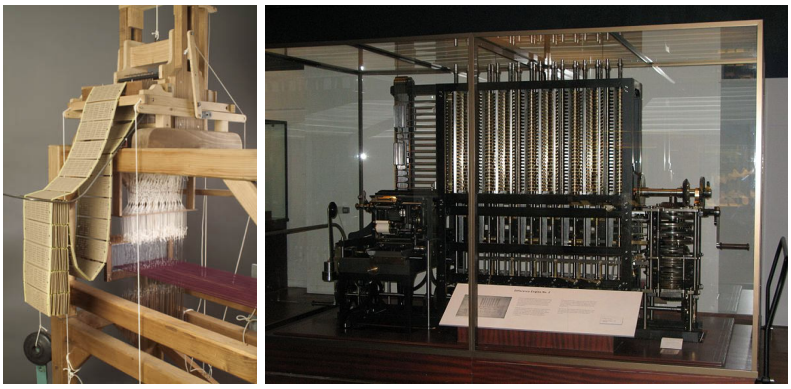


Slika 1.1: Šiber. Abakus. Paskalina

Mehanički uređaji. *Žozef Mari Žakard (fr. Joseph Marie Jacquard, 1752–1834)* konstruisao je 1801. godine mehanički tkački razboj koji je koristio bušene kartice. Svaka rupa na kartici određivala je jedan pokret mašina, a svaki red na kartici odgovarao je jednom redu šare. Time je bušena kartica predstavljala svojevrsni program za generisanje kompleksnih šara na tkanini. U prvoj polovini devetnaestog veka, engleski matematičar, filizof i izumitelj, *Čarls Bebidž (engl. Charles Babbage, 1791–1871)* dizajnirao je, mada ne i realizovao, prve programibilne mašine. Godine 1822. započeo je rad na *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, ali nije završena[2]. Ubrzo nakon propasti prvog projekta, Bebidž je započeo rad na novoj mašini nazvanoj *analitička mašina*. Osnovna razlika

[2]Dosledno sledeći Bebidžev dizajn, 1991. godine (u naučno-popularne svrhe) uspešno je konstruisana diferencijska mašina koja rad 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.

u odnosu na prethodnu mašinu bila je u tome da je zamišljeno da analitička mašina može da se programira, programima zapisanim na bušenim karticama, sličnim Žakardovim karticama. Program zapisan na karticama bi kontrolisao mehanički računar i omogućavao sekvencijalno izvršavanje naredbi, grananje i skokove, slično današnjim računarima. Osnovni delovi računara bili bi mlin (engl. mill) i skladište (engl. store), koji po svojoj funkcionalnosti sasvim odgovaraju procesoru i memoriji današnjih računara. *Augusta Ada Lovlejs (rođena Bajron) (engl. Augusta Ada King (rođ. Byron), Countess of Lovelace, 1815–1852)* napisala je prve programe za analitičku mašinu i da je mašina uspešno konstruisana, njeni programi bi mogli da računaju složene nizove brojeva (takozvane Bernulijeve brojeve). Zbog ovoga se Ada smatra prvim programerom u istoriji (i njoj u čast jedan programski jezik nosi ime Ada).



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 koga je konstruisao *Herman Holerit (engl. Herman Hollerith, 1860—1929)*. Ovaj uređaj korišćen je 1890. za obradu rezultata popisa stanovništva u SAD. Naime, obrada rezultata popisa iz 1880. godine trajala je preko 7 godina, a zbog naglog porasta broja stanovnika procenjeno je da bi obrada rezultata iz 1890. godine trajala preko 10 godina, što je bilo neprihvatljivo mnogo. Holeritova ideja je bila da se podaci prilikom popisa zapisuju na mašinski čitljivom medijumu (bušenim karticama), a kasnije obrađuju njegovom mašinom čime je obrada rezultata uspešno završena u toku godinu dana. Od Holeritove male kompanije kasnije je nastala kompanija *IBM*.

Godine 1941, nemački inženjer Konrad Cuze (nem. Konrad Zuse, 1910—1995) konstruisao je 22-bitni uređaj za računanje *Z3* koji je imao izvesne mogućnosti programiranja, te se često smatra i prvim realizovanim programabilnim računarom. Cuzeove mašine tokom rata naišle su samo na ograničene primene. Nakon rata, Cuzeova kompanija proizvela je oko 250 različitih tipova računara

do kraja šezdesetih, kada je postala deo kompanije Simens (nem. Siemens).

U okviru saradnje kompanije IBM i univerziteta Harvard, tim *Hauarda Aikena (engl. Howard Hathaway Aiken, 1900—1973)* završio je 1944. godine uređaj *Harvard Mark I.* Ovaj uređaj čitao je instrukcije sa bušene papirne trake, imao je preko 760000 delova, dužinu 17m i visinu 2.4m, težinu od preko 4.5 tone. Mark I mogao je da pohrani u memoriji (korišćenjem elekromehaničkih prekidača) 72 broja od po 23 dekadne cifre. Sabiranje i oduzimanje dva broja trajalo je trećinu sekunde, množenje šest, a deljenje petnaest.

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

Elektronski računari. Elektronski računari koriste se od kraja tridesetih godina dvadesetog veka do danas.

Jedan od prvih takvih računara (specijalnu namene — rešavanje linearnih jednačina) 1939. godine napravili su *Atanasov (engl. John Vincent Atanasoff, 1903–1995)* i *Beri (engl. Clifford Edward Berry, 1918 — 1963).* Mašina je prva koristila binarni sistem i električne kondenzatore (engl. capacitor) za skladištenje bitova — sistem koji se u svojim savremenim varijantama koristi i danas u okviru tzv. DRAM memorije. Međutim, mašina nije bila programabilna.

Krajem Drugog svetskog rada, u Engleskoj je tim u kojem je bio i *Alen Tjuring (engl. Alan Turing, 1912–1954)* konstruisao računar *Kolosus (engl. Colossus* namenjen dešifrovanju nemačkih poruka. Računar je omogućio razbijanje nemačke *Enigma* šifre čime su saveznici uspeli da presretnu komunikaciju nemačke podmorničke flote, što je značajno uticalo na ishod drugog svetskog rata.

U periodu između 1943. i 1946. od strane američke vojske i univerziteta u Pensilvaniji, tima koji su predvodili *Džon Moučli (John William Mauchly, 1907—1980)* i *Džej Ekert (engl. J. Presper Eckert, 1919–1995)* konstruisao je prvi elektronski računar opšte namene — *ENIAC ("Electronic Numerical Inte-*

grator and Calculator"). Imao je 1700 vakuumskih cevi, 30 tona, dužine 30m. 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 i nedeljama.

1.2 Računari Fon Nojmanove arhitekture

Kao što je već ranije rečeno, najranije mašine za računanje nisu bile programabilne, odnosno radile su po unapred fiksiranom programu. Takva arhitektura se i danas koristi kod nekih jednostavnijih mašina. Tako, na primer, današnji digitroni (džepni kalkulatori) predstavljaju mašine koje rade po unapred fiksiranom programu. Najraniji elektronski računari nisu programirani u današnjem smislu te reči, već su suštinski redizajnirani da bi izvršavali nove zadatke. Tako su, na primer, operaterima bile potrebne nedelje da bi prespojili žice u okviru kompleksnog sistema ENIAC i tako ga instruisali da izvršava novi zadatak.

Potpuna konceptualna promena došla je kasnih 1940-tih, sa pojavom računara koji programe na osnovu kojih rade čuvaju u memoriji zajedno sa podacima. Iako ideje za ovaj koncept datiraju još od Čarlsa Bebidža i njegove analitčke mašine i nastavljaju se kroz radove Tjuringa, Ekerta i Moučlija, za rodonačelnika ovakve arhitekture računara smatra se *Džon Fon Nojman (engl. John Von Neumann, 1903–1957).* Fon Nojman se u ulozi konsultanta priključio timu Ekerta i Moučlija, i 1945. je u svom izveštaju o računaru *EDVAC ("Electronic Discrete Variable Automatic Computer")* opisao arhitekturu računara koja se i dan danas koristi u najvećem broju savremenih računara i u kojoj se programi mogu učitavati isto kao i podaci za obradu. Računar EDVAC, naslednik računara ENIAC, koristio je binarni zapis brojeva, u memoriju je mogao da upiše 1000 44-bitnih podataka i, što je najvažnije, omogućavao učitavanje podataka u memoriju.

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.

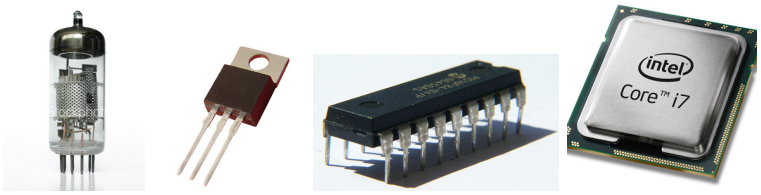
Procesor se sastoji od *kontrolne jedinice (engl. Control Unit)* koja upravlja njegovim radom i *aritmetičko-logičke jedinice (engl. Arithmetic Logic Unit)* koja je zadužena za izvođenje aritmetičkih operacija (sabiranje, oduzimanje, množenje, poređenje, ...) i logičkih operacija (konjunkcija, negacija, ...) nad brojevima. Procesor sadrži i određeni, manji broj, *registara* koji privremeno mogu da čuvaju podatke. Registri su obično fiksirane širine (8 bita[3], 16 bita, 32 bita, 64 bita). Aritmetičko logička jedinica sprovodi operacije nad podacima koji su smešteni u registre računara i rezultate ponovo smešta u registre

[3]bit (skraćeno od engl. *binary digit*) označava jednu binarnu cifru. Ako registar ima n bita, u njemu je moguće skladištiti n binarnih cifara.)

računara.

Memorija je linearno uređeni niz registara (najčešće bajtova[4]), pri čemu svaki registar ima svoju adresu. U memoriji se čuvaju podaci, ali i programi. I podaci i programi se zapisuju isključivo kao binari sadržaj i nema nikakve suštinske razlike između zapisa programa i zapisa podataka. Podaci se mogu prenositi između registara procesora i memorije (ponekad isključivo preko specijalizovanog registra koji se naziva akumulator). Sva obrada podataka se vrši isključivo u procesoru. Programi su predstavljeni nizom elementarnih instrukcija (kojima se procesoru zadaje koju akciju ili operaciju da izvrši). Kontrolna jedinica procesora dekodira instrukciju po instrukciju programa upisanih u memoriji i na osnovu njih određuje sledeću akciju sistema (npr. izvrši prenos podataka iz procesora na određenu memorijsku adresu, uradi određenu aritmetičku operaciju nad sadržajem u registrima procesora, ako je sadržaj dva registra jedak nakon ove izvrši instrukciju koja se nalazi na zadatoj memorijskoj adresi i slično).

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



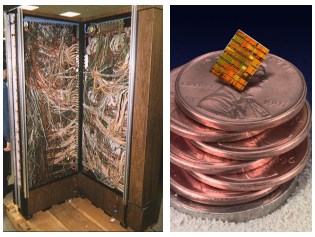
Slika 1.4: Osnovni gradivni elementi korišćeni u četiri različite generacije računara: vakuumska cev, tranzistor, integrisano kolo i mikroprocesor

I generacija računara (od početka do kraja pedesetih) koristila je *vakuumske cevi* kao logička kola i *magnetne doboše* za memoriju. Za programiranje je, gotovo isključivo, korišćen mašinski jezik a glavne primene su bile vojne i naučne. Računari su u početku unikatni i ne postoji serijska proizvodnja. Iako je EDVAC prvi dizajniran, prvi realizovani računari fon Nojmanove arhitekture bili su *Mančesterska „Beba“ (engl. Manchester „Baby“)* — eksperimentalna mašina na kojoj je testirana tehnologija vakuumskih cevi, i njen naslednik *Mančesterski Mark 1 (engl. Manchester Mark 1)*. Još jedan pretendent na titulu prvog realizovanog računara fon Nojmanove arhitekture je i *EDSAC* izgrađen takođe u Britaniji na osnovu planova za EDVAC opisanih u fon Nojmanovom izveštaju. 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.

[4]Bajt (engl. byte) je oznaka za 8 bita.

II generacija računara (od kraja pedesetih do polovine šezdesetih) koristila je *tranzistore* umesto vakuumskih cevi. Iako je tranzistor izumljen još 1947. godine, tek sredinom pedesetih počinje da se koristi namesto vakuumskih cevi kao osnovna elektronska komponenta u okviru računara. U poređenju sa vakuumskih cevima, tranzistori su manji, i zahtevaju manje energije te se i manje greju. Tranzistori su unapredili ne samo procesore i memoriju već i spoljašnje uređaje. Počinju se koristiti magnetni diskovi i trake, započinje ideja umrežavanja računara, i čak počinje korišćenje računara u zabavne svrhe (implementirana je prva računarska igrica *Spacewar* za računar *PDP-1*). Na računarima druge generacije, umesto mašinskoj jezika, sve više se koristio asembler, a razvijeni su i prvi jezici višeg nivoa (Fortran, Cobol, Lisp). Kompanija IBM dominira tržištem. Smatra se da je računar *IBM 1401* obuhvata oko trećine tržišta i prodat je u preko deset hiljada primeraka.

III generacija računara (od polovine šezdesetih do kraja sedamdesetih) bila je zasnovana na *integrisanim kolima* smeštenim na silikonski *(mikro)čipovima.*

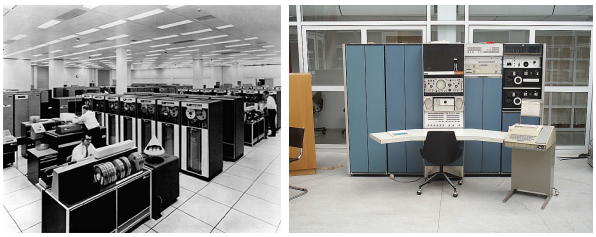


Slika 1.5: Integrisana kola dovela su do minijaturizacije i kompleksni žičani spojevi su mogli biti realizovani na izuzetno maloj površini.

Nova tehnologija omogućila je poslovnu primenu računara u mnogim oblastima. Ovom erom dominiraju *mejnfrejm (engl. mainframe)* računari koji su bili izrazito moćni za to doba, čija se brzina merila milionima instrukcija u sekundi (engl. 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. Sa preko 90%, kompanija IBM je imala apsolutnu dominaciju na tržištu ovih računara. Najčuveniji serije su *IBM 700/7000* i kasnije *IBM System/360.* Za razliku od ranijeg sistema unosa podataka isključivo preko bušenih kartica, ovi računari uvode i dodatni sistem *deljenja vremena (engl. timesharing)* koji dragoceno procesorsko vreme raspodeljuje i daje na uslugu različitim korisnicima istovremeno priključenim na računar koji sa računarom komuniciraju kroz specijalizovane *terminale.* Koriste se različiti operativni sistemi, uglavnom razivijeni u okviru kompanije IBM.

Pored mejnfrejm računara, koriste se i *mini računari (engl. minicomputers)* koji se mogu smatrati prvim oblikom ličnih (personalnih) računara. Procesor je, uglavnom, bio dodeljivan isključivo jednom korisniku. Obično su bili veličine ormana i retko su ih posedovali pojedinci (te se ne smatraju kućnim računarima).

Tržištem ovih računara dominira kompanija *DEC – Digital Equipment Corporation* sa svojim serijama računara *PDP* i *VAX*. Za ove računare, obično se vezuje operativni sistem *Unix* i programski jezik *C* razvijeni u *Belovim laboratorijama (engl. Bell Laboratories)*, kao i *hakerska*[5] 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 kraja sedamdesetih) zasnovana je na visoko integrisanim kolima kod kojih je na hiljade kola smešeno na jedan silikonski čip. 1971. godine, u kompaniji Intel napravljen je prvi mikroprocesor *Intel 4004* — celokupna centralna procesorska jedinica smeštena na jednom čipu. Iako prvobitno namenjena za ugradnju u kalkulatore, ova tehnologija omogućila je razvoj brzih a malih računara pogodnih za kućnu upotrebu.

1975. godine, časopis *„Popular electronics"* čitaocima nudi mogućnost naručivanja delova za sklapanje mikroračunara *MITS Altair 8800* zasnovanog na Intel 8080 procesoru (nasledniku procesora 4004). Prijem ovog uređaja među onima koji su se elektronikom bavili iz hobija je bio 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 još vezuje i za nastanak kompanije *Microsoft* — i danas jedne od najdominantnijih 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. Značajno je pomenuti *Apple* — prvi računar koji je prodavan već sklopljen i prodavan kao takav i na čijim temeljima je nastala kompanija *Apple* koja je danas jedan od lidera na tržištu računarske opreme.

Kućni računari počinju sve više da se koriste — uglavnom od strane entuzijasta za jednostavnije obrade podataka, učenje programiranja i igranje računarskih igrica. 1977. godine, pojavljuje se kompanija *Commodore* sa svojim računarom *Commodore PET* koji je zabeležio veliki uspeh. 1982. godine, pojavljuje se *Commodore 64* jedan od najuspešnijih jeftinih računara za kućnu upotrebu. Iz

[5]Termin *haker* se obično koristi za osobe koje neovlašćeno pristupaju računarskim sistemima, međutim, 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.

ove kompanije, čuvena je i serija *Amiga* računara sa kraja 1980-tih i početka 1990-tih. Pored companije *Commodore*, značajni proizvođači računara toga doba bili su i *Sinclair*, pre svega sa modelom *ZX Spectrum* koji je bio najprodavaniji u Velikoj Britaniji, zatim kompanija *Atari*, zatim *Amstrad*, itd. Računari su obično jeftiniji, imaju skromnije karakteristike i priključuju se na televizijske ekrane.

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

1981. godine, najznačajnija računarska kompanija toga doba se zvanično uključuje na tržište kućnih računara — pojavljuje se model *IBM PC 5150*, poznatiji jednostavno kao *IBM PC* ili *PC* (*engl. Personal computer*). Zasnovan na Intelovom procesoru *Intel 8088*, ovaj računar veoma brzo zauzima tržište računara za ličnu poslovnu upotrebu (obrada teksta, tabelarna proračunavanja, ...). S obzirom na veliki uspeh IBM PC računara, uskoro se pojavljuje određen broj klonova — računara koji nisu proizvedeni u okviru kompanije IBM, ali koji su kompatiblini sa IBM PC računarima. PC arhitektura polako postaje standard za kućne računare. Sredinom 1980-tih, pojavom naprednijih grafičkih (VGA) i zvučnih (SoundBlaster) kartica, IBM PC (i njegovi klonovi) stiču mogućnost naprednijih multimedijalnih aplikacija i polako počinju da sa tržišta istiskuju sve ostale proizvođače. I naslednici originalnog IBM PC (IBM PC/XT, IBM PC/AT) računara 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*. Naravno, PC arhitektura podržava i korišćenje drugih operativnih sistema (na primer, GNU/Linux).

Jedini veliki takmac 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 koji je koristio grafički korisnički interfejs i miša. Operativni sistem koji se i danas koristi na Apple računarima je *Mac OS*.

Tržištem današnjih računara dominiraju računari zasnovani na *PC* arhitekturi kao i *Apple Mac* računari. Pored stonih (engl. destkop) računara pojavljuju se i prenosni (engl. notebook ili laptop) računari. U najnovije vreme, javlja se trend tehnološke konvergencije i dolazi do stapanja različitih uređaja u jedinstvene celine. Tako smo svedoci pojave tabličnih (engl. tablet) računara i pametnih mobilnih telefona (engl. smartphone).

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

1.3 Organizacija savremenih računara

Iako u osnovi savremenih računarskih sistema i dalje leži Fon Nojmanova mašina (procesor i memorija), oni se danas ne mogu zamisliti bez niza modernih hardverskih komponenti koje olakšavaju rad sa računarom.

Hardver. Osnovu hardvera savremenih računara čine sledeće komponente:

Procesori - Procesor je jedna od dve centralne komponenta svakog računarskog sistema Fon Nojmanove arhitekture. Najznačajniji proizvođači mikroprocesora za PC računare danas su Intel i AMD. Važna karakteristika procesora je njegov radni takt (danas je obično nekoliko gigaherca (GHz))

— veći radni takt obično omogućava izvršavanje većeg broja operacija u jedinici vremena.

Glavna (radna ili operativna) memorija - Memorija je jedna od dve centralne komponenta svakog računarskog sistema Fon Nojmanove arhitekture. S obzirom da se kod ove memorije može sadržaju može pristupati u potpuno slučajnom redosledu, ova memorija se često naziva i RAM (engl. random access memory). Osnovni parametri memorija su kapacitet (danas obično meren gigabajtima (GB)) i brzina.

Okruženje - matična ploča, magistrala, ...

Memorijska hijerarhija - spoljašnje memorije (diskovi, fleš memorije, ...), keš memorija

Ulazni uređaji - tastature, miševi, skeneri, ...

Izlazni uređaji - grafički adapteri, monitori, štampači, ...

Softver. Osnovna podela softvera je podela na:

Sistemski softver - operativni sistemi (jezgro, uslužni programi), alati za programiranje (prevodioci, dibageri, profajleri, integrisana okruženja), ...

Aplikativni softver - pregledači Veba, klijenti elektronske pošte, kancelarijski softver, video igrice, multimedijalni softver, ...

1.4 Oblasti savremenog računarstva

Savremeno računarstvo ima mnogo podoblasti. Zbog njihove ispretpletenosti 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):

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

- *Veštačka inteligencija* (rešavanje problema u kojima se javlja kombinatorna eksplozija);
- *Robotika* (algoritmi za kotrolu 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);

Pitanja za vežbu

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

Pitanje 1.2. *Kakva je veza između tkačkih razboja s početka XIX veka i računara?*

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

Pitanje 1.4. *Koji je značaj Čarlsa Bebidža za razvoj računarstva i programiranja? U kom veku je dizajnirao svoje računske mašine? Koja od njegovih mašina je bila programabilna? Ko se smatra prvim programerom?*

Pitanje 1.5. *Na koji način je Herman Holerit doprineo izvršavanju popisa stanovnika u SAD 1890? Koja čuvena kompanija je nastala iz kompanije koju je Holerit osnovao?*

Pitanje 1.6. *Kada su nastali prvi elektronski računari? Nabrojati nekoliko najznačajnijih.*

Pitanje 1.7. *Na koji način je programiran računar ENIAC, a na koji računar EDVAC?*

Pitanje 1.8. *Koji su osnovne komponente računara Fon Nojmanove arhitekture? Gde se vrši obrada podataka u okviru računara Fon Nojmanove arhitekture? Šta se smešta u memoriju računara Fon Nojmanove arhitekture?*

Pitanje 1.9. *Šta su procesorske instrukcije? Navesti nekoliko primera.*

Pitanje 1.10. *Koji su uobičajeni delovi procesora? Da li se u okviru samog procesora nalazi određena količina memorije za smeštanje podataka? Kako se naziva?*

Pitanje 1.11. *Ukratko opisati osnovne elektronske komponente svake generacije računara savremenih elektronskih računara?*

Pitanje 1.12. *Koji tipovi računara se koriste u okviru III generacije?*

Pitanje 1.13. *Koji je najprodavaniji model firme Commodore?*

Glava 2

Reprezentacija podataka u računarima

Današnji računari su *digitalni*. Ovo znači da su svi podaci koji su u njima zapisani, zapisani isključivo kao celi brojevi tj. 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 medija za zapis podataka omogućava izgradnju elemenata koji imaju dva diskretna stanja što za zapis podataka daje azbuku od samo 2 različita simbola. Tako, na primer, ukoliko između dve tačke postoji napon viši od određenog praga, onda smatramo da je tom paru tačaka odgovara vrednost 1, a inače mu odgovara vrednost 0. Po istom principu, polje hard diska može biti 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 cifru 0, a ono koje jeste izbušesno cifru 1. U nastavku će biti pokazano da je azbuka od samo dva simbola u potpunosti dovoljna za predstavljanje svih vrsta brojeva, pa samim tim i svih digitalno zapisanih podataka.

2.1 Analogni i digitalni podaci i digitalni računari

Kontinualna priroda signala. Većina podataka koje računari zapisuju nastaju zapisivanjem prirodnih signala. Najznačajnije primeri signala, svakako, predstavljaju zvuk i slika, ali se pod signalima podrazumevaju i ultrazvučni signali, EKG signali, zračenja različite vrste itd.

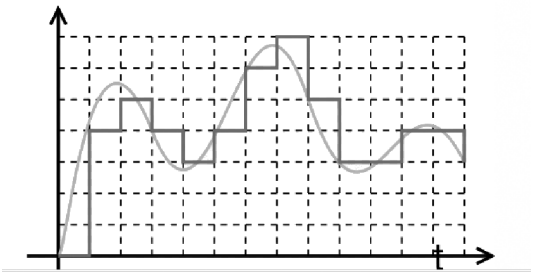
Signali koji nas okružuju u prirodi u većini slučajeva se prirodno mogu predstaviti neprekidnim funkcijama. Na primer, zvučni signal predstavlja promenu pritiska vazduha u zadatoj tački i to kao neprekidnu funkciju vremena. Slika predstavlja intenzitet svetlosti određene boje (tj. određene talasne dužine) u datom vremenskom trenutku i to kao neprekidnu funkciju 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, kao što i samo ime kaže, analogni zapis uspostavlja *analogiju* između određenog svojstva medijuma na kome je signal zapisan i samog signala koji je zapisan.

Osnovna prednost analogne tehnologije je činjenica da su ulaganja potrebna da bi se dobio zapis veoma mala, ukoliko se zadovoljimo relativno niskim kvalitetom. Tako su, na primer, još drevni narodi mogli da naprave nekakav zapis zvuka uz pomoću jednostavne igle prikačene na trepereću membranu.

Osnovi problem analogne tehnologije je što je izrazito teško na medijumu napraviti skoro identičan zapis signala koji se zapisuje. Takođe, problem predstavlja i inherentna nestalnost medijuma, njegova promenljivost tokom vremena i podložnost spoljašnjim uticajima. S obzirom da male varijacije medijuma direktno dovode do varijacije zapisanog signala, vremenom neizbežno dolazi do pada kvaliteta analogno zapisanog signala. Obrada analogno zapisanih signala je izuzetno 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 odnosno 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 da izmerene vrednosti takođe dolaze sa kontinualne skale, 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. Pita-

nje koje se neizbežno postavlja je koliko često je potrebno vršiti merenje, kako bi se polazni kontinualni signal mogao verno rekonstruisati. Odgovor daje teorema o odabiranju koja kaže da je signal dovoljno meriti dva puta češće od najveće frekvencije koja sa u njemu javlja. Na primer, pošto čovekovo uho čuje frekvencije do 20KHz, dovoljno je da frekvencija odabiranja bude oko 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 prouzrokuje 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 razpoznati.

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

Obrada digitalno zapisanih podataka se svodi na matematičku manipulaciju brojevima i ne zahteva više korišćenje specijalizovanih mašina.

Osnovi problem implementacije digitalnog zapisa predstavlja činjenica da je neophodno imati veoma razvijenu tehnologiju da bi se uopšte stiglo do iole upotrebljivog zapisa. 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. Međutim, kada se došlo do tehnološkog nivoa koji omogućava digitalni zapis, prednosti koje je donela digitalna tehnologija su izuzetne.

2.2 Zapis brojeva

Procesom 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, s obzirom na njihovu tehnološku osnovu, 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 omogućavaju jednostavnu konverziju između njih i binarnog sistema. Naglasimo 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 broj, bez

obzira u kom sistemu su ova tri broja zapisana.

2.2.1 Neoznačeni brojevi

Pod *neoznačenim brojevima* podrazumeva se neoznačeni zapis prirodnih brojeva. Oni su nenegativni, te se znak 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 $0 \leq a_i < b$.

Klasična definicija značenja zapisa broja u osnovi b je:

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

Ova se definicija može iskoristiti za određivanje vrednosti datog zapisa:

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

Na primer:
$(101101)_2 = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2 + 1 = 32 + 8 + 4 + 1 = 45$,
$(3245)_8 = 3{\cdot}8^3 + 2{\cdot}8^2 + 4{\cdot}8 + 5 = 3{\cdot}512 + 2{\cdot}64 + 4{\cdot}8 + 5 = 1536 + 128 + 32 + 5 = 1701$.

Primetimo da je za izračunavanje vrednosti nekog $(n+1)$-tocifrenog zapisa potrebno 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 $2n$ tj. na dva množenja po svakoj cifri. 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 ukoliko se koristi *Hornerova shema*:

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

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

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

```
x := 0
za svako i od n unazad do 0
    x := x · b + a_i
```

Za ilustraciju Hornerovog postupka, odredimo koji je broj zapisan sa $(9876)_{10}$.

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

Primetimo da 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 to ilustruje za 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 klasičan, jer je u svakom koraku dovoljno izvršiti samo jedno množenje i jedno sabiranje.

Određivanje zapisa datog broja. Pošto za svaku cifru a_i u zapisu broja u osnovi b važi da je $0 \leq a_i < b$, na osnovu Hornerove definicije broja $x = (a_n\, a_{n-1}\, \ldots\, a_1\, a_0)_b$, jasno je da mora da važi da je a_0 ostatak pri deljenju broja x osnovom b. Takođe, broj $(a_n\, a_{n-1}\, \ldots\, a_1)$ predstavlja celobrojni količnik pri istom deljenju. Dakle, izračunavanjem celobrojnog količnika i ostatka pri deljenju sa b, određena je poslednja cifra broja x i broj koji se dobije 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 tablele 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.

Primetimo da su ovaj algoritam i Hornerov algoritam međusobno komplementarni 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. Pritom, heksadekadni sistem omogućava da se binarni sadržaj memorije kompaktno zapiše (uz korišćenje znatno manjeg broja cifara). Tako se, na primer, 32-bitni sadržaj može zapisati korišćenjem samo 8 heksadekadnih cifara.

$(1011\,0110\,0111\,1100\,0010\,1001\,1111\,0001)_2 = (B67C29F1)_{16}$

Prevođenje se vrši 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

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

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

2.2.2 Označeni brojevi

Pod označenim brojevima se podrazumevaju celi brojevi kod kojih se u zapis uključuje i zapisivanje znaka broja (+ ili −). S obzirom da računari koriste binarni brojevni sistem, razmatraćemo samo zapise označenih brojeva u binarnom brojevnom sistemu. Postoji više tehnika zapisivanja označenih brojeva od kojih su najčešće u upotrebi *označena apsolutna vrednost* i *potpuni komplement*.

Označena apsolutna vrednost. Zapis se formira tako što se u registar unapred fiksirane dužine n, na prvu poziciju upiše znak broja, a na preostalih $n-1$ poziciju upiše zapis apsolutne vrednosti broja. Pošto se za zapis koriste samo dva simbola (0 i 1), poštuje se dogovor da se znak + zapisuje simbolom 0, a znak − se zapisuje 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 načni.

Potpuni komplement. Ovaj zapis 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.

Tako se, na primer, broj +100 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.

Zapis broja $(100\ldots00)_2^n$ je sam sebi komplementaran, a pošto počinje cifrom 1, uzima se da on predstavlja zapis najmanjeg negativnog broja. Tako dobijamo da je u zapisu potpunog komplementa $(\ldots)_2^n$ moguće zapisati brojeve od -2^{n-1} do $2^{n-1}-1$. U sledećoj tabeli su dati rasponi za najčešće korišćene zapise:

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

2.2.3 Razlomljeni brojevi

Za zapis razlomljenih brojeva se najčešće koristi tzv. *pokretni zarez* (engl. floating point). 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 $b = 16$). Standard koji definiše zapis brojeva u pokretnom zarezu se naziva *IEEE 754*.

2.3 Zapis teksta

ISO 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 obično tekst zamišljamo kao dvodimenzioni objekat, u računarima se tekst predstavlja kao jednodimenzioni (linearni) niz karaktera koji pripadaju određenom unapred fiksiranom skupu karaktera. Za zapis teksta, dakle, neophodno je uvesti specijalne karaktere 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 određeni (neoznačeni) ceo broj i to na unapred dogovoreni način (a koji se interno u računarima zapisuje binarno). Ovi brojevi se nazivaju *kodovima karaktera* (engl. character codes). Tehnička ograničenja ranih računara kao i neravnomeran razvoj računarstva između različitih nacija, doveli su do toga da postoji više različitih standardnih tablica koje dodeljuju numeričke kodove karakterima. S obzirom na broj različitih bitova potrebnih za kodiranje određenih karaktera, razlikovaćemo 7-bitne kodove, 8-bitne kodove, 16-bitne kodove, 32-bitne kodove, kao i kodiranja promenljive dužine koji različitim karakterima dodeljuju kodove različite dužine.

Potrebno je napraviti veoma jasnu razliku 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). Korespodencija izmedju karaktera i glifova ne mora biti jednoznačna. Naime, softver koji prikazuje tekst može više karaktera predstaviti jednim glifom (to su takozvane *ligature*), dok 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 odgovarajući način, bez obzira što je ispravno kodiran. Tako, fontovi koji se standardno 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.

Englesko govorno područje. Tokom razvoja računarstva, broj karaktera koje je bilo poželjno kodirati je postajao sve veći i veći. Pošto je u početku razvoja englesko govorno područje bilo dominantno osnovno je bilo predstaviti sledeće karaktere:

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

Specijalnim karakterima se najčešće ne pridružuje zaseban grafički lik.

Standardne tabele kodova ovih karaktera su se pojavile još tokom šezdesetih godina XX veka. Najpoznatije 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 *ANSI (American National Standard Institute)* koji definiše sedmobitan zapis koda svakog karaktera što daje mogućnost zapisivanja ukupno 128 različitih karaktera, pri čemu nekoliko kodova ima dozvoljeno slobodno korišćenje. *ISO (International Standard Organization)* takođe delimično definiše ASCII tablicu kao deo standarda *ISO 646 (US)*.

- Prva 32 karaktera $(00)_{16}$ do $(1F)_{16}$ su specijalni kontrolni karakteri.
- 95 karaktera ima pridružene grafičke likove (engl. printable characters)
- Cifre 0-9 su predstavljene kodovima $(30)_{16}$ do $(39)_{16}$ tako da se njihov ASCII zapis jednostavno dobija dodavanjem prefiksa 011 na njihov binarni zapis.

	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

- Kodovi 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.

Zanimljivo je napomenuti i da 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 tj. $(0D)_{16}$ i LF tj. $(0A)_{16}$, operativni sistem Unix i njegovi derivati podrazumevaju da se koristi samo karakter LF, dok MacOS podrazumeva korišćenje samo karaktera CR.

Nacionalne varijante ASCII tablice i ISO 646. Tokom osamdesetih godina XX veka, Jugoslovenski zavod za standarde definiše *YU-ASCII (YUSCII, JUS I.B1.002, JUS I.B1.003)* kao deo standarda ISO 646, tako što kodove koji imaju slobodno korišćenje (a koje u ASCII tablici uobičajeno kodiraju zagrade i određene interpunkcijske znakove) dodeljuje 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	~	$(7E)_{16}$

YUSCII je takođe deo standarda ISO 646. 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 tablice. Podaci se u računaru obično zapisuju bajt po bajt. Sa obzirom da je ASCII sedmobitan 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. Na žalost, ovih dodatnih 128 kodova nisu dovoljni

kako bi se kodirali svi karakteri koji su potrebni za zapis tekstova van engleskog govornog područja. Zbog toga je odlučeno da se umesto jedinstvene tabele koja bi proširivala ASCII na 256 karaktera standardizuje nekoliko tabela koje ovo čine, pri čemu svaka od tabela sadrži karaktere potrebne za zapis određenog jezika odnosno određene grupe jezika.

Problem je što postoji dvostruka standardizacija ovako kreiranih kodnih stranica i to od strane ISO (International Standard Organization) i od strane značajnih industrijskih korporacija, pre svega kompanije *Microsoft*.

ISO definiše familiju 8-bitnih kodnih stranica koje nose zajedničku oznaku *ISO/IEC 8859*. Kodovi $(00)_{16}$ do $(1F)_{16}$, $(7F)_{16}$ i od $(80)_{16}$ do $(9F)_{16}$ ostaju 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 definiše familju 8-bitnih stranica koje se označavaju kao *Windows-125x* (ove stranice se nekada nazivaju i *ANSI*). Za srpski jezik, značajne su kodne stranice:

Windows-1250	centralno i istočno-evropski jezici
Windows-1251	ćirilica većine slovenskih jezika

	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

Unicode. Iako navedene kodne strane omogućuju kodiranje tekstova koji nisu na engleskom jeziku, nije moguće, na primer, u istom tekstu mešati ćirilicu i našu latinicu. Takođe, za azijske jezike nije dovoljno 256 mesta za zapis svih karaktera. Pošto je kapacitet računara narastao, postepeno se krenulo sa standardizacijom skupova karaktera koji karaktere kodiraju sa više od jednog bajta. Kasnih osamdesetih godina XX veka, dve velike organizacije su pokušale standardizaciju tzv. Univerzalnog skupa karaktera (engl. Universal Character Set — UCS). To su bili ISO, kroz standard 10646 i projekat Unicode organizovan

	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

i finansiran uglavnom od strane američkih firmi koje su se bavile proizvodnjom višejezičkog softvera (Xerox Parc, Apple, Sun Microsystems, Microsoft, ...).

ISO 10646 je zamišljen kao četvorobajtni standard. Pri tome se prvih 65536 karaktera koriste kao osnovni višejezični skup karaktera, dok je ostali prostor ostavljen kao proširenje za drevne jezike, celokupnu 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 - 16.

	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

Početna verzija Unicode svakom karakteru dodeljuje dvobajtni kôd (tako da kôd svakog karaktera sadrži tačno 4 heksadekadne cifre). Ovim je moguće dodeliti kodove za ukupno 64K = 65536 različita karaktera. Vremenom se shvatilo da dva bajta neće biti dovoljno za zapis baš svih karaktera koji se na planeti koriste, tako da je odlučeno da se skup karaktera proširi i Unicode danas dodeljuje kodove od $(000000)_{16}$ do $(10FFFF)_{16}$ podeljenih u 16 tzv. ravni, pri čemu svaka ravan sadrži 64K karaktera. U upotrebi je najviše *osnovna višejezična ravan* (engl. basic multilingual plane) koja sadrži većinu danas korišćenih karaktera (uključujući čak i CJK — Chinese, Japanese, Korean — karaktere koji se najčešće koriste) čiji su kodovi između $(0000)_{16}$ i $(FFFF)_{16}$.

Vremenom su se pomenuta dva projekta 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 dodatno definišu.

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

UTF-8. Latinični tekstovi kodirani preko UCS-2 standarda sadrže veliki broj nula, koje obično u operativnim sistemima poput UNIX-a i u programskom jeziku C imaju specijalno značenje. Iz istog razloga 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 $(0041)_{16} = (0000\,0000\ 0100\,0001)_2$, tako da se na osnovu pvog reda prethodne tabele zapisuje kao $(01000001)_2 = (41)_{16}$ u UTF-8 kodiranju. Karakter Š ima Unicode kôd $(0160)_{16} = (0000\,0001\ 0110\,0000)_2$. Na njega se primenjuje drugi red prethodne tablice i dobija se da je njegov UTF-8 zapis $(1100\,0101\ 1010\,0000)_2 = (C5A0)_{16}$.

2.4 Zapis multimedijalnih sadržaja

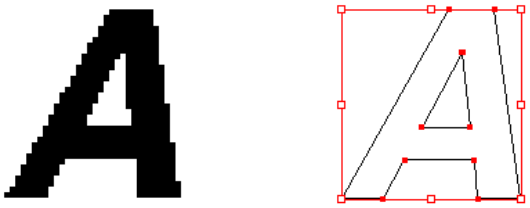
Računari imaju sve veću ulogu i većini oblasti svakodnevnog života. Od mašina koje su pre svega služile za izvođenje vojnih i naučnih proračunavanja, računari su postali i sredstvo za kućnu zabavu (gledanje filmova, slušanje muzike), izvor informacija (Internet, Veb) i nezaobilazno sredstvo u komunikaciji (elektronska pošta (engl. e-mail), ćaskanje (engl. chat, instant messaging), video konferencije, telefoniranje korišćenjem Interneta (skr. VoIP), ...). Ovako nagli razvoj i proširivanje osnovne namene računara je prouzrokovan velikim porastom količine multimedijalnih informacija (slika, zvuka, filmova, ...) koje su zapisane u digitalnom formatu. Ovo je opet prouzrokovano tehnološkim napretkom 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*.

Vektorski zapis slika podrazumeva da se slika sastoji od konačnog broja geometrijskih oblika (tačaka, linija, krivih, poligona), pri čemu se svaki oblik predstavlja svojim koordinatama. 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 da se objekti mogu nezavisno jedan od drugoga pomerati, menjati, dodavati i uklanjati.



Slika 2.8: Odnos rasterske (levo) i vektorske (desno) grafike

Rasterski zapis slika podrazumeva da je slika predstavljena pravougaonom matricom sitnih komponenti koji se nazivaju pikseli (engl. pixel - PICture ELement). Svaki piksel je opisan isključivo oznakom njegove boje. 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 (fotaparati, skeneri) koriste rasterski zapis.

Modeli boja. Za predstavljanje crno-belih slika, dovoljno je boju predstaviti isključivo količinom svetlosti. Različite količine svetlosti se diskretizuju u konačan broj nivoa nivoa osvetljenja i se dobija time odgovarajući broj nijansi sive boje. Ovakav model 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 isključivo sa dve boje (na primer, skenirani tekst nekog dokumenta) koristi se model pod nazivom *Duotone*. Boja se tada predstavlja sa jednim bitom.

Mešanjem crvene (R), zelene (G) i plave (B) svetlosti se dobijaju sve ostale boje. Tako se, na primer, mešanjem crvene i zelene svetlosti dobija žuta svetlost. Bela svetlost se dobija mešanjem sve tri osnovne komponente, dok crna boja predstavlja odsustvo svetlosti. Imajući ovo u vidu, informacija o boji se dobija beleženjem informacije o količini crvene, plave i zelene svetlosti. Ovaj model se naziva *RGB model (aditivni model)*. RGB model boja se koristi kod uređaja koji boje prikazuju mešanjem svetlosti (monitori, projektori, ...). Ukoliko se za informaciju o svakoj komponenti pojedinačno koristi 1 bajt, ukupan broj bajtova za zapis informacije o boji je 3 što daje $2^{24} = 16M = 16777216$ različitih boja. Ovaj model se često naziva *TrueColor* model boja.

Nasuprot aditivnog RGB modela boja, kod koga se bela boja dobija sabiranjem svetlosti tri osnovne komponente, u štampi se koristi subtraktivni CMY (Cyan-Magenta-Yellow) model boje kod koga se boje dobijaju mešanjima obojenih pigmenata na belom papiru. S obzirom da se potpuno crna boja veoma teško dobija mešanjem drugih pigmenata, 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.

Svaka boja se predstavlja preko Hue (H) komponente koja predstavlja ton boje na osnovu kojega i imenujemo boje, zatim Saturation (S) komponente koja predstavlja zasićenost boje odnosno njenu ,,jarkost" i Lightness (L), Value (V) ili Brightness (B) koja predstavlja osvetljenost boje.

Formati zapisa rasterskih slika. Rasterske slike su predstavljene matricom piksela, pri čemu se za svaki piksel čuva informacija o boji. Dimenzije ove matrice predstavljaju tzv. *apsolutnu rezoluciju* slike. Apsolutna rezolucija i model boja koji se koristi određuju broj bajtova pomoću kojih je moguće sliku predstaviti. Tako, 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 memorisanje slike. Kako bi se smanjila količina informacija potrebnih za zapis slike, pribegava se tehnikama 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, logotipova i sličnih računarski generisanih slika, dok je za kompresiju fotografija pogodan algoritam kompresije sa gubitkom JPEG.

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 čuje raspon frekvencija od nekih 20Hz do 20KHz (mada je ovo individualno), dovoljno je izvršiti odabiranje oko 40 000 puta u sekundi. Na primer, AudioCD standard koji se koristi prilikom snimanja običnih audio CD-ova, propisuje frekvenciju odabiranja 44.1KHz. Pored ovoga, za postizanje još većeg kvaliteta neki standardi (miniDV, DVD, digital TV) propisuju odabiranje na frekvenciji 48KHz. Ukoliko se snima samo ljudski govor (na primer, u mobilnoj telefoniji) frekvencije odabiranja mogu biti i znatno manje. Drugi važan parametar digitalizacije je broj bita sa kojim se zapisuje svaki pojedinačni odbirak. Najčešće se koristi 2 bajta po odbirku (16 bita) čime se dobija mogućnost zapisa 64K=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 sa 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).

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 ljudsko uho ne oseti promenu.

Pitanja i zadaci

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* 4321 *u osnovama* 2, 8 *i* 16.

Zadatak 2.3. *(a) Registar ima sadržaj*

$$1010101101001000111101010101011001101011101010101110001010010011.$$

Zapisati ovaj sadržaj heksadekadno.

(b) Registar ima sadržaj $A3BF461C89BE23D7$. *Zapisati ovaj sadržaj binarno.*

Zadatak 2.4. *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 2.5. *Ukoliko se koristi binarni zapis neoznačenih brojeva širine* 8 *bita, zapisati brojeve:*
(a) 12 *(b)* 123 *(c)* 255 *(d)* 300

Zadatak 2.6. *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 2.7. *Odrediti zapis narednih brojeva u binarnom potpunom komplementu širine* 8 *bita:*
(a) 12 *(b)* -123 *(c)* 0 *(d)* -18 *(e)* -128 *(f)* 200

Zadatak 2.8. *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 2.9. *HEX editori su programi koji omogućavaju direktno pregledanje, kreiranje i ažuriranje bajtova koji sačinjavaju sadržaja datoteka (vidi, na primer,* http://en.wikipedia.org/wiki/Hex_editor*). Korišćenjem HEX editora pregledati sadržaj nekoliko datoteka različite vrste (tekstualnih, izvršnih programa, slika, zvučnih zapisa, video zapisa, . . .).*

Zadatak 2.10. *Korišćenjem ASCII tablice odrediti kodove kojima se zapisuje tekst:* `"Programiranje 1"`. *Kodove zapisati heksadekadno, oktalno, dekadno i binarno. Šta je sa kodiranjem teksta* `Matematički fakultet`*?*

Zadatak 2.11. *Da li je moguće i koliko bajtova zauzima reč* `čačkalica` *ukoliko se kodira korišćenjem:*
(a) ASCII (b) Windows-1250 (c) ISO-8859-5 (d) ISO-8859-2
(e) Unicode (UCS-2) (f) UTF-8

Pitanje 2.12. *Koja od načina 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)?*

Zadatak 2.13. *Odrediti (heksadekadno predstavljene) kodove kojima se zapisuje tekst* `kružić` *u UCS-2 i UTF-8 kodiranjima. Rezultat proveriti korišćenjem HEX editora.*

Zadatak 2.14. *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 rezultat (uz pomoć odgovarajućih tablica koje prikazuju kodne rasporede).*

Pitanje 2.15. *Prilikom prikazivanja filma, neki program prikazuje titlove tipa* `"raèunari æe biti"`. *Objasniti u čemu je problem.*

Zadatak 2.16. *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.*

Zadatak 2.17. *Na Vebu se boje obično predstavljaju šestocifrenim heksadekadnim kodovima 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 kodom predstavlja čista žuta boja? Kako izgledaju kodovi za nijanse sive boje?*

Zadatak 2.18. *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

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 jasni algoritmi za sabiranje i za množenje prirodnih brojeva. U računarstvu, postoje jasni algoritmi za određivanje najmanjeg elementa niza, uređivanje elemenata po veličini i slično.

3.1 Formalizacije pojma algoritma

Precizni postupci za rešavanje matematičkih problema postojali su u vreme starogrčkih matematičara (npr. Euklidov algoritam za određivanje najvećeg zajedničkog delioca dva broja), a 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 precizan postupak, odnosno šta je to algoritam. U tome, bilo je dovoljno razmatrati algoritme za izračunavanje funkcija čiji su i argumenti i rezultujuće vrednosti prirodni brojevi (a svi ostali matematički algoritmi se mogu svesti na taj slučaj). Formalno zasnovan pojam

[1] Reč ,,algoritam“ ima koren u imenu persijskog astronoma i matematičara Al-Horesmija (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-Horesmi 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 značnjem ,,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 da odgovor na sva matematička pitanja. Ipak, 1930-tih, kroz radove Čerča, Tjuringa i Gedela pokazano da ovakav postupak ne može da postoji.

algoritma kasnije je omogućio da budu identifikovane i opisani problemi za koje postoje, kao i problemi za koje ne postoje algoritmi koji ih rešavaju.

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

- Tjuringove[3] mašine,
- rekurzivne funkcije[4],
- λ-račun[5],
- Postove[6] mašine,
- Markovljevi[7] algoritmi,
- Mašine sa beskonačnim registrima (eng. Unlimited Register Machines — URM)[8].

Iako koncipirani pre nastanka savremenih računara, većina ovih sistema podrazumeva određene apstraktne mašine. Naime, i u ono vreme bilo je jasno da je opis postupka dovoljno precizan ukoliko je moguće njime instruisati 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. Ipak, značajna razlika leži u činjenici da nabrojani formalizmi teže da budu što elementarniji tj. da uključe isključivo neophodne operacije i što jednostavije modele mašina, dok savremeni programski jezici, teže da budu što udobniji za programiranje i tako uključuju veliki broj operacija koje, sa teorijskog stanovišta, nisu neophodne jer se mogu definisati preko malog broja osnovnih operacija.

Ako se neka funkcija može izračunati u nekom od tih formalizama, onda kažemo da je ona *izračunljiva* u tom formalizmu. Iako su pomenuti formalizmi međusobno dosta različiti, može se dokazati da su klase funkcija izračunljivih funkcija identične za sve njih, tj. svi oni formalizuju jedinstven pojam. Zato se, umesto pojmova poput na primer Tjuring-izračunljiva ili URM-izračunljiva funkcija (i sličnih) može koristiti samo termin *izračunljiva funkcija*. Za sistem izračunavanja koji je dovoljno bogat da može da simulira Tjuringovu mašinu i tako da izvrši sva izračunavanja koja može da izvrši i Tjuringova mašina (a možda i neka druga) kaže se da je *Tjuring potpun* (engl. Turing complete). Za sistem izračunavanja koji izračunava identičnu klasu funkcija kao i Tjuringova mašina kažemo da je *Tjuring ekvivalentan* (engl. Turing equivalent). Svi navedeni formalizmi su Tjuring potpuni, ali i Tjuring ekvivalentni. Tjuringova mašina, pa i pojmovi Tjuring potpunosti i Tjuring ekvivalentnosti (a i

[3] Alan Turing (1912-1954), britanski matematičar

[4] Kurt Gödel (1906-1978), austrijsko-američki matematičar, 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).

drugi formalizmi za izračunavanje), podrazumevaju, pojednostavljeno rečeno, beskonaču raspoloživu memoriju. Zbog toga savremeni računari 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 Odnos formalnog i neformalnog pojma algoritma i 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 je sva izračunavanja koja intuitivno umemo da izvršimo zaista možemo da opišemo korišćenjem svakog od precizno definisanih formalizama izračunljivosti. Dok je odgovor potvrdan na prvo pitanje veoma verovatan, oko drugog pitanja postoji doza rezervi. Ipak, potrvrdan odgovor na oba pitanja formulisan je u okviru tzv. Čerč-Tjuringove teze[9].

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 najviše govori činjenica da je pokazano da su sve navedene formalne definicije algoritama međusobno ekvivalentne kao i da do sada nije pronađen nijedan primer efektivno izvodivog postupka koji nije moguće formalizovati u okviru pobrojanih formalnih sistema izračunavanja. Dodatno, svi poznati Tjuring potpuni sistemi su i Tjuring ekvivalentni, što učvršćuje uverenje da je Čerč-Tjuringove teza tačna.

3.3 UR mašine

U daljem tekstu, pojam izračunljivosti biće uveden i izučavan na bazi UR mašina.

UR mašine[10] (URM) su jedan od formalizama za definisanje pojma algoritma. To su apstraktne mašine koje predstavljaju matematičku idealizaciju računara.

Kao i drugi formalizmi izračunavanja, UR mašine su dizajnirane tako da koriste samo mali broj primitiva. Očigledno je da nije neophodno da formalizam za izračunavanje kao primitivnu operaciju ima 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 uvećavanja za vrednost jedan. Pored te operacije, UR mašine koriste

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

[10] Od engleskog *Unlimited Register Machine*.

samo još tri instrukcije. Sva ta četiri tipa instrukcija brojeve čitaju i upisuju ih na traku koja služi kao prostor za čuvanje podataka, tj. kao memorija mašine. Tu traku čine registari koje označavamo sa $R_1, R_2, R_3, \ldots$. Svaki od njih u svakom trenutku sadrži neki prirodan broj. Sadržaj k-tog registra (registra R_k) označavamo 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]

oznaka	naziv	efekat
$Z(m)$	nula–instrukcija	$0 \to R_m$ (tj. $r_m := 0$)
$S(m)$	instrukcija sledbenik	$r_m + 1 \to R_m$ (tj. $r_m := r_m + 1$)
$T(m,n)$	instrukcija prenosa	$r_m \to R_n$ (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. Izračunavanje 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 numeraciji).

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, kao i da rezultat treba smestiti 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.

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 sukcesivno dodaje vrednost 1 y puta, jer važi:*

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

Odgovarajući URM *program podrazumeva sledeću početnu konfiguraciju:*

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

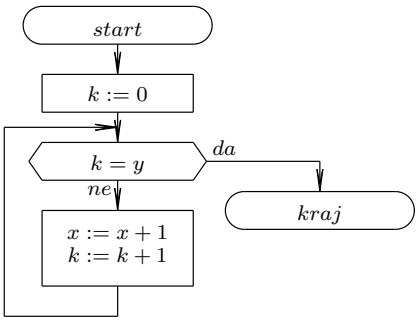
R_1	R_2	R_3	$\ldots$
x	y	0	$\ldots$

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 i skok se bezuslovno vrši.

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

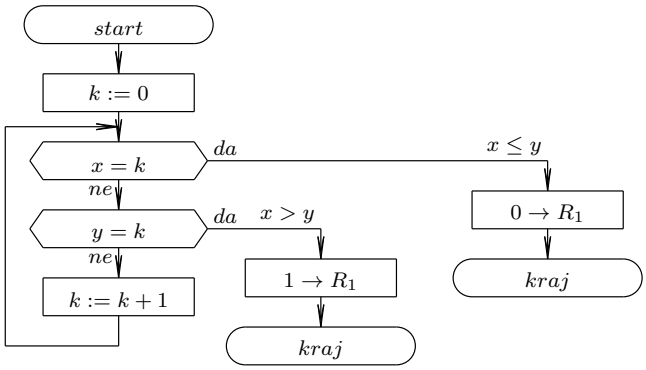
$$f(x, y) = \begin{cases} 0 & , \text{ ako } x \leq y \\ 1 & , \text{ inače} \end{cases}$$

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

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

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

[12]Broj 100 je odabran proizvoljno jer 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



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:

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

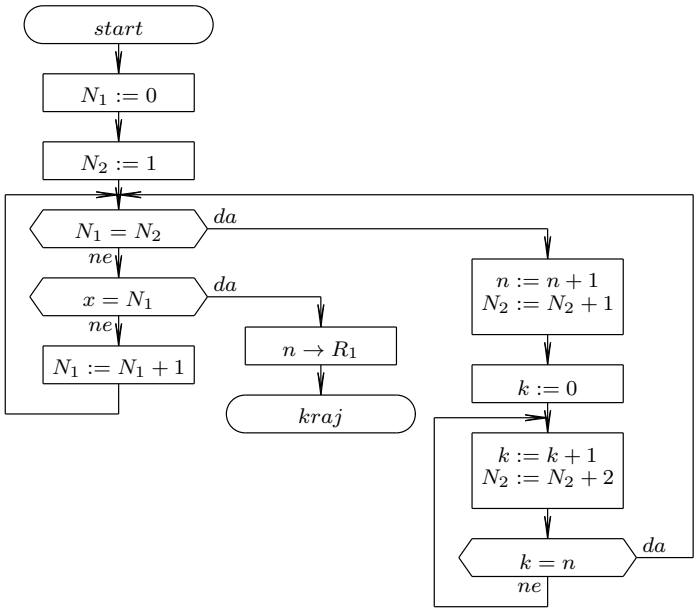
Odgovarajući URM *program podrazumeva sledeću početnu konfiguraciju:*

R_1	R_2	$\ldots$
x	0	$\ldots$

i sledeću konfiguraciju u toku svog rada:

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

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 je N_2 postaviti na vrednost $(n+1)^2$, odnosno potrebno je uvećati za $(n+1)^2 - n^2 = 2n+1$. Ovo se realizuje tako što se N_2 prvo 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.	$Z(5)$	$l := 0$
11.	$S(5)$	$l := l + 1$
12.	$S(4)$	$N_2 := N_2 + 1$
13.	$S(4)$	$N_2 := N_2 + 1$
14.	$J(5,2,4)$	$l = n?$
15.	$J(1,1,11)$	
16.	$T(2,1)$	$n \to R_1$

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

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

Nedefinisanost funkcije se postiže time što se napravi program koji se nezaustavlja.

1. $J(1,2,100)$ $x = 0?$
2. $J(1,1,1)$

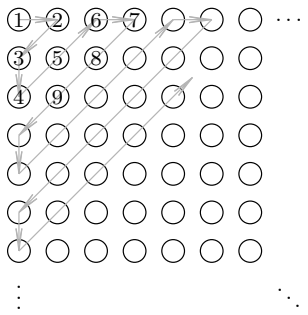
3.4 Enumeracija URM programa

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*. Primetimo da su skupovi prebrojivi ako i samo ako je njihove 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 (nem. Georg Cantor, 1845–1918) je prvi pokazao da skup realnih brojeva ima veću kardinalnost od skupa prirodnih brojeva, odnosno 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 skupova prirodnih i parnih brojeva).*

Stav 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 sledećoj slici:



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. □

Korišćenjem sličnih ideja, mogu se pokazati i sledeći stavovi.

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

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

Enumeracija URM **programa.** Pokažimo sada da URM programa ima prebrojivo mnogo.

Lema 3.1. *Postoji prebrojivo mnogo različitih* URM *instrukcija.*

Dokaz: Instrukcija tipa Z, trivijalno, ima prebrojivo mnogo (Z instrukcije se nižu sa $Z(1), Z(2), \ldots$). Slično je i sa instrukcijama tipa S. T instrukcije bijektivno odgovaraju parovima prirodnih brojeva, pa ih (na osnovu stava 3.2) ima prebrojivo mnogo[13]. Slično, J instrukcije odgovaraju bijektivno trojkama prirodnih brojeva pa ih (na osnovu stava 3.2) ima prebrojivo mnogo[14]. Skup svih instrukcija je unija ova četiri prebrojiva skupa pa je na osnovu stava 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 stava 3.2) kao konačan Dekartov stepen prebrojivog skupa instrukcija. Dakle, skup svih URM programa je prebrojiv (na osnovu stava 3.3) kao prebrojiva unija prebrojivih skupova. □

Dakle, moguće je uspostaviti bijekciju između skupa svih URM programa i skupa prirodnih brojeva, tj. moguće je svakom programu dodeliti broj koji ga jedinstveno identifikuje.

3.5 Neizračunljivost i neodlučivost

Razmatrimo narednih nekoliko problema.

1. Neka su data dva konačna skupa reči. Razmatrajmo 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. Pronaći opšti postupak (algoritam, računarski program) koji

[13] Takođe, T instrukcije je moguće nanizati i sa $T(1,1)$, $T(2,1)$, $T(2,2)$, $T(1,2)$, $T(3,1)$, $T(3,2)$, $T(3,3)$, $T(2,3)$, $T(1,3)$, $\ldots$, pri čemu obilazi matricu po kvadratima, za razliku od ranije pomenutog obilaska po dijagonalama.

[14] Takođe, moguće ih je nanizati i sa $J(1,1,1)$, $J(1,1,2)$, $J(1,2,1)$, $J(2,1,1)$, $J(2,2,1)$, $J(2,1,2)$, $J(1,2,2)$, $J(2,2,2)$, $\ldots$

za proizvoljna dva zadata skupa reči određuje da li tražena nadovezivanja postoje.[15]

2. Razmatrajmo Diofantske jednačine $p(x_1, \ldots, x_n) = 0$ gde je p polinom sa celobrojnim koeficijentima. Pronaći opšti postupak (algoritam, računarski program) kojim se može odrediti da li proizvoljna zadata diofantska jednačina ima racionalnih rešenja.[16]

3. Pronaći opšti postupak (algoritam, računarski program) koji proverava da li se proizvoljni zadati program P zaustavlja za date ulazne parametre.[17]

4. Pronaći opšti postupak koji za proizvoljni zadati skup aksioma proverava i proizvoljno zadato tvrđenje proverava da li je ono posledica tih aksioma.[18]

Za sva četiri navedena problema pokazano je da su *algoritamski nerešivi* ili *neodlučivi*. Ovo ne znači da nije moguće rešiti pojedine instance problema, već samo da ne postoji jedinstven opšti postupak koji bi mogao da reši proizvoljnu instancu problema. Iako veoma značajni, ovo nisu jedini primeri neodlučivih problema.

U nastavku će i formalno, korišćenjem formalizma UR mašina, biti precizno opisan treći od navedenih problema, tj. *halting problem*, koji je izuzetno važan za programiranje.

Halting problem. Pitanje zaustavljanja računarskih programa je jedno od najznačajnijih opštih pitanja u programiranju. 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 se može odgovoriti. No, nije očigledno da li postoji opšti postupak kojim se za proizvoljni dati program i date ulazne parametre može proveriti da li se program zaustavlja ako se pokrene sa tim parametrima na ulazu.

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

Napisati 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.*

Problem prethodne formulacije je činjenica da traženi URM program mora na ulazu da prihvati kao svoj parametar drugi URM program, što je naizgled

[15] Ovaj problem se naziva *Post's correspondence problem*, jer ga je postavio Emil Post 1946. i pokazao je da takav opšti postupak (tj. algoritam) ne postoji.

[16] Ovaj problem je 10. Hilbertov problem izložen 1900. kao deo skupa problema koje ,,matematičari XIX veka ostavljaju u amanet matematičarima XX veka. Matijaševič je 1970-tih pokazao da takav opšti postupak (tj. algoritam) ne postoji.

[17] Ovaj problem se naziva *halting problem*, a Alan Tjuring je pokazao 1936. da takav opšti postupak (tj. algoritam) ne postoji.

[18] Ovaj, već pomenut, problem, formulisao je David Hilbert 1928. godine, a nešto kasnije su rezultati Čerča, Tjuringa i Gedela pokazali da takav postupak ne postoji.

nemoguće, s obzirom da URM programi kao parametre mogu da imaju samo prirodne brojeve. Ipak, ovo je relativno jednostavno razrešava imajući u vidu da je svakom URM programu P moguće dodeliti jedinstveni prirodan broj n koji ga identifikuje, i obratno, svakom broju n dodeliti program P_n (kao što je opisano u poglavlju 3.4).

Imajući ovo u vidu, dolazimo do precizne definicije halting problema za URM mašine.

Teorema 3.2 (Halting problem)**.** *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 H koji izračunava funkciju h, tj. program koji za zadatu vrednost x 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 postoji i program Q koji za zadatu vrednost 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 $h(x,x) = 1$). Ako postoji takav program Q, onda se i on nalazi u nizu svih programa tj. postoji redni broj k koji ga jedinstveno identifikuje. No, definicija njegovog ponašanja je 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. 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 h je jedna od najznačajnih funkcija 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 ponavaljanja) i koliko registara se

koristi. U URM programu navedenom i 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.

Pitanja i zadaci za vežbu

Pitanje 3.1. *Da li postoji algoritam koji opisuje neku brojevnu funkciju i koji može da se isprogramira u programskom jeziku C i izvrši na savremenom računaru, a ne može na Tjuringovoj mašini?*

Zadatak 3.2. *Napisati* URM *program koji izračunava funkciju*

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

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

$$f(x,y) = xy$$

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

$$f(x) = 2^x$$

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

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

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* $f(x) = \left[\frac{2x}{3}\right]$.

Zadatak 3.8. *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 3.9. *Napisati* URM *program koji izračunava funkciju* $f(x) = x!$.

Zadatak 3.10. *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 3.11. *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 3.12. *Koliko ima racionalnih brojeva? Koliko ima kompleksnih brojeva? Koliko ima različitih programa za Tjuringovu mašinu? Koliko ima različitih programa u programskom jeziku C?*

Pitanje 3.13. *Da li je moguće napraviti opšti postupak kojim se ispituje da li data izračunljiva funkcija (ona za koju postoji npr.* URM *program)* f *zadovoljava da je* $f(0) = 0$*?*

Pitanje 3.14.

Da li se svakom URM *programu može pridružiti jedinstven realan broj? Da li se svakom* URM *programu može pridružiti jedinstven prirodan broj? Da li se svakom realnom broju može pridružiti jedinstven* URM *program? Da li se svakom prirodnom broju može pridružiti jedinstven* URM *program?*

Zadatak 3.15. *Kako se naziva problem ispitivanja zaustavljanja programa? Da li je on odlučiv?*

Glava 4

Programski jezici

Razvoj programskih jezika, u bliskoj je vezi sa razvojem računara tj. sa razdvojem hardvera. Programiranje kakvo danas poznajemo je nastalo sa pojavom Fon Nojmanovog tipa računara 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 pedesetih godina dvadesetog veka nastali su jezici višeg nivoa koji su drastično olakšali programiranje.

4.1 Mašinski zavisni programski jezici

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 se želi programiranje novog računara, programer je primoran da iznova izučava detalje njegove arhitekture (na primer, skup instrukcija procesora, broj registara, organizaciju memorije). Programi napisani za jedan računar mogu da se izvršavaju isključivo na tom računaru i prenosivost programa nije moguća. Primitivne instrukcije koje programerima stoje na raspolaganju su veoma elementarne (na primer, samo sabiranje dva broja, instrukcija skoka i slično) i veoma je teško kompleksne i apstraktne matematičke algoritme izraziti korišćenjem isključivo tog uskog skupa elementarnih instrukcija.

4.1.1 Asemblerski jezici

Asemblerski (ili simbolički) jezici su jezici koji su veoma bliski mašinskom jeziku računara, pri čemu se, umesto korišćenja binarnog sadržaja za zadavanje 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*.

Ilustrujmo programiranje na mašinski zavisnim jezicima na minijaturnom modelu računara Fon Nojmanove arhitekture. Neka se procesor sadrži dva osmobitna registra `ax` i `bx` i neka podržava instrukcije:

`mov x y`	premešta se broj ili sadržaj lokacije `y` na lokaciju `x`, lokacija može biti ili registar ili memorijska adresa,
`add r1 r2`	sabira sadržaj registara `r1` i `r2` i zbir smešta u registar `r1`,
`mul r1 r2`	množi sadržaj registara `r1` i `r2` i zbir smešta u registar `r1`,
`cmp x y`	poredi vrednosti `x` i `y` i pamti rezultat,
`jle label`	ukoliko je vrednost prethodnog poređenja vrednosti (instrukcijom `cmp`) manja ili jednaka, skače na datu labelu.

Razmotrimo niz instrukcija:

```
loop:
  mov ax, [10]
  mov bx, [11]
  mul ax, bx
  mov bx, 1
  add ax, bx
  mov [10], ax
  cmp ax, 7F
  jle  loop
```

Nakon izvršenja prve instrukcije sadržaj memorijske adrese `10` se upisuje u registar `ax`. Nakon izvršenja druge instrukcije sadržaj memorijske adrese `11` se upisuje u registar `bx`. Trećom instrukcijom množi se sadržaj registara `ax` i `bx` i rezultat se smešta u `ax`. Nakon toga, u registar `bx` upisuje se vrednost `1` i sabira se sa tekućom vrednošću registra `ax`, smeštajući rezultat u registar `ax`. Zatim se tekuća vrednost registra `ax` (a to je proizvod vrednosti sa adresa `10` i `11` uvećan za 1) prenosi nazad na adresu `10`. Nakon toga, vrši se poređenje tekuće vrednosti registra `ax` (koja se nije promenila prenosom u memoriju) i vrednosti `7F`, i ukoliko je vrednost `ax` manja ili jednaka ceo postupak se ponavlja.

Dakle, ukoliko sadržaj memorijske adrese `10` označimo sa `x`, a adrese `11` sa `y`, prethodne instrukcije izvršavaju sledeći program:

```
radi
  x := x*y + 1
dok je x <= 127
```

4.1.2 Mašinski jezik

Programi na prethodno uveden asembleskom jeziku mogu da se binarno kodiraju i pravila tog prevođenja određuju jedan mašinski jezik. Na primer, binarne kodove za svaku od pet različitih instrukcija moguće je uvesti na sledeći način:

```
mov  001
add  010
mul  011
cmp  100
jle  101
```

Takođe, pošto neke instrukcije primaju podatke različite vrste (neposredno navedeni brojevi, registri, memorijske adrese), potrebno je uvesti posebne kodove za svaki od različitih vidova adresiranja. Na primer:

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

Neka registar `ax` ima oznaku `00`, a registar `bx` ima oznaku `01`.

Pretpostavimo da su sve adrese osmobitne.

Imajući sve ovo u vidu, instrukcija `mov [10], ax` se 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 je moguće binarno kodirati kao:

```
001 01 10 00 00010000
001 01 10 01 00010001
011 00 01
001 01 00 01 00000001
010 00 01
001 10 01 00010000 00
100 01 00 00 01111111
101 11110000
```

Pretpostavlja se da se ovaj kod smešta u memoriju na adresu `F0`, pošto poslednja instrukcija uslovnog skoka prenosi kontrolu upravo na tu adresu.

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

Specifični hardver koji čini kontrolnu jedinicu procesora dekodira jednu po jednu instrukciju i izvršava akcije njom kodirane. Naravno, prilikom dizajna realnih procesora, broj instrukcija i načini adresiranja je mnogo bogatiji i prilikom samog dizajna mašinskog jezika potrebno je uzeti u obzir mnoge aspekte na koje se u ovom jednostavnom prikazu nije obraćala pažnja.

4.2 Programski jezici višeg nivoa

Jedna od ključnih momenata u razvoju programiranja i računarstva bio je razvoj programskih jezika višeg nivoa. Viši programski jezici 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 kojima 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 povećanje nivoa apstrakcije prilikom procesa programiranja što u mnogome olakšava ovaj proces.

Prvim višim programskim jezikom najčešće se smatra jezik *Fortran*, nastao u periodu 1953-1957 godine. Njegov glavni autor je Džon Bakus, koji je implementirao i prvi interpretator i prvi kompilator za ovaj jezik. Najkorišćeniji viši programski jezici danas su jezici *C*, *C++*, *Java*, *PHP*, *JavaScript*, *Python*, *C#*, ...

Najkorišćeniji programski jezici danas spadaju u grupu (tzv. programsku paradigmu) *imperativnih* programskih jezika. S obzirom da u ovu grupu spada i programski jezik C, u nastavku će biti najviše reči upravo o ovakvim jezicima. U ovim jezicima program 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*, itd. Detaljni pregled viših programskih jezika, istorije njihovog razvoja i različitih načina njihove klasifikacije biće dat kasnije.

4.2.1 Leksika, sintaksa, semantika

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 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 osnovnim gradivnim elementima jezika, a sintaksa načinima za kombinovanje tih osnovnih elemenata u ispravne jezičke konstrukcije. Pitanjem značenja programa bavi *semantika programskih jezika*. Leksika, sintaksa i semantika se izučavaju ne samo za programske jezike, veće i za druge veštačke jezike, ali i za prirodne jezike.

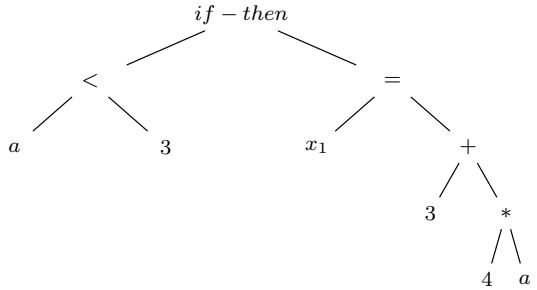
Leksika. Osnovni leksički elementi prirodnih jezika su reči, pri čemu se razlikuje nekoliko različitih vrsta reči (imenice, glagoli, pridevi, ...) i reči imaju različite oblike (padeži, vremena, ...). Zadatak leksičke analize prirodnog jezika je da identifikuje reči u rečenici i svrsta ih u odgovarajuće kategorije. Slično važi i za programske jezike. Programi se u računar zadaju kodirani 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 C koda:

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

Ovaj fragment koda predstavlja `if-then` iskaz kome je uslov dat izrazom poređenja vrednosti promenljive `a` i konstante `3`, a telo predstavlja iskaz dodele promenljivoj `x` vrednosti izraza koji predstavlja zbir konstante `3` i proizvoda konstante `4` i vrednosti promenljive `a`. U ovom kodu, 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

Sintaksa. Sintaksa prirodnih jezika definiše na 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 izmedju 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. Sintaksna struktura rečenica ili programa se može predstaviti u obliku stabla. Prikazani fragment koda je u jeziku C sintaksno ispravan i njegova sintaksna struktura se može predstaviti sa:



Semantika. Semantika pridružuje značenje sintaksno ispravnim niskama jezika. Za prirodne jezike, ovo znači povezivanje rečenica sa nekim specifičnim objektima, mislima i osećanjima. Za programske jezike, semantika za dati program opisuje koje je izračunavanje određeno tim programom.

Tako se na primer naredbi C jezika `if (a < 3) x1 = 3+4*a;` 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 i sintaksno ispravne rečenice kojima nije moguće dodeliti ispravno značenje, tj. nisu semantički korektne. 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 značenje, već dovodi do greške prilikom izvršavanja (iako prevođenje na mašinski jezik prolazi bez problema).

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

Dok većina savremenih jezika ima precizno i formalno definisanu sintaksu, formalna definicija semantike postoji samo za neke programske jezike. 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 sračunavanje 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 za izračunavanje izraza `f() + g()` prvo poziva funkcija `f` ili funkcija `g`).

4.2.2 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.

Promenljive i tipovi podataka. U Fon Nojmanovskom modelu, podaci se smeštaju u memoriju računara (najčešće kodirani nizom bitova). Međutim,

programski jezici uobičajeo, kroz koncept *promenljivih*, nude programerima apstraktniji pogled na podatke. 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.

Naredni fragment C koda 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 koda, prevodilac dodeljuje određeni broj bitova u memoriji promenljivama `x`, `y` i `z` (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, brinući pri tom o načinu reprezentacije koji se koristi za zapis. Tako se, u zavisnosti od tipa operanada, ista operacija `+` ponekad prevodi u mašinsku instrukciju kojom se sabiraju brojevi zapisani u potpunom komplementu, a ponekad u instrukciju kojom se sabiraju brojevi zapisani u pokretnom zarezu.

Slično, C kôd `int x = 3.1;` prouzrokuje da se promenljiva `x` deklariše kao celobrojna, da se za nju odvoji određeni broj bitova u memoriji i da se ta memorija inicijalizuje binarnim zapisom broja `3`. Naime, realnu vrednost `3.1` nije moguće zapisati u celobrojnu promenljivu, tako da se vrši konverzija tipa i zaokruživanje vrednosti.

Najčešći tipovi podataka direktno podržani programskim jezicima su celi brojevi (0, 1, 2, -1, -2, ...), razlomljeni brojevi (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 ili slogovi koji 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, oduzmanje, 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).

Neki programski jezici (između ostalog i C) zahtevaju da korisnik definiše tip svake promenljive koja se koristi u programu i da se taj tip ne menja tokom izvršavanja programa (*statički tipizirani jezici*). S druge strane, postoje jezici gde ista promenljiva može da sadrži podatke različitog tipa tokom raznih faza izvršavanja programa (*dinamički tipizirani jezici*). U nekim slučajevima dopušteno je vršiti operacije nad promenljivima različitog tipa, pri čemu se, naravno, 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 potrebna 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 sretane kontrolne strukture su granajuće naredbe (if-then-else), petlje (for, while, do-while, repeat-until) i naredbe skoka (goto). Za naredbe skoka (goto) je pokazano da ih nije potrebno koristiti, 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 petlju (na primer, do-while).[1] Ovaj važan teorijski rezultat ima svoju punu praktičnu primenu i u današnjem programiranju naredba skoka se skoro uopšte ne koristi jer dovodi do nerazumljivih (tzv. špageti) programa.

Potprogrami. Najčešća vrsta potprograma su funkcije (procedure, sabrutine, metode). Njima se u programu izoluje određena vrsta izračunavanja koju je kasnije moguće pozivati tj. koristiti na više različitih mesta u različitim kontekstima. Potprogrami su obično parametrizovani i obično imaju mogućnost vraćanja vrednosti izračunavanja pozivaocu. 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);
```

[1] Ovo tvrđenje je poznato kao *teorema o strukturnom programiranju*, autora Korado Bema i Đuzepea Jakopinija iz 1966.

Modularnost. Modularnost podrazumeva razbijanje većeg programa na nezavisne celine. Celine se obično smeštaju u različite datoteke i sadrže biblioteke srodnih definicija podataka i funkcija. Ovim se postiže lakše održavanje kompleksnih sistema kao i mogućnost višestruke upotrebe biblioteka u okviru različitih programa.

4.2.3 Jezički procesori

Jezički procesori (ili programski prevodioci) su programi čija je uloga da analiziraju leksičku, sintaksnu 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 su programski prevodioci kod kojih su faza prevođenja i faza izvršavanja programa potpuno razdvojene. Nakon analize *izvornog koda* programa višeg programskog jezika, kompilatori generišu *izvršni (mašinski) kôd* koji se zatim snima u izvršne binarne datoteke. 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[2]. Jedan od problema rada sa kompilatorima je da se gubi svaka veza između izvršnog i izvornog koda programa. Svaka (i najmanja) izmena u izvornom kodu programa zahteva ponovno prevođenje celokupnog programa. S druge strane, kompilirani programi su obično veoma efikasni jer se pre snimanja u izvršne datoteke mašinski kôd dodatno optmizuje.

Interpretatori. Interpretatori su programski prevodioci kod koji su faza prevođenja i faza izvršavanja programa isprepletane. Interpretatori analiziraju deo po deo (najčešće naredbu po naredbu) izvornog koda 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 koda. Iz ovog razloga, programi koji se interpretiraju se obično izvršavaju znatno sporije nego u slučaju kompilacije. S druge strane, razvojni ciklus programa je često kraći ukoliko se koriste interpretatori. Naime, prilikom malih izmena programa nije potrebna iznova vršiti analizu celokupnog koda.

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

Često je slučaj da se za jedan izvorni (programski) jezik gradi i interpretator i kompilator. 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 najefikasnijim 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 asemblerski 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, zatim kod .Net jezika (C#, VB), itd.

Pitanja i zadaci za vežbu

Zadatak 4.1. *Na opisanom asemblerskom jeziku opisati izračunavanje vrednosti izraza* `x := x*y + y + 3`. *Generisati i mašinski kôd za napisani program.*

Zadatak 4.2. *Na opisanom asemblerskom jeziku opisati izračunavanje:*

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

Zadatak 4.3. *Na opisanom asemblerskom jeziku opisati izračunavanje:*

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

Zadatak 4.4. *Na opisanom asemblerskom jeziku opisati izračunavanje kojim se izračunava* $[\sqrt{x}]$, *pri čemu se* x *nalazi na adresi 100.*

Pitanje 4.5. *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? 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 4.6. *Za koji programski jezik su izgrađeni prvi interpretator i kompilator i ko je bio njihov autor?*

Pitanje 4.7.

Navesti nekoliko programskih jezika sa kraja 1950-tih i početka 1960-tih.
Navesti nekoliko programskih jezika nastalih 1970-tih.
Navesti nekoliko najpopularnijih savremenih programskih jezika.
Navesti nekoliko jezika koji su pre svega namenjeni za učenje programiranja.
Navesti nekoliko jezika koji se koriste za Veb programiranje.

Pitanje 4.8. *Kojim aspektima jezika se bavi sintaksa, kojim semantika, a kojim pragmatika?*

Pitanje 4.9. *Šta karakteriše jedan tip u nekom programskom jeziku?*

Pitanje 4.10. *Šta tvrdi teorema o strukturnom programiranju?*

Deo II

Jezik C

Glava 5

O programskom jeziku C

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

Jezik C spada u grupu imperativnih (proceduralnih) programskih jezika. S obzirom da je bio namenjen za sistemsko programiranje, programerima nudi prilično direktan pristup memoriji i konstrukcije jezika su tako osmišljene da se jednostavno prevode na mašinski jezik. Time su u C-u pre svega kreirani programi koji bi se ranije uglavnom pisali na asemblerskom jeziku. Pored ovoga, jezik C od samog početka insistira na prenosivosti koda (tzv. portabilnosti), tj. kroz pridržavanje standarda jezika, moguće je pisati programe koji se mogu koristiti na više različitih platformi. Jezik je kreiran u skladu sa minimalističkim duhom — direktno je podržan mali broj konstrukata, dok se šira funkcionalnost programerima 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 standardnih funkcija).

5.1 Standardizacija jezika

K&R C. 1978. godine, Brajan Kerningen (eng. Brian Kerninghan) i Denis Riči (eng. Dennis Ritchie) objavljuju prvo izdanje knjige *Programski jezik C (The C Programming Language)*. Ova knjiga, među programerima obično zvana

K&R je godinama služila kao neformalna specifikacija jezika. Čak i posle pojave novih standarda, K&R je dugo vremena služio kao ,,najmanji zajednički imenilac" koji je korišćen kada je bilo potrebno postići veliki stepen prenosivosti, s obzirom da su konstrukte opisane u prvoj verziji K&R knjige uglavnom podržavali svi C prevodioci.

ANSI C i ISO C. Tokom 1980-tih godina, C se raširio na veliki broj izrazito heterogenih platformi i time se javila potreba za zvaničnom standarizacijom jezika. 1989. godine, američki nacionalni zavod za standardizaciju (ANSI) objavljuje standard pod zvaničnim nazivom veliki stepen prenosivosti, s obzirom da su konstrukte opisane u prvoj verziji K&R knjige uglavnom podržavali svi C prevodioci.

ANSI C i ISO C. Tokom 1980-tih godina, C se raširio na veliki broj izrazito heterogenih platformi i time se javila potreba za zvaničnom standarizacijom jezika. 1989. godine, američki nacionalni institut za standardizaciju (ANSI) objavljuje standard pod zvaničnim nazivom *ANSI X3.159-1989 "Programming Language C"*. Ova verzija jezika C se obično jednostavnije naziva *ANSI C* ili *C89*. 1990. godine međunarodna organizacija za standardizaciju (ISO) usvaja ovaj dokument (uz sitnije izmene, uglavnom formatiranje) pod oznakom *ISO/IEC 9899:1990* čime briga o standrdizaciji jezika C prelazi od američkog pod međunarodno okrilje. Ova verzija se ponekad naziva *C90*, tako da se *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. 1999. godine usvojen je novi standard jezika C *ISO/IEC 9899:1999*, poznat pod kraćim imenom *C99*. Ovaj standard pravi sitnije izmene u odnosu na prethodni i uglavnom proširuje jezik novim konstruktima (u velikoj meri inspirisanim modernim programskim jezicima kakav je C++).

C1X. 2007. godine počeo je rad na novom standardu jezika C koji se očekuje u drugoj deceniji XXI veka (s obzirom da se ne zna koje će se godine pojaviti, trenutno se ovaj standard neformalno označava sa *C1X*).

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

Glava 6

Prvi programi

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

6.1 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 uslediti njihov detaljan opis.

Program 6.1.

```
#include <stdio.h>

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

Program se sastoji 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 kreće 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 (eng. 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 `;`.

Programi obično imaju više funkcija i funkcija `main` poziva te druge funkcije za obavljanje raznovrsnih podzadataka. Funkcije koje se pozivaju mogu

da budu bilo korisnički definisane (tj. napisane od strane istog programera koji piše program), bilo 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 navedenom programu je poziv standardne bibliotečke funkcije `printf` koja ispisuje tekst na takozvani *standardni izlaz* (obično ekran). U ovom slučaju, tekst koji se ispisuje je `Zdravo!`. Tekst koji treba ispisati se zadaje između para znakova `"`. 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 njen tip povratne vrednosti i tip njenih argumenata. Ovi podaci o funkciji `printf`, njen svojevrsni opis, takozvana *deklaracija*, nalaze se u takozvanoj *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 konsultuje sadržaj datoteke `stdio.h` je takozvana *preprocesorska direktiva*. To nije naredba C jezika, već instrukcija C preprocesoru koji predstavlja nultu fazu kompilacije i koji pre kompilacije program priprema tako što umesto prve linije upiše 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. Iz funkcije `main`, obično se 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 doprinose izgradnji izvršnog programa.

Primetimo da je svaka naredba u okviru prikazanog programa 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"* (eng. indentation) kôda u skladu sa njegovom sintaksnom strukturom nezaobilazna praksa dobrih programera i na njoj od početka treba jako insistirati. Naglasimo i da postoje različiti stilovi nazubljivanja (npr. da li otvorena vitičasta zagrada počinje u istom ili sledećem redu), međutim, obično se smatra da nije bitno koji se stil koristi dokle god se kôd unosi na uniforman način.

6.2 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 6.2.

```
#include <stdio.h>

int main() {
   int a;
   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 takozvanog *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 okviru funkcije `scanf`, tj. da će pročitani broj biti smešten na adresu promenljive `a`. Korišćenjem funkcije `printf`, pored teksta koji se ne menja, može se ispisati i vrednost neke promenljive ili izraza. U formatu ispisa, na mestu u okviru teksta gde treba ispisati vrednost izraza zapisuje se format tog ispisa — u ovom slučaju `%i` jer će biti ispisana celobrojna vrednost. Nakon formatirajućeg teksta, navodi se izraz koji treba ispisati — u ovom slučaju vrednost `a*a`, tj. kvadrat broja koji je korisnik uneo.

6.3 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 zahteva rad sa razlomljenim brojevima, tj. brojevima 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 `%f` umesto niske `%d`. Dodatno, za računanje kvadratnog korena koristi se funkcija `sqrt` deklarisana u zaglavlju `math.h`.

Program 6.3.

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

int main() {
  double x1, y1, x2, y2;
  printf("Unesi koordinate prve tacke: ");
  scanf("%f%f", &x1, &y1);
  printf("Unesi koordinate druge tacke: ");
  scanf("%f%f", &x2, &y2);
  printf("Rastojanje je: %f\n",
     sqrt((x2 - x1)*(x2 - x1) + (y2 - y1)*(y2 - y1)));
  return 0;
}
```

Nakon unosa podataka, traženo rastojanje se jednostavno 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 svodi na množenje.

Napomenimo da je prilikom prevođenja programa koji koriste matematičke funkcije, prilikom korišćenja GCC prevodioca potrebno navesti `-lm` parametar (na primer, `gcc -lm -o rastojanje rastojanje.c`).

6.4 Program koji ispituje da li je uneti broj paran

Naredni program ispituje da li je uneti broj paran.

Program 6.4.

```
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 `==`. Naglasimo da je operator poređenja `==` različit od operatora dodele `=`. Naredba `if` je takozvana *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 tačka-zapeta `;`. U slučaju da je uslov ispunjen, izvršava se prva naredba napisana nakon uslova (takozvana `then` grana), dok se u slučaju da uslov nije ispunjen izvršava naredba navedena nakon reči `else` (takozvana `else` grana). I `then` i `else` grana mogu da sadrže samo jednu naredbu a ukoliko ih ima više, one čine *blok* čiji se početak i kraj moraju označiti zagradama `{` i `}`. Grana `else` ne mora da postoji.

Pitanja i zadaci za vežbu

Pitanje 6.1. *Funkciju kojeg imena mora da sadrži svaki C program?*

Pitanje 6.2. *Šta su to preprocesorske direktive?*

Pitanje 6.3. *Šta su to datoteke zaglavlja?*

Pitanje 6.4. *Šta je to standardna biblioteka?*

Pitanje 6.5. *Kojom naredbom se vraća rezultat funkcije u okruženje iz kojeg je ona pozvana?*

Zadatak 6.6. *Napisati program koji za uneti poluprečnik r izračunava obim i površinu kruga (za broj π koristiti konstantu* `M_PI` *iz zaglavlja* `math.h`*).*

Zadatak 6.7. *Napisati program koji za unete koordinate temena trougla izračunava njegov obim i površinu.*

Zadatak 6.8. *Napisati program koji za unete dve dužine stranica trougla i ugla između njih (u stepenima) izračunava dužinu treće stranice i veličine dva preostala ugla. Napomena: koristiti sinusnu i/ili kosinusnu teoremu i funkcije* `sin()` *i* `cos()` *iz zaglavlja* `math.h`*.*

Zadatak 6.9. *Napisati program koji za unetu brzinu u $\frac{km}{h}$ ispisuje odgovarajuću brzinu u $\frac{m}{s}$.*

Zadatak 6.10. *Napisati program koji za unetu početnu brzinu v_0, ubrzanje a i vreme t izračunava trenutnu brzinu tela v i pređeni put s koje se kreće ravnomerno ubrzano.*

Zadatak 6.11. *Napisati program koji izračunava sumu cifara unetog četvorocifrenog broja.*

Zadatak 6.12. *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*.*

Napomena: celobrojni količnik se može izračunati operatorom `/`*, a ostatak pri deljenju operatorom* `%`*.*

Zadatak 6.13. *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 6.14. *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 6.15. *Napisati program koji ispisuje najveći od 3 uneta broja.*

Zadatak 6.16. *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 6.17. *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 6.18. *Napisati program koji za unete koeficijente a, b i c izračunava i ispisuje realna rešenja kvadratne jednačine $ax^2 + bx + c = 0$.*

Glava 7

Promenljive, tipovi, deklaracije, operatori, izrazi

U ovoj glavi biće detaljnije opisani naredni pojmovi:

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

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

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

7.1 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.) su određena *identifikatorima*. U jednom od prethodnih programa koristi se promenljiva čije je ime `a`. Generalno, identifikator može da sadrži slova i cifre, kao i simbol `_` (koji je pogodan za duga imena), ali identifikator ne može počinjati cifrom. Ključne reči jezika C (na primer, `if`, `for`, `while`) se ne mogu koristiti kao identifikatori.

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

U okviru identifikatora, 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 oslikava 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 je tzv. kamilja notacija (CamelCase). Postoje različite konvencije za imenovanje promenljivih. U nekim, kao što je *mađarska notacija*, početna slova imena promenljivih predstavljaju kratku oznaka tipa te promenljive (na primer, `iBrojStudenata`).[1].

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

7.2 Deklaracije

Sve promenljive moraju biti deklarisane pre korišćenja. Deklaracija specifikuje tip i sadrži listu od jedne ili više promenljivih tog tipa.

```
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. Prilikom deklaracije može se izvršiti i početna inicijalizacija.

```
int vrednost = 5; /* deklaracija sa inicijalizacijom */
```

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

```
const int KILO = 1024;
```

Ukoliko je promenljiva deklarisana u okviru neke funkcije, 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.

[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).

7.3 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. 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).

7.3.1 Tip `int`

U jeziku C, cele brojeve opisuje tip `int` (od engleskog *integer*, *ceo broj*). Podrazumeva se da su brojevi označeni i reprezentuju se najčešće koristeći potpuni komplement. Nad podacima ovog tipa mogu se koristiti aritmetičke operacije (na primer, `+`, `-`, `*`, `/`, `%`), relacije (na primer, `<`, `>=`) i druge. Nije propisano koliko tačno bitova koriste podaci tipa `int`, propisano je samo 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 trideset i dva bita, tj. četiri bajta.

Tipu `int` mogu biti pridruženi kvalifikatori `short`, `long` i (samo u 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, propisano je samo da `short` zauzima bar dva bajta, da `int` zauzima barem onoliko bajtova koliko i `short int`, da `long int` zauzima barem onoliko bajtova koliko `int`, 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 uiz 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 uobičajenih vrednosti dat je u tabeli 7.1.

	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) samo u 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 7.1: Uobičajeni opseg tipova (važi i na x86 + gcc platformi).

7.3.2 Tip `char`

Veoma male cele brojeve opisuje tip `char` (od engleskog *character — karakter, simbol, znak*).[3] I nad podacima ovog tipa mogu se koristiti aritmetičke operacije i relacije. Podatak tipa `char` zauzima tačno jedan bajt. Nije propisano da li se podatak tipa `char` smatra označenim ili neoznačenim brojem (na nekom sistemu se smatra označenim, a na nekom drugom se može smatrati neoznačenim). Na tip `char` mogu se primeniti kvalifikatori `unsigned` i `signed`. Kvalifikator `unsigned` označava 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` označava 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.

7.3.3 Tipovi `float`, `double` i `long double`

Realne brojeve (preciznije brojeve u pokretnom zarezu) opisuju tipovi `float`, `double` i (samo u C99) `long double`. Tip `float` opisuje realne brojeve osnovne tačnosti, tip `double` realne broj dvostruke tačnosti, a tip `long double` realne brojeve proširene tačnosti. Nije propisano koliko ovi tipovi zauzimaju bitova, propisano je samo 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>`.

[3] Brojevna vrednost promenljive `c` tipa `char` se može 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 poglavlju 19.1.

I nad podacima ovih tipova mogu se koristiti uobičajene aritmetičke operacije (ali, na primer, nije moguće računati ostatak pri deljenju `%`) i relacije.

Realni brojevi u savremenim računarima se najčešće zapisuju u skladu sa IEEE754 standardom. Ovaj standard uključuje i mogućnost zapisa specijalnih vrednosti (npr. $+\infty$, $-\infty$, NaN) i one se ravnopravno koriste prilikom izvođenja aritmetičkih operacija. Npr. 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).

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 koji na kojima je izvođenje operacija u dvostrukoj tačnosti veoma skupo (u današnje vreme, većina računara podržava efikasnu manipulaciju sa brojevima zapisanim u dvostrukoj tačnosti).

7.3.4 Logički tip podataka

Iako mnogi programski jezici uvode poseban tip za predstavljanje (logičkih) istinitosnih vrednosti, programski jezik C sve do standarda C99 nije uvodio 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 vrednot *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*.

7.3.5 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 savremenim računarima vrednost ovog izraza je najčešće 4.

7.4 Brojevne konstante i konstantni izrazi

Svaki izraz koji se pojavljuje u programu je ili promenljiva ili konstanta ili složen izraz. Konstante su fiksne, zadate vrednosti kao, na primer, `0`, `2` ili `2007`. U ovom delu teksta biće reči o tome kako se zapisuju brojevne konstante i kako su određeni njihovi tipovi.

Celobrojne konstante. Celobrojna konstanta kao što je `123` je tipa `int`. Velike celobrojne konstante koje ne mogu biti smeštene u tip `int` a mogu u tip `long` su tipa `long`. Ukoliko se želi da se neka celobrojna konstanta tretira da je tipa `long`, onda se na njenom kraju piše slovo `l` ili `L`, na primer, `123456789L`. Ukoliko se želi da se neka celobrojna konstanta tretira kao `unsigned`, onda se na njenom kraju piše slovo `u` ili `U`. Na primer, tip konstante `12345ul` je `unsigned long`.

Celobrojna konstanta može biti zapisana i u oktalnom i u heksadecimalnom sistemu. Zapis konstanta u oktalnom sistemu počinje cifrom `0`, a zapis konstanta u heksadecimalnom 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` (heksadecimalni zapis). I oktalne i heksadecimalne konstante mogu da budu označene slovima `U` i `L` na kraju svog zapisa.

Realne konstante. Konstante realnih brojeva sadrže decimalnu tačku (na primer, `123.4`) ili eksponent (`1e-2`) ili i jedno i drugo. Njihov tip je `double`, osim ako na kraju zapisa imaju slovo `f` ili `F` kada je njihov tip `float`. Slova `L` i `l` na kraju zapisa označavaju da je tip vrednosti `long double`.

Karakteri. U programskom jeziku C, tip `char` se istovremeno koristi za kodiranje malih celih brojeva i karaktera. Na savremenim računarima i C implementacijama, najčešće korišćeno kodiranje karaktera je ASCII (mada C standardi kodiranje ostavljaju nespecifikovanim). Ovo znači da su karakteri u potpunosti identifikovani svojim ASCII kodovima i obratno. Međutim, direktno specifikovanje karaktera korišćenjem numeričkih kodova 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 skupu karaktera računara (danas je to najčešće 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 kodu 65, `'a'` je karakterska konstanta čija je vrednost u ASCII kodu 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 */
```

Specijalni karakteri se mogu navesti korišćenjem specijalnih sekvenci karaktera koje počinju karakterom \ (eng. 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`	octal number
`\xhh`	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 istakakla karakterska priroda ovog izraza.

Pošto se karakterske konstante identifikuju sa njihovim numeričkim vrednostima one predstavljaju podatke tipa `char` i mogu ravnopravno da učestvuju u aritmetičkim izrazima (npr. `'0' <= c && c <= '9'` ili `c - '0'`).

7.5 Operatori i izrazi

Izrazi u programskom jeziku C se 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. Konstantni izraz je izraz koji sadrži samo konstante. Takvi izrazi mogu se izračunati u fazi prevođenja i mogu se koristiti na svakom mestu na kojem se može pojaviti konstanta.

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. Podržani su i operacije koje operišu nad pojedinačnim bitovima, ali s obzirom da se ove operacije koriste u malo naprednijim programima, o njima će biti reči u narednim glavama. Svaki operator ima definisan prioritet i asocijativnost koji određuju kako se operatori primenjuju u okviru složenih izraza. U principu, unarni operatori imaju prioritet u odnosu na binarne, binarni (osim dodela) u odnosu na ternarni, dok dodele imaju najniži prioritet. Aritmetički operatori imaju viši prioritet u odnosu na relacijske, koji imaju viši prioritet u odnosu na logičke.

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

7.5.1 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 videćemo 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).

Suštinski, dodeljivanje vrednosti predstavlja izraz čija je vrednost jednaka vrednosti koja se dodeljuje, dok je promena vrednosti promenljivoj samo *bočni efekat* do koga dolazi prilikom izračunavanja vrednosti izraza. Na primer, vrednost izraza `broj_studenata = 80;` je `80`, pri čemu se prilikom izračunavanja te vrednosti menja vrednost promenljive `broj_studenata`. To se može iskoristiti i za višestruko dodeljivanje. Na primer, nakon sledeće naredbe, sve tri promenljive `x`, `y` i `z` imaće vrednost `0`.

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

7.5.2 Aritmetički operatori

Aritmetički operatori. Nad operandima celobrojnih i realnih tipova mogu se koristiti sledeći aritmetički operatori:

`+` 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 `%` je moguće primeniti isključivo na operandima celobrojnog tipa.

Operator deljenja označava različite operacije u zavisnosti od tipa svojih operanada. Kada se operator deljenja primenjuje na dve celobrojne vrednosti primenjuje se celobrojno deljenje (tj. rezultat je ceo deo količnika). Na primer, izraz `9/5` ima vrednost `1`. U ostalim slučajevima primenjuje se realno deljenje. Na primer, izraz `9.0/5.0` ima vrednost `1.8` (jer se koristi deljenje realnih brojeva). U slučaju da je jedan od operanada ceo, a drugi realan broj, vrši se implicitna konverzija (promocija) celobrojnog operanda u realni i primenjuje se realno deljenje.

Unarni operatori `+` i `-` imaju viši prioritet od binarnih operatora `+` i `-`.

Inkrementiranje i dekrementiranje.

`++` (prefiksno i postfiksno) inkrementiranje;
`--` (prefiksno i postfiksno) dekrementiranje.

Operator inkrementiranja (uvećavanja za 1) zapisuje se `++`, a operator dekrementiranja (umanjivanja za 1) se zapisuje `--`. Oba ova 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 se `++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 bočni efekat, uvećava vrednost promenljive `n`.

Na primer, ako promenljiva `n` ima vrednost `5`, onda

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

dodeljuje promenljivoj `x` vrednost `5`, a

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

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

Ukoliko ne postoji širi kontekst, tj. ako inkrementiranje čini čitavu naredbu, vrednost izraza se zanemaruje i onda nema razlike između naredbe `n++;` i `++n;`. Inkrementiranje i dekrementiranje se mogu primenjivati samo nad l-vrednostima. Tako, na primer, izraz `5++` nije ispravan.

7.5.3 Relacioni i logički operatori

Relacioni operatori. Nad celobrojnim i realnim tipovima mogu se koristiti sledeći binarni relacioni operatori:

`>` veće;

`>=` veće ili jednako;

`<` manje;

`<=` manje ili jednako;

`==` jednako;

`!=` različito.

Relacioni operatori poretka `<`, `<=`, `>` i `>=` imaju isti prioritet i to viši od operatora jednakosti `==` i različitosti `!=`. Rezultat relacionog operatora primenjenog nad dva broja je vrednost 0 (koja odgovara istinitosnoj vrednosti *netačno*) ili vrednost 1 (koja odgovara istinitosnoj čkoj vrednosti *tačno*). 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.

Binarni relacioni operatori imaju niži prioritet od binarnih aritmetičkih operatora.

Potrebno je voditi računa o tome da su operator `==` koji proverava da li su neke dve vrednosti jednake i operator dodele `=` različiti operatori i imaju potpuno drugačiju semantiku. Nihovo nehotično mešanje je čest uzrok grešaka u C programima.

Logički operatori. Logički operatori se primenjuju nad istinitosnim vrednostima, koje se, kao što je već rečeno, predstavljaju preko brojevnih vrednosti.

Ukoliko je broj jednak 0, onda je njegova logička vrednost 0 (netačno), a inače je njegova logička vrednost 1 (tačno).

Postoje sledeći logički operatori:

`!` logička negacija — *ne*;

`&&` logička konjunkcija — *i*;

`||` logička disjunkcija — *ili*.

Operator `&&` ima veći prioritet u odnosu na operator `||`. 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.

Na primer,

- vrednost izraza `5 && 4` jednaka je `1`
- vrednost izraza `10 || 0` jednaka je `1`
- vrednost izraza `0 && 5` jednaka je `0`
- vrednost izraza `!1` jednaka je `0`
- vrednost izraza `!9` jednaka je `0`
- vrednost izraza `!0` jednaka je `1`
- vrednost izraza `!(2>3)` jednaka je `1`
- 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* (eng. 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 izrazima u kojima 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`.

7.5.4 Bitovski operatori

`~` bitska negacija;
`&` bitska konjunkcija;
`|` bitska disjunkcija;
`^` bitska ekskluzivna disjunkcija;
`<<` pomeranje (šiftovanje) bitova ulevo;
`>>` pomeranje (šiftovanje) bitova udesno.

C podržava naredne operatore za rad nad pojedinačnim bitovima (moguće ih je primenjivati samo na celobrojne argumente):

`&` – **bitovsko *i*** — primenom ovog operatora vrši se konjunkcija pojedinačnih bitova dva navedena 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 da se prevođenje 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 `0x4A`, broj `x2` bi imao vrednost `0x57`, a rezultat bi bio `0x42`.

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

`^` – **bitovsko *ekskluzivno ili*** — primenom ovog operatora vrši se ekskluzivna disjunkcija pojedinačnih bitova dva navedena argumenta. Za brojeve iz tekućeg primera, rezultat izraza `x1 ^ x2` bio bi `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 uvek 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 bitovi prvog argumenta se zanemarju, a što se tiče početnih 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. U C-u, tip prvog argumenta određuje kako će se vršiti pomeranje. 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 `unsigned char` i vrednost `0x95`, tj. `10010101`, vrednost izraza `x >> 1` je `11001010`, tj. `0xCA`. Ukoliko je tip promenljive `x` `signed char`, tada je vrednost izraza `x >> 1` broj `01001010`, tj. `0x4A`.

Napomenimo da 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`).

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.

7.5.5 Složeni operatori dodele

`+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `<<=`, `>>=`

Dodela koja uključuje aritmetički operator `i = i + 2;` može se zapisati 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 (na primer, `-=`, `/=`, `%=`, `&&=`, `||=`, itd.)

Generalno, naredba

```
izraz1 op= (izraz1)
```

ima isto dejstvo kao i

`izraz1 = (izraz1) op (izraz2)`.
Međutim, treba biti obazriv jer postoje slučajevi kada su ova dva izraza različita[4].

Kraći zapis uz korišćenje složenih operatora dodele najčešće daje i nešto efikasniji kôd.

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

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

Program 7.1.
```
#include <stdio.h>

int main() {
    unsigned char c = 254;
    c += 1;
    printf("Vrednost promenljive c je jednaka %i\n", c);
    c += 1;
    printf("vrednost promenljive c je jednaka %i\n", c);
    return 0;
}
```

Izlaz programa:
```
Vrednost promenljive c je jednaka 255
Vrednost promenljive c je jednaka 0
```

7.5.6 Operator uslova

Ternarni operator uslova `?:` se koristi u sledećem opštem obliku:
```
izraz1 ? izraz2 : izraz3;
```

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, nakon
```
n = 0;
x = (2 > 3) ? n++ : 9;
```
promenljiva `x` imaće vrednost `9`, a promenljiva `n` će zadržati vrednost `0`.

Pitanja i zadaci za vežbu

Pitanje 7.1. *Da li ime promenljive može počinjati simbolom* `_`*? Da li se ovaj simbol može koristiti u okviru imena? Da li ime promenljive može počinjati*

[4]Na primer, prilikom izračunavanja izraza `a[f()] += 1` (`a[i]` označava `i`-ti element niza `a`), funkcija `f()` se poziva samo jedan put, dok se kod `a[f()] = a[f()] + 1` 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 kasnijim poglavljima.

cifrom? Da li se cifre mogu koristiti u okviru imena?

Pitanje 7.2. *Šta karakteriše jedan tip podataka?*

Pitanje 7.3. *U opštem slučaju, ukoliko celobrojna promenljiva nije inicijalizovana, koja je njena početna vrednost?*

Pitanje 7.4. *Koliko bajtova zauzima podatak tipa:*
(a) `char` *(b)* `int` *(c)* `short int` *(d)* `unsigned long`

Pitanje 7.5. *Koja je razlika između konstanti* `3.4` *i* `3.4f`*?*

Pitanje 7.6. *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`

Pitanje 7.7. *Da li C omogućava direktan zapis binarnih konstanti? Koji je najkoncizniji način da se celobrojna promenljiva inicijalizuje na binarni broj* `10111010110101001110011`*0?*

Pitanje 7.8. *Šta je uloga kvalifikatora* `const`*?*

Pitanje 7.9. *Deklarisati označenu karaktersku promenljivu pod imenom* `c` *i inicijalizovati je tako da sadrži kôd karaktera* `Y`*.*

Pitanje 7.10. *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`

Pitanje 7.11. *Postaviti zagrade u naredne izraze u skladu sa podrazumevanim prioritetom i asocijativnosti operatora (npr.* `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`

Pitanje 7.12. *Kolika je vrednost promenljivih* `a`*,* `b`*,* `x` *i* `y` *nakon naredbi:*

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

Pitanje 7.13. *Šta je efekat naredbe* `a &&= 0;`*? A naredbe* `a ||= -7;`*?*

Pitanje 7.14. *Napisati izraz kojim se proverava da li je godina* `g` *prestupna.*

Zadatak 7.15. *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 7.16. *Napisati program koji za uneta tri pozitivna realna broja ispituje da li postoji trougao čije su dužine stranica upravo ti brojevi — uslov provere napisati u okviru jednog izraza.*

Glava 8

Naredbe i kontrola toka

Osnovni elementi kojima se opisuju izračunavanja u C programima su *naredbe*. U prethodnom delu teksta jedine korišćene naredbe su bile naredba dodele, naredbe koje odgovaraju pozivima funkcija za učitavanje i ispis (`scanf` i `printf`) i opisane su samo jezičke konstrukcije koje omogućavaju pisanje sekvencijalnih programa (programa u kojima nema grananja i u kojima je tok izvršavanja uvek isti) i programa sa jednostavnim grananjem (naredba `if-else`). Naredbe za kontrolu toka omogućavaju različita 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 ona često dovodi do loše strukturiranih programa, nečitljih i teških za održavanje, a, s druge strane, 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 zarad 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.

8.1 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 kada se završi karakterom `;` postaje 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 (npr. `=`, `++`, `+=`, itd.) ili ukoliko se pozivaju funkcije. Naredba dodele, kao i naredbe koje odgovaraju pozivima funkcija sintaksno su obuhvaćene naredbom izraza.

```
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.

8.2 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 su sintaksno ekvivalentni pojedinačnim naredbama. Vitičaste zagrade koje okružuju naredbe jedne funkcije su jedan primer njihove upotrebe, ali one se, za grupisanje naredbi, koriste i drugim kontekstima. Svaki blok se sastoji od (moguće praznog) spiska deklaracija promeljivih (njihov doseg je taj blok) i, zatim, od spiska naredbi (elementarnih ili složenih, tj. novih blokova). Iza zatvorene vitičaste zagrade ne piše se znak `;`. Postoje različite konvencije za nazubljivanje vitičastih zagrada prilikom unosa izvornog koda.

8.3 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 realne vrednosti) 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 sintaksno ispravan, ali verovatno je semantički pogrešan:

```
a = 3;
if (a = 0)
   printf("a je nula\n");
else
   printf("a nije nula\n");
```

Naime, efekat ovog koda je da postavlja vrednost promenljive `a` na nulu (a ne da ispita da li je `a` jednako `0`), a zatim da ispisuje `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 uslovno izvršavaju mogu da sadrže nove naredbe uslova, tj. može biti više ugnježdenih `if` naredbi. Ukoliko vitičastim zagradama nije obezbeđeno drugačije, `else` se odnosi na prvi 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 formatiranje 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
```

8.4 Konstrukcija `else-if`

Za višestruke odluke se često 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 pozitivan\n");
else if (a < 0)
  printf("A je negativan\n");
else /* if (a == 5) */
  printf("A je nula\n");
```

8.5 Naredba `if-else` i operator uslova

Naredna naredba

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

računa i smešta u `x` veću od vrednosti `a` i `b`. Naredba ovakvog oblika se može zapisati kraće korišćenjem ternarnog operatora uslova `?:` (koji je opisan u poglavlju 7.5.6), na sledeći način:

```
x = (a > b) ? a : b;
```

8.6 Naredba `switch`

Naredba `switch` se koristi za višestruko odlučivanje. Naredbe koje treba izvršiti navode se kao *slučajevi* (eng. *case*) za različite moguće pojedinačne vrednosti zadatog izraza. Jednom slučaju pridružen je jedan ili više konstantnih celobrojnih izraza. Ukoliko zadati izraz ima vrednost jednog od tih konstantnih izraza, onda se izvšava naredba pridružena tom slučaju, a nakon toga se nastavlja sa izvršavanjem i naredbi koje odgovaraju sledećim uslovima *iako njihovi uslovi nisu ispunjeni.* Slučaj `default` označava da uslov za nijedan drugi slučaj nije ispunjen. Ovaj slučaj je opcioni i ukoliko nije naveden, a nijedan postojeći slučaj nije ispunjen, onda se ne izvršava nijedna naredba u okviru bloka `switch`.

Sintaksno — slučajevi mogu biti navedeni u proizvoljnom poretku (uključujući i slučaj `default`), ali različiti poretci mogu da daju različito ponašanje programa.

Naredba `switch` ima sledeći opšti oblik:

```
switch (izraz) {
    case konstantan_izraz1: naredba1
    case konstantan_izraz2: naredba2
    ...
    default:                naredba_n
}
```

U narednom primeru proverava se da li je uneti broj deljiv sa tri, korišćenjem naredbe `switch`.

Program 8.1.

```
#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`.

Kada se naiđe na naredbu `break`, napušta se naredba `switch`. Da u gornjem primeru 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).

Najčeće se naredbe pridužene svakom slučaju završavaju naredbom `break` (čak i nakon slučaja `default`, iako to ne menja ponašanje programa, ali može da spreči greške ukoliko se kôd modifikuje). Time se obezbeđuje da poredak slučajeva ne utiče na izvršavanje programa, te je takav kôd jednostavniji za održavanje.

Previđanje činjenice da se, ukoliko nema naredbi `break`, mogu izvršiti naredbe više 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).

8.7 Petlja `while`

Petlja `while` ima sledeći opšti oblik:

```
while(izraz)
   naredba
```

Ispituje se vrednost izraza `izraz` i ako ona predstavlja istinitosnu vrednost tačno (tj. ako je vrednost izraza različita od nule), izvršava se `naredba` (koja je ili pojedinačna naredba ili blok naredbi). Zatim se uslov `izraz` iznova testira i to se ponavlja sve dok mu istinosna vrednost ne postane *netačno* (tj. dok ne postane jednak 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 zagradama. Na primer:

```
while (i < j)
  i++;
```

Sledeća `while` petlja se izvršava beskonačno:

```
while (1)
  i++;
```

8.8 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` naredbe dodele ili inkrementiranja a `izraz2` je relacioni izraz čija se istinitosna vrednost ispituje. Obično se `izraz1` koristi za inicijalizaciju vrednosti promenljivih, izrazom `izraz2` se testira uslov izlaska iz petlje, a naredbom `izraz3` se menjaju vrednosti relevantnih promenljivih.

Izraz `izraz1` se izračunava samo jednom, na početku izvršavanja petlje. Petlja se izvršava sve dok izraz `izraz2` ima ne-nula vrednost (tj. sve dok mu je istinitosna vrednost *tačno*, a izraz `izraz3` se izračunava na kraju svakog prolaska kroz petlju. 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 je i početno dodeljivanje i uslov i izmene lako vidljivo na samom početku petlje). To ilustruje sledeća tipična forma `for` petlja:

```
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 koja prekida izvršavanje, npr. `break` ili `return`):

```
for (;;)
  ...
```

U okviru izraza `izraz1`, `izraz2`, `izraz3` mogu se naći zapete koji razdvajaju podizraze. Na primer,

```
for (i = 0, j = 10; i < j; i++, j--)
    ...
```

Iako se koristi najčešće u okviru `for` petlje, operator zapete je operator kao i svaki drugi (najnižeg je prioriteta od svih operatora u C-u) i prilikom izračunavanja vrednosti izraza izgrađenog primenom operatora zapeta, izračunavaju se oba operanda, pri čemu se vrednost celokupnog izraza definiše kao vrednost desnog operanda.

Sledeći program, koji ispusuje tablicu množenja, ilustruje dvostruku `for` petlju:

Program 8.2.

```
#include<stdio.h>

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

8.9 Petlja `do-while`

Petlja `do-while` ima sledeći opšti oblik:

```
do {
  naredbe
} while(izraz)
```

Naredba (ili blok naredbi naveden između vitičastih zagrada) `naredba` se izvršava i onda se izračunava `izraz`. Ako je on tačan, `naredba` se izračunava ponovo i to se nastavlja sve dok izraz `izraz` nema vrednost `0` (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, s desna na levo.

Program 8.3.

```
#include <stdio.h>
int main() {
  unsigned n;
  scanf("%u", &n);
  do {
    printf("%u", n % 10);
    n /= 10;
  } while (n > 0);
  return 0;
}
```

Primetimo da, u slučaju da je korišćena `while`, a ne `do-while` petlja, za broj `0` ne bi bila ispisana ni jedna cifra.

8.10 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`). Na primer,

```
for(i = 1; i < n; i++) {
  if(i>10)
    break;
  ...
}
```

Korišćenjem naredbe `break` se narušava strukturiranost koda, te može da ugrozi rezonovanje o njemu. Kôd koji koristi naredbu `break` uvek se može napisati i bez nje, međutim korišćenje naredbe `break` može da poboljša čitljivost. U datom primeru, odgovarajući alternativni kôd je, na primer:

```
for(i = 1; i<n && i<=10; i++)
  ...
```

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 koda, 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 */
    ...
```

Pitanja i zadaci za vežbu

Zadatak 8.1. *Napisati program koji ispisuje sve neparne brojeve manje od unetog neoznačenog celog broja* `n`.

Zadatak 8.2. *Napisati program koji izračunava i ispisuje vrednost funkcije* $\sin(x)$ *u 100 ekvidistantno razmaknutih tačaka intervala* $[0, 2\pi]$.

Zadatak 8.3. *Napisati program koji učitava realan broj* x *i neoznačeni ceo broj* n *i izračunava* x^n.

Zadatak 8.4. *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 8.5. *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`*). Napisati i program koji određuje sumu svih delilaca broja.*

Zadatak 8.6. *Napisati program koji određuje da li je uneti neoznačeni ceo broj prost.*

Zadatak 8.7. *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 8.8. *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 8.9. *Napisati program koji učitava cele brojeve sve dok se ne unese* 0 *i izračunava dužinu najduže serije uzastopnih jednakih brojeva.*

Zadatak 8.10. *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 8.11. *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 8.12. *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 8.13. *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 8.14. *Napisati program koji ,,nadovezuje" dva uneta neoznačena cela broja (na primer, za unos* `123` *i* `456` *program ispisuje* `123456`*).*

Zadatak 8.15. *Napisati program koji izračunava broj dobijen obrtanjem cifara unetog celog broja (na primer, za unos* `1234` *program ispisuje broj* `4321`*).*

Zadatak 8.16. *Napisati program koji izbacuje sve neparne cifre iz zapisa unetog neoznačenog celog broja (na primer, za uneti broj* 123456 *program ispisuje* 246*).*

Zadatak 8.17. *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 8.18.

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 8.19. *Napisati program koji za zadati dan, mesec i godinu ispituje da li je uneti datum korektan (uzeti u obzir i prestupne godine).*

Glava 9

Struktura programa i funkcije

Svaki C program sačinjen je od funkcija. Funkcija `main` mora da postoji, a druge funkcije se koriste kako bi kôd bio kraći, čitljiviji, upotrebljiviji, itd. U funkciju se obično izdvaja neko izračunavanje, neka obrada koja se koristi više puta u programu. Tako se dobija kraći, jednostavniji kôd, ali i elegantniji u smislu da je neko izračunavanje skriveno, enkapsulirano u funkciji i ona može da se koristi čak i ako se ne zna kako je ona implementirama, dovoljno je znati šta ona radi, tj. kakav je rezultat njenog rada za zadate argumente. Izvršavanje programa uvek počinje izvršavanjem funkcije `main`.

9.1 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 9.1.

```
#include <stdio.h>

int kvadrat(int n) {
    return n*n;
}

int main() {
    printf("Kvadrat od %i je %i\n", 5, kvadrat(5));
    printf("Kvadrat od %i je %i\n", 9, kvadrat(9));
    return 0;
}
```

Red `int kvadrat(int n);` *deklariše* funkciju `kvadrat` koja će biti *definisana* kasnije u kodu. U okviru funkcije `printf` poziva se funkcija `kvadrat` za vrednosti `5` i `9` i ispisuje se rezultat, tipa `int`, koji ona vraća. Definicija funkcije `kvadrat` je jednostavna: ona ima jedan argument (ceo broj `n`) i, kao rezultat, vraća kvadrat ove vrednosti.

U narednom programu, funkcija `stepen` izračunava celobrojni stepen realne promenljive.

Program 9.2.

```
#include <stdio.h>

float stepen(float x, unsigned n);

main() {
  unsigned i;
  for(i = 0; i < 10; 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;
}
```

9.2 Definicija funkcije

Definicija funkcije ima sledeći opšti oblik:

```
tip-povratne-vrednosti ime-funkcije(niz deklaracija argumenata) {
   deklaracije
   naredbe
}
```

Imena funkcija su proizvoljni identifikatori i za njih važe potpuno ista pravila kao i za imena promenljivih. Radi čitljivosti koda, poželjno je da ime funkcije oslikava ono šta ona radi.

Definicije funkcija se mogu 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.

9.3 Povratna vrednost funkcije

Funkcija može da vraća rezultat i `tip-povratne-vrednosti` označava tip vrednosti koja se vraća kao rezultat. Funkcija rezultat vraća naredbom `return r;` gde je `r` vrednost zadatog tipa ili tipa koji se može prevesti u taj tip. Generalno, naredba `return r;` ne samo da vraća vrednost `r` kao rezultat rada funkcije, nego i prekida njeno izvršavanje. Ukoliko funkcija ne vraća rezultat, onda je kao tip povratne vrednosti navodi specijalan tip `void` i tada naredba `return` nema argumenata. Štaviše, u tom slučaju nije 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.

9.4 Argumenti funkcije

Funkcija može imati argumente (tj. parametre), promenljive koje obrađuje, i oni se navode u okviru definicije, iza imena funkcije i između zagrada. Pre svakog argumenta neophodno je navesti njegov tip. Ukoliko funkcija nema argumenata, onda se između zagrada ne navodi ništa ili se navodi ključna reč `void`.

Promenljive koje su deklarisane kao argumenti funkcije su lokalne za tu funkciju i njih ne mogu da koriste druge funkcije. Štaviše, bilo koja druga funkcija može da koristi isto ime za neki svoj argument ili za neku svoju lokalnu promenljivu.

Kao i imena promenljivih, imena argumenata treba da oslikavaju njihovo značenje i ulogu u programu.

9.5 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. Na primer, funkcija `kvadrat` iz primera može biti pozvana sa `kvadrat(5)`, ali i sa `kvadrat(2+3)`.

Argumenti (osim nizova) se uvek prenose *po vrednosti*. To znači da se vrednost koji je poslata kao argument funkcije *kopira* kada počne izvršavanje funkcije i ona 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 argument funkcije i funkcija će koristiti samo tu kopiju u svom radu. Prenos argumenata ilustruje i funkcija `swap` definisana na sledeći način:

```
void swap(int a, int b) {
   int temp = a;
   a = b;
   b = temp;
}
```

U okviru funkcije `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.

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 daljem tekstu.

9.6 Deklaracija funkcije

U navedenom primeru, deklaracija (ili prototip) `int kvadrat(int n);` ukazuje prevodiocu da će u programu biti korišćena funkcija sa ovim tipom povratne vrednosti i ovakvim argumentima. Zahvaljujući tome, kada prevodilac, u okviru funkcije `main`, naiđe na poziv funkcije `kvadrat`, može da proveri da li je njen poziv ispravan. Pošto prototip služi samo za proveravanje tipova u pozivima, nije neophodno navoditi imena argumenata, dovoljno je navesti njihove tipove. U navedenom primeru, dakle, prototip je mogao da glasi i `int kvadrat(int);`. Ukoliko se prototip ne navede, pretpostavlja se da funkcija vraća vrednost tipa `int`, dok se provera ispravnosti argumenata u pozivima funkcije ne vrši. Slično, ukoliko se u okviru prototipa navedu prazne zagrade (npr. `int f();`), time se naglašava da se ne želi provera korektnosti argumenata u pozivima. Ukoliko se želi naglasiti da funkcija nema argumenata i ukoliko se želi prijavljivanje greške za svaki poziv sa argumentima, u okviru prototipa je potrebno navesti tip void za argumente (na primer, `int f(void);`).

Definicija funkcija mora da bude u skladu sa navedenim prototipom, tj. moraju da se poklapaju tipovi povratne vrednosti i tipovi argumenata.

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. Zato je praksa da se na početku programa navode prototipovi funkcija, čime poredak njihovih definicija postaje irelevantan.

Pitanja i zadaci za vežbu

Pitanje 9.1. *Kako se prenose argumenti u C-u?*

Zadatak 9.2. *Napisati program koji proverava da li je uneti pozitivan ceo broj prost. Napisati varijantu koja ima samo funkciju* `main` *i varijantu u kojoj se koriste pomoće funkcije.*

Zadatak 9.3. *Za tri prirodna broja* a*,* b *i* c *kaže se da čine* Fermaovu trojku *ako postoji prirodan broj* n *takav da važi* $a^n + b^n = c^n$*. Napisati program koji za zadata tri prirodna broja proverava da li čine Fermaovu trojku za neko* n *koje je manje od 1000 (napisati i koristiti funkciju za stepenovanje).*

Zadatak 9.4. *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 9.5. *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.*

Glava 10

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 vrši na uniforman način, korišćenjem petlji.

Razmotrimo, kao ilustraciju, program kojim se izračunava koliko se puta javlja svaka od cifara u tekstu učitanom sa standardnog ulaza, do prve pojave tačke. Bez korišćenja nizova, program zahteva 10 različitih brojačkih promenljivih.

```
#include <stdio.h>
int main() {
  int b0 = 0, b1 = 0, b2 = 0, b3 = 0, b4 = 0,
      b5 = 0, b6 = 0, b7 = 0, b8 = 0, b9 = 0, c;
  while ((c = getchar()) != '.') {
    switch(c) {
      case '0': b0++; break;
      case '1': b1++; break;
      case '2': b2++; break;
      case '3': b3++; break;
      case '4': b4++; break;
      case '5': b5++; break;
      case '6': b6++; break;
      case '7': b7++; break;
      case '8': b8++; break;
      case '9': b9++; break;
    }
  }
  printf("%d %d %d %d %d %d %d %d %d %d\n",
          b0, b1, b2, b3, b4, b5, b6, b7, b8, b9);
  return 0;
}
```

Za čitanje pojedinačnih karaktera sa standardnog ulaza upotrebljena je funkcija standardne C biblioteke `getchar()` deklarisana u zaglavlju `stdio.h` o kojoj će više reči biti u poglavlju 20.

Umesto 10 različitih srodnih promenljivih, moguće je upotrebiti niz i time značajno uprostiti prethodni program.

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

int main() {
  int b[10], c, i;

  for (i = 0; i < 10; i++)
     b[i] = 0;

  while ((c = getchar()) != '.')
    if (isdigit(c))
       b[c - '0']++;

  for (i = 0; i < 10; i++)
    printf("%d ", b[i]);

  return 0;
}
```

Na i-tom mestu u nizu `b` se smešta broj pojavljivanja cifre i. Funkcija standardne biblioteke `isdigit` deklarisana u zaglavlju `ctype.h`, je upotrebljena da bi se proverilo da li je uneti karakter cifra. Ako jeste, od njegovog koda se oduzima ASCII kod karaktera `'0'` kako bi se dobila odgovarajuća pozicija u nizu brojača.

10.1 Deklaracija niza

Nizovi u programskom jeziku C se mogu 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 početna inicijalizacija:

```
int a[5] = { 1, 2, 3, 4, 5 };
int b[] = { 1, 2, 3 };
```

Nakon ovih deklaracija, sadržaj niza `a` jednak je:

1	2	3	4	5

a sadržaj niza `b` jednak je:

1	2	3

U slučaju inicijalizacije, moguće je bilo izostaviti dimenziju (kada se ona izračunava na osnovu broja elemenenata u inicijalizatoru), bilo navesti veću dimenziju od broja navedenih elemenata (kada se inicijalizuju samo početni elementi deklarisanog niza). Neispravnim se smatra navođenje manje dimenzije od broja elemenata inicijalizatora.

Pošto se zahteva da je u fazi prevođenja programa moguće odrediti memorijski prostor koji niz zauzima, broj elemenata niza (koji se navodi u deklaraciji) mora biti konstantan izraz. Dimenziju niza je moguće izostaviti samo ako je prilikom deklaracije izvršena i inicijalizacija niza. U narednom primeru, deklaracija niza `a` je ispravna, dok su deklaracije nizova `b` i `c` neispravne:

```
int x;
char a[100+10];
int b[];
float c[x];
```

Kada se pristupa elementu niza, indeks može da bude proizvoljan izraz celobrojne vrednosti, na primer:

```
a[2*2+1]
```

C prevodilac ne vrši nikakvu proveru da li je indeks pristupa nizu u njegovim granicama i moguće je bez ikakve prijave greške ili upozorenja od strane prevodioca pristupiti i elementu koji se nalazi van opsega deklarisanog niza (recimo — pristupiti elementu a[13] u prethodnom primeru). Ovo najčešće dovodi do fatalnih grešaka prilikom izvršavanja programa.

Nizovi ne predstavljaju l-vrednosti i nije im moguće dodeljivati vrednosti niti menjati (ove operacije su moguće nad pojedinačnim elementima nizova, ali nisu nad celim nizovima). 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. */
```

10.2 Nizovi kao argumenti funkcija

Niz se ne može preneti kao argument funkcije. Umesto toga, moguće je kao argument proslediti ime niza. Imenu niza pridružena je informacija o adresi

početka niza, o tipu elemenata niza i o broju elemenata niza. Kada se ime niza prosledi kao argument funkcije, onda do funkcije (čiji je tip odgovarajućeg argumenta pokazivački tip) stiže informacija o adresi početka niza i o tipu elemenata (ali ne i o broju elemenata niza). Prenos takvog argumenta (pokazivača na početak niza) vrši se — kao i uvek — po vrednosti. S obzirom na to da funkcija koja je pozvana dobija informaciju o adresi početka niza, ona može da neposredno menja njegove elemente.

Program 10.1.

```
#include <stdio.h>

void ucitaj_broj(int a) {
  scanf("%d", &a);
}

void ucitaj_niz(int a[], int n) {
  int i;
  for (i = 0; i < n; i++)
     scanf("%d", &a[i]);
}

int main() {
  int x;
  int y[10];
  ucitaj_broj(x);
  ucitaj_niz(y, 10);
}
```

Funkcija `ucitaj_broj` ne uspeva da učita i promeni vrednost broja `x`, dok funkcija `ucitaj_niz` ispravno unosi elemente niza `y`.

Prilikom prenosa niza u funkciju, uz ime niza neophodno je proslediti i broj elemenata niza (kako bi funkcija koja prima argument imala tu informaciju). Izuzetak od ovog pravila predstavljaju funkcije koje obrađuju niske karaktera jer je u tom slučaju na osnovu sadržaja niske moguće odrediti i njegov broj elemenata.

10.3 Niske

Posebno mesto u programskom jeziku C zauzimaju nizovi koji sadrže karaktere, tj. *niske karaktera* (*stringovi* (eng. *strings*), ili jednostavno *niske*). Niske u programskom jeziku C se navode između dvostrukih navodnika `""` (na primer, `"ja sam niska."`). U okviru niski specijalni karakteri navode se korišćenjem specijalnih sekvenci (na primer, `prvi red\ndrugi red`).

Tehnički, niske su predstavljene kao nizovi karaktera na čiji desni kraj se dopisuje karakter `'\0'` (tzv. terminalna nula (eng. *null terminator*)). Iz ovoga razloga, niske u C-u se nazivaju *niske terminisane nulom* (eng. *null terminated*

strings)[1]. Posledica ovoga je da ne postoji ograničenje za dužinu niske u C-u, ali da 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). Dakle, potrebno je jasno razlikovati karakterske konstante (npr. `'x'`) koje predstavljaju pojedinačne karaktere i niske (npr. `"x"`) koje sadrže dva karaktera (`'x'` i `'\0'`).

Ukoliko se u programu niske neposredno nađu jedna uz drugu, one se automatski spajaju. Na primer:

```
printf("Zdravo, " "svima");
```

je ekvivaletno sa

```
printf("Zdravo, svima");
```

10.4 Standardne funkcije za rad sa niskama

C standardna biblioteka definiše veliki broj funkcija za rad sa niskama. Prototipovi ovih funkcija se nalaze u zaglavlju `string.h`. U ovom poglavlju, ilustracije radi, biće prikazano kako se mogu implementirati neke od njih. Naravno, u svakodnevnom programiranju, ne preporučuje se definisanje ovih funkcija nego se preporučuje korišćenje standardnih, već definisanih, funkcija. U narednom poglavlju, kada bude uvedeni pokazivači, biće prikazane alternativni načini implementacije ovih funkcija.

Funkcija `strlen` služi za izračunavanje dužine niske. Pri računanju dužine niske, ne računa se terminalna nula.

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

Mnoge funkcije koje obrađuju niske obrađuju ih karakter po karakter i sadrže petlju oblika:

[1] 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++)
   ...
```

Imajući u vidu da terminalna nula ima i numeričku vrednost 0 (što predstavlja istinitosnu vrednost netačno), poređenje u prethodnoj petlji može da se izostavi:

```
for (i = 0; s[i]; i++)
   ...
```

Česta greška (zbog neefikasnosti) je pozivanje funkcije `strlen` za izračunavanje dužine niske u svakom koraku iteracije:

```
for (i = 0; i < strlen(s); i++)
   ...
```

Funkcija `strcpy` vrši kopiranje jedne niske u drugu, pretpostavljajući da je niska kojoj se dodeljuje vrednost dovoljno velika da primi nisku koja se dodeljuje. Ukoliko ovo nije ispunjeno, vrši se izmena sadržaja memorijskih lokacija koje su van dimenzija niza što može dovesti do fatalnih grešaka prilikom izvršavanja programa.

```
void strcpy(char dest[], char src[]) {
  int i = 0;
  while ((dest[i] = src[i]) != '\0')
     i++;
}
```

U nisku `dest` se prebacuje karakter po karakter niske `src` sve dok dodeljeni karakter ne bude terminalna nula.

Funkcija `strchr` proverava da li data niska sadrži dati karakter. Funkcija `strstr` proverava da li jedna niska sadrži drugu. Obe ove funkcija iz standardne biblioteke imaju drugačiju povratnu vrednost nego što je ovde prikazano. Funkcije identične onima iz standardne biblioteke će biti prikazane kada bude uveden pojam pokazivača.

```
/* Pronalazi prvu poziciju karaktera c u stringu s, odnosno -1
   ukoliko s ne sadrzi c */
int strchr(char s[], char c) {
  int i;
  for (i = 0; s[i] != '\0'; i++)
     if (s[i] == c)
        return i;

  return -1;
}
```

Funkcija `strchr` koristi tzv. linearnu pretragu niza. Česta greška prilikom implementacije linearne pretrage je prerano vraćanje negativne vrednosti:

```
...
  for (i = 0; s[i] != '\0'; i++)
     if (s[i] == c)
        return i;
     else
        return -1;
...
```

Funkcija `strstr` proverava da li jedna zadata niska sadrži drugu zadatu nisku (postoje efikasniji algoritmi za rešavanje ovog problema, od naivnog algoritma koji je ovde naveden).

```
/* Proverava da li niska str sadrzi nisku sub.
   Vraca poziciju na kojoj sub pocinje, odnosno -1 ukoliko ga nema
*/
int strstr(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;
}
```

Funkcija `strrev` obrće datu nisku.

```
void strrev(char s[]) {
  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;
  }
}
```

Funkcija `strcmp` poredi dve niske leksikografski (kao u rečniku, sortiranom po ASCII redosledu). 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.

```
int strcmp(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];
}
```

Pitanja i zadaci za vežbu

Zadatak 10.1. *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 10.2. *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 10.3. *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. *proverava da li je dati niz palindrom (čita se isto sleva i zdesna);*
6. *obrće dati niz brojeva;*
7. *rotira sve elemente datog niza brojeva za* `k` *pozicija ulevo;*
8. *rotira sve elemente datog niza brojeva za* `k` *pozicija udesno;*
9. *izbacuje višestruka pojavljivanja elemenata iz datog niza brojeva (napisati varijantu u kojoj se zadržava prvo pojavljivanje i varijantu u kojoj se zadržava poslednje pojavljivanje).*
10. *spaja dva niza brojeva koji su sortirani neopadajući u treći niz brojeva koji je sortiran neopadajući.*

Zadatak 10.4. *Sa standardnog ulaza se unose dva niza brojeva. Napisati program koji određuje njihov presek, uniju i razliku (redosled prikaza elemenata nije bitan, pri čemu se elementi ne ponavljaju).*

Zadatak 10.5. *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 10.6. *Napisati program koji razbija niz brojeva na najveći mogući broj grupa uzastopnih elemenata. Na primer, niz* `7 2 3 8 9 7 4 2 6 8 5 6 1` *se deli na* `7 2 4 8 9 7 4 2`, `3`, `6 8 5 6` *i* `1`.

Zadatak 10.7. *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 10.8. *Napisati funkciju koja proverava da li je data niska palindrom (čita se isto sleva i zdesna), pri čemu se razmaci zanemaruju i ne pravi se razlika između malih i velikih slova (npr.* `Ana voli Milovana` *je palindrom).*

Glava 11

Pretprocesor

Kada se od izvornog programa napisanog na jeziku C proizvodi program na mašinskom jeziku (izvršni program), pre samog prevodioca poziva se C pretprocesor. C pretprocesor ignoriše naredbe napisane u jeziku C i obrađuje samo takozvane pretprocesorske direktive (one se ne smatraju delom jezika C). Suštinski, obrade koje vrši C pretprocesor ekvivalentne su jednostavnim operacijama nad tekstualnim sadržajem programa. Dve najčešće korišćene pretprocesorske direktive su `#include` (za uključivanje sadržaja neke druge datoteke) i `#define` koja zamenjuje neki tekst drugim tekstom (tzv. makroi). Pretprocesor omogućava i definisanje makroa sa argumentima, kao i uslovno prevođenje.

Rezultat rada pretprocesora, može se dobiti korišćenje GCC prevodioca navođenjem opcije `-E` (npr. `gcc -E program.c`).

11.1 Uključivanje datoteka zaglavlja

Veliki programi su obično podeljeni u više datoteka, radi preglednosti i lakšeg održavanja. U svakoj takvoj datoteci bi na početku mogli da budu navedeni prototipovi funkcija koje su u njoj definisane. Međutim, i funkcije iz drugih datoteka mogu da koriste funkcije iz te datoteke. Kako bi se pojednostavilo baratanje takvim funkcijama a i ubrzalo prevođenje, obično se samo prototipovi izdvajaju u zasebne datoteke koje se onda koriste gde god je potrebno. Takve datoteke zovu se *datoteke zaglavlja*. U datoteku koja se prevodi, sadržaj neke druge datoteke se uključuje 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 posebne kolekcije direktorijuma *include path* (koja se najčešće prevodiocu zadaje preko `-I`) i koja obično podrazumevano sadrži direktorijum u kome se nalazi datoteka u koju se vrši uključivanje. Uko-

liko 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.

Tipično, uključuju se zaglavlja iz standardne biblioteke (na primer, `stdio.h`) ili datoteke koje sadrže deklaracije funkcija koje čine deo programa (a koji se sastoji od više datoteka). Time se, prvenstveno, omogućava prevođenje datoteke koja sadrži pozive drugih funkcija. Dodatno, time se olakšava modularizaciju programa.

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`.

11.2 Makro zamene

Pretprocesorska direktiva `#define` omogućava zamenjivanje niza karaktera u datoteci drugim nizom karaktera pre samog prevođenja. Njen opšti oblik je:

```
#define originalni_tekst novi_tekst
```

U najjednostavnijem obliku, ova direktiva se koristi za zadavanje vrednosti nekom simboličkom imenu, na primer:

```
#define NAJVECA_DUZINA_IMENA 80
```

Ovakva definicija se koristi kako bi se izbeglo navođenje 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, `NAJVECA_DUZINA_IMENA` je samo simboličko ime i nikako ga ne treba mešati sa promenljivom (čak ni sa promenljivom koja je `const`). Naime, za ime `NAJVECA_DUZINA_IMENA` se nigde ne rezerviše prostor, već se svako njeno projavljivanje, pre samog prevođenja zamenjuju zadatom vrednošću (u navedenom primeru — vrednošću 80).

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 definicije direktiva koje im prethode. Doseg primene direktive zamene je od mesta na kojem se nalazi do kraja datoteke ili do reda oblika:

```
#undef originalni_tekst
```

Zamene se ne vrše u okviru konstantnih niski niti u okviru drugih simboličkih imena. Na primer, gore navedena direktiva neće uticati na komandu

```
printf("NAJVECA_DUZINA_IMENA je 80");
```

niti na simboličko ime

`NAJVECA_DUZINA_IMENA_ULICE`.

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 okviru neke naredbe navedno `max(2,3)`, to će, pre prevođenja, biti zamenjeno sa `((2) > (3) ? (2) : (3))`. Ukoliko je navedeno `max(x+2,3*y)`, ono se zamenjuje sa `((x+2) > (3*y) ? (x+2) : (3*y))`. Naglasimo da `max(2,3)` (i slično) nije poziv funkcije i da 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 vrednosi `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.

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 ispred imena parametra stavi `#` u okviru teksta kojim se zamenjuje, kombinacija će biti zamenjena vrednošću parametara navedenom u okviru 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)
```

Tekst

```
dprint(x/y);
```

u okviru programa će biti zamenjen sa:

```
printf("x/y" " = %g\n", x/y);
```

Pošto se niske automatski nadovezuju, efekat je isti kao:

```
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;
```

Makroi i definicije funkcija mogu izgledati slično, ali se suštinski razlikuju. Kod makroa 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. Sve u svemu, makro zamene sa argumentima imaju i dobre loše i svoje loše strane i treba ih koristiti oprezno.

Mnoge standardne ,,funkcije" za ulaz i izlaz su u C standardnoj biblioteci su zapravo makroi (na primer, `getchar` preko `getc`, `printf` preko `fprintf`, itd.).

11.3 Uslovno prevođenje

Pretprocesorskim direktivama je moguće isključiti delove koda 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 direktive `#else` ili 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 direktivno, 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 11.1.

```
#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. Primetimo da ovo grananje nije isto kao grananje koje koristi C naredbu `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.

Glava 12

Konverzije tipova

Konverzija tipova predstavlja pretvaranja podatka jednog tipa u podatak drugog tipa. 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.

Razlikujemo eksplicitne konverzije (koje se izvršavaju na zahtev programera) i implicitne konverzije (koje se izvršavaju automatski kada je to neophodno). Neke konverzije je moguće izvesti bez gubitka informacija, dok se u nekim slučajevima prilikom konverzije vrši izmena same vrednosti podatka. *Promocija* predstavlja konverziju vrednosti manjeg tipa u vrednost većeg tipa (na primer, `int` u "float"ili `float` u `double`) u kom slučaju ne dolazi do gubitka informacije. *Democija* 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.

```
float a = 4;  /* 4 je int pa se implicitno promovise u float */
int b = 7.0f; /* 7.0f je float pa se vrsi democija u 7 */
int c = 7.5f; /* 7.5f je float i ovde se takoe vrsi democija u 7,
                 pri cemu se gubi informacija */
unsigned char d = 256;
```

Prilikom konverzije iz realnih u celobrojne tipove podataka potrebno je potpuno izmeniti interni zapis podataka (na primer, iz IEEE754 zapisa u zapis u obliku potpunog komplementa).

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 (eng. zero extension). U slučaju da je polazni tip označen, promocija se vrši proširivanjem vodećeg bita (eng. *sign extension*).

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, 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 (eng. printable characters) uvek imati pozitivne kodove, tako da oni ne predstavljaju problem prilikom konverzija. Ukoliko se tip `char` koristi za smeštanje nečeg drugog osim karaktera koji mogu da se štampaju, obavezno se preporučije eksplicitno navođenje kvalifikatora `signed` ili `unsigned` kako bi se programi izvršavali identično na svim računarima.

U slučaju konverzija realnih tipova, ukoliko je vrednost takva da se ne može zapisati u novom tipu, vrši se zaokruživanje na najbliže vrednosti koje se mogu zapisati (tj. odsecanje).

Kao što se može naslutiti, brojevi (i celi i realni) se prevode u istinitosne vrednosti tako što se 0 prevodi u 0, a sve ne-nula vrednosti se prevode u 1.

12.1 Eksplicitne konverzije

Operator eksplicitne konverzije tipa ili operator kastovanja (eng. type cast operator) se navodi tako što se ime rezultujućeg tipa navedi u malim zagradama ispred izraza koji se konvertuje, tj. oblika je `(tip)izraz`. Operator kastovanja je unaran 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 želi primena realnog deljenja na celobrojne operande:

```
int a = 13, b = 4;
printf("%d\n", a/b);
printf("%f\n", (double)a/(double)b);
```

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 drugi operand implicitno konvertovao.

12.2 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 prethodno primeru, prilikom dodele promenljivoj `a`, vrši se democija `double` konstante 3.5 u vrednost 3 tipa `int`, a zatim, prilikom dodele promenljivoj `b`, promocija te vrednosti u `double` vrednost 3.0.

12.3 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* (eng. *usual arithmetic conversions*). Ove konverziju se, na primer, primenjuju prilikom primene aritmetičkih (`+`, `-`, `*`, `/`) i relacijskih binarnih operatora (`<`, `>`, `<=`, `>=`, `==`, `!=`), prilikom primene uslovnog operatora `?:`.

Pored ujednačavanja tipa operanada, 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* (eng. integer promotion).

```
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 se dobila vrednost $(100 * 3)/4$). Pošto se promocija vrši, dobija se 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. zatim, 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 komplikovanija.

1. Ako je neoznačeni operand širi[1] 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 raznih 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.

Međutim, važi da je `-1l > 1ul`. Zaista, u ovom primeru 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` 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) {
  ...
}
```

12.4 Konverzije tipova argumenata funkcije

Prilikom poziva funkcije, ukoliko je poznata njena deklaracija, vrši se implicitna konverzija stvarnih argumenata u tipove formalnih argumenata. Slično, prilikom vraćanja vrednosti funkcije (`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`:

[1] Preciznije, standard uvodi pojam konverzionog ranga (koji raste sa od `char` ka `long long`) i na ovom mestu se razmatra konverzioni rang.

```
#include <stdio.h>

void f(int a) {
  printf("%d\n", a);
}

int main() {
  f(3.5);
}
```

Ukoliko deklaracija funkcije nije poznata ne vrši se nikakva provera ispravnosti tipa argumenata i vrši se podrazumevana promocija argumenata koja obuhvata celobrojne promocije (tipovi `char` i `short` se promovišu u `int`), i promociju tipa `float` u tip `double`.

Razmotrimo, na primer, standardnu C funkciju `double sqrt(double);` koja izračunava kvadratni koren. Deklaracija ove funkcije nalazi se u zaglavlju `<math.h>`. Ukoliko se u programu uključi ovo zaglavlje, a zatim funkcija pozove sa `sqrt(2)` doći će do implicitne konverzije celobrojne konstante 2 u tip `double` i funkcija će korektno izračunati koren. Međutim, ukoliko se ne uključi `<math.h>`, prototip funkcije `sqrt` neće biti poznat i neće se vršiti konverzija celobrojne u realnu vrednost već se funkciji prosleđuje binarni sadržaj koji predstavlja celobrojnu konstantu 2 (obično u obliku potpunog komplementa). S druge strane, funkcija taj binarni sadržaj tumači kao realan broj (obično po IEEE754 standardu) i to dovodi do greške (tj. neće biti pročitan broj 2.0 nego neki drugi). U ovom primeru, stvar je još pogoršana činjenicom da je tip `int` obično kraći od tipa `double` tako da funkcija pored vrednosti koja joj je prosleđena tumači i neki relativno slučajni memorijski sadržaj koji se nalazi iza zapisa broja 2.

Glava 13

Izvršno okruženje i organizacija memorije

U ovoj glavi biće objašnjeno kako je organizovana i kako se koristi memorija koja je dodeljena jednom programu. Tekst u nastavku odnosi se, kada to nije drugačije naglašeno na širok spektar platformi, pa su, zbog toga, načinjena i neka pojednostavljivanja.

13.1 Organizacije memorije dodeljene programu

Kada operativni sistem učita izvršni program, dodeljuje mu određenu memoriju i započinje njegovo izvršavanje. Dodeljena memorija organizovana je u nekoliko delova koje zovemo *segmenti* ili *zone*:

- segment koda (eng. code segment ili text segment)
- segment podataka (eng. data segment)
- stek segment (eng. stack segment)
- hip segment (eng. heap segment)

U nastavku će biti opisana prva tri, dok će o hip segmentu biti više reči u glavi 18, posvećenoj dinamičkoj alokaciji memorije.

13.2 Segment koda

U ovom segmentu se nalazi sâm izvršni kôd programa — njegov mašinski kôd koji uključuje sve korisnikove funkcije a može da uključuje i korišćene sistemske funkcije. 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 koda. U tom slučaju, za svaku instancu se, naravno, zasebno čuva informacija o tome do koje naredbe je stiglo izračunavanje.

13.3 Segment podataka

U data segmentu čuvaju se određene vrste promenljivih koje su zajedničke za ceo program (tzv. statičke i spoljašnje promenljive), kao i konstantne niske. Ukoliko se istovremeno izvršava više instanci istog programa, svaka instanca ima svoj zaseban segment podataka.

13.4 Stek segment

U stek segmentu (koji se naziva i *programski stek poziva (eng. call stack)*) čuvaju se svi podaci koji karakterišu izvršavanje funkcija. Podaci koji odgovaraju jednoj funkciji (ili, preciznije, jednoj instance jedne funkcije — jer, na primer, jedna funkcija može da poziva samu sebe i da tako u jednom trenutku bude aktivno više njenih instanci) organizovani su u takozvani *stek okvir (eng. stack frame)*. Stek okvir jedne instance funkcije obično, između ostalog, sadrži:

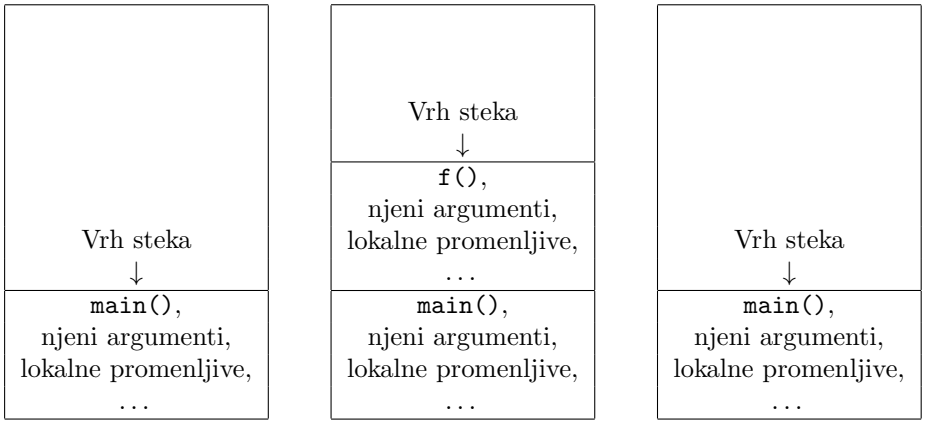
- argumente funkcije;
- lokalne promenljive (promenljive deklarisane unutar funkcije);
- međurezultate izračunavanja;
- adresu povratka (koja ukazuje na to odakle treba nastaviti izvršavanje programa nakon povratka iz funkcije);
- adresu stek okvira funkcije pozivaoca.

Stek poziva je struktura tipa *LIFO* ("last in - first out")[1]. To znači da se stek okvir može dodati samo na vrh steka i da se sa steka može ukloniti 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 briše, oslobađa (preciznije, smatra se nepostojećim) onda kada se završi izvršavanje funkcije.

Izvršavanje programa počinje izvršavanjem funkcije `main`, te se prvi stek okvir 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 13.1). 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).

Opisana organizacija steka omogućava jednostavan mehanizam međusobnog pozivanja funkcija, kao i rekurzivnih poziva. Predefinisana veličina steka C prevodioca se može promeniti zadavanjem odgovarajuće opcije.

[1] Ime *stack* je zajedničko ime za strukture podataka koje su okarakterisane ovim načinom pristupa.



Slika 13.1: 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()`.

13.5 Ilustracija funkcionisanja izvršnog okruženja: rekurzija

Rekurzija je situacija u kojoj jedna funkcija poziva sebe samu direktno ili indirektno. Razmotrimo, kao primer, funkciju koja rekurzivno izračunava faktorijel:[2]

```
#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

[2] Vrednost faktorijela se, naravno, može izračunati i iterativno, bez korišćenja rekurzije.

trenutku ima 5 aktivnih instanci funkcije `faktorijel`, postoji i koristi se samo jednan primerak izvršnog koda ove funkcije (u kôd segmentu), a svaki stek okvir pamti za svoju instancu dokle je stiglo izvršavanje funkcije, tj. koja naredba u kôd segmentu tekuća.

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.

Glava 14

Doseg i životni vek

U ovoj glavi biće reči o tome šta je i koje su vrste dosega identifikatora koji određuje u kojim delovima programa je moguće određene identifikatore koristiti. Takođe, biće reči i o životnom veku promenljivih koji određuje kada promenljive nastaju i kada se oslobađaju.

14.1 Doseg

Doseg identifikatora (eng. scope) određuje deo teksta programa u kome je moguće koristiti određeni identifikator i u kome taj identifikator identifikuje određeni objekat (na primer, promenljivu ili funkciju). Jezik C spada u grupu jezika sa statičkim pravilima dosega što znači da se doseg svakog identifikatora može jednoznačno utvrditi analizom izvornog koda, bez obzira na to kako teče proces izvršavanja programa. U programskom jeziku C, razlikuje se *doseg nivoa datoteke* (eng. file level scope) koji podrazumeva da ime važi od tačke uvođenja do kraja datoteke, *doseg nivoa bloka* (eng. block level scope) koji podrazumeva da ime važi od tačke uvođenja do kraja bloka u kome je uvedeno, *doseg nivoa funkcije* (eng. function level scope) koji podrazumeva da ime važi od tačke uvođenja do kraja funkcije u kojoj je uvedeno i *doseg nivoa prototipa funkcije* (eng. function prototype scope) koji podrazumeva da ime važi u okviru prototipa (deklaracije) funkcije. Prva dva nivoa dosega su najznačajnija. 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*. Doseg nivoa funkcije imaju samo labele (koje se najčešće koriste uz `goto` naredbu), dok doseg nivoa prototipa funkcije imaju samo imena parametara u okviru prototipova funkcije. U ranijim verzijama jezika C nije bilo moguće definisati funkciju u okviru definicije druge funkcije (te nema lokalnih funkcija), dok standard C99 uvodi i takvu mogućnost.

Navedimo primere različitog nivoa 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 */
```

Jezik C dopušta tzv. konflikt identifikatora tj. moguće je da postoji više identifikatora istog imena, pri čemu su njihovi dosezi jedan u okviru drugog i 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
```

14.2 Životni vek

Životni vek (eng. storage duration, lifetime) objekta je deo vremena izvršavanja programa u kome se garantuje da je za tu promenljivu rezervisan deo memorije. Jezik C razlikuje *statički (eng. static)*, *automatski (eng. automatic* i *dinamički (eng. dynamic)* životni vek. Statički životni vek znači da je objekat dostupan tokom celog izvršavanja programa. Automatski životni vek najčešće imaju promenljive koje se automatski stvaraju i uklanjaju prilikom pozivanja funkcija. O ova dva tipa životna veka biće reči u nastavku ove glave, dok će dinamički vek biti diskutovan u glavi 18. Životni vek nekog objekta se određuje na osnovu pozicije u kodu na kojoj je objekat uveden ili na osnovu eksplicitnog korišćenja nekog od kvalifikatora `auto` (automatski životni vek) ili `static` (statički životni vek).

14.3 Lokalne automatske promenljive

Najčešće korišćene lokalne promenljive su promenljive deklarisane u okviru bloka. Njihov doseg je nivoa bloka i one su dostupne od tačke deklaracije do kraja bloka. Lokalne promenljive nije moguće koristiti van bloka u kome su deklarisane. Ne postoji nikakva veza između promenljivih istog imena deklarisanih u različitim blokovima.

Lokalne promenljive su podrazumevano automatskog životnog veka (sem ako je na njih primenjen kvalifikator `static` ili kvalfikator `extern` — videti u nastavku). Iako je moguće je i eksplicitno okarakterisati životni vek korišćenjem kvalifikatora `auto`, ovo se obično ne radi jer se za ovakve promenljive automatski životni vek podrazumeva. Početna vrednost lokalnih automatskih promenljivih nije određena.

Automatske promenljive ,,postoje" samo tokom izvršavanja funkcije u kojoj su deklarisane i prostor za njih je rezervisan u stek okviru te funkcije. Ne postoji veza izmedju promenljive jednog imena u različitim aktivnim instancama jedne funkcije (jer svaka instanca funkcije ima svoj stek okvir a 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.

14.4 Lokalne statičke promenljive

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 oslobađaju prilikom završetka rada programa. Tako modifikovana promenljiva ne čuva se u stek okviru svoje funkcije, već u segmentu podataka. 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 rada programa[1]. Doseg ovih promenljivih je i dalje doseg nivoa bloka tj. promenljive su i dalje lokalne.

```
#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
```

14.5 Globalne statičke promenljive i funkcije

Globalne promenljive su promenljive deklarisane van svih funkcija i njihov doseg je doseg nivoa datoteke što znači da su dostupne od tačke deklaracije do kraja izvorne datoteke. Globalne promenljive mogu da budu korišćene u svim funkcijama koje su u njihovom dosegu. Životni vek ovih promenljivih je statički, tj. prostor za ove promenljive je trajanje 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. 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 mogu da ih koriste više funkcija, one mogu da zamene prenos podataka između funkcija. Međutim, to treba činiti samo sa dobrim razlogom

[1] Obratiti pažnju da je ovo različito u odnosu na jezik C++ gde se inicijalizacija vrši prilikom prvog ulaska u blok u kome je ovakva promenljiva definisana.

(na primer, ako najveći broj funkcija treba da koristi neku zajedničku promenljivu) jer, inače, program može postati nečitljiv i težak za održavanje.

Kao što je već rečeno, u ranijim verzijama jezika C sve funkcije su globalne (ali standard C99 uvodi i lokalne funkcije). Doseg globalnih funkcija je doseg nivoa datoteke i proteže se od mesta deklaracije pa do kraja datoteke.

Napomenimo da su globalno definisane promenljive i funkcije spoljašnji objekti i oni se, za razliku od lokalnih unutrašnjih objekata mogu povezati sa spoljašnjim objektima definisanim u nekoj drugoj jedinici prevođenja (datoteci). O ovome će biti više reči u poglavlju 21 koje govori o programima koji se sastoje od više jedinica prevođenja i njihovom povezivanju .

Glava 15

Pokazivači

U ovoj glavi će biti reči o pokazivačkim tipovima podataka i o njihovom korišćenju. U jeziku C, pokazivači imaju veoma značajnu ulogu i praktično je nemoguće napisati iole kompleksniji program bez upotrebe pokazivača.

15.1 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 predstavljaju tip podataka u C-u takav da su vrednosti ovog tipa memorijske adrese. U zavisnosti od sistema, adrese obično zauzimaju četiri (ranije dva, a novije vreme i osam) bajta. Pokazivačke promenljive (promenljive pokazivačkog tipa) su promenljive koje sadrže memorijske adrese. Iako su pokazivačke vrednosti (adrese) u suštini celi brojevi, pokazivački tipovi se striktno razlikuju od celobrojnih. Takođe, jezik C razlikuje više pokazivačkih tipova i tip pokazivača se određuje na osnovu tipa podataka na koji pokazuje. Ovo znači da pokazivači implicitno čuvaju informaciju o tipu onoga na šta ukazuju[1].

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, što je čest izvor grešaka.

```
int *p1;
int* p2;
int* p3, p4;
```

Dakle, u ovom primeru, `p1`, `p2` i `p3` su pokazivači koji ukazuju na `int` dok je `p4` običan `int`.

[1] Sa izuzetkom pokazivača tipa `void` koji nema informaciju o tipu podataka na koji ukazuje. O tome će biti reči u nastavku.

Kako bi pokazivačka promenljiva sadržala adresu nekog smislenog podatka potrebno je programeru dati mogućnost određivanja adresa objekata. Unarni operator `&` (koji zovemo ,,operator referenciranja“ ili ,,adresni operator”) vraća adresu svog operanda. On može biti primenjen samo na promenljive i elemente, 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. Derefencirani pokazivač može biti l-vrednost i u tom slučaju izmene dereferenciranog pokazivača utiču neposredno na prostor na koji se ukazuje. Ukoliko pokazivač ukazuje na neku promenljivu, posredno se menja i njen sadržaj. Na primer, nakon dodela

```
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 `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`.

Još jednom naglasimo da pokazivački i celobrojni tipovi 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 ovo treba oprezno koristiti. Jedini izuzetak je `0` koja se može tumačiti i kao ceo broj i kao pokazivač. Tako je moguće dodeliti nulu pokazivačkoj promenljivoj i porediti pokazivač sa nulom. Simbolička konstanta `NULL` se često koristi umesto nule, kao jasniji indikator da je u pitanju specijalna pokazivačka vrednost. `NULL` je definisana u zaglavlju `<stdio.h>` i ubuduće ćemo uvek koristiti `NULL` kao vrednost za pokazivač koji ne pokazuje ni na šta smisleno. Pokazivač koji ima vrednost `NULL` nije moguće dereferencirati. Pokušaj dereferenciranja dovodi do greške tokom izvršavanja programa (najčešće “segmentation fault”).

Generički pokazivački tip (void*). 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 (čak se u C-u, za razliku od C++-a, vrši i implicitna konverzija prilikom dodele). Međutim, na-

[2] Simbol `*` koristi i za označavanje pokazivačkih tipova i za operator dereferenciranja i poželjno je jasno razlikovanje ove njegove dve različite uloge.

ravno, nije moguće vršiti dereferenciranje pokazivača tipa `void*` jer nije moguće odrediti tip takvog izraza kao ni broj bajtova u memoriji koji predstavljaju njegovu vrednost.

15.2 Pokazivači i argumenti funkcija

U jeziku C argumenti funkcija (osim u slučaju nizova[3]) se uvek prenose po vrednosti. To znači da promenljiva koja je upotrebljena kao argument funkcije ostaje nepromenjena nakon poziva funkcije. Na primer, ako je data sledeća funkcija

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}
```

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, 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 poslati 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 `pa` i `pb` (pokazivačkog tipa) i zatim se vrednosti `&x` i `&y` *kopiraju* u taj prostor. Primetimo da se time ponovo vrši prenos argumenata *po vrednosti*. Funkcija se izvršava koristeći samo kopije pokazivača, ali zahvaljujući i njima ona zaista pristupa promenljivama `x` i `y` i zamenjuje im vrednosti. Nakon povratka iz funkcije, vrednosti `x` i `y` ostaju zamenjene.

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 ukratko opisana.

Prenos pokazivača može da se iskoristi i da funkcija vrati više različitih vrednosti. Na primer, naredna funkcija računa količnik i ostatak pri deljenju

[3] Umesto da se kao argumenti prenosi čitav niz, prenosi se samo adresa njegovog početka (kao i uvek, po vrednosti).

dva broja, pri tom ne koristeći operatore / i %.

```
#include <stdio.h>

void deljenje(unsigned a, unsigned b,
              unsigned* pk, unsigned* po) {
  *pk = 0; *po = a;
  while (*po >= b) {
    (*pk)++;
    *po -= b;
  }
}

int main() {
  unsigned k, o;
  deljenje(14, 3, &k, &o);
  printf("%d %d\n", k, o);
  return 0;
}
```

15.3 Pokazivači i nizovi

Postoji čvrsta veza pokazivača i nizova. Operacije nad nizovima mogu se iskazati i korišćenjem pokazivača i sve češće nećemo praviti razliku između dva načina za pristupanje elementima niza.

Deklaracije `int a[10]` deklariše niz veličine 10. Početni element je `a[0]`, a deseti element je `a[9]` i oni su poređani uzastopno u memoriji. Imenu niza `a` pridružena je informacija o adresi početnog elementa niza, o tipu elemenata niza, kao i o broju elemenata niza. Ime niza `a` nije l-vrednost (jer `a` uvek ukazuje na isti prostor koji je rezervisan za elemente niza). Dakle, vrednost `a` nije pokazivačkog tipa, ali mu je vrlo bliska.

Nakon deklaracije `int a[10]`, 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`. 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 drugih promenljivih koje se nalaze na tim lokacijama ili da dovede do prekida rada programa zbog pristupanja memoriji koja mu nije dodeljena.

Kao što je rečeno u poglavlju 9.5, 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 (tj. ne prenosi se nijedan element niza). Funkcija koja prihvata takav argument za njegov tip ima pokazivač zapisan u formi `char *a` ili `char a[]`. Ovim se kao argument prenosi (kao i uvek — po vrednosti) samo pokazivač na početka niza, ali ne i informacija o dužini niza. Na primer, kôd dat u narednom primeru (na mašini na kojoj su tip

`int` i adresni tip reprezentovani sa 4 bajta) ispisuje, u okviru funkcije `main`, broj 20 (5 elemenata tipa `int` čija je veličina 4 bajta) i, u okviru funkcije `f`, broj 4 (veličina adrese koja je preneta u funkciju). Dakle, funkcija `f` nema informaciju o broju elemenata niza `a`.

```
void f(int a[]) {
  printf("%d\n", sizeof(a));
}

int main() {
  int a[] = {1, 2, 3, 4, 5};
  printf("%d\n", sizeof(a));
  f(a);
  return 0;
}
```

Dakle, prilikom prenosa niza u funkciju, uz njegovo ime (tj. adresu njegovog početka), najčešće je neophodno proslediti i broj elemenata niza, jer je ovaj broj nemoguće odrediti u okviru funkcije samo na osnovu prenete adrese početka. Izuzetak od ovog pravila predstavljaju funkcije koje obrađuju prosleđene niske karaktera jer je u tom slučaju na osnovu sadržaja niske moguće odrediti i njegov broj elemenata[4]. U narednom poglavlju biće razmotrene neke takve funkcije.

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 ime niza `a` i ako funkcija `f` ima prototip `f(int x[]);` (ili — ekvivalentno — `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)`). Ni u jednom od ovih slučajeva funkcija `f`, naravno, nema informaciju o tome koliko elemenata niza ima nakon prosleđene adrese.

15.4 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, izrazu `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 — onoliko koliko bajtova na tom sistemu zauzima podatak tipa `int`. Tako, ako je `p` pokazivač na `int` koji sadrži na adresu 100, na mašini na kojoj `int` zauzima 4 bajta, vrednost `p+3` će biti adresa $100 + 3 \cdot 4 = 112$.

[4] Preciznije, i dalje nije moguće odrediti veličinu niza, već je samo moguće odrediti njegov efektivni sadržaj do pojave terminalne nule.

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.

Pored dereferenciranja, dodavanja i oduzimanja celih brojeva, nad pokazivačima je moguće izvoditi još neke operacije.

Pokazivači se mogu porediti relacijskim operatorima (na primer, `p1 < p2`, `p1 == p2`, ...). Ovo ima smisla ukoliko pokazivači ukazuju na elemente istog niza. Tako je, na primer, `p1 < p2` tačno akko `p1` ukazuje na raniji element niza od pokazivača `p2`.

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.

Unarni operatori `&` i `*` imaju viši prioritet nego binarni aritmetiči 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 s leva na desno, imaju viši prioritet od unarnih operatora koji se primenjuju s desna nalevo, tako da `*p++` vraća sadržaj na lokaciji `p`, dok se kao bočni efekat `p` inkrementira.

Ilustrujmo sve ovo primerom strukture podataka poznate kao *stek (eng. stack)*. Stek je predstavljen nizom elemenata i elementi se dodaju i uzimaju sa istog kraja.

```
#define MAX_SIZE 1024
int a[MAX_SIZE];
int *top = a;

int push(int x) {
  if (top < a + SIZE - 1)
     *top++ = x;
  else {
    printf("Greska"); exit(EXIT_FAILURE);
  }
}

int pop() {
  if (top > a)
     return *--top;
  else {
     printf("Greska"); exit(EXIT_FAILURE);
  }
}

int size() { return top - a; }
```

Prilikom dodavanja elementa vrši se provera da li je stek možda pun i to izrazom `top < a + SIZE - 1` koji uključuje pokazivačku aritmetku sabiranje pokazivača (preciznije adrese početka niza) i broja, i zatim oduzimanje broja od dobijenog pokazivača. Slično, prilikom uklanjanja elementa, vrši se da li je stek prazan i to izrazom `top > a` koji uključuje poređenje dva pokazivača (preciznije, pokazivača i niza). Napokon, funkcija koja vraća broj elemenata postavljenih na stek uključuje oduzimanje dva pokazivača (preciznije, pokazivača i niza). Potrebno je naglasiti i suptilnu upotrebu prefiksnog operatora dekrementiranja u funkciji `pop`, odnosno postfiksnog operatora inkrementiranja u funkciji `push`, kao i njihov odnos sa operatorom dereferenciranja.

15.5 Karakterski pokazivači i funkcije

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. Funkcije kao `printf` umesto niske zapravo kao argument prihvataju adresu njenog početka. Konstantne niske se u memoriji uvek čuvaju u segmentu podataka.

Ukoliko promenljiva `p` ukazuje na početak niske `"informatika"`, onda je `p[0]` jednako `'i'`, `p[1]` je jednako `'n'` i tako dalje. Na primer, to važi nakon sledeće deklaracije sa inicijalizacijom: `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: promenljiva `p` će se čuvati u segmentu podataka i ukazivati na nisku `"informatika"` — koja se čuva negde drugde u segmentu podataka. I u jednom i u drugom slučaju, pokušaj da se promeni sadržaj lokacije na koji ukazuje `p` (na primer, `p[0]='x';` ili `p[1]='x';`) dovodi do nedefinisanog ponašanja (najčešće greške u fazi izvršavanja). S druge strane, promena vrednosti samog pokazivača `p` (na primer, `p++;`) je u oba slučaja dozvoljena.

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";` se kreira niz `a` dužine `12` i prilikom inicijalizacije se on popunjava karakterima niske `"informatika"`. Sada je dozvoljeno menjanje vrednosti `a[0]`, `a[1]`, ..., `a[9]`, ali nije dozvoljeno menjanje vrednosti `a`.

Razmotrimo funkciju — `strlen` koja vraća dužinu zadate niske. U ovom slučaju, njen argument `s` je pokazivač na početak niske (koja se, kao i obično, može tretirati kao niz). Ukoliko se funkcija pozove kao `strlen(p)`, pri čemu je `p` bilo pokazivač bilo niz koji predstavlja neku nisku, promenljiva `s` nakon poziva zapravo predstavlja kopiju tog pokazivača `p` i može se menjati bez uticaja na originalni pokazivač koji je prosleđen kao argument.

```
int strlen(char *s) {
   int n;
   for (n = 0; *s != '\0'; s++)
     n++;
   return n;
}
```

Još jedna verzija ove funkcije koristi mogućnost oduzimanja pokazivača:

```
int strlen(char *s) {
   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 funkciju `strcpy` koja prihvata dva karakterska pokazivača (ili dve niske) i sadržaj druge kopira u prvu.[5] Jedna verzija ove funkcije je data u prethodnom tekstu, sada navodimo nove verzije.

[5]Da bi jedna niska bila iskopirana u drugu nije dovoljno prekopirati pokazivač, već je neophodno prekopirati svaki karakter druge niske pojedinačno.

```
void strcpy(char *s, char *t) {
  while ((*s = *t) != '\0') {
     s++;
     t++;
  }
}
```

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. U navedenom kodu, oba pokazivača se povećavaju za po jedan (operatorom inkrementiranja), sve dok drugi pokazivač ne dođe do završne nule, tj. do kraja niske. Ovo inkrementiranje može se izvršiti (u postfiksnom obliku) i u okviru same `while` petlje:

```
void strcpy(char *s, 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, char *t) {
  while (*s++ = *t++);
}
```

15.6 Pokazivači na funkcije

Iako funkcije u C-u nisu ,,građani prvog reda" (engl. first class citizen) tj. ne mogu se direktno prosleđivati kao argumenti drugim funkcijama, vraćati kao rezultat funkcija i ne mogu se dodeljivati promenljivima, ove operacije je moguće posredno izvršiti ukoliko se koriste pokazivači na funkcije. Razmotrimo nekoliko ilustrativnih primera.

```
#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 kodu kopiraju elemente niza `a` u niz `b` prethodno ih transformišući na neki način. Poželjno 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`.

Kao što je i kôd običnih pokazivača striktno poštovan tip na koji se pokazuje, i pokazivači na funkcije se razlikuju u zavisnosti od tipa funkcije (tipa argumenata i tipa povratne vrednosti na koje ukazuju). Deklaracija promenljive tipa pokazivača na funkciju se vrši tako što se ime promenljive kome 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. Na primer,

```
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 (`*`). Iako se nekada oznake ovih operacija mogu izostaviti i koristiti samo ime funkcije, odnosno pokazivača (na primer, u prethodnom programu je bilo moguće navesti `map(a, N, b, inc1)`, odnosno `b[i] = f(a[i])`), ovo se ne preporučuje zbog portabilnosti programa.

Moguće je kreirati i nizove pokazivača na funkcije. Ovi nizovi se mogu i incijalizovati (na uobičajeni način). Na primer,

```
int (*fje[3]) (int) = {&inc1, &mul2, &parni0};
```

`fje` predstavlja niz od 10 pokazivača na funkcije koje vraćaju `int`, i primaju argument tipa `int`. Funkcije čije se adrese nalaze u nizu se mogu direktno i pozvati, te naredni primer ispisuje `4`.

```
printf("%d", (*fje[0])(3));
```

Glava 16

Višedimenzionalni nizovi i nizovi pokazivača

16.1 Višedimenzionalni nizovi

Jezik C dozvoljava definisanje klasičnih višedimenzionalnih nizova, iako se u praksi oni koriste mnogo reće nego nizovi pokazivača.

Višedimenzionalni nizovi u programskom jeziku C se mogu deklarisati kao:

```
tip ime-niza[broj-elemenata][broj-elemenata]...[broj-elemenata];
```

Dvodimenzionalni nizovi se tumače kao jednodimenzioni nizovi čiji su elementi nizovi. Zato se elementima pristupa sa:

```
ime-niza[vrsta][kolona]
```

a ne sa `ime-niza[i,j]`.

Elementi se u memoriji smeštaju po vrstama tako da se najdešnji indeks najbrže menja kada se elementima pristupa u redosledu po kojem su u memoriji smešteni. Niz se može inicijalizovati navođenjem liste inicijalizatora u vitičastim zagradama; pošto se elementi opet nizovi, svaki od njih se opet navodi u okviru vitičastih zagrada.

Na primer, razmotrimo dvodimenzionalni 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[2][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}
};
```

Primetimo da se tip `char` koristi za skladištenje malih prirodnih brojeva. Takođe, pošto zauzeće prostora nije presudno, u nultu kolonu su upisane nule kako bi se podaci za mesec `m` nalazili upravo u koloni `m` (tj. kako bi se mesecima pristupalo sa 1-12, umesto 0-11). Dakle, niz `broj_dana` je moguće koristiti, na

primer, na sledeći način:

```
char broj_dana_u_mesecu(unsigned godina, char mesec) {
  int prestupna = (godina % 4 == 0 && godina % 100 != 0) ||
                  godina % 400 == 0;
  return broj_dana[prestupna][mesec];
}
```

Podsetimo se da je vrednost logičkog izraza u C-u uvek nula (netačno) ili jedan (tačno), tako da se ova vrednost može koristi kao indeks pri pristupanju nizu.

Ukoliko se dvodimenzioni niz prenosi u funkciju, 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. Na primer, funkcija `f` koja prima niz `broj_dana` može biti definisana na jedan od tri naredna načina.

```
void f(char broj_dana[2][13]);
void f(char broj_dana[][13]);
void f(char (*broj_dana)[13]);
```

U poslednjem slučaju, eksplicitno je rečeno da se prenosi pokazivač na niz od 13 karaktera. Zagrade `()` su neophodne jer zagrade `[]` imaju viši prioritet nego `*`. Bez zagrada, deklaracija

```
void f(char *broj_dana[13]);
```

označava da se prenosi niz od 13 pokazivača na karaktere.

U slučaju dimenzija većih od 2, samo je prva dimenzija (indeks) slobodan, dok je sve naredne dimenzije neophodno navesti.

Kao što smo ranije naučili, ime jednodimenzionog niza (na primer, `a` za `int a[10]`) može se tretirati kao pokazivač na prvi element niza. Ime dvodimenzionog niza, na primer, tipa `int` može se tretirati kao pokazivač na pokazivač na `int`. Na primer, ako je

```
int d[10][20];
```

onda je `d[0]` (kao i `d[1]`, `d[2]`, ...) pokazivač na `int`. Pokazivač `d[0]` sadrži adresu elementa `d[0][0]`, i, opštije, `d[i]` sadrži adresu elementa `d[i][0]`. Vrednost `d` je tipa `int **`, ona je pokazivač na pokazivač na `int` i sadrži adresu pokazivača `d[0]`.

16.2 Odnos višedimenzionalnih nizova i nizova pokazivača

Razjasnimo detaljnije 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 sintaksno ispravna referisanja na pojedinačni `int`. Ali `a` je pravi dvodimenzioni niz: 200 ćelija veličine `int`-a su rezervisane i uobičajena računica `20 * v + k` se koristi da bi se pronašao element `a[v][k]`. Za niz `b`, međutim, definicija samo alocira 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 smešteno 200 ćelija veličine `int`-a i još dodatno 10 ćelija 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 sadrži imena meseci. Ukoliko bi sva imena bila iste dužine, rešenje bi mogao biti dvodimenzioni niz, međutim, pošto svi 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.

```
char *meseci[] = {
           "Greska",
           "Januar", "Februar", "Mart",
           "April", "Maj", "Jun",
           "Jul", "Avgust", "Septembar",
           "Oktobar", "Novembar", "Decembar"};
```

Primetimo da nije bilo neophodno navesti broj elemenata niza pošto je izvršena inicijalizacija.

Uporedite to sa dvodimenzionalnim nizom

```
char meseci[][9] = {
           "Greska",
           "Januar", "Februar", "Mart",
           "April", "Maj", "Jun",
           "Jul", "Avgust", "Septembar",
           "Oktobar", "Novembar", "Decembar"};
```

Glava 17

Korisnički definisani tipovi

17.1 Strukture

Osnovni C tipovi često nisu dovoljni da opišu podatke. Ukoliko je neki podatak složene prirode, njegovi pojedinačni podaci mogu se čuvati nezavisno, ali to vodi nejasnim programima, teškim za održavanje. Umesto toga, pogodnije je koristiti *strukture*. Struktura objedinjuje jednu ili više promenljivih, ne nužno istih tipova. Defisanjem 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.

17.1.1 Osnovne osobine struktura

Razlomak je opisan parom koji čine imenilac i brojilac, recimo celobrojnog tipa. U jeziku C ne postoji tip koji opisuje razlomke, ali moguće je definisati strukturu koja opisuje razlomke. Imenilac (svakog) razlomka zvaće se `imenilac` a brojilac (svakog) razlomka zvaće se `brojilac`. Struktura `razlomak` se može definisati na sledeći način:

```
struct razlomak {
    int imenilac;
    int brojilac;
};
```

Ključna reč `struct` označava deklaraciju strukture. Nakon nje navodi se ime strukture, a zatim opis njenih članova. Imena članova strukture se ne vide kao samostalne promenljive, one postoje samo kao deo složenijeg objekta.

Deklaracija strukture uvodi novi tip i nakon nje se ovaj tip može koristiti kao i bilo koji drugi, pri čemu je ponovo neophodno korišćenje ključne reči `struct` na primer:

```
struct razlomak a, b, c;
```

Prvom deklaracijom je, dakle, opisano da se razlomci sastoje od brojioca i imenioca, dok se drugom deklaracijom uvode tri 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, ovom deklaracijom je uveden razlomak `a` čiji je brojilac 1, a imenilac 2.

Definisanje strukture i deklarisanje i inicijalizacija odgovarajućih promenljivih se može uraditi istovremeno (otuda i neuobičajen simbol `;` nakon zatvorene vitičaste zagrade prilikom definicije 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 sledi simbol `.` 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).

Dalje, moguće je definisati i pokazivače na strukture. U slučaju da se članovima strukture pristupa preko pokazivača, umesto kombinacije operatora `*` i `.` (tj. umesto `(*pa).imenilac`), moguće je koristiti operator `->`. I ovaj operator je operator najvišeg prioriteta.

```
struct razlomak *pa = &a;
printf("%d/%d", pa->brojilac, pa->imenilac);
```

Deklaracije struktura mogu biti ugnježdene. Na primer:

```
struct dvojni_razlomak {
    struct razlomak gore;
    struct razlomak dole;
};
```

Nad promenljivama tipa strukture dozvoljene su operacije dodele, referenciranja i pristupanja članovima a nisu dozvoljeni aritmetički i relacioni operatori.

17.1.2 Strukture i funkcije

Strukture mogu biti i argumenti i povratne vrednosti funkcija.

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 argumenata i istoimenih članova strukture. Naime, imena članova strukture su uvek vezana za ime promenljive (u ovom primeru `rezultat`).

Sledeći primer ilustruje funkcije sa argumentima 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;
}
```

Naglasimo da se i strukture u funkciju prenose po vrednosti. S obzirom 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;
}
```

Sledeća funkcija poredi dva razlomka i vraća 1 ako je prvi veći, 0 ako su jednaki i -1 ako je prvi manji:

```
int poredi_razlomke(struct razlomak a, struct razlomak b)
{
    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;
    }
}
```

Na sličan način mogu se implentirati i druge operacije nad razlomcima, kao množenje razlomaka, skraćivanje razlomka, itd.

17.1.3 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;
    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):

```
struct opis_meseca meseci[] = {
  { "",0 },
  { "januar",31 },
  { "februar",28 },
  { "mart",31 },
  ...
  { "decembar",31 }
}
```

(U ovom primeru se zanemaruje da februar može imati i 29 dana.)

U navednoj inicijalizaciji unutrašnje vitičaste zagrade je moguće izostaviti:

```
struct opis_meseca meseci[] = {
  "",0,
  "januar",31,
  "februar",28,
  "mart",31,
  ...
  "decembar",31
}
```

Kao i obično, broj elemenata ovako inicijalizovanog niza može se izračunati na sledeći način:

```
sizeof(meseci)/sizeof(struct opis_meseca)
```

Nakon ovakve deklaracije, ime prvog meseca se može dobiti sa `meseci[1].ime`, njegov broj dana sa `meseci[1].broj_dana`. Kao i obično, ukoliko je niz `meseci` automatska promenljiva onda se njeni elementi čuvaju u steku segmentu memorije, a ukoliko je statičkog životnog veka, onda se njeni elementi čuvaju u segmentu podataka. U oba slučaja pokazivači `ime` ukazuju na niske koje se nalaze u segmentu podataka.

17.2 Unije

17.3 Nabrojivi tipovi (`enum`)

17.4 `Typedef`

U jeziku C moguće je kreirati nova imena tipova koristeći direktivu `typedef`.

Na primer, deklaracija

```
typedef int Length;
```

uvodi ime `Length` kao sinonim za tip `int`. Ime tipa `Length` se nadalje može koristiti u deklaracijama, eksplicitnim konverzijama i slično, na isti način kao što se može koristiti i `int`.

```
Length len, maxlen;
Length* lengths[];
```

Slično, deklaracija

```
typedef char* String;
```

uvodi ime tipa `String` kao sinomim za tip `char *`.

```
String p, lineptr[MAXLINES], alloc(int);
int strcmp(String, String);
```

Primetimo da se novo ime tipa koje se uvodi navodi kao poslednje, na poziciji na kojoj se u deklaracijama obično navodi ime promenljive, a ne neposredno nakon ključne reči `typedef`. Običaj je da se novo uvedena imena tipova pišu velikim slovima kako bi se istakla.

Veoma često korišćenje `typedef` deklaracija je u kombinaciji sa strukturama kako bi se izbegla potreba za pisanje ključne reči `struct` pri svakom korišćenju imenu strukturnog tipa. Na primer:

```
typedef struct point Point;
struct point {
  int x, y;
};

Point a, b;
```

Definicija strukture i `typedef` se mogu raditi istovremeno. Na primer:

```
typedef struct point {
  int x, y;
} Point;

Point a, b;
```

Ukoliko se očekuje da se strukturi isključivo pristupa preko pokazivača, obično se uvodi `typedef` za tip pokazivača na strukturu.

```
typedef struct tnode* Treeptr;
typedef struct tnode {
    int data;       /* data */
    Treeptr left;   /* left child */
    Treeptr right;  /* right child */
} Treenode;
```

Naglasimo da se `typedef` deklaracijom ne kreira novi tip već se samo uvodi novo ime za potojeći tip. Dakle, `typedef` je veoma slična direktivi `#define`, jedino što je deo kompilatora i može da da rezultat i u slučajevima u kojima jednostavne pretprocesorske tekstualne zamene 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`" i koje se može koristiti na primer kao:

```
PFI strcmp, numcmp;
```

Osim čisto estetskih pitanja, postoje dva glavna razloga za korišćenje `typedef`. Prvo, je parametrizovanje tipova u programu kako bi se dobilo na njegovoj pre-

nosivosti. 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. Tipovi `size_t` i `ptrdiff_t` iz standardne biblioteke takođe spadaju u ovu grupu primera.

Druga svrha upotrebe `typedef` je poboljšanje čitljivosti programa - tip sa imenom `Treeptr` se može nekada jednostavnije razumeti nego kompleksna deklaracija pokazivača na strukturu.

Glava 18

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. U nekim slučajevima, moguće je unapred zadati gornje ograničenje, međutim 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 dolazi u vidu *dinamičke alokacije memorije* koja omogućava da program u toku svog rada (od operativnog sistema) zahteva određenu količinu memorije. U trenutku kada mu memorija koja je dinamički alocirana više nije potrebna, program je dužan da je oslobodi i tako je vrati operativnom sistemu na upravljanje.

18.1 Funkcije C 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>`).

Alociranje i oslobađanje. Funkcija

```
void *malloc(size_t n);
```

alocira blok memorije veličine `n` bajtova i vraća adresu alociranog bloka memorije (niza uzastopnih bajtova) u vidu generičkog pokazivača (tipa `void*`). U slučaju da zahtev za memorijom u nekom slučaju nije moguće ispuniti (npr. zahteva se više memorije nego što je na raspolaganju), ova funkcija vraća `NULL`. Memorija koju funkcija `malloc` vrati nije inicijalizovana i njen sadržaj je, u prin-

cipu, slučajan (preciznije rečeno, zavisi od podataka koji su ranije bili čuvani u tom delu memorije).

Funkcija

```
void *calloc(size_t n, size_t size)
```

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`, memorija je inicijalizovana na nulu.

Kako bi se dobijenoj memoriji moglo pristupiti, potrebno je (poželjno eksplicitno) konvertovati dobijeni pokazivač tipa `void*` u neki konkretni pokazivački tip.

Nakon poziva funkcije `malloc()` ili `calloc()` poželjno je 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, u nastavku programa se `p` može koristiti kao (statički alociran) niz celih brojeva.

U trenutku kada ovaj „niz“ (tj. dinamički alociran blok memorije) više nije potreban, poželjno je osloboditi ga. To se postiže funkcijom:

```
void free(void* p);
```

Poziv `free(p)` oslobađa memoriju na koju ukazuje pokazivač `p` (a ne memoriju koja 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 alokaciranja.

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.

```
#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;
}
```

Realociranje. U nekim slučajevima potrebno je promeniti veličinu već alociranog bloka memorije. To se postiže korišćenjem funkcije

```
void *realloc(void *memblock, size_t size);
```

Parametar `memblock` je pokazivač na prethodno alociran blok memorije, a parametar `size` je nova veličina u bajtovima. Funkcija `realloc` vraća pokazivač tipa `void*` na realociran blok memorije ili 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 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 duplo uveća).

```
#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;
}
```

Naravno, 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.

18.2 Greške u radu sa dinamičkom memorijom

Curenje memorije (memory leaking). Jedna od najopasnijih grešaka u radu sa dinamički alociranom memorijom je tzv. *curenje memorije (eng. memory leaking)*. 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 kraja izvršavanja programa) izgubljen (rezervisan za korišćenje od strane programa koji više nema načina da mu pristupi). Na 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 iterativnog toka programa jer time velika količina memorije može da bude izgubljena. Uko-

liko se ne obraća pažnja na pojavu curenja memorije, moguće je da će u nekom trenutku program iscrpeti raspoloživu memoriju i biti prekinut od strane operativnog sistema. Čak i da se to ne desi, moguće je da se iscrpi fizički 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 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 (tzv. dibagera) detektuje da u programu postoji curenje memorije, ali ne može da pomogne u lociranju odgovarajuće greške u kodu. Postoje specijalizovani programi (*memory leaks profilers*) koji olakšavaju otkrivanje uzroka curenja memorije.

Pristup oslobođenoj memoriji. Jednom kada se pozove `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, često se preporučuje 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. Kada se jednom pozove `free(p)`, svaki naredni poziv `free(p)` za istu vrednost pokazivača `p` prouzrokuje "nedefinisano ponašanje" programa i trebalo bi ga izbegavati. Takozvana *dupla oslobađanja* mogu da dovedu do pada programa i poznato je da su ponekad i izvor bezbednosnih problema.

Prosleđivanje pokazivača koji ne čuva adresu vraćenu od strane `malloc`, `calloc` ili `realloc`. 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 stvori ozbiljne probleme. Naravno, upis je često 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. Prekoračenje bloka prilikom upisa menja te dodatne informacije što, naravno, uzrokuje pad sistema za dinamičko upravljanje memorijom.

18.3 Fragmentisanje memorije

Čest je slučaj da ispravne aplikacije u kojima ne postoji curenje memorije (a koje često vrše dinamičku alokacije i dealokacije memorije) tokom dugog rada pokazuju degradaciju u performansama i na kraju umiru. Uzrok ovome je najčešće pojava poznata pod imenom *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, a u stanju je da izvrši alokaciju 5 traženih bajtova.

Pristupi izbegavanja fragmentacije memorije su sledeći. 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 memorij, poželjno je memoriju alocirati u većim blokovima i umesto alokacije jednog objekta alocirati prostor za nekoliko njih odjednom (kao što je to npr. bilo rađeno u primeru datom prilikom opisa funkcije `realloc`). Na kraju, postoje tehnike ,,ručnog" efikasnijeg rukovanja memorijom (npr. memory pooling).

18.4 Hip i dinamički životni vek

U poglavlju 13.1, rečeno je da je memorija dodeljena programu organizovana u segment koda, 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.

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 (eng. 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.

Sva memorija koja je dinamički alocirana ima *dinamički životni vek*. Ovo znači da se memorija i alocira i oslobađa isključivo na eksplicitni zahtev i to tokom rada programa.

Glava 19

Standardna biblioteka i ulaz/izlaz

S obzirom da jezik C teži da bude mali jezik, ulaz i izlaz nisu direktno podržani samim jezikom već se izvršavaju korišćenjem specijalizovanih funkcija. Ipak, s obzirom na značaj ulaza i izlaza, funkcije za ulaz i izlaz su prisutne u okviru standardne biblioteke jezika C i tako prisutne u svakom C okruženju. Pošto su ove funkcije deo standarda jezika C, mogu se slobodno koristiti u programima uz garantovanu prenosivost programa između različitih sistema. Svaka izvorna datoteka u kojoj se koriste ulazno/izlazne funkcije, trebalo bi da uključi standradno zaglavlje `<stdio.h>`.

19.1 Standardni ulaz, standardni izlaz i standardni izlaz za greške

Stanardna biblioteka implementira jednostavan model tekstualnog ulaza i izlaza. *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
```

dešava se da program `prog` čita karaktere iz datoteke `infile`.

Takođe, mnoga okruženja omogućavaju da se izvrši preusmeravanje (redirekcija) standardnog izlaza u neku datoteku. Na primer, ukoliko se program pokrene sa

```
./prog > outfile
```

dešava se da program `prog` piše karaktere u datoteku `outfile`.

Ukoliko bi se poruke o greškama štampale na standardni izlaz, zajedno sa ostalim rezultatima rada programa, postojala bi opasnost da se, u slučaju preusmeravanja standardnog izlaza u datoteku, poruke o greškama korisniku ne prikažu na ekranu, već da se smeste u datoteku i da ih korisnik ne vidi. Ovo je glavni razlog uvođenja standardnog izlaza za greške koji se, i u slučaju da se standardni izlaz preusmeri u datoteku, obično prikazuje na ekranu. Takođe, moguće je izvršiti i redirekciju standardnog izlaza za greške. Na primer:

```
./prog 2> errorfile
```

19.1.1 Ulaz i izlaz pojedinačnih karaktera

Najjednostavniji mehanizam čitanja sa standardnog ulaza je da se čita jedan po jedan karakter 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 najšće -1, ali testovi bi trebalo da se vrše u odnosu na simbol `EOF`, nezavisno od specifične vrednosti na koju je ova konstanta definisana. Pomenimo da se funkcija `getchar()` najčešće realizuje tako što karaktera uzima iz privremenog bafera koji se puni čitanjem jedne po jedne linije ulaza. Dakle, pri interaktivnom radu sa programom, `getchar()` neće imati efekta dok se ne unese prelazak u novi red ili oznaka za kraj datoteke.

Najjednostavniji mehanizam pisanja na standardni izlaz je da se piše jedan po jedan po jedan karakter korišćenjem funkcije `putchar`:

```
int putchar(int);
```

Funkcija `putchar(c)` štampa karakter `c` na standardni izlaz, najčešće ekran, a vraća karakter koji je ispisala, odnosno `EOF` ukoliko dođe do greške.

Kao primer rada sa pojedinačnim karakterima, razmotrimo program koji prepisuje standardni ulaz na standardni izlaz, pretvarajući pri tom velika u mala slova.

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

int main()
{
    int c;
    while ((c = getchar()) != EOF)
        putchar(tolower(c));
    return 0;
}
```

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, čime se izbegava dodatno vreme i prostor potreban za realizaciju funkcijskog poziva.

19.1.2 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 terminira nulom. Vašno je obratiti pažnju da se ne vrši nikakva provera da li niska `s` sadrži dovoljno prostora da prihvati pročitani sadržaj. 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);
```

piše nisku na koju ukazuje `s` na standardni izlaz, dodajući automatski pri tom oznaku za kraj reda. Terminirajući karakter `'\0'` se ne zapisuje. Funkcija puts vraća `EOF` u slučaju da je došlo do greške, a nenegativnu vrednost inače.

19.1.3 Formatirani izlaz — `printf`

Funkcija `printf` je već korišćena u prethodnim poglavljima. Izlaz koji kreira ova funkcija takođe se štampa na standardni izlaz. Pozivi funkcija `putchar` i `printf` mogu biti isprepletani — izlaz se vrši u istom redosledu u kojem su ove funkcije pozvane.

Funkcija `printf` prevodi vrednosti osnovnih tipova podataka u serije karaktera i tako ih štampa na standardni izlaz. Slučajevi korišćenja ove funkcije iz prethodnih poglavlja jesu najuobičajeniji, ali svakako nisu potpuni.

```
int printf(char *format, arg1, arg2, ...);
```

Funkcija `printf` prevodi, formatira i štampa njene argumente na na standardni izlaz pod kontrolom date format niske. Funkcija vraća broj odštampanih karaktera. Format niska sadrži dve vrste objekata: obične karaktere, koji se bukvalno prepisuju u izlazni tok, 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.
- Broj koji specifikuje najmanju širinu polja. Konvertovani argument se štampa u polju koje je bar 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, 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 argumenta; Štampa se kao
d,i	`int`; dekadni broj
o	`int`; neoznačeni oktalni broj (bez vodeće 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 %

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);
```

Treba imati u vidu da se prilikom poziva funkcije `printf` na osnovu prvog argumenta (format niske) određuje 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 % */
   printf("%s", s);    /* bezbedno */
```

Primeri

Prikaz celih brojeva.

```
int    count = -9234;
printf("Celobrojni formati:\n"
       "\tDekadni: %d  Poravnat: %.6d  Neozna\v cen: %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  Neozna\v cen: 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
```

Prikaz karaktera.

```
   char  ch = 'h';
   printf("Karakteri u polju date sirine:\n%10c%c\n", ch, ch);
```

```
Karakteri u polju date sirine:
         hh
```

Prikaz realnih brojeva.

```
double fp = 251.7366;
printf( "Realni brojevi:\n\t%f %.2f %e %E\n", fp, fp, fp, fp);
```

```
Realni brojevi:
   251.736600 251.74 2.517366e+002 2.517366E+002
```

Prikaz 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     :
```

19.1.4 Formatirani ulaz — `scanf`

Funkcija `scanf` je ulazni analogon funkcije `printf`, obezbeđujući mnoge konverzija u suprotnom smeru.

```
int scanf(char *format, ...)
```

Funkcija `scanf` čita karaktere sa standardnog ulaza, interpretira ih na osnovu specifikacije navedene format niskom i smešta rezultat na mesta određena ostalim argumentima.

Opis formata će biti objašnjen u nastavku. Ostali argumenti moraju biti pokazivači i određuju adrese na koje se smešta konvertovani ulaz. Kao i u slučaju funkcije `printf`, u ovom poglavlju će biti dat samo prikaz najuobičajenijih 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 mustru navedenu format niskom. Funkcija vraća broj uspešno uklopljenih i dodeljenih izlaznih podataka. U slučaju kraja datoteke, funkcija vraća `EOF`. Primetimo da je ovo različito od 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 kome 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 ulaznog toka.
- Specifikacije konverzija, koje se sastoje od karaktera %, opcionog karaktera `*` koji sprečava dodeljivanje, opcionog broja kojim se navodi maksimalna širina polja, opcione `h`, `l` ili `L` kojima se navodi ako se očekuje `short` ili `long` podatak, 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 ulaznog toka jer se prelasci u novi red tretiraju kao beline[1].

- Konverzioni karakter upućuje na željenu interpretaciju ulaznog polja.

U narednoj tabeli navedeni su konverzioni karakteri.

Karakter	Ulazni podatak; Tip argumenta
d	dekadni ceo broj; int *
i	ceo broj; int *. Broj može biti i oktalan (ako ima vodeću 0) ili heksadekadan (vodeći 0x ili 0X).
o	oktalni ceo broj (sa ili bez vodeće nule); int *
u	neoznačeni dekadni broj; unsigned int *
x	heksadekadni broj (sa ili bez vodećih 0x ili 0X); int *
c	karakteri; char *. 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	niska (bez navodnika); char *, ukazuje na niz karaktera dovoljno dugačak za nisku uključujući i terminalnu `'\0'` koja se automatski dopisuje.
e,f,g	broj u pokretnom zarezu sa opcionim znakom, opcionom decimalnom tačkom i opcionim eksponentom; float *
%	doslovno %; ne vrši se dodela.

Ispred karaktera konverzije `d`, `i`, `o`, `u`, i `x` moguće je navesti karakter `h` koji ukazuje da se u listi argumenata očekuje pokazivač na `short` ili karakter `l` koji ukazuje da se u listi argumenata očekuje pokazivač na `long`.

Još jednom naglasimo: argumenti funkcije scanf moraju biti pokazivači. Ubedljivo najčešći oblik pogrešnog korišćenja ove funkcije je:

```
scanf("%d", n);
```

umesto

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

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`.

[1] Pod belinama se podrazumevaju razmaci, tabulatori, prelasci u novi redovi (carriage return i/ili line feed), vertikalni tabulatori i formfeed.

Primeri

U narednom primeru, korišćenjem funkcije `scanf` implementiran je jednostavni kalkulator:

```
#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 imena mesecan nema potrebe navesti `&` jer je ime niza vrsta pokazivača. 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);
```

19.2 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, char *format, arg1, arg2, ...);
```

`sprintf` formatira argumente arg1, arg2, etc., na osnovu formatirajuće 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(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 */
}
```

19.3 Ulaz iz datoteka i izlaz u datoteke

Dosada obrađene 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 školjke operativnog sistema. Napomenimo još jednom da se sve potrebne deklaracije za rad sa datotekama nalaze u zaglavlju `<stdio.h>`.

19.3.1 Tekstualne i binarne datoteke

Iako se svaka datoteka može posmatrati 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 koje 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 piše linija po linija. Kao što se vidi, 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 nativnog 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` bi će 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 osim vidljivih karaktera, oznake kraja reda i tabulatora i slično.

Drugi način je prilagođen 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, C jezik 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.

Kao što za tekstualne datoteke postoje specifičnosti za čitanje i pisanje, tako postoje i neke specifičnosti za binarne datoteke koje ne ćemo navoditi. Ovo nije specifično za programski jezik C već je to slučaj i sa drugim programskim jezicima a sve to proističe iz specifičnosti operativnih sistema kao i potrebe za specifičnim obradama razlicitih tipova datoteka. Naglasimo da je razlika između tekstualnih i binarnih datoteka još oštrija u jezicima koji podržavaju UNICODE jer se u tom slučaju više bajtova čita i konvertuje u jedan karakter.

19.3.2 Pristupanje datoteci

Kako bi se pristupilo datoteci, bili za čitanje, bilo za pisanje, potrebno je izvršiti određenu vrstu povezivanja datoteke i programa. Za ovo se koristi bibliotečka funkcija

```
FILE* fopen(char* filename, 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) i 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 pozicija u okviru bafera, informaciju o tome da li se došlo do kraja datoteke, da li je došlo do greške i slično. Korisnici ne moraju i ne bi trebalo da direktno koriste ove informacije, već je dovoljno da upamte pokazivač na strukturu `FILE` i da ga prosleđuju svim funkcijama koje bi trebalo da pristupaju datoteci. Primetimo da je `FILE` ime tipa, a ne ime strukture; ovo ime je uvedeno korišćenjem `typedef`.

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, ona se 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 `fopen` vraća NULL. Modovi `r+`, `w+` i `a+` ukazuju da će rad sa datotekom podrazumevati i čitanje i pisanje (ili nadopisivanje). S obzirom da se pravi razlika između tekstualnih i binarnih datoteka. 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 datoteke i obezbeđuje pokazivače na njih. Preko ovih datoteka se može pristupati standardnom ulazu, standardnom izlazu i standardnom izlazu za greške. Odgovarajući pokazivači se nazivaju:

```
FILE* stdin;
FILE* stdout;
FILE* stderr;
```

Funkcija

```
int fclose(FILE *fp)
```

je inverzna funkciji `fopen`, i prekida vezu između programa i datoteke koju je funkcija `fopen` ostvarila. Pri pozivu 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 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.

19.3.3 Ulaz i izlaz pojedinačnih karaktera

Nakon što se otvori datoteka, iz nje se čita ili se u nju piše sadržaj. Razmotrimo prvo funkcije koje rade sa pojedinačnim karakterima.

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), tako da bi pažljivi programi trebalo da vrše 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.

```
#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;
}
```

Vraćanje karaktera u datoteku. 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. Naglasimo da, iako ova funkcije utiče na na naredne operacije čitanja datoteke, ona ne menja fizički sadržaj same datoteke (naime vraćeni karakteri se najčešće smeštaju u memorijski bafer, a ne u datoteku na disku, odakle se dalje čitaju). Funkcija vraća `EOF` ukoliko je došlo do greške, a vraćeni karakter inače.

Naredni primer čita karaktere iz datoteke dok god su u pitanju cifre i pri tom gradi dekadni ceo broj. Pošto poslednji karakter koji je pročitan nije cifra, on se vraća u datoteku 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);
}
```

19.3.4 Provera i grešaka i kraja datoteke

Funkcija `feof` vraća vrednost tačno (različito od nule) ukoliko se došlo do kraja date datoteke.

```
int feof(FILE *fp)
```

Funkcija `ferror` vraća vrednost tačno (različito od nule) ukoliko se došlo do kraja date datoteke.

```
int ferror(FILE *fp)
```

19.3.5 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`; najviše `maxline-1` karaktera će biti pročitano. Rezultujuća niska je terminisana sa `'\0'`. U normalnom toku izvršavanja funkcija `fgets` vraća liniju; u slučaju kraja datoteke ili greške vraća `NULL`.

Za ispis, funkcija `fputs` ispisuje nisku (koja ne mora da sadrži oznaku kraja reda) u datoteku:

```
int fputs(char *line, FILE *fp)
```

Ona vraća `EOF` u slučaju da dođe do greške, a neki nenegativan broj inače.

Primetimo, da, malo neobično, `gets` briše završni `'\n'`, a `puts` ga dodaje, različito od funkcija `fgets` i `fputs`.

Kako bi se pokazalo da implementacija bibliotečkih funkcija nije značajno različito od implementacije ostalih funkcija, prikazujemo implementacije funkcija `fgets` and `fputs`, u obliku doslovno preuzetom iz nekih realnih implementacija 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;
}
```

Iz neočiglednih razloga, standard propisuje različite povratne vrednosti za funkcije `ferror` i `fputs`.

Korišćenjem funkcije `fgets` jednostavno je napraviti funkciju koja čita liniju po liniju sa standardnog ulaza, vraćajući dužinu pročitane linije i nulu za kraj:

```
/* getline:  read a line, return length */
int getline(char *line, int max)
{
     if (fgets(line, max, stdin) == NULL)
        return 0;
     else
        return strlen(line);
}
```

19.3.6 Formatirani ulaz i izlaz

Za formatirani ulaz i izlaz mogu se koristiti funkcije `fscanf` `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. Formatirajuća niska je u ovom slučaju drugi argument.

```
int fscanf(FILE *fp, char *format, ...)
int fprintf(FILE *fp, char *format, ...)
```

19.3.7 Rad sa binarnim datotekama

C standardna biblioteka nudi funkcije za direktno čitanje i pisanje bajtova u binarne datoteke. Takođe, korisnicima su na raaspolaganju funkcije za pozicioniranje u okviru datoteka tako da se pristup binarnih datoteka može posmatrati i kao pristup nekoj vrsti memorije sa slučajnim 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);
```

Prvi argument je adresa memorije u 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.

Navedimo jedan primer koji ilustruje primenu ovih funkcija.

```
#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);
}
```

19.4 Argumenti komandne linije programa

Još jedan način da se određeni podaci proslede programu je 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 sam naziv programa) navedenih prilikom pokretanja programa. Drug argument (koji se obično naziva `argv`, od engleskog argument vector) je (pokazivač na) niz niski karaktera koje sadrže argumente — svaka niska direktno odgovara jednom arugmentu. Po dogovoru, `argv[0]` je ime koje je korišćeno za poziv programa tako da je `argc` uvek barem 1. Ako je `argc` tačno 1, nisu navođeni dodatni argumenti nakon imena programa. Dakle, `argv[0]` je ime programa, `argv[1]` do `argv[argc-1]` su argumenti, dok se na poziciji

`argv[argc]` nalazi `NULL` pokazivač. Naglasimo još jednom da su `argc` i `argv` proizvoljni identifikatori i da je funkcija `main` mogla biti deklarisana i kao:

```
int main (int br_argumenata, char* argumenti[]);
```

Navedimo sada jednostavan program koji štampa broj argumenata kao i sadržaj vektora argumenata.

```
#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 tom, moguće je iza jednog `-` navesti više karaktera koji određuju opcije. Na primer, pozivom:

```
./program -a -bcd 134 -ef zdravo
```

se programu zadaju opcije `a`, `b`, `c`, `d` `e` i `f` i argumenti `134` i `zdravo`.

Naredni program ispisuje sve opcije pronađene u komandnoj liniji:

```
#include <stdio.h>

int main(int argc, char* argv[])
{
    /* Za svaki argument komande linije, pocevsi od argv[1]
       (preskacemo ime programa) */
    int i;
    for (i = 1; i < argc; i++)
    {
        /* Ukoliko i-ti argument pocinje crticom */
        if (argv[i][0] == '-')
        {   /* Ispisujemo 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.

```
#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 god ne dodjemo do kraja tekuce niske */
     while (c=*++argv[0])
       printf("Prisutna opcija : %c\n",c);
  return 0;
}
```

Glava 20

Pregled standardne biblioteke

Standardnu C biblioteku čini skup datoteka zaglavlja i rutina koje realizuju neke uobičajene operacije kao što su ulaznu/izlazne operacije i rad sa niskama. Standardna C biblioteka nije stvarna biblioteka koja se koristi za povezivanje sa programima napisanom u jeziku C, ona pre predstavlja standard u koji mora da se uklopi svaki C prevodilac. Kako je implementirana standardna biblioteka zavisi od konkretnog sistema.

Standardna C biblioteka koju propisuje ISO standard jezika C sadrži dvadeseti i četiri datoteke zaglavlja koje se mogu uključivati u programe korišćenjem pretprocesorskih direktiva. Svaka datoteka zaglavlja sadrži deklaracije jedne ili više funkcija, definicije tipova i makroe. U nastavku je dat pregled nekoliko često korišćenih datoteka zaglavlja:

`<limits.h>` — sadrži opsege brojevnih tipova za konkretnu implementaciju;

`<stdio.h>` — sadrži deklaracije osnovnih ulazno/izlaznih funkcija;

`<stdlib.h>` — sadrži deklaracije operacija za konverziju, funkcije za generisanje pseudoslučanih brojeva, dinamičku alokaciju memorije, itd.

`<ctype.h>` — sadrži deklaracije funkcije za klasifikovanje i konvertovanje karaktera, nezavisno od karakterske tabele koja se koristi (ASCII ili neke druge);

`<math.h>` — sadrži deklaracije za često korišćene matematičke funkcije, kao i funkcije za sortiranje i pretraživanje;

`<string.h>` — sadrži deklaracije funkcija za obradu niski;

`<time.h>` — sadrži podršku za podatke o vremenu i datumu;

`<assert.h>` — sadrži makro `assert`.

U daljem tekstu će biti ukratko opisane neke od funkcija deklarisanih u ovim zaglavljima.

20.1 Zaglavlje `string.h`

```
size_t strlen  (char* str);

char*  strcat  (char* destination, char* source);
char*  strncat (char* destination, char* source, size_t num);
char*  strcpy  (char* destination, char* source);
char*  strncpy (char* destination, char* source, size_t num);

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);
```

20.2 Zaglavlje `stdlib.h`

`rand` Funkcija `rand()` 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. Prototip funkcije `rand` je:

```
int rand(void);
```

i ona vraća sledeći pseudo-slučajan broj u svom nizu. Pseudo-slučajni realni brojevi 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 `srand` 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. Prototip funkcije `srand` je:

```
void srand(unsigned int);
```

`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 startuje naredba 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 grešava). Vrednost -1 vraća se u slučaju greške (na primer, komandni interpretator nije rasplożiv, nema dovoljno memorije da se izvrši komanda, lista argumenata nije ispravna).

`exit` Prototip funkcije `exit` je:

```
void exit(int);
```

Funkcija `exit` 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. Poziv `exit(e)` u okviru funkcije `main` ekvivalentan je naredbi `return e`. Funkcija `exit` automatski poziva `fclose` za svaku datoteku otvorenu u okviru programa.

20.3 Zaglavlje `ctype.h`

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`;

20.4 Zaglavlje `math.h`

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` 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 interavala $(-\pi, \pi]$. Vrednost nije definisana za koordinatni početak.

`exp(x)` vraća vrednost e^x;

`log(x)` vraća vrednost $\ln x$ (mora da važi $x > 0$);

`log10(x)` vraća vrednost $\log_1 0x$ (mora da važi $x > 0$);

`pow(x, y)` vraća vrednost x^y;

`sqrt(x)` vraća vrednost $\sqrt{x}$ (mora da važi $x > 0$);

`fabs(x)` vraća apsolutni vrednost od x.

20.5 Zaglavlje `assert.h`

`assert.h` Prototip funkcije `assert` je:

```
void assert(int)
```

Funkcija `assert` se koristi 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 njihovom ispravljanju. 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.

Glava 21

Programi koji se sastoje od više jedinica prevođenja

C program predstavlja kolekciju spoljašnjih (globalnih) objekata (promenljivih i funkcija) koji mogu biti zadati u više *jedinica prevođenja (eng. compilation units)*. Iako se pod jedinicom prevođenja najčešće podrazumeva datoteka, ovo može biti donekle neprecizno. Naime, korišćenjem C pretprocesora moguće je da se više datoteka sastavi u jedinstvenu jedinicu prevođenja. U slučaju da je izvorni program sačinjen od nekoliko jedinica prevođenja, koje se kompiliraju nezavisno u tzv. *objektne module (eng. object module)*, a tek *povezivač (eng. linker)* dobijene objektne module povezuje u jedan izvršni program. Pri tom, obično se prilikom povezivanja koriste i funkcije standardne biblioteke koje se čuvaju u zasebnim, unapred prevedenim objektnim modulima.

21.1 Primer kreiranja i korišćenja korisničke biblioteke

Razmotrimo primer programa koji proverava da li su u liniji teksta koja se unosi sa standardnog ulaza dobro uparene zagrade (`()`, `[]` i `{}`). Prvi način da se ovo uradi je da se celokupan kôd programa smesti u jednu datoteku (na primer `brackets.c`).

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

#define MAX 100

int stack[MAX];
int sp = 0;

void push(int x) {
  stack[sp++] = x;
}

int pop() {
  return stack[--sp];
}

int empty() {
  return sp == 0;
}

#define ROUND   0
#define SQUARE  1
#define CURLY   2

int bracket_type(char c) {
  if (c == '(' || c == ')') return ROUND;
  if (c == '[' || c == ']') return SQUARE;
  if (c == '{' || c == '}') return CURLY;
}

int main() {
  int c;
  while((c = getchar()) != '\n')
    if (c == '(' || c == '[' || c == '{')
        push(bracket_type(c));
    else if (c == ')' || c == ']' || c == '}')
      if (pop() != bracket_type(c)) {
        printf("Error\n"); exit(1);
      }

  printf(empty() ? "OK\n" : "ERROR\n");
  return 0;
}
```

Ukoliko se program sastoji samo od jedne jedinice prevođenja (datoteke), običaj je da se ta datoteka prevede i odmah poveže sa C standardnom bibliote-

kom kako bi se dobio izvršni program. U slučaju da se koristi GCC kompilator, to bi se postiglo sa

```
gcc -o brackets brackets.c
```

Međutim, moguće je razdvojiti ova dva koraka i prvo prevesti jedinicu prevođenja i eksplicitno napraviti objektni modul (u ovom slučaju bi se zvao `brackets.o`), a tek zatim ovako napravljeni objektni modul povezati sa funkcijama standardne biblioteke jezika C i dobiti izvršni programa. U slučaju da se koristi GCC, to bi se postiglo sa

```
gcc -c brackets.c
gcc -o brackets brackets.o
```

Parametar `-c` govori sistemu GCC da ne bi trebalo da vrši povezivanje već da je potrebno da stane sa radom nakon kompilacije i da rezultat eksplicitno skladišti u objektni modul. Drugi poziv zatim vrši povezivanje objektnog modula sa funkcijama standardne biblioteke i gradi izvršni program.

Ukoliko se u prikazanom primeru bolje pogleda struktura koda, razaznaju se dve logički razdvojene celine. Prvi deo datoteke definiše implementaciju strukture podataka stek korišćenjem niza, dok drugi deo koda koristi ovu strukturu podataka kako bi reši problem ispitivanja uparenosti zagrada. Poželjno je ove dve logičke celine razdvojiti u dve zasebne jedinice prevođenja. Ovim bi se dobila mogućnost korišćenja implementacije strukture podataka steka i u drugim programima.

Datoteka `stack.c` bi tada imala sledeći sadržaj.

```
#define MAX 100

int stack[MAX];
int sp = 0;

void push(int x) {
  a[sp++] = x;
}

int pop() {
  return a[--sp];
}

int empty() {
  return sp == 0;
}
```

Datoteka `brackets.c` bi tada imala sledeći sadržaj.

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

#define ROUND   0
#define SQUARE  1
#define CURLY   2

int bracket_type(char c) {
  if (c == '(' || c == ')') return ROUND;
  if (c == '[' || c == ']') return SQUARE;
  if (c == '{' || c == '}') return CURLY;
}

int main() {
  int c;
  while((c = getchar()) != '\n')
    if (c == '(' || c == '[' || c == '{')
        push(bracket_type(c));
    else if (c == ')' || c == ']' || c == '}')
      if (pop() != bracket_type(c)) {
        printf("Error\n"); exit(1);
      }

  printf(empty() ? "OK\n" : "Error\n");
  return 0;
}
```

Prevođenje se u svakom slučaju vrši nezavisno, pri čemu je moguće pokrenuti prevođenje obe datoteke istovremeno sa:

```
gcc -o brackets brackets.c stack.c
```

Ipak, bolje rešenje u ovom slučaju je eksplicitno kreiranje objektnih modula.

```
gcc -c brackets.c
gcc -c stack.c
gcc -o brackets brackets.o stack.o
```

Naime, u slučaju da se promeni neka od dve izvorne datoteke, potrebno je ponoviti samo njeno prevođenje i povezivanje, dok se druga ne mora iznova prevoditi. Alat koji značajno može pomoći u ovakvim zadacima je `make` o kome će biti reči u nastavku.

Naizgled bi se moglo pomisliti da bi umesto korišćenja dve jedinice prevođenja, dve datoteke mogle biti spojene korišćenjem pretrocesora tako što bi se datoteka `stack.c` uključila u datoteku `brackets.c` sa:

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.c"

#define ROUND   0
...
```

Međutim, ovo rešenje je višestruko loše i suštinski se ne razlikuje od polazne varijante sa jednom datotekom. Naime, prilikom svakog prevođenja programa, kod iz obe datoteke biva prevođen. U slučaju komplikovanijih programa, moguće je da bi postojala potreba da se struktura podataka stek koristi u više različitih jedinica prevođenja, a na ovaj način bi se definicije promenljivih funkcija uključile na dva mesta što bi svakako dovelo do konflikta i nemogućnosti ispravnog povezivanja i izgradnje izvršnog programa. Generalno, poželjno je poštovati pravilo da se definicije funkcija i promenljivih smeštaju u `.c` datoteke koje nikako ne bi trebalo uključivati korišćenjem pretprocesorske `#include` direktive. Ova direktiva bi trebalo da se koristi isključivo za uključivanje deklaracija koje se obično smeštaju u `.h` datoteke.

Dalje, loša stvar u konkretnom primeru je činjenica da se prilikom prevođenja datoteke `brackets.c` apsolutno ništa ne zna o funkcijama za rad sa stekom, te se prilikom nailaska na pozive ovih funkcija ne vrši nikakva provera korektnosti argumenata niti bilo kakve implicitne konverzije. Rešenje ovog problema je umetanje prototipova funkcija za rad sa stekom na početak datoteke `brackets.c`.

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

void push(int x);
int pop();
int empty();

#define ROUND   0
...
```

Međutim, ovo može biti naporno i podložno greškama, tako da je mnogo bolje rešenje da se ove deklaracije grupišu u zasebnu datoteku zaglavlje `stack.h`, a zatim da se ta datoteka uključi u datoteku `brackets.c`. Dobar je običaj da se prototipovi uključe i u datoteku koja sadrži defincije funkcija (kako bi kompilator proverio da je sve konzistentno).

Datoteka `stack.h`

```
void push(int x);
int pop();
int empty();
```

Datoteka `brackets.c`

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

#define ROUND   0
...
```

Datoteka `stack.c`

```
#include "stack.h"

#define MAX 100
...
```

Time par datoteka `stack.c` i `stack.h` postaje biblioteka koja stoji na raspolaganju korisnicima. Ukoliko se ne želi distribuiranje izvornog koda, biblioteka može da se isporuči u prevedenom obliku `stack.o`, sa pratećim deklaracijama u `stack.h`. Često se grupa prevedenih objektnih modula arhivira u tzv. bibliotečke arhive (obično sa ekstenzijom `.a` ili `.lib`) koje se onda distribuiraju zajedno sa datotekama zaglavlja `.h`.

U nekim slučajevima, u datoteci zaglavlja se pored prototipova nalaze i neke definicije (najčešće struktura, a ponekad i funkcija i promenljivih). U slučaju da se greškom datoteka zaglavlje više puta uključi u neku jedinicu prevođenja (što se često događa preko posrednog uključivanja), desilo bi se da neka jedinica prevođenja sadrži višestruko ponavljanje nekih definicija što predstavlja grešku prilikom kompilacije. Rešenje dolazi uz pomoć pretprocesora i uslovno prevođenja. 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` karakteristican 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 mehanisam olakšava održavanje datoteka zaglavlja.

Tako bi datoteka `stack.h` trebalo da ima sadržaj:

```
#ifndef __STACK_H__
#define __STACK_H__
void push(int x);
int pop();
int empty();
#endif
```

21.2 Povezivanje

U ovom poglavlju će biti reči o vrstama *povezivanja promenljivih (eng. linkage of identifiers)* i funkcije koje određuje međusobni odnos između promenljivih i funkcija definisanih u različitim jedinicama prevođenja. Ključna pitanja koji moraju da se razreše prilikom povezivanja je da li je i kako moguće koristiti objekte (promenljive i funkcije) definisane u okviru jedne jedinice prevođenja u okviru neke druge jedinice prevođenja.

Jezik C razlikuje identifikatore *bez povezivanja*, identifikatore sa *spoljašnjim povezivanjem (eng. external linkage)* i identifikatore sa *unutrašnjim povezivanjem*. Naglasimo da ne bi trebalo mešati termine spoljašnjeg i unutrašnjeg dosega sa spoljašnjim i unutrašnjim povezivanjem (otuda korišćenje alternativnih termina globalni i lokalni za doseg) — ovi koncepti su nezavisni i povezivanje (i unutrašnje i spoljašnje) se najčešće odnosi isključivo na globalne objekte (tj. spoljašnjeg dosega).

Identifikatori bez povezivanja nisu vidljivi prilikom procesa povezivanja i mogu da se ponavljaju u različitim jedinicama prevođenja. Svaka deklaracija identifikatora bez povezivanja određuje jedinstveni nezavisni objekat. Lokalni objekti najčešće su objekti bez povezivanja.

Sve deklaracije identifikatora sa spoljašnjim povezivanjem u skupu jedinica prevođenja odredjuju jedinstveni objekat, tj. sve pojave ovakvog identifikatora u različitim jedinicama prevođenja se odnose na jedan isti objekat.

Sve deklaracije identifikatora sa unutrašnjim povezivanjem u okviru jedne jedinice prevođenja se odnose na isti objekat. Identifikatori sa unutrašnjim povezivanjem se ne mogu koristiti kroz različite jedinice prevođenja.

21.2.1 Unutrašnje povezivanje i kvalifikator `static`

Osnovna uloga unutrašnjeg povezivanja je obično u tome da neki deo koda učini zatvorenim, u smislu da je nemoguće menjanje nekih njegovih globalnih (spoljašnjih) promenljivih ili pozivanje nekih njegovih funkcija iz drugih datoteka. Time se taj deo koda „enkapsulira“ i čini dostupnim samo kroz preostale tačke komunikacije (preostale spoljašnje promenljive i preostale funkcije). Ovim autor koda ne želi da onemogući druge autore da koriste deo njegovog koda, već pre da im pomogne da jasnije vide način njegovog korišćenja (ali naravno i da onemogući nehotične greške). Ukoliko se u nekoj drugoj datoteci deklariše globalna (spoljašnja) promenljiva ili funkcija istog imena, to ne dovodi do konflikta i greške u prevođenju.

Kako bi se naglasilo da neki globalni objekat ima unutrašnje povezivanje, neophodno je korišćenje kvalifikatora `static`. Kvalifikator `static` u programskom jeziku C ima dvojaku ulogu i različito dejstvo kada se primenjuje na lokalne promenljive i kada se primenjuje na globalne promenljive ili funkcije. Već je navedeno da se u slučaju lokalni promenljivih ovaj kvalifikator koristi kako bi se naglasilo da promenljiva treba da ima statički životni vek. Međutim, u slučaju da se koristi uz globalnu promenljivu ili funkciju, uloga kvalifikatora `static` je da naglasi da ta globalna promenljiva ili funkcija ima unutrašnje poveziva-

nje. Drugim rečima, kvalifikator `static` onemogućava korišćenje promenljive ili funkcije iz drugih datoteka koje čine program.

U ranije navedenom primeru, poželjno je zabraniti korisnicima biblioteke direktan pristup promenljivim `sp` i `a`.

```
static int stack[MAX];
static int sp = 0;
```

Ovim se povezivanje ovih imena ograničava na datoteku `stack.c` i u slučaju da se u nekoj drugoj jedinci prevođenja nađu globalne promenljive sa istim imenom, smatra se da one predstavljaju potpuno nezavisne objekte.

21.2.2 Spoljašnje povezivanje i kvalifikator `extern`

Kvalifikator `extern` je u tesnoj vezi sa spoljašnjim povezivanjem i koristi se kod programa koji se sastoje iz više jedinica prevođenja. Za razliku od funkcija, gde smo jasno razlikovali deklaraciju (kojom se kompilator samo obaveštava o postojanju funkcije datog imena i o njenim tipovima), od definicije (kojom se kompilatoru saopštava celokupan kôd funkcije), u slučaju promenljivih, takva distinkcija nije rađena. Svaka deklaracija promenljive je obezbeđivala i određeni prostor za promenljivu. U slučaju spoljašnjeg povezivanja, potrebno je da se isti identifikator koristi za isti objekat u okviru više jedinica prevođenja. Kada bi se koristili do sada prikazani načini deklarisanja promenljivih, prilikom prevođenja svake jedinice, bile bi kreirane pojedinačne instance odgovarajućeg objekta i prilikom povezivanja bi došlo do prijavljivanja greške. Umesto ovoga, koristi se `extern` kvalifikator kojim se promenljiva deklariše i uvodi se njen doseg, ali se naglašava njeno spoljašnje povezivanje i ne obezbeđuje se poseban prostor za nju, već se samo govori da je takva promenljiva (promenljiva tog tipa i imena) definisana negde drugde (u nastavku datoteke ili u nekoj drugoj datoteci). U okviru programa, može da postoji samo jedna prava definicija promenljive (i postoji samo jedna takva promenljiva — koja se čuva u segmentu podataka), dok može da postoji više (na primer, u svakoj jedinici prevođenja programa koja je koristi) `extern` deklaracija. Inicijalizacija promenljive moguća je samo u okviru prave deklaracije (a ne i u okviru `extern` deklaracije). Za niz u `extern` deklaraciji nije potrebno navoditi dimenziju.

Za funkcije se podrazumeva spoljašnje povezivanje i uz deklaracije funkcije nema potrebe eksplicitno navoditi kvalifikator `extern`.

Kako bismo ilustrovali spoljašnje povezivanje i kvalifikator `extern`, razmotrimo alternativnu varijantu, u kojoj želimo da korisniku biblioteke za rad sa stekovima omogućimo direktan pristup nizu `stack` i promenljivoj `sp`. Premeštanje definicija ovih objekata u datoteku `stack.h` nije rešenje. Naime, iako prevođenje obe jedinice prolazi bez problema, prilikom povezivanja javlja se sledeća greška:

```
stack.o:(.bss+0x0): multiple definition of ‘sp’
brackets.o:(.bss+0x0): first defined here
collect2: ld returned 1 exit status
```

Naime, primećeno je da u dva različita objektna modula (`stack.o` i `brackets.o`)

postoji definicija promenljive `sp`, pri čemu ta promenljiva ima spoljašnje povezivanje, što nije dopušteno.

Rešenje zahteva korišćenje kvalifikatora `extern`. Naime, definicije niza `stack` i promenljive `sp` mogu da ostanu u okviru datoteke `stack.c`, bez navođenja bilo kakvog kvalifikatora. Pri tom, poželjno je obezbediti da se u okviru datoteke `brackets.c` nađe `extern` deklaracija ovih objekata. Ovo je najbolje uraditi tako što se ove deklaracije smeste u u zaglavlje `stack.h`.

```
#ifndef __STACK_H__
#define __STACK_H__
extern int sp;
extern int stack[];
void push(int x);
int pop();
int empty();
#endif
```

Ovo je zadovoljavajuće rešenje.

Napomenimo, još da, u slučaju da u datoteci `stack.c` nije postojala prava definicija objekta, bila prijavljena greška prilikom povezivanja.

```
stack.o: In function ‘push’:
stack.c:(.text+0xe): undefined reference to ‘stack’
stack.o: In function ‘pop’:
stack.c:(.text+0x34): undefined reference to ‘stack’
collect2: ld returned 1 exit status
```

Dodatak A

Tabela prioriteta operatora

Operator	Opis	Asocijativnost
`()`	Poziv funkcije	s leva na desno
`[]`	indeksni pristup elementu niza	
`.`	pristup članu strukture ili unije	
`->`	pristup članu strukture ili unije preko pokazivača	
`++ --`	postfiksni inkrement/dekrement	
`++ --`	prefiksni inkrement/dekrement	s desna na levo
`+ -`	unarni plus/minus	
`! ~`	logička negacija/bitski komplement	
`(type)`	eksplicitna konverzija tipa (cast)	
`*`	dereferenciranje	
`&`	referenciranje (adresa)	
`sizeof`	veličina u bajtovima	
`* / %`	množenje, deljenje, ostatak	s leva na desno
`+ -`	sabiranje i oduzimanje	s leva na desno
`<< >>`	bitsko pomeranje ulevo i udesno	
`< <=`	relacije manje, manje jednako	s leva na desno
`> >=`	relacije veće, veće jednako	
`== !=`	relacija jednako, različito	s leva na desno
`&`	bitska konjunkcija	s leva na desno
`^`	bitska ekskluzivna disjunkcija	s leva na desno
`|`	bitska disjunkcija	s leva na desno
`&&`	logička konjunkcija	s leva na desno
`||`	logička disjunkcija	s leva na desno
`?:`	ternarni uslovni	s desna na levo
`=`	dodele	s desna na levo
`+= -=`		
`*= /= %=`		
`&= ^= |=`		
`<<= >>=`		
`,`	spajanje izraza u jedan	s leva na desno

Dodatak B

Rešenja zadataka

Rešenje zadatka 3.2:

Predloženi algoritam se zasniva na sledećoj osobini:

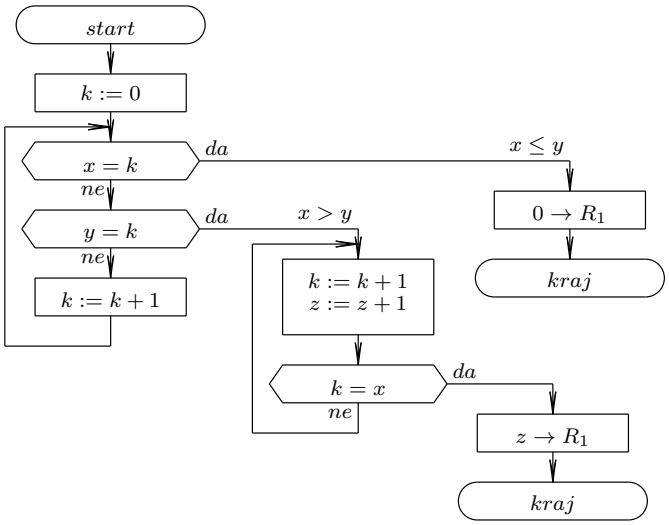
$$z = x - y \Leftrightarrow x = y + z$$

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.	$T(4,1)$	$z \to R_1$

Rešenje zadatka 3.3:

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:

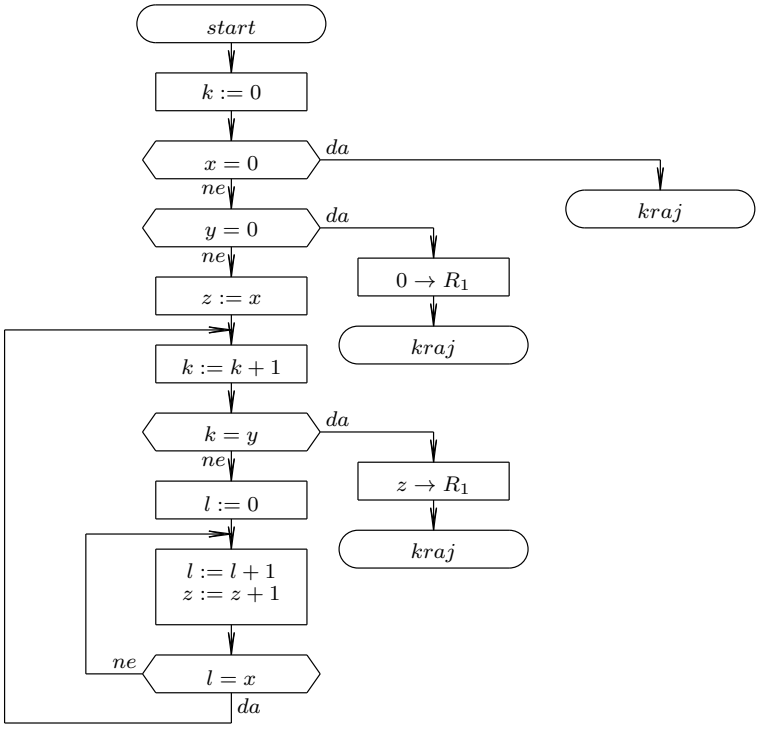
R_1	R_2	R_3	$\ldots$
x	y	0	$\ldots$

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 6.6:

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

int main() {
   double r;
   printf("Unesi poluprecnik kruga: ");
   scanf("%lf", &r);
   printf("Obim kruga je: %lf\n", 2*r*M_PI);
   printf("Povrsina kruga je: %lf\n", r*r*M_PI);
   return 0;
}
```

Rešenje zadatka 6.7:

```
#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 6.8:

```
#include <stdio.h>
#include <math.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);
   printf("Unesi duzinu stranice b: ");
   scanf("%lf", &b);
   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 6.9:

```
#include <stdio.h>

int main() {
int kmh;
printf("Unesi brzinu u km/h: ");
scanf("%d", &kmh);
printf("Brzina u ms/s je: %f\n", kmh * 1000.0 / 3600.0);
return 0;
}
```

Rešenje zadatka 6.10:

```
#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);
}
```

Rešenje zadatka 6.12:

```
#include <stdio.h>

int main() {
  int n, c0, c1;
  printf("Unesi broj: ");
  scanf("%d", &n);
  c0 = n % 10;
  c1 = (n / 10) % 10;
  printf("Zamenjene poslednje dve cifre: %d\n",
     (n / 100)*100 + c0*10 + c1);
  return 0;
}
```

Rešenje zadatka 6.14:

```
#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
  ...
  */
  int x, y, z, n;
  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: %d\n", n);
  return 0;
}
```

Rešenje zadatka 6.15:

```
#include <stdio.h>

int main() {
  int a, b, c, m;
  printf("Unesi tri broja: ");
  scanf("%d %d %d", &a, &b, &c);
  m = a;
  if (b > m)
```

```
    m = b;
  if (c > m)
    m = c;
  printf("Najveci broj je: %d\n", m);
  return 0;
}
```

Rešenje zadatka 6.16:

```
#include <stdio.h>

int main() {
  float A, B;
  printf("Unesi koeficijente A i B jednacine A*X + B = 0: ");
  scanf("%f %f", &A, &B);
  if (A != 0)
    printf("Jednacina ima jedinstveno resenje: %f\n", -B/A);
  else if (B == 0)
    printf("Svaki realan broj zadovoljava jednacinu\n");
  else
    printf("Jednacina nema resenja\n");
  return 0;
}
```

Rešenje zadatka 6.18:

```
#include <stdio.h>
#include <math.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);
  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 7.15:

```
#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 8.1:

```
#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 8.2:

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

#define N 100

int main() {
  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 8.3:

```
#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 8.4:

```
#include <stdio.h>

#define N 4

int main() {
  int i, j;

  /* Deo pod (a): */
```

```
for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++)
    putchar('*');
  putchar('\n');
}

printf("--------------\n");

/* Deo pod (b): */
for (i = 0; i < N; i++) {
  for (j = 0; j < N - i; j++)
    putchar('*');
  putchar('\n');
}

printf("--------------\n");

/* Deo pod (c): */
for (i = 0; i < N; i++) {
  for (j = 0; j < i + 1; j++)
    putchar('*');
  putchar('\n');
}

printf("--------------\n");

/* Deo pod (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 pod (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 pod (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 pod (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 8.5:

```
#include <stdio.h>

int main() {
  unsigned n, d;
  scanf("%u", &n);
  for (d = 1; d <= n; d++)
    if (n % d == 0)
      printf("%u\n", d);
  return 0;
}
```

```
#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. */
int main() {
  unsigned n, s, d;
  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;
}
```

Rešenje zadatka 8.6:

```
#include <stdio.h>

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 8.7:

```
#include <stdio.h>

int main() {
  unsigned d, n;
  scanf("%u", &n);

  d = 2;
  while (n > 1) {
    if (n % d == 0) {
      printf("%u\n", d);
      n /= d;
    } else
      d++;
  }
  return 0;
}
```

Rešenje zadatka 8.8:

```
#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 8.9:

```
#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;
      if (tb == pb)
        ts++;
      else
        ts = 1;

      if (ts > ns)
        ns = ts;
      pb = tb;
    }
  }
  printf("%d\n", ns);
  return 0;
}
```

Rešenje zadatka 8.10:

```
#include <stdio.h>

int main() {
  unsigned x, N1, N2, n, l;
  n = 0; N1 = 0; N2 = 1;
  scanf("%d", &x);
  while (x != N1) {
    N1++;
    if (N1 == N2) {
      n++;
      N2++;
      l = 0;
      do {
        l++;
        N2 += 2;
      } while (l != n);
    }
  }
  printf("%d\n", n);
}
```

Rešenje zadatka 8.11:

```
#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 8.12:

```
#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 8.13:

```
#include <stdio.h>

int main() {
  unsigned n;
  printf("Unesi broj: ");
  scanf("%u", &n);
  do {
```

```
    printf("%u\n", n % 10);
    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 8.15:

```
#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 8.16:

```
#include <stdio.h>

int main() {
  unsigned n, s = 1, r = 0;
  scanf("%u", &n);
  while (n > 0) {
    unsigned c = n % 10;
```

```
    if (c % 2 == 0) {
      r = n % 10 * s + r;
      s *= 10;
    }
    n /= 10;
  }
  printf("%u\n", r);
  return 0;
}
```

Rešenje zadatka 8.18:

```
#include <stdio.h>

int main() {
  unsigned n, a, b, s, poc, sred, kraj;
  scanf("%u", &n);

  if (n < 10) {
    printf("%u\n", n);
    return 0;
  }

  b = n; s = 1;
  while (b >= 10) {
    s *= 10;
    b /= 10;
  }

  poc = n / s;
  sred = (n % s) / 10;
  kraj = n % 10;

  printf("%u\n", s*kraj + 10 * sred + poc);

  return 0;

}
```

```
/* Rotiranje ulevo */
#include <stdio.h>

int main() {
  unsigned n, a, b, s;
  scanf("%u", &n);

  b = n; s = 1;
```

```
  while (b >= 10) {
    s *= 10;
    b /= 10;
  }

  printf("%u\n", s * (n % 10) + n / 10);

  return 0;
}
```

```
/* Rotiranje udesno */
#include <stdio.h>

int main() {
  unsigned n, a, b, s;
  scanf("%u", &n);

  b = n; s = 1;
  while (b >= 10) {
    s *= 10;
    b /= 10;
  }

  printf("%u\n", n % s * 10 + n / s);

  return 0;
}
```

Rešenje zadatka 8.19:

```
#include <stdio.h>

int main() {
  int d, m, g, dm;
  scanf("%d%d%d", &d, &m, &g);
  switch(m) {
  case 1: case 3: case 5: case 7: case 8: case 10: case 12:
    dm = 31;
    break;
  case 4: case 6: case 9: case 11:
    dm = 31;
    break;
  case 2:
    dm = (g % 4 == 0 && g % 100 != 0 || g % 400 == 0) ? 29 : 28;
    break;
  default:
    printf("Pogresan mesec\n");
```

```
    return 1;
  }
  if (d < 1 || d > dm) {
    printf("Pogresan dan\n");
    return 1;
  }
  if (g < 0) {
    printf("Pogresna godina\n");
    return 1;
  }
  return 0;
}
```

Rešenje zadatka 9.2:

```
#include <stdio.h>

int prost(unsigned n) {
  unsigned d;
  for (d = 2; d * d <= n; d++)
    if (n % d == 0)
      return 0;
  return 1;
}

int main() {
  unsigned n;
  scanf("%u", &n);
  printf(prost(n) ? "Prost\n" : "Slozen\n");
}
```

Rešenje zadatka 9.4:

```
#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 9.5:

```
#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 10.1:

```
#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; 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 10.2:

```
#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 10.3:

```
#include <stdio.h>

/* Napomena: odredjen broj funkcija u ovom zadatku se moze
   implementirati i efikasnije, medjutim, uz prilicno komplikovaniji
```

```
   kod. */

int najduza_serija(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;
  for (i = 0; i + m - 1 < n; i++) {
    for (j = 0; j < m; j++)
      if (a[i + j] != b[j])
        break;
    if (j == m)
      return 1;
  }
  return 0;
}

int podniz(int a[], int n, int b[], int m) {
  int i, j;
  for (i = 0, j = 0; i < n && j < m; i++)
    if (a[i] == b[j])
      j++;
  return j == m;
}

int palindrom(int a[], int n) {
  int i, j;
  for (i = 0, j = n-1; i < j; i++, j--)
    if (a[i] != a[j])
      return 0;
  return 1;
}

void obrni(int a[], int n) {
```

```
  int i, j;
  for (i = 0, j = n-1; i < j; i++, j--) {
    int tmp = a[i];
    a[i] = a[j];
    a[j] = tmp;
  }
}

void rotiraj_desno(int a[], int n, int k) {
  int i, j;
  for (j = 0; j < k; j++) {
    int tmp = a[n-1];
    for (i = n-1; i > 0; i--)
      a[i] = a[i-1];
    a[0] = tmp;
  }
}

void rotiraj_levo(int a[], int n, int k) {
  int i, j;
  for (j = 0; j < k; j++) {
    int tmp = a[0];
    for (i = 0; i + 1 < n; i++)
      a[i] = a[i+1];
    a[n - 1] = tmp;
  }
}

void ispisi(int a[], int n) {
  int i;
  for (i = 0; i < n; i++)
    printf("%d ", a[i]);
  printf("\n");
}

int izbaci_duplikate(int a[], int n) {
  int i, j, k;
  for (i = 0; i < n; i++) {
    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;
  i = 0, j = 0, k = 0;
  while (i < n && j < m)
    c[k++] = a[i] < b[j] ? a[i++] : b[j++];
  while (i < n)
    c[k++] = a[i++];
  while (j < m)
    c[k++] = b[j++];
  return k;
}
```

Rešenje zadatka 10.7:

```
#include <stdio.h>

/* Provera da li je godina prestupna */
int prestupna(int g) {
  return (g % 4 == 0 && g % 100 != 0) || (g % 400 == 0);
}

int main() {
  int d, m, g, b, M;
  int br_dana[] = {-1, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
  scanf("%d%d%d", &d, &m, &g);
  /* 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);
}
```