1. To C++ or not to C++?

Više nego bilo kada u povijesti, čovječanstvo se nalazi na razmeđi. Jedan put vodi u očaj i krajnje beznađe, a drugi u potpuno istrebljenje. Pomolimo se da ćemo imati mudrosti izabrati ispravno.

Woody Allen, „Popratne pojave" (1980)

Prilikom pisanja ove knjige mnogi su nas, s podsmijehom kao da gledaju posljednji primjerak snježnog leoparda, pitali: „No zašto se bavite C++ jezikom? To je kompliciran jezik, spor, nedovoljno efikasan za primjenu u komercijalnim programima. Na kraju, ja to sve mogu napraviti u običnom C-u." Prije nego što počnemo objašnjavati pojedine značajke jezika, nalazimo važnim pokušati dati koliko-toliko suvisle odgovore na ta i slična pitanja.

Ovo poglavlje zamišljeno je da dade okvirne informacije čitateljici ili čitatelju. U njemu će se pojaviti mnoštvo pojmova koji nisu neophodni za daljnje čitanje knjige. Neki od tih pojmova bit će dodatno objašnjeni u kasnijim poglavljima. Stoga, ako vam nešto nije jasno, nemojte se obeshrabrivati – jednostavno pređite preko toga i nastavite.

1.1. Povijesni pregled razvoja programskih jezika

Osnovno pitanje je što C++ čini boljim i pogodnijim općenamjenskim jezikom za pisanje raznovrsnih programa, od operacijskih sustava do baza podataka. Da bismo to razumjeli, pogledajmo kojim putem je tekao povijesni razvoj jezika. Na taj način će možda biti jasnija motivacija Bjarne Stroustrupa, „oca i majke" jezika C++. Dakle, krenimo „od stoljeća sedmog".

Prva računala koja su se pojavila bila su vrlo složena za korištenje. Njih su koristili isključivo stručnjaci koji su bili osposobljeni za komunikaciju s računalom. Ta komunikacija se sastojala od dva osnovna koraka: davanje uputa računalu i čitanje rezultata obrade. I dok se čitanje rezultata vrlo brzo učinilo koliko-toliko snošljivim uvođenjem pisača na kojima su se rezultati ispisivali, unošenje uputa – programiranje – se sastojalo od mukotrpnog unosa niza nula i jedinica. Ti nizovi su davali računalu upute kao što su: „zbroji dva broja", „premjesti podatak s neke memorijske lokacije na drugu", „skoči na neku instrukciju izvan normalnog slijeda instrukcija" i slično. Kako je takve programe bilo vrlo složeno pisati, a još složenije čitati i ispravljati, ubrzo su se pojavili prvi programerski alati nazvani *asembleri* (engl. *assemblers*) koji su instrukcije iz asemblerskog jezika prevodili u strojne instrukcije.

U asemblerskom jeziku svaka strojna instrukcija predstavljena je mnemonikom tj. nekom pridruženom kraticom koja je razumljiva ljudima koji čitaju program. Tako se za

zbrajanje najčešće koristi mnemonik `ADD`, dok se za premještanje podataka s jednog mjesta u memoriji na neko drugo mjesto koristi simbol `MOV`. Time je postignuta bolja čitljivost programa, no i dalje je bilo vrlo složeno pisati programe i ispravljati ih jer je bilo potrebno davati sve, pa i najdetaljnije upute računalu za svaku pojedinu operaciju. Javlja se problem koji će kasnije, nakon niza godina, dovesti i do pojave programskog jezika C++: potrebno je razviti programerski alat koji će osloboditi programera rutinskih poslova te mu dopustiti da se usredotoči na problem koji rješava.

Zbog toga su se pojavili viši programski jezici koji su preuzeli na sebe dio rutinskih „dosadnih" programerskih poslova. Primjerice, viši programski jezici omogućili su da programer prilikom pisanja programa više ne mora voditi računa o tome na kojoj je adresi u memoriji podatak pohranjen, već da podatke dohvaća preko simboličkih imena koja sam programer zadaje, koristeći pri tome uobičajenu matematičku notaciju. Kako su se računala koristila za sve raznovrsnije namjene, tako su stvarani i novi programski jezici koji su bili pogodniji za dotične namjene. Tako je FORTRAN bio posebno pogodan za matematičke proračune, zatim BASIC koji se vrlo brzo učio, te COBOL koji je bio u pravilu namijenjen upravljanju bazama podataka.

Oko 1972. se pojavljuje jezik C, koji je direktna preteča današnjeg jezika C++. To je bio prvi jezik opće namjene te je postigao neviđen uspjeh. Više je razloga tome: jezik je bio jednostavan za učenje, omogućavao je modularno pisanje programa, sadržavao je samo naredbe koje se mogu jednostavno prevesti u strojni jezik, davao je brzi izvedbeni kôd. Jezik nije bio opterećen mnogim složenim funkcijama, kao na primjer *skupljanje smeća* (engl. *garbage collection*) : ako je takav podsustav nekome trebao, korisnik ga je sam napisao. Jezik je omogućavao vrlo dobru kontrolu strojnih resursa te je na taj način omogućio programerima da optimiziraju svoj kôd. Početkom devedesetih godina prošlog stoljeća, većina komercijalnih programa bila je napisana u C-u, ponegdje dopunjena odsječcima u strojnom jeziku kako bi se kritični dijelovi sustava učinili dovoljno brzima.

No kako je razvoj programske podrške napredovao, stvari su se i na području programskih jezika počele mijenjati. Složeni projekti od nekoliko stotina tisuća, pa i više redaka na kojima rade timovi od desetak ili stotinjak programera, više nisu rijetkost, pa je zbog toga bilo potrebno uvesti dodatne mehanizme kojima bi se takvi programi učinili jednostavnijima za izradu i održavanje, te kojima bi se omogućilo da se jednom napisani kôd iskoristi u više različitih projekata.

Bjarne Stroustrup (rođen u Danskoj) je 1979. godine započeo rad na jeziku „C s klasama" (engl. *C with Classes*). Prije toga, on je radio na svom doktoratu u *Computing Laboratory of Cambridge* te istraživao distribuirane sustave, granu računalne znanosti u kojoj se proučavaju modeli obrade podataka na više jedinica istodobno. Pri tome je koristio jezik Simula, koji posjedovao neka važna svojstva koja su ga činila prikladnim za taj posao. Na primjer, Simula je posjedovala pojam *klase* (engl. *class* - vrsta, razred, klasa) – strukture koja objedinjava podatke i operacije nad podacima. Korištenje klasa omogućilo je da se koncepti problema koji se rješava izraze direktno pomoću jezičnih konstrukcija. Dobiveni kôd je bio vrlo čitljiv i razumljiv, a g. Stroustrup je bio posebno fasciniran načinom na koji je sam programski jezik upućivao programera u razmišljanju

o problemu. Također, jezik je posjedovao sustav tipizacije, koji je često pomagao korisniku u pronalaženju pogrešaka već prilikom prevođenja.

Naoko idealan u teoriji, jezik Simula je posrnuo u praksi: prevođenje je bilo iznimno dugotrajno, a kôd se izvodio izuzetno sporo. Dobiveni program je bio neupotrebljiv i, da bi ipak pošteno zaradio svoj doktorat, gospodin Stroustrup se morao potruditi i ponovo napisati cjelokupni program u jeziku BCPL – jeziku niske razine koji je omogućio vrlo dobre performanse prevedenog programa. No iskustvo pisanja složenog programa u takvom jeziku je bilo frustrirajuće i g. Stroustrup je, po završetku svog posla na Cambridgeu, čvrsto sebi obećao da više nikada neće takav složen problem pokušati riješiti neadekvatnim alatima poput BCPL-a ili Simule.

Kada se 1979. zaposlio u *Bell Labs* (kasnije *AT&T*) u Murray Hillu, započeo je rad na onome što će kasnije postati C++. Na osnovu svog iskustva stečenog prilikom rada na doktoratu pokušao je stvoriti univerzalni jezik koji će udovoljiti današnjim zahtjevima. Pri tome je uzeo dobra svojstva niza jezika: Simula, Clu, Algol68 i Ada, a kao osnovu za sintaksu je uzeo jezik C, koji je već tada bio vrlo popularan i koji je uostalom bio stvoren u *Bellovim laboratorijima*.

1.2. Osnovna svojstva jezika C++

Četiri su važna svojstva jezika C++ koja ga čine objektno orijentiranim:

1. *enkapsulacija* (engl. *encapsulation*),
2. *skrivanje podataka* (engl. *data hiding*),
3. *nasljeđivanje* (engl. *inheritance*) i
4. *polimorfizam* (engl. *polymorphism*).

Sva ta svojstva doprinose ostvarenju takozvane *objektno orijentirane paradigme* programiranja.

Da bismo to bolje razumjeli, pogledajmo koji su se programski modeli koristili u prošlosti. Pri tome je svakako najvažniji model proceduralno strukturiranog programiranja.

Proceduralno programiranje zasniva se na promatranju programa kao niza jednostavnih programskih odsječaka, *procedura*. Svaka procedura je konstruirana tako da obavlja jedan manji zadatak, a cijeli se program sastoji od niza procedura koje sudjeluju u rješavanju zadatka.

Kako bi koristi od ovakve podjele programa bile što izraženije, smatralo se dobrom programerskom taktikom odvojiti proceduru od podataka koje ona obrađuje: time je bilo moguće pozvati proceduru za različite ulazne podatke i na taj način iskoristiti je na više mjesta. *Strukturirano programiranje* je samo dodatak na proceduralni model: ono definira niz osnovnih jezičnih konstrukcija, kao što su petlje, grananja i pozivi procedura, koje unose red u programe i čine samo programiranje daleko jednostavnijim, a napisani kôd čitljivijim.

Princip kojim bismo mogli obilježiti proceduralno strukturirani model jest *podijeli-pa-vladaj*: cjelokupni program je presložen da bi ga se moglo razumjeti pa se zbog toga on rastavlja na niz manjih zadataka – procedura – koje su dovoljno jednostavne da bi se

mogle izraziti pomoću naredbi programskog jezika. Pri tome, pojedina procedura također ne mora biti riješena monolitno: ona može svoj posao obaviti kombinirajući rad niza drugih procedura.

Ilustrirat ćemo to primjerom: zamislimo da želimo izraditi kompleksan program za obradu trodimenzionalnih objekata. Kao jednu od mogućnosti koje moramo ponuditi korisnicima jest rotacija objekata oko neke točke u prostoru. Koristeći proceduralno programiranje, taj zadatak bi se mogao raščlaniti na sljedeće operacije:

1. listaj sve objekte redom;
2. za pojedini objekt odredi njegov tip;
3. ovisno o tipu, pozovi odgovarajuću proceduru koja će izračunati novu poziciju objekta;
4. u skladu s tipom podataka ažuriraj koordinate objekta.

Operacije određivanja tipa, izračunavanje nove pozicije objekta i ažuriranje koordinata mogu se dalje predstaviti pomoću procedura koje sadržavaju niz jednostavnijih akcija.

Ovakav programski pristup je bio vrlo uspješan do kasnih osamdesetih godina dvadesetog stoljeća, kada su njegovi nedostaci postajali sve očitiji. Naime, odvajanje podataka i procedura čini programski kôd težim za čitanje i razumijevanje. Prirodnije je o podacima razmišljati preko operacija koje možemo obaviti nad njima – u gornjem primjeru to znači da o kocki ne razmišljamo pomoću koordinata njenih vrhova već pomoću mogućih operacija, kao što je rotacija kocke.

Nadalje, pokazalo se složenim istodobno razmišljati o problemu i odmah strukturirati rješenje. Umjesto rješavanja problema, programeri su mnogo vremena provodili pronalazeći načine da programe usklade sa zadanom strukturom.

Također, današnji programi se pokreću pomoću miša, prozora, izbornika i dijaloga. Programiranje je *pogonjeno događajima* (engl. *event-driven*) za razliku od starog, sekvencijalnog načina. Proceduralni programi su korisniku, u trenutku kada je bila potrebna interakcija korisnika (na primjer zahtjev za ispis na pisaču) prikazivali ekran nudeći mu opcije; ovisno o odabranoj opciji, izvođenje kôda se usmjeravalo na određeni programski odsječak. *Pogonjeno događajima* znači da se program ne odvija po unaprijed određenom slijedu, već se programom upravlja pomoću niza događaja. Događaja ima raznih: pomicanje miša, pritisak na tipku, izbor stavke iz izbornika i slično. Sada su sve opcije dostupne istodobno, a program postaje interaktivan, što znači da promptno odgovara na korisnikove zahtjeve i odmah (ovo ipak treba shvatiti uvjetno) prikazuje rezultat svoje akcije na zaslonu računala.

Kako bi se takvi zahtjevi jednostavnije proveli u praksi, razvijen je *objektni pristup programiranju*. Osnovna ideja je razbiti program u niz zatvorenih cjelina (*objekata*) koje međusobno surađuju u rješavanju problema. Umjesto specijaliziranih procedura koje barataju proizvoljnim podacima, radimo s *objektima* koji objedinjavaju podatke i operacije nad tim podacima. Pri tome je važno što objekt radi, a ne kako on to radi. To omogućava da se pojedini objekt može po potrebi izbaciti i zamijeniti drugim koji će istu zadaću obaviti bolje.

Objedinjavanje podataka i operacija naziva se *enkapsulacija* (engl. *encapsulation*) – objekt se ne sastoji isključivo od podataka već sadrži i metode neophodne za operacije nad tim podacima. Pritom su podaci privatni za svaki objekt te nisu dostupni ostalim

dijelovima programa. Na taj su način podaci zaštićeni od neovlaštene promjene izvan objekta koja bi mogla narušiti njegovu cjelovitost. Mehanizam zaštite podataka naziva se *skrivanje podataka* (engl. *data hiding*). Svaki objekt svojoj okolini pruža isključivo podatke i operacije koji su neophodni da bi okolina objekt mogla koristiti. Ti javno dostupni podaci zajedno s operacijama koje ih prihvaćaju ili vraćaju čine *sučelje* (engl. *interface*) objekta. Programer koji će koristiti taj objekt više se ne mora zamarati razmišljajući o načinu na koji objekt iznutra funkcionira – on jednostavno preko sučelja traži od objekta određenu uslugu.

Objektni pristup ima izravnu analogiju sa svakodnevnim životom pa pokušajmo opisane principe ilustrirati na praktičnom primjeru. Iako mnogi automobil percepiraju kroz tehničke podatke kao što su maksimalna brzina, ubrzanje, potrošnja goriva ili boja karoserije, osnovna namjena automobila jest prijevoz osoba ili robe. Dakle, automobil promatramo kroz operacije koje on pruža tj. koliko osoba ili robe može prevesti i u kojem vremenu. Za obavljanje te operacije potrebna mu je između ostaloga određena količina goriva koje pokreće motor i akumulator koji osigurava struju za elektropokretač kojim se motor aktivira. Srećom, za pokretanje motora vozač ne mora spajati žice akumulatora na elektropokretač, kao što ni tijekom vožnje ne mora sam dolijevati gorivo u cilindar automobila. Te operacije su *enkapsulirane* u mehanizmu motora, odnosno automobila. Cijev za dotok goriva, kablovi za svjećice i turbo-ubrizgavači su *skriveni* od vozača, pod pokrovom motora. Da nije tako, lako bi se moglo dogoditi da vozač ili slučajni prolaznik nestručnim prčkanjem zamijeni cijev dotoka goriva i kablove svjećica te prouzroči zapaljenje automobila. Vozaču je za upravljanje automobilom na raspolaganju volan, papučica gasa, kočnica, spojka i ručica mjenjača – te kontrole čine *sučelje* preko kojega vozač obavlja operacije nad automobilom.

Kada PC preprodavač, vlasnik poduzeća „Taiwan/tavan-Commerce” sklapa računalo, on zasigurno treba kućište (iako se i to ponekad pokazuje nepotrebnim). To ne znači da će on morati početi od nule (miksajući atome željeza u čašici od *Kinderlade*); on će jednostavno otići kod susjednog dilera i kupiti gotovo kućište koje ima priključak na mrežni napon, ATX napajanje, ladice za montažu diskova te otvor za DVD pogon. Tako je i u programiranju: moguće je kupiti gotove programske komponente koje se zatim mogu iskoristiti u programu. Nije potrebno razumjeti kako komponenta radi – dovoljno je poznavati njeno sučelje da bi ju se moglo iskoristiti.

Također, kada projektanti u Renaultu žele izraditi novi model automobila, imaju dva izbora: ili mogu početi od nule i ponovo proračunavati svaki najmanji dio motora, šasije i ostalih dijelova, ili mogu jednostavno novi model bazirati na nekom starom modelu. Kompaniji je cilj što brže razviti novi model kako bi pretekla konkurenciju, pa će zasigurno jednostavno uzeti uspješan model automobila i samo izmijeniti neka njegova svojstva: promijenit će mu liniju, pojačati motor, dodati elektroniku za pouzdanije upravljanje. Slično je i s programskim komponentama: prilikom rješavanja nekog problema možemo uzdahnuti i početi kopati, ili možemo uzeti neku već gotovu komponentu koja je blizu rješenja i samo dodati nove mogućnosti. To se zove *ponovna iskoristivost* (engl. *reusability*) i vrlo je važno svojstvo. Za novu programsku komponentu kaže se da je *naslijedila* (engl. *inherit*) svojstva komponente iz koje je izgrađena.

Korisnik koji kupuje auto sigurno neće biti presretan ako se njegov novi model razlikuje od starog po načinu korištenja, primjerice da se umjesto pritiskom na papučicu gasa auto ubrzava povlačenjem ručice na krovu vozila ili spuštanjem suvozačevog sjedala. On jednostavno želi pritisnuti gas, a stvar je nove verzije automobila kraće vrijeme ubrzanja od 0 do 100 km/h. Slično je i s programskim komponentama: korisnik se ne treba opterećivati time koju verziju komponente koristi – on će jednostavno tražiti od komponente uslugu, a na njoj je da to obavi na adekvatan način. U primjeru s rotacijama tijela, korisnik će od svakog pojedinog tijela zatražiti istu operaciju – da se zakrene – a tijelo će operaciju izvesti na njen svojstven način. Svojstvo da različiti objekti istu operaciju obavljaju na različite načine zove se *polimorfizam* (engl. *polimorphism*).

Gore navedena svojstva zajedno sačinjavaju *objektno orijentirani model programiranja*. Evo kako bi se postupak rotiranja trodimenzionalnih likova proveo koristeći objekte:

1. listaj sve objekte redom;
2. zatraži od svakog objekta da se zarotira za neki kut.

Sada glavni program više ne mora voditi računa o tome koji se objekt rotira – on jednostavno samo zatraži od objekta da se zarotira. Sam objekt zna to učiniti ovisno o tome koji lik on predstavlja: kocka će se zarotirati na jedan način, a *kubični spline* na drugi. Također, ako se kasnije program proširi novim tijelima, nije potrebno mijenjati program koji rotira sve objekte – dovoljno je samo u novododanom objektu definirati operaciju rotacije.

Dakle, ono što C++ jezik čini vrlo pogodnim jezikom opće namjene za izradu složenih programa jest mogućnost jednostavnog uvođenja novih tipova te naknadnog dodavanja novih operacija.

Razliku između proceduralnog i objektno orijentiranog modela mogli bismo poistovjetiti s razlikom između šahovskog meča i bitke na bojnom polju. U oba slučaja sukobljavaju se dvije vojske i tijekom borbe obje strane gube sudionike, a pobjednik je ona strana kojoj ključne figure ostanu netaknute. Međutim, u šahu igrači povlače poteze tako da osobno svaku figuru pomiču prema definiranim pravilima. S druge strane, u stvarnoj bici zapovjednici samo izdaju naredbe koje vojnici izvršavaju. Zapovjednik ne mora znati kako se puca iz pojedine puške ili gađa iz topa; on samo mora znati kada lijevo krilo treba krenuti u napad te izdati odgovarajuće zapovijedi. Svaki pojedini vojnik će shodno toj naredbi izvesti odgovarajuće radnje u skladu sa svojim položajem i mogućnostima.

1.3. Usporedba s C-om

Mnogi okorjeli C programeri, koji sanjaju strukture i dok se voze u tramvaju ili razmišljaju o tome kako će svoju novu rutinu riješiti pomoću pokazivača na funkcije, dvoume se oko toga je li C++ doista dostojan njihovog kôda: mnogi su u strahu od nepoznatog jezika te se boje da će im njihov supermunjeviti program za zbrajanje dvaju jednoznamenkastih brojeva na novom jeziku biti sporiji od programa za računanje fraktalnog skupa. Drugi se, pak, kunu da je C++ odgovor na sva njihova životna pitanja, te u fanatičnom zanosu umjesto Kristovog rođenja slave rođendan gospodina Stroustrupa.

Moramo odmah razočarati obje frakcije: niti jedna nije u pravu te je njihovo mišljenje rezultat nerazumijevanja nove tehnologije. Kao i sve drugo, objektna tehnologija ima svoje prednosti i mane, a također nije svemoguća te ne može riješiti sve probleme (na primjer, ona vam neće pomoći da opljačkate banku i umaknete Interpolu).

Kao prvo, C++ programi nisu nužno sporiji od svojih C ekvivalenata. U vrijeme adolescencije jezika C++ to je bio čest slučaj kao posljedica neefikasnih prevoditelja – zbog toga se uvriježilo mišljenje kako programi pisani u jeziku C++ daju sporiji izvedbeni kôd. Međutim, kako je rastao broj prevoditelja na tržištu (a time i međusobna konkurencija), kvaliteta izvedbenog kôda koju su oni generirali se poboljšavala. Štoviše, današnji prevoditelji često bolje daju efikasniji izvedbeni kôd iz C++ izvornog koda nego iz ekvivalentnog C kôda. Osim toga, jezik C++ sadrži neke konstrukcije, kao što su umetnute (engl. *inline*) funkcije, koje mogu znatno pospješiti brzinu konačnog izvedbenog koda. Ipak, u pojedinim slučajevima izvedbeni kôd dobiven iz C++ izvornog kôda doista može biti sporiji, a na programeru je da shvati kada je to dodatno usporenje prevelika smetnja da bi se toleriralo.

Nadalje, koncept klase i enkapsulacija uopće ne usporavaju dobiveni izvedbeni program. Dobiveni strojni kôd bi u mnogim slučajevima trebao biti potpuno istovjetan onome koji će se dobiti iz analognog C programa. Funkcijski članovi pristupaju podatkovnim članovima objekata preko pokazivača, na sličan način na koji to korisnici proceduralne paradigme čine ručno. No C++ kôd će biti čitljiviji i jasniji te će ga biti lakše napisati i razumjeti. Ako pojedini prevoditelj i daje lošiji izvedbeni kôd, to je posljedica lošeg prevoditelja, a ne mana jezika.

Također, korištenje nasljeđivanja ne usporava dobiveni kôd ako se ne koriste *virtualni funkcijski članovi* i *virtualne osnovne klase*. Nasljeđivanje samo uštеđuje programeru višestruko pisanje kôda te olakšava ponovno korištenje već napisanih programskih odsječaka.

Virtualne funkcije i virtualne osnovne klase usporavaju program. Virtualne funkcije se izvode tako da se prije poziva konzultira posebna tablica, pa je jasno da će poziv svake takve funkcije biti nešto sporiji. Također, članovima virtualnih osnovnih klasa se redovito pristupa preko jednog pokazivača više. No, usporenje je u većini realnih slučajeva neprimjetno i zanemarivo u odnosu na koristi koje korištenje virtualnih funkcija donosi. Na programeru je da odredi hoće li koristiti „inkriminirana" svojstva jezika ili ne. Da bi se precizno ustanovilo kako djeluje pojedino svojstvo jezika na izvedbeni kôd, nije dobro nagađati i kriviti C++ za loše performanse, već izmjeriti vrijeme izvođenja te locirati problem.

Sagledajmo još jedan aspekt: asembler je jedini jezik u kojemu programer točno zna što se dešava u pojedinom trenutku prilikom izvođenja programa. Kako je programiranje u asembleru bilo složeno, razvijeni su viši programski jezici koji to olakšavaju. Prevedeni C kôd također nije maksimalno brz – posebno optimiran asemblerski kôd će sigurno dati bolje rezultate. No pisanje takvog kôda danas jednostavno nije moguće: problemi koji se rješavaju su ipak previše složeni da bi se mogli rješavati tako da se pazi na svaki ciklus procesora. Danas, kada na izradi programa nerijetko radi istovremeno desetak i više programera, dizajneri programa prvo moraju definirati module (objekte), njihovu funkcionalnost i specificirati njihova sučelja. Tek na osnovu tih specifikacija

pojedini programeri razvijaju module i testiraju ih, a na kraju dolazi povezivanje modula i testiranje cjelokupnog programa. Ukoliko su sučelja na početku bila dobro definirana, povezivanje modula će biti vrlo jednostavno i program će skladno raditi. Proces se može usporediti s izgradnjom naselja: prvo urbanisti definiraju globalni raspored naselja te gabarite i funkcionalnost koje pojedine zgrade moraju zadovoljavati: okvirne dimenzije, gdje će se nalaziti ulaz, hoće li to biti stambena ili poslovna zgrada, vrtić ili škola. Tek potom arhitekti na osnovu tih postavki projektiraju pojedine zgrade. U tom slučaju ne može se dogoditi da se na četverokatnu zgradu naslanja prizemnica, a ispred njih uopće nema nogostupa niti dovoljno parkirališnog prostora, jer se nasuprot, tik uz kolnik, nalazi neboder od osam katova.

Možda će konačni izvedbeni kôd biti sporiji i veći od ekvivalentnog C kôda, ali će jednostavnost njegove izrade omogućiti da dobiveni program bude bolji po nizu drugih karakteristika: bit će jednostavnije izraditi program koji će biti lakši za korištenje, s više mogućnosti i slično. Uostalom, manje posla – veća zarada – raj zemaljski! Osim toga, danas je često vrlo važno bržim razvojem preteći konkurenciju i prvi izaći na tržište, diktirajući uvjete (i cijene). Što vrijedi programeru ili firmi pisati program u C-u ili assembleru zato da bi dobili maksimalnu brzinu izvođenja, ako će se na tržištu s gotovim proizvodom pojaviti nekoliko mjeseci ili godina kasnije od konkurencije, kada će tržište već biti zasićeno sličnim, neznatno sporijim proizvodima? Nova računala s većim mogućnostima su sposobna učiniti gubitak performansi beznačajnim u odnosu na dobitak u brzini razvoja programa. Tehnologija ide naprijed: dok se gubi neznatno na brzini i memorijskim zahtjevima, dobici su višestruki.

Naravno, C++ nije svemoćan. Korištenje objekata neće napisati pola programa umjesto vas: ako želite provesti crtanje objekata u tri dimenzije i pri tome ih realistično osjenčati, namučit ćete se pošteno koristite li C ili C++. To niti ne znači da će poznavanje objektne tehnologije jamčiti da ćete ju i ispravno primijeniti: ako se ne potrudite prilikom dizajniranja objekata te posao ne obavite u duhu objektnog programiranja, neće biti ništa od ponovne iskoristivosti kôda. Čak i ako posao obavite ispravno, to ne znači da jednog dana nećete naići na problem u kojem će jednostavno biti lakše zaboraviti sve napisano i početi „od jajeta”.

Ono što vam objektna tehnologija pruža jest mogućnost da manje pažnje obratite jeziku i načinu na koji ćete svoju misao izraziti, a usredotočite se na ono što zapravo želite učiniti. U već spominjanom slučaju trodimenzionalnog crtanja objekata to znači da ćete manje vremena provesti razmišljajući gdje ste pohranili podatak o položaju kamere koji vam baš sad treba, a više ćete razmišljati o tome kako da ubrzate postupak sjenčanja ili kako da ga učinite realističnijim.

Objektna tehnologija je pokušala dati odgovore na neke potrebe ljudi koji rješavaju svoje zadatke računalom; na vama je da procijenite koliko je to uspješno, a u svakom slučaju da prije toga pročitate ovu knjigu do kraja i preporučite ju prijateljima, naravno. *Jer tak' dobru i guba knjigu niste vidli već sto godina i baš vam je bilo fora ju čitat.*

1.4. Usporedba s Javom

Nakon što je izašlo prvo izdanje knjige, jedan od češćih komentara je bio: „A zašto (radije) niste napisali knjigu o Javi?”. Odgovor je vrlo jednostavan: da smo onda napisali knjigu o Javi, ta knjiga bi već nakon godinu dana bila zastarjela! Kada smo krajem 1995. godine počeli pisati knjigu, Java je bila tek u povojima (u svibnju te godine objavljena je prva *alfa* verzija Jave).

Budući da su u međuvremenu i C++ i Java prošli kroz niz dopuna i izmjena, nezahvalno je raditi isključive usporedbe tipa „ovo je puno bolje napravljeno u jeziku A nego u jeziku B”. Pokušajmo samo naznačiti koje su značajnije razlike između Jave i jezika C++; konačne zaključke prepuštamo čitatelju.

1.4.1. Java je potpuno objektno orijentirani programski jezik

To znači da se sve operacije odvijaju isključivo kroz objekte, odnosno preko njihovih funkcijskih članova (metoda). Stoga je programer od početka na neki način prisiljen razmišljati na objektno orijentirani način. S druge strane, jezik C++ omogućava pisanje i proceduralnog kôda. Iako se nekome može učiniti kao prednost, ovo je zamka u koju vrlo lako mogu upasti početnici, posebice ako prelaze sa nekog proceduralnog programskog jezika, kao što je jezik C. Naime, ako usvoji proceduralni način razmišljanja, programer će se teško „prešaltati” na objektni pristup i iskoristiti sve njegove pogodnosti. Ovo je danak što ga jezik C++ plaća zbog insistiranja na kompatibilnosti s jezikom C. Stvaratelji jezika C++ (uključujući i gospodina Stroustrupa) nisu željeli izmisliti potpuno novi jezik koji bi iziskivao da se sve aplikacije prepišu u tom novom jeziku, već su nastojali da postojeći izvorni kôdovi mnoštva aplikacija pisanih u jeziku C (ne zaboravimo da je jezik C osamdesetih godina XX stoljeća bio najrasprostranjeniji programski jezik) budu potpuno združivi s novim jezikom i da se ti kôdovi bez poteškoća mogu prevesti na prevoditeljima za jezik C++. Ovakav pristup podrazumijevao je mnoštvo kompromisa, ali je omogućio tisućama programera i programskih kuća postepeni i bezbolan prijelaz s jezika C na jezik C++. To je značajno doprinijelo popularnosti i općem prihvaćanju jezika C++.

S druge strane, Java je potpuno novi programski jezik, neopterećen takvim nasljeđem – zbog toga su neke stvari riješene daleko elegantnije nego u jeziku C++. Uostalom, Javu je stvorila grupa C++ programera koji su bili frustrirani nedostacima jezika C++.

Ipak, napomenimo da je i u Javi moguće napisati proceduralni kôd koristeći samo jednu klasu i/ili definirajući sve funkcijske i podatkovne članove statičkima. Ovo je nerijetko slučaj kada se Jave dohvati programer naučen na proceduralni način razmišljanja. Jedino opravdanje za ovakvu zloupotrebu jezika jest neznanje.

1.4.2. Java izvedbeni kôd može se izvoditi na bilo kojem računalu

Java izvorni kôd se ne prevodi u strojni kôd matičnog procesora (*matični kôd*, engl. *native code*) nego u poseban, tzv. *Java binarni kôd* (engl. *Java bytecode*) kojeg *Java virtualni stroj* (engl. *Java Virtual Machine*, *JVM*) interpretira i izvodi. Prednost ovakvog

pristupa jest da se prevedeni Java kôd (teoretski) može izvršavati na bilo kojem računalu s bilo kojim procesorom i operacijskim sustavom – uostalom, ovo je jedan od udarnih reklamnih slogana za Javu. Ovo ipak vrijedi uz neke ograde!

Kao prvo, na tom stroju mora postojati i „vrtiti se" (odgovarajući) virtualni stroj. Za popularnije platforme, poput *Windows* ili *Linux* platformi postoji zadnja verzija virtualnog stroja, no za neke egzotične platforme mogu iskrsnuti problemi. Uz to, da bi program pisan u Javi bio stvarno prenosiv na bilo koju platformu, ne smije sadržavati operacije specifične za određenu platformu. Na primjer, operacije vezane uz datotečni sustav ili grafičko sučelje mogu se značajno razlikovati od platforme do platforme.

Osim toga, Java binarni kôd neće biti optimiziran za dotični procesor. Na primjer, velika većina današnjih procesora ima ugrađene instrukcije za operacije s realnim brojevima koje omogućavaju da se dijeljenje dva realna broja izvede u svega nekoliko taktova procesora. Budući da se Java binarni kôd mora izvoditi i na računalima s procesorima koja nemaju ugrađene instrukcije za realne brojeve, izvođenje takvih operacija bit će općenito sporije.

Zbog toga, kao i zbog činjenice da se Java binarni kôd interpretira, a ne izvodi izravno, Java programi su i do desetak puta sporiji od ekvivalentnih programa prevedenih iz nekog drugog jezika. Doduše, novije verzije Java virtualnih strojeva pružaju mogućnost *pravovremenog prevođenja* (engl. *Just-In-Time compilation*), kojim se prilikom pokretanja programa Java binarni kôd prevodi u matični strojni kôd i kao takav izvodi. Brzina izvođenja takvog kôda je usporediva s brzinom matičnog izvedbenog kôda. Međutim, taj kôd treba prvo prevesti, što znači da se dio vremena namijenjenog izvođenju programa troši na prevođenje iz Java binarnog u matični izvedbeni kôd. Za programe u kojima se veći dio kôda izvodi samo jednom to može predstavljati značajni gubitak vremena, odnosno usporenje programa.

Što je s prenosivošću kôda pisanog u jeziku C++? Budući da se program pisan u jeziku C++ prevodi u strojni kôd, prilikom prevođenja se mora navesti za koju platformu se taj kôd generira – taj izvedbeni kôd je neupotrebljiv na drugim platformama. Trebamo li isti program prevesti tako da se može izvoditi na Windowsima i na Linuxu, morat ćemo ga posebno prevesti za svaku od tih platformi.

Programski kôd često poziva funkcije iz programskog sučelja operacijskog sustava što omogućava vrlo brzo izvođenje programa i maksimalno korištenje sklopovskih mogućnosti računala. No, kako se programska sučelja razlikuju za pojedine platforme, pisanje programa koji bi omogućio stvaranje izvedbenog kôda za različite platforme može biti vrlo mukotrpno. Srećom, postoje biblioteke (poput biblioteke Boost) koje omogućavaju jednostavniju komunikaciju s različitim platformama. No, za razliku od virtualnog stroja koji predstavlja „međusloj" tijekom izvođenja, multiplatformne biblioteke u jeziku C++ su međusloj koji se koristi tijekom prevođenja programa. Tijekom izvođenja su one potpuno transparentne pa je njihov utjecaj na brzinu zanemariv.

1.4.3. Java nema pokazivača

Davno su prošla vremena kada su se na sâm spomen riječi „pokazivač" (engl. *pointer*) zatvarali prozori i zaključavala vrata, a djecu tjeralo na spavanje (i kada se osoba koja je

javno izjavila da je napravila funkciju koja vraća pokazivač na neki objekt smatrala genijalnim ekscentrikom koji i usred ljeta nosi vunenu kapu i duge gaće). Međutim, još i danas su pokazivači baba-roga kojom se plaše programeri-žutokljunci. Stoga će takvi aklamacijom prihvatiti istrjebljenje pokazivača.

Doduše, zavirimo li „pod haubu" vidjet ćemo da se u Javi svi objekti osim nekolicine ugrađenih, dohvaćaju preko pokazivača[1] – ono što je dobro jest da nema pretvorbi između pokazivača i objekata koje omogućavaju raznorazne (obično nenamjerne) „hakeraje" i redovito završavaju rušenjem ili blokiranjem programa. Iako su pokazivači jedan od najčešćih uzroka pogrešaka u početničkim C/C++ programima, mnogi iskusniji programeri znaju koliko je neke stvari teže napraviti bez pokazivača na funkcije ili funkcijske članove.

1.4.4. Java nema višestrukog nasljeđivanja

Nasljeđivanje omogućava kreiranje korisnički definiranih tipova podataka koristeći i proširujući osobine već postojećih tipova. Često je poželjno naslijediti osobine različitih tipova – tada se u jeziku C++ primjenjuje višestruko nasljeđivanje. Kao primjer, uzmimo tipku u nekoj aplikaciji koja umjesto standardnog oblika (tipka sa sjenčanjima i tekst) ima oblik bitmapirane slike. Tipična primjena takve tipke bi bila u pregledniku datoteka sa slikama – umjesto da izlista imena datoteka taj preglednik će prikazati male sličice, a pritiskom na tipku otvorit će se aplikacija u kojoj sliku možemo obrađivati. Naša tipka očito ima svojstva obične tipke (reagira na klik miša i u tom slučaju pokreće neku akciju), kao i svojstva bitmapirane slike (mora se znati iscrtati). Umjesto da prepišemo cjelokupan kôd oba tipa objekta, bit će dovoljno naslijediti njihova svojstva, eventualno modificirajući neka od njih.

U Javi nema višestrukog nasljeđivanja implementacije. Kako se višestruko nasljeđivanje ne koristi često, tvorci jezika Java smatrali su da ga nema smisla dozvoliti, posebice uzevši u obzir probleme koje višestruko nasljeđivanje može prouzročiti sustavu za skupljanje smeća. Međutim, Java dozvoljava višestruko nasljeđivanje sučelja. To zahtijeva nešto drugačiji stil programiranja koji je orijentiran prema sučeljima objekata.

1.4.5. Java ima ugrađeni sustav za automatsko upravljanjem memorijom

U jeziku C++ programer mora eksplicitno navesti u kôdu da želi osloboditi memoriju koju je prethodno zauzeo za neki objekt. Propusti li to napraviti, tijekom izvođenja programa memorija će se „trošiti", smanjujući raspoloživu memoriju za ostatak programa ili za ostale programe. Napravi li pak to prerano u kôdu, uništit će objekt koji bi eventualno mogao kasnije zatrebati. Curenje memorije ili prerano uništenje objekata je vrlo česti uzrok „blokiranja" rada programa ili cijelog računala. Dobar C++ programer mora

[1] U terminologiji Jave, umjesto *pokazivača* (engl. *pointer*) koristi se naziv *referenca* (engl. *reference*) iako se radi o istoj stvari. Budući da su pokazivači bili prilično ozloglašeni među programerima, autori Jave su odabrali drugi naziv da ne bi u samim počecima odvratili programere. Valja napomenuti da reference u Javi nisu isto što reference u jeziku C++.

imati pod kontrolom gdje je rezervirao prostor za neki objekt i gdje će taj prostor osloboditi.

U Javi se sam sustav brine o oslobađanju memorije koja više nije potrebna – *sustav za automatsko upravljanje memorijom* (engl. *garbage collector* - sakupljač smeća, smetlar) ugrađen je u Java virtualni stroj i automatski se pokreće po potrebi. Naravno da će se za sve one koji su iskusili gore opisane probleme u jeziku C++ ovo učiniti kao zemlja Dembelija. Doduše, za jezik C++ postoje komercijalne i besplatne biblioteke koje ugrađuju mehanizme za skupljanje smeća u kôd, međutim činjenica je da ono nije ugrađeno u sam izvedbeni sustav.

S druge strane, mnogi programeri imaju zamjerku na automatsko sakupljanje smeća jer sustav sam procjenjuje kada treba to sakupljanje započeti i nerijetko se događa da to bude upravo u trenutku kada vaš program obavlja neku zahtjevnu, vremenski kritičnu operaciju te će sakupljanje smeća usporiti izvođenje programa. Zato i kod ugrađenog automatskog sustava za sakupljanje smeća treba često paziti da se novi objekti u memoriji ne stvaraju neracionalno i bez ozbiljnije potrebe.

Štoviše, budući da programer nema nadzora nad uništavanjem, ne može eksplicitno uništiti objekt te ne može kontrolirati redoslijed koji se objekti uništavaju.

1.4.6. Java ne podržava preopterećenje operatora

Preopterećenje operatora omogućava da se funkcionalnost operatora može proširiti i za korisnički definirane podatke. Primjerice, operator zbrajanja + definiran je za ugrađene tipove podataka kao što su cijeli ili realni brojevi i znakovni nizovi. No, ako uvedemo kompleksne brojeve kao svoj novi tip podataka, realno je za očekivati da ćemo poželjeti zbrajanje kompleksnih brojeva izvoditi pomoću operatora +. Bez mehanizma preopterećenja operatora to nije moguće, već smo prisiljeni koristiti nečitljiviju sintaksu preko poziva funkcijskog člana.

1.4.7. U Javi su operacije s decimalnim brojevima lošije podržane

Izvorno je Java bila koncipirana uz pretpostavku da ju većina korisnika neće upotrebljavati za sofisticirane numeričke proračune. Budući da je osnovna misao vodilja autora Jave bila prenosivost kôda, nisu do kraja implementirani svi zahtjevi koje postavlja IEEE 754 standard za računanje s decimalnim brojevima te nisu do kraja iskorištene sve sklopovske mogućnosti nekih procesora (npr. Intelovih) koji imaju ugrađene instrukcije za operacije s decimalnim brojevima. Rezultat toga je i nešto sporije izvođenje takvih operacija. No, kako je u specifikaciji jezika Java zapisano, prosječni korisnik ove nedostatke najvjerojatnije neće nikada primijetiti.

1.4.8. Java ima na raspolaganju ogromnu biblioteku

Java okruženje (engl. *Java Platform*) osim virtualnog stroja zaduženog za izvođenje programa, sadrži i vrlo bogatu biblioteku od nekoliko tisuća gotovih klasa i sučelja koji podržavaju čitav niz operacija uključujući rad s datotekama i mrežom, matematičke

funkcije, grafičke operacije, rukovanje bazama podataka, obradu teksta, kriptografiju. Sve te klase su dostupne programeru prilikom pisanja kôda, ali i korisniku tijekom izvođenja programa na odredišnom računalu. Upravo ta bogata biblioteka klasa je jedan od glavnih razloga popularnosti Jave, jer programer ne mora sam pisati često banalne operacije, niti mora tražiti i kupovati neku komercijalnu biblioteku.

C++ je u svoj prvi standard iz 1998. godine (poznat pod oznakom *C++98*), osim standardne biblioteke funkcija naslijeđene iz jezika C, uključio i standardnu biblioteku predložaka (*STL*) koja je podržavala rad sa skupovima podataka i generičke operacije. Za specijalizirane namjene, programeri su bili prisiljeni kupovati komercijalne proizvode, koristiti malobrojne besplatne biblioteke ili ih sami pisati. Srećom, 1998. godine počeo je razvoj *Boost* biblioteke, besplatne i javno dostupne biblioteke klasa koja, poput biblioteke u Javi, podržava široki spektar operacija. Razvoj te biblioteke je bio tako uspješan da je njen veliki dio ušao u sljedeću verziju standarda jezika C++ prihvaćenu 2011. godine, poznatu pod oznakom *C++11*.

1.4.9. Java je jezik u „vlasništvu" jedne tvrtke

Iako su prevoditelj za Javu i Java virtualni stroj besplatni za individualne korisnike, činjenica jest da su oni vlasništvo i pod kontrolom jedne tvrtke. To znači da neke komponente morate licencirati, a ako ta tvrtka jednog dana propadne, ne postoji jamstvo da će netko preuzeti cjelokupnu tehnološku podršku. Doduše, krajem 2006. godine tvrtka Sun (pod čijim je okriljem Java stvorena) je „otvorila" kôd u namjeri da ga učini slobodnim i javno dostupnim programskim kôdom (engl. *free and open-source software*). U međuvremenu, tvrtku Sun je kupila tvrtka Oracle i nastavila s održavanjem i razvojem Jave.

S druge strane, jezik C++ od samih početaka nije u ničijem vlasništvu (ili što bi neki rekli: „opće je društveno dobro"), a njegov razvoj je pod nadzorom međunarodne organizacije za standardizaciju ISO/IEC. Odbor za standardizaciju čini nekoliko desetaka eksperata iz cijelog svijeta i podržan je od najjačih svjetskih softverskih i hardverskih tvrtki.

1.5. Usporedba s jezikom C♯

C♯ je jezik koji je izmislio Microsoft kao odgovor na Javu (a .NET platformu kao odgovor na Java virtualni stroj). Iako imenom podsjeća na C++, jezik C♯ je daleko sličniji Javi, s tom razlikom da C♯ podržava neke stvari kojih u Javi nema. No, osnovne stvari koje su rečene za Javu u prethodnom odjeljku u najvećoj mjeri vrijede i za C♯.

1.6. Ima li smisla učiti jezik C++?

Zbog ponekad komplicirane sintakse, C++ često prati glas vrlo nečitljivog i kompliciranog jezika. Jedan od glavnih uzroka tome je višestruki pristup podacima: podacima možemo pristupati izravno, preko pokazivača (odn. adrese) i preko reference. Ti različiti načini pristupa podacima pružaju fleksibilnost koja nije moguća u drugim jezicima, ali

su nerijetko i uzrok pogreškama koje početnicima znaju zagorčati život. Java i C♯, koji su razvijeni na iskustvima jezika C++, nemaju tako kompliciranu sintaksu i dobrim dijelom podržavaju programske konstrukcije koje podržava jezik C++. Zbog toga je razvoj programa u tim (i sličnim novijim programskim jezicima) brži nego u jeziku C++. Međutim, C++ ima nekoliko bitnih prednosti od kojih ćemo spomenuti najvažnije.

Izvedbeni kôd je u strojnom jeziku i optimiran za odredišno računalo tako da je brzina izvođenja maksimalna. Štoviše, kôd u jeziku C++ može biti napisan tako da izravno barata s memorijom ili sa sklopovljem računala.

Predlošci (engl. *templates*) u jeziku C++ omogućavaju pisanje generičkog kôda koji se može primijeniti na različite tipove podataka. No to nije sve: predlošci se mogu koristiti za automatizirano generiranje kôda. Iako Java i C♯ podržavaju generičke tipove, ta podrška je, u odnosu na ono što pružaju predlošci u jeziku C++, simbolična i služi isključivo za osiguranje tipske sigurnosti (engl. *type-safety*), tj. onemogućava pridruživanje i operacije između međusobno nekompatibilnih tipova.

Zadnja, ali ne najmanja važna prednost jezika C++ jest njegova rasprostranjenost i popularnost. Po svim statistikama, jezik C++ je trenutno (siječanj 2014.) među tri-četiri najpopularnija programska jezika (vidi npr. *http://www.langpop.com*), a zajedno s jezikom C (koji se u tim statistikama vodi zasebno) uvjerljivo vodi.

1.7. Zašto primjer iz knjige ne radi na mom računalu?

Primjeri kôda u knjizi su usklađeni s aktualnim Standardom jezika C++ iz 2011. godine. Testirali smo ih pomoću prevoditelja koji je uglavnom usklađen sa Standardom, ... ali nikad se ne zna. Lako se može dogoditi da se poneki primjer ne može prevesti ili da ne radi ono što je napisano u knjizi zbog pogreške koju smo napravili u pripremi knjige.

Drugi razlog može biti neusklađenost prevoditelja kojeg koristite sa Standardom. Naime, zahvaljujući velikoj popularnosti jezika C, programski jezik C++ se brzo proširio i prilično rano su se pojavili mnogi komercijalno dostupni prevoditelji. Od svoje pojave, jezik C++ se dosta mijenjao, a prevoditelji su „kaskali" za tim promjenama. Zbog toga se često događalo da je program koji se dao korektno prevesti na nekom prevoditelju, bio neprevodiv već na sljedećoj inačici prevoditelja istog proizvođača.

Takva raznolikost i nekompatibilnost prevoditelja nagnala je Američki Nacionalni Institut za Standarde (*American National Standards Institute*, ANSI) da krajem 1989. godine osnuje odbor (s kôdnom oznakom X3J16) čiji je zadatak bio napisati Standard za jezik C++. Osim eminentnih stručnjaka (poput g. Stroustrupa) u radu odbora sudjelovali su i predstavnici svih najznačajnijih softverskih tvrtki. U lipnju 1991. rad odbora je prešao u nadležnost Međunarodne Organizacije za Standarde (*International Standards Organization*, ISO). 14. studenog 1997. godine prihvaćen je ISO standard jezika C++ (ISO/IEC 14882, poznat pod nazivom *C++98*), koji je ratificiran 1998. 2003. godine prihvaćena je ažurirana verzija standarda (*C++03*) koja je uključivala samo ispravke i nije donijela nikakve bitne promjene u jezik.

Odbor za standardizaciju nastavio je raditi na sljedećoj verziji Standarda jezika, koja je prihvaćena u kolovozu 2011. pod nazivom *C++11*. Taj Standard unosi u jezik niz novina koje će biti opisane u ovoj knjizi, ali održava i kompatibilnost prema postojećem

kôdu – sav postojeći kôd (uključujući i primjere iz prethodnih izdanja knjige) bi se i dalje, bez promjena, morao moći prevoditi na novijim prevoditeljima usklađenim sa standardom C++11. Za očekivati je da će proteći još koja godina prije nego svi prevoditelji u potpunosti budu podržavali zadnju inačicu Standarda (osobito se to odnosi na neke „egzotičnije" konstrukte) pa se nemojte iznenaditi ako vam prevoditelj koji koristite otkaže poslušnost i počne prijavljivati pogreške na ispravnom kôdu. U takvim situacijama svakako provjerite dokumentaciju koja dolazi uz prevoditelja.

Sljedeća verzija standarda, pod oznakom *C++14* trebala bi se pojaviti tijekom 2014. godine, a uključivala bi tek neznatna poboljšanja i ispravke pogrešaka.

Na kraju spomenimo da osim Standardiziranog C++-a, postoji i takozvani *C++/CLI* (od engl. *Common Language Infrastructure*). To je Microsoftovo proširenje jezika C++ sintaksom koja omogućava korištenje .NET okruženja (engl. *.NET Platform*). Budući da ova proširenja izlaze iz okvira standardnog jezika C++, nećemo se njima baviti.

1.8. Literatura

Tijekom pisanja koristili smo literaturu navedenu na kraju knjige. Ponegdje se pozivamo na neku od tih knjiga, navodeći pripadajuću oznaku u uglatoj zagradi. Od svih navedenih knjiga, „najreferentniji" je trenutno aktualni standard jezika C++ iz 2011. godine; može se kupiti kod Američkog Nacionalnog Instituta za Standarde (ANSI) ili kod Međunarodne Organizacije za Standardizaciju (ISO). Na Internetu se mogu naći besplatne radne verzije (engl. *draft*) Standarda[1], dostupne u PDF (*Acrobat Reader*) formatu.

Sâm Standard ne preporučujemo za učenje jezika C++, a također vjerujemo da neće trebati niti iskusnijim korisnicima – Standard je prvenstveno namijenjen proizvođačima softvera; svi *prevoditelji* (engl. *compilers*) bi se trebali ponašati u skladu s tim standardom. No, smatramo da bi svaki „ozbiljni" C++ programer morao imati knjigu B. Stroustrupa: *The C++ Programming Language* (četvrto izdanje!) – to je uz Standard svakako najreferentnija knjiga u kojoj ćete naći odgovor na vjerojatno svaki detalj vezan uz jezik C++ (knjigu ne preporučujemo za učenje jezika). Uz nju, najtopliju preporuku zaslužuje knjiga S. Lippmana, J. Lajoie, B. E. Moo: *C++ Primer* (peto izdanje) u kojoj su neki detalji jasnije objašnjeni nego u Stroustrupovoj knjizi.

Zadnje poglavlje ove knjige posvećeno je principima objektno orijentiranog programiranja. Literatura vezana za to područje nije striktno vezana uz jezik C++, tako da je u popisu literature navedena zasebno. S navedenog popisa, svaki ozbiljniji programer trebao bi pročitati naslove [Meyer97] i [Gamma95].

Uz to, mnoštvo odgovora te praktičnih i korisnih rješenja može se naći na Internetu. Svakako preporučujemo posjet mrežnim stranicama navedenima u popisu literature.

Nakon što je izašlo prvo izdanje knjige, jedan od najčešćih upita koje smo dobivali jest bio: „Može li se pomoću vaše knjige programirati u (*MS*) *Windowsima*?". Ova knjiga pruža osnove jezika C++ i vjerujemo da ćete se, kada nakon par mjeseci zaklopite zadnju stranu, moći pohvaliti da ste naučili jezik C++. Za pisanje *Windows* programa trebaju vam, međutim, specijalizirane knjige koje će vas naučiti kako stečeno znanje

[1] Iako se radi o *radnim verzijama*, one su gotovo u potpunosti identične službenom Standardu.

jezika C++ iskoristiti za tu namjenu. Od takvih knjiga svakako vam preporučujemo (to nije samo naša preporuka!) sljedeće dvije knjige, točno navedenim redoslijedom:

- Jeff Prosise: *Programming Windows with MFC (2nd Edition)*, Microsoft Press, 1999, ISBN 1-57231-695-0
- Jeffrey Richter: *Windows via C/C++*, Microsoft Press, 2007, ISBN 978-0735663770.

Iako se radi o dosta starim naslovima, osnovni koncepti se nisu mijenjali i te knjige su u velikoj mjeri još uvijek aktualne.

I na kraju, preporučujemo naslov koji smo koristili za neke od algoritama u primjerima:

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest: *Introduction to Algorithms*, The MIT Press, Cambridge, MA, 1999, ISBN 0-262-03141-8

1.9. Zahvale

Zahvaljujemo se svima koji su nam izravno ili posredno pomogli pri izradi ove knjige. Posebno se zahvaljujemo Branimiru Pejčinoviću (*Portland State University*) koji nam je omogućio da dođemo do Nacrta ANSI C++ standarda iz 1997. godine, Vladi Glaviniću (*Fakultet elektrotehnike i računarstva*) koji nam je omogućio da dođemo do knjiga [Stroustrup94] i [Carroll95] te nam je tijekom pripreme za tisak dao na raspolaganje laserski pisač, te Zdenku Šimiću (*Fakultet elektrotehnike i računarstva*) koji nam je, tijekom svog boravka u SAD, pomogao pri nabavci dijela literature.

Posebnu zahvalu upućujemo Ivi Mesarić koja je pročitala cijeli rukopis te svojim korisnim i konstruktivnim primjedbama znatno doprinijela kvaliteti iznesene materije. Također se zahvaljujemo Zoranu Kalafatiću (*Fakultet elektrotehnike i računarstva*) i Damiru Hodaku koji su čitali dijelove rukopisa i dali na njih korisne opaske.

Boris se posebno zahvaljuje gospođama Šribar na tonama kolača pojedenih za vrijeme dugih, zimskih noći čitanja i ispravljanja rukopisa, a koje su ga koštale kure mršavljenja.

I naravno, zahvaljujemo se Bjarne Stroustrupu i dečkima iz *AT&T*-a što su izmislili C++, naš najdraži programski jezik. Bez njih ne bi bilo niti ove knjige (ali možda bi bilo slične knjige iz FORTRAN-a).

1.9.1. Zahvale uz drugo i treće izdanje

U prvom redu zahvaljujemo se svima koji su kupili prvo izdanje knjige koje je rasprodano u relativno kratkom vremenu – bez njih ne bi bilo drugog izdanja knjige. A onima koji su fotokopirali ili skenirali knjigu poručujemo da će se posthumno peći u vječnoj vatri *Xeroxovih* i *Canonovih* tonera, a djeca će im izgledati kao najlošiji skenovi na prastarim UMAX skenerima (naravno, svoje grijehe će okajati uništе li odmah fotokopiju/sken i kupe primjerak knjige u knjižari).

Zahvale upućujemo i svima koji su dali prijedloge ili nas upozorili na pogreške u prvom i drugom izdanju.

2. Vrijeme je da se krene...

„Što je to naredba?”
„Da ugasim svjetiljku. Dobra večer.” I on je ponovo upali.
„Ne razumijem” reče mali princ.
„Nemaš što tu razumjeti” reče noćobdija. „Naredba je naredba. Dobar dan.” I on ugasi svoju svjetiljku.

Antoine de Saint-Exupéry (1900–1944), „Mali princ”

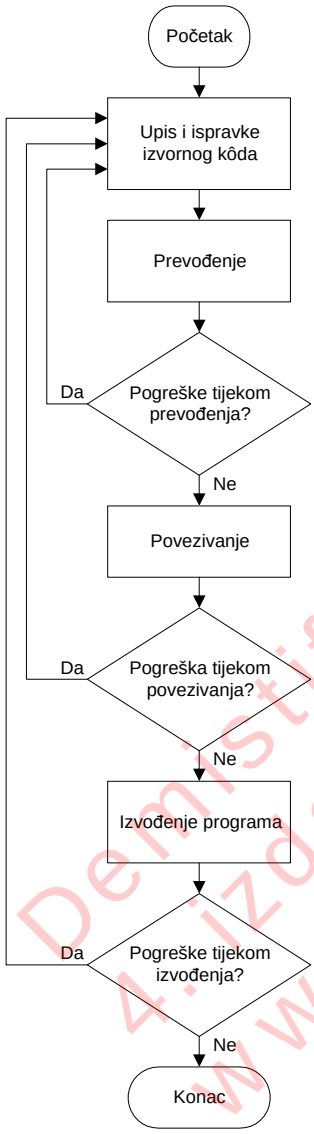
U prvom poglavlju proletjet ćemo kroz osnovne pojmove vezane uz programiranje i upoznat ćemo se sa strukturom najjednostavnijih programa pisanih u programskom jeziku C++. Prvi odjeljak prvenstveno je namijenjen čitateljicama i čitateljima koji nisu nikada pisali programe u bilo kojem višem programskom jeziku i onima koji su to možda radili, ali je „od tada prošla čitava vječnost”. Slijedi odjeljak u kojem je opisano što čitateljici ili čitatelju sve treba za upis, prevođenje i testiranje primjera. Odjeljkom 2.3 počinje „konkretni” dio knjige.

2.1. Što je program i kako ga napisati

Elektronička računala su danas postala pribor kojim se svakodnevno koristimo kako bismo si olakšali posao ili se zabavili. Istina, točnost prvog navoda će mnogi poricati, ističući kao protuprimjer činjenicu da im je za podizanje novca prije trebalo znatno manje vremena nego otkad su šalteri u banci kompjuterizirani. Ipak, činjenica je da su mnogi poslovi danas nezamislivi bez računala; u krajnjoj liniji, dokaz za to je knjiga koju upravo čitate koja je u potpunosti napisana pomoću računala.

Samo računalo, čak i kada se uključi u struju, nije kadro učiniti ništa korisno. Na današnjim računalima se ne može čak ni zagrijati prosječni ručak, što je inače bilo moguće na računalima s elektronskim cijevima. Ono što vam nedostaje jest pamet neophodna za koristan rad računala: programi, programi, … mnoštvo programa. Pod programom tu podrazumijevamo niz naredbi u strojnom jeziku koje procesor u vašem računalu izvodi i shodno njima obrađuje podatke, provodi matematičke proračune, ispisuje tekstove, iscrtava krivulje na zaslonu ili gađa vaš avion F-16 raketama srednjeg dometa. Prilikom pokretanja s diska, DVD-ROM-a ili USB memorije, program se učitava u radnu memoriju računala i procesor počinje s mukotrpnim postupkom njegova izvođenja.

Programi koje pokrećete na računalu su u *izvedbenom obliku* (engl. *executable*), razumljivom samo procesoru vašeg (i njemu sličnih) računala, sretnim procesorovim roditeljima negdje u Silicijskoj dolini i nekolicini zadrtih hakera širom svijeta koji još uvijek programiraju u strojnom kôdu. U suštini se strojni kôd sastoji od nizova binarnih znamenki: nula i jedinica. Budući da su današnji programi tipično veličine više mega-



Slika 2.1. Tipičan razvoj programa

bajta, naslućujete da ih autori nisu pisali izravno u strojnom kôdu (kamo sreće da je tako – ne bi *Microsoft* tek tako svake dvije godine izbacivao nove *Windowse!*).

Gotovo svi današnji programi se pišu u nekom od *viših programskih jezika* (FORTRAN, BASIC, Pascal, C) koji su donekle razumljivi i ljudima (barem onima koji imaju nekog pojma o engleskom jeziku). Naredbe u tim jezicima se sastoje od mnemonika. Kombiniranjem tih naredbi programer slaže *izvorni kôd* (engl. *source code*) programa, koji se pomoću posebnih programa *prevoditelja* (engl. *compiler*) i *povezivača* (engl. *linker)* prevodi u izvedbeni kôd. Prema tome, pisanje programa u užem smislu podrazumijeva pisanje izvornog kôda. Međutim, kako pisanje kôda nije samo sebi svrhom, pisanjem programa u širem smislu uključuje i prevođenje, odnosno povezivanje programa u izvedbeni kôd. Stoga možemo govoriti o četiri faze izrade programa:

1. pisanje izvornog kôda
2. prevođenje izvornog kôda,
3. povezivanje u izvedbeni kôd te
4. testiranje programa.

Da bi se za neki program moglo reći da je potpuno zgotovljen, treba uspješno proći kroz sve četiri faze.

Kao i svaki drugi posao, i pisanje programa iziskuje određeno znanje i vještinu. Prilikom pisanja programera vrebaju Scile i Haribde, danas poznatije pod nazivom pogreške ili *bugovi* (engl. *bug* - stjenica) u programu. Uoči li se pogreška u nekoj od faza izrade programa, izvorni kôd treba doraditi i ponoviti sve prethodne faze. Zbog toga postupak izrade programa nije pravocrtan, već manje-više podsjeća na mukotrpno petljanje u krug. Na slici 2.1 je shematski prikazan cjelokupni postupak izrade programa, od njegova začetka do njegova okončanja. Analizirajmo najvažnije faze izrade programa.

Prva faza programa je pisanje izvornog kôda. U principu se izvorni kôd može pisati u bilo kojem programu za uređivanje teksta (engl. *text editor*), međutim velika većina današnjih prevoditelja i povezivača se isporučuje kao cjelina zajedno s ugrađenim prog-

ramom za upis i ispravljanje izvornog kôda. Te programske cjeline poznatije su pod nazivom *integrirane razvojne okoline* (engl. *integrated development environment*, *IDE*). Danas najpopularnije razvojne okoline za C++ su *Visual Studio* tvrtke Microsoft te besplatni i javno dostupni *Code::Blocks*, *CodeLite*, *Eclipse CDT* i *Dev-C++* neovisnih autora. Nakon što je (prema obično nekritičkom mišljenju programera) pisanje izvornog kôda završeno, on se pohrani u datoteku izvornog kôda na disku. Toj datoteci se obično daje nekakvo smisleno ime, pri čemu se ono za kôdove pisane u programskom jeziku C++ obično proširuje nastavkom `.cpp`, `.cp` ili samo `.c`, na primjer `pero.cpp`. Nastavak je potreban samo zato da bismo izvorni kôd kasnije mogli lakše pronaći.

Slijedi prevođenje izvornog kôda. U integriranim razvojnim okolinama program za prevođenje se pokreće pritiskom na neku tipku na zaslonu, pritiskom odgovarajuće tipke na tipkovnici ili iz nekog od *izbornika* (engl. *menu*) – ako prevoditelj nije integriran, poziv je nešto složeniji[1]. Prevoditelj tijekom prevođenja provjerava sintaksu napisanog izvornog kôda i u slučaju uočenih ili naslućenih pogrešaka ispisuje odgovarajuće poruke o pogreškama ili upozorenja. Pogreške koje prijavi prevoditelj nazivaju se *pogreškama pri prevođenju* (engl. *compile-time errors*). Nakon toga programer će pokušati ispraviti sve navedene pogreške i ponovo prevesti izvorni kôd – sve dok prevođenje kôda ne bude uspješno okončano, neće se moći pristupiti povezivanju kôda. Prevođenjem izvornog dobiva se datoteka *objektnog kôda* (engl. *object code*), koja se lako može prepoznata po tome što obično ima nastavak `.o` ili `.obj` (u našem primjeru bi to bio `pero.obj`).

Nakon što su ispravljene sve pogreške uočene prilikom prevođenja i kôd ispravno preveden, pristupa se povezivanju (engl. *linking*) objektnih kôdova u izvedbeni. U većini slučajeva objektni kôd dobiven prevođenjem programerovog izvornog kôda treba povezati s postojećim *bibliotekama* (engl. *libraries*). Biblioteke su datoteke u kojima se nalaze već prevedene gotove funkcije ili podaci. One se isporučuju zajedno s prevoditeljem, mogu se zasebno kupiti ili ih programer može tijekom rada sam razvijati. Bibliotekama se izbjegava opetovano pisanje vrlo često korištenih operacija. Tipičan primjer za to je biblioteka matematičkih funkcija koja se redovito isporučuje uz prevoditelje, a u kojoj su definirane sve funkcije poput trigonometrijskih, hiperbolnih, eksponencijalnih i sl. Prilikom povezivanja provjerava se mogu li se svi pozivi kôdova realizirati u izvedbenom kôdu. Uoči li povezivač neku nepravilnost tijekom povezivanja, ispisat će poruku o pogreški i onemogućiti generiranje izvedbenog kôda. Ove pogreške nazivaju se *pogreškama pri povezivanju* (engl. *link-time errors*) – sada programer mora prionuti ispravljanju pogrešaka koje su nastale pri povezivanju. Nakon što se isprave sve pogreške, kôd treba ponovno prevesti i povezati.

Uspješnim povezivanjem dobiva se izvedbeni kôd programa. Taj kôd će raditi upravo ono što je programer zadao naredbama izvornog kôda, no sasvim je moguće da to ne odgovara onome što je izvorno zamislio. Program će sadržavati logičke pogreške. Primjerice, može se dogoditi da program radi pravilno samo za neke podatke, dok se za druge podatke ponaša nepredvidivo. U tom slučaju radi se o *pogreškama pri izvođenju* (engl. *run-time errors*). Da bi program bio potpuno korektan, programer treba istestirati program da bi uočio i ispravio te pogreške, što znači ponavljanje cijelog postupka u

[1] Ovdje nećemo opisivati konkretno kako se pokreću postupci prevođenja ili povezivanja, jer to varira ovisno o *prevoditelju*, odnosno *povezivaču*.

lancu *ispravljanje izvornog kôda - prevođenje - povezivanje - testiranje*. Broj ponavljanja će biti manji što je program jednostavniji i iskustvo programera veće. Međutim, kako raste složenost programa, tako se povećava broj mogućih pogrešaka i cijeli postupak izrade programa neiskusnom programeru može postati mukotrpan.

Za ispravljanje pogrešaka pri izvođenju, programeru na raspolaganju stoje programi za otkrivanje pogrešaka (engl. *debugger* - doslovni prijevod bio bi *istjerivač stjenica* ili *buba*, odnosno *stjeničji terminator*). Radi se o programima koji omogućavaju prekid izvođenja izvedbenog kôda programa koji testiramo na unaprijed zadanim naredbama, izvođenje programa naredbu po naredbu, ispis i promjene trenutnih vrijednosti pojedinih podataka u programu. Najjednostavniji programi za otkrivanje pogrešaka ispisuju izvedbeni kôd u obliku strojnih naredbi. Međutim, većina današnjih naprednih programa za otkrivanje pogrešaka su *simbolički* (engl. *symbolic debugger*) – iako se izvodi prevedeni, strojni kôd, izvođenje programa se prati preko izvornog kôda pisanog u višem programskom jeziku. To omogućava vrlo lagano lociranje i ispravljanje pogrešaka u izvornom kôdu programa. Štoviše, većina integriranih razvojnih okolina u sebi uključuje module za otkrivanje pogrešaka, tako da programer unutar nje ne samo da piše, prevodi i povezuje program, već ga odmah može testirati i ispravljati.

Valja napomenuti da za simboličko *debuggiranje*, prevoditelj u izvedbeni kôd mora uključiti i dodatne podatke, kao što je na primjer veza prema datotekama izvornog kôda. Ti podaci omogućavaju da u svakom koraku izvođenja izvedbenog kôda *debugger* zna koja naredba u izvornom kôdu je taj odsječak izvedbenog kôda generirala. Uočimo riječ „odsječak" u prethodnoj rečenici: prevoditelj svaku pojedinu naredbu u izvornom kôdu razlaže u niz strojnih instrukcija (na slici 2.2 prikazan je jedan primjer) te bi praćenje izvođenja kôda preko strojnih instrukcija (čak i preko njihovih *asemblerskih* mnemonika) bilo izuzetno kompliciranо čak i vičnim programerima. Također, dodatni podaci u izvedbenom kôdu omogućavaju da dobijemo ispis vrijednosti varijable preko imena koji smo za nju koristili ili čak da vrijednost varijable sami promijenimo. Očito je da će zbog tih dodatnih podataka koji su potrebni samo za *debuggiranje*, duljina izvedbenog kôda biti veća. Za konačnu verziju kôda koji se isporučuje korisniku ti se podaci izostavljaju pomoću odgovarajućih opcija u postavkama prevoditelja i povezivača.

Osim pogrešaka, prevoditelj i povezivač redovito dojavljuju i upozorenja. Ona ne onemogućavaju nastavak prevođenja, odnosno povezivanja kôda, ali predstavljaju potencijalnu opasnost. Upozorenja se mogu podijeliti u dvije grupe. Prvu grupu čine upo-

naredba u izvornom kodu	generirani strojni kôd	*asemblerski* mnemonici
`a = 5`	`c7 45 ec 05 00 00 00`	`mov dword ptr [a],5`

`b = a + 3`	`8b 45 ec`
	`83 c0 03`
	`89 45 e0`
		`eax,dword ptr [a]`
		`eax,3`
		`dword ptr [b],eax`

Slika 2.2. Primjer naredbi u višem programskom jeziku i pripadajućeg generiranog strojnog kôda u heksadekadskom prikazu te *asemblerskih mnemonika*

zorenja koja javljaju da kôd nije potpuno korektan. Prevoditelj ili povezivač će zanemariti našu pogrešku i prema svom nahođenju generirati kôd. Drugu grupu čine poruke koje upozoravaju da „nisu sigurni je li ono što smo napisali upravo ono što smo željeli napisati”, tj. radi se o dobronamjernim upozorenjima na zamke koje mogu proizići iz načina na koji smo program napisali. Iako će, unatoč upozorenjima, program biti preveden i povezan (možda čak i korektno), pedantan programer neće ta upozorenja nikada zanemariti – ona često upućuju na uzrok pogrešaka pri izvođenju gotovog programa. Za precizno tumačenje poruka o pogreškama i upozorenja neophodna je dokumentacija koja se isporučuje uz prevoditelj i povezivač.

Da zaključimo: „Što nam dakle treba za pisanje programa u jeziku C++?” Osim računala, programa za pisanje i ispravljanje teksta, prevoditelja i povezivača trebaju vam još samo tri stvari: interes, puno slobodnih popodneva i dooobra knjiga. Interes vjerojatno postoji ako ste s čitanjem stigli čak do ovdje. Slobodno vrijeme će vam trebati da isprobate primjere i da se sami okušate u bespućima C++ zbiljnosti. Jer, kao što stari južnohrvatski izrijek kaže: „Nima dopisne škole iz plivanja”. Stoga ne gubite vrijeme i odmah otkažite sudar dečku ili curi. A za dooobru knjigu…(„*ta-ta-daaam*” – tu sada mi upadamo!).

2.2. Boj ne bije svjetlo oružje, već srce u junaka

Nakon što ste uz hrpu neuvjerljivih isprika uspjeli otkazati sudar, otvorili ste knjigu na ovoj stranici, uključili računalo i zbunjeno zurite u ekran ... Što sada? Što mi još fali? Ponovimo još jednom što nam je neophodno za izradu programa (osnovne upute za korištenje nekih najpopularnijih prevoditelja čitatelj može naći u Prilogu D).

2.2.1. Integrirana razvojna okolina

Za početnika će svakako biti najbolje da instalira neku integriranu razvojnu okolinu koja u sebi već uključuje prevoditelja, povezivač i modul za otkrivanje pogrešaka. Uz najpoznatiju komercijalnu razvojnu okolinu, Microsoftov *Visual Studio*, na raspolaganju su i besplatne razvojne okoline[1]. Microsoft nudi besplatnu inačicu *Visual Studio Express*, koja se od komercijalne verzije razlikuje samo po nekim mogućnostima koje za početnika nisu bitne. Za one koji zaziru od Microsofta ili žele raditi na operacijskom sustavu koji ne spada u skupinu Microsoftovih *Windowsa*, preporučujemo *Code::Blocks* ili *Eclipse* koji omogućavaju razvoj programa na *Windows*, *Linux* i *Mac OS X* operacijskim sustavima. *Code::Blocks* je razvojna okolina napravljena isključivo za pisanje programa u jeziku C++. *Eclipse* je razvojna okolina koja se može koristiti za pisanje programa u čitavom nizu programskih jezika, a za izradu C++ programa neophodan je *CDT* dodatak (engl. *plugin*). Obje razvojne okoline koriste zasebne prevoditelje i module za otkrivanje pogrešaka trećih proizvođača. Ti moduli mogu se zasebno instalirati i konfigurirati, ali su redovito već uključeni u instalacijski program.

[1] Korištenje takvih razvojnih okolina je besplatno samo za osobnu upotrebu; za komercijalnu izradu programa valja pročitati uvjete licenciranja.

2.2.2. Prevoditelj i povezivač

Prevoditelj (engl. *compiler*) je program koji će nam omogućiti da programe pisane u jeziku C++ prevedemo u izvedbeni kôd. Današnji prevoditelji najčešće imaju pridružen povezivač (engl. *linker*), tako da ga ne treba posebno instalirati. Prevoditelji i povezivači se pokreću iz komandne linije, navođenjem neophodnih parametara, što za početnika može biti prilično mukotrpno i podložno pogreškama. Daleko je jednostavnije pokretati ih iz razvojne okoline, pritiskom na neku tipku ili odabirom stavke u izborniku.

Osim Microsoftovog prevoditelja koji se isporučuje uz *Visual Studio*, najpopularniji prevoditelji za programe pisane u jeziku C++ su *GCC* (*GNU Compiler Collection*) i *Clang*. Oba prevoditelja su besplatna, a GCC je vrlo često uključen u instalacije razvojnih okolina, poput spomenutih *Code::Blocks* i *Eclipse*.

Bez obzira hoćete li koristiti neki komercijalni ili neki besplatni prevoditelj, za isprobavanje primjera iz ove knjige važno je da bude usklađen s najnovijim standardom jezika C++ ili da ga barem većim dijelom podržava.

2.2.3. Program za uređivanje teksta

Ako imate integriranu razvojnu okolinu, u njoj je uključen i vrlo moćan urednik teksta. Većina tih urednika podržava sintaksno isticanje (engl. *syntax highlighting*) tako da ključne riječi jezika, imena varijabli, tekstove, komentare prikazuje u različitim bojama. To omogućava uočavanje pogrešaka već tijekom pisanja kôda.

Odlučite li se koristiti prevoditelj iz komandne linije, trebat će vam zasebni urednik teksta (engl. *text editor*) u kojem ćete izvorni kôd upisivati, ispravljati te pohraniti na disk prije pokretanja prevoditelja. Postoje urednici teksta koji podržavaju sintaksno isticanje za jezik C++, a mogu se i konfigurirati da se iz njih pokreće prevođenje programa. U krajnjoj nuždi, za pisanje programa može poslužiti i *Notepad* pod *Windowsima* ili *vi-editor* pod *Linuxom*. Namjeravate li koristiti neki moćniji urednik teksta, pripazite da izvorni kôd pohranjujete kao „čisti" tekst, a ne u nekom posebnom formatu.

2.2.4. Program za otkrivanje pogrešaka

On će vam omogućiti da, kada jednom uspješno prevedete i pokrenete program, lakše otkrijete pogreške u programu. Integrirane razvojne okoline imaju ga već u sebi uključene tako da je moguće vrlo jednostavno pratiti izvođenje kôda, naredbu po naredbu, te u svakom trenutku provjeriti vrijednosti pojedinih podataka.

Uz komandne prevoditelje ponekad dolaze i pripadajući *debuggeri*, koji doduše nemaju raskoš integriranih programa za otkrivanje pogrešaka, ali služe svojoj svrsi.

2.3. Moj prvi i drugi C++ program

Nakon što smo se konačno naoružali svim potrebnim, bez mnogo okolišanja i filozofiranja, napišimo najjednostavniji program u jeziku C++:

```
int main()
{
    return 0;
}
```

Utipkate li nadobudno ovaj program u svoje računalo, pokrenete odgovarajući prevoditelj, te nakon uspješnog prevođenja pokrenete program, na zaslonu računala sigurno nećete dobiti ništa! Nemojte se odmah hvatati za glavu, laćati telefona i zvati svoga dobavljača računala. Gornji program zaista ništa ne radi, a ako nešto i dobijete na zaslonu vašeg računala, znači da ste negdje pogriješili prilikom utipkavanja. Unatoč jalovosti gornjeg kôda, promotrimo ga, redak po redak.

U prvom retku je napisano `int main()`. `main` je naziv za glavnu funkciju u svakom C++ programu (u nekim programskim jezicima glavna funkcija se zove *glavni program*, a sve ostale funkcije *potprogrami*). Izvođenje svakog programa započinje i završava naredbama koje se nalaze u njoj.

Svaki program napisan u C++ jeziku mora imati točno (ni manje ni više) jednu `main()` funkciju.

Pritom valja uočiti da je `main` samo simboličko ime koje daje do znanja prevoditelju koji se dio programa treba prvo početi izvoditi – ono nema nikakve veze s imenom izvedbenog programa koji se nakon uspješnog prevođenja i povezivanja dobiva. Željeno ime izvedbenog programa određuje sam programer: ako se korišteni prevoditelj pokreće iz komandne linije, ime se navodi kao parametar u komandnoj liniji, a ako je prevoditelj ugrađen u neku razvojnu okolinu, tada se ime navodi kao jedna od postavki projekta. Točne detalje definiranja imena izvedbenog imena čitateljica ili čitatelj naći će u uputama za prevoditelj kojeg koristi.

Riječ `int` ispred oznake glavne funkcije ukazuje na to da će `main()` po završetku izvođenja naredbi i funkcija sadržanih u njoj kao rezultat tog izvođenja vratiti cijeli broj (`int` dolazi od engleske riječi *integer* koja znači *cijeli broj*). Budući da se glavni program pokreće iz operacijskog sustava (Linux, Mac OS, MS Windows), rezultat glavnog programa se vraća operacijskom sustavu. Najčešće je to kôd koji signalizira pogrešku nastalu tijekom izvođenja programa ili obavijest o uspješnom izvođenju. Taj kôd će operacijski sustav ili program koji je pozvao naš program, eventualno iskoristiti da bi odabrao daljnje akcije koje treba poduzeti.

Iza riječi `main` slijedi par otvorena-zatvorena zagrada `()`. Unutar te zagrade trebali bi doći opisi podataka koji se iz operacijskog sustava prenose u `main()`. Ti podaci nazivaju se *argumenti* ili *parametri* funkcije. Za funkciju `main()` to su parametri koji se pri pokretanju programa navode u komandnoj liniji iza imena programa, kao na primjer:

```
dir /p /s
```

Ovom naredbom se pokreće program (komanda) `dir` za ispis sadržaja kazala i pritom se predaju dva parametra: `/p` i `/s`. U našem C++ primjeru unutar zagrada nema ništa, što

znači da ne prenosimo nikakve argumente. Tako će i ostati do daljnjega, točnije do poglavlja 8 u kojem ćemo detaljnije obraditi funkcije, a zasebno i funkciju `main()`.

Iza `main()` slijedi otvorena vitičasta zagrada `{`. Ona označava početak *bloka naredbi* u kojem će se nalaziti naredbe glavne funkcije, dok zatvorena vitičasta zagrada `}` u zadnjem retku označava kraj tog bloka . U samom bloku prosječni čitatelj (neupućen u vještinu čitanja tarot karata) uočit će samo jednu naredbu, `return 0`. Tom naredbom glavni program vraća pozivnom programu broj 0, a to je poruka operacijskom sustavu da je program uspješno okončan, štogod on radio.

Uočimo znak `;` (točka-zarez) iza naredbe `return 0`. On označava kraj naredbe te prevoditelju daje do znanja da sve znakove koji slijede interpretira kao novu naredbu.

Znak `;` mora zaključivati svaku naredbu u jeziku C++.

Radi kratkoće ćemo u većini primjera u knjizi izostavljati uvodni i zaključni dio glavne funkcije te ćemo podrazumijevati da oni u konkretnom kôdu postoje.

Pogledajmo još na trenutak što bi se dogodilo da smo kojim slučajem napravili neku pogrešku prilikom upisivanja gornjeg kôda. Recimo da smo zaboravili desnu vitičastu zagradu na kraju kôda:

```
int main()
{
    return 0;
```

Prilikom prevođenja prevoditelj će uočiti da funkcija `main()` nije pravilno zatvorena, te će ispisati poruku o pogreški oblika „Pogreška u prvi.cpp redak *xx*: složenoj naredbi nedostaje } u funkciji main()”. U ovoj poruci je `prvi.cpp` ime datoteke u koju smo naš izvorni kôd pohranili, a *xx* je broj retka u kojem se pronađena pogreška nalazi.

Zaboravimo li napisati naredbu `return`:

```
int main()
{
}
```

neki prevoditelji će javiti upozorenje oblika „Funkcija bi trebala vratiti vrijednost”. Iako se radi o pogreški, prevoditelj će umetnuti odgovarajući kôd prema svom nahođenju i prevesti program. Ne mali broj korisnika će zbog toga zanemariti takva upozorenja. Međutim, valja primijetiti da često taj umetnuti kôd ne odgovara onome što je programer zamislio. Zanemarivanjem upozorenja programer gubi pregled nad korektnošću kôda, pa se lako može dogoditi da rezultirajući izvedbeni kôd daje na prvi pogled neobjašnjive rezultate.

Zadatak: *Izbacite iz gornjeg kôda desnu vitičastu zagradu } te pokušajte prevesti kôd. Pogledajte koje će pogreške i upozorenja javiti prevoditelj.*

Zadatak: *U nekim knjigama starijeg datuma nije rijetkost naći da je funkcija* `main()` *navedena („definirana”) kao:*

```
void main()
```

gdje je `void` *ključna riječ koja označava da funkcija ne vraća ništa, tj. da (eventualno) obavlja nekakvu operaciju, ali ne vraća nikakav rezultat kôdu koji je tu funkciju pozvao. Iako je ovakva definicija funkcije* `main` *neispravna jer nije u skladu sa standardom jezika C++, zamijenite u prethodnom primjeru* `int` *oznaku tipa funkcije* `main()` *oznakom* `void` *i pokušajte prevesti kôd. Nakon toga još uklonite naredbu* `return 0` *iz tijela funkcije* `main` *te ponovno prevedite program.*

U početku, dok se ne naviknete na „jezik” vašeg prevoditelja, namjerno ubacujte pogreške u kôd i pogledajte koje će poruke o pogreškama i upozorenja prevoditelj javiti.

Dok su vaši programi relativno kratki, lakše ćete razlučiti pogrešku i povezati je s porukom prevoditelja. Gornji savjet će od posebne koristi biti onima koji nisu vični engleskom jeziku (u kojem su poruke … pa valjda svih C++ prevoditelja).

Programi poput gornjeg nemaju baš neku praktičnu primjenu, te daljnju analizu gornjeg kôda prepuštamo poklonicima minimalizma. *We need some action, man!* Stoga pogledajmo sljedeći primjer:

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Ja sam za C++!!! A vi?" << endl;
    return 0;
}
```

U ovom primjeru uočavamo tri nova retka. U prvom retku nalazi se naredba `#include <iostream>` kojom se od prevoditelja zahtijeva da u naš program uključi zaglavlje biblioteke `iostream`. U toj biblioteci nalazi se *izlazni tok* (engl. *output stream*) te funkcije koje omogućavaju ispis podataka na zaslonu. Ta biblioteka nam je neophodna da bismo u prvom retku glavnoga programa ispisali tekst poruke. Naglasimo da `#include` nije naredba C++ jezika, nego se radi o *pretprocesorskoj naredbi* (§23.2). Naletjevši na nju, prevoditelj će prekinuti postupak prevođenja kôda u tekućoj datoteci, skočiti u datoteku `iostream`, prevesti ju, a potom se vratiti u početnu datoteku na redak iza naredbe `#include`. Sve pretprocesorske naredbe počinju znakom #.

`iostream` je primjer *datoteke zaglavlja* (engl. *header file*). U takvim datotekama se nalaze deklaracije funkcija te definicije konstanti i klasa sadržanih u odgovarajućim bibliotekama. Datoteke zaglavlja neophodno je uključiti u izvorni kôd da bi prevoditelj prepoznao konstantu, funkciju ili klasu koje koristimo u našem kôdu te provjerio radimo

li s njima dozvoljene operacije. Jedna od osnovnih značajki (zli jezici će reći mana) jezika C++ jest vrlo oskudan skup funkcionalnosti ugrađenih u sam jezik: u sam jezik nisu ugrađeni niti jedna funkcija ili klasa, jezik definira svega desetak ugrađenih tipova (§3.4), a skup ključnih riječi (tj. riječi koji imaju specijalno značenje u kôdu) sastoji se svega od 70-ak riječi (vidi tablicu 3.1 na str. 46), od kojih se četvrtina vrlo rijetko koristi. Ta oskudnost olakšava učenje samog jezika, te bitno pojednostavnjuje i ubrzava postupak prevođenja. Za specifične zahtjeve na raspolaganju je veliki broj odgovarajućih biblioteka funkcija i klasa.

Datoteke zaglavlja nalaze se obično u potkazalu `include` unutar kazala u kojem je instaliran prevoditelj. Znatiželjnicima željnima znanja preporučujemo da zavire u te datoteke budući da se u njima može naći dosta korisnih informacija, ali svakako valja paziti da se sadržaj tih datoteka ne mijenja.

U drugom retku kôda napisana je naredba `using namespace std`. `using` i `namespace` su ključne riječi jezika C++ kojima se „aktivira" određeni *imenski prostor* ili *imenik* (engl. *namespace*), a `std` je naziv imenika u kojem su definirane sve standardne funkcije i tipovi, uključujući i one iz već spomenute biblioteke `iostream`. Imenici su uvedeni da se izbjegne sraz istih imena funkcija, klasa ili objekata iz različitih biblioteka. Na primjer, ako dvije različite funkcije iz različitih biblioteka imaju isto ime, prevoditelj će javiti pogrešku ako ne može razlučiti koju od tih funkcija želimo doista pozvati (slično kao što dvije osobe s istim imenom možemo jednoznačno razlikovati samo ako navedemo i njihova prezimena). Kada ne bismo imali na raspolaganju imenike, jedino rješenje u takvom slučaju bilo bi promijeniti ime funkcije u jednoj od biblioteka, što često nije moguće jer proizvođači redovito te biblioteke isporučuju u već prevedenom obliku.

Da nismo aktivirali cijeli imenik `std`, za svako ime definirano u tom imeniku morali bismo navesti puni naziv koji obuhvaća imenik i ime, međusobno odvojene operatorom `::` (dvije dvotočke). Konkretno, prethodni program bismo morali napisati kao:

```
#include <iostream>

int main()
{
    std::cout << "Ja sam za C++!!! A vi?" << std::endl;
    return 0;
}
```

Zbog jednostavnosti ćemo u primjerima podrazumijevano uključivati imenik `std`. Detaljnije o imenicima bit će govora u poglavlju 19, a do tada uzmite naredbu `using namespace std` „zdravo za gotovo".

Treća novina u gornjem primjeru jest naredba unutar glavne funkcije koja počinje sa `cout`. `cout` je ime izlaznog toka definiranog u biblioteci `iostream`, pridruženog zaslonu računala. Operatorom << (dva znaka „manje od") podatak koji slijedi upućuje se na izlazni tok, tj. na zaslon računala. U gornjem primjeru to je kratka promidžbena poruka:

```
Ja sam za C++!!! A vi?
```

Ona je u programu napisana unutar znakova navodnika, koji upućuju na to da se radi o tekstu koji treba ispisati doslovce. Biblioteka `iostream` bit će detaljnije opisana kasnije, u poglavlju 21.

Međutim, to još nije sve! Iza znakovnog niza ponavlja se operator za ispis, kojeg slijedi `endl`. `endl` (od engl. *end-of-line*, kraj retka) je *manipulator* (§21.6.6) u biblioteci `iostream` koji prebacuje ispis u novi redak, to jest vraća kursor na početak sljedećeg retka na zaslonu.

Maštovita čitateljica ili čitatelj će sami zaključiti da bi se operatori za ispis mogli dalje nadovezivati u istoj naredbi:

```
cout << "Ja sam za C++!!! A vi?" << endl << "Ne, hvala!";
```

Zadatak: *Utipkajte kôd gore opisanog programa u računalo, pohranite ga u datoteku te pokrenite prevoditelj/povezivač. Izvršite dobiveni program i pogledajte ispis na zaslonu.*

Ako koristite prevoditelj na operacijskom sustavu s grafičkim sučeljem (npr. Microsoft Windows), a prevedeni program ne pokrećete iz tekstovnog (komandnog) prozora već izravno iz grafičkog okruženja, tada se lako može dogoditi da se nakon pokretanja programa na trenutak otvori tekstovni prozor, koji će se nestati prije nego što uspijete pročitati ijedno slovo teksta koji se ispisao. Osjećat ćete se nasamareni („Iaaa!!! Zar sam zato punih 20 minuta na *torrentu* tražio najnoviju verziju prevoditelja?"). Možda će vam pasti na pamet neka od sljedećih perverznih ideja:

- nabaviti neko ultra-staro, hiper-sporo računalo s ekstremno sporom grafičkom karticom i monitor s perzistencijom od desetak sekundi;
- nabaviti najmoderniji foto-aparat s vrlo brzim zatvaračem. Bljeskalica doduše nije neophodna, ali bi dobro došao mrežni (po mogućnosti gigabitni) priključak na računalo preko kojeg bi se okidanje aparata sinkroniziralo s otvaranjem prozora (alternativa je video kamera s mogućnošću ubrzanog snimanja);
- pozvati cijelu obitelj pred računalo i objaviti natječaj s posebnom nagradom za onog tko prvi uspije pročitati cijeli tekst (ako ste izuzetno sadistički nastrojeni, proširit ćete tekst poruke na barem dva retka).

Većina integriranih razvojnih okolina omogućava da program nakon uspješnog prevođenja pokrenete izravno iz nje. Pritom će spriječiti da se tekstovni prozor sam zatvori po završetku programa. U protivnom, preostaju vam sljedeća dva rješenja:

- otvorite komandni prozor i program pokrenite iz komandne linije, ili
- pri kraju kôda, ispred naredbe `return` dodajte sljedeće dvije naredbe:

```
char z;
cin >> z;
```

Njima se deklarira znakovna (`char`) varijabla `z`, i potom u naredbi `cin` program očekuje da utipkamo vrijednost te varijable. Kada program prilikom izvođenja dođe na naredbu `cin`, izvođenje će se prekinuti sve dok ne pritisnemo neki znak (samo jedan znak!) i potom tipku *Enter*. To će nam omogućiti da u miru Božjem, duboko u noć proučavamo ispis naših umotvorina. No, unaprijed vas upozoravamo da neke od ispisanih poruka

nećete uspjeti vidjeti niti na ovaj način. Naime, u poglavlju o klasama i njihovim destruktorima neki od primjera poruke ispisuju pri izlasku iz programa, tj. nakon gore spomenutih naredbi.

Zadatak: *Kako bi izgledao izvorni kôd ako bismo željeli ispis* `endl` *znaka staviti u posebnu naredbu? Dodajte u gornji primjer ispis poruke „*`Imamo C++!!!`*” u sljedećem retku (popratno euforičko sklapanje ruku iznad glave nije neophodno). Nemojte zaboraviti dodati i ispis* `endl` *na kraju tog retka!*

Osvrnimo se još jednom na smisao uključivanja datoteka zaglavlja. Podsjetimo se kako smo zbog izlaznog toka `cout` morali na početku programa uključiti zaglavlje `iostream` i aktivirati imenik `std`. Kada tijekom analize programskog kôda naleti na riječ `cout`, budući da ona nije ključna riječ jezika, prevoditelj će pretražiti sva uključena zaglavlja ne bi li u njima pronašao značenje (tj. definiciju) te riječi. Nakon što uspješno pronađe njenu definiciju, prevoditelj nastavlja s operatorom `<<` navedenim odmah iza objekta `cout`. Operator `<<` je *binarni operator* koji obavlja neku operaciju, koristeći dva objekta koji su navedeni s njegove lijeve i desne strane. Lijevo od operatora je objekt `cout`, a desno od njega znakovni niz. Stoga je sljedeći prevoditeljev korak provjeriti postoji li u uključenim zaglavljima definicija tog operatora kojoj s lijeve strane može stajati objekt `cout`, a s desne strane znakovni niz. Dakle, uključivanje zaglavlja je neophodno da prevoditelj može provjeriti jesu li programske jedinke koje navodimo u kôdu (a koje nisu dio jezika) definirane te koristimo li ih na ispravan način, u skladu s njihovim definicijama i u duhu jezika. Naravno da prevoditelj neće uočiti jesmo li pritom napravili neku logičku pogrešku.

Zadatak: *Izbacite iz gornjeg kôda pretprocesorsku naredbu* `#include` *te pokušajte prevesti kôd. Pogledajte koje će pogreške javiti prevoditelj. Ponovite to za slučaj da izbacite naredbu* `using namespace`.

Zadatak: *Operator* `<<` *zamijenite nekim drugim operatorom (npr.* `+`*) te pogledajte koje će pogreške javiti prevoditelj.*

Zadatak: *Umjesto objekta* `cout`*, ispred operatora* `<<` *napišite decimalni broj, npr.* `3.14` *te pogledajte koje će pogreške javiti prevoditelj.*

Okorjeli C-programeri su vjerojatno sa gnušanjem prokomentirali prethodni kôd riječima: „Halo?! A gdje je tu `printf`? Ispis bi se puuuuno elegantnije napravio pomoću `printf`-a!”. Za neupućene: `printf()` je standardizirana C-ovska funkcija za formatirani ispis na uobičajeni izlaz (engl. *standard output*), tj. konzolu i detaljno je opisana u Prilogu B. Ona omogućava precizno definiranje formata ispisa. Štoviše, ona *zahtijeva* da se taj format precizno definira, uključujući i tip podatka koji treba ispisati. Na primjer, ako se ispisuje cijeli broj, tada funkciji `printf()` treba proslijediti `"%d"` za oznaku tipa, dok za decimalni broj treba proslijediti `"%f"`. Problemi će iskrsnuti u slučajevima kada ne znamo unaprijed kojeg će tipa biti podatak koji se ispisuje ili ako promijenimo tip podatka, a zaboravimo promijeniti format u pozivu funkcije. Tada se može dogoditi da dobijemo potpuno nesuvisli ispis, a u nekim slučajevima se program može i srušiti zbog neodgovarajućeg formata.

S druge strane, korištenje izlaznih tokova pojednostavljuje pisanje kôda za ispis podataka. Programer ne mora eksplicitno specificirati kojeg je tipa podatak, već će prevoditelj sam to zaključiti i prikazati ga na odgovarajući način. Ukoliko nam taj podrazumijevani način ne odgovara, na raspolaganju su nam dodatni operatori kojima možemo dodatno „fino ugoditi” prikaz. Osim toga, funkcija `printf()` ograničena je samo na ispis u konzolu. Želimo li iste podatke upisati u datoteku, morat ćemo upotrijebiti funkciju `fprinf()`, a za ispis u znakovni niz koji nam može poslužiti kao spremnik (*buffer*) posegnut ćemo za funkcijom `sprintf()`. S druge strane, izlazni tokovi omogućavaju da se istim naredbama podaci mogu ispisivati u konzolu, u datoteku ili u memorijski spremnik. Ulazno-izlazni tokovi bit će detaljno će opisani u 21. poglavlju.

Prije nego što završimo s ovim odjeljkom, pozabavimo se još malo manipulatorom `endl` kojim smo osigurali da se kursor prebaci na početka novog retka. Isti efekt postigli bismo da smo umjesto `endl` u izlazni tok stavili znak za novi redak `'\n'`:

```
cout << "Ja sam za C++!!! A vi?" << '\n';
```

ili jednostavno, tekst koji želimo ispisati zaključili njime:

```
cout << "Ja sam za C++!!! A vi?\n";
```

Između manipulatora `endl` i znaka za novi redak postoji jedna suptilna razlika: `endl` šalje znak za novi redak u izlazni tok i dodatno ga „isplahne” (engl. *flush*), osiguravajući da svi podaci koji su eventualno još u memoriji budu poslani na izlaznu jedinicu. Kod ispisa na ekran efektivno nema nikakve razlike koristimo li `endl` ili samo znak za novi redak, no razlike može biti kod zapisa u datoteku kada se podaci ne zapisuju izravno u datoteku, već se pohranjuju u *međuspremnik* (*buffer*). Kada se međuspremnik popuni, podaci iz njega se u paketu pohranjuju na disk, a međuspremnik se prazni i priprema za prihvat novih podataka. Takvo zapisivanje podataka u paketima je puno efikasnije nego da se podaci zapisuju bajt po bajt na disk, no ima i jedan nedostatak: ako se program sruši ili u tom trenutku nestane struje, podaci koji su u tom trenutku bili u međuspremniku neće biti zapisani na disk, već će se nepovratno izgubiti. Isplahnjivanje će prisilno isprazniti međuspremnik i osigurati da se svi podaci poslani do tog trenutka na izlazni tok doista zapišu u datoteku.

Iako ovakvo prisilno isplahnjivanje može značiti nešto sporiji zapis u datoteku, jer se ne prebacuje sadržaj cijelog međuspremnika nego samo njegovog dijela, u većini primjena nećete primijeti razliku budući da sama elektronika diska provodi konačnu optimizaciju zapisivanja.

Ako vam je prvenstveno bitna brzina zapisivanja u datoteku, izbjegavajte manipulator `endl` i koristite znak za novi redak. Ako je bitna pouzdanost zapisa, tada je dobro koristiti `endl`.

Radi preglednosti i bolje uočljivosti, mi ćemo uglavnom koristiti manipulator `endl`.

2.4. Moj treći C++ program

Sljedeći primjer je pravi mali dragulj interaktivnog programa:

```
#include <iostream>
using namespace std;

int main()
{
    int a;                      // deklariramo varijablu za prvi broj
    cout << "Upiši prvi broj:";
    cin >> a;                   // unos prvog broja

    int b;                      // deklariramo varijablu za drugi broj
    cout << "Upiši i drugi broj:";
    cin >> b;                   // unos drugog broja

    int c;                  // deklariramo varijablu za rezultat i
    c = a + b;              // pridružujemo joj zbroj unesenih brojeva
    // ispisujemo rezultat:
    cout << "Njihov zbroj je: " << c << endl;
    return 0;
}
```

U ovom primjeru uočavamo nekoliko novina. Prvo, to je redak

```
int a;
```

u kojem se deklarira varijabla `a`. Ključnom riječi (*identifikatorom tipa* ili *oznakom tipa*) `int` deklarirali smo ju kao cjelobrojnu, tj. tipa `int`. Deklaracijom smo pridijelili simboličko, nama razumljivo i lako pamtljivo ime memorijskom prostoru u koji će se pohranjivati vrijednosti te varijable. Naišavši na deklaraciju, prevoditelj će rezervirati odgovarajući prostor u memoriji računala, pridružiti tom prostoru ime `a` i zapamtiti da se u taj prostor pohranjuje cjelobrojna varijabla tipa `int`. Kada tijekom prevođenja ponovno sretne varijablu s tih imenom, prevoditelj će znati da se radi o cjelobrojnoj varijabli i znat će gdje bi se ona u memoriji trebala nalaziti. Tip varijable određuje na koji će se način podatak pohranjivati u memoriji, raspon dozvoljenih vrijednosti te koje su operacije moguće nad njime. Opširnije o deklaracijama varijabli i o tipovima podataka govorit ćemo u sljedećem poglavlju.

Slijedi ispis poruke „`Upiši prvi broj:`”, čije značenje ne trebamo objašnjavati. Iza poruke nismo dodali ispis znaka `endl`, jer želimo da nam kursor ostane iza poruke, u istom retku, spreman za unos prvog broja.

Druga novina je *ulazni tok* (engl. *input stream*) `cin` koji je zajedno s izlaznim tokom definiran u datoteci zaglavlja `iostream`. On služi za učitavanje podataka s konzole, tj. tipkovnice. Operatorom >> (dvostruki znak „veće od”) podaci s konzole upućuju se u memoriju varijabli `a`. Naglasimo da učitavanje počinje tek kada se na tipkovnici pritisne tipka za novi redak, tj. tipka *Enter*. To znači da kada pokrenete program, nakon ispisane

poruke možete utipkavati bilo što i to po potrebi brisati – tek kada pritisnete tipku *Enter*, program će analizirati unos te pohraniti broj koji je upisan u memoriju računala.

Svaki puta kada nam u programu treba ispis podataka na zaslonu ili unos podataka preko tipkovnice pomoću ulazno-izlaznih tokova `cin` ili `cout`, treba uključiti zaglavlje odgovarajuće `iostream` biblioteke pretprocesorskom naredbom `#include`.

Radi sažetosti kôda, u većini primjera koji slijede, pretprocesorska naredba za uključivanje `iostream` biblioteke neće biti napisana, ali se ona podrazumijeva. Koriste li se ulazno-izlazni tokovi, njeno izostavljanje prouzročit će pogrešku kod prevođenja.

No, završimo s našim primjerom. Nakon naredbi za unos prvog broja slijede potpuno istovjetne naredbe za unos drugog broja `b`, a zatim slijedi naredba za računanje zbroja

```
c = a + b;
```

Ona kaže računalu da treba zbrojiti vrijednosti varijabli `a` i `b`, te taj zbroj pohraniti u novodeklariranu varijablu `c`, tj. u memorijski prostor koji je prevoditelj rezervirao za tu varijablu. Vrijednost te varijable ispisuje se zadnjom naredbom prije naredbe `return`.

U početku će čitatelj zasigurno imati problema s razlučivanjem operatora << i >>. U vjeri da će olakšati razlikovanje operatora za ispis i učitavanje podataka, dajemo sljedeći naputak.

Operatore << i >> možemo zamisliti kao strelice koje pokazuju smjer prijenosa podataka [Lippman91].

Primjerice,

```
>> a
```

preslikava vrijednost u `a`, dok

```
<< c
```

preslikava vrijednost iz `c`.

Zadatak: *Pokrenite „Moj treći program" nekoliko puta, unoseći svaki puta sve veće brojeve i pratite rezultate. Na primjer, pokušajte sa sljedećim parovima: 326766 i 1; 326766 i 2; 2147483646 i 1; 2147483646 i 2. Ponovite to i za identične negativne brojeve.* ***Uputa:*** *Za neki od navedenih primjera dogodit će se brojčani „preljev"- uneseni broj i rezultat će nadmašiti najveću pozitivnu odnosno najveću negativnu vrijednost koju `int` tip podataka može sadržavati pa će rezultat zbrajanja će biti netočan. O brojčanom preljevu će biti riječ u sljedećem poglavlju.*

Zadatak: *Izostavite deklaracije varijabli a i b pa provjerite koje će pogreške javiti prevoditelj.*

Ako ste primjere iz prethodnog odjeljka isprobavali na računalu s Microsoft Windows operacijskim sustavom, zasigurno ste primijetili da se pri izvođenju programa slovo „š" ne ispisuju ispravno. Problem nije u vašem računalu, već u načinu na koji konzola u Windowsima tretira znakove specifične hrvatskom jeziku (isto vrijedi za većinu znakova koji nisu iz engleske abecede). Budući da se radi o problemu koji je specifičan za izvedbenu platformu a ne sam jezik C++, rješenje problema navest ćemo u prilogu knjige. U daljnjim primjerima ćemo stoga izbjegavati znakove specifične za hrvatski jezik.

2.5. Komentari

Na nekoliko mjesta u navedenim primjerima kôda možemo uočiti tekstove koji započinju dvostrukim kosim crtama (`//`). Radi se o *komentarima*. Kada prevoditelj naleti na dvostruku kosu crtu, on će zanemariti sav tekst koji slijedi do kraja tekućeg retka i prevođenje će nastaviti u sljedećem retku. Komentari dakle ne ulaze u izvedbeni kôd programa i služe programerima za opis značenja pojedinih naredbi ili dijelova kôda. Komentari se mogu pisati u zasebnom retku ili recima, što se obično rabi za dulje opise, primjerice što određeni program ili funkcija radi:

```
//
// Ovo je moj treći C++ program, koji zbraja
// dva broja unesena preko tipkovnice, a zbroj
// ispisuje na zaslonu.
//                  Autor: N. N. Hacker III
//
```

Za kraće komentare, na primjer za opise varijabli ili nekih operacija, komentar se piše u nastavku naredbe, kao što smo vidjeli u primjerima.

Uz gore navedeni oblik komentara, jezik C++ podržava i komentare unutar para znakova `/* */`. Takvi komentari započinju slijedom `/*` (kosa crta i zvjezdica), a završavaju slijedom `*/` (zvjezdica i kosa crta). Kraj retka ne znači podrazumijevani završetak komentara, pa se ovakvi komentari mogu protezati na nekoliko redaka izvornog kôda, a da se pritom znak za komentiranje ne mora ponavljati u svakom retku:

```
/*
    Ovakav način komentara
    preuzet je iz programskog
    jezika C.
*/
```

Stoga je ovakav način komentiranja naročito pogodan za (privremeno) isključivanje dijelova izvornog kôda. Ispred naredbe u nizu koji želimo isključiti dodat ćemo oznaku `/*` za početak komentara, a iza zadnje naredbe u nizu nadodat ćemo oznaku `*/` za zaključenje komentara.

Iako komentiranje programa iziskuje dodatno vrijeme i napor, u kompleksnijim programima ono se redovito isplati. Dogodi li se da netko drugi mora ispravljati vaš kôd, ili (još gore) da nakon dugo vremena vi sami morate ispravljati svoj kôd, komentari će vam olakšati da proniknete u ono što je autor njime htio reći. Svaki ozbiljniji programer ili onaj tko to želi postati mora biti svjestan da će nakon desetak ili stotinjak napisanih programa početi zaboravljati čemu pojedini program služi. Zato je vrlo korisno na početku datoteke izvornog programa u komentaru navesti osnovne „generalije", na primjer ime programa, ime datoteke izvornog kôda, kratki opis onoga što bi program trebao raditi, funkcije i tipovi definirani u datoteci te njihovo značenje, autor(i) kôda, uvjeti pod kojima je program preveden (operacijski sustav, ime i oznaka prevoditelja):

```
/*********************************************************
Program:    Moj treći C++ program
Datoteka:   Treci.cpp
Funkcije:   main() - cijeli program je u jednoj datoteci
Opis:       Učitava dva broja i ispisuje njihov zbroj
Autori:     Boris (bm)
            Julijan (jš)
Okruženje:  Pendžeri MEeee
            Gledljivi C++ 14.0 prevoditelj
*********************************************************/
```

Prije su se u komentarima na početku datoteke kronološki navodile i promjene koje su na toj datoteci rađene, tj. koji su dijelovi kôda i funkcionalnosti dodavani, mijenjani ili uklanjani. Danas se to više ne radi jer svaki ozbiljniji programski projekt koristi neki alat za praćenje i nadzor promjena u izvornom kôdu (engl. *revision control*, *version control* ili *source control*) u kojem se vodi evidencija o svim promjenama.

Dobro i razumljivo pisani kôd iziskuje vrlo malo komentara. Iako komentari u samom izvornom kôdu mogu doprinijeti njegovoj razumljivosti, s količinom komentara ne treba pretjerivati, jer će u protivnom izvorni kôd postati nepregledan. Naravno da ćete izbjegavati banalne komentare poput:

```
int a, b, c;            // deklaracija varijabli a, b i c
c = a + b;              // zbraja a i b
++i;                    // uvećava i za 1
y = sqrt(x)             // poziva funkciju sqrt
```

Nudimo vam sljedeće naputke gdje i što komentirati:

- Na početku datoteke izvornog kôda opisati sadržaj datoteke.
- Kod deklaracije objekata obrazložiti njihovo značenje i primjenu.
- Ispred definicije tipa (klasa, struktura, pobrojenja) dati opis čemu služi.
- Ispred deklaracije funkcije ili funkcijskog člana dati opis što radi, što su joj argumenti i što vraća kao rezultat. Eventualno dati opis algoritma koji se primjenjuje.
- Dati sažeti opis na mjestima u programu gdje nije potpuno očito što kôd radi.

Tekst komentara mora biti precizan i razumljiv te treba izbjegavati kriptična razmetanja oblika:

```
// koji hack!
c = a;
```

Prije ili kasnije takva će se razmetanja obiti samom autoru o glavu, kao što se doista dogodilo jednom programeru kada je nakon nekoliko godina naletio na svoj komentar:

```
// Ovo je ružan hack. Unaprijed se ispričavam bilo kome tko će
// se u budućnosti morati njime pozabaviti.
```

Privremeni komentari mogu dobro doći tijekom razvoja programa: kada treba napisati programski kôd za neki postupak, prvo se komentarima, korak po korak opiše postupak, tj. skicira se postupak. Potom se komentari u pojedinim koracima nadopunjuju stvarnim naredbama. Ako su naredbe u potpunosti razumljive, pripadajuće komentare ćemo izbrisati.

Uvijek pišite kôd kao da će njegovo održavanje preuzeti osoba koja je nasilni psihopat i koja zna gdje živite.

Martin Golding

Napomenimo da će, radi bolje razumljivosti, primjeri u knjizi biti redovito prekomentirani, tako da će komentari biti pisani i tamo gdje je iskusnom korisniku jasno značenje kôda. Osim toga, komentare ćemo pisati i uz naredbe za koje bi prevoditelj javio pogrešku.

I na kraju jedna, na prvi pogled možda neprimjerena, ali dugoročno gledano (nadamo se) vrlo korisna preporuka:

Komentare pišite na engleskom jeziku.

Nekome će se to u prvi mah učiniti nepotrebnim razmetanjem. No, danas nije rijetkost da se programi razvijaju za međunarodno tržište i da u njihovoj izradi sudjeluju programeri iz više zemalja koji razmjenjuju izvorne kôdove tijekom razvoja. Zamislite da ste u jednom takvom timu i da kolega koji ne zna niti riječi hrvatskog naleti na sljedeći komentar iznad (na prvi pogled) banalne naredbe:

```
// ovo nije važno - može se izbaciti
c = a + b;
```

Svakako bi komentar na engleskom jeziku, pa ma koliko lošim engleskim jezikom napisan, bio razumljiviji velikoj većini čitatelja:

```
// this not very important – can trow out
c = a + b;
```

Čak i ako komentar stavljate kao privremenu zabilješku za sebe što još treba dodati ili promijeniti, napravite to na engleskom. Što prije počnete redovito pisati komentare na engleskom, prije ćete se naviknuti na to.

Budući da je ova knjiga zamišljena da bude udžbenik jezika C++ na hrvatskom jeziku, mi ćemo komentare, usprkos gornjoj preporuci, ipak pisati na hrvatskom.

2.6. Rastavljanje naredbi

Razmotrimo još dva problema važna svakoj početnici ili početniku: praznine u izvornom kôdu i produljenje naredbi u više redaka. U primjerima u knjizi intenzivno ćemo koristiti praznine između imena varijabli i operatora, iako ih iskusniji programeri često izostavljaju. Tako smo umjesto

```
c = a + b;
```

mogli pisati

```
c=a+b;
```

Umetanje praznina doprinosi preglednosti kôda, a često je i neophodno da bi prevoditelj interpretirao djelovanje nekog operatora onako kako mi očekujemo od njega. Ako je negdje dozvoljeno umetnuti prazninu, tada broj praznina koje se smiju umetnuti nije ograničen, pa smo tako gornju naredbu mogli pisati i kao

```
c=          a    +          b          ;
```

što je još nepreglednije! Istaknimo da se pod prazninom (engl. *whitespace*) ne podrazumijevaju isključivo prazna mjesta dobivena pritiskom na razmaknicu (engl. *space*), već i praznine dobivene tabulatorom (engl. *tabs*) te znakove za pomak u novi red (engl. *newlines*).

Naredbe u izvornom kôdu nisu ograničene na samo jedan redak, već se mogu protezati na nekoliko redaka – završetak svake naredbe jednoznačno je određen znakom `;`. Stoga pisanje naredbe možemo prekinuti na bilo kojem mjestu gdje je dozvoljena praznina, te nastaviti pisanje u sljedećem retku, na primjer

```
c =
a + b;
```

Naravno da je ovakvim potezom kôd iz razmatranog primjera postao nepregledniji. Razdvajati naredbe u više redaka ima smisla kod vrlo dugačkih naredbi, kada one ne stanu u jedan redak, odnosno postanu zbog svoje duljine nepregledne. Većina programera ograničava duljinu retka na 80-100 znakova, koliko ih stane na zaslon. Zbog formata knjige duljine redaka u našim primjerima su manje.

Posebnu pažnju treba obratiti na razdvajanje znakovnih nizova. Pokušamo li prevesti sljedeći primjer, dobit ćemo pogrešku prilikom prevođenja:

```
#include <iostream>
using namespace std;

int main()
{
    // pogreška: nepravilno razdvojeni znakovni niz
    cout << "Pojavom jezika C++ ostvaren je
             tisućljetni san svih programera" << endl;
    return 0;
}
```

Prevoditelj će javiti pogrešku da znakovni niz `Pojavom ... ostvaren je` nije zaključen znakom dvostrukog navodnika, jer prevoditelj ne uočava da se niz nastavlja u sljedećem retku. Da bismo mu to dali do znanja, završetak prvog dijela niza treba označiti lijevom kosom crtom \ (engl. *backslash*):

```
    cout << "Pojavom jezika C++ ostvaren je \
tisućljetni san svih programera" << endl;
```

pri čemu nastavak niza ne smijemo uvući, jer bi prevoditelj dotične praznine prihvatio kao dio niza. Stoga je u takvim slučajevima preglednije niz rastaviti u dva odvojena niza:

```
    cout << "Pojavom jezika C++ ostvaren je "
            "tisućljetni san svih programera" << endl;
```

Pritom se ne smije zaboraviti staviti prazninu na kraj prvog ili početak drugog niza. Gornji niz mogli smo rastaviti na bilo kojem mjestu:

```
    cout << "Pojavom jezika C++ ostvaren je tis"
            "ućljetni san svih programera" << endl;
```

ali je očito takav pristup manje čitljiv.

Budući da znak ; označava kraj naredbe, moguće je više naredbi napisati u jednom retku. Tako smo „Moj treći program” mogli napisati i na sljedeći način:

```
#include <iostream>
using namespace std;

int main()
{
    int a, b, c; cout << "Upiši prvi broj:"; cin >> a;
    cout << "Upiši i drugi broj:"; cin >> b; c = a + b;
    cout << "Njihov zbroj je: " << c << endl;
    return 0;
}
```

Program će biti preveden bez pogreške i raditi će ispravno. No, očito je da je ovako pisani izvorni kôd daleko nečitljiviji. Osim toga, otkrivanje pogrešaka alatima za ispravljanje pogrešaka (*debuggerima*) će u tako pisanom kôdu biti teže. Današnji programi za otkrivanje pogrešaka omogućavaju izvođenje programa redak po redak izvornog kôda, pri čemu je cijela linija kôda čije naredbe se trenutno izvode jasno istaknuta, obično drugom bojom pozadine. Ako redak sadrži više naredbi, *debugger* neće moći pokazati koja se naredba u tom retku trenutno izvodi pa će biti vrlo teško prepoznati naredbu koja je prouzročila neočekivani rezultat.

Pisanje više naredbi u istom retku treba izbjegavati gdjegod je to moguće.

2.7. Upravljanje slijedom izvođenja programa

Često će nam u programima zatrebati operacije koje već postoje u standardnoj biblioteci jezika C++ ili u nekoj drugoj dostupnoj biblioteci koju je razvio neki neovisni proizvođač ili programer (engl. *third-party library*). Umjesto da sami pišemo implementaciju operacije, u takvim slučajevima bolje je posegnuti za gotovim i provjerenim kôdom. Ilustrirajmo to primjerom programa kojim izračunavamo hipotenuzu pravokutnog trokuta pomoću Pitagorina poučka:

$$c = \sqrt{a^2 + b^2}.$$

Korisnik će programu zadati duljine kateta a i b, a program će izračunati i ispisati duljinu hipotenuze. Umjesto cjelobrojnih varijabli tipa `int`, za pohranjivanje duljina stranica koristit ćemo decimalne varijable tipa `double` (§3.4.1), tako da program neće biti ograničen na cjelobrojne duljine stranica:

```
#include <iostream>
#include <cmath>        // trebamo uključiti zbog funkcije sqrt()
using namespace std;
int main()
{
    // učitajmo prvu stranicu:
    cout << "Stranica1 = ";
    double str1;
    cin >> str1;
    // učitajmo drugu stranicu:
    cout << "Stranica2 = ";
    double str2;
    cin >> str2;
    // izračunajmo i ispišimo hipotenuzu
    double str3 = sqrt(str1 * str1 + str2 * str2);
    cout << "Stranica3 = " << str3 << endl;
    return 0;
}
```

Za izračunavanje kvadratnog korijena upotrijebili smo standardnu funkciju `sqrt()`. Ona je deklarirana u zaglavlju `cmath` pa smo to zaglavlje također morali uključiti u naš kôd pretprocesorskom naredbom `#include`. Budući da je funkcija definirana u imeniku `std`, njega ne moramo ga dodatno aktivirati jer je već to napravljeno zbog klasa `cin` i `cout`.

Funkciju `sqrt()` u kôdu pozivamo na isti način kao što se funkcije pozivaju u matematici: unutar zagrada iza imena funkcije prosljeđujemo vrijednost čiji kvadratni korijen želimo izračunati, a rezultat koji funkcija vraća pridružujemo novoj varijabli `str3`, čiju vrijednost potom ispisujemo. Općenito, prilikom poziva funkcije izvođenje programa se preusmjerava u kôd kojim je funkcija definirana. Nakon što se kôd funkcije izvede, program se nastavlja iza mjesta gdje je funkcija bila pozvana. Funkcijama ćemo se detaljno pozabaviti u poglavlju 8.

Zadatak: *Napišite program koji će izračunati i ispisati sinus i kosinus zadanog broja pozivom standardnih funkcija `sin()`, odnosno `cos()`, deklariranih u zaglavlju `cmath`.*

Prethodni program je izračunavao duljinu hipotenuze za zadane katete. Pretpostavimo da želimo poopćiti program tako da može izračunati duljinu katete na osnovu zadanih duljina hipotenuze i druge katete:

$$b = \sqrt{c^2 - a^2}.$$

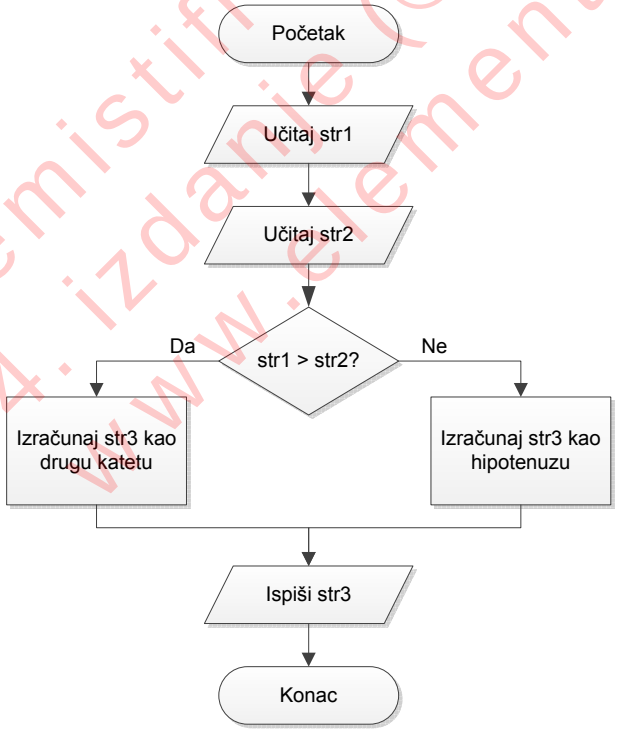
Bez obzira želimo li izračunati duljinu hipotenuze ili duljinu druge katete, uvijek treba unijeti duljine dviju stranica, a program mora na kraju ispisati izračunatu duljinu treće stranice. Međutim, formule za izračunavanje hipotenuze, odnosno katete međusobno se razlikuju. Zbog toga se izvođenje programa, nakon unosa duljina dviju stranica (`str1`, odnosno `str2`), mora razdvojiti u dvije grane (slika 2.3). Jedna grana će se izvoditi prilikom računanja hipotenuze, a druga grana prilikom računanja druge katete. Po završetku se obje grane stapaju da bi se ispisala izračunata duljina treće stranice `str3`.

Nameće se pitanje koji uvjet treba određivati hoće li se izvoditi prva ili druga grana. Mogli bismo zatražiti od korisnika da upiše neko slovo, na primjer ‘h’ za izračunavanje hipotenuze ili ‘k’ za izračunavanje druge katete. Da ne proširujemo program novim unosima, umjesto toga ćemo jednostavno usporediti dvije unesene duljine: ako je prva duljina veća od druge, uzet ćemo da je prva stranica hipotenuza i program će računati duljinu druge katete. U protivnom (ako je prva stranica kraća ili jednaka drugoj) ćemo smatrati da su obje zadane stranice katete te da treba izračunati duljinu hipotenuze:

```
#include <iostream>
#include <cmath>        // trebamo uključiti zbog funkcije sqrt()
using namespace std;

int main()
{
    // učitajmo prvu stranicu:
    cout << "Stranica1 = ";
    double str1;
    cin >> str1;
    // učitajmo drugu stranicu:
```

```
    cout << "Stranica2 = ";
    double str2;
    cin >> str2;
    // deklariramo str3 koja se izračunava unutar grana:
    double str3;
    // ako je prva zadana stranica dulja, ...
    if (str1 > str2)
    {
        // ... onda računaj drugu katetu,
        str3 = sqrt(str1 * str1 - str2 * str2);
    }
    else
    {
        // ...a inače računaj hipotenuzu.
        str3 = sqrt(str1 * str1 + str2 * str2);
    }
    // ispiši rezultat:
    cout << "Stranica3 = " << str3 << endl;
    return 0;
}
```



Slika 2.3. Prikaz grananja

Grananje smo realizirali pomoću naredbe `if` (§4.2.1): ako je zadovoljen uvjet naveden u naredbi (`str1 > str2`), izvode se naredbe unutar prvog para vitičastih zagrada; u protivnom se izvode naredbe unutar vitičastih zagrada navedenih ispod `else`. Nakon što se izvedu naredbe u bilo kojoj od grana, programa se nastavlja naredbama ispod `if-else` sklopa – u našem slučaju se samo ispisuje izračunata treća stranica. Ovakvo grananje je samo jedan primjer kako se može mijenjati slijed izvođenja programa. S naredbama za kontrolu toka upoznat ćemo se detaljnije u poglavlju 4.

2.8. Korisnički definirani tipovi

Osim jednostavnih podataka, poput cijelih ili decimalnih brojeva, programi pisani u jeziku C++ koriste i složene tipove podataka koji mogu sadržavati u sebi jedan ili više podataka drugih tipova. Na primjer, tip `string`, koji je definiran u standardnoj biblioteci, omogućava jednostavnu pohranu proizvoljnog teksta koji se može sastojati od gotovo neograničenog broja znakova. No, osim što takvi korisnički definirani podaci sadrže druge podatke, oni definiraju i operacije nad tim podacima. Tako tip `string` definira operacije za pretraživanje sadržanog teksta, brisanje pojedinih znakova ili umetanje novih znakova.

Upotrebu korisnički definiranih tipova ilustrirat ćemo jednostavnim programom koji za zadani tekst mijenja prvo slovo u rečenici u veliko slovo. Za početak ćemo napisati program koji cijeli zadani tekst ispisuje velikim slovima:

```
#include <iostream>
#include <string>       // u njemu je definiran tip string

using namespace std;

int main()
{
    cout << "Upišite neki tekst:" << endl;
    // deklariramo objekt tipa string u koji ćemo učitati tekst:
    string tekst;
    // učitavamo cijeli redak upisanog teksta:
    getline(cin, tekst);
    // uzimamo svaki pojedini znak iz teksta:
    for (char znak : tekst)
    {
        // pretvaramo ga u veliko slovo:
        char velikoSlovo = toupper(znak);
        // i ispisujemo veliko slovo:
        cout << velikoSlovo;
    }
    cout << endl;
    return 0;
}
```

Učitani tekst pohranjujemo u objekt koji smo maštovito nazvali `tekst`. Učitavamo ga standardnom funkcijom `getline()` kako bismo učitali sav upisani tekst do znaka za novi redak, a ne samo tekst do prve praznine (što bi bio slučaj da smo koristili operator >>). Potom, pomoću slijedne petlje `for` (§4.3.4) prolazimo kroz cijeli tekst i iz njega dohvaćamo pojedine znakove. Za svaki znak izvode se naredbe unutar vitičastih zagrada: znak se pomoću standardne funkcije `toupper()` pretvara u veliko slovo i to slovo se ispisuje. `for` je naredba za kontrolu toka programa koja omogućava da se skup naredbi ponovi određeni (pa čak i neograničeni) broj puta.

Budući da se za pretvorbu u velika slova koristi standardna funkcija `toupper()`, ona će voditi računa o ispravnosti pretvorbe. Ako je pronađeni znak već veliko slovo ili neki znak drugi znak (npr. znamenka ili simbol) koji nema veliko slovo, funkcija će vratiti taj znak bez promjene. Iako je funkcija `toupper()` deklarirana u zaglavlju `cctype`, primijetimo kako to zaglavlje nismo uključili. `cctype` je standardno zaglavlje naslijeđeno iz jezika C, a kako većina prevoditelja podrazumijevano uključuje ta zaglavlja, nema potrebe eksplicitno ih uključivati (iako uključivanje ne može škoditi).

Gornji kôd ima nedostatak da hrvatska slova č, ć, đ, š i ž najvjerojatnije neće promijeniti u velika slova. Da bismo to postigli, moramo postaviti lokalizacijske postavke za hrvatski jezik pomoću standardne funkcije `setlocale()`. Njoj treba proslijediti dva podatka: informaciju koje lokalizacijske postavke treba mijenjati (jezična pravila, formatiranje brojeva, formatiranje datuma ili nešto četvrto) te za koji jezik. Budući da želimo promijeniti postavke koje kontroliraju rukovanje znakovima, prvi argument će biti konstanta `LC_CTYPE`, definirana u standardnom zaglavlju `clocale`. Drugi argument funkciji `setlocale()` mora biti niz znakova koji identificiraju jezik za koji se postavke trebaju primijeniti. Tako bi na Windows operacijskim sustavima za hrvatski jezik naredba izgledala[1]:

```
setlocale(LC_TYPE, "croatian");
```

Naredbu treba umetnuti prije prvog poziva funkcije `toupper()`, na primjer prije `for` petlje:

```
    // početni dio isti kao u prethodnom ispisu
    setlocale(LC_TYPE, "croatian");
    for (char znak : tekst)
    // ostatak koda nepromijenjen...
```

Riješili smo problem pretvorbe u velika slova pa u sljedećem koraku možemo ograničiti pretvorbu samo na prvo slovo:

```
#include <iostream>
#include <string>

using namespace std;
```

[1] Na Linux operacijskim sustavima naredba `setlocale()` uglavnom nema efekta za naš slučaj zbog UTF-8 kodiranja koje se koristi za pohranu teksta na većini prevoditelja.

```
int main()
{
    setlocale(LC_TYPE, "croatian");

    cout << "Upišite neki tekst:" << endl;
    // deklariramo objekt tipa string u koji ćemo učitati tekst:
    string tekst;
    // učitavamo cijeli redak upisanog teksta:
    getline(cin, tekst);

    // tražimo poziciju prvog znaka iza eventualnih bjelina
    // na početku rečenice:
    size_t pozicija = tekst.find_first_not_of(' ');
    // iz teksta očitamo znak na pronađenoj poziciji:
    char znak = tekst[pozicija];
    // znak pretvorimo u veliko slovo:
    char velikoSlovo = toupper(znak);
    // veliko slovo vratimo u tekst na istu poziciju:
    tekst[pozicija] = velikoSlovo;

    // ispišimo promijenjeni tekst:
    cout << tekst << endl;
    return 0;
}
```

Nakon što smo tekst učitali, pomoću funkcije `find_first_not_of()` tražimo u njemu prvi znak različit od praznine da bismo ga pretvorili u veliko slovo. Kao što se iz imena funkcije može iščitati („nađi prvog koji nije”), ona traži poziciju prvog znaka koji nije iz skupa znakova navedenih unutar zagrada u pozivu funkcije. U našem primjeru taj se skup sastoji od samo jednog znaka – bjeline (`' '`). Funkcija kao rezultat vraća redni broj odgovarajućeg znaka. Važno je uočiti način na koji pozivamo funkciju:

```
tekst.find_first_not_of(' ');
```

Budući da funkcija pripada objektu (za takvu funkciju se kaže da je *funkcijski član* objekta), moramo ispred imena funkcije navesti ime objekta (`tekst`) nad kojim se funkcija izvodi; ime objekta i ime funkcijskog člana odvojeni su međusobno točkom.

Pomoću pozicije koju je vratila funkcija `find_first_not_of()` dohvaćamo znak tako da tu poziciju navedemo unutar uglatih zagrada iza imena objekta koji sadrži tekst te taj znak pridružujemo varijabli `znak`. Potom pozivamo funkciju `toupper()` koja kao rezultat vraća veliko slovo proslijeđenog znaka. Dobiveno veliko slovo vraćamo u tekst na istu poziciju pa će ono zamijeniti izvorno malo slovo.

Gornji kôd ima jedan nedostatak: unesemo li prazan redak koji se sastoji samo od bjelina, program će se „srušiti”. Naime, da bi pokazala da nije uspjela pronaći niti jedan znak koji zadovoljava zadane uvjete, funkcija `find_first_not_of()` u takvim slučajevima vraća posebnu vrijednost jednaku najvećem cijelom broju koji se može na dotičnom računalu prikazati. Prihvatimo li tu vrijednost kao poziciju znaka u tekstu i pokušamo preko nje dohvatiti znak, naš program će zaći u dio memorije koji mu ne pripada.

Operacijski sustav računala će prepoznati takvo nedozvoljeno brljanje po memoriji i nasilno prekinuti izvođenje programa. Da to izbjegnemo, moramo provjeriti vrijednost koju funkcija vraća i dozvoliti promjenu slova samo ako je pozicija ispravna. Budući da specijalna konstanta koju funkcija vraća u slučaju neuspjele pretrage ima oznaku `string::npos`, u programu moramo usporediti vraćenu poziciju s tom konstantom i dozvoliti promjenu slova samo ako pozicija nije jednaka toj konstanti:

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    setlocale(LC_TYPE, "croatian");

    // deklariramo objekt tipa string u koji ćemo učitati tekst:
    string tekst;
    // učitavamo cijeli redak upisanog teksta:
    getline(cin, tekst);

    // tražimo poziciju prvog znaka iza eventualnih bjelina
    // na početku rečenice:
    size_t pozicija = tekst.find_first_not_of(' ');
    // ako je znak pronađen:
    if (pozicija != string::npos)
    {
        // iz teksta očitamo znak na pronađenoj poziciji:
        char znak = tekst[pozicija];
        // znak pretvorimo u veliko slovo:
        char velikoSlovo = toupper(slovo);
        // veliko slovo vratimo u tekst na istu poziciju:
        tekst[pozicija] = velikoSlovo;
    }

    // ispišimo promijenjeni tekst:
    cout << tekst << endl;
    return 0;
}
```


Učitani tekst općenito se može sastojati od više rečenica. Napravimo stoga poboljšanu inačicu koja se neće zaustaviti samo na jednoj rečenici: nakon što promijeni slovo u početnoj rečenici, program mora potražiti točku na kraju rečenice i od nje ponoviti pretragu za prvim znakom različitim od bjeline. Potragu za točkom obavit ćemo pomoću funkcijskog člana `find()`, na sličan način kao što smo to napravili za `find_first_not_of()`. Funkciji `find()` kao argument trebamo proslijediti znak za točku (`'.'`). Ako je točka pronađena, gore opisani postupak promjene početnog slova se ponavlja. Budući da taj postupak treba ponoviti za svaku sljedeću rečenicu, uvjet `if` ćemo zamijeniti naredbom `while` (§4.3.2). Naredba `while` je još jedna naredba koja

omogućava ponavljanje skupa naredbi, ali za razliku od opisane petlje `for`, naredbe se ponavljaju sve dok je uvjet naveden u naredbi zadovoljen. U našem primjeru ona će osigurati da se skup naredbi koji mijenja prvo slovo u rečenici izvede za prvo slovo na početku teksta, a potom ponavlja sve dok još ima točaka koje označavaju kraj rečenice:

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    setlocale(LC_TYPE, "croatian");

    cout << "Upišite neki tekst:" << endl;
    string tekst;
    getline(cin, tekst);

    // tražimo poziciju prvog znaka iza bjelina:
    size_t pozicija = tekst.find_first_not_of(' ');
    // ponavljamo sve dok nismo došli do kraja teksta:
    while (pozicija != string::npos)
    {
        // iz teksta očitamo znak na pronađenoj poziciji:
        char slovo = tekst[pozicija];
        // znak pretvorimo u veliko slovo:
        char velikoSlovo = toupper(slovo);
        // veliko slovo vratimo u tekst na istu poziciju:
        tekst[pozicija] = velikoSlovo;
        // od sljedećeg znaka tražimo točku, tj. kraj rečenice:
        pozicija = tekst.find('.', pozicija + 1);
        // ako je točka nađena, tražimo prvi znak iza nje:
        if (pozicija != string::npos)
            pozicija = tekst.find_first_not_of(' ', pozicija + 1);
    }

    // ispišimo promijenjeni tekst:
    cout << tekst << endl;
    return 0;
}
```

Dosadašnji primjeri poslužili su kao jednostavan prikaz što se i na koji način može napraviti programima pisanima u jeziku C++. U sljedećim poglavljima upoznat ćemo se detaljnije s principima koji su ovdje bili prikazani.

3. Osnovni tipovi podataka

Postoje dva tipa ljudi: jedni koji nose nabijene pištolje, drugi koji kopaju...

Clint Eastwood, u filmu „Dobar, loš, zao”

Svaki program sadrži u sebi podatke koje obrađuje. Njih možemo podijeliti na nepromjenjive *konstante*, odnosno promjenjive *varijable* (*promjenjivice*). Najjednostavniji primjer konstanti su brojevi (5, 10, 3.14159). Varijable su podaci koji općenito mogu mijenjati svoj iznos. Stoga se oni u izvornom kôdu predstavljaju ne svojim iznosom već simboličkom oznakom, *imenom varijable*.

Svaki podatak ima dodijeljenu oznaku tipa koja govori o tome kako se dotični podatak pohranjuje u memoriju računala, koji su njegovi dozvoljeni rasponi vrijednosti, kakve su operacije moguće s tim podatkom i sl. Tako razlikujemo cjelobrojne, decimalne, logičke, pokazivačke tipove podatke. U poglavlju koje slijedi upoznat ćemo se ugrađenim tipovima podataka i pripadajućim operatorima.

3.1. Identifikatori

Mnogim dijelovima C++ programa (objektima, funkcijama, klasama) potrebno je dati određeno ime, tzv. *identifikator* (engl. *identifier*). Imena koja dodjeljujemo su proizvoljna, uz uvjet da se poštuju sljedeća tri osnovna pravila:

1. Identifikator može biti sastavljen od kombinacije slova, znamenki (`0` - `9`) i znaka za podcrtavanje '`_`' (engl. *underscore*);
2. Prvi znak mora biti slovo ili znak za podcrtavanje;
3. Identifikator ne smije biti jednak nekoj od ključnih riječi (vidi tablicu 3.1) ili nekoj od alternativnih oznaka operatora (tablica 3.2). To ne znači da ključna riječ ne može biti dio identifikatora – `moj_int` je dozvoljeni identifikator iako je `int` ključna riječ. Također, treba izbjegavati da naziv identifikatora sadrži dvostruke znakove podcrtavanja (`__`) ili da započinje znakom podcrtavanja i velikim slovom, jer su takve oznake rezervirane za C++ implementacije i standardne biblioteke (npr. `__LINE__`, `__FILE__`).

Stoga si možemo pustiti mašti na volju pa svoje varijable i funkcije nazivati svakojako. Pritom je vjerojatno svakom jasno da je zbog razumljivosti kôda poželjno imena odabirati tako da odražavaju stvarno značenje varijabli, na primjer:

```
pribrojnik1
pribrojnik2
rezultat
```

Tablica 3.1. Ključne riječi jezika C++

alignas	do	new	this
alignof	double	noexcept	thread_local
asm	dynamic_cast	nullptr	throw
auto	else	operator	true
bool	enum	private	try
break	explicit	protected	typedef
case	export	public	typeid
catch	extern	register	typename
char	false	reinterpret_cast	union
char16_t	float	return	unsigned
char32_t	for	short	using
class	friend	signed	virtual
const	goto	sizeof	void
constexpr	if	static	volatile
const_cast	inline	static_assert	wchar_t
continue	int	static_cast	while
decltype	long	struct	
default	mutable	switch	
delete	namespace	template	

a izbjegavati imena poput:

```
snjeguljica_i_7_patuljaka
mojaPrivatnaVarijabla
frankieGoesToHollywood
rockyXXVII
```

Iako standard dozvoljava upotrebu jezično-specifičnih slova u identifikatorima, radi predostrožnosti dobro je ograničiti se na slova engleske abecede (`A - Z`, `a - z`). Naime, mnogi prevoditelji (još uvijek) ne podržavaju u potpunosti ne-engleska slova u imenima. Čak i ako koristite prevoditelja koji prihvaća takve identifikatore, zbog prenosivosti kôda te znakove izbjegavajte; program s varijablama `teksaškiMasakrMotornjačom` ili `bežiJankec` na mnogim prevoditeljima prouzročit će pogrešku prilikom prevođenja.

Valja uočiti da jezik C++ razlikuje velika i mala slova u imenima, tako da sljedeći nazivi predstavljaju tri različita identifikatora:

```
maliNarodiVelikeIdeje
MaliNarodiVelikeIdeje
malinarodiVELIKEIDEJE
```

Tablica 3.2. Alternativne oznake operatora

and	bitand	compl	not_eq	or_eq	xor_eq
and_eq	bitor	not	or	xor	

Ponekad je zbog boljeg razumijevanja kôda pogodno koristiti imena koja su složena od više riječi. Iz gornjih primjera čitateljica ili čitatelj mogli su razlučiti dva najčešća pristupa označavanju složenih imena. U prvom pristupu riječi od kojih je ime sastavljeno odvajaju se znakom za podcrtavanje, poput:

```
snjeguljica_te_sedam_patuljaka
```

Uočimo kako bi razmak umjesto znaka podcrtavanja između riječi označavao prekid imena, što bi uzrokovalo pogrešku prilikom prevođenja.

U drugom pristupu riječi od kojih je ime složeno pišu se spojeno, s velikim početnim slovom:

```
snjeguljicaTeSedamPatuljaka
```

Ovaj način pisanja u engleskoj terminologiji naziva se *CamelCase* (engl. *camel* - deva, engl. *case* - veličina slova) budući da podsjeća na devine grbe. Među C++ programerima danas je popularniji potonji pristup slaganju imena jer takva imena zauzimaju ipak nešto manje mjesta u izvornom kôdu, a znak podcrtavanja je na tipkovnici s engleskim rasporedom znakova prilično nepristupačan. Stoga ćemo ga i mi koristiti u primjerima u knjizi.

Standard ne ograničava broj znakova u imenima, no mnogi prevoditelji interpretiraju imena samo do određenog broja znakova. Srećom, taj broj je kod današnjih prevoditelja dovoljno velik (preko 1024 znaka) da ne moramo paziti na duljinu imena. Budući da se sva imena, koliko god dugačka bila, u konačnici prevode u memorijske adrese, duljina imena neće imati nikakvog utjecaja na veličinu prevedenog programa ili brzinu njegova izvođenja. Predugačka imena stoga treba izbjegavati isključivo radi svoje komocije prilikom tipkanja kôda. Pritom je dobro držati se pravila koje kaže da se za objekte, varijable ili funkcije koji se dohvaćaju u širem dijelu kôda ili izvana koriste dulja i preciznija imena, dok se za elemente vidljive samo unutar neke funkcije ili bloka mogu upotrijebiti kraća imena.

Iako ne postoji nikakvo ograničenje na početno slovo naziva varijabli, većina programera preuzela je iz programskog jezika FORTRAN pravilo da imena cjelobrojnih varijabli započinje slovima `i`, `j`, `k`, `l`, `m` ili `n`. Također, uobičajilo se da imena varijabli počinju malim slovom te ćemo se kroz knjigu nastojati držati tog običaja.

Svojedobno je bilo popularno korištenje tzv. *mađarske notacije*[1] (engl. *Hungarian notation*) u kojoj su prva slova u imenu varijable opisivala njen tip. Na primjer, `pszName` je ime za pokazivač na niz znakova (***p**ointer to **z**ero-terminated **s**tring*). Programer u tom slučaju ne mora tražiti gdje je deklarirana varijabla da bi vidio kojeg je ona tipa, već tip može lagano iščitati iz imena. Mađarsku notaciju koristili su programeri koji su radili na razvoju Windowsa u Microsoftu pa se odatle proširila. Međutim, danas se ta notacija smatra nepotrebnom budući da današnje razvojne okoline mogu jednostavno pružiti informaciju o tipu podatka, bez potrebe da ga programer „dešifrira” iz imena. Osim

[1] Notaciju je izmislio Charles (Károly) Simonyi, računalni stručnjak rođen u Mađarskoj, po čemu je notacija dobila i ime.

toga, ona je nepraktična ako tijekom razvoja programa treba promijeniti tip varijable. Dosljedno korištenje mađarske notacije bi značilo da pri svakoj takvoj promjeni treba mijenjati i ime, što može biti nezgodno ako se ono koristi na puno mjesta.

I na kraju, ono što smo predložili za tekst komentara, zbog istih razloga vrijedi primijeniti i na identifikatore:

Za identifikatore koristite engleske nazive.

Mi ćemo hrvatske nazive koristiti isključivo zbog razumljivosti i lakšeg razlučivanja od ključnih riječi jezika C++.

3.2. Varijable, objekti i tipovi

Bez obzira na jezik u kojem je pisan, svaki program sastoji se od niza naredbi koje mijenjaju vrijednosti podataka (*objekata*) pohranjenih u memoriji računala. Računalo dijelove memorije u kojima su smješteni objekti razlikuje pomoću pripadajuće memorijske adrese. Da programer tijekom pisanja programa ne bi morao pamtiti memorijske adrese, svi programski jezici omogućavaju da se vrijednosti objekata dohvaćaju preko simboličkih naziva razumljivih inteligentnim bićima poput ljudi-programera. Gledano iz perspektive običnog računala (lat. *computer vulgaris ex terae Tai-Wan*), objekt je samo dio memorije u koji je pohranjena njegova vrijednost u binarnom obliku.

Svaki objekt u programskom kôdu je nekog *tipa*. Općenito gledano, tip određuje koje se operacije mogu izvesti nad objektima dotičnog tipa i koji će biti rezultat tih operacija. U konkretnim implementacijama prevoditelja, tip definira i način na koji će se objekt pohraniti u memoriju, uključujući duljinu i raspored bitova u memoriji. Za ugrađene brojčane tipove to će određivati broj točnih znamenki i raspon vrijednosti koji se podacima tog tipa mogu prikazati.

U programskom jeziku C++ pod objektima u užem smislu riječi obično se podrazumijevaju složeni tipovi podataka opisani pomoću posebnih „receptura” – *klasa* (*klasa*), što će biti opisano u kasnijim poglavljima. Jednostavni objekti koji pamte jedan cijeli ili decimalni broj se često nazivaju varijablama. Radi općenitosti ćemo u daljnjem tekstu najčešće govoriti o objektima, iako će to uključivati i brojčane podatke.

3.2.1. Deklaracija

Da bi prevoditelj pravilno preveo naš izvorni C++ kôd u strojni jezik, svaki objekt treba prije njegova korištenja u kôdu *deklarirati*, odnosno jednoznačno odrediti njegov tip. U „Našem trećem C++ programu” varijable su bile deklarirane naredbama:

```
int a;
// ...
int b;
```

```
// ...
int c;
```

Tim deklaracijama je prevoditelju jasno dano do znanja da će imena `a`, `b` i `c` predstavljati cjelobrojne varijable – prevoditelj će vrijednosti tih varijabli pohranjivati u memoriju prema svom pravilu za cjelobrojne podatke. Također će znati kako treba provesti operacije na njima.

Prilikom deklaracije objekta treba paziti da se u dijelu programa u kojem je on vidljiv (npr. unutar funkcije `main()`) ne smije deklarirati drugi objekt s istim imenom, čak i ako je drugog tipa. Zato će, prilikom prevođenja sljedećeg kôda, prevoditelj javiti pogrešku o višekratnoj deklaraciji objekta s imenom `a`:

```
int a;
string a;               // pogreška: ponovno korištenje imena a
```

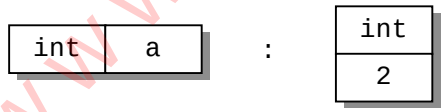
U protivnom, kada bi bilo dozvoljeno koristiti isto ime za više objekata, prilikom dohvaćanja objekta preko imena prevoditelj ne bi mogao razlučiti koji od istoimenih objekta doista želimo dohvatiti.

3.2.2. Inicijalizacija

Varijable postaju realna zbiljnost tek kada im se pokuša pristupiti, na primjer kada im se pridruži vrijednost:

```
a = 2;
b = a;
c = a + b;
```

Prevoditelj tek tada pridružuje memorijski prostor u koji se pohranjuju podaci. Postupak deklaracije varijable `a` i pridruživanja vrijednosti simbolički možemo prikazati slikom 3.1. Lijevo od dvotočke je kućica koja simbolizira varijablu s njenim tipom i imenom.

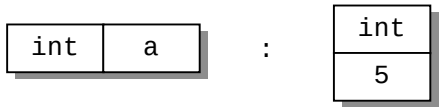


Slika 3.1. Deklaracija varijable i pridruživanje vrijednosti

Desno od dvotočke je kućica koja predstavlja konkretni objekt s njegovim tipom i vrijednošću. Ako nakon toga istoj varijabli `a` pridružimo novu vrijednost naredbom:

```
a = 5;
```

tada se u memoriju računala na mjestu gdje je prije bio smješten broj 2 pohranjuje broj 5, kako je prikazano slikom 3.2.



Slika 3.2. Pridruživanje nove vrijednosti

Deklaracija varijable i pridruživanje vrijednosti mogu se obaviti u istom retku kôda. Umjesto da pišemo poseban redak s deklaracijama varijabli `a`, `b` i `c` i zatim im pridružujemo vrijednosti, *inicijalizaciju* možemo provesti ovako:

```
int a = 2;
int b = a;
int c = a + b;
```

Inicijalizacija pomoću operatora pridruživanja `=` naslijeđena je iz jezika C i uobičajena je za ugrađene tipove podataka. Međutim, prilikom ovakvog pridruživanja moguće su implicitne pretvorbe koje često rezultiraju gubitkom dijela podatka. Takve pretvorbe nazivaju se *sužavajuće pretvorbe* (engl. *narrowing conversion*) Na primjer, naredbom:

```
int a = 3.14;   // cjelobrojnoj varijabli pridružujemo decimalni broj!
```

cjelobrojnoj varijabli se pridružuje decimalni broj. Budući da cijeli brojevi ne poznaju decimalne znamenke, one će se prilikom pridruživanja izgubiti. Prevoditelj će generirati izvedbeni kôd koji će decimalni broj pretvoriti u cijeli, odbacujući pritom decimalne znamenke, pa će varijabla a poprimiti vrijednost 3.

Iako je u gornjoj naredbi ovakav rasplet prilično očekivan i očigledan, čest je slučaj da se implicitna pretvorba provede, a da programer nje ne bude svjestan. Doduše, neki prevoditelji će na ovakvo pridruživanje izbaciti upozorenje oblika: „pretvorba iz tipa A u tip B – mogući gubitak podataka”. Da bi se izbjegla ovakva neugodna iznenađenja, standard C++11 uveo je novu sintaksu za inicijalizaciju koja onemogućava implicitne pretvorbe. Vrijednosti kojima se objekt inicijalizira navode se u *listi inicijalizatora* (engl. *initializer list*) omeđenoj vitičastim zagradama `{}`:

```
int a{2};       // deklaracija i inicijalizacija varijable a
```

Lista inicijalizatora u ovom slučaju sadrži samo jedan član jer je za inicijalizaciju cijelog broja potrebna samo jedna vrijednost. Za složenije tipove lista inicijalizatora će obično sadržavati više članova.

Uoči li prevoditelj da je za inicijalizaciju neophodno provesti sužavajuću pretvorbu, prijavit će pogrešku:

```
int b{3.14};    // pogreška: sužavajuća pretvorba!
```

Prethodne dvije inicijalizacije možemo napisati i na sljedeći način:

```
int a = {5};    // isto kao i: int a {5};
int b = {3.0};  // pogreška: sužavajuća pretvorba!
```

no znak pridruživanja = kod ovakve inicijalizacije nije neophodan. Primijetimo kako u zadnjoj naredbi, kojom inicijaliziramo cijeli broj `d`, zbog decimalne točke prevoditelj broj `3.0` shvaća kao decimalni, bez obzira što on nema značajnih decimalnih znamenki. Budući da svaka pretvorba decimalnog broja u cijeli općenito povlači mogući gubitak podatka, prevoditelj će prijaviti pogrešku, ne vodeći računa o konkretnoj vrijednosti. Prevoditelj ne barata s vrijednostima već samo s tipovima podataka.

Kako su se već u prijašnjim verzijama jezika C++ nizovi podataka (§5.1) mogli inicijalizirati listama inicijalizatora, proširenjem njihove primjene na pojedinačne objekte postignuta je jednoobrazna sintaksa za inicijalizaciju svih tipova, bez obzira radi li se o pojedinačnom podatku ili o nizu podataka. Zbog svega navedenoga, u knjizi ćemo davati prednost ovakvoj inicijalizaciji (iako će nam zasigurno negdje promaknuti i „stari" oblik):

Objekte je preporučljivo inicijalizirati pomoću liste inicijalizatora.

Proslijedimo li objektu praznu listu inicijalizatora:

```
int a{};        // inicijalizira na 0
```

on će biti inicijaliziran na podrazumijevanu vrijednost. Za brojčane tipove to znači da će oni biti inicijalizirani na nulu.

Spomenimo kako postoji i treći oblik zapisa naredbi za inicijalizaciju, takozvanom *konstruktorskom sintaksom*`()`:

```
int a(2);
int b(a);
```

Ovakav zapis se koristio prvenstveno za složene korisnički definirane tipove čija inicijalizacija iziskuje više od jednog parametra. Primjerice, da bi se inicijalizirao kompleksni broj, općenito treba zadati dva parametra: vrijednost realnog i vrijednost imaginarnog dijela. Operatorom pridruživanja inicijalizacija takvog objekata nije moguća. Detaljnije o ovakvom načinu inicijalizacije govorit ćemo u poglavlju o korisnički definiranim tipovima – *klasama*. Napomenimo da i ovaj način inicijalizacije pati od istih boljki kao inicijalizacija operatorom pridruživanja (prešutna sužavajuća pretvorba) pa ju valja izbjegavati gdje je god to moguće.

Objekt mora uvijek biti deklariran prije nego što se prvi puta upotrijebi. Iako deklaracija može biti navedena bilo gdje unutar programa, uz uvjet da je ta deklaracija vidljiva svim naredbama koje taj objekt koriste (o vidljivosti, odnosno dosegu imena govorit ćemo detaljnije u kasnijim poglavljima), dobra je navika držati se sljedećeg pravila:

Objekte je najbolje deklarirati neposredno prije njihove prve primjene.

Time ograničavamo vidljivost imena na minimalno potrebno područje pa je lakše pratiti tko sve koristi i mijenja dotični objekt. Osim toga, ako nam taj objekt iz nekog razloga postane suvišan u kôdu i uklonimo sve naredbe koje ga koriste, lakše ćemo uočiti njegovu deklaraciju te i nju maknuti iz kôda.

Zbog istih razloga treba izbjegavati deklaraciju više objekata istom naredbom. Na primjer, u „Mom trećem programu" smo umjesto tri zasebne deklaracije, varijable `a`, `b` i `c` mogli deklarirati jednom naredbom na početku programa, prije prve naredbe koja koristi jednu od varijabli:

```
// ...
int main()
{
    int a, b, c;        // deklariramo sve varijable na jednom mjestu!
    cout << "Upiši prvi broj:";
    cin >> a;
    // ...
}
```

Zadatak: *U „Mom trećem programu" deklarirajte sve tri varijable jednom naredbom na početku funkcije* `main()` *i prevedite program. Potom prebacite tu deklaraciju u sredinu, odnosno na kraj funkcije* `main()` *(ispred naredbe* `return`*) te pogledajte pogreške koje će prevoditelj prijaviti.*

3.2.3. Neinicijalizirane varijable

Za varijablu kojoj se pristupa prije nego što joj je pridružena vrijednost kaže se da je *neinicijalizirana*:

```
int a;
cout << a << endl;  // ispisujemo vrijednost neinicijaliziranog objekta
```

Budući da varijabla `a` nije inicijalizirana, njen sadržaj je nepredvidiv jer će biti određen trenutnim stanjem bitova u memoriji koju je varijabla zauzela. Stoga je nemoguće unaprijed znati što će gornji ispis dati i on će vrlo vjerojatno biti drugačiji svaki puta kada ponovno izvedemo te naredbe.

U pravilu želimo da se naš program ponaša jednako svaki puta kada ga pokrenemo. Zatrebaju li nam u programu elementi slučajnosti, to ćemo daleko djelotvornije napraviti korištenjem odgovarajućih funkcija koje generiraju slučajne brojeve, kako će biti prikazano kasnije u knjizi. Iako će mnogi prevoditelji za naredbe u kojima se pristupa neinicijaliziranim varijablama prijaviti pogrešku ili barem upozorenje, uvijek valja biti na oprezu. Naime, neinicijalizirane varijable mogu prouzročiti velike probleme i u kompleksnijim programima im je teško ući u trag. Zato je dobro držati se sljedećeg pravila:

Kada god je to moguće, varijable treba inicijalizirati odmah prilikom njihove deklaracije.

Gornje razmatranje vrijedi samo za lokalne varijable, poput varijabli deklariranih unutar funkcije `main()`. Osim lokalnih objekata (§8.8.1), u programu možemo definirati i globalne objekte (§8.8.2) te statičke objekte (§8.8.3) i statičke članove klasa (§9.8.1). Te objekte inicijalizira prevoditelj prilikom generiranja izvedbenog kôda i to odmah u naredbi u kojoj su deklarirani. Ako se ne navede lista inicijalizatora, ti objekti će biti inicijalizirani na podrazumijevanu vrijednost, tj. kao da smo naveli praznu listu `{}`. Na primjer, u sljedećem kôdu je varijabla `globA` deklarirana izvan funkcije `main()` pa je ona globalna. Zbog toga će ispis njene vrijednosti dati 0:

```
#include <iostream>
using namespace std;

int globA;          // deklaracija globalne varijable

int main()
{
    cout << globA << endl;      // ispisat će 0
    return 0;
}
```

3.3. Operator pridruživanja

Operatorom pridruživanja mijenja se vrijednost nekog objekta, pri čemu tip objekta ostaje nepromijenjen. Najčešći operator pridruživanja je znak jednakosti (=) kojim se objektu na lijevoj strani pridružuje neka vrijednost s desne strane. Ako su objekt s lijeve strane i vrijednost s desne strane različitih tipova, vrijednost se svodi na zajednički tip prema definiranim pravilima pretvorbe, što će biti objašnjeno u odjeljku 3.4. Očito je da s lijeve strane operatora pridruživanja mogu biti isključivo promjenjivi objekti, pa se stoga ne može pisati ono što je inače matematički korektno:

```
2 = 4 / 2           // pogreška!!!
3 * 4 = 12          // pogreška!!!
3.14159 = pi        // Bingooo!!! Treća pogreška!
```

Pokušamo li prevesti ovaj kôd, prevoditelj će javiti pogreške. Objekti koji se smiju nalaziti s lijeve strane znaka pridruživanja nazivaju se *l-vrijednosti* (engl. *lvalues*, kratica od *left-hand side values* - vrijednosti slijeva). Pritom valja znati da se ne može svakoj l-vrijednosti pridruživati nova vrijednost – neke varijable se mogu deklarirati kao konstantne (§3.4.4) i pokušaj promjene njihove vrijednosti prevoditelj će naznačiti kao pogrešku. Stoga se posebno govori o *promjenjivim l-vrijednostima*. S desne strane operatora pridruživanja mogu biti i l-vrijednosti i konstante. Te vrijednosti se skupno nazi-

vaju *r-vrijednosti*[1] (engl. *rvalues*, kratica od *right-hand side values* - vrijednosti zdesna). L-vrijednosti možemo shvatiti kao objekte koji imaju svoje ime (preciznije rečeno: svoju adresu u memoriji) i zbog toga mogu postojati kroz više od jedne naredbe, dok su r-vrijednosti samo privremeni objekti koji postoje isključivo unutar izraza u kojem se navode. Na primjer, u naredbi:

```
int b = a + 3;
```

`x` je l-vrijednost, dok je izraz `a + 3` r-vrijednost (koji sadrži l-vrijednost `a`).

Evo tipičnog primjera pridruživanja kod kojeg se ista varijabla nalazi i s lijeve i s desne strane znaka jednakosti:

```
int i = 5;
i = i + 3;
```

Naredbom u prvom retku deklarira se varijabla tipa `int`, te se njena vrijednost inicijalizira na 5. Dosljedno gledano, operator `=` ovdje nema značenje pridruživanja, jer se inicijalizacija varijable `i` provodi već tijekom prevođenja, a ne prilikom izvođenja programa. To znači da je broj 5 ugrađen u izvedbeni strojni kôd dobiven prevođenjem i povezivanjem programa. Obratimo, međutim, pažnju na drugu naredbu!

Matematički gledano, drugi redak u ovom primjeru je besmislen: nema broja koji bi bio jednak samom sebi uvećanom za 3! Ako ga pak gledamo kao uputu računalu što mu je činiti, onda taj redak treba početi čitati neposredno iza znaka pridruživanja: „uzmi vrijednost varijable `i`, dodaj joj broj 3...". Došli smo do kraja naredbe (znak `;`), pa se vraćamo na operator pridruživanja: „...i dobiveni zbroj s desne strane pridruži varijabli `i` koja se nalazi s lijeve strane znaka jednakosti". Nakon izvedene naredbe varijabla `i` imat će vrijednost 8.

Jezik C++ dozvoljava više operatora pridruživanja u istoj naredbi. Pritom pridruživanje ide od krajnjeg desnog operatora prema lijevo:

```
a = b = c = 0;
```

te ih je tako najsigurnije i čitati: „broj `0` pridruži varijabli `c`, čiju vrijednost pridruži varijabli `b`, čiju vrijednost pridruži varijabli `a`". Budući da se svakom od objekata lijevo od znaka `=` pridružuje neka vrijednost, svi objekti izuzev onih koji se nalaze desno od krajnjeg desnog znaka `=` moraju biti l-vrijednosti:

```
a = b = c + d;      // OK!
a = b + 1 = c;      // pogreška: b + 1 nije l-vrijednost
```

[1] Pravilan prijevod bio bi *d-vrijednost* (od *zdesna*), no zbog lakšeg povezivanja s engleskim izvornikom upotrijebili smo naziv *r-vrijednost*.

3.4. Tipovi podataka i operatori

U C++ jeziku ugrađeni su neki osnovni tipovi podataka i definirane operacije na njima. Za te su tipove precizno definirana pravila provjere i pretvorbe. *Pravila provjere tipa* (engl. *type-checking rules*) uočavaju neispravne operacije nad objektima, dok *pravila pretvorbe* (engl. *conversion rules*) određuju što će se dogoditi ako neka operacija očekuje jedan tip podataka, a umjesto njega se pojavi drugi. U sljedećim poglavljima prvo ćemo se upoznati s brojevima i operacijama na njima, da bismo kasnije prešli na pobrojenja, logičke vrijednosti i znakove.

3.4.1. Brojevi

U jeziku C++ ugrađena su u suštini dva osnovna tipa brojeva: cijeli brojevi (engl. *integers*) i decimalni brojevi (tzv. *brojevi s pomičnom decimalnom točkom*, engl. *floating-point*). Najjednostavniji tip brojeva su cijeli brojevi – njih smo već upoznali u „Našem trećem C++ programu". Cjelobrojna varijabla deklarira se ključnom riječju `int` i njena će vrijednost u memoriji računala zauzeti najmanje dva *bajta*[1] (engl. *byte*), tj. 16 bitova. Prvi bit je uvijek rezerviran za predznak, a preostali bitovi služe za pohranu vrijednosti. Valja napomenuti da standardom nije strogo određena duljina za `int`, već je to prepušteno implementaciji prevoditelja. Danas je za 32-bitne operacijske sustave `int` uobičajeno duljine 4 bajta (32 bita) pa se njime mogu obuhvatiti svi brojevi od najmanjeg broja (najvećeg negativnog broja)

$$-2^{31} = -2\,147\,483\,648$$

do najvećeg broja

$$2^{31} - 1 = 2\,147\,483\,647$$

Za većinu praktičnih primjena taj opseg vrijednosti je dostatan. Međutim, ukaže li se potreba za većim cijelim brojevima varijablu možemo deklarirati kao `long long int`, ili kraće samo `long long`:

```
long long int zrnacaPrasineNaMomMonitoru;
              // mogao bih uzeti krpu za prašinu i svesti to na int!
```

Takva varijabla u memoriji zauzima barem 8 bajtova (64 bita) i njome se mogu prikazati brojevi u rasponu od –9 223 372 036 854 775 808 do 9 223 372 036 854 775 807.

Kao što smo rekli, standard ne određuje koje su duljine pojedinih cjelobrojnih tipova, nego samo zahtijeva da tip `int` bude barem duljine 16 bita. Stvarne duljine su prepuštene implementacijama prevoditelja. Ako prevoditelj koristi minimalnu duljinu od 16 bita (koja pokriva raspon vrijednosti od –32768 do 32767), za veće cijele brojeve može se upotrijebiti tip `long int`, ili kraće samo `long` koji je u pravilu duljine 32-bita. Očito će na većini 32-bitnih sustava duljine tipova `long int` i `long long int` biti međusobno jednake.

[1] Ponekad se može naići na naziv *oktet*, budući da se bajt najčešće sastoji od 8 bitova. U engleskom riječ byte nema nikakvo izvorno značenje, već je samo umjetna modifikacija riječi *bite* (engl. *ugriz*), koja po obliku podsjeća na riječ *bit* (engl. *komadić, trunka*).

Cjelobrojne konstante se mogu u izvornom kôdu prikazati u različitima brojevnim sustavima:

- dekadskom,
- binarnom,
- oktalnom,
- heksadekadskom.

Dekadski prikaz je razumljiv većini običnih „nehakerskih” smrtnika koji se ne spuštaju na razinu računala. Međutim, kada treba raditi operacije nad pojedinim bitovima, binarni, oktalni i heksadekadski prikazi su daleko primjereniji.

U dekadskom brojevnom sustavu, koji koristimo svakodnevno, postoji 10 različitih znamenki (0, 1, 2,..., 9) čijom kombinacijom se dobiva bilo koji željeni višeznamenkasti broj. Pritom svaka znamenka ima deset puta veću težinu od znamenke do nje desno. U binarnom sustavu, znamenke mogu biti samo 0 i 1. U oktalnom brojevnom sustavu broj znamenki je ograničen na 8 (0, 1, 2,..., 7), tako da svaka znamenka ima osam puta veću težinu od svog desnog susjeda (vidi tablicu 3.3). Na primjer, 11 u oktalnom sustavu odgovara broju 9 u dekadskom sustavu ($11_8 = 9_{10}$ – donji indeks označava bazu za prikaz). U heksadekadskom sustavu postoji 16 znamenki: 0, 1, 2,..., 9, A, B, C, D, E, F. Budući da za prikaz brojeva postoji samo 10 brojčanih simbola, za prikaz heksadekadskih znamenki iznad 9 koriste se prva slova engleskog alfabeta, tako da je $A_{16} = 10_{10}$, $B_{16} = 11_{10}$, itd. Oktalni i heksadekadski prikaz predstavljaju kompromis između dekadskog prikaza kojim se koriste ljudi i binarnog sustava kojim se podaci pohranjuju u memoriju računala. Naime, binarni prikaz pomoću samo dvije znamenke (0 i 1) iziskuje puno prostora u programskom kôdu, te je često nepregledan. Oktalni i heksadekadski prikazi iziskuju manje prostora i iskusnom korisniku omogućuju brzu pretvorbu brojeva iz dekadskog u binarni oblik i obrnuto.

Tablica 3.3. Usporedba brojeva u različitim sustavima

Dekadski	Binarno	Oktalno	Heksadekadski
1	1	1	1
2	10	2	2
⋮	⋮	⋮	⋮
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
17	10001	21	11
⋮	⋮	⋮	⋮
32	100000	40	20

Binarne konstante se pišu tako da se ispred prve znamenke napiše slog `0b` ili `0B` (nula i malo ili veliko slovo 'b'):

```
int samuraja = 0b111;                // odgovara dekadskom 7
```

Oktalne konstante se pišu tako da se ispred prve znamenke napiše broj `0` iza kojeg slijedi oktalni prikaz broja:

```
int snjeguljicaIPatuljci = 010;      // odgovara dekadskom 8
```

Heksadekadske konstante započinju s `0x` ili `0X`:

```
int tucetPatuljaka = 0x0C;           // dekadsko 12
```

Slova za heksadekadske znamenke mogu biti velika ili mala.

Vodeća nula kod oktalnih brojeva uzrokom je čestih početničkih pogrešaka, jer korisnik obično smatra da će ju prevoditelj zanemariti i interpretirati broj kao obični dekadski.

Sve brojčane konstante koje započinju s brojem `0` prevoditelj interpretira kao oktalne brojeve. To znači da će nakon prevođenja `010` i `10` biti dva različita broja!

Još jednom uočimo da će bez obzira na brojevni sustav u kojem broj zadajemo, njegova vrijednost u memoriju biti pohranjena prema predlošku koji je određen deklaracijom pripadne varijable. To najbolje ilustrira sljedeći primjer:

```
int i{32};
cout << i << endl;

i = 040;                 // zadajemo 32 u oktalnom prikazu
cout << i << endl;

i = 0x20;                // zadajemo 32 u heksadekadskom prikazu
cout << i << endl;
```

Bez obzira što smo varijabli `i` pridruživali broj 32 u tri različita prikaza, u sva tri slučaja na zaslonu će se ispisati u dekadskom formatu isti broj.

Ako je potrebno računati s decimalnim brojevima, najčešće se koriste varijable tipa `double`:

```
double pi{3.141592653};
double brzinaSvjetlosti{2.997925e8};
double nabojElektrona{-1.6E-19};
```

Prvi primjer odgovara uobičajenom načinu prikaza brojeva s decimalnim zarezom (osim što se koristi decimalna točka, a ne zarez!). U drugom i trećem primjeru brojevi su pri-

kazani u znanstvenoj notaciji, kao umnožak mantise i potencije na bazi 10 ($2{,}997925 \cdot 10^{8}$, odnosno $-1{,}6 \cdot 10^{-19}$). Kod prikaza u znanstvenoj notaciji, slovo 'e' koje razdvaja mantisu od eksponenta može biti veliko ili malo.

Praznine unutar broja, na primjer iza predznaka, ili između znamenki i slova ‘e’ nisu dozvoljene.

Prevoditelj će broj:

```
double planckovaKonstanta{6.626 e -34};          // pogreška!!!
```

interpretirati samo do četvrte znamenke. Prazninu koja slijedi prevoditelj će shvatiti kao završetak broja, pa će po nailasku na znak `e` javiti pogrešku.

Osim tipa `double`, za rukovanje decimalnim brojevima na raspolaganju su i tipovi `float`, odnosno `long double`. Tipovi `double`, `float` i `long double` međusobno se razlikuju po prostoru koji zauzimaju u memoriji. Kao i za cijele brojeve, standard ne definira veličinu pojedinih tipova, osim što tip `double` mora imati barem istu točnost kao tip `float`, a tip `long double` mora imati barem istu točnost kao tip `double`.

Prilikom rada s decimalnim brojevima treba biti svjestan dva njihova svojstva: raspona vrijednosti i točnosti. Raspon vrijednosti omeđen je najmanjim i najvećim brojem koji se može u dotični tip pohraniti, dok je točnost broj znamenki koje taj tip pamti. Decimalni brojevi se u računalu predstavljaju u obliku:

$$mantisa \times 2^{eksponent},$$

te se u memoriju pohranjuju mantisa i eksponent zajedno sa svojim predznacima. Mantisa i eksponent zauzimaju određeni, za svaki tip različiti broj bitova, što za programera u konačnici znači da će različiti tipovi podržavati različiti opseg vrijednosti i različiti broj točnih znamenki (vidi tablicu 3.4). Na primjer, ako tip `double` zauzima 8 bajtova (64 bita), mantisa će biti predstavljena s 53 bita, a eksponent s preostalih 11 bitova[1]. Odbijemo li bit za predznak, mantisi preostaju 52 bita u koja je moguće pohraniti nešto više od 15 dekadskih znamenki decimalnog broja. Eksponent se pohranjuje kao cijeli broj, a u 11 bitova eksponenta mogu se pohraniti brojevi u rasponu od –1022 do 1023. Dekadski ekvivalenti najmanjem eksponentu je:

$$2^{-1022} \approx 2 \cdot 10^{-308},$$

a najvećem eksponentu:

$$2^{1023} \approx 9 \cdot 10^{307}.$$

Na osnovu ovih razmatranja slijedi da se tipom `double` (uz pretpostavku da zauzima 8 bajtova i da se pohranjuje u skladu sa standardom IEEE 754) mogu prikazati negativni brojevi u rasponu:

$$-2{,}225 \cdot 10^{-308} \text{ do } -1{,}797 \cdot 10^{308}$$

[1] Prema standardu IEEE 754. Većina prevoditelja koristi zapis decimalnih brojeva prema tom standardu.

i pozitivni brojevi u rasponu:

$2{,}225 \cdot 10^{-308}$ do $1{,}797 \cdot 10^{308}$.

Osim raspona vrijednosti koje se u računalu mogu predstaviti decimalnim brojem, treba biti svjestan da je i broj decimalnih znamenki u mantisi ograničen. Za razliku od cijelih brojeva kod kojih se pamte sve zadane znamenke, kod decimalnih brojeva se pamti samo ograničeni broj najvažnijih znamenki. Čak i ako se pridruži broj s više znamenki, prevoditelj će odbaciti sve niže decimalne znamenke koje ne stanu u mantisu. Ilustrirajmo to tipom `float` koji pamti samo 7 najznačajnijih dekadskih znamenki. Zato će u sljedećim inicijalizacijama obje varijable sadržavati potpuno jednake vrijednosti, pa će zadnja naredba ispisati nulu:

```
float piTocniji    = 3.141592654;
float piManjeTocan = 3.1415927;
cout << piTocniji - piManjeTocan << endl;   // ispisuje 0!
```

Iako `double` ima veću duljinu i zauzima više mjesta u memoriji od tipa `float`, operacije s `double` podacima ne moraju nužno biti sporije. Naime, velika većina današnjih procesora ima ugrađene instrukcije za aritmetičke operacije s brojevima tipa `double` pa se te operacije izvode gotovo jednako brzo kao i na kraćim tipovima podataka. Stoga se danas uobičajilo za decimalne brojeve koristiti tip `double`.

U tablici 3.4 dane su tipične duljine ugrađenih brojevnih tipova u bajtovima, rasponi vrijednosti koji se njima mogu obuhvatiti, a za decimalne brojeve i broj točnih decimalnih znamenki. Neki cjelobrojni tipovi imaju i skraćene nazive (koji su daleko popularniji) te su oni u tablici navedeni u zagradama. Valja napomenuti da duljina memorijskog prostora i opseg vrijednosti koje određeni tip varijable zauzima nisu standardizirani te variraju ovisno o prevoditelju i o platformi za koju se prevoditelj koristi. Stoga čitatelju preporučujemo da za svaki slučaj konzultira dokumentaciju uz prevoditelj koji koristi. Relevantni podaci za cjelobrojne tipove mogu se naći u datoteci `climits`, a za decimalne tipove podataka u `cfloat`. Tako, primjerice su konstante `INT_MIN` i `INT_MAX` (definirane pretprocesorskom naredbom `#define` u datoteci `climits`) jednake najmanjoj (najvećoj negativnoj) i najvećoj vrijednosti brojeva tipa `int`.

Ako želite sami u kôdu provjeriti veličinu pojedinog tipa, to možete učiniti i pomoću operatora `sizeof`:

```
cout << "Veličina brojeva tipa int: " << sizeof(int);
cout << "Veličina brojeva tipa double: " << sizeof(double);
long double masaSunca{2e30};
cout << "Veličina broja tipa long double: „ << sizeof(masaSunca);
```

Gornje naredbe će ispisati broj bajtova koliko ih zauzimaju podaci tipa `int`, `double`, odnosno `long double`. Numerička ograničenja pojedinih tipova možete provjeriti ispisom odgovarajućih konstanti iz prije navedenih datoteka `climits` i `cfloat`, na primjer:

Tablica 3.4. Ugrađeni brojevni tipovi, njihove tipične duljine i rasponi vrijednosti

tip konstante	bajtova	raspon vrijednosti	točnost
`char`	1	–128 do 127	
`short int` (`short`)	2	–32768 do 32767	
`int`	2 4	–32768 do 32767 –2147483648 do 2147483647	
`long int` (`long`)	4	–2147483648 do 2147483647	
`long long int` (`long long`)	8	–9223372036854775808 do 9223372036854775807	
`float`	4	$-3{,}40\cdot10^{38}$ do $-1{,}17\cdot10^{-38}$ i $1{,}17\cdot10^{-38}$ do $3{,}40\cdot10^{38}$	7 dek. znamenki
`double`	8	$-1{,}79\cdot10^{308}$ do $-2{,}22\cdot10^{-308}$ i $2{,}22\cdot10^{-308}$ do $1{,}79\cdot10^{308}$	15 dek. znamenki
`long double`	16	$-1{,}18\cdot10^{4932}$ do $-3{,}36\cdot10^{-4932}$ i $3{,}36\cdot10^{-4932}$ do $1{,}18\cdot10^{4932}$	34 dek. znamenke

```
cout << "Najmanji int: " << INT_MIN;
cout << "Najveći int: " << INT_MAX;
cout << "Najmanji long: " << LONG_MIN;
cout << "Najveći long: " << LONG_MAX;
cout << "Najveći float: " << FLT_MAX;
cout << "Broj točnih dekadskih znamenki za float: " << FLT_DIG;
cout << "Najmanji double: " << DBL_MIN;
cout << "Broj točnih dekadskih znamenki za double: " << DBL_DIG;
```

Kada ćete isprobavati gornji kôd, nemojte zaboraviti uključiti zaglavlja `climits` i `cfloat`! Drugi, još elegantniji način jest korištenje klase `numeric_limits` definirane u zaglavlju `limits`:

```
cout << "Najmanji int: " << numeric_limits<int>::min() << endl;
cout << "Najveći int: "  << numeric_limits<int>::max() << endl;
cout << "Najmanji double: " << numeric_limits<double>::min() << endl;
cout << "Najveći double: "  << numeric_limits<double>::max() << endl;
cout << "Broj točnih dekadskih znamenki za long double: "
     << numeric_limits<long double>::digits10 << endl;
```

Radi se o predlošku klase (§12.3) koji omogućava dohvaćanje niza informacija o brojevnim tipovima. Kao parametar predlošku, unutar znakova < i > („manje” i „veće”) u gornjim naredbama smo proslijedili tip za koji tražimo podatke. Iza para dvotočki `::`

navedeni su članovi koji sadrže tražene podatke za dotični tip, u ovom slučaju najmanju (`min`), najveću (`max`) vrijednost te broj točnih dekadskih znamenki (`digits10`).

Zadatak: *Proširite gornji kôd za sve ostale brojčane tipove podataka iz tablice 3.4. Provjerite u dokumentaciji koje još članove sadrži klasa `numeric_limits` pa dodajte ispis vrijednosti koje ti članovi vraćaju.*

Svaki od do sada navedenih brojevnih tipova obuhvaća praktički jednake raspone pozitivnih i negativnih vrijednosti te u sebi sadrži i predznak. Podrazumijevano su ti tipovi deklarirani s ključnom riječi `signed` tako da su sljedeće dvije deklaracije potpuno ekvivalentne:

```
signed int sEksplicitnimPredznakom{-5};
int bezEksplicitnogPredznaka{-25};
```

Budući da je specifikacija `signed` za sve tipove (osim za tip `char` te za polja bitova) redundantna, ona se vrlo rijetko koristi.

Međutim, brojevni tipovi mogu biti deklarirani i bez predznaka, dodavanjem riječi `unsigned` ispred njihove deklaracije:

```
unsigned int i{40000};
unsigned long double brojZvijezdaUSvemiru;
```

U tom slučaju bit za predznak prevoditelj koristi kao dodatnu binarnu znamenku, pa se najveća moguća vrijednost udvostručuje u odnosu na varijable s predznakom, ali se naravno ograničava samo na pozitivne vrijednosti. Pridružimo li `unsigned` varijabli negativan broj, ona će poprimiti vrijednost nekog pozitivnog broja. Na primjer, programski slijed:

```
unsigned int i{-1};
cout << i << endl;
```

će za varijablu `i` ispisati pozitivni broj koji je za 1 manji od najvećeg dozvoljenog `unsigned int` broja, tj. ispisat će 4294967295 (ili 65535). Zanimljivo da prevoditelj za pridruživanje u gornjem kôdu neće prijaviti pogrešku!

Prevoditelj ne prijavljuje pogrešku ako se `unsigned` varijabli pridruži negativna konstanta – korisnik mora sam paziti da se to ne dogodi!

Stoga ni za živu glavu nemojte u programu za evidenciju stanja na tekućem računu koristiti `unsigned` varijable. U protivnom se lako može dogoditi da ostanete bez računa, a svoj bijes iskalite na računalu, ni krivom ni dužnom.

Lako se može dogoditi da uletite u iskušenje upotrijebiti `unsigned` varijablu zato jer omogućava pohranjivanje većih vrijednosti. Čak i ako ste apsolutno sigurni da dotična varijable neće nigdje u programu poprimiti negativnu vrijednost, daleko je bolje rješenje upotrijebiti tip koji podržava širi raspon vrijednosti, na primjer, umjesto tipa `int` upotri-

jebiti tip `long long int`. Dobitak zbog korištenja `unsigned` varijable je zanemariv jer ona može primiti samo dvostruko veću vrijednost od `signed` tipa.

U pravilu se `unsigned` tipovi koriste isključivo za cjelobrojne varijable kojima se mijenjaju stanja pojedinih bitova.

Budući da svi bitovi kod tipa `unsigned` predstavljaju binarne znamenke, zadavanje i očitavanje stanja pojedinih bitova je za programera jednostavnije – za sâm izvedbeni kôd nema nikakve razlike definiramo li tip kao `signed` ili `unsigned`. Operacije kojima možemo dohvaćati i mijenjati pojedine bitove opisane su u odjeljku 7.4.1.

Kao što smo spomenuli, veličina ugrađenih tipova nije strogo definirana standardom. To znači da ih različiti prevoditelji čak i na istom operacijskom sustavu mogu obrađivati različito. U konačnici se može dogoditi da se program dobiven iz istog izvornog kôda nakon prevođenja na različitim prevoditeljima ponaša različito. Da se izbjegnu ovakvi problemi s kompatibilnošću kôda, standard omogućava zadavanje cjelobrojnih tipova preko željene duljine (tablica 3.5). Tako `int8_t` predstavlja tip koji zauzima

Tablica 3.5. Cjelobrojni tipovi zadane duljine

`int8_t`	`int_fast8_t`	`int_least8_t`
`int16_t`	`int_fast16_t`	`int_least16_t`
`int32_t`	`int_fast32_t`	`int_least32_t`
`int64_t`	`int_fast64_t`	`int_least64_t`
`uint8_t`	`uint_fast8_t`	`uint_least8_t`
`uint16_t`	`uint_fast16_t`	`uint_least16_t`
`uint32_t`	`uint_fast32_t`	`uint_least32_t`
`uint64_t`	`uint_fast64_t`	`uint_least64_t`

točno 8 bitova. Valja napomenuti da stavke u tablici ne predstavljaju zasebne tipove, nego u zaglavlju `cstdint` definirane sinonime za ugrađene tipove. Sadržaj tog zaglavlja ovisit će o implementaciji prevoditelja. Primjerice, u nekoj implementaciji bi to zaglavlje moglo sadržavati sljedeće deklaracije:

```
// ...
typedef short int16_t;
typedef int int32_t;
typedef long long int64_t;
// ...
```

Iako još nismo objasnili ključnu riječ `typedef` (§3.4.15), oštroumni čitatelj može zaključiti da će se pri deklaraciji varijable tipa `int16_t`:

```
int16_t sesnaestbitniCijeliBroj{34};
```

u dotičnoj implementaciji zapravo upotrijebiti tip `short int`, tj. kao da smo sami umjesto prethodne naredbe napisali:

```
short int sesnaestbitniCijeliBroj{34};
```

Budući da ne postoji jamstvo da će na nekom sustavu postojati cjelobrojni tip željene duljine (npr. na 10-bitnom računalu nije moguće adresirati prostor koji zauzima 8 bita), tip tražene duljine ne mora nužno biti definiran – sinonimi u prvom stupcu tablice su neobavezni. Stoga su definirani sinonimi „`least`”, kojima se pridružuje najbliži cjelobrojni tip iste ili veće duljine. Tako bi na 10-bitnom računalu `int_least8_t` odgovarao cjelobrojnoj varijabli duljine 10 bitova. Sinonimi „`fast`” u tablici 3.5 pridružuju cjelobrojne tipove koji omogućavaju najbržu obradu, a da su barem tražene veličine.

3.4.2. Aritmetički operatori

Da bismo objekte u programu mogli mijenjati, na njih treba primijeniti odgovarajuće operacije. Za ugrađene brojčane tipove podataka definirani su osnovni operatori, poput zbrajanja, oduzimanja, množenja i dijeljenja (vidi tablicu 3.6). Ti operatori se mogu podijeliti na *unarne*, koji djeluju samo na jedan objekt, te na *binarne* za koje su neophodna dva objekta. Osim unarnog plusa i unarnog minusa koji mijenjaju predznak broja, u jeziku C++ definirani su još i unarni operatori za uvećavanje (*inkrementiranje*) i umanjivanje vrijednosti (*dekrementiranje*) broja. Operator `++` uvećat će vrijednost varijable za 1, dok će operator `--` umanjiti vrijednost varijable za 1:

```
int i{0};
++i;                    // uvećа za 1
cout << i << endl;      // ispisuje 1
--i;                    // umanji za 1
cout << i << endl;      // ispisuje 0
```

Tablica 3.6. Aritmetički operatori

unarni operatori	`+x`	unarni plus
	`-x`	unarni minus
	`x++`	uvećaj nakon
	`++x`	uvećaj prije
	`x--`	umanji nakon
	`--x`	umanji prije
binarni operatori	`x + y`	zbrajanje
	`x - y`	oduzimanje
	`x * y`	množenje
	`x / y`	dijeljenje
	`x % y`	modulo

Pritom valja uočiti razliku između operatora kada je on napisan ispred varijable i operatora kada je on napisan iza nje. U prvom slučaju (*prefiks* operator), vrijednost varijable će se prvo uvećati ili umanjiti, a potom će biti dohvaćena njena vrijednost. U drugom slučaju (*postfiks* operator) je obrnuto: prvo se dohvati vrijednost varijable, a tek onda slijedi promjena. To najbolje dočarava sljedeći kôd:

```
int i{1};
cout << i << endl;              // ispiši za svaki slučaj
cout << (++i) << endl;          // prvo poveća, pa ispisuje 2
cout << i << endl;              // ispisuje opet, za svaki slučaj
cout << (i++) << endl;          // prvo ispisuje 2, a tek onda uveća
cout << i << endl;              // vidi, stvarno je uvećao: 3!
```

Na raspolaganju imamo pet binarnih aritmetičkih operatora: za zbrajanje, oduzimanje, množenje, dijeljenje i *modulo* operator:

```
double a{2.};
double b{3.};
cout << (a + b) << endl;        // ispisuje 5.
cout << (a - b) << endl;        // ispisuje -1.
cout << (a * b) << endl;        // ispisuje 6.
cout << (a / b) << endl;        // ispisuje 0.666667
```

Operator modulo može se primijeniti samo na dva cijela broja i kao rezultat vraća ostatak njihova cjelobrojna dijeljenja:

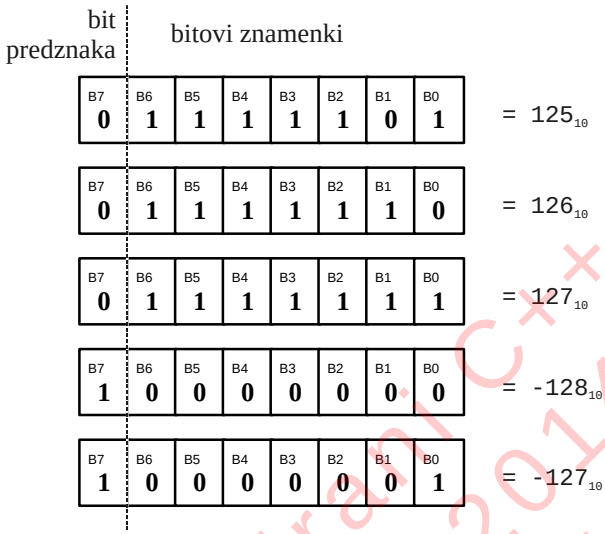
```
int i{6};
int j{4};
cout << (i % j) << endl;        // ispisuje 2
cout << (i % 3) << endl;        // ispisuje 0
```

On se vrlo često koristi za ispitivanje djeljivosti cijelih brojeva: ako su brojevi djeljivi, ostatak nakon dijeljenja će biti nula, kao što je vidljivo iz zadnjeg ispisa.

Za razliku od većine matematički orijentiranih jezika, jezik C++ nema ugrađeni operator za potenciranje, već programer mora sam napisati funkciju ili koristiti funkciju `pow()` iz standardne matematičke biblioteke deklarirane u datoteci zaglavlja `cmath`.

Uočimo dva suštinska problema vezana uz aritmetičke operatore. Prvi problem vezan je uz pojavu *aritmetičkog preljeva*, kada uslijed neke operacije rezultat nadmaši opseg koji dotični tip podatka pokriva. Drugi problem sadržan je u pitanju „kakvog će tipa biti rezultat binarne operacije s dva broja različitih tipova?”.

Razmotrimo pojavu aritmetičkog (ili *numeričkog*, *brojčanog*) preljeva (engl. *arithmetic overflow*, *numeric overflow*). Donje naredbe će prvo ispisati najveći broj tipa `short int`. Naredbom u trećem retku se vrijednost uvećava i postaje većom nego što ju taj tip može pohraniti. Prema standardu, rezultat nakon numeričkog preljeva je nedefiniran, što znači da je implementacijama prepušteno proizvoljno ponašanje (na nekim računalima preljev može baciti iznimku). Pa ipak, na većini prevoditelja, na zaslonu računala treća naredba će ispisati najveći negativni `short` (-32768)!



Slika 3.3. Prikaz prelјeva na cijelom broju s osam bitova

```
short i{SHRT_MAX};
cout << i << endl;      // ispisuje 32767
cout << (++i) << endl;  // ispisuje -32768
```

Uzrok tome je prelјev bitova do kojeg došlo zato što stvarni rezultat više ne stane u bitove predviđene za `int` varijablu. Podaci koji su se „prelili” ušli su u bit za predznak (zbog čega je rezultat negativan), a raspored preostalih bitova daje broj koji odgovara upravo onom što nam je računalo ispisalo. Slika 3.3 prikazuje takvu situaciju za cijeli broj koji je predstavljen s 8 bitova. Slično će se dogoditi oduzmete li od najvećeg negativnog broja neki broj. Ovo ponašanje se može predstaviti brojevnom kružnicom na kojoj zbrajanje odgovara kretanju po kružnici u smjeru kazaljke na satu, a oduzimanje odgovara kretanju u suprotnom smjeru (slika 3.4).

Ako postoji mogućnost pojave numeričkog prelјeva, tada deklarirajte varijablu s većim opsegom – u gornjem primjeru umjesto `int` uporabite `long int`.

Gornja situacija može se poistovjetiti s kupovinom automobila na sajmu rabljenih automobila. Naravno da auto star 10 godina nije prešao samo 5000 kilometara koliko piše na brojaču kilometara (*kilometer-cajgeru*), već je nakon 99999 kilometara brojač ponovno krenuo od 00000. Budući da brojač ima mjesta za 5 znamenki najviša znamenka se izgubila! Suštinska razlika je jedino u tome da je kod automobila do prelјeva došlo „hardverskom” intervencijom prethodnog vlasnika, dok u programu do prelјeva obično dolazi zbog nepažnje programera pri pisanju kôda.

Ponekad se pojava preljeva može izbjeći promjenom redoslijeda operacija. Pogledajmo to na primjeru računanja aritmetičke sredine dva broja:

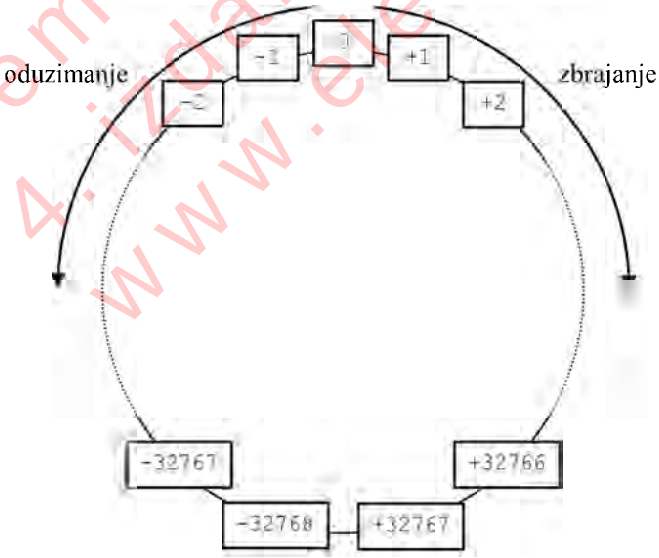
```
int prvi{INT_MAX - 1};
int drugi{prvi - 2};
int aritmetickaSredina = (prvi + drugi) / 2;
```

Zagrade oko zbroja su neophodne, jer dijeljenje ima viši prioritet od zbrajanja (o hijerarhiji operacija govorit ćemo u odjeljku 3.7) – da su zagrada izostavljene, s 2 bi se dijelio samo drugi pribrojnik. Oba su pribrojnika manja od najvećeg dozvoljenog broja tipa `int` pa od toga mora biti manja i njihova aritmetička sredina. Međutim, međurezultat (zbroj oba pribrojnika prije dijeljenja s 2) će premašiti tu granicu – doći će do njegovog preljeva zbog čega će i krajnji rezultat biti pogrešan. Prepišemo li zadnju naredbu na drugačiji način:

```
int aritmetickaSredina = prvi / 2 + drugi / 2;
```

opasnost od preljeva ćemo otkloniti. Međutim, ako su varijable `prvi` i `drugi` neparni brojevi, potonja naredba će dati pogrešan rezultat! Uzrok tome ćemo shvatiti kada razjasnimo cjelobrojno dijeljenje, na str. 68.

Pri pojavi preljeva kod operacija s decimalnim brojevima (tipa `float` i `double`), prema standardu IEEE 754 rezultat mora poprimiti posebne vrijednosti koje će upućivati da je došlo do pogreške. Na primjer, ispis umnoška najvećeg broja tipa `float` s brojem 10:



Slika 3.4. Prikaz preljeva na brojevnoj kružnici

```
float prevelikiFloat = numeric_limits<float>::max() * 10;
cout << prevelikiFloat << endl;
```

na zaslonu će (ovisno o platformi i prevoditelju) dati `1.#INF`, iz čega se može vidjeti da rezultat premašuje raspon vrijednosti koji `float` podržava. Slično je i dijeljenje nule s nulom nedefinirano:

```
float nula{0.};
cout << nula / nula << endl;
cout << 1 / nula << endl;
```

Zadatak: *Provjerite što će gornje nedozvoljene operacije ispisati u programu prevedenom prevoditeljem koji koristite.*

Napomenimo da neki prevoditelji prije aritmetičkih operacija podatke tipa `float` pretvaraju u tip `double` i operacije obavljaju s tako proširenim brojevima. Tek po završetku operacija, rezultat vraćaju u tip `float`. Zbog toga će teže doći do preljeva u međurezultatu, a samim tim je smanjena mogućnost pogreške u krajnjem rezultatu. Međutim, kako standard ne propisuje da se dva `float` broja prije međusobne aritmetike moraju pretvoriti u `double` (kao što će se vidjeti u tekstu koji slijedi), ne treba se previše oslanjati na ovakvo ponašanje, posebice ako želimo da kôd jednako radi i kada je preveden drugim prevoditeljima.

Drugo pitanje vezano na aritmetičke operatore koje se nameće jest kakvog će biti tipa rezultat binarne operacije na dva broja različitog tipa. Za ugrađene tipove točno su određena pravila *uobičajene aritmetičke pretvorbe*. Ako su oba operanda istog tipa, tada je i rezultat tog tipa, a ako su operandi različitih tipova, tada se oni prije operacije svode na *zajednički tip* (to je obično složeniji tip), prema sljedećim pravilima [ISO/IEC2011]:

1. Ako je jedan od operanada tipa `long double`, tada se i drugi operand pretvara u `long double`.
2. Inače, ako je jedan od operanada tipa `double`, tada se i drugi operand pretvara u `double`.
3. Inače, ako je jedan od operanada tipa `float`, tada se i drugi operand pretvara u `float`.
4. Inače se provodi *cjelobrojna promocija* (engl. *integral promotion*) oba operanda (ovo je bitno samo za operande tipa `bool`, `wchar_t` i pobrojenja, tako da ćemo o cjelobrojnoj promociji govoriti u odgovarajućim poglavljima o tim tipovima).
5. Ako su nakon toga oba operanda istog tipa, nema potrebe za daljnjim pretvorbama.
6. Inače, ako su oba operanda tipa `signed` ili ako su oba tipa `unsigned`, operand užeg tipa se pretvara u operand šireg tipa.
7. Inače, ako operand tipa `unsigned` pokriva širi ili isti raspon vrijednosti, tada se drugi operand pretvara u tip koji je uz `unsigned` tip.

8. Inače, ako tip uz `signed` operand može pokriti sve vrijednosti iz raspona podržanog tipom uz `unsigned` operand, operand tipa bez predznaka (`unsigned`) se pretvara u operand tipa s predznakom (`signed`).
9. Inače se oba operanda pretvaraju u tip bez predznaka koji odgovara tipu operanda s predznakom.

Na primjer, izvođenje kôda

```
int i{3};
double a{0.5};
cout << (a * i) << endl;
```

uzrokovat će ispis broja `1.5` na zaslonu, jer je od tipova `int` i `double` potonji složeniji tip. Cjelobrojnu varijablu `i` prevoditelj prije množenja pretvara u `double` (prema pravilu u točki 3), tako da se provodi množenje dva decimalna broja, pa je i rezultat tipa `double`. Da smo umnožak prije ispisa pridružili nekoj cjelobrojnoj varijabli

```
int c = a * i;
cout << c << endl;
```

dobili bismo ispis samo cjelobrojnog dijela rezultata, tj. broj `1`. Decimalni dio rezultata se gubi prilikom pridruživanja umnoška cjelobrojnoj varijabli `c`.

Problem pretvorbe brojevnih tipova najjače je izražen kod dijeljenja cijelih brojeva, što početnicima (ali i nepažljivim C++ „guruima") zna prouzročiti dosta glavobolje. Pogledajmo sljedeći jednostavni primjer:

```
int brojnik{1};
int nazivnik{4};
double kvocijent = brojnik / nazivnik;
cout << kvocijent << endl;
```

Suprotno svim pravilima zapadnoeuropske klasične matematike, na zaslonu će se kao rezultat ispisati `0.`, tj. najobičnija nula! Koristi li vaše računalo aboriđanski brojevni sustav, koji poznaje samo tri broja (*jedan*, *dva* i *puno*)? Iako smo rezultat dijeljenja pridružili `double` varijabli, pri izvođenju programa to pridruživanje slijedi tek nakon što je operacija dijeljenja dva cijela broja bila završena (ili, u duhu južnoslavenske narodne poezije, „Kasno `double` na Dijeljenje stiže!"). Budući da su varijable `brojnik` i `nazivnik` cjelobrojne, prevoditelj provodi cjelobrojno dijeljenje u kojem se zanemaruju decimalna mjesta. Stoga je rezultat cjelobrojni dio kvocijenta varijabli `brojnik` i `nazivnik` (0.25), a to je 0. Slična je situacija i kada dijelimo cjelobrojne konstante:

```
double diskutabilniKvocijent = 3 / 2;
```

Brojeve 3 i 2 prevoditelj će shvatiti kao cijele, jer ne sadrže decimalnu točku. Zato će primijeniti cjelobrojni operator `/`, pa će rezultat toga dijeljenja biti cijeli broj 1.

Da bi se izbjegle sve nedoumice ovakve vrste, dobra je programerska navada sve konstante koje trebaju biti decimalne (`float` i `double`) pisati s decimalnom točkom, čak i kada nemaju decimalnih mjesta.

Evo pravilnog načina da se provede željeno dijeljenje:

```
double tocniKvocijent = 3. / 2.;
```

Dovoljno bi bilo decimalnu točku staviti samo uz jedan od operanada – prema pravilima aritmetičke pretvorbe i drugi bi operand bio sveden na `double` tip. Iako je kôd ovako dulji, izaziva manje nedoumica i stoga svakom početniku preporučujemo da ne štedi na decimalnim točkama.

Neupućeni početnik postavit će logično pitanje zašto uopće postoji razlika između cjelobrojnog dijeljenja i dijeljenja decimalnih brojeva. Za programera bi bilo najjednostavnije kada bi se oba operanda bez obzira na tip, prije primjene operatora svela na `double` te na tako modificirane operande primijenio željeni operator. U tom slučaju programer uopće ne bi trebao paziti kojeg su tipa operandi – rezultat bi uvijek bio korektan. Nedostatak ovakvog pristupa jest njegova neefikasnost. Zamislimo da treba zbrojiti dva broja tipa `int`. U gornjem „programer-ne-treba-paziti” pristupu, izvedbeni kôd dobiven prevođenjem trebao bi prvi cjelobrojni operand pretvoriti u `double`. Budući da se različiti tipovi podataka pohranjuju u memoriju računala na različite načine, ta pretvorba nije trivijalna, već iziskuje određeni broj strojnih instrukcija. Istu pretvorbu treba ponoviti za drugi operand. Uočavamo dvije operacije pretvorbe tipa koje kod izravnog računanja s cijelim brojevima ne postoje! Za mali broj operacija korisnik zasigurno ne bi osjetio razliku u izvođenju programa s izravno primijenjenim operatorima i operatorima *à la* „programer-ne-treba-paziti”. Međutim, u složenim programima s nekoliko tisuća ili milijuna operacija, ta razlika može biti zamjetna, a često i kritična. U krajnjoj liniji, shvatimo da prevoditelj poput vjernog psa nastoji što brže ispuniti gospodarevu zapovijed, ne pazeći da li će pritom protrčati kroz upravo uređen susjedin cvjetnjak ili zagaziti u svježe betoniran pločnik.

Svakako valja naglasiti da će cjelobrojno dijeljenje s nulom baciti iznimku (§16) i ako u kôdu ne obradimo tu iznimku, program će se „srušiti”. Stoga:

Prije naredbe u kojoj može doći do cjelobrojnog dijeljenja s nulom poželjno je umetnuti ispitivanje i onemogućiti takvu operaciju.

Zadatak: *Što će ispisati sljedeće naredbe:*

```
int a{10};
double b{10};

cout << a / 3 << endl;
cout << b / 3 << endl;
```

Zadatak: *Da li će postojati razlika pri ispisu u donjim naredbama (varijable a i b deklarirane su u prethodnom zadatku):*

```
double c = a / 3;
cout << c * b << endl;
c = a / 3.;
cout << c * b << endl;
c = b * a;
cout << c / 3 << endl;
```

Zadatak: *Što će ispisati donje naredbe? Obrazložite zašto će donje naredbe ispisati različite rezultate uvijek kada su i a i b neparni brojevi:*

```
int a{3};
int b{5};
cout << (a + b) / 2 << endl;
cout << a / 2 + b / 2 << endl;
```

3.4.3. Operator dodjele tipa

Što učiniti želimo li podijeliti dvije cjelobrojne varijable, a da pritom ne izgubimo decimalna mjesta? Dodavanje decimalne točke iza imena varijable nema smisla, jer će prevoditelj javiti pogrešku. Za eksplicitnu promjenu tipa varijable valja primijeniti operator `static_cast` kojim se nekom izrazu eksplicitno može *dodjeliti tip* (engl. *type cast*, kraće samo *cast*):

```
int brojnik{1};
int nazivnik{3};
double tocniKvocijent = static_cast<double>(brojnik) /
                        static_cast<double>(nazivnik);
```

Općeniti oblik primjene operatora `static_cast` jest:

```
static_cast< Tip >( izraz )
```

gdje se između znakova < i > („manje” i „veće”) navodi tip u koji želimo pretvoriti *`izraz`* unutar zagrada koje slijede. U gornjem kôdu se tako vrijednosti varijabli `brojnik` i `nazivnik` pretvaraju u tip `double` prije nego što se one međusobno podijele, pa je krajnji rezultat ispravan. Naravno, bilo bi dovoljno operator dodjele tipa primijeniti samo na jedan operand – prema pravilima aritmetičke pretvorbe i drugi bi operand bio sveden na `double` tip. Da ne bi bilo zabune, same varijable `brojnik` i `nazivnik` i nadalje ostaju tipa `int`, tako da će njihovo naknadno dijeljenje

```
double opetKriviKvocijent = brojnik / nazivnik;
```

opet kao rezultat dati kvocijent cjelobrojnog dijeljenja.

Prečesto korištenje operatora dodjele tipa može biti pokazatelj lošeg dizajna („Zašto su `brojnik` i `nazivnik` deklarirani kao `int` ako želimo s njima provesti `double` dijeljenje?"), tim više što on predstavlja mogući izvor programskih pogrešaka – njegovom uporabom programer onemogućava striktnu provjeru tipa koju provodi prevoditelj. Međutim, postoje slučajevi kada ga jednostavno ne možemo izbjeći [Lipmann98]:

- kada pokazivač tipa `void*` („neodređenog" tipa) moramo usmjeriti na pokazivač zadanog tipa (o pokazivačima će biti riječi u 6. poglavlju);
- kada želimo zaobići uobičajene pretvorbe (spomenute u prethodnom odjeljku);
- kada nije jednoznačno određeno u koji će se tip određeni objekt pretvoriti, jer postoji mogućnost više pretvorbi; ovo je izraženo kod korisnički definiranih tipova koji su izvedeni iz više različitih tipova – taj problem bit će obrađen u 10. poglavlju posvećenom nasljeđivanju.

Također, operator dodjele tipa neizbježan je kada se koristi naslijeđeni kôd kojeg je nezgodno ili nemoguće mijenjati; ubacivanjem operatora dodjele tipa isključuju se upozorenja o neistovjetnosti tipova podataka koja bi inače prevoditelj izbacivao.

Operator dodjele tipa može se koristiti i u obrnutom slučaju, npr. kada želimo iz decimalnog broja izlučiti samo cjelobrojne znamenke:

```
double pi{3.1415926}
cout << "Cijeli dio broja pi: " << static_cast<int>(pi) << endl;
```

Prije uvođenja operatora `static_cast`, dodjela tipa u jeziku C++ se obavljala na identičan način kao i u jeziku C navođenjem željenog tipa u okruglim zagradama `()` ispred izraza:

```
double tocniKvocijent = (double)brojnik / (double)nazivnik;
```

Jezik C++ dozvoljava i „funkcijski" oblik dodjele tipa u kojem se tip navodi ispred zagrade, a ime varijable u zagradi:

```
double tocniKvocijent2 = double(brojnik) / double(nazivnik);
```

Iako su ova dva oblika dodjele tipa u potpunosti podržana standardom, njihova upotreba se ne preporučuje. Naime, osim statičke dodjele koju provodi prevoditelj tijekom generiranja izvedbenog kôda, postoje i dodjele tipa koje se obavljaju tijekom izvođenja programa. Operator `()` obuhvaća obje vrste dodjela, bez njihovog jasnog razlučivanja. To znači da je prevoditelju prepušteno da sam odluči koji oblik dodjele će se obaviti i programer ne može u potpunosti biti siguran kakva će se dodjela doista izvesti. O operatorima dodjele tipa bit će još detaljno govora u 17 poglavlju koje se bavi identifikacijom tipa tijekom izvođenja.

Zadatak: *Odredite koje će od navedenih naredbi za ispis:*

```
int a{100000};
int b{200000};
long c = a * b;
```

```
cout << c << endl;
cout << (a * b) << endl;
cout << (static_cast<double>(a) * b) << endl;
cout << static_cast<long>(a * b) << endl;
cout << (a * static_cast<long>(b)) << endl;
```

dati ispravan umnožak, tj. 20 000 000 000.

Zadatak: *Odredite što će se ispisati na zaslonu po izvođenju sljedećeg kôda:*

```
double a{2.71};
double b = static_cast<int>(a);
cout << b << endl;
```

3.4.4. Dodjeljivanje tipa brojčanim konstantama

Kada se u kôdu pojavljuju brojčane konstante, prevoditelj ih pohranjuje u formatu nekog od osnovnih tipova. Tako s brojevima koji sadrže decimalnu točku ili slovo `e`, odnosno `E`, prevoditelj barata kao s podacima tipa `double`, dok sve ostale brojeve tretira kao `int`. Operatore dodjele tipa moguće primijeniti i na konstante, na primjer:

```
cout << static_cast<long>(10) << endl;          // long int
cout << static_cast<unsigned>(60000) << endl;   // unsigned int
```

No, za jednostavnije specificiranje konstanti koriste se *sufiksi*, posebni znakovi kojima se eksplicitno određuje tip brojčane konstante (vidi tablicu 3.7). Tako cjelobrojnu konstantu sa sufiksom `l`, odnosno `L` prevoditelj interpretira kao `long`, a decimalnu konstantu s istim sufiksom kao `long double`:

```
long hrvatskiDugovi{345763045547557195252 5L};
long double a = 1.602e-4583L / 645672L;    // (long double) / long
```

dok će sufiksi `u`, odnosno `U` cjelobrojne konstante pretvoriti u `unsigned int`:

```
cout << 65000U << endl;     // unsigned int
```

Tablica 3.7. Djelovanje sufiksa na brojčane konstante

broj ispred sufiksa	sufiks	rezultirajući tip
cijeli		`int`
	`L`, `l`	`long int`
	`LL`, `ll`	`long long int`
	`U`, `u`	`unsigned int`
decimalni		`double`
	`F`, `f`	`float`
	`L`, `l`	`long double`

Sufiks `U` (`u`) može se kombinirati sa sufiksima `L` (`l`) i `LL` (`ll`) u bilo kojem redoslijedu (`LU`, `LLU`, `UL`, `ULL`, `Ul`, `uL`, `ul`, `lu`...) za zadavanje cjelobrojnih konstanti tipa `unsigned long int`, odnosno `unsigned long long int`.

Sufiks `f`, odnosno `F` će konstantu s decimalnom točkom interpretirati kao tip `float`:

```
float b{1.234F};
```

Velika i mala slova sufiksa su potpuno ravnopravna. U svakom slučaju je preglednije koristiti veliko slovo `L`, da bi se izbjegla zabuna zbog sličnosti slova `l` i broja 1.

Sufiksi se rijetko koriste, budući da u većini slučajeva prevoditelj obavlja sam neophodne pretvorbe, prema već spomenutim pravilima. Izuzetak je pretvorba `double` u `float` koju provodi sufiks `F`. Zaboravimo li staviti sufiks:

```
float c{1.34};          // pogreška: „širi” tip pridružujemo „užem”
float d = 1.34;         // upozorenje: „širi” tip pridružujemo „užem”
```

prevoditelj će u prvoj naredbi prijaviti pogrešku, a u drugoj upozorenje da se tip `double` pokušava pridružiti tipu `float`.

3.4.5. Simboličke konstante

U programima se redovito koriste simboličke veličine čija se vrijednost tijekom izvođenja ne želi mijenjati. To mogu biti fizičke ili matematičke konstante, ali i parametri poput maksimalnog broja prozora ili maksimalne duljine znakovnog niza, koji se namještaju prije prevođenja kôda i ulaze u izvedbeni kôd kao konstante.

Zamislimo da za neki zadani polumjer želimo izračunati i ispisati opseg kružnice, površinu kruga, oplošje i volumen kugle. Pri računanju sve četiri veličine treba nam Ludolfov broj π=3,14159...:

```
double opseg = 2. * r * 3.14159265359;
double povrsina = r * r * 3.14159265359;
double oplosje = 4. * r * r * 3.14159265359;
double volumen = 4. / 3. * r * r * r * 3.14159265359;
```

Naravno da bi jednostavnije i pouzdanije bilo definirati zasebnu varijablu koja će sadržavati broj π:

```
double pi{3.14159265359};
double opseg = 2. * r * pi;
double povrsina = r * r * pi;
double oplosje = 4. * r * r * pi;
double volumen = 4. / 3. * r * r * r * pi;
```

Manja je vjerojatnost da ćemo pogriješiti prilikom utipkavanja samo jednog broja, a osim toga, promijeni li se, kojim slučajem jednog dana, vrijednost broja π, bit će ga

lakše ispraviti kada je definiran samo na jednom mjestu, nego raditi pretraživanja i zamjene brojeva po cijelom izvornom kôdu.

Korištenje simboličkih konstanti doprinosi razumljivosti kôda, jer se umjesto „čarobnih brojeva" u kôdu pojavljuju opisna imena. To je posebno izraženo ako se koriste vrijednosti koje nisu općepoznate konstante poput broja π:

```
double foot = 12. * centimetaraPoInchu;
int dan = 24 * brojSekundiUSatu;
```

Pretvaranjem konstantnih veličina u varijable izlažemo ih pogibelji od nenamjerne promjene vrijednosti. Nakon izvjesnog vremena jednostavno zaboravite da dotična varijabla predstavlja konstantu, te negdje u kôdu dodate naredbu:

```
pi = 2 * pi;
```

kojom ste promijenili vrijednost varijable `pi`. Prevoditelj vas neće upozoriti („Opet taj prokleti kompjutor!") i dobit ćete da zemaljska kugla ima dva puta veći volumen nego što ga je imala prošli puta kada ste koristili isti program, ali bez inkriminirane naredbe. Koristite li program za određivanje putanje svemirskog broda, ovakva pogreška sigurno će rezultirati odašiljanjem broda u bespuća svemirske zbiljnosti.

Da bismo izbjegli ovakve peripetije, na raspolaganju nam je kvalifikator `const` pomoću kojeg možemo deklarirati simboličke konstante i koji prevoditelju daje na znanje da jednom inicijalizirana varijabla ostaje nepromjenjiva. Na svaki pokušaj promjene vrijednosti takve varijable, prevoditelj će javiti pogrešku:

```
const double pi{3.14159265359};
pi = 2 * pi;                        // sada je to pogreška!
```

Drugi često korišteni pristup zasniva se na pretprocesorskoj naredbi `#define`:

```
#define PI 3.14159265359
```

Ona daje uputu prevoditelju da, prije nego što započne prevođenje izvornog kôda, sve pojave prvog niza (`PI`) zamijeni drugim nizom znakova (`3.14159265359`). Stoga će prevoditelj naredbe:

```
double volumen = 4. / 3. * r * r * r * PI;
PI = 2 * PI;                                    // pogreška!
```

interpretirati kao da se u njima umjesto simbola `PI` nalazi odgovarajući broj. U drugoj naredbi će javiti pogrešku, jer se taj broj našao s lijeve strane operatora pridruživanja. Valja napomenuti da ove zamjene prevoditelj (točnije, njegov pretprocesor §23.3) ne radi u tekstu izvornog kôda, već samo u privremenom kôdu iz kojeg prevoditelj potom generira izvedbeni kôd. Tekst izvornog kôda kojeg je programer napisao ostaje nepromijenjen.

Na prvi pogled nema razlike između ova dva pristupa – oba pristupa će osigurati prijavu pogreške pri pokušaju promjene konstante. Razlika postaje očita tek kada poku-

šate program ispraviti koristeći *debugger*. Ako ste konstantu definirali pretprocesorskom naredbom, njeno ime neće postojati u simboličkoj tablici koju prevoditelj generira, jer je prije prevođenja svaka pojava imena nadomještena brojem. Ne možete čak ni provjeriti da li ste možda pogriješili prilikom poziva imena.

Konstante definirane pomoću deklaracije `const` dohvatljive su i iz programa za simboličko lociranje pogrešaka.

Napomenimo da vrijednost simboličke konstante mora biti inicijalizirana prilikom deklaracije, budući da prevoditelj sve pojave imena konstante koje slijede u kôdu mora nadomjestiti njenom vrijednošću. U protivnom ćemo dobiti poruku o pogreški:

```
const double mojaMalaKonstanta;          // pogreška!
```

Prilikom inicijalizacije vrijednosti nismo ograničeni na brojeve – možemo pridružiti i vrijednost neke prethodno definirane varijable. Vrijednost koju pridružujemo konstanti ne mora biti čak ni zapisana u kôdu:

```
double a;
cin >> a;
const double DvijeTrecine = a;
```

3.4.6. Konstante definirane konstantnim izrazima

U prethodnom odjeljku smo vidjeli kako kvalifikator `const` onemogućava naknadne promjene objekta. U slučaju da ga inicijaliziramo nekim promjenjivim objektom, njegova inicijalizacija će biti provedena tijekom izvođenja programa, kao što je bio slučaj u zadnjem primjeru iz prethodnog odjeljka, kada smo vrijednost dobili unosom preko konzole, ili u sljedećem primjeru:

```
const double korijenIzDva = sqrt(2.);
```

gdje koristimo standardnu funkciju `sqrt()` za izračunavanje kvadratnog korijena, deklariranu u zaglavlju `cmath`.

Standardom C++11, osim simboličkih konstanti, jezik C++ uveo je deklaraciju konstantnih izraza (engl. *constant expression*) koji se uvijek izračunavaju tijekom prevođenja kôda. Konstantni izrazi deklariraju se pomoću ključne riječi `constexpr`:

```
constexpr double masaElektrona = 9.10938291e-31;
```

Ako takav izraz ne može izračunati tijekom prevođenja, prevoditelj će javiti pogrešku:

```
const double pi = 3.14159265358979323846264338327795;
constexpr double dvaPi = 2 * pi; // OK: pi je poznat tijekom prevođenja
constexpr double korijenIzDva = sqrt(2.);   // pogreška!
```

Iako je `pi` definiran kao simbolička konstanta, njegova vrijednost je poznata u trenutku prevođenja jer je inicijalizirana literalom – i on je u biti konstantan izraz unatoč tome da nije deklariran s ključnom riječi `constexpr`. Zbog toga će prevoditelj moći izračunati vrijednost i inicijalizirati `dvaPi`. S druge strane, `korijenIzDva` je simbolička konstanta čija vrijednost nije poznata u trenutku prevođenja kôda, budući da iziskuje poziv funkcije `sqrt()`, pa pomoću nje nije moguće inicijalizirati konstantni izraz `dvaKorijenaIzDva`. Dakle, konstantni izrazi mogu se inicijalizirati samo podacima koji su sami konstanti izrazi.

Nameće se pitanje zašto je uz već postojeću ključnu riječ `const` dodana nova riječ `constexpr`? Riječ `constexpr` jamči da će vrijednost biti inicijalizirana tijekom prevođenja. Ako prevoditelj to ne može napraviti, javit će pogrešku. S druge strane, ključna riječ `const` samo garantira da se jednom inicijalizirana vrijednost neće moći promijeniti. Ona se inicijalizira tijekom izvođenja, što može biti problematično u slučajevima kada je vrijednost definirana u nekoj drugoj datoteci izvornog kôda.

Osim jednostavnih izraza, `constexpr` omogućava i definiciju funkcija čija se vrijednost izračunava tijekom prevođenja (§8.12).

3.4.7. Kvalifikator `volatile`

U prethodnim odsječcima smo pokazali kako deklarirati konstantne objekte, da bismo spriječili njihove naknadne promjene. Svi ostali objekti su promjenjivi (engl. *volatile* - nepostojan, lako hlapljiv). Promjenjivost objekta se može naglasiti tako da se ispred tipa umetne ključna riječ `volatile`:

```
volatile Tip ime_promjenjivice;
```

Time se prevoditelju daje na znanje da se vrijednost varijable može promijeniti njemu nedokučivim načinima, te da zbog toga mora isključiti sve optimizacije kôda prilikom pristupa.

Valja razjasniti koji su to načini promjene koji su izvan prevoditeljevog znanja. Na primjer, u sistemskom programu neki vanjski uređaj priključen na računalo može mijenjati sadržaj memorijske adrese unutar obrade *prekida* (engl. *interrupt*) – time prekidna rutina može signalizirati programu da je određeni uvjet zadovoljen. U takvom slučaju prevoditelj ne smije optimizirati pristup navedenoj varijabli. Na primjer, u sljedećem primjeru prevoditelj analizom petlje može zaključiti da se varijabla `izadjiVan` ne mijenja unutar petlje pa će optimizirati izvođenje tako potpuno izbaci naredbu `while` (§4.3.2) kojom se provjerava vrijednost varijable `izadjiVan`:

```
int izadjiVan = 0;

while (izadjiVan != 1)
{
    // čekaj dok se vrijednost izadjiVan ne postavi na 1
}
```

Prekidna rutina koja će eventualno postaviti vrijednost `izadjiVan` na 1 zbog toga neće okončati petlju. Da bismo to spriječili, `izadjiVan` moramo deklarirati kao `volatile`:

```
volatile int izadjiVan;
```

Moguće je definirati i nepostojanu konstantu `const volatile`:

```
const volatile int zaTebeSamKonstantanZaDrugePromjenjiv = 8;
```

Ovako deklarirani objekt bit će nepromjenjiv iz kôda i prevoditelj će prijaviti pogrešku na svaki pokušaj pridruživanja nove vrijednosti, ali će isto tako prevoditelj biti svjestan da se objekt može promijeniti izvana te će izbjeći možebitne optimizacije kôda.

3.4.8. Logički tipovi i operatori

Logički podaci mogu poprimiti samo dvije vrijednosti, na primjer: da/ne, istina/laž, dan/noć. Jezik C++ za prikaz podataka logičkog tipa ima ugrađen tip `bool`, koji može poprimiti vrijednosti `true` (engl. *true* – *točno*) ili `false` (engl. *false* – *pogrešno*)[1]:

```
bool jeLiDanasNedjelja = true;
bool jaZnamCPlusPlus = false;
```

Pri ispisu logičkih tipova, te pri njihovom korištenju u aritmetičkim izrazima, logički tipovi se pravilima cjelobrojne promocije (vidi prethodno poglavlje) pretvaraju u `int`: `true` se pretvara u cjelobrojni 1, a `false` u 0. Isto tako, logičkim varijablama se mogu pridruživati aritmetički tipovi: u tom slučaju se vrijednosti različite od nule pretvaraju u `true`, a nula se pretvara u `false`.

Gornja svojstva tipa `bool` omogućavaju da se za predstavljanje logičkih podataka koriste i cijeli brojevi. Broj 0 u tom slučaju odgovara logičkoj neistini, a bilo koji broj različit od nule logičkoj istini. Iako je za logičku istinu na raspolaganju vrlo široki raspon cijelih brojeva (zlobnici bi rekli da ima više istina), ona se ipak najčešće zastupa brojem 1. Na ovakvo predstavljanje logičkih podataka se često može naići u postojećem kôdu budući da je tip `bool` dosta kasno uveden u prvu verziju standarda jezika C++.

Tablica 3.8. Logički operatori

`!x`	logička negacija
`x && y`	logički i
`x || y`	logički ili

[1] Naziv `bool` dolazi od prezimena engleskog matematičara Georgea Boolea (1815–1864), utemeljitelja logičke algebre.

Tablica 3.9. Stanja za logički *i*

a		*b*		*a* `.i.` *b*	
točno	(1)	točno	(1)	točno	(1)
točno	(1)	pogrešno	(0)	pogrešno	(0)
pogrešno	(0)	točno	(1)	pogrešno	(0)
pogrešno	(0)	pogrešno	(0)	pogrešno	(0)

Za logičke podatke definirana su svega tri operatora: `!` (*logička negacija*), `&&` (*logički i*), te `||` (*logički ili*)[1] (tablica 3.8). Logička negacija je unarni operator koji mijenja logičku vrijednost varijable: istinu pretvara u neistinu i obrnuto. Logički *i* daje kao rezultat istinu samo ako su oba operanda istinita; radi boljeg razumijevanja u tablici 3.9 dani su rezultati logičkog *i* za sve moguće vrijednosti oba operanda. Logički *ili* daje istinu ako je bilo koji od operanada istinit (vidi tablicu 3.10).

Tablica 3.10. Stanja za logički *ili*

a		*b*		*a* `.ili.` *b*	
točno	(1)	točno	(1)	točno	(1)
točno	(1)	pogrešno	(0)	točno	(1)
pogrešno	(0)	točno	(1)	točno	(1)
pogrešno	(0)	pogrešno	(0)	pogrešno	(0)

Razmotrimo primjenu logičkih operatora na sljedećem primjeru u kojem koristimo obično pobrojenje zato što se ono lako pretvara u logički tip i ispisuje:

```
enum Logika {Neistina, Istina, DrugaIstina = 124};
Logika a{Neistina};
Logika b{Istina};
Logika c{DrugaIstina};

cout << "a = " << a << ", b = " << b << ", c = " << c << endl;
cout << "Suprotno od a = " << !a << endl;           // ispisuje 1
cout << "Suprotno od b = " << !b << endl;           // ispisuje 0
cout << "Suprotno od c = " << !c << endl;           // ispisuje 0
cout << "a .i. b = " << (a && b) << endl;           // ispisuje 0
cout << "a .ili. c = " << (a || c) << endl;         // ispisuje 1
```

Unatoč tome da smo varijabli `c` pridružili vrijednost `DRUGAISTINA = 124`, njenom logičkom negacijom dobiva se 0, tj. logička neistina. Operacija `a`-*logički i*-`b` daje neistinu (broj 0), jer operand `a` ima vrijednost *pogrešno*, dok `a`-*logički ili*-`c` daje istinu, tj. 1, jer operand `c` ima logičku vrijednost *točno*.

[1] | je znak koji je na tipkovnici označen prekinutom vertikalnom crtom ¦.

Umjesto simboličkih operatora navedenih u tablici 3.8, neki programeri daju prednost duljim, ali razumljivijim oznakama navedenima u tablici 3.14. Uz korištenje tih oznaka, naredbe iz prethodnog primjera napisali bismo kao:

```
#include <ciso646>
// ...
cout << "Suprotno od c = " << not c << endl;        // ispisuje 0
cout << "a .i. b = " << (a and b) << endl;          // ispisuje 0
cout << "a .ili. c = " << (a or c) << endl;         // ispisuje 1
```

Budući da su te alternativne oznake definirane u zaglavlju `ciso646`, ne smijemo ga zaboraviti uključiti.

Prilikom korištenja složenih izraza s logičkim operatorima vrijedno je znati da se izraz s desne strane izračunava samo ako on može utjecati na krajnji rezultat. Na primjer, ako izraz s lijeve strane operatora *logički i* daje vrijednost *pogrešno*, izraz s desne strane se neće niti izračunavati jer će ukupni rezultat biti *pogrešno* bez obzira na rezultat izraza s desne strane. Slično, ako izraz s lijeve strane operatora *logički ili* daje vrijednost *točno*, nema potrebe izračunavati desnu stranu. Preskakanjem operacija koje su nebitne za konačni rezultat ubrzava se izvođenje logičkih izraza i takvo ponašanje se naziva *kratkospojna procjena* (engl. *short-circuit evaluation*).

Ovakvo ponašanje se ponekad koristi da se izbjegnu potencijalno opasne operacije. Razjasnimo to na primjeru u kojem provjeravamo cjelobrojnu djeljivost brojeva:

```
int a{4};
int b{0};
bool aJeDjeljivSb = (b != 0) && (a % b == 0);
```

Djeljivost provjeravamo pomoću operatora modulo, no pri njegovom korištenju treba paziti da djelitelj (varijabla `b`) ne bude jednak nuli, jer će u tom slučaju biti bačena iznimka i naš program će se srušiti. Zato smo u izraz koji provjerava djeljivost umetnuli prethodnu provjeru je li djelitelj različit od nule. Prilikom izvođenja, prvo se izračunava izraz s lijeve strane operatora `&&`. Ako on daje vrijednost *pogrešno* (tj. ako je varijabla `b` jednaka nuli), desna strana se neće niti izračunavati jer ne može promijeniti konačni rezultat. Time je izbjegnuto dijeljenje s nulom, a konačni rezultat će biti ispravan.

Zadatak: *Napišite sličnu naredbu za izračunavanje varijable `aNijeDjeljivSb` u kojoj koristite logički ili.*

Logički operatori i operacije s njima uglavnom se koriste u naredbama za grananje toka programa (§4), pa ćemo ih tamo upoznati još detaljnije.

3.4.9. Poredbeni operatori

Osim aritmetičkih operacija, jezik C++ omogućava i usporedbe dva broja (vidi tablicu 3.11). Kao rezultat usporedbe dobiva se tip `bool`: ako je uvjet usporedbe zadovoljen, rezultat je `true`, a ako nije rezultat je `false`. Tako će se izvođenjem kôda:

Tablica 3.11. Poredbeni operatori

`x < y`	manje od
`x <= y`	manje ili jednako
`x > y`	veće od
`x >= y`	veće ili jednako
`x == y`	jednako
`x != y`	različito

```
cout << boolalpha;              // da nam ispiše 'true' ili 'false'
cout << (5 > 4) << endl;        // je li 5 veće od 4?
cout << (5 >= 4) << endl;       // je li 5 veće ili jednako 4?
cout << (5 < 4) << endl;        // je li 5 manje od 4?
cout << (5 <= 4) << endl;       // je li 5 manje ili jednako 4?
cout << (5 == 4) << endl;       // je li 5 jednako 4?
cout << (5 != 4) << endl;       // je li 5 različito od 4?
```

na zaslonu ispisati redom: `true`, `true`, `false`, `false`, `false` i `true`. Prije ispisa, u izlazni tok smo dodali manipulator (§21.6.6) `boolalpha`, koji je osigurao da se logičke vrijednosti ispišu kao odgovarajući tekst („`true`”, odnosno „`false`”). Da smo prvu naredbu izostavili, ispisali bi se cjelobrojni ekvivalenti: 1, 1, 0, 0, 0, 1.

Poredbeni operatori (ponekad nazvani i *relacijski operatori* od engl. *relational operators*) se koriste prvenstveno u naredbama za grananje toka programa, gdje se, ovisno o tome je li neki uvjet zadovoljen, izvođenje programa nastavlja u različitim smjerovima. Stoga ćemo poredbene operatore podrobnije upoznati kod naredbi za grananje, u poglavlju 4.

Uočimo suštinsku razliku između jednostrukog znaka jednakosti (=) koji je simbol za pridruživanje, te dvostrukog znaka jednakosti (==) koji je operator za usporedbu!

Što će se dogoditi ako umjesto operatora pridruživanja `=`, pogreškom u nekom izrazu napišemo operator usporedbe `==`? Na primjer:

```
int a = 3;
a == 5;
```

U drugoj naredbi će se umjesto promjene vrijednosti varijable `a`, ona usporediti s brojem 5. Rezultat te usporedbe je `false`, odnosno 0, ali to ionako nema nikakvog značaja, jer se rezultat usporedbe u ovom slučaju ne pridružuje niti jednoj varijabli – naredba nema nikakvog efekta. A što je najgore, prevoditelj neće prijaviti pogrešku, već možda samo upozorenje! Kod složenijeg izraza poput

```
a = 1.23 * (b == c + 5);
```

to upozorenje će izostati, jer ovdje poredbeni operator može imati smisla. Za početnike jedno od nezgodnih svojstava jezika C++ jest da trpi i besmislene izraze, poput:

```
i + 5;
```

ili:

```
i == 5 == 6;
```

Bolji prevoditelji će u gornjim primjerima prilikom prevođenja dojaviti upozorenje o tome da kôd nema nikakvog efekta.

3.4.10. Znakovi

Znakovne konstante tipa `char` pišu se uglavnom kao samo jedan znak unutar jednostrukih znakova navodnika:

```
char slovoA{'a'};
cout << 'b' << endl;
```

Za znakove koji se ne mogu prikazati na zaslonu koriste se *posebne sekvence* (engl. *escape sequence*) koje počinju lijevom kosom crtom (engl. *backslash*) (tablica 3.12). Na primjer, kôd:

```
cout << '\n';       // znak za novi redak
```

uzrokovat će pomak značke na početak sljedećeg retka, tj. ekvivalentan je ispisu konstante `endl`.

Znakovne konstante najčešće se koriste u *znakovnim nizovima* (engl. *strings*) za ispis tekstova, te ćemo ih tamo detaljnije upoznati. Za sada samo spomenimo da se znakovni nizovi sastoje od nekoliko znakova unutar dvostrukih navodnika. Zanimljivo je da se `char` konstante i varijable mogu uspoređivati, poput brojeva:

```
cout << ('a' < 'b') << endl;          // ispisuje 1
cout << ('a' < 'B') << endl;          // ispisuje 0
cout << ('A' > 'a') << endl;          // ispisuje 0
cout << ('\'' != '\"') << endl;       // usporedba jednostrukog
                                      // i dvostrukog navodnika
```

pri čemu se u biti uspoređuju brojčani kôdovi kojima su ti znakovi predstavljeni u aktivnom kodiranju znakova. Za slova engleske abecede, znamenke i standardne simbole interpunkcije su kodovi identični u gotovo svim kodiranjima i odgovaraju ASCII kodovima (kratica od *American Standard Code for Information Interchange*) – znamenkama, slovima engleske abecede i uobičajenim simbolima pridruženi su brojevi od 32 do 126, dok specijalni znakovi imaju kôdove od 0 do 31 uključivo. U prethodnom primjeru,

Tablica 3.12. Posebni znakovi

`\n`	novi redak
`\t`	horizontalni tabulator
`\v`	vertikalni tabulator
`\b`	pomak za mjesto unazad (*backspace*)
`\r`	povrat na početak retka (*carriage return*)
`\f`	nova stranica (*form feed*)
`\a`	zvučni signal (*alert*)
`\\`	kosa crta ulijevo (*backslash*)
`\?`	upitnik
`\'`	jednostruki navodnik
`\"`	dvostruki navodnik
`\0`	završetak znakovnog niza
`\`*ddd*	znak čiji je kôd zadan oktalno s 1, 2 ili 3 znamenke
`\x`*ddd*	znak čiji je kôd zadan heksadekadski

prva naredba ispisuje 1, jer je ASCII kôd malog slova a (97) manji od kôda za malo slovo `b` (98). Druga naredba ispisuje 0, jer je ASCII kôd slova B jednak 66 (nije važno što je B po abecedi iza a!). Treća naredba ispisuje 0, jer za veliko slovo `A` kôd iznosi 65. Zbog tog razloga četvrta naredba ispisuje 1. ASCII kôdovi za jednostruki navodnik i dvostruki navodnik su 39 odnosno 34, pa zaključite sami što će četvrta naredba ispisati.

Znakovi se mogu uspoređivati sa cijelim brojevima, pri čemu se oni pretvaraju u cjelobrojni kôdni ekvivalent. Naredbom

```
cout << (32 == ' ') << endl;    // usporedba broja 32 i praznine
```

ispisat će se broj 1, jer je ASCII kôd praznine upravo 32.

Štoviše, tip `char` spada u cjelobrojne tipove pa se na znakove mogu primjenjivati i svi aritmetički operatori. Budući da se pritom znakovi cjelobrojnom promocijom pretvaraju u tip `int`, za njih vrijede ista pravila kao i za operacije sa cijelim brojevima. Ilustrirajmo to sljedećim primjerom:

```
char a = 'h';                   // ASCII kôd 104
cout << (a + 1) << endl;        // ispisuje 105
char b = a + 1;
cout << b << endl;              // ispisuje 'i'
```

Kod prvog ispisa znakovna varijabla `a` (koja ima vrijednost slova `h`) se, zbog operacije sa cijelim brojem 1, pretvara se u ASCII kôd za slovo `h`, tj. u broj 104, dodaje joj se 1, te ispisuje broj 105. Druga naredba za ispis daje slovo `i`, jer se broj 105 pridružuje znakovnoj varijabli `b`, te se 105 ispisuje kao ASCII znak, a ne kao cijeli broj.

Zadatak: *Razmislite i provjerite što će se ispisati izvođenjem sljedećeg kôda:*

```
char a = 'a';
char b = a;
int asciiKod = a;
cout << a << endl;
cout << b << endl;
cout << 'b' << endl;
cout << asciiKod << endl;
```

Tip `char` je u pravilu dovoljan ako se u programu koriste samo znakovi jednog jezika. Međutim, želimo li omogućiti nesputani ispis i upis za bilo koji jezik te ako želimo da se program jednako ponaša na bilo kojem računalu u bilo kojem dijelu svijeta, za pohranu znakova morat ćemo upotrijebiti neki drugi, širi tip. Razjasnimo ukratko problem pohranjivanja znakova.

Tip `char` je tipično duljine 1 bajta, tj. 8 bitova. Budući da se s 8 bitova može prikazati 2^8=256 različitih brojeva (tj. brojevi od 0 do 255), to znači da će se tipom `char` moći istovremeno prikazati najviše 256 znakova. ASCII je rezervirao kôdove od 0 do 127 za sve znakove engleskog alfabeta, decimalne znamenke, znakove interpunkcije i kontrolne kôdove. Za nacionalne znakove ne-engleskih jezika na raspolaganju su ostali kodovi od 127 do 255[1]. Taj skup je dovoljan da obuhvati jezično specifične znakove samo za nekoliko jezika. Zato su uvedena tablice (tzv. „kodiranja”) za 8-bitne znakove za pojedine grupe srodnih jezika. Tako primjerice prema ISO-8859-1 kôdiranju taj skup popunjavaju znakovi zapadnoeuropskih pisama, prema ISO-8859-2 kôdiranju popunjavaju znakovi srednjeeuropskih jezika (uključujući Hrvatski), a prema ISO-8859-5 kôdiranju slova ćirilice.

Time je riješen problem razmjene tekstova unutar područja u kojem se koristi isto 8-bitno kodiranje. Na primjer, pošaljete li datoteku s našim slovima kodiranu prema standardu ISO-8859-2 nekom Mađaru, on će sve naše znakove vidjeti korektno, jer se i u Mađarskoj koristi isti standard. Međutim, pošaljete li taj dokument nekome Dancu ili bilo kome u zapadnoj Europi, njemu će se umjesto naših slova pojaviti potpuno drugačiji znakovi. Na primjer, umjesto slova „ć”, on će na ekranu imati slovo „æ”. Uspije li promijeniti kodiranje tako da mu se prikaže pravi znak, više neće imati na raspolaganju „svoje” znakove.

Da se izbjegnu ovakva ograničenja u razmjeni dokumenata, napravljen je univerzalni svjetski standard – *Unicode*. U taj standard su uključeni znakovi svih svjetskih pisama, uključujući ćirilicu, starogrčko, arapsko, japansko, kinesko pismo, egipatske hijeroglife, Braillovo pismo, simbole poput notnih znakova, matematičkih simbola, šahovskih figura. Zbog toga će bilo koji dokument pisan prema standardu Unicode biti korektno interpretiran i prikazan[2] u bilo kojem dijelu svijeta. Zato korištenje Unicoda u programima olakšava izradu programa za jezično različita područja. Pri tome treba biti svjestan da Unicode definira nekoliko kodiranja kojima se omogućava pohranjivanje i

[1] Doduše, postoje neki lokalni 7-bitni „standardi” koji znakove unutar ASCII skupa nadomještaju jezično specifičnim znakovima, no oni se danas koriste izuzetno rijetko.

[2] Uvjet za ispravan prikaz jest da pismo (*font*) koji je na raspolaganju na računalu krajnjeg korisnika sadrži grafeme za sve znakove.

prikaz znakova: UTF-8, UTF-16 i UTF-32[1]. Koristimo li neko od Unicode kodiranja u programskom kôdu, operacijski sustav na računalu mora podržavati to kodiranje budući da je operacijski sustav zadužen za pretvorbu brojčanih kôdova u grafički prikaz na zaslonu i obrnuto.

Tip `wchar_t` je širi od tipa `char` pa omogućava pohranjivanje šireg skupa znakova. Točna duljina tipa `wchar_t` nije definirana standardom jezika C++; `wchar_t` zauzima barem 2 bajta (odnosno, omogućava prikaz barem 65536 različitih znakova) i u tom slučaju možemo unutar istog programa kombinirati slova različitih pisama podržana Unicode kodiranjem UTF-16:

```
wchar_t domaceJeDomace{L'ć'};
wchar_t ovakoToPisuPrekoDrine{L'ћ'};        // ćirilično slovo 'ć'
wchar_t negdjeDalje{L'ŋ'};                  // starogrčko slovo 'eta'
wchar_t josDalje{L'哀'};                    // japansko slovo
```

Prefix `L` naznačava da je znak unutar jednostrukih navodnika tipa `wchar_t`. Napomenimo da za ispravan prikaz svih znakova u datoteci izvornog kôda mora i datoteka biti pohranjena u tom kodiranju. U protivnom smo prisiljeni umjesto odgovarajućih simbola pisati njihove Unicode kodove u heksadekadskom obliku. Konkretno, za prethodne znakove trebali bismo pisati:

```
wchar_t domaceJeDomace{L'\u0107'};          // slovo 'ć'
wchar_t ovakoToPisuPrekoDrine{L'\u045b'};   // ćirilično slovo 'ć'
wchar_t negdjeDalje{L'\u03b7'};             // starogrčko slovo 'eta'
wchar_t josDalje{L'\u54c0'};                // japansko slovo
```

Niz `\u` je uputa prevoditelju da četiri heksadekadske znamenke koje slijede predstavljaju Unicode kôd koji treba prije prevođenja pretvoriti u odgovarajući simbol. Za znakove čiji kodovi se ne mogu prikazati pomoću četiri heksadekadske znamenke, tj. čiji kôd nadmašuje 65536, koristi se prefiks `\U` iza kojeg mora slijediti niz od 8 heksadekadskih znamenki. Za slova engleske abecede, dekadske znamenke i simbole interpunkcije nema takvih problema budući da su njihovi brojčani kodovi u svim kodiranjima jednaki!

Za pravilan ispis znakova tipa `wchar_t` treba koristiti izlazni tok `wcout`:

```
wcout << L'a' << endl;
```

Naime, kako su znakovi `wchar_t` dulji od jednog bajta, izlazni tok `cout` ih neće prepoznati kao znakove već će ih interpretirati kao brojeve i kao takve ispisivati.

Zadatak: *Usporedite što će ispisati sljedeće naredbe:*

```
cout << L'a' << endl;
cout << (int)'a' << endl;
```

[1] UTF-8 koristi jedan bajt za pohranjivanje znakova engleske abecede, dok se dva, tri ili četiri bajta koriste za ostala pisma. UTF-16 koristi 2 ili 4 bajta, dok UTF-32 koristi 4 bajta za sve znakove.

Budući da duljina tipa `wchar_t` nije strogo definirana standardom, nemoguće je unaprijed znati koje kodiranje će se tim tipom moći prikazati. Neke implementacije taj tip definiraju da je duljine 16, a neke 32 bita. Zato duljina tipa `wchar_t` može biti prevelika da se u potpunosti iskoristi prostor koji pruža.

Da bi se izbjegle takve nedosljednosti standard C++11 uveo je tipove `char16_t` i `char32_t` koji su duljine 16 odnosno 32 bita pa služe za pohranu znakova u UTF-16, odnosno UTF-32 kodiranju. Za označavanje kodiranja koriste se novi prefiksi `u` (UTF-16) i `U` (UTF-32):

```
char16_t slovoA{u'a'};
// u nekoj datoteci izvornog kôda koja je kodirana po UTF-16:
char16_t dvijeOsminke{u'♫'};
// u nekoj datoteci izvornog kôda koja je kodirana po UTF-32:
char32_t omega{U'ω'};
```

Prilikom korištenja bilo kojeg kodiranja treba biti svjestan da ga u prvom redu mora podržavati prevoditelj. Osim toga, za jednostavan unos lokaliziranih znakova bilo bi poželjno i da urednik teksta u kojem se piše izvorni kôd također podržava dotično kodiranje (i da može pohraniti datoteku u tom kodiranju) da bi se ti znakovi prikazali u svom pravom obliku. Na primjer, ako bismo se odlučili koristiti UTF-16 kodiranje, u postavkama urednika teksta moramo zadati da se tekst izvornog kôda pohranjuje u datoteku s tim kodiranjem, tako da prevoditelj za znak ♫ u prethodnom primjeru doista dobije kôd tog znaka u UTF-16, a ne u nekoj drugoj kodnoj stranici. Međutim, ako promijenimo kodiranje datoteka izvornog kôda, naletjet ćemo na drugi problem: sva zaglavlja biblioteka koje uključujemo moraju biti u istom kodiranju, jer prevoditelji u pravilu ne trpe miješanje kodnih stranica datoteka. Jedno rješenje za takvu situaciju jest koristiti neko 8-bitno kodiranje (npr. ASCII ili UTF-8) za datoteke izvornog kôda, a sve znakove koji nisu iz skupa ASCII znakova unositi preko njihovih brojčanih kodova, kao na primjer:

```
// u nekoj datoteci izvornog kôda koja ne mora biti kodirana po UTF-16
char16_t dvijeOsminke{u'\u266b'};
```

Sljedeći problem na koji možemo naletjeti jest da standardne funkcije za rad sa znakovima ne podržavaju odabrano kodiranje. I konačno zadnja stavka o kojoj valja voditi računa: operacijski sustav na kojem će se program izvoditi mora podržavati odabranu kodnu stranicu.

Ukoliko želimo razvijati aplikacije koje će pružati dobru podršku za internacionalizaciju, najjednostavnije je za datoteke izvornog kôda koristiti UTF-8 kodiranje koje podržava većina današnjih prevoditelja i većina urednika teksta. U tom kodiranju, znakovi iz ASCII skupa imaju potpuno identične kôdove pa možemo bez ikakvih problema uključivati standardne biblioteke – one sadrže isključivo znakove iz ASCII skupa budući da su pisane na engleskom jeziku. Osim toga, u aplikacijama prilagođenima internacionalizaciji (što znači da se prilikom izvođenja programa sve poruke prikazuju na jeziku kojeg odabere krajnji korisnik), tekstovi poruka se ne upisuju u datoteke u kojima se nalazi izvorni kôd, već se izdvajaju u zasebne *datoteke resursa* koje se učitavaju iz programskog kôda. Time se odvaja prikaz od logike programa, što omogućava pisanje

programskog kôda neovisno o tekstovima poruka. Na taj način program možemo vrlo jednostavno prilagoditi za drugi jezik tako da samo prevedemo tekstove poruka u datotekama resursa, bez promjena u glavnom programskom kôdu.

Pažljiviji čitatelj će se zasigurno zapitati: „Ako postoje posebni tipovi za UTF-16 i za UTF-32 znakove, zašto ne postoji za UTF-8?”. Znakovi u UTF-8 kodiranju nemaju jednake duljine pa je rukovanje pojedinačnim znakovima vrlo nespretno. Za zadavanje znakovnih nizova u UTF-8 kodiranju postoji poseban prefiks i njega ćemo spomenuti u poglavlju 5.2.

3.4.11. Operatori pridruživanja (2 ½)

Osim već obrađenog operatora =, jezik C++ za aritmetičke i bitovne operatore podržava i operatore *obnavljajućeg pridruživanja* (engl. *update assignment*) koji se sastoje se od znaka odgovarajućeg aritmetičkog ili bitovnog operatora i znaka jednakosti. Operatori obnavljajućeg pridruživanja omogućavaju kraći zapis naredbi. Na primjer, naredba

```
a += 5;
```

ekvivalentna je naredbi

```
a = a + 5;
```

U tablici 3.13 dani su svi operatori pridruživanja. Primjenu nekolicine operatora ilustrirat ćemo sljedećim kôdom:

Tablica 3.13. Operatori pridruživanja

=	+=	-=	*=	/=	%=	>>=	<<=	^=	&=	|=

```
int n{10};
n += 5;                 // isto kao: n = n + 5
cout << n << endl;      // ispisuje: 15
n -= 20;                // isto kao: n = n - 20
cout << n << endl;      // ispisuje: -5
n *= -2;                // isto kao: n = n * (-2)
cout << n << endl;      // ispisuje: 10
n %= 3;                 // isto kao: n = n % 3
cout << n << endl;      // ispisuje 1
```

Pri korištenju operatora obnavljajućeg pridruživanja valja znati da operator pridruživanja ima niži prioritet od svih aritmetičkih i bitovnih operatora. Stoga, želimo li naredbu

```
a = a - b - c;
```

napisati kraće, nećemo napisati

```
a -= b - c;          // to je zapravo a = a - (b - c)
```

već kao

```
a -= b + c;          // a = a - (b + c)
```

Operator `-=` ima niži prioritet od ostalih aritmetičkih operatora. Izraz s desne strane je zapravo izraz u zagradi ispred znaka oduzimanja.

3.4.12. Alternativne oznake operatora

Na nekim starijim operacijskim sustavima (npr. starije inačice MS DOS-a) koristilo se 7-bitno kodiranje, gdje za prikaz svih znakova stoji na raspolaganju samo 128 kôdova. Da bi se s takvim kodiranjem mogli prikazati nacionalni znakovi, neki rjeđe korišteni specijalni znakovi poput `[`, `]`, `{`, `}`, `|`, `\`, `~` i `^` nadomješteni su u pojedinim europskim jezicima nacionalnim znakovima kojih nema u engleskom alfabetu. Tako su u hrvatskoj inačici (udomaćenoj pod nazivom CROSCII) ti znakovi nadomješteni slovima Š, Ć, š, ć, đ, Đ, č, odnosno Č. Očito je da će na računalu koje podržava samo takav sustav znakova biti nemoguće pregledno napisati čak i najjednostavniji C++ program. Na primjer, na zaslonu bi kôd nekog trivijalnog programa mogao izgledati ovako (onaj tko „dešifrira" kôd, zaslužuje brončani *Velered Bjarnea Stroustrupa*, bez pletera):

```
int main() š
    int a = 5;
    char b = 'Đ0';
    int c = ča Č b;
    return 0;
ć
```

Budući da prevoditelj interpretira kôdove pojedinih znakova, a ne njihove grafičke prezentacije na zaslonu, program će biti preveden korektno. Međutim, za čovjeka je kôd nečitljiv. Osim toga, ispis programa nećete moći poslati svom prijatelju u Njemačkoj, koji ima svojih briga jer umjesto vitičastih zagrada ima znakove ü i ß.

Da bi se izbjegla ograničenja ovakve vrste, ANSI komitet je predložio alternativne oznake za operatore koji sadrže „problematične" znakove (tablica 3.14). Srećom, današnji operacijski sustavi za predstavljanje znakova koriste standarde koji omogućavaju istovremeno korištenje nacionalnih znakova i specijalnih znakova neophodnih za pisanje programa u jeziku C++, tako da većini korisnika tablica 3.14 neće nikada zatrebati. Ipak, neki programeri prednost daju duljim oznakama logičkih operatora (npr. `and` umjesto `&&` ili `or` umjesto `||`) jer ih nalaze čitljivijima, kao u sljedećem primjeru:

Tablica 3.14. Alternativne oznake operatora

osnovna	*alternativa*	*osnovna*	*alternativa*	*osnovna*	*alternativa*
{	<%	&&	and	~	compl
}	%>	||	or	!=	not_eq
[	<:	!	not	&=	and_eq
]	:>	&	bitand	|=	or_eq
#	%:	|	bitor	^=	xor_eq
##	%:%:	^	xor		

```
bool prva = true;
bool druga = false;
bool treca = prva and druga;
cout << treca << endl;
cout << not treca << endl;
cout << prva or druga << endl;
```

3.4.13. Deklaracija `auto` i operator `decltype`

U dosadašnjim primjerima eksplicitno smo definirali tip podatka. Međutim, to možemo prepustiti i prevoditelju tako da tip deklariramo ključnom riječi `auto`:

```
auto a = 5.34;      // tipa double
auto b = 3;         // tipa int
auto c = a * b;     // tipa double
```

Prevoditelj će analizom kôda sam odrediti kojeg tipa su izrazi koji se pridružuju varijablama `a`, `b`, odnosno `c` i na osnovu toga zadati njihov tip. Za varijable `a` i `b` očito je da će biti tipa `double`, odnosno `int`, s obzirom da su tog tipa konstante desno od operatora pridruživanja. U trećoj naredbi prevoditelj će, analizom izraza i koristeći pravila aritmetičke pretvorbe, prepoznati da je umnožak tipa `double` pa će varijablu `c` deklarirati da je tipa `double`. Dakle, prevoditelj će naredbe prevesti kao da smo napisali:

```
// umjesto prevoditelja, mogli smo sami zaključiti:
double a{5.34};
int b{3};
double c = a * b;
```

U gornjem primjeru smo varijable deklarirali kao `auto` iz čiste komocije (profinjeni izraz za lijenost). U pravilu nije dobro prepustiti prevoditelju da sam odlučuje o tipu objekta prilikom inicijalizacije, jer će nam se to prije ili kasnije obiti o glavu – može se dogoditi da objekt ne bude tipa kojeg očekujemo i da to ne primijetimo pravovremeno. Pretpostavimo da smo izvorno zamislili da varijabla `b` bude tipa `double` i da kasnije želimo izračunati polovicu njene vrijednosti. Uz gornje deklaracije će naredba:

```
double polovica = b / 2;
```

rezultirati cjelobrojnim dijeljenjem te neće dati stvarnu polovicu broja. No postoje situacije kada je tip doista nemoguće jednostavno odrediti, primjerice kada ga treba odrediti iz rezultata nekog složenog izraza, rezultata koji vraća funkcija ili predložak funkcije (§12.2).

Primijetimo kako deklaracija `auto` ne unosi nikakvu tipsku nesigurnost u programski kôd – prevoditelj i dalje ima potpuni nadzor nad tipom podatka i prijavit će pogreške pokuša li programer provesti nedozvoljene operacije nad dotičnim objektom. Na primjer, dodamo li naredbu za bitovni pomak varijable `c` za 4 bita udesno:

```
// ...
auto c = a * b;
c = c >> 4;         // pogreška: operator >> nije definiran za double
```

prevoditelj će prijaviti pogrešku da za tip `double` nije definiran operator `>>`.

Budući da prevoditelj mora odrediti tip izraza, varijablu tipa `auto` nije dovoljno samo deklarirati, već ju odmah treba i inicijalizirati. Zato će prevoditelj za sljedeću deklaraciju javiti pogrešku:

```
auto d;         // pogreška: nije moguće odrediti kojeg je tipa d
```

Isto tako, inicijalizacija objekta tipa `auto` pomoću liste inicijalizatora gotovo da nema smisla, jer neće stvoriti tip koji najvjerojatnije očekujemo. Tako bismo u sljedećem primjeru sasvim logično mogli pretpostaviti da će varijabla `d` biti tipa `int`, no naredba koja slijedi će nas u tome razuvjeriti:

```
auto d{5};
d += 4;         // pogreška: nije definiran operator += za tip
                // std::initializer_list<int>
```

Kako unutar liste inicijalizatora općenito može biti više podataka, prevoditelj u ovom slučaju ne može jednoznačno odrediti želimo li inicijalizirati objekt tipa `int`, niz podataka (§5.1) tipa `int` koji sadrži samo jedan član ili neki objekt čijem se konstruktoru (§9.4.1) prosljeđuje podatak tipa `int`. Stoga, na osnovu podataka unutar vitičastih zagrada, stvara objekt tipa `initializer_list<`*`Tip`*`>`, tj. listu objekata nekog tipa *`Tip`*. Tip `initializer_list` je predložak klase (§12.3) definiran u zaglavlju `initializer_list`. Prevoditelj će sam odrediti koji je *`Tip`* objekta u listi – u gornjem slučaju će to biti tip `int`. S takvom listom ne možemo puno toga napraviti jer ona podržava samo dohvaćanje broja članova te dohvaćanje pojedinih članova u listi. Zato će, u konačnici, za zadnju naredbu prevoditelj prijaviti pogrešku.

Objekt tipa `auto` uvijek treba inicijalizirati operatorom pridruživanja.

Trebamo li neki objekt deklarirati da je istog tipa kao neki već postojeći objekt, možemo upotrijebiti operator `decltype`. Operator `decltype` kao rezultat vraća tip izraza koji mu se navede. Kako se taj tip određuje tijekom prevođenja, rezultat se može upotrijebiti za deklaraciju tipa. Na primjer:

```
double a{5.34};
int b{3};
decltype(a) c = a + 3;      // c će biti istog tipa kao a (tj. double)
decltype(b) d = a + 3;      // d će biti istog tipa kao b (tj. int)
decltype(a + b) e = a + b;  // isto kao da smo stavili auto e = a + b;
```

U zadnjoj naredbi smo kao argument operatoru `decltype` naveli izraz. Prevoditelj će na osnovu pravila aritmetičke pretvorbe (str. 67) zaključiti da je rezultat tog izraza tipa `double` pa će i objekt `e` proglasiti da je tog tipa. Isti učinak mogao se postići jednostavnom deklaracijom `auto`:

```
auto e = a + b;             // isti efekt kao i prethodna naredba
```

Uočimo kako prevoditelj pri određivanju tipa ne izračunava brojčanu vrijednost izraza jer vrijednosti varijabli `a` i `b` u općem slučaju ne moraju biti poznate tijekom prevođenja – to mogu biti podaci učitani iz datoteke ili uneseni tipkovnicom.

Operator `decltype` koristit ćemo u sličnim situacijama kao i deklaraciju `auto`: kada ne znamo unaprijed kojeg će tipa biti neki objekt ili je to vrlo teško iz kôda iščitati. U primjeru s početka ovog odjeljka prepustili smo prevoditelju da odredi kojeg će tipa biti varijabla `c`. Pretpostavimo da želimo uvesti varijablu `d` za koju želimo da je istog tipa. U tom slučaju možemo napisati:

```
double a{5.34};
int b{3};
auto c = a * b;
decltype(c) d = b * b;      // d će biti tipa double, bez obzira
                            // što je kvadrat od b tipa int
```

Pravu korist deklaracije `auto` i operatora `decltype` upoznat ćemo u sljedećim poglavljima, posebice kada ćemo govoriti o predlošcima (§12) i o standardnim spremnicima (§13.3).

3.4.14. Korisnički definirani tipovi i operatori

Osim ugrađenih tipova podataka koje smo upoznali u prethodnim odjeljcima, te operatora koji su za njih definirani, u programskom jeziku C++ na raspolaganju su izvedeni tipovi podataka kao što su polja, pokazivači i reference (njih ćemo upoznati u sljedećem poglavlju). Međutim, ono što programski jezik C++ čini naročito moćnim su *klase* (*razredi*) koje omogućavaju uvođenje potpuno novih, *korisnički definiranih tipova*. Tako programer više nije sputan osnovnim tipovima koji su ugrađeni u jezik, već ih može po volji dopunjavati, definirajući nove tipove na osnovu već postojećih. Primjerice, može definirati tip koji će predstavljati kompleksni broj. Takav tip će se sastojati od dva de-

cimalna broja koji će sadržavati podatke o realnom i imaginarnom dijelu kompleksnog broja te će definirati pripadajuće operacije nad kompleksnim brojevima. Drugi primjer je tip `string` kao jedan od najzanimljivijih tipova definiranih u standardnoj biblioteci. On služi za pohranu niza znakova, tj. nekog proizvoljnog teksta:

```
#include <string>       // zaglavlje u kojem je definiran tip string
#include <iostream>
using namespace std;

//...
string tekst{"Trla baba lan"};  // inicijaliziramo objekt tekst
                                // tipa string
cout << text << endl;           // ispisujemo sadržaj objekta
tekst = "da joj proće dan";     // mijenjamo sadržaj objekta
```

Svakako valja naglasiti da korisnički definirani tipovi ne sadrže samo podatke nego i objedinjuju operacije nad tim podacima. Na primjer, tip `string` definira funkcijski član `find()` koji pretražuje niz znakova sadržanih u dotičnom objektu:

```
string tekst{"Trla baba lan"};
// u gore zadanom tekstu tražimo prvu pojavu teksta "la":
cout << tekst.find("la") << endl;          // ispisuje 2
```

U ovom primjeru smo stvorili objekt tipa `string` i njega inicijalizirali nekim tekstom. U drugoj naredbi smo, navodeći ime objekta i ime funkcijskog člana međusobno odvojene točkom, pozvali funkcijski član `find()` tog objekta i proslijedili smo mu tekst (`"la"`) koji želimo pronaći unutar početno zadanog teksta. Funkcijski član `find()` kao rezultat vraća prvu poziciju na kojoj će pronaći traženi tekst. Budući da početno slovo `'t'` u nizu ima indeks 0, druga naredba će u našem primjeru kao rezultat ispisati broj 2.

Detaljnije o smislu klasa, kako ih koristiti te na koji način ih možemo sami definirati bit će govora u poglavlju 9.

3.4.15. Pseudonimi tipova

Ponekad je čitljivije ako se umjesto oznaka ugrađenih tipova prilikom deklaracije objekta upotrijebi neki sinonim čiji naziv preciznije opisuje sam objekt. Na primjer, želimo li rukovati visinom i težinom osoba u našem kôdu, možemo napisati sljedeće:

```
using visinaCm = int;       // uveli smo novi naziv za tip int
using masaKg = double;      // uveli smo novi naziv za tip double

cout << "Upišite svoju visinu u cm: ";
visinaCm v;
cin >> v;
cout << "Upišite svoju masu u kg: ";
masaKg m;
cin >> m;
```

```
double itm = m / (v * v) * 10000;
cout << "Vaš indeks tjelesne mase je: " << itm << endl;
```

Prvim dvjema naredbama uveli smo prilagođene nazive za tipove `int`, odnosno `double`, koje potom koristimo u daljnjem kôdu. Za deklaraciju pseudonima (engl. *alias declaration*) koristi se ključna riječ `using`, s kojom smo se već dobro upoznali kroz dosadašnje primjere budući da smo pomoću nje aktivirali imenski prostor `std`. No, za razliku od tih deklaracija, deklaracija pseudonima ima oblik:

```
using noviNazivTipa = izvorniNazivTipa;
```

Osim u ovakvim jednostavnim slučajevima, pseudonimi se često koriste za deklaracije pokazivača i referenci na objekte, pokazivača na funkcije, kod deklaracija polja, a posebice pri korištenju predložaka klasa. Naime, sintaksa kojom se ti tipovi opisuju često je vrlo složena, zbog čega programski kôd može postati nečitljiv. U tom slučaju, jednostavnije definirati jezgrovitiji i čitljiviji sinonim, kao npr.:

```
using obilazTablice = unordered_map<string, double>::const_iterator;
```

Pseudonimi za tipove često mogu olakšati sustavne promjene kôda. Pretpostavimo da smo napisali program za neki numerički proračun tako da su nam svi brojčani podaci tipa `float`. Nakon nekog vremena utvrdili smo da nam `float` ne zadovoljava uvijek glede točnosti, te da neke brojeve moramo deklarirati kao `double`. U najgorem slučaju to znači pretraživanje kôda i ručnu zamjenu odgovarajućih deklaracija `float` u deklaracije `double` ili obrnuto (ako se predomislimo).

```
// prije promjene:
float a, b;
float k, h;
// ...
float x, y;
```

```
// nakon promjene:
double a, b;
float k, h;
// ...
double x, y;
```

Kada je broj objekata mali to i nije teško, ali za veliki broj deklaracija te zamjene mogu biti naporne i podložne pogreškama. Posao ćemo si značajno olakšati ako u gornjem primjeru uvedemo pseudonim za podatke kojima ćemo po potrebi mijenjati tip:

```
using brojevi = float;
brojevi a, b;
float k, h;
// ...
brojevi x, y;
```

Želimo li sada promijeniti tipove, dovoljno je samo gornju deklaraciju pseudonima zamijeniti sljedećom:

```
using brojevi = double;
```

Nakon toga će svi podaci tipa `brojevi` biti prevedeni kao podaci tipa `double`, bez potrebe za daljnjim izmjenama u izvornom kôdu.

Budući da deklaracija pseudonima ne uvodi novi tip podataka niti mijenja uobičajena pravila pretvorbe podataka, sljedeća pridruživanja su dozvoljena i neće prouzročiti nikakve promjene u točnosti:

```
double a = pi;
using plivajuci = double;
plivajuci b = pi;
```

Pseudonimi su potpuno ravnopravni svim ostalim tipovima, te se mogu pojaviti na svim mjestima na kojima se može pojaviti bilo koji tip: prilikom deklaracije objekata, specificiranja parametara funkcije, kao parametar operatoru `sizeof` (§3.5) i sl.

Prije nego što je standardom C++11 uvedena deklaracija pseudonima pomoću ključne riječi `using`, novo ime za već postojeći tip moglo se uvesti isključivo[1] korištenjem ključne riječi `typedef`:

```
typedef double broj;
typedef unordered_map<string, double>::const_iterator obilazTablice;
```

Kao što možemo primijetiti, nema operatora pridruživanja, a redoslijed navođenja novog imena i imena postojećeg tipa je obrnut. Iako je deklaracija pomoću `typedef` i dalje podržana, sintaksa deklaracije pseudonima pomoću `using` je logičnija pa ćemo njoj davati prednost.

3.5. Operator `sizeof`

Operator `sizeof` je unarni operator koji kao rezultat daje broj bajtova što ih operand zauzima u memoriji računala:

```
cout << "Duljina podataka tipa int je " << sizeof(int)
     << " bajtova" << endl;
```

Valja naglasiti da standard jezika C++ ne definira veličinu bajta (tj. broj bitova koji čine bajt), osim u smislu rezultata što ga daje `sizeof` operator; tako `sizeof(char)` uvijek mora biti jednak 1. Naime, duljina bajta ovisi o arhitekturi računala. Mi ćemo u knjizi uglavnom podrazumijevati da bajt sadrži 8 bitova, što je najčešći slučaj u praksi.

Operand operatora `sizeof` može biti identifikator tipa (npr. `int`, `float`, `char`) ili konkretni objekt koji je već deklariran:

```
float f;
cout << sizeof(f) << endl;      // duljina float-a
int i;
cout << sizeof(i) << endl;      // duljina int-a
```

[1] Izuzmemo li zamjene pomoću pretprocesora (§23.3)

Operator `sizeof` se može primijeniti i na izraz, koji se u tom slučaju ne izračunava već se određuje memorijski prostor koji njegov rezultat zauzima. Zbog toga će sljedeće naredbe ispisati duljine `float`, odnosno `int` rezultata:

```
float f;
int i;
cout << sizeof(f * i) << endl;                      // duljina float-a
cout << sizeof(static_cast<int>(i * f)) << endl;    // duljina int-a
```

Operator `sizeof` se može primijeniti i na pokazivače, reference, polja, korisnički definirane tipove i objekte, s kojima ćemo se upoznati u sljedećim poglavljima. Ne može se primijeniti na funkcije (na primjer, da bi se odredilo njihovo zauzeće memorije), ali se može primijeniti na pokazivače na funkcije. U svim slučajevima on vraća ukupnu duljinu tih objekata izraženu u bajtovima. Kako pokazivači u biti predstavljaju memorijske adrese, operator `sizeof` će vratiti broj bajtova koliko ih pokazivač zauzima u memoriji. Memorijske adrese su, bez obzira na tip objekta na koji ti pokazivači pokazuju jednake duljine, pa će `sizeof` uvijek vratiti isti broj, ovisan samo o prevoditelju i platformi.

Rezultat operatora `sizeof` je tipa `size_t`, cjelobrojni tip bez predznaka koji ovisi o implementaciji prevoditelja, definiran u zaglavlju `cstddef`.

Operator `sizeof` se uglavnom koristi kod dinamičkog alociranja memorijskog prostora kada treba izračunati koliko memorije treba osigurati za neki objekt, o čemu će biti govora u kasnije u knjizi.

3.6. Operator razdvajanja

Operator razdvajanja , (zarez) koristi se za razdvajanje izraza u naredbama. Izrazi razdvojeni zarezom se izvode postepeno, s lijeva na desno. Tako će nakon naredbe:

```
j = (i = 10, i + 5);
```

varijabli `j` biti pridružena vrijednost 15. Prilikom korištenja operatora razdvajanja u složenijim izrazima valja biti vrlo oprezan, jer on ima najniži prioritet (vidi sljedeći odjeljak o hijerarhiji operatora). To se posebice odnosi na pozive funkcija u kojima se zarez koristi za razdvajanje argumenata.

Ovaj operator se često zna koristiti kod istovremene deklaracije više varijabli istog tipa. Na primjer, naredbom:

```
int i, j;
```

deklariramo dvije cjelobrojne varijable. Varijable možemo odmah i inicijalizirati, ali treba voditi računa da se inicijalizacija mora provesti za svaku varijablu zasebno:

```
int i = 1, j = 1;
int m, n = 5;       // oprez: varijabla m nije inicijalizirana!
```

Budući da se deklaracija tipa može navesti samo na početku naredbe, operatorom razdvajanja ne možemo deklarirati varijable različitog tipa:

```
int i, float a;     // pogreška: tip se smije navesti samo na
                    // početku naredbe!
```

Upravo zbog navedenih zamki i nedostataka treba izbjegavati višestruke deklaracije (i inicijalizacije) u istoj naredbi.

Preglednije je deklaracije i eventualne inicijalizacije više varijabli razdvojiti u zasebne naredbe:

```
int i = 1;
int j = 1;
```

Operator razdvajanja ponekad se koristi za razdvajanje više izraza u parametrima `for` petlje (§4.3.1).

Zadatak: *Provjerite hoće li za naredbe:*

```
int a;
a = 3, 4, 5;
```

prevoditelj prijaviti pogrešku. Ako neće, provjerite što će biti rezultat varijable a nakon izvođenja naredbi. Obrazložite rezultat.

3.7. Hijerarhija i redoslijed izvođenja operatora

U matematici postoji strogo utvrđena hijerarhija operacija prema kojoj neke operacije imaju prednost pred drugima. Podrazumijevani slijed operacija je slijeva nadesno, ali ako se dvije operacije različitog prioriteta nađu jedna do druge, prvo se izvodi operacija s višim prioritetom. Na primjer u matematičkom izrazu

$$a + b \cdot c / d$$

množenje broja *b* s brojem *c* ima prednost pred zbrajanjem s brojem *a*, tako da se ono izvodi prvo. Umnožak se zatim dijeli s *d* i tek se tada pribraja broj *a*.

I u programskom jeziku C++ definirana je hijerarhija operatora. Prvenstveni razlog tome je kompatibilnost s matematičkom hijerarhijom operacija, što omogućava pisanje računskih izraza na gotovo identičan način kao u matematici. Stoga gornji izraz u jeziku C++ možemo pisati kao

```
y = a + b * c / d;
```

Redoslijed izvođenja operacija će odgovarati matematički očekivanom. Operacije se izvode prema hijerarhiji operacija, počevši s operatorima najvišeg prioriteta. U tablici

3.15 na stranici 97 dani su svi operatori svrstani po hijerarhiji od najvišeg do najnižeg. Operatori s istim prioritetom smješteni su u zajedničke blokove.

Striktno definiranje hijerarhije i slijeda izvođenja je neophodno i zato jer se neki znakovi koriste za više namjena. Tipičan primjer je znak - (minus) koji se koristi kao binarni operator za oduzimanje, kao unarni operator za promjenu predznaka, te u operatoru za umanjivanje (--). Takva višeznačnost može biti uzrokom čestih pogrešaka.
Ilustrirajmo to sljedećim primjerom. Neka su zadana dva broja *a* i *b*; želimo od broja *a* oduzeti broj *b* prethodno umanjen za 1. Neopreznom programeru može se dogoditi da umjesto:

```
c = a - --b;
```

za to napiše naredbu:

```
c = a---b;
```

Što će stvarno ta naredba uraditi pouzdano ćemo saznati ako ispišemo vrijednosti svih triju varijabli nakon naredbe:

```
int main()
{
    int a{2};
    int b{5};
    int c = a---b;
    cout << "a = " << a << ", b = " << b << ", c = " << c << endl;
    return 0;
}
```

Izvođenjem ovog programa na zaslonu će se ispisati

```
a = 1, b = 5, c = -3
```

što znači da je prevoditelj naredbu interpretirao kao

```
c = a-- - b;
```

tj. uzeo je vrijednost varijable `a`, od nje oduzeo broj `b` te rezultat pridružio varijabli `c`, a varijablu `a` je umanjio (ali tek nakon što je upotrijebio njenu vrijednost).

Kao i u matematici, okruglim zagradama može se zaobići ugrađena hijerarhija operatora, budući da one imaju viši prioritet od svih operatora. Tako će se u kôdu

```
d = a * (b + c);
```

prvo zbrojiti varijable `b` i `c`, a tek potom će se njihov zbroj pomnožiti s varijablom `a`. Često je zgodno zagrade koristiti i radi čitljivosti kôda kada one nisu neophodne. Na primjer, u gore razmatranom primjeru mogli smo za svaki slučaj pisati:

Tablica 3.15. Hijerarhija operatora

operator	značenje	primjer
`[](){ }`	lambda izraz	
`::`	globalno područje	*::ime*
`::`	područje klase	*klasa::ime*
`::`	područje imenika	*imenik::ime*
`.`	izbor člana	*objekt.clan*
`->`	izbor člana	*pokazivac->clan*
`[]`	indeksiranje	*varijabla*[indeks]
`()`	poziv funkcije	*funkcija*(*argumenti*)
`()`	stvori tip	*tip*(*argumenti*)
`++`	uvećaj nakon	*lvrijednost*++
`--`	umanji nakon	*lvrijednost*--
`typeid`	identifikacija tipa	typeid(*izraz*)
`const_cast`	pretvorba nepromjenjivosti tipa	const_cast<*tip*>(*izraz*)
`dynamic_cast`	pretvorba tipa tijekom izvođenja	dynamic_cast<*tip*>(*izraz*)
`static_cast`	pretvorba tipa tijekom prevođenja	static_cast<*tip*>(*izraz*)
`reinterpret_cast`	neprovjerena pretvorba tipa	reinterpret_cast<*tip*>(*izraz*)
`sizeof`	veličina objekta	sizeof *izraz*
`sizeof`	veličina tipa	sizeof(*tip*)
`sizeof...`	veličina paketa parametara	sizof... *ime*
`alignof`	poravnavanje tipa	alignof(*tip*)
`++`	uvećaj prije	++*lvrijednost*
`--`	umanji prije	--*lvrijednost*
`+ - ! ~`	unarni operatori	~*lvrijednost*
`*`	dereferenciranje	**izraz*
`&`	adresa objekta	&*lvrijednost*
`new`	stvori (alociraj) objekt	new *tip*
`new`	stvori (i inicijaliziraj) objekt	new *tip*(*argumenti*)
`new`	stvori (smjesti) objekt	new(*argumenti*) *tip*
`new`	stvori (smjesti i inicijaliziraj) objekt	new(*argumenti*) *tip*(*argumenti*)
`delete`	uništi (dealociraj) objekt	delete *pokazivac*
`delete []`	uništi (dealociraj) niz	delete[] *pokazivac*
`noexcept`	baca li iznimku	neoexcept(*izraz*)
`()`	dodjela tipa	(*tip*) *izraz*
`.*`	izbor člana	*objekt.*pokazivac_na_clan*
`->*`	izbor člana	*pokazivac->*pokazivac_na_clan*
`* / %`	množenja	*izraz* % *izraz*
`+ -`	zbrajanja	*izraz* - *izraz*
`<< >>`	bitovni pomaci	*izraz* >> *izraz*
`< > <= >=`	poredbeni operatori	*izraz* <= *izraz*
`== !=`	operatori jednakosti	*izraz* != *izraz*

(nastavlja se)

```
c = a - (--b);
```

Dobra je navika stavljati zagrade i praznine svugdje gdje postoji dvojba o hijerarhiji operatora i njihovom pridruživanju. Time kôd postaje pregledniji i razumljiviji. Pritom valja paziti da broj lijevih zagrada u naredbi mora biti jednak broju desnih zagrada – u protivnom će prevoditelj javiti pogrešku.

Da bi izbjegli ovakve zamke, mnogi programeri prilikom pisanja kôda prvo napišu lijevu i desnu zagradu, a tek potom upisuju izraz unutar zagrade. Srećom, mnogi urednici teksta danas olakšavaju provjeru uparenosti zagrada, npr. tako da istaknu u tekstu zagradu koja je u paru sa zagradom na kojoj se kursor trenutno nalazi ili definiraju prečicu kojom se kursor prebacuje na lokaciju uparene zagrade.

Prilikom pisanja složenih izraza treba voditi računa o tome da nije definiran redoslijed kojim se izračunavaju izrazi s obje strane operatora, tj. ne mora se nužno prvo izračunavati izraz s lijeve, a potom s desne strane izraza[1]. U jednostavnim matematičkim izrazima poput:

```
d = (a – b) * (b + c);
```

taj redoslijed nije niti bitan. Problem može nastati ako u izrazu pozivamo primjerice dvije funkcije:

```
d = f(c) * g(c);    // koja će biti prva pozvana: f() ili g()???
```

Tablica 3.15 *(nastavak)*

operator	značenje	primjer
`&`	bitovni *i*	*izraz* `&` *izraz*
`^`	bitovno *isključivo ili*	*izraz* `^` *izraz*
`|`	bitovni *ili*	*izraz* `|` *izraz*
`&&`	logički *i*	*izraz* `&&` *izraz*
`||`	logički *ili*	*izraz* `||` *izraz*
`?:`	uvjetni izraz	*izraz* `?` *izraz* `:` *izraz*
`= *= /= += -= &= ^= |= %= >>= <<=`	pridruživanja	*lvrijednost* `+=` *izraz*
`throw`	baci iznimku	`throw` *izraz*
`,`	razdvajanje	*izraz* `,` *izraz*

[1] Izuzetak su operatori *logički i* te *logički ili*, kod kojih se uvijek prvo vrednuje lijeva strana, kako je bilo objašnjeno u odjeljku 3.4.8.

a zbog nekog, samo nama dokučivog razloga, izuzetno je važno da se funkcija `g()` ne pozove prije funkcije `f()`.

Problem u složenim izrazima mogu izazvati i popratne pojave koje nastaju tijekom izvođenja izraza. Tipičan primjer su operatori inkrementiranja i dekrementiranja: ako se neka varijabla pojavljuje u izrazu na više mjesta, a njena vrijednost se inkrementira ili dekrementira unutar izraza, vrijednost varijable će na pojedinim mjestima biti nedefinirana pa će i konačni rezultat biti neodređen (ovisno o implementaciji prevoditelja). Najjednostavniji (iako ne baš smisleni) primjer za to jest:

```
int a{5};
a = a++;        // neodređeni izraz!
```

Standard ne definira hoće li se prvo vrijednost varijable uvećati i kao takva pridružiti ili će se prvo obaviti pridruživanje, a tek potom desna strana uvećati. Na primjer:

```
int a{5};
int zbroj = (++a) + (++a);      // neodređeni izraz!
```

Iako bi netko možda očekivao (uključujući i autore knjige) da će `zbroj` poprimiti vrijednost 13, to ne mora nužno biti rezultat! U izrazu se varijabla `a` mijenja (uvećava) na dva mjesta i standard ne jamči kada i kojim redoslijedom će se ta uvećanja izvesti: hoće li se prvo varijabla `a` uvećati i zatim ta vrijednost dodati ponovno uvećanoj vrijednosti varijable `a` ili će se varijabla `a` dvokratno uvećati i onda te vrijednosti zbrojiti.

Ovakve nedorečenosti su u standardu ostavljene namjerno kako bi se omogućilo prevoditeljima da obave optimizacije kôda unutar izraza. Da korisnici ipak ne bi bili prepušteni na milost i nemilost prevoditelja, uveden je pojam *točke slijeda* (engl. *sequence point*). Prema standardu, točka slijeda jest točka u tijeku izvođenja kôda u kojoj su okončane sve popratne pojave prethodnog izraza, a popratne pojave sljedećeg izraza nisu još iskrsnule [ISO/IEC98]. Točke slijeda čine:

- točka-zarez `;` na kraju naredbe,
- operator `,` za odvajanje izraza,
- operatori *logički i* (`&&`) i *logički ili* (`||`),
- uvjetni operator `? :` (§4.2.2),
- točka nakon izračunavanja svih argumenata funkcijama, ali prije izvođenja prve naredbe u funkciji,
- točka nakon što je funkcija preslikala objekt koji vraća kao rezultat, ali prije nego što se kôd neposredno nakon poziva funkcije počeo izvoditi,
- točka nakon inicijalizacije svih baznih tipova (§10.6) i članova u inicijalizacijskim listama njihovih konstruktora (§9.4.1).

Da se ne bismo izlagali neugodnim iznenađenjima, unutar izraza svakako treba izbjegavati višekratne promjene varijabli te korištenje varijabli čija se vrijednost mijenja.

Na primjer, zadnju naredbu trebamo razbiti u dva izraza:

```
int zbroj = ++a, zbroj += ++a;
```

ili dvije naredbe:

```
int zbroj = ++a;
zbroj += ++a;
```

Zadatak: *Objasnite zašto je izraz u drugoj naredbi neodređen:*

```
int n{2};
int rezultat = n * sqrt(--n);
```

Koji su sve mogući rezultati tog izraza? (`sqrt()` je funkcija koja vraća kvadratni korijen broja).

4. Naredbe za kontrolu toka programa

„Tko kontrolira prošlost," glasio je slogan Stranke, „kontrolira i budućnost: tko kontrolira sadašnjost kontrolira prošlost."

George Orwell (1903–1950), „1984"

U većini realnih problema tok programa nije pravocrtan i jedinstven pri svakom izvođenju. Redovito postoji potreba da se pojedini odsječci programa ponavljaju – programski odsječci koji se ponavljaju nazivaju se *petljama* (engl. *loops*). Ponavljanje može biti unaprijed određeni broj puta, primjerice želimo li izračunati umnožak svih cijelih brojeva od 1 do 100. Međutim, broj ponavljanja može ovisiti i o rezultatu neke operacije. Kao primjer za to uzmimo učitavanje podataka iz neke datoteke – datoteka se čita podatak po podatak, sve dok se ne učita znak koji označava kraj datoteke (*end-of-file*). Duljina datoteke pritom može varirati za pojedina izvođenja programa.

Gotovo uvijek se javlja potreba za grananjem toka, tako da se ovisno o postavljenom uvjetu u jednom slučaju izvodi jedan dio programa, a u drugom slučaju drugi dio. Na primjer, želimo izračunati realne korijene kvadratne jednadžbe. Prvo ćemo izračunati diskriminantu – ako je diskriminanta veća od nule, izračunat ćemo oba korijena, ako je jednaka nuli izračunat ćemo jedini korijen, a ako je negativna ispisat ćemo poruku da jednadžba nema realnih korijena. Grananja toka i ponavljanja dijelova kôda omogućavaju posebne naredbe za kontrolu toka.

4.1. Blokovi naredbi

Dijelovi programa koji se uvjetno izvode ili čije se izvođenje ponavlja grupiraju se u *blokove naredbi* – jednu ili više naredbi koje su omeđene parom otvorena-zatvorena vitičasta zagrada `{}`. Izvana se taj blok ponaša kao jedinstvena cjelina, kao da se radi samo o jednoj naredbi. S blokom naredbi susreli smo se već u prvom poglavlju, kada smo opisivali strukturu glavne funkcije `main()`. U prvim primjerima na stranicama 23 i 30 cijeli program sastojao se od po jednog bloka naredbi. Blokovi naredbi se redovito pišu uvučeno. To uvlačenje radi se isključivo radi preglednosti, što je naročito važno ako imamo blokove ugniježđene jedan unutar drugog.

Važno svojstvo blokova jest da su varijable deklarirane u bloku vidljive samo unutar njega. Zbog toga će prevođenje kôda

```
#include <iostream>
using namespace std;
```

```
int main()
{
    {                               // početak bloka naredbi
        int a = 1;                  // lokalna varijabla u bloku
        cout << a << endl;
    }                               // kraj bloka naredbi
    return 0;
}
```

proći uredno, ali ako naredbu za ispis lokalne varijable a prebacimo izvan bloka

```
#include <iostream>
using namespace std;

int main()
{
    {
        int a{1};
    }
    cout << a << endl;          // pogreška: a više ne postoji!
    return 0;
}
```

dobit ćemo poruku o pogreški da varijabla a koju pokušavamo ispisati ne postoji. Ovakvo ograničenje *područja* lokalne varijable (engl. *scope* – domet, doseg) omogućava da se unutar blokova deklariraju varijable s istim imenom kakvo je već upotrijebljeno izvan bloka. Lokalna varijabla će unutar bloka zakloniti istoimenu varijablu prethodno deklariranu izvan bloka, u što se najbolje možemo uvjeriti sljedećim primjerom

```
#include <iostream>
using namespace std;

int main()
{
    int a{5};
    {
        int a{1};
        cout << a << endl;      // ispisuje 1
    }
    cout << a << endl;          // ispisuje 5
    return 0;
}
```

Prva naredba za ispis dohvatit će lokalnu varijablu `a = 1`, budući da ona ima prednost pred istoimenom varijablom `a = 5` koja je deklarirana ispred bloka. Po izlasku iz bloka, lokalna varijabla `a` se gubi te je opet dostupna samo varijabla `a = 5`. Naravno da bi ponovna deklaracija istoimene varijable bilo u vanjskom, bilo u unutarnjem bloku rezultirala pogreškom tijekom prevođenja. Područjem dosega varijable pozabavit ćemo se detaljnije u kasnijim poglavljima.

Ako se blok u naredbama za kontrolu toka sastoji samo od jedne naredbe, tada se vitičaste zagrade mogu i izostaviti.

4.2. Naredbe za grananje toka

Naredbe za grananje toka omogućavaju da se, ovisno o nekom uvjetu, izvode različiti sljedovi naredbi. Jezik C++ definira dvije naredbe za grananje: `if` i `switch`. One su međusobno dosta slične i svako grananje naredbom `switch` može se jednostavno prepisati korištenjem naredbe `if` (ali obrnuto ne vrijedi). Ipak, među njima postoje razlike koje čine odabir jedne ili druge važnim za preglednost izvornog i efikasnost izvedbenog kôda.

4.2.1. Naredba `if`

Naredba `if` omogućava uvjetno grananje toka programa ovisno o tome da li je ili nije zadovoljen uvjet naveden iza ključne riječi `if`. Najjednostavniji oblik naredbe za uvjetno grananje je:

```
if ( logički_izraz )
    // blok_naredbi
```

Ako je vrijednost izraza iza riječi `if` logička istina (`true`), izvodi se blok naredbi koje slijede iza izraza. U protivnom se taj blok preskače i izvođenje nastavlja od prve naredbe iza bloka. Na primjer:

```
if (a < 0)
{
    cout << "Broj a je negativan!" << endl;
}
```

U slučaju da blok sadrži samo jednu naredbu, vitičaste zagrade koje omeđuju blok mogu se i izostaviti, pa smo gornji primjer mogli pisati i kao:

```
if (a < 0)
    cout << "Broj a je negativan!" << endl;
```

ili

```
if (a < 0) cout << "Broj a je negativan!" << endl;
```

Zbog preglednosti kôda i nedoumica koje mogu nastati prepravkama, početniku preporučujemo redovitu uporabu vitičastih zagrada i pisanje naredbi u novom retku.

U protivnom se može dogoditi da nakon dodavanja naredbe u blok programer zaboravi omeđiti blok zagradama i time dobije nepredviđene rezultate:

```
if (a < 0)
    cout << "Broj a je negativan!" << endl;
    cout << "Njegova apsolutna vrijednost je " << -a
         << endl;
```

Druga naredba ispod `if` uvjeta izvest će se i za pozitivne brojeve, jer više nije u bloku, pa će se za pozitivne brojeve ispisati pogrešna apsolutna vrijednost! Inače, u primjerima ćemo izbjegavati pisanje vitičastih zagrada gdje god je to moguće radi uštede na prostoru.

Želimo li da se ovisno u rezultatu izraza u `if` uvjetu izvode dva odvojena programska odsječka, primijenit ćemo sljedeći oblik uvjetnog grananja:

```
if ( logički_izraz )
    // prvi_blok_naredbi
else
    // drugi_blok_naredbi
```

Kod ovog oblika, ako izraz u `if` uvjetu daje kao rezultat logičku istinu (`true`), izvest će se prvi blok naredbi. Po završetku bloka, izvođenje programa nastavlja se od prve naredbe iza drugog bloka. Ako izraz daje kao rezultat logičku neistinu (`false`), preskače se prvi blok, a izvodi samo drugi blok naredbi, nakon čega se nastavlja izvođenje naredbi koje slijede. Evo jednostavnog primjera u kojem se računaju presjecišta pravca s koordinatnim osima. Pravac je zadan jednadžbom $a \cdot x + b \cdot y + c = 0$.

```
#include <iostream>
using namespace std;

int main()
{
    double a, b, c;                     // koeficijenti pravca
    cout << "Upiši koeficijente pravca:" << endl;
    cout << "a = ";
    cin >> a;                           // učitaj koeficijent a
    cout << "b = ";
    cin >> b;                           // učitaj koeficijent b
    cout << "c = ";
    cin >> c;                           // učitaj koeficijent c
    cout << "Koeficijenti: " << a << "," << b << "," << c << endl;
                                        // ispiši koeficijente

    cout << "Presjecište s apscisom: ";
    if (a != 0)
        cout << -c / a << ", ";
    else
        cout << "nema, ";               // pravac je horizontalan
```

```
    cout << "presjecište s ordinatom: ";
    if (b != 0)
        cout << -c / b << endl;
    else
        cout << "nema" << endl;         // pravac je vertikalan

    return 0;
}
```

Kada prevedete i pokrenete gornji program, ne zaboravite nakon upisa svake vrijednosti pritisnuti tipku *Enter*.

Zadatak: *Napišite program kojim ćete (pomoću operatora modulo) ispitati da li je upisani broj paran ili neparan i shodno tome ispisati poruku.*

Blokovi `if` naredbi se mogu nadovezivati:

```
if ( logički_izraz1 )
    // prvi_blok_naredbi
else if ( logički_izraz2 )
    // drugi_blok_naredbi
else if ( logički_izraz3 )
    // treći_blok_naredbi
...
else
    // zadnji_blok_naredbi
```

Ako je *logički_izraz1* točan, izvest će se prvi blok naredbi, a zatim se izvođenje nastavlja od prve naredbe iza zadnjeg `else` bloka u nizu, tj. iza bloka *zadnji_blok_naredbi*. Ako *logički_izraz1* nije točan, izračunava se *logički_izraz2* i ovisno o njegovoj vrijednosti izvodi se *drugi_blok_naredbi*, ili se program nastavlja iza njega. Ilustrirajmo to primjerom u kojem tražimo realne korijene kvadratne jednadžbe:

```
#include <iostream>
using namespace std;

int main()
{
    double a, b, c;
    cout << "Unesi koeficijente kvadratne jednadžbe:" << endl;
    cout << "a = ";
    cin >> a;
    cout << "b = ";
    cin >> b;
    cout << "c = ";
    cin >> c;

    auto diskriminanta = b * b - 4 * a * c;
```

```
    cout << "Jednadžba ima ";
    if (diskriminanta == 0)
        cout << "dvostruki realni korijen." << endl;
    else if (diskriminanta > 0)
        cout << "dva realna korijena." << endl;
    else
        cout << "dva kompleksna korijena." << endl;

    return 0;
}
```

Blokovi `if` naredbi mogu se ugnježđivati jedan unutar drugoga. Ilustrirajmo to primjerom u kojem gornji kôd poopćujemo i na slučajeve kada je koeficijent *a* kvadratne jednadžbe jednak nuli, tj. kada se kvadratna jednadžba svodi na linearnu jednadžbu:

```
#include <iostream>
using namespace std;

int main()
{
    double a, b, c;
    cout << "Unesi koeficijente kvadratne jednadzbe:" << endl;
    cout << "a = ";
    cin >> a;
    cout << "b = ";
    cin >> b;
    cout << "c = ";
    cin >> c;

    if (a)
    {
        auto diskriminanta = b * b - 4 * a * c;
        cout << "Jednadžba ima ";
        if (diskriminanta == 0)
            cout << "dvostruki realni korijen." << endl;
        else if (diskriminanta > 0)
            cout << "dva realna korijena." << endl;
        else
            cout << "kompleksne korijene." << endl;
    }
    else
        cout << "Jednadžba je linearna." << endl;

    return 0;
}
```

Za logički izraz u prvom `if` uvjetu postavili smo samo vrijednost varijable `a` – ako će ona biti različita od nule, uvjet će biti zadovoljen jer će se pravilima pretvorbe vrijednost varijable svesti na logičku istinu pa će se izvesti naredbe u prvom `if` bloku. Taj blok sastoji se od niza `if-else` blokova identičnih onima iz prethodnog primjera. Ako

početni uvjet nije zadovoljen, tj. ako varijabla `a` ima vrijednost 0 (što rezultira logičkom neistinom nakon pretvorbe), preskače se cijeli prvi blok i ispisuje poruka da je jednadžba linearna. Uočimo u gornjem primjeru dodatno uvlačenje ugniježđenih blokova.

Primijetimo kako je u prethodnim primjerima varijabla `diskriminanta` deklarirana ključnom riječi `auto`. U navedenim slučajevima će ona u stvari biti tipa `double`, budući da su tog tipa deklarirane varijable `a`, `b` i `c` koje se pojavljuju u izrazu kojim inicijaliziramo varijablu `diskriminanta`. Korištenje deklaracije `auto` osigurava da će prevoditelj automatski ažurirati i njen tip ako promijenimo tip varijabli `a`, `b` i `c`.

Zadatak: *Proširite prethodni zadatak za provjeru parnosti tako da za neparne brojeve dodatno ispitate djeljivost s 3 – ako je broj djeljiv s 3, ispišite dodatnu poruku o tome.*

Pri definiranju logičkog izraza u naredbama za kontrolu toka početnik treba biti oprezan. Na primjer, želimo li da se dio programa izvodi samo za određeni opseg vrijednosti varijable `b`, naredba

```
if (-10 < b < 0) //...
```

neće raditi onako kako bi se prema ustaljenim matematičkim pravilima očekivalo. Ovaj logički izraz u biti se sastoji od dvije usporedbe: prvo se ispituje je li –10 manji od `b`, a potom se rezultat te usporedbe uspoređuje s 0, tj.

```
if ((-10 < b) < 0) //...
```

Kako rezultat prve usporedbe može biti samo `true` ili `false`, što se pretvara u 1 ili 0, druga usporedba dat će uvijek logičku neistinu. Da bi program poprimio željeni tok, usporedbe moramo razdvojiti i logički izraz formulirati ovako: „ako je –10 manji od `b` ***i*** ako je `b` manji od 0”:

```
if (-10 < b && b < 0) //...
```

Druga nezgoda koja se može dogoditi jest da se umjesto operatora za usporedbu `==`, u logičkom izrazu napiše operator pridruživanja `=`. Na primjer:

```
if (k = 0)      // pridruživanje, a ne usporedba!!!
    ++k;
else
    k = 0;
```

Umjesto da se varijabla `k` uspoređuje s nulom, njoj se u logičkom izrazu pridjeljuje vrijednost 0. Rezultat logičkog izraza jednak je vrijednosti varijable `k` (0 odnosno `false`), tako da se prvi blok naredbi nikada ne izvodi. Bolji prevoditelj će na takvom mjestu korisniku prilikom prevođenja dojaviti upozorenje.

Neki programeri se s ovim problemom nose tako da u usporedbama s konstantom, konstantu navode s lijeve strane izraza usporedbe:

```
if (0 == k)     // "sigurniji" način usporedbe
    // ...
```

Nađe li u ovakvoj naredbi operator pridruživanja (=) umjesto operatora jednakosti (==), prevoditelj će javiti pogrešku da se konstanti ne može mijenjati vrijednost:

```
if (0 = k)      // pogreška: konstanti se ne može mijenjati vrijednost!
    // ...
```

Prilikom usporedbe decimalnih brojeva treba voditi računa da se oni pohranjuju u memoriju u binarnom obliku, koristeći konačni broj znamenaka. Pri pretvorbi u binarni oblik te u međusobnim operacijama redovito nastaje pogreška na najnižim znamenkama, što može uzrokovati da usporedbom rezultata koji bi inače po svim pravilima matematike bili jednaki, dobijemo da su oni različiti. Probate li sljedeći primjer:

```
double a{0.3};
double b = a * 10 - 2.7;
if (a == b)
    cout << a << " je jednako " << b << endl;
else
    cout << a << " nije jednako " << b << endl;
```

ne biste se trebali iznenaditi ako se ispiše:

```
0.3 nije jednako 0.3
```

Iako i naše skromno poznavanje matematike, kao i ispisane vrijednosti tvrde da `a` i `b` imaju jednake vrijednosti, ispisana poruka ukazuje na suprotno. Operator usporedbe je baratao sa sadržajem memorije i uočio razliku u sadržaju varijabli na njihovim najnižim, znamenkama. S druge strane, operator `cout` je prilikom ispisa zaokružio brojeve, tako da se razlika koja postoji u najnižim znamenkama izgubila.

Ovakvi problemi se obično rješavaju tako da se, umjesto izravne usporedbe decimalnih brojeva, provjerava je li njihova razlika po apsolutnoj vrijednosti manja od nekog dovoljno malog broja. Za prethodni primjer to znači da smo usporedbu mogli napisati na sljedeći način:

```
if (fabs(a - b) < 1e-4)
```

pri čemu je `fabs` matematička funkcija za računanje apsolutne vrijednosti decimalnih brojeva.

Naredba `if` može umjesto logičkog izraza sadržavati deklaraciju i inicijalizaciju neke varijable. U tom slučaju se uvjet za izvođenje pripadajućeg bloka naredbi svodi na provjeru je li varijabla inicijalizirana na vrijednost različitu od nule, a za logičke tipove različito od `false`. Pogledajmo to u prerađenom primjeru sa stranice 106 (navodimo samo promijenjeni dio kôda):

```
// ...
if (a)
{
    cout << "Jednadžba ima ";
    // varijablu deklariramo i inicijaliziramo u naredbi if:
    if (auto diskriminanta = b * b - 4 * a * c)
    {
        if (diskriminanta > 0)
            cout << "dva realna korijena." << endl;
        else
            cout << "kompleksne korijene." << endl;
    }
    else
        cout << "dvostruki realni korijen." << endl;
}
// ...
```

Nakon inicijalizacije varijable `diskriminanta`, u promijenjenoj naredbi `if` se provjerava je li njena vrijednost različita od nule – u tom slučaju izvodi se najugniježđeniji blok `if-else`, a u protivnom se ispisuje da jednadžba ima dvostruki realni korijen.

Prilikom deklaracije varijable u naredbi `if` treba paziti da ne uletimo u iskušenje dodati usporedbu, pretpostavljajući kako će operator usporedbe djelovati na inicijaliziranu varijablu i da će rezultat te usporedbe naredbi `if` poslužiti za odabir smjera kojim će se izvođenje programa nastaviti. Dodamo li usporedbu u naredbu `if` iz prethodnog primjera:

```
// ...
// oprez: diskriminanta će biti tipa bool!
if (auto diskriminanta = b * b - 4 * a * c != 0)
// ...
```

varijabla `diskriminanta` više neće sadržavati vrijednost izraza $b^2 - 4\,a\,c$, već rezultat usporedbe tog izraza s nulom. Rezultat usporedbe je logička vrijednost tipa `bool` pa će i varijabla `diskriminanta` biti tog tipa. Pokušamo li ovo zaobići dodavanjem zagrada:

```
// ...
// oprez: diskriminanta će biti tipa bool!
if ((auto diskriminanta = b * b - 4 * a * c) != 0)
// ...
```

prevoditelj će javiti pogrešku, jer sada izraz u naredbi `if` više nije deklaracija. U deklaraciji mora tip kojim se deklarira započinjati izraz, a u ovom slučaju mu prethodi otvarajuća zagrada.

Dodajmo na kraju da je unutar naredbe `if` moguće deklarirati samo jednu varijablu. Ovo ograničenje je prilično logično, jer u protivnom ne bi bilo jasno koji bi trebao biti ishod naredbe `if` prilikom inicijalizacije dviju varijabli različitih vrijednosti. Unatoč ograničenim mogućnostima naredbe `if` koja umjesto logičkog izraza sadrži deklaraciju,

ona ima jednu značajnu prednost: deklarirana varijabla ima doseg samo unutar pripadajućih blokova `if` i `else`, što smanjuje opasnost nekontrolirane promjene njene vrijednosti.

4.2.2. Uvjetni operator ? :

Iako ne spada među naredbe za kontrolu toka programa, uvjetni operator po strukturi je sličan `if-else` bloku, tako da ga je zgodno upravo na ovom mjestu predstaviti. Sintaksa operatora uvjetnog pridruživanja je:

```
uvjet ? izraz1 : izraz2 ;
```

Ako izraz *uvjet* daje logičku istinu, izračunava se *izraz1*, a u protivnom *izraz2*. U primjeru

```
x = (x < 0) ? -x : x;        // x = fabs(x)
```

ako je `x` negativan, izračunava se prvi izraz, te se varijabli `x` na lijevoj strani znaka jednakosti pridružuje njegova pozitivna vrijednost, tj. varijabla `x` mijenja svoj predznak. Naprotiv, ako je `x` pozitivan, tada se izračunava drugi izraz i varijabli `x` na lijevoj strani pridružuje njegova nepromijenjena vrijednost.

Izraz u uvjetu mora kao rezultat vraćati logički tip ili tip koji se može svesti na `bool`. Zato je potpuno suvišno pisati naredbe oblika:

```
bool veceOdPet = (x > 5) ? true : false;
```

budući da sam uvjet kao rezultat daje logičku vrijednost tipa `bool`.

Alternativni izrazi desno od znaka upitnika moraju davati rezultat međusobno istog tipa ili se moraju dati svesti na isti tip preko ugrađenih pravila pretvorbe.

Uvjetni operator koristite samo za jednostavna ispitivanja kada naredba stane u jednu liniju. U protivnom kôd postaje nepregledan.

Koliko čitatelja odmah shvaća da u sljedećem primjeru zapravo računamo korijene kvadratne jednadžbe?

```
((diskr = b * b -4 * a * c) >= 0) ?
    (x1 = (-b + diskr) / 2 / a, x2 = (-b - diskr) / 2 / a)  :
    (cout << "Ne valja ti korijen!", x1 = x2 = 0);
```

Zadatak: *Koristeći uvjetni operator, ispitajte parnost unesenog broja i ako je broj neparan povećajte mu vrijednost za 1.*

4.2.3. Naredba `switch`

Kada izraz uvjeta daje više različitih cjelobrojnih rezultata, a za svaki od njih treba provesti različite odsječke programa, tada je umjesto `if` grananja često preglednije koristiti `switch` grananje. Kod tog grananja se prvo izračunava neki izraz koji daje cjelobrojni rezultat. Ovisno o tom rezultatu, tok programa se preusmjerava na neku od grana unutar `switch` bloka naredbi. Općenita sintaksa `switch` grananja izgleda ovako:

```
switch ( cjelobrojni_izraz )
{
case konstantan_izraz1 :
    // prvi_blok_naredbi
case konstantan_izraz2 :
    // drugi_blok_naredbi
    break;
case konstantan_izraz3 :
case konstantan_izraz4 :
    // treći_blok_naredbi
    break;
default:
    // četvrti_blok_naredbi
    break;
}
```

Prvo se izračunava *`cjelobrojni_izraz`*, koji mora davati cjelobrojni rezultat. Ako je rezultat tog izraza jednak nekom od konstantnih izraza u `case` uvjetima, tada se izvode sve naredbe koje slijede pripadajući `case` uvjet sve do prve `break` naredbe. Nailaskom na `break` naredbu, izvođenje kôda u `switch` bloku se prekida i nastavlja se od prve naredbe iza `switch` bloka. Ako `izraz` daje rezultat koji nije naveden niti u jednom od `case` uvjeta, tada se izvodi blok naredbi iza ključne riječi `default`. Razmotrimo tokove programa za sve moguće slučajeve u gornjem primjeru. Ako *`cjelobrojni_izraz`* kao rezultat daje:

- *`konstantan_izraz1`*, tada će se prvo izvesti *`prvi_blok_naredbi`*, a zatim *`drugi_blok_naredbi`*. Nailaskom na naredbu `break` prekida se izvođenje naredbi u `switch` bloku. Program iskače iz bloka i nastavlja od prve naredbe iza bloka.
- *`konstantan_izraz2`*, izvodi se *`drugi_blok_naredbi`*. Nailaskom na naredbu `break` prekida se izvođenje naredbi u `switch` bloku i program nastavlja od prve naredbe iza bloka.
- *`konstantan_izraz3`* ili *`konstantan_izraz4`* izvodi se *`treći_blok_naredbi`*. Naredbom `break` prekida se izvođenje naredbi u `switch` bloku i program nastavlja od prve naredbe iza bloka.
- Ako rezultat nije niti jedan od navedenih *`konstantnih_izraza`*, izvodi se *`četvrti_blok_naredbi`* iza `default` naredbe.

Evo i konkretnog primjera `switch` grananja u kojem za zadani datum ispisujemo pripadajući dan u tjednu prema Gregorijanskom kalendaru:

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Upiši datum u formatu DD MM GGGG:";
    int dan, mjesec, godina;
    cin >> dan >> mjesec >> godina;

    int datum = dan;
    int m = (mjesec + 10) % 12;                    // pomoćna varijabla
    // uočimo pretvorbu u cijeli broj:
    datum += static_cast<int>(2.6 * m - 0.2);
    int g = (mjesec >= 3) ? godina : godina - 1;  // pomoćna varijabla
    datum += 5 * (g % 4);
    datum += 4 * (g % 100);
    datum += 6 * (g % 400);

    cout << dan << "." << mjesec << "." << godina << ". pada u ";

    switch (datum % 7)
    {
    case 0:
        cout << "nedjelju";
        break;
    case 1:
        cout << "ponedjeljak";
        break;
    case 2:
        cout << "utorak";
        break;
    case 3:
        cout << "srijedu";
        break;
    case 4:
        cout << "četvrtak";
        break;
    case 5:
        cout << "petak";
        break;
    case 6:
        cout << "subotu";
        break;
    }
    cout << "." << endl;    // ispišimo zaključnu točku i novi redak
    return 0;
}
```

Upotrijebljen je Gaussov algoritam koji se zasniva na cjelobrojnim dijeljenjima, tako da sve varijable treba deklarirati kao cjelobrojne. Uočimo u gornjem kôdu operator dodjele tipa `static_cast<int>()`

```
/*...*/ static_cast<int>(2.6 * m - 0.2)  /*...*/
```

kojim se rezultat množenja i zbrajanja brojeva s pomičnim zarezom pretvara u cijeli broj, tj. odbacuju decimalna mjesta. U `switch` naredbi se rezultat prethodnih računa normira na neki od sedam dana u tjednu pomoću operatora `%` (*modulo*).

Zadatak: *Potražite na Wikipediji sličan algoritam za Julijanski kalendar te proširite program tako da za datume prije 4. 10. 1582. godine dane u tjednu računa po Julijanskom, a za datume poslije 15. 10. 1582. godine po Gregorijanskom kalendaru. Za provjeru: 4. 10. 1582. godine je po Julijanskom kalendaru bio četvrtak.*

U pravilu, svaki blok naredbi bi trebao biti zaključen naredbom `break` ili nekom drugom naredbom koja će prekinuti izvođenje naredbi u grani. Tako smo u prethodnom programu, umjesto naredbi `break` mogli staviti naredbe `return`, budući da se nakon grananja `switch` ionako izvođenje programa praktički završavalo:

```
    // ...
    switch (datum % 7)
    {
    case 0:
        cout << "nedjelju." << endl;
        return 0;
    case 1:
        cout << "ponedjeljak." << endl;
        return 0;
    case 2:
        cout << "utorak." << endl;
        return 0;
    //...
    case 6:
        cout << "subotu." << endl;
        return 0;
    }
}
```

Sada iza grananja `switch` više nema potrebe staviti naredbu `return` jer grane u `switch`-u pokrivaju sve moguće sljedove izvođenja programa i sve završavaju naredbom `return 0`.

Iako naredba `break` na kraju zadnje grane nije neophodna, dobra je navika staviti ju za slučaj da iza nje naknadno dodamo još neku granu.

Grananje `switch` može sadržavati i blok `default` čije se naredbe izvode ako niti jedan uvjet u ispitivanjima `case` nije zadovoljen. To znači da smo u našem programu za određivanje dana u tjednu mogli grananje `switch` napisati i na sljedeći način:

```
// ...
switch (datum % 7)
{
case 0:
    cout << "nedjelju";
    break;
case 1:
    cout << "ponedjeljak";
    break;
// ...
case 5:
    cout << "petak";
    break;
default:                    // umjesto case 6:
    cout << "subotu";
    break;
}
// ...
```

Međutim, blok `default` se najčešće koristi za provjeru ima li *`cjelobrojni_izraz`* u naredbi `switch` neku vrijednost koja nije pokrivena svim granama `case`. Prevoditelj će obično prijaviti pogrešku ako se uvjet grananja ponavlja, npr. da imamo dvije grane `case 1`. No, jedini način da provjerimo jesmo li izostavili neku granu jest da testiramo program za različite ulazne podatke i pazimo da program nije ušao u granu `default`. Na primjer, zaboravimo li napisati granu za ispis petka, za datum koji pada u petak (npr. 3.1.2014.) tijek izvođenja će završiti u grani `default` pa će se ispisati poruka o pogrešci:

```
// ...
switch (datum % 7)
{
case 0:
    cout << "nedjelju";
    break;
case 1:
    cout << "ponedjeljak";
    break;
case 2:
    cout << "utorak";
    break;
case 3:
    cout << "srijedu";
    break;
case 4:
    cout << "četvrtak";
    break;
// nedostaje case 5:
case 6:
    cout << "subotu";
    break;
```

```
    // provjera da nije promakla neka nepredviđena vrijednost:
    default:
        cerr << "OPA!!! Neki nepredviđeni dan" << endl;
        return 1;
    }
    // ...
```

Za ispis pogreške u bloku `default` koristi se izlazni tok `cerr`, deklariran u zaglavlju `iostream`. To je uobičajeni izlazni tok za pogreške, koji podrazumijevano ide na zaslon. Nakon ispisa pogreške slijedi naredba `return 1`, kojom završava izvođenje i signalizira se da program nije završio ispravno.

Blok `default` može biti smješten bilo gdje unutar grananja `switch`, no zbog preglednosti preporučljivo je uvijek ga smjestiti na kraj grananja.

Zadatak: *U gornjem primjeru pomaknite* `default` *blok naredbi ispred* `case` *naredbi i pokrenite program. Što bi se dogodilo da naredbu* `break` *na kraju tako premještenog bloka* `default` *izostavite?*

Zadatak: *Program za određivanje dana u tjednu proširite tako da dodatno ispituje pada li zadani datum na neki blagdan (za početak ispitajte samo za 1. siječnja). Ako pada na blagdan ili ako pada u dane vikenda, ispišite da je taj dan neradni, inače da je radni. Uputa: na početku programa definirajte* `bool` *varijablu* `jeLiPraznik` *i postavite ju početno na* `false`*. Vrijednost te varijable promijenite ako je zadovoljen uvjet ispitivanja za blagdane ili ako se unutar* `switch`*-a ispostavi da datum pada u subotu ili nedjelju. Na kraju programa ispitajte vrijednost varijable i ispišite odgovarajuću poruku.*

U dosadašnjim primjerima svaki `case` je bio zaključen naredbom `break` pa se netko s pravom može zapitati zašto je uopće neophodan `break` – zar ne bi bilo jednostavnije definirati da se program podrazumijevano izvršava samo do sljedeće naredbe `case`? Postoje dva slučaja kada nam takav podrazumijevani `break` ne bi odgovarao.

Ponekad želimo da se isti blok naredbi izvodi za više različitih vrijednosti izraza u naredbi `switch`. Modificirajmo naš program tako da ispisuje samo pada li datum u dane vikenda ili ne:

```
    switch (datum % 7)
    {
    case 0:
    case 6:
        cout << "dane weekenda";
        break;
    // case-ove možemo napisati i u istom retku:
    case 1: case 2: case 3: case 4: case 5:
        cout << "radne dane";
        break;
    default:
```

```
        cerr << "OPA!!! Neki nepredviđeni dan" << endl;
        return 1;
    }
```

Budući da odmah iza naredbe `case 0:` slijedi `case 6:`, blok naredbi do prve naredbe `break` će se izvršavati i za slučaj kada je rezultat izraza u `switch` jednak 0. Slično vrijedi za drugu grupu grananja `case`.

Drugi slučaj kada nam odgovara neobavezni smještaj naredbe `break` jest kada želimo da različite grane dijele dio kôda. Predstavimo to sljedećom, vrlo naivnom izvedbom programa koji odbrojava sekunde (radi kratkoće smo se ograničili na 5 sekundi – čitatelju prepuštamo da program po želji proširi):

```
#include <iostream>
#include <thread>       // zaglavlje za std::this_thread::sleep_for
#include <chrono>       // zaglavlje za std::chrono::seconds
using namespace std;
using namespace std::this_thread;
using namespace std::chrono;

int main()
{
    cout << "Upiši broj sekundi za odbrojavanje: ";
    int odbrojavanje;
    cin >> odbrojavanje;

    seconds sekunda{1};             // interval od jedne sekunde

    switch (odbrojavanje)
    {
    case 5:
        cout << "pet..." << endl;       // ispiši broj i...
        sleep_for(sekunda);             // napravi stanku od 1 sekunde
    case 4:
        cout << "četiri..." << endl;
        sleep_for(sekunda);
    case 3:
        cout << "tri..." << endl;
        sleep_for(sekunda);
    case 2:
        cout << "dva..." << endl;
        sleep_for(sekunda);
    case 1:
        cout << "jedan..." << endl;
        sleep_for(sekunda);
    case 0:
        break;
    default:
        cerr << "Nedozvoljeni interval!" << endl;
        return 1;
    }
```

```
    cout << "GOTOVO!!!" << endl;
    return 0;
}
```

Budući da naredbe `break` nema sve do zadnje grane (`case 0:`), nakon ulaska u jednu od grana, izvest će se naredbe u svim granama koje slijede te ispisati brojevi od zadanog broja do broja jedan.

U kôdu je korištena funkcija `sleep_for()` koja zaustavlja izvođenje programa za zadani interval vremena. Ona je deklarirana u standardiziranom zaglavlju `thread`, unutar imenika `std::this_thread`. Kao argument toj funkciji (unutar zagrada) proslijeđen je objekt `sekunda`:

```
    sleep_for(sekunda);             // napravi stanku od 1 sekunde
```

Objekt `sekunda` je tipa `seconds`, inicijaliziran brojem 1, tj. on predstavlja vremenski interval od jedne sekunde:

```
    seconds sekunda{1};                 // interval od jedne sekunde
```

Tip `seconds` je deklariran u zaglavlju `chrono`, a unutar imenika `std::chrono`.

Nadovezivanje naredbi u blokovima `case` koje je izvedeno u gornjem primjeru naziva se *propadanje* (engl. *fallthrough*). Budući da su naredbe u granama čvrsto vezane i redoslijed grana je od ključnog značaja za ispravnost programa, propadanje može izazvati nepredviđene probleme pri naknadnim promjenama grananja `switch`.

Propadanje u grananju `switch` treba izbjegavati gdje god je to moguće, posebice ako je broj grana jako velik.

Uostalom, prethodni primjer se mogao puno elegantnije riješiti korištenjem petlje koja bi pozivala funkciju za ispis broja. Ta funkcija bi sadržavala grananje `switch`, ali bez propadanja. Kada upoznamo petlje, napravit ćemo i tu izvedbu.

Ako se ipak pokaže da propadanje može pojednostavniti kôd, na mjestu propadanja dobro je staviti komentar kojim će autor dati do znanja da je namjerno izostavio naredbu `break`:

```
// definicija pobrojenja s mogućim naredbama
enum class Naredba
{
    SejvajPodDrugimImenom,
    Sejvaj,
    Klouzaj
};

// ...
Naredba naredba{Naredba::SejvajPodDrugimImenom};
```

```
switch (naredba)
{
case Naredba::SejvajPodDrugimImenom:
    PromijeniIme();         // neka funkcija koja mijenja ime datoteke
    // namjerno propadanje
case Naredba::Sejvaj:
    SejvajFajlu();          // neka funkcija koja pohranjuje datoteku
    break;
case Naredba::Klouzaj:
    KlouzajFajlu();         // neka funkcija koja oslobađa datoteku
    break;
default:
    cerr << "Nepoznata naredba!" << endl;
    break;
}
```

Jedan od problema na koji će početnik prije ili kasnije naletjeti pri korištenju grananja `switch` jest definiranje lokalne varijable unutar neke od grana:

```
    switch (uvjet)
    {
    case 1:
        int a{0};   // deklariramo i inicijaliziramo lokalnu varijablu
        // ...
        break;
    case 2:
        // ...
        break;
    }
```

U prvoj grani deklariramo i inicijaliziramo lokalnu varijablu a koja nam treba samo u toj grani. Prevoditelj će se međutim pobuniti da je inicijalizacija varijable a preskočena u drugoj grani (odnosno svim ostalim granama ako ih ima više). Naime, varijabla ima doseg u cijelom `switch` bloku, unutar vitičastih zagrada. Zato će ona biti vidljiva u svim granama, ali za ostale grane inicijalizacija nije provedena.

Jedno rješenje bi bilo da izostavimo inicijalizaciju, tj. razdvojimo na deklaraciju i posebnu naredbu kojom pridružujemo početnu vrijednost:

```
    switch (uvjet)
    {
    case 1:
        int a;      // samo deklaracija
        a = 0;      // u ovoj grani pridružujemo vrijednost -
                    // u ostalim granama nedefinirana vrijednost
        break;
    case 2:
        // ...
        break;
    }
```

U ostalim granama varijabla će biti vidljiva, ali će njena vrijednost biti nedefinirana, na što će neki prevoditelji javiti upozorenje. Zato je daleko popularnije rješenje cijelu granu ugnijezditi u novi blok:

```
switch (uvjet)
{
case 1:
    {                   // početak novog bloka
        int a = 0;
        // ...
    }                   // kraj novog bloka
    break;
case 2:
    // ...
    break;
}
```

i time ograničiti doseg lokalne varijable samo na pripadajuću granu.

Mnogi programeri zaziru od grananja `switch` (u nekim programskim jezicima ni ne postoji takva vrsta grananja), budući da se ekvivalentno grananje može napisati i pomoću niza `if-else` `if-else` uvjeta. Grananje `if` prvenstveno ima smisla koristiti kada se u uvjetu grananja radi relacijska usporedba oblika „veće”, „veće ili jednako”, „manje”, „različito”, koja zadovoljava neograničeni broj ulaznih podataka ili ako se provjerava jednakost s jednom vrijednošću. Međutim, kada je odabir grane koja će se izvesti određen vrijednostima iz malog skupa (kao što su dani u tjednu), kôd napisan grananjem `switch` je pregledniji. Osim toga, prevoditelji mogu iz njega napraviti brži izvedbeni kôd nego za slučaj kada se koristi niz `if-else` `if-else` uvjeta. Glavna zamka kod grananja `switch` je u nehotičnom izostavljanju naredbe `break`, što rezultira nekontroliranim propadanjem kroz pojedine grane.

4.3. Naredbe za ponavljanje

U programima često treba uzastopno ponavljati dijelove kôda, na primjer kada se neki postupak sastoji od naredbi koje treba ponoviti određeni broj puta ili sve dok je zadovoljen neki uvjet. Također, nerijetko treba istu operacija obaviti nad svakim članom niza podataka. Sklopovi naredbi koje se ponavljaju (*petlje*) mogu se zadati naredbama `for`, `while` i `do` – one će detaljno biti opisane u sljedećim odjeljcima.

4.3.1. Petlja `for`

Ako je broj ponavljanja poznat prije ulaska u petlju, najprikladnije je koristiti `for` petlju. To je najopćenitija vrsta petlje i ima sljedeći oblik:

```
for ( početni_izraz ; uvjet_izvođenja ; izraz_prirasta )
    // blok_naredbi
```

Iza ključne riječi `for`, unutar okruglih zagrada `()` navode se tri grupe naredbi, međusobno odvojenih znakom točka-zarez `;`. Postupak izvođenja `for`-bloka je sljedeći:

1. Izračunava se *`početni_izraz`*. Najčešće je to pridruživanje početne vrijednosti cjelobrojnom brojaču kojim će se kontrolirati ponavljanje petlje.
2. Izračunava se *`uvjet_izvođenja`*, izraz čiji rezultat mora biti tipa `bool`. Ako je rezultat jednak logičkoj neistini, preskače se *`blok_naredbi`* i program se nastavlja prvom naredbom iza bloka.
3. Ako je *`uvjet_izvođenja`* jednak logičkoj istini, izvodi se *`blok_naredbi`*.
4. Na kraju se izračunava *`izraz_prirasta`* (npr. povećavanje brojača petlje). Program se vraća na početak petlje, te se ona ponavlja od točke 2.

Programski odsječak se ponavlja sve dok *`uvjet_izvođenja`* na početku petlje daje logičku istinu; kada rezultat tog izraza postane logička neistina, programska petlja se prekida.

Kao primjer za `for` petlju, napišimo program za izračunavanje faktorijela prirodnog broja. Da se podsjetimo: faktorijela od n je umnožak svih cijelih brojeva od 1 do n:

$$n! = 1 \cdot 2 \cdot \ldots \cdot (n-2) \cdot (n-1) \cdot n$$

Pogledajmo početnu izvedbu kôda:

```
#include <iostream>
using namespace std;

int main()
{
    int n;
    cout << "Upiši prirodni broj: ";     // ne veći od 12(!?)
    cin >> n;

    long int fjel{1};
    for (auto i = 1; i <= n; ++i)
        fjel *= i;
    cout << n << "! = " << fjel << endl;
    return 0;
}
```

Prije ulaska u petlju trebamo definirati početnu vrijednost varijable `fjel` u koju ćemo gomilati umnoške. Tu početnu vrijednost postavljamo na 1 jer ćemo unutar petlje njome množiti rastući brojač petlje. Ako bismo ju postavili na 0, konačni rezultat bi uvijek bio 0. Na ulasku u petlju deklariramo brojač petlje, cjelobrojnu varijablu `i` te joj pridružujemo početnu vrijednost 1 (*`početni_izraz:`* `auto i = 1`). Unutar same petlje množimo `fjel` s brojačem petlje, a na kraju tijela petlje uvećavamo brojač (*`izraz_prirasta:`* `++i`). Petlju ponavljamo sve dok je brojač manji ili jednak unesenom broju (*`uvjet_izvođenja:`* `i <= n`).

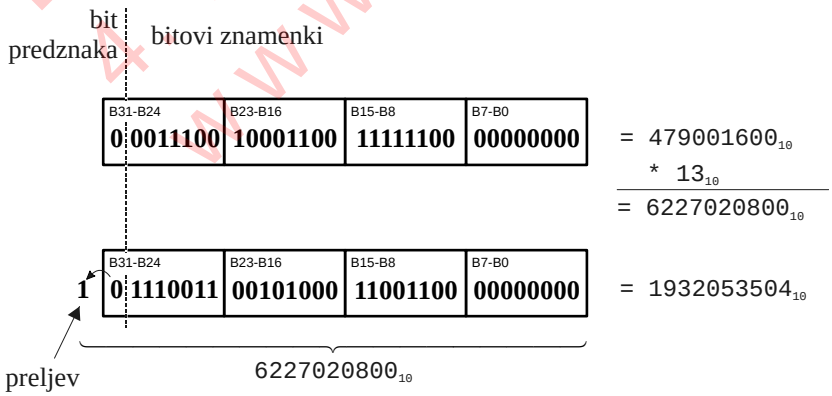
Uočimo kako program ima jedan banalnu manjkavost: petlja se početno izvodi za `i = 1` pa se u prvom prolazu petlje izvodi ne previše korisno množenje `fjel` s 1. Da to izbjegnemo, petlju možemo bez ikakvih posljedica početi od `i = 2`:

```
// petlju počinjemo od 2:
for (auto i = 2; i <= n; ++i)
    fjel *= i;
// ...
```

Što će se dogoditi ako za `n` unesemo brojeve manje od 2? Već pri prvom ulasku u petlju neće biti zadovoljen uvjet ponavljanja petlje (`i <= n`) te će odmah biti preskočeno tijelo petlje i ona se neće izvesti niti jednom! Varijabla `fjel` će prilikom završnog ispisa sadržavati početnu vrijednost 1, što nam u potpunosti odgovara jer su 1! = 1 i (po definiciji) 0! = 1.

Pri isprobavanju programa treba paziti da uneseni broj ne smije biti prevelik, jer će inače doći do brojčanog preljeva varijable `fjel` te će ispisani rezultat biti pogrešan. Pretpostavimo da je tip `long int` duljine 32 bita pa je 2.147.483.647 najveći broj koji on može sadržavati (vidi tablicu 3.4). Budući da je 12! = 479.001.600, zadamo li programu broj veći od 12, u prolazu petlje za `i = 13` imat ćemo množenje 479.001.600 · 13, čiji rezultat nadmašuje najveću vrijednost za koju smo pretpostavili da 32-bitna varijabla tipa `long int` može sadržavati. Množenje će prouzročiti brojčani preljev: varijabla `fjel` će sadržavati samo bitove koji stanu u njen memorijski prostor, dok će bitovi koji su se „prelili" bespovratno biti izgubljeni (slika 4.5). Tako okljaštreni bitovi umnoška, nakon pretvorbe u dekadski prikaz, daju broj koji nema nikakve veze s točnim rezultatom: 1.932.053.504.

Problem možemo odgoditi prema većim vrijednostima tako da varijablu `fjel` deklariramo da je tipa `long long int`. Međutim, korisnik će i dalje, bez ikakva upozorenja, dobivati pogrešne rezultate za prevelike zadane vrijednosti. Da bismo to izbjegli, u kôd ćemo dodati provjeru. Idealno bi bilo kada bismo unaprijed znali koju najveću vrijednost smije korisnik unijeti pa ga na to odmah upozorimo. Za sada ćemo provjeru raditi unutar petlje, prije množenja varijable `fjel` s brojačem. Ako je trenutna vrijednost varijable `fjel` veća od kvocijenta najvećeg broja koji stane u tip kojega je varijabla `fjel` i



Slika 4.5. Prikaz preljeva prilikom množenja

trenutne vrijednosti brojača, tada bi množenje `fjel` s brojačem uzrokovalo preljev. Zato ćemo u tom trenutku prekinuti izvođenje programa i ispisati poruku o pogrešci:

```
#include <iostream>
#include <limits>
using namespace std;

int main()
{
    int n;
    cout << "Upiši prirodni broj: ";
    cin >> n;

    long long fjel{ 1 };    // fjel je sada tipa long long int
    // najveći broj koji fjel može sadržavati:
    auto fjelMax = numeric_limits<decltype(fjel)>::max();

    for (auto i = 2; i <= n; ++i)
    {
        // prije množenja provjeravamo hoće li doći do preljeva:
        if (fjel > fjelMax / i)
        {
            cerr << "Prevelika ulazna vrijednost!" << endl;
            return 1;
        }
        fjel *= i;
    }
    cout << n << "! = " << fjel << endl;
    return 0;
}
```

Na kraju spomenimo još da smo petlju mogli napisati i tako da odbrojava od najvećeg broja (tj. unesenog broja `n`) do 2:

```
    // petlju počinjemo s n, odbrojavamo prema dolje, do 2:
    for (auto i = n; i > 1; --i)
        fjel *= i;
    // ...
```

Rezultati bi bili isti. Korištenjem petlje `while` (§4.3.2) program se može napisati još sažetije (vidi str. 128), no u ovom trenutku to nam nije bitno.

Može se dogoditi da je uvjet izvođenja petlje uvijek zadovoljen, pa će se petlja izvesti neograničeni broj puta. Program će uletjeti u slijepu ulicu iz koje nema izlaska, osim „ubijanjem” dotičnog procesa (na starijim operacijskim sustavima to je bilo moguće isključivo pomoću tipki *Power* ili *Reset* na kućištu računala). Na primjer:

```
// ovaj primjer pokrećete na vlastitu odgovornost!
cout << "Beskonačna petlja";
```

```
for (auto i = 0; i < 10; )
    cout << "a";
```

Budući da je brojač inicijaliziran na 0, a nigdje se unutar petlje njegova vrijednost ne mijenja, uvjet izvođenja petlje će biti ispunjen zauvijek. Ukoliko se usudite pokrenuti ovaj program, ekran će vam vrlo brzo biti preplavljen slovom `a`. Iako naizgled banalne, ovakve pogreške mogu početniku zadati velike glavobolje.

Uočimo da smo u gornjem kôdu izostavili *`izraz_prirasta`*. Općenito, može se izostaviti bilo koji od tri izraza u `for` naredbi – jedino su oba znaka `;` obavezna.

Izostavi li se *`uvjet_izvođenja`*, podrazumijevana vrijednost će biti `true` i petlja će biti beskonačna!

Štoviše, mogu se izostaviti i sva tri izraza:

```
for ( ; ; )     // opet beskonačna petlja!
```

ali čitatelju prepuštamo da sam zaključi koliko je to smisleno.

Koristan savjet (ali ne apsolutno pravilo) je izbjegavati mijenjanje vrijednosti kontrolne varijable unutar bloka naredbi `for` petlje. Sve njene promjene najsigurnije je zadavati isključivo u izrazu prirasta.

U protivnom se lako može dogoditi da petlja postane beskonačna, poput sljedećeg primjera:

```
for (auto i = 0; i < 5; ++i)
    --i;     // opet beskonačna petlja!
```

Naredba unutar petlje potpuno potire izraz prirasta, te varijabla `i` alternira između –1 i 0.

Početni izraz i izraz prirasta mogu se sastojati i od više izraza odvojenih operatorom nabrajanja `,` (zarez) . To nam omogućava da program za računanje faktorijela napišemo i (nešto) kraće:

```
#include <iostream>
using namespace std;

int main()
{
    int n;
    cout << "Upiši prirodni broj: ";
    cin >> n;

    long long fjel;
    int i;
```

```
    for (i = 2, fjel = 1; i <= n; fjel *= i, ++i)
        ;  // prazna naredba

    cout << n << "! = " << fjel;
    return 0;
}
```

U početnom izrazu postavljaju se brojač i varijabla `fjel` na svoje inicijalne vrijednosti. Naredba za množenje s brojačem prebačena je iz bloka naredbi u izraz prirasta, tako da je od bloka ostala prazna naredba, tj. sam znak `;`. Njega ne smijemo izostaviti, jer bi inače prevoditelj prvu sljedeću naredbu (a to je naredba za ispis konačnog rezultata) obuhvatio u petlju.

Netko će vjerojatno zapitati zašto smo morali izbacili deklaracije obje varijable ispred petlje; zar ne bi bilo dovoljno izbaciti samo `fjel`:

```
    long long fjel;
    // pozor: deklaracija nove varijable fjel
    for (auto i = 2, fjel = 1; i <= n; fjel *= i, ++i) ;
```

Pokušate li izvesti kôd s ovakvim izmjenama, program će ispisati pogrešan rezultat, iako ništa nije mijenjano u logici programa! Budući da su inicijalizacije u početnom izrazu `for` petlje odvojene samo zarezom, prevoditelj će shvatiti da treba deklarirati ne samo brojač `i`, već i (novu) varijablu `fjel`. Unutar petlje sve operacije se obavljaju nad tim lokalno deklariranim varijablama, a izlaskom iz petlje te varijable nestaju, jer izlazimo iz bloka unutar kojeg su deklarirane. Naredba za ispis dohvaćat će varijablu `fjel` tipa `long long` koja je deklarirana ispred petlje, a kako je ta varijabla ostala neinicijalizirana, ispis će biti nepredvidiv. Većina prevoditelja će javiti upozorenje da se u naredbi za ispis koristi neinicijalizirana varijabla `fjel` (neki prevoditelji će to prijaviti kao pogrešku), no početniku najvjerojatnije neće biti jasno o čemu se radi i zanemarit će to upozorenje.

Pokušaj zamjene naredbi u početnom izrazu:

```
    for (fjel = 1, auto i = 2; i <= n; fjel *= i, ++i) ;
```

prouzročit će pogrešku pri prevođenju, budući da se tip smije deklarirati samo na početku naredbe.

Ako `for` naredba sadrži deklaraciju varijable, tada se područje te varijable prostire samo do kraja petlje.

U ovo se možemo uvjeriti pokušamo li prevesti sljedeći program:

```
#include <iostream>
using namespace std;

int main()
```

```
{
    for (auto i = 1; i <= 10; ++i)
        cout << i << endl;
    cout << i << endl;      // pogreška: nedeklarirana varijabla
    return 0;
}
```

Ako je usklađen sa standardom, prevoditelj će javiti da u naredbi za ispis koristimo nedeklariranu varijablu `i`. Ona je deklarirana unutar petlje i izvan nje nije dohvatljiva.

Zadatak: *Napišite program s dvije uzastopne petlje koje će ispisati tablicu sa slovima engleskog alfabeta. Tablica neka ima dva stupca: u prvom stupcu neka je ASCII kôd (od 65 do 90 uključivo za velika slova, odnosno 97 do 122 uključivo za mala slova). Uputa: slovo ispišite koristeći dodjelu tipa* `static_cast<char>`*. Potom preuredite program tako da imate samo jednu petlju i ispis u četiri stupca.*

Petlje `for` mogu biti ugniježđene jedna unutar druge. Ponašanje takvih petlji razmotrit ćemo na sljedećem programu:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    for (auto redak = 1; redak <= 10; ++redak)
    {
        for (auto stupac = 1; stupac <= 10; ++stupac)
            cout << setw(5) << redak * stupac;
        cout << endl;
    }
    return 0;
}
```

Nakon prevođenja i pokretanja programa, na zaslonu će se ispisati već pomalo zaboravljena, ali generacijama pučkoškolaca omražena tablica množenja do 10:

```
    1    2    3    4    5    6    7    8    9   10
    2    4    6    8   10   12   14   16   18   20
    3    6    9   12   15   18   21   24   27   30
    4    8   12   16   20   24   28   32   36   40
    5   10   15   20   25   30   35   40   45   50
    6   12   18   24   30   36   42   48   54   60
    7   14   21   28   35   42   49   56   63   70
    8   16   24   32   40   48   56   64   72   80
    9   18   27   36   45   54   63   72   81   90
   10   20   30   40   50   60   70   80   90  100
```

Kako se gornji program izvodi? Pri ulasku u prvu (vanjsku) petlju, inicijalizira se brojač redaka na vrijednost 1 te se s njom ulazi u drugu, unutarnju petlju. U unutarnjoj petlji se brojač stupaca mijenja od 1 do 10 i za svaku pojedinu vrijednost izračunava se njegov

umnožak s brojačem redaka (potonji je cijelo to vrijeme `redak = 1`). Po završenoj unutarnjoj petlji ispisuje se znak za novi redak, čime završava blok naredbi vanjske petlje. Slijedi prirast brojača redaka (`redak = 2`) i povrat na početak vanjske petlje. Unutarnja petlja se ponavlja od `stupac = 1` do `stupac = 10` itd. Kao što vidimo, za svaku vrijednost brojača vanjske petlje izvodi se cjelokupna unutarnja petlja.

Da bismo dobili ispis brojeva u pravilnim stupcima, u gornjem primjeru rabili smo manipulator `setw()`. Argument tog manipulatora (tj. cijeli broj u zagradi) određuje koliki će se najmanji prostor predvidjeti za ispis podatka koji slijedi u izlaznom toku. Ako je podatak kraći od predviđenog prostora, preostala mjesta bit će popunjena praznina-ma. Manipulator `setw()` definiran je u datoteci zaglavlja `iomanip`.

Zadatak: *Preuredite prethodni primjer tako da ispiše samo članove u donjem lijevom trokutu:*

```
1
2    4
3    6    9
4    8   12   16
...
```

Uputa: u unutarnjoj petlji kao uvjet izvođenja postavite da je stupac manji ili jednak retku.

Zadatak: *Kôd iz prethodnog zadatka modificirajte tako da se ispišu samo članovi u gornjem desnom trokutu. Uputa: unutar vanjske petlje smjestite dvije petlje po stupcima, s time da prva treba ispisivati odgovarajući broj praznina, a druga tražene brojeve.*

4.3.2. Naredba `while`

Druga od tri petlje kojima jezik C++ raspolaže jest `while` petlja. Ona se koristi uglavnom za ponavljanje segmenta kôda kod kojeg broj ponavljanja nije unaprijed poznat, već se naredbe ponavljaju sve dok je zadovoljen logički izraz čiji se rezultat može promijeniti tijekom izvođenja petlje. Sintaksa `while` bloka je

```
while ( uvjet_izvođenja )
    // blok_naredbi
```

`uvjet_izvođenja` je izraz čiji je rezultat tipa `bool`. Tok izvođenja `while`-bloka je sljedeći:

1. Izračunava se logički izraz *`uvjet_izvođenja`*.
2. Ako je rezultat jednak logičkoj neistini, preskače se *`blok_naredbi`* i program se nastavlja od prve naredbe iz bloka.
3. Ako je *`uvjet_izvođenja`* jednak logičkoj istini izvodi se *`blok_naredbi`*. Potom se program vraća na `while` naredbu i izvodi od točke 1.

Konkretnu primjenu `while` bloka dat ćemo programom kojim se ispisuje sadržaj datoteke s brojevima. U donjem kôdu je to datoteka `brojevi.txt`, ali uz promjene odgovarajućeg imena, to može biti i neka druga datoteka.

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    ifstream ulazniTok{"brojevi.txt"};
    cout << "Sadržaj datoteke:" << endl << endl;
    double broj;
    while (ulazniTok >> broj)
        cout << broj << endl;
    return 0;
}
```

Brojevi u datoteci `brojevi.txt` moraju biti upisani u tekstovnom obliku, razdvojeni prazninama, tabulatorima ili napisani u zasebnim recima (možemo ih upisati pomoću najjednostavnijeg programa za upis teksta te ih pohraniti na disk). Da program tijekom izvođenja ne bi imao problema s pronalaženjem datoteke na disku, možete navesti puno ime do datoteke, primjerice:

```
    ifstream ulazniTok("c:\\temp\\brojevi.txt");
```

Ne zaboravite pri tome da u znakovnom nizu treba lijevu kosu crtu \ prikazati dvostrukim znakom, jer sama lijeva kosa crta služi za posebne znakove (§3.4.10).

Na početku programa se stvara objekt `ulTok` tipa `ifstream`. `ifstream` je standardizirana klasa definirana u zaglavlju `fstream`, namijenjena za učitavanje podataka iz datoteka. Za sada je dovoljno reći da je objekt te klase (u našem programu je to objekt `ulTok`) sličan ulaznom toku `cin` kojeg smo do sada koristili za unos podataka s tipkovnice. Osnovna razlika je u tome što je `cin` povezan na tipkovnicu, dok se `ulTok` veže za datoteku čiji je naziv zadan u dvostrukim navodnicima u zagradama[1].

Usredotočimo se na blok `while` naredbe. Na početku `while`-petlje, u uvjetu izvođenja bloka se naredbom:

```
ulTok >> broj
```

čita podatak. Ako je učitavanje broja bilo uspješno, `ulTok` će biti različit od nule (neovisno o vrijednosti učitanog broja). Shodno tome, izraz uvjeta izvođenja daje logičku istinu pa se izvodi blok naredbi u `while`-petlji – broj se ispisuje na izlaznom toku `cout`. Nakon što se izvede naredba iz bloka, izvođenje se vraća na naredbu `while` u kojoj se ponovno izračunava uvjet izvođenja petlje, tj. izvodi naredba kojom se učitava sljedeći broj. Ako nema više brojeva u datoteci, `ulTok` će poprimiti vrijednost 0. Uvjet ponavlja-

[1] Ulazni i izlazni tokovi te razred `fstream` detaljno su obrađeni u 21. poglavlju.

nja petlje više nije zadovoljen i petlja se prekida: blok petlje se preskače, a izvođenje se nastavlja prvom naredbom iza bloka.

Primijetimo zgodno svojstvo petlje `while`: ako je datoteka prazna, ulazni tok će odmah poprimiti vrijednost 0 te će uvjet odmah prilikom prvog testiranja biti neispunjen. Blok petlje se u tom slučaju neće izvesti niti jednom, što je primjereno našem slučaju.

Zadatak: *Dopunite gornji program tako da nakon sadržaja datoteke ispiše i broj iščitanih brojeva i njihovu srednju vrijednost.*

Zadatak: *Promijenite program tako da čita i ispisuje sadržaj tekstovne datoteke, znak po znak. Uputa: umjesto* `double broj`*, deklarirajte* `char znak`*, a u naredbi* `while` *znak čitajte naredbom* `ulazniTok.get(znak)`*.*

Zanimljivo je uočiti da nema suštinske razlike između `for`- i `while`-bloka naredbi – svaki `for`-blok se uz neznatne preinake može napisati kao `while`-blok i obrnuto. Da bismo se u to osvjedočili, napisat ćemo program za računanje faktorijela iz prethodnog poglavlja korištenjem `while` naredbe:

```
#include <iostream>
using namespace std;

int main()
{
    int n;
    cout << "Upiši prirodni broj: ";
    cin >> n;
    long long fjel{1};
    auto i = 2;
    while (i <= n)
    {
        fjel *= i;
        ++i;
    }
    cout << n << "! = " << fjel << endl;
    return 0;
}
```

Trebalo je samo početni izraz (`auto i = 2`) izlučiti ispred `while` naredbe, a izraz prirasta (`++i`) prebaciti u blok naredbi. Ipak, deklaracijom varijable `i` izvan petlje, proširili smo njen doseg izvan područja u kojem nam je neophodna.

Primijetimo da u gornjem primjeru varijablu `n` nismo dirali da bismo ju mogli ispisati u zadnjoj naredbi. Međutim, računanje faktorijela možemo riješiti i bez uvođenja nove varijable:

```
//...
    cout << n << "! = ";        // ispišimo početni dio poruke tako
                                // da možemo mijenjati n
```

```
    long long fjel{1};
    while (n > 1)
    {
        fjel *= n;
        --n;
    }
    cout << fjel << endl;
```

Zadatak: *Program za ispis datoteke napišite tako da umjesto `while`-bloka upotrijebite `for`-blok naredbi.*

Koji će se pristup koristiti (`for`-blok ili `while`-blok) prvenstveno ovisi o sklonostima programera. Mnogi programeri su skloniji korištenju naredbe `while` jer smatraju da je, zbog njenog sažetijeg izgleda, generirani izvedbeni kôd kraći i brži. No, većina prevoditelja će za obje naredbe generirati gotovo identičan izvedbeni kôd! Zbog preglednosti i razumljivosti kôda `for`-blok je preporučljivo koristiti kada se broj ponavljanja petlje kontrolira cjelobrojnim brojačem. U protivnom, kada je uvjet ponavljanja određen nekim logičkim uvjetom, praktičnije je koristiti `while` naredbu.

4.3.3. Blok `do-while`

Zajedničko `for` i `while` naredbi jest ispitivanje uvjeta izvođenja prije izvođenja naredbi bloka. Zbog toga se može dogoditi da se blok naredbi ne izvede niti jednom. Međutim, ponekad je potrebno da se prvo izvede neka operacija te da se, ovisno o njenom ishodu, ta operacija eventualno ponavlja. Za ovakve slučajeve svrsishodnija je `do-while` petlja:

```
do
    // blok_naredbi
while ( uvjet_ponavljanja );
```

Svakako treba uočiti da:

`while` uvjet ponavljanja mora biti zaključen znakom točka-zarezom `;` (za razliku od naredbe `while` koju je neposredno slijedio blok naredbi)

Primjenu `do-while` bloka ilustrirat ćemo igricom u kojem treba pogoditi slučajno generirani `brojKojSePogadja`. Nakon svakog našeg pokušaja ispisuje se samo poruka da li je broj veći ili manji od traženog. Petlja se ponavlja sve dok se ne pogodi traženi broj, tj. sve dok je pokušaj (`mojBroj`) različit od traženog broja.

```
#include <iostream>
#include <random>
#include <chrono>
using namespace std;
using namespace std::chrono;
```

```
int main()
{
    const int najmanji{1};
    const int najveci{100};
    // značenje sljedeće četiri naredbe detaljno je objašnjeno u tekstu
    // koji slijedi iza kôda i nemojte se previše udubljivati u njih!
    // na osnovu trenutnog vremena kreiramo klicu...
    auto klica = system_clock::now().time_since_epoch().count();
    // kojom inicijaliziramo generator pseudoslučajnih brojeva
    default_random_engine generator(static_cast<unsigned>(klica));
    // objekt koji predstavlja uniformnu raspodjelu:
    uniform_int_distribution<int> raspodjela(najmanji, najveci);
    // slučajni broj generiramo tako da raspodjeli proslijedimo
    // generator slučajnih brojeva
    int brojKojiSePogadja = raspodjela(generator);

    cout << "Trebaš pogoditi broj između " << najmanji << " i "
         << najveci << endl;

    int brojPokusaja{0};
    int mojBroj;
    do
    {
        cout << ++brojPokusaja << ". pokušaj: ";
        cin >> mojBroj;
        if (mojBroj > brojKojiSePogadja)
            cout << "MANJE!" << endl;
        else if (mojBroj < brojKojiSePogadja)
            cout << "VEĆE!" << endl;
    } while (mojBroj != brojKojiSePogadja);

    cout << "BINGO!!!" << endl;
    cout << "Iz " << brojPokusaja << ". pokušaja!" << endl;
    return 0;
}
```


Prije početka petlje do-while program odabire slučajni broj koji treba pogađati. Kôd za generiranje slučajnog broja je početniku zasigurno nerazumljiv. Za razumijevanje tog kôda potrebno je poznavanje materije koja će biti opisana u kasnijim poglavljima knjige pa se čitatelj ne mora previše njime opterećivati – ovdje ćemo ukratko opisati samo princip rada tog kôda.

Za generiranje slučajnih brojeva upotrijebljen je objekt tipa `default_random_engine` iz zaglavlja `random`, koji može dati niz *pseudoslučajnih* brojeva, tj. brojeva koji su dobiveni određenim algoritmom, a unutar tog niza pojedini brojevi imaju jednaku vjerojatnost pojave. Budući da se brojevi generiraju pomoću matematičkog algoritma, za neku određenu početnu vrijednost, brojevi koji slijede će se uvijek pojavljivati u jednakom redoslijedu. To možemo lako provjeriti sljedećim programom u kojem ispisujemo prvih 20 slučajnih brojeva koje `default_random_engine` generira:

```
#include <iostream>
#include <random>
using namespace std;

int main()
{
    // stvaramo objekt generator tipa default_random_engine:
    default_random_engine generator;
    for (int i = 0; i < 20; ++i)
    {
        // generirajmo i ispišimo sljedeći slučani broj:
        cout << generator() << endl;
    }
    return 0;
}
```

Koliko god puta pokrenemo ovaj program, uvijek će se ispisati isti niz brojeva!

Igra pogađanja brojeva sigurno ne bi bila zanimljiva ako bismo nakon nekoliko pokretanja primijetili da računalo uvijek zadaje isti broj koji treba pogoditi. Da to izbjegnemo, generatoru slučajnih brojeva treba zadati neku *početnu klicu*, tj. broj koji će mu poslužiti kao inicijalna vrijednost za algoritam koji generira slučajne brojeve. Ta klica mora biti drugačija prilikom svakog pokretanja igre i idealno bi bilo da na nju igrač ne može izravno utjecati. Najčešće se za klicu koristi sistemsko vrijeme na računalu. To je vrijeme obično izraženo u djelićima sekunde (npr. milisekundama) pa je gotovo nemoguće da generator slučajnih brojeva prilikom uzastopnih pokretanja programa dobije istu klicu (osim ako ga uspijemo pokrenuti nekoliko puta u istoj milisekundi). Klicu u našem primjeru dobivamo od globalnog objekta `system_clock` koji u sebi sadrži podatke o sistemskom vremenu; taj objekt deklariran je u zaglavlju `chrono`, unutar imenika `std::chrono`. Pozivom:

```
auto klica = system_clock::now().time_since_epoch().count();
```

dobivamo broj koji odgovara vremenskom intervalu (npr. broju milisekundi) od nekog referentnog trenutka. Referentni trenutak ovisi o sustavu na kojem se program izvodi. Tu klicu prosljeđujemo objektu generator prilikom njegove inicijalizacije tako da će sada generator svaki puta započeti niz pseudoslučajnih brojeva nekim drugim brojem.

Zadatak: *Prethodni primjer kojim smo ispisivali prvih 20 slučajnih brojeva koje generira objekt tipa `default_random_generator` modificirajte tako da korisnik ručno može unijeti klicu i potom tu klicu proslijedite generatoru. Pokrenite program nekoliko puta i provjerite kako se mijenja niz brojeva za različite zadane klice.*

Zadatak: *Umjesto ručnog unošenja klice, upotrijebite sistemsko vrijeme. Pokrenite program nekoliko puta uzastopno te usporedite ispisane brojeve.*

`default_random_generator` generira brojeve u nekom rasponu vrijednosti. Najmanji i najveći broj koji generator može dati možemo saznati od njegovih funkcijskih članova `min()` i `max()`:

```
default_random_engine generator;
cout << generator.min() << endl;    // najmanji broj, npr. 0
cout << generator.max() << endl;    // najveći broj, npr. 4294967295
```

Napomenimo da su te vrijednosti implementacijski specifične i mogu se razlikovati za različite prevoditelje.

Budući da nam za igru treba broj između 1 i 100, broj kojeg generator ponudi trebamo na neki način svesti na traženi interval. Primjerice, mogli bismo na njega primijeniti operator modulo kojim bismo izdvojili najniže tri znamenke:

```
int najmanji{1};
int najveci{100};
auto klica = system_clock::now().time_since_epoch().count();
default_random_engine generator(static_cast<unsigned>(klica));
int brojKojiSePogadja = generator() % (najveci - najmanji + 1)
                        + najmanji;
```

Kako najveći broj kojeg generator može dati ne završava na 99, pažljivi čitatelj će primijetiti da ovakvo rješenje ne daje potpuno ravnomjernu raspodjelu – uz rezultate navedene u ispisu članova `min()` i `max()`, brojevi do 95 će se pojavljivati češće (doduše, neznatno češće) nego brojevi veći od 95.

Umjesto da sami izmišljamo postupak za normiranje intervala, upotrijebili smo klase `uniform_int_distribution`. Radi se o predlošku klase (§12.3), definiranom u zaglavlju `random`. Prilikom inicijalizacije objekta, proslijedili smo donju i granicu intervala. Slučajni broj iz intervala dobivamo pozivom preopterećenog operatora `()` (§18.3.4), kojem kao argument prosljeđujemo već opisani generator slučajnih brojeva (točnije: referencu na njega).

Spomenimo kako `default_random_generator` nije konkretna klasa već sinonim za konkretnu klasu koju implementira algoritam za generiranje slučajnih brojeva. Koji algoritam će biti stvarno korišten ovisi o implementaciji u biblioteci koja se isporučuje uz prevoditelja. Uz `default_random_generator`, standardna biblioteka sadrži nekoliko konkretnih generatora iz čijeg se imena može iščitati naziv algoritma koji koriste, npr. `mt19937` – Mersenne Twister 19937 (trenutno najpopularniji generator), `knuth_b`, `minstd_rand`, `ranlux24`. Prednost korištenja sinonima `default_random_generator` jest da će njemu u budućnosti možda biti pridružen neki bolji algoritam, a da pritom neće trebati mijenjati kôd koji ga koristi.

Prije nego što su u standardnu biblioteku uvedeni ovi generatori, najpopularnije rješenje za generiranje slučajnih brojeva je bila funkcija `rand()`, deklarirana u zaglavlju `cstdlib`. Ta funkcija je naslijeđena iz jezika C. Ona kao rezultat vraća slučajni broj između 0 i `RAND_MAX`, pri čemu je `RAND_MAX` konstanta također definirana u biblioteci `cstdlib`. Prije prvog poziva funkcije `rand()` treba pozvati funkciju `srand()` za inicijalizaciju klice i njoj kao argument navesti neki cijeli broj na osnovu kojeg će se izračunati vrijednost klice. Za generiranje klice se koristila funkcija `time()`, definirana u standardnoj biblioteci `ctime`, koja učitava se trenutno sistemsko vrijeme (broj sekundi što ih

odbrojava interni sat u računalu od referentnog trenutka. Ovako bi izgledala implementacija u našem programu:

```
#include <cstdlib>
#include <ctime>

    // generiranje slučanog broja na stari način:
    srand(time(0));     // inicijalizira generator slučajnih brojeva
    int slucajniBroj = rand();
    int brojKojiSePogadja = slucajniBroj % (najveci - najmanji + 1)
                            + najmanji;
    // ...
```

Danas se korištenje funkcije `rand()` ne preporučuje jer ne daje dovoljno uniformu raspodjelu.

Zadatak: *Napišite program u kojem se računa približna vrijednost funkcije sinus pomoću reda potencija*

$$\sum_{i=0}^{\infty}(-1)^i\frac{x^{2i+1}}{(2i+1)!}=x-\frac{x^3}{3!}+\frac{x^5}{5!}-\frac{x^7}{7!}+\frac{x^9}{9!}-\ldots$$

Petlju za računanje članova reda treba ponavljati sve dok je apsolutna vrijednost zadnjeg izračunatog člana reda veća od nekog zadanog broja (npr. 10^{-7}). Uputa: ispred `do-while` *petlje definirajte decimalnu varijablu* `zbrojReda` *(početno 0), te cjelobrojne varijable* `red` *i* `preznak` *koje početno postavite na +1. Unutar* `do-while` *petlje definirajte decimalnu varijablu* `ntiClan` *i inicijalizirajte ju na vrijednost učitanog* `x`*, iza čega slijedi* `for` *petlja. Indeks petlje* `j` *ide od 1 do* `red` *i unutar nje se* `ntiClan` *množi s kvocijentom* `x/j`*. Na izlasku iz* `for` *petlje pomnožite tako množeni* `ntiClan` *s* `predznak`*, dodajte to u* `zbrojReda`*, promijenite* `predznak` *i na kraju ispitajte da li je* `ntiClan` *veći od postavljene granice – ako jest, petlju ponovite.*

Zadatak: *Napišite program koji će simulirati bacanje kocke prilagodbom kôda iz igre za pogađanje brojeva. Dodajte u simulaciju i drugu kocku te napravite petlju koja će se ponavljati sve dok se na obje kocke ne pojavi broj 6. Pokrenite program nekoliko puta te provjerite li prosjek broja bacanja 36 (vjerojatnost da se pojave dvije šestice je 1/36). Naputak: Postoje dva moguća rješenja za simulaciju dvije kocke: (1) kreiraju se dva objekta tipa* `default_random_generator`*, koji predstavljaju dvije kocke te se oni naizmjenično prosljeđuju objektu* `uniform_int_distribution` *ili (2) se bacanje dvije kocke simulira s dva uzastopna poziva operatora* `()` *na objektu tipa* `uniform_int_distribution`*. Drugo rješenje je dostatno, a ako implementirate prvo rješenje, treba paziti da se generatori inicijaliziraju različitim klicama, jer će u protivnom brojevi na kockama biti kod pojedinog bacanja međusobno jednaki! U oba slučaja, inicijalizaciju objekata dovoljno je obaviti ispred petlje za ponavljanje; unutar petlje se samo poziva operator koji generira brojeve.*

4.3.4. Slijedna naredba `for`

Jezik C++ omogućava definiranje tipova koji u sebi sadrže niz podataka nekog tipa. Želimo li uzastopno dohvatiti sve članove niza na jednostavan i brz način, upotrijebit ćemo *slijednu naredbu* `for` (engl. *range-based* `for`) koja ima općeniti oblik:

```
for ( deklaracija_člana : niz )
    // blok_naredbi
```

Ovakva petlja `for` iz objekta *`niz`* dohvaća uzastopno članove niza. U svakom prolazu uzima sljedeći član i pridružuje ga imenu navedenom u *`deklaraciji_člana`*. Preko tog imena možemo unutar pripadajućeg bloka naredbi rukovati tim članom. Na primjer, tip `string` u sebi sadrži niz znakova. Želimo li tekst sadržan u objektu tog tipa ispisati velikim slovima, možemo napisati sljedeću petlju:

```
string pjesmica{"tata kupi mi auto, bicikl i mobitel"};
locale lokal;
// uzmi svaki znak iz teksta...
for (auto znak : pjesmica)
{
    // ...i ispiši njegovo veliko slovo:
    cout << toupper(znak, lokal);
}
```

Unutar petlje korištena je standardna funkcija `toupper()` koja kao rezultat vraća veliko slovo za slovo koje je joj je proslijeđeno kao argument. Budući da je pretvorba malih u velika slova i obrnuto specifična za pojedine jezike, kao drugi argument funkciji treba proslijediti objekt tipa `locale` koji određuje prema kojim jezičnim pravilima će se pretvorba obaviti.

Slijedna petlja `for` može se primijeniti samo na objekte koji omogućavaju obilazak po članovima niza. Stoga će prevoditelj za sljedeći oblik javiti pogrešku:

```
int broj{5};
for (auto a : broj)     // pogreška: objekt se ne može obilaziti
    // ...
```

Valja napomenuti da će u gornjim primjerima naredba `for` dohvatiti član niza i na osnovu njega stvoriti presliku kojom će rukovati unutar bloka naredbi. Zato na ovaj način ne možemo promijeniti članove niza:

```
string pjesmica{"tata kupi mi auto, bicikl i mobitel"};
locale lokal;
// neuspjeli pokušaj promjene znakova u velika slova:
for (auto znak : pjesmica)
    znak = toupper(znak, lokal);
cout << pjesmica << endl;   // ispisat će nepromijenjeni tekst!
```

Da bismo mogli izravno dohvatiti i promijeniti član niza, moramo taj član dohvatiti preko reference (§6.2):

```
// u bloku baratamo referencom na član niza:
for (auto& znak : pjesmica)
    znak = toupper(znak, lokal);
cout << pjesmica << endl;   // ispisat će tekst velikim slovima
```

Budući da se nizovima i referencama bavi sljedeće poglavlje, u njemu ćemo se dodatno pozabaviti slijednom petljom `for`.

4.3.5. Naredbe za prekid petlje

Ponekad treba prekinuti izvođenje naredbi unutar petlje prije nego što dođemo do provjere uvjeta izvođenja navedenog u naredbi `for`, odnosno `while`. Naredbom `break`, koju smo upoznali u `switch` grananjima (§4.2.3), prekida se izvođenje okolne `for`, `while` ili `do-while` petlje. Na primjer, želimo da se izvođenje našeg program za ispis brojeva iz datoteke `brojevi.dat` (na str. 127) prekine kada se učita 0. Tada ćemo `while`-petlju modificirati na sljedeći način:

```
// ...
while ((ulazniTok >> broj) != 0)
{
    if (broj == 0)
        break;                  // prekida petlju učitavanja
    cout << broj << endl;
}
// ...
```

Nailaskom na znak broj 0 program iskače iz `while`-petlje te se prekida daljnje čitanje datoteke i ispis njena sadržaja.

Naredba `continue` također uzrokuje skok programa na kraj petlje, ali se potom njeno ponavljanje nastavlja. Na primjer, želimo li da naš program za ispis sadržaja datoteke ispisuje samo one znakove koji se mogu prikazati na zaslonu, jedna moguća varijanta bila bi:

```
// ...
char znak;
while (ulazniTok.get(znak))
{
    if (iscntrl(znak))
        continue;       // preskače naredbe do kraja petlje
    cout << znak;
}
// ...
```

Nailaskom na znak koji predstavlja kontrolni kôd (znakovi koji imaju kod 0x00 - 0x1F te znak DEL koji ima kod 0x7F), standardna funkcija `iscntrl()` će vratiti cijeli broj

različit od 0 pa se izvodi naredba `continue` – preskaču se sve naredbe do kraja petlje (u ovom slučaju je to naredba za ispis znaka), ali se ponavljanje petlje dalje nastavlja kao da se ništa nije dogodilo.

Ako je više petlji ugniježđeno jedna unutar druge, naredba `break` ili naredba `continue` prouzročit će prekid ponavljanja, odnosno nastavak samo najbliže petlje u kojoj se naredba nalazi.

Valja izbjegavati često korištenje `break` i `continue` naredbi u petljama, jer one narušavaju strukturiranost programa.

Koristite ih samo za „izvanredna" stanja, kada ne postoji drugi prikladan način da se izvođenje naredbi u petlji prekine. Naredba `continue` redovito se može izbjeći `if`-blokom.

Ipak, postoje slučajevi kada `break` naredba unutar petlje može značajno pojednostavniti kôd. To se prvenstveno odnosi na petlje unutar kojih se moraju izvesti dvije ili više operacija, a ponavljanje petlje ovisi o rezultatu prve operacije (ili operacije koja nije posljednja u petlji). Na primjer, pretpostavimo da treba napraviti statistiku omjera visine i tjelesne mase za grupu ljudi. U petlji prvo učitavamo visinu osobe, a potom njenu masu i te podatke dodajemo zbirnim visinama i masama koje ćemo na kraju podijeliti. Upis podataka ćemo prekinuti tako da za visinu upišemo 0. Kôd bi mogao izgledati ovako:

```
int ukupnaVisina = 0;
int ukupnaMasa = 0;
int visina;
cin >> visina;                  // učitavamo visinu
while (visina != 0)
{
    ukupnaVisina += visina;
    int masa;
    cin >> masa;
    ukupnaMasa += masa;
    cin >> visina;              // učitavamo visinu
}
cout << static_cast<double>(ukupnaMasa) / ukupnaVisina << endl;
```

Ono što bode u oči jest da naredbu za učitavanje visine imamo na dva mjesta: ispred petlje i kao zadnju naredbu u petlji. Ako bismo uspjeli ukloniti jednu od naredbi (očito je da to mora biti ona izvan petlje) pojednostavnili bismo kôd i olakšali naknadne promjene. Prebacimo ispitivanje na kraj, tj. zamijenimo `while`-petlju `do-while`-petljom:

```
// izostavili smo deklaracije i inicijalizacije
do
{
    cin >> visina;              // samo jedno učitavanje visine
    if (visina != 0)
```

```
    {
        ukupnaVisina += visina;
        int masa;
        cin >> masa;
        ukupnaMasa += masa;
    }
} while (visina != 0);
```

Učitavanje visine sada obavljamo samo na početku petlje, nakon čega slijedi provjera: ako je visina različita od 0, izvodi se ostatak petlje; u protivnom se skače na kraj petlje gdje nas čeka isto ispitivanje! Očiti je nedostatak ovog kôda da se isto ispitivanje ponavlja dva puta. Ako uvjet u `if` naredbi obrnemo, možemo pomoću `break` naredbe odmah iskočiti iz petlje, bez dodatnog ispitivanja:

```
do
{
    cin >> visina;              // samo jedno učitavanje visine
    if (visina == 0)
        break;
    ukupnaVisina += visina;
    int masa;
    cin >> masa;
    ukupnaMasa += masa;
} while (visina != 0);
```

Sada je ispitivanje na kraju petlje potpuno besmisleno, jer do kraja petlje ionako dolazimo samo u slučaju kada je `visina` različita od 0! Umjesto da radimo provjeru, procesorsko vrijeme ćemo uštedjeti tako da navedemo izraz koji će uvijek biti istinit:

```
do
{
    int visina;     // definiciju možemo sada ubaciti u petlju
    // ...
} while (true);
```

Kako je ovime petlja postala beskonačna, možemo se vratiti na početnu `while`-petlju:

```
while (true)
{
    // potpuno isto tijelo petlje
}
```

S druge strane, ako uvjet ponavljanja petlje ovisi o zadnjoj operaciji, tada je korištenje `break` naredbe nesvrhovito:

```
while (true)            // beskonačna petlja!
{
    int i;
    cin >> i;
    if (i == 0)         // ako je učitani broj 0,
        break;          // tada prekini petlju
}
```

Ova petlja je razumljivija ako je napisana ovako:

```
int i;
do
{
    cin >> i;
} while(i != 0);
```

Za razliku od izvornog rješenja, uvjet izvođenja petlje je odmah uočljiv. Doduše, za ovako kratku petlju možemo se još i nekako sporiti koliko je prvi oblik nepregledniji. Međutim, ako petlja sadrži 50-ak naredbi, među njima će teško biti uočiti naredbu i uvjet za iskakanje.

Zbog čitljivosti i lakšeg održavanja kôda valja izbjegavati korištenje beskonačnih petlji i iskakanje iz njih `break` naredbama.

Prije nego što se odlučite za beskonačnu petlju, prisjetite se izreke koja se pripisuje Albertu Einsteinu: *„Samo su dvije stvari beskonačne: svemir i ljudska glupost; no, za svemir nisam potpuno siguran”*. Beskonačne petlje nerijetko upućuju na to da u trenutku kada ju je započinjao pisati, programer nije imao blagu ideju do kada se petlja treba ponavljati, a onda ga je u jednom uzvišenom trenutku tijekom pisanja tijela petlje ukazanje prosvijetlilo i izvelo na pravi put! Uostalom, kada ste trebali učiti za neki ispit, na samom početku ste postavili uvjet „učit ću dok ne naučim” ili „učit ću do ispitnog roka”, a nikako „učit ću u beskonačno” uz nadu da će negdje uletjeti prekid te beskonačne petlje.

Nasuprot beskonačnim petljama, postoje i programeri koji pišu petlje koje će se izvesti samo jednom! Jedan od autora ove knjige bio je zabezeknut kada je u jednom komercijalnom projektu vidio sljedeći kôd i trebalo mu je dosta vremena da shvati njegov smisao (kôd prenosimo u nešto skraćenom obliku):

```
do                  // primjer besmislene petlje!
{
    bool uvjet = funkcija();
    if (uvjet)
        break;
    // nekoliko jednostavnih naredbi
} while (false);
// nastavak kôda...
```

Zadatak: *Objasnite tijek izvođenja programa ovisno o rezultatu funkcije. Kako bi se gornji kôd dao puno razumljivije napisati, s istom funkcionalnošću?*

4.4. Naredbe za bacanje i obradu iznimki

Tijekom izvođenja programa može doći do odstupanja od očekivanog slijeda naredbi. Na primjer, program očekuje od korisnika da unese cijeli broj, a korisnik umjesto toga unese neki proizvoljni tekst. Za takve *iznimke* (engl. *exceptions*) treba u programski kôd dodati naredbe koje će ih na odgovarajući način obraditi. U nekim situacijama obrada iznimke omogućit će oporavak i normalni nastavak programa. Za gore spomenuti primjer, korisniku se može ispisati upozorenje da je unos neispravan te mu omogućiti ponovni unos cijelog broja. Međutim, kada bi se podaci učitavali iz datoteke, jednostavan oporavak nakon neispravnog podatka ne bi bio moguć. U takvim se prilikama najčešće korisniku ispiše poruka, a potom program prekine.

Mehanizam bacanja i obrade iznimki je u današnjim programskim jezicima vrlo popularan jer omogućava jednostavno razdvajanje glavne logike kôda od dijela koji je zadužen za obradu pogrešaka i oporavak programa. Ovo je naročito korisno u situacijama kada se iznimka može pojaviti u nekoj od funkcija koje se pozivaju u kompleksnim izrazima. Umjesto da se provjerava uspješnost svake pojedine funkcije koja se poziva u izrazu, cijeli izraz se može napisati u jednoj naredbi, a iznimka bačena iz funkcije će prekinuti izračunavanje cjelokupnog izraza, bez bojazni da će se dobiti neki nesuvisli rezultat.

4.4.1. Bacanje iznimke

Pojava iznimke u programskom kôdu signalizira se *dizanjem iznimke* (engl. *raising exception*), pomoću ključne riječi `throw`:

```
throw objekt_nekog_tipa;
```

Budući da *throw* na engleskom jeziku znači „baciti”, redovito se umjesto *dizanja iznimke* govori o *bacanju iznimke* (engl. *throwing exception*). Iza ključne riječi `throw` navodi se objekt koji se „baca”. To može biti objekt bilo kojeg tipa koji se može preslikati (§9.4.8) ili pomaknuti (§9.13.4), uključujući i ugrađene tipove, na primjer:

```
throw 3;        // bacamo cijeli broj
```

Međutim, u pravilu se bacaju korisnički i posebno za tu namjenu definirani tipovi koji u sebi mogu sadržavati dodatne informacije o iznimci, uobičajeno neki tekstovni opis. Tip objekta koji se baca definira kojeg je tipa iznimka. U standardnoj biblioteci već je definirano nekoliko tipova iznimki, no njima ćemo se detaljno pozabaviti u poglavlju o iznimkama.

Kako se baca korisnički definirana iznimka pokazat ćemo na sljedećem primjeru kojim računamo korijene kvadratne jednadžbe. Ako kvadratna jednadžba za unesene

koeficijente ima realne korijene, izračunavamo ih i ispisujemo. U protivnom bacamo standardnu iznimku tipa `runtime_error`, definiranu u zaglavlju `stdexcept`:

```
#include <iostream>
#include <stdexcept>        // definira runtime_error
using namespace std;

int main()
{
    cout << "a = ";
    double a;
    cin >> a;
    cout << "b = ";
    double b;
    cin >> b;
    cout << "c = ";
    double c;
    cin >> c;

    double diskriminanta = b * b - 4 * a * c;
    if (diskriminanta < 0)
        throw runtime_error{"Za zadane koeficijente ne postoje"
                            " realni korijeni"};

    cout << "x1 = " << (-b + sqrt(diskriminanta)) / 2 << endl;
    cout << "x2 = " << (-b - sqrt(diskriminanta)) / 2 << endl;

    return 0;
}
```

Prilikom bacanja iznimke, stvaramo objekt tipa `runtime_error` i prosljeđujemo mu opis zbog čega je došlo do iznimke. Taj opis postaje dio objekta kako bi bio dostupan kôdu koji će iznimku uhvatiti.

Naredbom za bacanje iznimke trenutno se prekida izvođenje kôda u tekućem bloku naredbi i traži se okružujući blok koji je zadan da hvata i obrađuje iznimku tog tipa. Ako se takav blok ne uspije pronaći, iznimka će ostati *neobrađena* (engl. *unhandled exception*) i program će se srušiti uz odgovarajuću poruku operacijskog sustava. U gornjem primjeru ne postoji okolni blok koji bi uhvatio iznimku tipa `runtime_error`, budući da se naredba `throw` nalazi izravno unutar funkcije `main()`. Zbog toga će program za nepodržane vrijednosti koeficijenata srušiti.

Zadatak. *Pokrenite program i unesite za sve koeficijente vrijednost 1 te pogledajte kako će se manifestirati rušenje programa na vašem računalu.*

Naravno da u pravilu ne želimo da svaka pogreška u programu uzrokuje njegovo rušenje. Da to izbjegnemo, u gornji kôd treba dodati okružujući blok koji će uhvatiti iznimku i eventualno omogućiti povratak izvođenja na početak.

4.4.2. Hvatanje i obrada iznimke

Postupak hvatanja i obrade iznimke obavlja se pomoću para blokova `try-catch`:

```
try
{
    // blok „pokušaja"
}
catch ( deklaracija_iznimke )
{
    // blok „hvatanja"
}
```

U bloku `try` (*blok pokušaja*) navode se naredbe glavne logike kôda, koje eventualno mogu baciti iznimku. Ako su sve naredbe u tom bloku uspješno izvedene bez bačene iznimke, program se nastavlja prvom naredbom iza bloka `catch` (*bloka hvatanja*). Međutim, ako je neka naredba u bloku pokušaja bacila iznimku, izvođenje naredbi u tom bloku se odmah prekida i tijek izvođenja se nastavlja naredbama u bloku hvatanja.

U naredbi `catch` se unutar zagrada navodi deklaracija tipa iznimke. Tom deklaracijom se određuje koji tipovi iznimki će uopće biti uhvaćeni i obrađeni u dotičnom bloku hvatanja. Ako iznimka bačena u bloku pokušaja ne odgovara tom tipu (precizno rečeno: ako nije tog tipa ili tipa koji je izveden iz njega, vidi §10), ona neće biti obrađena u tom bloku već će se tražiti sljedeći okružujući blok koji hvata taj tip iznimke. Ako se ne uspije pronaći odgovarajući blok hvatanja, iznimka će ostati neobrađena i program će se srušiti.

Obradu iznimki ilustrirat ćemo programom koji izračunava rezultat jednostavne matematičke operacije zbrajanja, oduzimanja, množenja ili dijeljenja dva broja. Izraz ćemo upisivati u liniji za unos, a program će taj cijeli unos učitati, iz njega izlučiti lijevi i desni operand te operaciju. Ako je sve ispravno zadano, program će izračunati i ispisati vrijednost izraza.

Budući da se učitava cijeli izraz, on će se početno učitati kao niz znakova, do kraja unesenog retka. Taj niz znakova pohranit ćemo u objekt tipa `string` koji služi za pohranu teksta, a upoznali smo ga u primjerima iz uvodnog dijela knjige. Iz tog niza znakova program mora izdvojiti znakove koji pripadaju broju lijevo od operatora i pretvoriti te znakove u decimalni broj. Zatim treba izlučiti sam operator i na kraju znakove desno od operatora pretvoriti u decimalni broj. Na osnovu operatora izvest će se jedna od računskih operacija s oba izlučena broja.

Za pretvorbu niza znakova u broj iskoristit ćemo standardnu funkciju `stod()`, definiranu u zaglavlju `string`. Funkciji treba proslijediti niz znakova, a ona kao rezultat vraća decimalni broj tipa `double`:

```
double broj = stod("10e-3");    // broj će biti 0,001
```

Ukoliko niz ne sadrži ispravan decimalni broj, funkcija će baciti iznimku tipa `invalid_argument` i na taj način signalizirati nedozvoljeni unos.

Za početak napišimo program bez obrade iznimki. Kada proučimo koji dio kôda može baciti iznimke i kojeg tipa, uklopit ćemo blokove pokušaja i hvatanja.

```
#include <iostream>
#include <string>       // definicija tipa string i funkcije stod()
using namespace std;

int main()
{
    // želimo li koristiti decimalne zareze umjesto točaka, treba
    // aktivirati lokalne postavke:
    // setlocale(LC_NUMERIC, "croatian");
    string izraz;           // ovdje ćemo pohraniti upisani izraz
    getline(cin, izraz);    // cijeli redak učitavamo u objekt izraz
    // petlju ponavljamo sve dok ne stisnemo odmah Enter:
    while (izraz.size() > 0)
    {
        size_t pozicija;    // pozicija kraja prvog broja
        // pretvaramo niz u broj i postavljamo poziciju kraja broja:
        double rezultat = stod(izraz, &pozicija);
        // ako smo došli do kraja izraza, nešto ne valja...
        if (pozicija == string::npos)
            throw runtime_error{"Nepotpuni izraz"};
        // preskačemo eventualne praznine iza prvog broja:
        pozicija = izraz.find_first_not_of(' ', pozicija);
        // ako smo došli do kraja izraza, nešto ne valja...
        if (pozicija == string::npos)
            throw runtime_error{"Nepotpuni izraz"};
        // pohranimo znak operatora (+, -, / ili *)
        char operacija = izraz.at(pozicija);
        // izlučimo ostatak teksta koji sadrži desni broj:
        string desniBroj = izraz.substr(pozicija + 1);
        // za različite operacije se tijek grana:
        switch (operacija)
        {
        case '+':
            rezultat += stod(desniBroj);
            break;
        case '-':
            rezultat -= stod(desniBroj);
            break;
        case '*':
            rezultat *= stod(desniBroj);
            break;
        case '/':
            rezultat /= stod(desniBroj);
            break;
        default:
            throw runtime_error{"Neispravni operator"};
        }
        // ispisujemo izraz i rezultat:
```

```
        cout << izraz << " = " << rezultat << endl;
        // učitavamo redak sljedećeg unosa i ponavljamo petlju:
        getline(cin, izraz);
    }
    return 0;
}
```

Cijeli redak unosa učitavamo u objekt `izraz` (tipa `string`) pomoću standardne funkcije `getline()` koja kao prvi argument prima ulazni tok, a drugi argument je objekt u koji treba uneseni tekst prebaciti.

Da ne bismo morali pisati zasebne naredbe koje će prethodno provjeravati korektnost unesenog izraza, provjeru radimo tijekom izračunavanja. Provjeru zapisa lijevog, odnosno desnog operatora prepustili smo funkciji `stod()` koja će u slučaju neispravnog zapisa broja baciti iznimku.

Nakon što prvim pozivom funkcije `stod()` izlučimo lijevi operand, iz teksta trebamo izlučiti operator. Netko bi u prvi mah pomislio da ga možemo izlučiti tako da u nizu `izraz` jednostavno potražimo neki od znakova `+`, `-`, `*` ili `/`:

```
size_te pozicijaOperatora = izraz.find_first_of("+-*/");
```

Međutim, sami operandi mogu sadržavati znakove `+` i `-` kao predznake, uključujući predznak eksponenta za broj zadan u znanstvenoj notaciji. Prema tome, potragu za operatorom ne smijemo početi od prvog znaka unosa, nego od prvog znaka iza lijevog operanda. Kako ćemo znati gdje je kraj lijevog operanda? Za to smo jednostavno iskoristili funkciju `stod()`: ako se funkciji kao drugi parametar proslijedi adresa neke cjelobrojne varijable, funkcija će po završenoj pretvorbi u tu varijablu pohraniti poziciju iza zadnjeg znaka kojeg je ona interpretirala kao dio broja:

```
// cjelobrojna varijabla u koju će funkcija stod()
// pohraniti poziciju zadnjeg znaka u broju:
size_t pozicija;
double rezultat = stod(izraz, &pozicija);
```

Prvo smo deklarirali varijablu `pozicija`. Nju nema potrebe inicijalizirati jer će njenu vrijednost postaviti funkcija `stod()`. Deklaracijom će prevoditelj osigurati memorijski prostor za varijablu, a funkciji `stod()` prosljeđujemo adresu te varijable tako da ispred njenog imena navedemo operator adrese `&` (§6.1). Funkcija zadani izraz pretvara u decimalni broj, znak po znak, sve dok ne naiđe na znak koji više ne može biti dio broja, na primjer prazninu, neko slovo izuzev slova `e` ili neki simbol. U tom trenutku se čitanje zadanog niza prekida, na adresu koja je proslijeđena kao drugi argument se zapisuje pozicija iza zadnjeg ispravnog znaka, a funkcija kao rezultat vraća ono što je pretvorbom u decimalni broj do tog znaka dobila.

Nakon poziva funkcije `stod()` provjeravamo da slučajno nismo došli do kraja unesenog izraza. To bi značilo da uneseni izraz sadrži samo lijevi operand, tj. da je izraz nepotpun pa u tom slučaju bacamo standardnu iznimku tipa `runtime_error`. Toj iznimci prosljeđujemo poruku koju ćemo kasnije u bloku hvatanja ispisati.

U općenitom slučaju, između lijevog operanda i operatora mogu biti praznine. Njih ćemo preskočiti pozivom funkcijskog člana `find_first_not_of()`:

```
pozicija = izraz.find_first_not_of(' ', pozicija);
```

Prvi argument je znak koji trebamo preskočiti, a drugi je pozicija od koje trebamo preskakanje započeti. U našem primjeru preskakanje počinjemo od prvog znaka iza lijevog operanda. Funkcija kao rezultat vraća poziciju prvog znaka koji nije iz zadanog skupa; u ovom primjeru će to biti pozicija prvog znaka koji nije praznina. Ako je izraz dobro upisan, taj znak bi trebao biti operator. Nakon poziva funkcijskog člana, ponovno provjeravamo jesmo li stigli do kraja upisanog izraza te u tom slučaju bacamo iznimku.

Znak za operator učitavamo pomoću funkcijskog člana `at()`:

```
char operacija = izraz.at(pozicija);
```

Iako bi bilo logično da se ova varijabla zove `operator`, taj naziv nismo mogli upotrijebiti jer je `operator` ključna riječ u jeziku C++ (vidi tablicu 3.1). Ovisno o vrijednosti tog znaka, tôk se dalje grana za različite operacije. U svakoj od grana se pomoću funkcije `stod()` učitava desni operand te izvodi dotična operacija. Ako znak nije niti jedan od simbola predviđenih za operatore, izraz je očito pogrešno zadan. Zbog toga u grani `default` bacamo iznimku tipa `runtime_error`.

Revni čitatelj će primijetiti da smo učitavanje desnog operanda mogli smjestiti ispred grananja `switch` te time izbjeći ponavljanje poziva funkcije `stod()` u pojedinim granama `case`:

```
        // umjesto teksta, izlučimo desni broj odmah:
        string desniBroj = izraz.substr(pozicija + 1);
        double desni = stod(desniBroj);
        switch (operacija)
        {
        case '+':
            rezultat += desni;
            break;
        // ...
```

Ovime bismo dobili nešto kraći izvorni i izvedbeni kôd, ali bi redoslijed provjere izraza a time i signalizacije pogrešaka bio poremećen. U slučaju neispravnog desnog argumenta, funkcija `stod()` bi bacila iznimku prije nego što bismo uopće stigli provjeriti ispravnost operatora. To znači da bi za unos oblika:

```
1.23 ? b
```

program prijavio neispravni (desni) operand, iako već prije njega imamo neispravni operator.

Cijeli kôd za provjeru i izračunavanje izraza smješten je u petlji `while` čiji uvjet osigurava da se ona ponavlja sve dok ne unesemo prazan izraz.

Da bismo uklopili blokove pokušaja i blokove hvatanja, kao prvo moramo odlučiti što napraviti u slučaju iznimke: da li samo ispisati poruku o pogrešci i prekinuti izvođenje programa ili ispisati poruku o pogrešci i omogućiti korisniku da upiše novi izraz. Budući da se drugi scenarij čini svrsishodnijim, blokove pokušaja i hvatanja ćemo smjestiti unutar petlje `while`:

```
// dopunjeni kôd s umetnutim blokom pokušaja:
while (izraz.size() > 0)
{
    try
    {
        size_t pozicija;
        double rezultat = stod(izraz, &pozicija);
        //...
        cout << izraz << " = " << rezultat << endl;
    }
    catch (/*...*/)
    {
        //...
    }
    getline(cin, izraz);
}
//...
```

Slijedeći korak jest prepoznati koje iznimke mogu biti bačene i koje od njih želimo hvatati, tj. omogućiti oporavak i nastavak programa ako budu bačene. Kao što smo već spomenuli, ne uspije li zadani tekst pretvoriti u decimalni broj, funkcija `stod()` će baciti iznimku tipa `invalid_argument`. Stoga, dodajmo blok hvatanja za tu iznimku:

```
// dopunjeni kôd s blokom hvatanja:
while (izraz.size() > 0)
{
    try
    {
        //...
    }
    catch (invalid_argument)
    {
        cerr << "Neispravni operand" << endl;
    }
    // ...
}
//...
```

Iznimka koju baca funkcija `stod()` u sebi sadrži i tekstovni opis, nešto poput „neispravni argument funkciji stod”. Kako u našem primjeru takav opis nije baš previše koristan, nismo ga u bloku hvatanja niti iskoristili već smo ispisali svoju poruku. Kasnije ćemo pokazati kako se taj tekst može dohvatiti iz iznimke i ispisati.

Zadatak. *Dodajte u program kontrolnu cjelobrojnu varijablu* `operandBr` *koja će omogućiti da prilikom ispisa poruke navedemo koji je operand prouzročio iznimku:*

```
        catch (invalid_argument)
        {
            cerr << "Neispravni " << operandBr << ". operand" << endl;
        }
```

Budući da ta varijabla mora biti vidljiva u bloku hvatanja, treba ju deklarirati ispred, a ne unutar bloka pokušaja. Naravno, ne zaboravite varijablu inicijalizirati prije učitavanja i povećati poslije učitavanja prvog operanda.

Osim iznimke koju baca funkcija `stod()`, unutar bloka pokušaja sami bacamo iznimke tipa `runtime_error`. U trenutnoj izvedbi tu iznimku ne hvatamo, što znači da bi ona bila neobrađena. Općenito, ako iza bloka pokušaja treba hvatati različite tipove iznimki, za svaki željeni tip navodi se zasebna naredba `catch` s deklaracijom pripadajućeg tipa:

```
try
{
    // blok „pokušaja”
}
catch ( deklaracija_iznimke1 )
{
    // ...
}
catch ( deklaracija_iznimke2 )
{
    // ...
}
```

Unutar svakog pojedinog bloka hvatanja je kôd specifičan za oporavak od dotične iznimke.

Dodajmo blok hvatanja za iznimke tipa `runtime_error`:

```
    while (izraz.size() > 0)
    {
        try
        {
            //...
        }
        catch (invalid_argument)
        {
            cerr << "Neispravni operand" << endl;
        }
        catch (runtime_error iznimka)
        {
            cerr << iznimka.what() << endl;
        }
```

```
        // ...
    }
    //...
```

Unutar dodanog bloka hvatanja ispisujemo opis sadržan u iznimci pozivom njenog funkcijskog člana `what()`. Taj je funkcijski član zajednički za sve standardne iznimke u biblioteci jezika C++. Pritom se na iznimku referiramo koristeći ime `iznimka` koje je navedeno u deklaraciji iznimke unutar naredbe `catch`.

Zadatak. *Dodajte u prethodni primjer provjeru da li nakon učitanog desnog operanda ima još znakova različitih od praznine. U tom slučaju bacite iznimku tipa `runtime_error` s odgovarajućom porukom (npr. „Izraz nije pravilno zaključen”).*

Zadatak. *Izmijenite program za izračunavanje izraza tako da umjesto cijelog retka učitavate zasebno lijevi operand, operator i desni operand pomoću ulaznog tôka `cin`. Uspješnost učitavanja pojedinog podatka možete provjeriti pomoću vrijednost tôka `cin` nakon učitavanja:*

```
if (cin >> lijeviBroj)
    // ...
```

Ta vrijednost će postati nula u slučaju neispravnog unosa.

Zadatak. *Dodajte blokove pokušaja i hvatanja u kôd za izračunavanje korijena kvadratne jednadžbe iz odjeljka 4.4.1.*

4.5. Ostale naredbe za skok

Naredba `goto` omogućava bezuvjetni skok na neku drugu naredbu unutar iste funkcije. Njen općeniti oblik je:

```
goto ime_oznake ;
```

`ime_oznake` je simbolički naziv koji se mora nalaziti ispred naredbe na koju se želi prenijeti kontrola, odvojen znakom : (dvotočka). Na primjer

```
if (a < 0)
    goto negativniBroj;
//...
negativniBroj:      //naredba
```

Naredba na koju se želi skočiti može se nalaziti bilo gdje (ispred ili iza naredbe `goto`) unutar iste funkcije. *`ime_oznake`* mora biti jedinstveno unutar funkcije, ali može biti jednako imenu nekog objekta ili funkcije. Ime oznake jest identifikator, pa vrijede pravila navedena u odjeljku 3.1.

Naredbom `goto` ne može se skočiti iza deklaracije varijable koja se koristi iza naredbe na koju se skače:

```
    goto izlaz;     // preskače deklaraciju varijable a
    // ...
        int a{5};
    // ...
izlaz:
    cout << a;      // pogreška: varijabla a nije deklarirana
```

Moguće je skakanje ispred deklaracije i u tom slučaju se varijabla ponovno inicijalizira:

```
// ovo je u biti beskonačna petlja:
ponovniPocetak:
    int a{21};
    if (a == 21)
    {
        ++a;
        goto ponovniPocetak;
    }
```

U pravilno strukturiranom programu naredba `goto` uopće nije potrebna, te ju velika većina programera uopće ne koristi.

Budući da narušava logičku strukturiranost programa, `goto` naredba čini kôd nečitljivijim i težim za praćenje i održavanje, a uz to prevoditelju otežava optimizaciju izvedbenog kôda.

Zadnju naredbu za kontrolu toka koju ćemo ovdje spomenuti upoznali smo na samom početku knjige. To je naredba `return` kojom se prekida izvođenje funkcija te ćemo se njome pozabaviti u poglavlju 8 posvećenom funkcijama.

4.6. O strukturiranju izvornog kôda

Uvlačenje blokova naredbi i pravilan raspored vitičastih zagrada doprinose preglednosti i čitljivosti izvornog kôda. Programeri koji tek započinju pisati u programskim jezicima C ili C++ često se nađu u nedoumici koji stil strukturiranja koristiti. Obično preuzimaju stil knjige iz koje pretipkavaju svoje prve programe ili od kolege preko čijeg ramena kriomice stječu prve programerske vještine, ne sagledavajući nedostatke i prednosti tog ili nekog drugog pristupa. Nakon što im taj stil „uđe u krv”, teško će prijeći na drugi bez obzira koliko je on bolji (u to su se uvjerili i sami autori knjige tijekom njena pisanja!).

Prvi problem jest raspored vitičastih zagrada koje označavaju početak i kraj blokova naredbi. Navedimo nekoliko najčešćih pristupa (redoslijed navođenja je slučajan):

1. vitičaste zagrade uvučene i međusobno poravnate (*Whitesmithsov stil*, engl. *Whitesmiths style*):

```
for ( /*...*/ )
    {
    // blok naredbi
    // ...
    }
```

2. početna zagrada na kraju naredbe za kontrolu, završna poravnata s naredbama u bloku (*stil Waynea Ratliffa*, engl. *Ratliff style*):

```
for ( /*...*/ ) {
    // blok naredbi
    // ...
    }
```

3. zagrade izvučene, međusobno poravnate (*stil Erica Allmana*, engl. *Allman style*):

```
for ( /*...*/ )
{
    // blok naredbi
    // ...
}
```

4. početna zagrada na kraju naredbe za kontrolu, završna izvučena (*stil Kernighana i Ritchiea*, engl. *Kernighan and Ritchie style*, *K&R style*):

```
for ( /*...*/ ) {
    // blok naredbi
    // ...
}
```

Pristup 3 ima još podvarijantu u kojoj naredbe bloka počinju u istom retku u kojem je početna zagrada, uvučene (*stil Cay Horstmanna*, engl. *Horstmann style*):

```
for ( /*...*/ )
{   // blok naredbi
    // ...
}
```

Već letimičnim pogledom na četiri gornja pristupa čitatelj će uočiti da izvučena završna zgrada u 3. odnosno 4. pristupu jače ističe kraj bloka. U 3. pristupu zagrade u paru otvorena-zatvorena vitičasta zagrada međusobno su poravnate, tako da je svakoj zagradi lako uočiti njenog para, što može biti vrlo korisno prilikom ispravljanja programa. Zbog navedenih razloga, danas je najpopularniji 3. pristup pa ga stoga i mi koristimo u knjizi (*Kud svi Turci, tuda i mali Mi*).

Druga nedoumica jest koliko duboko uvlačiti blokove. Za uvlačenje blokova najprikladnije je koristiti tabulatore. Urednici teksta (*tekst editori*) obično imaju početno ugrađeni pomak tabulatora od po 8 znakova, međutim za iole složeniji program s više blokova ugniježđenih jedan unutar drugoga, to je previše. Najčešće se za izvorni kôd koristi uvlačenje po 4 ili samo po 2 znaka. Uvlačenje po 4 znaka je dovoljno duboko da

bi se i početnik mogao lagano snalaziti u kôdu, pa smo ga zato i mi koristimo u knjizi. Iskusnijem korisniku dovoljno je uvlačenje i po samo 2 znaka, što ostavlja dovoljno prostora za dugačke naredbe. Naravno da kod definiranja tabulatora treba voditi računa o tipu znakova (*fontu*) koje se koristi u editoru. Za pravilnu strukturiranost kôda neophodno je koristiti neproporcionalno pismo (npr. *Courier New* ili *Consolas*) kod kojeg je širina svih znakova jednaka.

Treći „estetski" detalj vezan je uz praznine oko operatora. Početniku u svakom slučaju preporučujemo umetanje praznina oko binarnih operatora, ispred prefiks-operatora i iza postfiks operatora, te umetanje zagrada kada je god u nedoumici oko hijerarhije operatora. U protivnom osim estetskih, možete imati i sintaktičkih problema. Uostalom, neka sam čitatelj procijeni koji je kôd je čitljiviji:

```
a = b * c - d / (2.31 + e) + e / 8.21e-12 * 2.43;
```

ili

```
a=b*c-d/(2.31+e)+e/8.21e-12*2.43;
```

Sličan problem je vezan za praznine u naredbama koje definiraju petlje. Neki programeri smatraju da je praznina ispred zagrade suvišna pa će radije napisati:

```
for(int i = 0; i < 10; ++i)
```

dok s druge strane, neki vole praznine stavljati i s unutrašnje strane zagrada:

```
for ( int i = 0; i < 10; ++i )
```

Koji ćete pristup preuzeti ovisi isključivo o vašoj odluci, ali ga onda svakako koristite dosljedno.

4.7. Kutak za buduće C++ „gurue"

Jedna od odlika programskog jezika C++ jest mogućnost sažetog pisanja naredbi. Podsjetimo se samo naredbe za inkrementiranje koja umjesto:

```
i = i + 1;
```

omogućava jednostavno napisati:

```
++i;
```

Također, mogućnost višekratne uporabe operatora pridruživanja u istoj naredbi, dozvoljava da se primjerice umjesto dviju naredbi:

```
a = b + 25.6;
c = e * a - 12;
```

napiše samo jedna:

```
c = e * (a = b + 25.6) - 12;
```

s potpuno istim efektom. Ovakvo sažimanje kôda ne samo da zauzima manje prostora u uredniku teksta, odnosno izvornom kôdu, već često olakšava prevoditelju generiranje kraćeg i bržeg izvedbenog kôda. Štoviše, mnoge naredbe jezika C++ (poput naredbe za inkrementiranje) vrlo su bliske strojnim instrukcijama mikroprocesora, pa se njihovim prevođenjem dobiva maksimalno efikasan kôd. Pri sažimanju kôda posebno valja paziti na hijerarhiju operatora (vidi tablicu 3.15). Često se zaboravlja da logički operatori imaju niži prioritet od poredbenih operatora, a da operatori pridruživanja imaju najniži prioritet. Zbog toga nakon naredbe:

```
c = 4 * (a = b - 5);
```

varijabla a neće imati istu vrijednost kao nakon naredbe:

```
c = 4 * ((a = b) - 5);
```

ili nakon naredbi:

```
a = b;
c = 4 * (b - 5);
```

U prvom primjeru će se prvo izračunati `b - 5` te će rezultat dodijeliti varijabli `a`, što je očito različito od drugog, odnosno trećeg primjera, gdje se prvo varijabli `a` dodijeli vrijednost od `b`, a zatim se provede oduzimanje.

Kod „C-gurua” su uobičajena sažimanja u kojima se umjesto eksplicitne usporedbe s nulom, kao na primjer:

```
if (a != 0)
{
    // ...
}
```

piše implicitna usporedba:

```
if (a)
{
    // ...
}
```

Iako će oba kôda raditi potpuno jednako, suštinski gledano je prvi pristup ispravniji (i čitljiviji) – izraz u ispitivanju `if` po definiciji mora davati logički rezultat tipa `bool`. U drugom, sažetom primjeru se, sukladno ugrađenim pravilima pretvorbe, aritmetički tip

pretvara u tip `bool` tako da se brojevi različiti od nule pretvaraju u `true`, a nula se pretvara u `false`, pa je konačni ishod isti.

Slična situacija je prilikom ispitivanja da li je neka varijabla jednaka nuli. U „dosljednom" pisanju ispitivanje bismo pisali kao:

```
if (a == 0)
{
    // ...
}
```

dok bi nerijetki „gurui" to kraće napisali kao:

```
if (!a)
{
    // ...
}
```

I u ovom slučaju će oba ispitivanja polučiti isti izvedbeni kôd, ali će neiskusnom programeru iščitavanje drugog kôda biti zasigurno teže. Štoviše, neki prevoditelji će prilikom prevođenja ovako sažetih ispitivanja ispisati upozorenja.

Mogućnost sažimanja izvornog kôda (s eventualnim popratnim zamkama) ilustrirat ćemo programom u kojem tražimo zbroj svih cijelih brojeva od 1 do 100. Budući da se radi o trivijalnom računu, izvedbeni program će i na najsporijim suvremenim strojevima rezultat izbaciti za manje od sekunde. Stoga nećemo koristiti Gaussov algoritam za rješenje problema, već ćemo „zdravo-seljački" napraviti petlju koja će zbrojiti sve brojeve unutar zadanog intervala. Krenimo s prvom verzijom:

```
// ver. 1.0
#include <iostream>
using namespace std;

int main()
{
    int zbroj = 0;
    for(int i = 1; i <= 100; ++i)
        zbroj = zbroj + i;
    cout << zbroj << endl;
    return 0;
}
```

Odmah uočavamo da naredbu za zbrajanje unutar petlje možemo napisati kraće:

```
// ver. 2.0
//...
for (int i = 1; i <= 100; ++i)
{
     zbroj += i;
//...
```

Štoviše, zbrajanje možemo ubaciti u izraz prirasta `for` petlje:

```
// ver. 3.0
//...
for (int i = 1; i <= 100; zbroj += i, ++i) ;
//...
```

Izrazi odvojeni znakom `,` (zarezom) izračunavaju se postupno, slijeva na desno. Stoga, da su oni u naredbi `for` napisani obrnutim redoslijedom:

```
for (int i = 1; i <= 100; ++i, zbroj += i)      // pogrešan zbroj!
```

konačni zbroj bi bio pogrešan – prvo bi se uvećao brojač, a zatim tako uvećan dodao zbroju – kao rezultat bismo dobili zbroj brojeva od 2 do 101.

Dva izraza u naredbi prirasta možemo stopiti u jedan izraz. No, u tom slučaju prefiksni operator inkrementiranja brojača `i` treba zamijeniti postfiksnim, tj. prvo se mora dohvatiti vrijednost brojača, dodati zbroju i tek potom brojač povećati:

```
// ver. 4.0
//...
for (int i = 1; i <= 100; zbroj += i++) ;
//...
```

Usporedimo li zadnju inačicu (`4.0`) s prvom, skraćenje kôda je očevidno. Ali je isto tako očito da je kôd postao nerazumljiviji: prebacivanjem naredbe za pribrajanje brojača u naredbu `for` zakamuflirali smo osnovnu naredbu zbog koje je uopće petlja napisana. Budući da većina današnjih prevoditelja ima ugrađene vrlo efikasne postupke optimizacije izvedbenog kôda, pitanje je koliko će i hoće li uopće ovakva sažimanja rezultirati bržim i efikasnijim programom. Stoga vam preporučujemo:

Ne trošite previše energije na ovakva sažimanja, jer će najvjerojatnije rezultat biti neproporcionalan uloženom trudu, a izvorni kôd postati nečitljiv(iji).

Tek kada je cijeli program ili modul gotov, provjerava se brzina izvođenja te ako ona ne zadovoljava, pristupa se optimizaciji kôda. Pri tome je dobro znati da se u svim većim programima vrlo mali dio kôda izvodi stalno, a velik dio kôda se izvodi tek povremeno. Iskusni programeri to zovu „pravilo 10-90”: 10 % kôda se izvodi 90 % vremena. Zato se prvenstveno treba usredotočiti na dio kôda koji se izvodi često, jer će se i minimalne uštede u tom dijelu višestruko odraziti na brzinu programa. Na primjer, ako uspijemo izvođenje bloka naredbi unutar petlje koja se izvodi 10000 puta skratiti za 1/100 sekunde, trajanje cijele petlje će se skratiti za 10 sekundi!

Za provjeru trajanja izvođenja dijelova kôda unutar programa koriste se posebni programi ili moduli za *profiliranje* (engl. *profiler*). Oni vode statistiku tijekom izvođenja programa, zapisujući ukupno i postotno trajanje izvođenja pojedinih funkcija ili naredbi, broj poziva pojedinih funkcija i sl. Na osnovi tih statističkih podataka, progra-

mer će lako uočiti koji odsječci programa se izvode presporo ili koji se dijelovi kôda izvode često te će svoje napore za optimizacijom usmjeriti na te dijelove kôda.

Spomenimo još jednu dilemu koja se često pojavljuje i kod iskusnijih programera: da li koristiti prefiks ili postfiks verziju unarnih operatora ++ i --. Naravno, u složenim naredbama u kojima je redoslijed operacija *dohvaćanje vrijednosti – promjena vrijednosti* bitan (kao npr. u ver. 4.0 našeg gornjeg primjera), dileme nema. Za ugrađene tipove podataka neće biti nikakve razlike među brzinama izvođenja obiju varijanti. No, za korisnički definirane tipove podataka će postfiks operator redovito generirati nešto sporiji izvedbeni kôd (vidi §18.3.6). Stvaranje navike koja će se isplatiti tek kasnije je razlogom zašto smo i u dosadašnjim primjerima prednost davali prefiks operatoru, iako smo računali isključivo s ugrađenim tipovima podataka.