0. 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.

Osnovno pitanje je što C++ čini boljim i pogodnijim općenamjenskim jezikom za pisanje 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".

0.1. Povijesni pregled razvoja programskih jezika

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*).

U asemblerskom jeziku svaka strojna instrukcija predstavljena je mnemonikom koji je razumljiv ljudima koji čitaju program. Tako se zbrajanje najčešće obavlja mnemonikom `ADD`, dok se premještanje podataka obavlja mnemonikom `MOV`. Time se postigla bolja čitljivost programa, no i dalje je bilo vrlo složeno pisati programe i ispravljati ih jer je bilo potrebno davati sve, pa i najmanje upute računalu za svaku pojedinu operaciju. Javlja se problem koji će kasnije, nakon niza godina, dovesti i do

pojave C++ programskog jezika: potrebno je razviti programerski alat koji će osloboditi programera rutinskih poslova te mu dopustiti da se usredotoči na problem koji rješava.

Zbog toga se pojavljuje niz viših programska jezika, koji preuzimaju na sebe neke "dosadne" programerske poslove. 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 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. Do unatrag nekoliko godina, 99% komercijalnih programa bili su pisani u C-u, ponegdje dopunjeni 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 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 razredima" (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 koristio se jezikom Simula, koji posjedovao neka važna svojstva koja su ga činila prikladnim za taj posao. Na primjer, Simula je posjedovala pojam razreda: strukture podataka koje objedinjavaju podatke i operacije nad podacima. Korištenje razreda 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 užasno 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++. Pri tome je iskoristio svoje iskustvo stečeno

prilikom rada na doktoratu te pokušao stvoriti univerzalni jezik koji će udovoljiti današnjim zahtjevima. Pri tome je uzeo dobra svojstva niza jezika: Simula, Clu, Algol68 i Ada, a kao osnovu je uzeo jezik C.

0.2. 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.

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 riješiti ovako:

1. Listaj sve objekte redom.
2. Za pojedini objekt odredi njegov tip.
3. Ovisno o tipu, pozovi ispravnu 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, 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 kutova 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 uvjetno shvatiti) 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 koje zatim međusobno surađuju u rješavanju problema. Umjesto specijaliziranih procedura koje barataju podacima, radimo s objektima koji objedinjavaju operacije i podatke. 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, boljim, ako oba rade istu stvar.

Ključ za postizanje takvog cilja jest spajanje podataka i operacija, poznato pod nazivom *enkapsulacija* (engl. *encapsulation*). Pritom su podaci privatni za svaki objekt te ne smiju biti dostupni ostalim dijelovima programa. To svojstvo se naziva *skrivanje podataka* (engl. *data hiding*). Svaki objekt svojoj okolini pruža isključivo podatke koji su joj neophodni da bi se objekt mogao iskoristiti. Ti podaci zajedno s operacijama koje ih prihvaćaju ili vraćaju čine *sučelje* objekta. Programer koji će koristiti taj objekt više se ne mora zamarati razmišljajući o načinu na koji objekt funkcionira – on jednostavno traži od objekta određenu uslugu.

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 disketu. 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. Kako je kompaniji vrlo vjerojatno u cilju što brže razviti novi model kako bi pretekla konkurenciju, gotovo sigurno će jednostavno uzeti uspješan model automobila i samo izmijeniti neka njegova svojstva: promijenit će mu liniju, pojačati motor, dodati ABS kočnice. 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 primjerice 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. To se zove *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 bilo kada kasnije program proširi novim tijelima, nije potrebno mijenjani program koji rotira sve objekte – samo je za novi objekt potrebno 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.

0.3. 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. No 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 razreda 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 *virtualni osnovni razredi*. Nasljeđivanje samo ušteđuje programeru višestruko pisanje kôda te olakšava ponovno korištenje već napisanih programskih odsječaka.

Virtualne funkcije i virtualni osnovni razredi, naprotiv, mogu unijeti značajno usporenje u program. Korištenje virtualnih funkcija se obavlja tako da se prije poziva konzultira posebna tablica, pa je jasno da će poziv svake takve funkcije biti sporiji. Također, pristup članovima virtualnih osnovnih razreda se redovito obavlja preko jednog pokazivača više. 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.

No razmislimo o još jednom problemu: 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. Složeni problemi zahtijevaju nove pristupe njihovom rješavanju: zbog toga imamo na raspolaganju nova računala s većim mogućnostima koja su sposobna učiniti gubitak performansi beznačajnim u odnosu na dobitak u brzini razvoja programa.

Slično je i s objektnom tehnologijom: možda će i dobiveni kôd biti sporiji i veći od ekvivalentnog C kôda, no jednostavnost njegove izrade će sigurno 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!) Tehnologija ide naprijed: dok se gubi neznatno na brzini i memorijskim zahtjevima, dobici su višestruki.

Također, 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 izrade razreda 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 gornjem 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*.

0.4. 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). Danas (listopad 2000.) je to već prilično zrela tehnologija koja je prošla niz izmjena, a realno je očekivati još takvih promjena do njena potpunog "sazrijevanja". Upravo zbog toga je (barem za sada) nezahvalno raditi isključive usporedbe tipa "ovo je puno bolje napravljeno u jeziku A nego u jeziku B". Pokušajmo stoga samo naznačiti koje su značajnije razlike između Jave i jezika C++; konačne zaključke prepuštamo čitatelju.

0.4.1. 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 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 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++.

0.4.2. Java kôd se izvodi na virtualnom stroju

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 će prevedeni Java kôd biti izvodljiv na bilo kojem računalu s bilo kojim procesorom i operacijskim sustavom. Naravno, pod uvjetom da na tom stroju postoji i "vrti se" (odgovarajući) virtualni stroj. No, takav 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, Java virtualni strojevi pružaju mogućnost *Prevođenja upravo na vrijeme* (engl. *Just-In-Time compilation*), kojim se Java binarni kôd prevodi u matični izvedbeni 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 će predstavljati značajni gubitak vremena, odnosno usporenje programa.

0.4.3. Java nema pokazivača

Davno su prošla vremena kada su se na sâm spomen riječi "pokazivač" 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 istrijebljenje pokazivača.

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

0.4.4. Java nema predložaka niti višestrukog nasljeđivanja.

Predlošci omogućavaju da se isti kôd može koristiti za različite tipove podataka, bez potrebe da se taj kôd ponovno piše za nove tipove ili da se metodom *kopiraj-i-umetni* preslikava na različita mjesta i naknadno dorađuje.

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. U Javi nema višestrukog nasljeđivanja implementacije, ali nudi se višestruko nasljeđivanje sučelja korisnički definiranih podataka prema korisniku. To zahtijeva drugačiji stil programiranja koji je orijentiran prema sučeljima objekata.

0.4.5. Java ima ugrađeno automatsko sakupljanje smeća.

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

U Javi se sam sustav brine da oslobodi memoriju koja više nije potrebna – mehanizam za *sakupljanje smeća* (engl. *garbage collection*) ugrađen je u sustav i automatski se pokreće po potrebi. Naravno da za sve one koji su iskusili gore opisane probleme u jeziku C++ ovo predstavlja pravi melem. 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 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 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.

0.4.6. Java podržava višenitnost

Višenitnost (engl. *multithreading*) omogućava pisanje programa koji izvode dvije ili više operacija istovremeno. Primjerice, možete napisati sportsku simulaciju u kojoj vaš automobil vozite trkaćom stazom, a istovremeno se u gornjem kutu ekrana odbrojava vrijeme – svaku od ove dvije radnje stavit ćete u vlastitu nit. Procesor će, ovisno o prioritetu pojedinih niti, naizmjence određeni dio vremena posvetiti izvođenju instrukcija u svakoj od niti.

U Javu je višenitnost podržana od početka, uključujući i sve detalje, kao što su međusobna sinkronizacija niti te zaključavanje kritičnih podataka i metoda. U jeziku C++ višenitnost nije ugrađena, već morate imati odgovarajuću biblioteku koja sadrži podršku.

0.4.7. Java ne podržava preopterećenje operatora

Preopterećenje operatora omogućava da se funkcija 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.

0.4.8. 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.

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

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 redovito 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). U listopadu 1997. godine prihvaćen je ISO standard jezika C++ (ANSI/ISO/IEC 14882).

Pojavom završnog oblika standarda je dinamički proces promjena u jeziku C++ zaustavljen (barem na neko vrijeme) . Stoga bi bilo logično očekivati da se svi prevoditelji koji su se na tržištu pojavili nakon 1997. ponašaju u skladu sa Standardom. Nažalost, to nije slučaj, tako da vam se može lako dogoditi da i na najnovijem prevoditelju nekog proizvođača (imena nećemo spominjati – "Sjedni Bille, nitko ti nije rekao da se digneš!") ne možete prevesti kôd iz knjige ili da vam prevedeni kôd radi drukčije od onoga što smo mi napisali.

Naravno (unaprijed se posipamo pepelom), ne otklanjamo mogućnost da smo i mi napravili pogrešku. Većinu primjera smo istestirali pomoću prevoditelja koji je uglavnom usklađen sa Standardom, ... ali nikad se ne zna. Oznaku prevoditelja nećemo navoditi da nam netko ne bi prigovorio da objavljujemo (strane) plaćene reklame (a ne zato jer bi se radilo o piratskoj kopiji – kopija je legalno kupljena i licencirana).

0.6. Literatura

Tijekom pisanja knjige koristili smo sljedeću literaturu (navodimo ju abecednim slijedom autora):

[ANSI95] *Working Paper for Draft Proposed International Standard for Information Systems – Programming Language C++*, Technical Report X3J16/95-0087, American National Standards Institute (ANSI), April 1995

[Barton94] John J. Barton, Lee R. Nackman: *Scientific and Engineering C++ – An Introduction with Advanced Techniques and Examples*, Addison-Wesley, Reading, MA, 1994, ISBN 0-201-53393-6

[Borland94] *Borland C++ 4.5 Programmer's Guide*, Borland International, Scotts Valley, CA

[Carroll95] Martin D. Carroll, Margaret A. Ellis: *Designing and Coding Reusable C++*, Addison-Wesley, Reading, MA, 1995, ISBN 0-201-51284-X

[Eckel99] Bruce Eckel: *Thinking in C++*, Prentice Hall, Englewood Cliffs , NJ, 1999, ISBN 0-13-917709-4

[Ellis90] Margaret A. Ellis, Bjarne Stroustrup: *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, MA, 1990, ISBN 0-201-51459-1

[Hanly95] Jeri R. Hanly, Elliot B. Koffman, Joan C. Horvath: *C Program Design for Engineers*, Addison-Wesley, Reading, MA, 1994, ISBN 0-201-59064-6

[Horstmann96] Cay S. Horstmann: *Mastering C++ – An Introduction to C++ and Object-Oriented Programming for C and Pascal Programmers (2nd Edition)*, John Wiley and Sons, New York, 1996, ISBN 0-471-10427-2

[ISO/IEC98] ISO/IEC 14882 International Standard: *Programming Languages – C++*, American National Standards Institute, 1998.

[Josuttis99] Nicolai M. Josuttis: *The C++ Standard Library – A Tutorial and Handbook*, Addison Wesley Longman, Reading MA, 1999, ISBN 0-201-37926-0

[Kernighan88] Brian Kernighan, Dennis Ritchie: *The C Programming Language (2nd Edition)*, Prentice-Hall, Englewood Cliffs, NJ, 1988, ISBN 0-13-937699-2

[Kukrika89] Milan Kukrika: *Programski jezik C*, Školska knjiga, Zagreb, 1989, ISBN 86-03-99627-X

[Liberty96] Jesse Liberty, J. Mark Hord: *Teach Yourself ANSI C++ in 21 Days*, Sams Publishing, Indianapolis, IN, 1996, ISBN 0-672-30887-6

[Lippman91] Stanley B. Lippman: *C++ Primer (2nd Edition)*, Addison-Wesley, Reading, MA, 1991, ISBN 0-201-53992-6

[Lippman96] Stanley B. Lippman: *Inside the C++ Object Model*, Addison-Wesley, Reading, MA, 1996, ISBN 0-201-83454-5

[Lippman98] Stanley B Lippman, Josée Lajoie: *C++ Primer (3rd Edition)*, Addison-Wesley Longman, Reading, MA, 1998, ISBN 0-201-82470-1

[Lippman00] Stanley B Lippman: *Essential C++*, Addison-Wesley Longman, Reading, MA, 2000, ISBN 0-201-48518-4

[Meyers98] Scott Meyers: *Effective C++ – 50 Specific Ways to Improve Your Programs and Design (2nd Edition)*, Addison-Wesley Longman, Reading, MA, 2000, ISBN 0-201-92488-9

[Murray93] Robert B. Murray: *C++ Strategies and Tactics*, Addison-Wesley, Reading, MA, 1993, ISBN 0-201-56382-7

[Schildt90] Herbert Schildt: *Using Turbo C++*, Osborne/McGraw-Hill, Berkeley, CA, 1990, ISBN 0-07-881610-6

[Stepanov95] Alexander Stepanov, Meng Lee: *The Standard Template Library*, Hewlett-Packard Laboratory, Palo Alto, CA, 1995

[Stroustrup91] Bjarne Stroustrup: *The C++ Programming Language (2nd Edition)*, Addison-Wesley, Reading, MA, 1991, ISBN 0-201-53992-6

[Stroustrup94] Bjarne Stroustrup: *The Design and Evolution of C++*, Addison-Wesley, Reading, MA, 1994, ISBN 0-201-54330-3

[Strostrup00] Bjarne Stroustrup: *The C++ Programming Language (Special Edition)*, Addison-Wesley Longman, Reading, MA, 2000, ISBN 0-201-70073-5

[Weidl96] Johannes Weidel: *The Standard Template Library Tutorial*, Technical University Vienna, 1996

Ponegdje se pozivamo na neku od knjiga, navodeći pripadajuću, gore navedenu oznaku u uglatoj zagradi. Također smo koristili članke iz časopisa *C++ Report* u kojem se može naći mnoštvo najnovijih informacija. Časopis izlazi od 1989. godine, deset puta godišnje, a izdaje ga SIGS Publications Group, New York. Godišta 1991–1995 objavljena su na CDROM-u (SIGS Books, New York, ISBN 1-884842-24-0). Krajem 2000. godine, časopis se "stopio" sa sličnim časopisom *Journal of Object-Oriented Programming*. Uz to, mnoštvo odgovora te praktičnih i korisnih rješenja može se naći na *Usenet* grupama za rasprave (*newsgroups*): *comp.lang.c++*, *comp.lang.c++.moderated* kao i na mnoštvu WWW stranica na Internetu. Obavezno posjetite osobne stranice Bjarnea Stroustrupa (*http://www.research.att.com/~bs*), na kojima ćete naći mnoštvo zanimljivih informacija o povijesti i specifičnostima jezika C++, kao i putokaze na druge zanimljive stranice.

Od svih navedenih knjiga, "najreferentniji" je C++ standard; može se kupiti kod Američkog Nacionalnog Instituta za Standarde (ANSI) ili se (uz plaćanje USD 18) može skinuti u PDF formatu sa WWW stranica Nacionalnog Odbora za Standarde Informatičke Tehnologije (*National Committee for Information Technology Standards*) http://www.ncits.org/. Na Internetu (npr. ftp://ftp.research.att.com/dist/c++std/WP/CD2/) se još uvijek može naći završna inačica nacrta Standarda[†] iz travnja 1995., koja je besplatna. Dostupna je u čistom tekstovnom formatu, PDF (Acrobat Reader) formatu ili u PostScript formatu. Broji oko 600 stranica formata A4 (oko 4MB komprimirane datoteke), a kao kuriozitet navedimo da je preko CARNet mreže trebalo u kolovozu 1995. oko 6 sati vremena da se prebaci na naše računalo (toliko o Internetu u Hrvata!).

Sâm Standard ne preporučujemo za učenje jezika C++, a također vjerujemo da neće trebati niti iskusnijim korisnicima – Standard je prvenstveno namijenjen proizvođačima softvera; svi noviji *prevoditelji* (engl. *compilers*) bi se trebali ponašati u skladu s tim standardom (što, na žalost, nije slučaj). No, smatramo da bi svaki "ozbiljni" C++ programer morao imati knjigu B. Stroustrupa: *The C++ Programming Language* (treće 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).

[†] Iako se radi o *nacrtu* Standarda, on u potpunosti definira sve karakteristike programskog jezika kako je definirano u konačnoj verziji Standarda. Sve promjene koje su se događale od objave Nacrta u travnju 1995. odnosile su se isključivo na tekst standarda – sintaksa jezika i definicije biblioteka ostale su nepromijenjene.

Uz nju, najtopliju preporuku zaslužuje knjiga S. Lippmana i J. Lajoie: *C++ Primer* (treće izdanje) u kojoj su mnogi detalji jasnije objašnjeni nego u Stroustrupovoj knjizi.

Zadnje poglavlje ove knjige posvećeno je principima objektno orijentiranog programiranja. Naravno da to nije dovoljno za ozbiljnije sagledavanje – detaljnije o objektno orijentiranom programiranju zainteresirani čitatelj može pročitati u referentnoj knjizi Grady Booch: *Object-Oriented Analysis and Design with Applications (2nd Edition)*, Benjamin/Cummings Publishing Co., Redwood City, CA, 1994, ISBN 0-8053-5340-2, kao i u nizu članaka istog autora u časopisu *C++ Report*. Jedna od knjiga koju bi svaki ozbiljniji programer trebao na svojoj polici jest *Design Patterns: Elements of Reusable Object-Oriented Software*, čiji su autori Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, izdavač Adison Wesley Longman, Reading Massachusetts, 1995 (ISBN 0201633612).

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 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: *Programming Applications for Microsoft Windows*, Microsoft Press, 1999, ISBN 1-57231-996-8.

0.7. 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, 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).

0.7.1. Zahvale uz drugo 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 knjigu poručujemo da će se posthumno peći u vječnoj vatri *Xeroxovih* i *Canonovih* tonera, a djeca će im izgledati kao izblijedjele fotokopije dobivene mokrim postupkom (naravno, svoje grijehe će iskupiti uništе li odmah fotokopiju i kupe primjerak knjige u knjižari).

Zahvale upućujemo i svima koji su dali prijedloge ili nas upozorili na pogreške u prvom izdanju. Također, zahvaljujemo se Julijanovim bivšim kolegama sa Zavoda za elektroniku, mikroelektroniku, računalne i inteligentne sustave Fakulteta elektrotehnike i računarstva koji su dozvolili da na njihovom poslužiteljima održavamo WWW stranice vezane uz ovu knjigu.

1. “Oluja” kroz jezik C++

‘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”

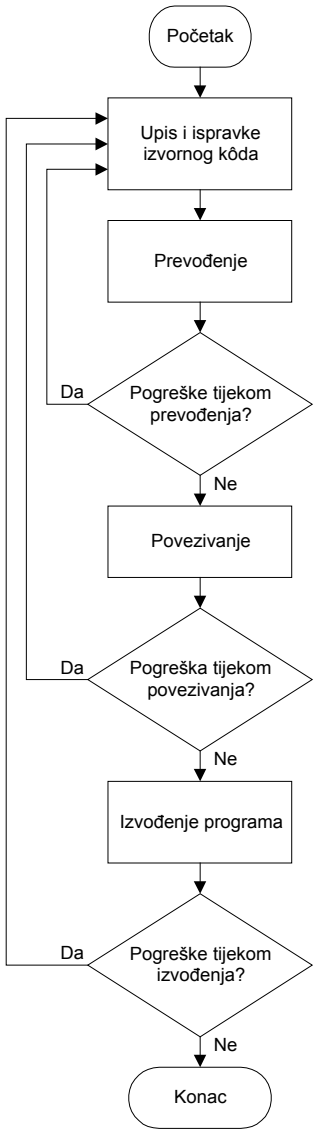
U prvom poglavlju proletjet ćemo kroz osnovne pojmove vezane uz programiranje i upoznat ćemo se s strukturom najjednostavnijih programa pisanih u programskom jeziku C++. Ovo poglavlje prvenstveno je namijenjeno čitateljicama i čitateljima koji nisu nikada pisali programe u bilo kojem višem programskom jeziku i onima koji su to možda radili, ali je “od tada prošla čitava vječnost”.

1.1. 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. Pokretanjem programa s diska, diskete ili CD-ROM-a, 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 duljine nekoliko



Slika 1.1. Tipičan razvoj programa

megabajta, 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 nešto 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, pod pisanjem programa u širem smislu podrazumijeva se 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 uspješno 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 1.1 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 programom za upis i ispravljanje izvornog kôda. Te programske cjeline poznatije su pod nazivom *integrirane razvojne okoline* (engl. *integrated development environment, IDE*). 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[†]. Prevoditelj tijekom prevođenja provjerava sintaksu napisanog izvornog kôda i u slučaju uočenih ili naslućenih pogrešaka ispisuje odgovarajuće poruke o pogreškama, odnosno 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 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. Međutim, takav izvedbeni kôd još uvijek ne jamči da će program raditi ono što ste zamislili. Primjerice, može se dogoditi da program radi pravilno za neke podatke, ali da se za druge podatke ponaša

[†] Ovdje nećemo opisivati konkretno kako se pokreću postupci prevođenja ili povezivanja, jer to varira ovisno o prevoditelju, odnosno povezivaču. Sažeti opis za neke popularnije prevoditelje dan je u Prilogu D na kraju knjige.

nepredvidivo. U tom se slučaju radi 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 lancu ‘ispravljanje izvornog kôda-prevođenje-povezivanje-testiranje’. Kod jednostavnijih programa broj ponavljanja će biti manji i smanjivat će se proporcionalno s rastućim iskustvom programera. 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*). 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 programu.

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 upozorenja 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!).

1.2. Moj prvi i drugi C++ program

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. Izvođenje svakog programa počinje 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 to 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 opcija. 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 (DOS, UNIX, 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.

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:

```
pkunzip -t mojzip
```

[†] U nekim programskim jezicima glavna funkcija se zove *glavni program*, a sve ostale funkcije *potprogrami*.

Ovom naredbom se pokreće program `pkunzip` i pritom mu se predaju dva parametra: `-t` i `mojzip`. U našem C++ primjeru unutar zagrada nema ništa, što znači da ne prenosimo nikakve argumente. Tako će i ostati do daljnjega, točnije do poglavlja 5.11 u kojem ćemo detaljnije obraditi funkcije, a posebice funkciju `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 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 služi kao poruka prevoditelju 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 *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 javiti prevoditelj.*

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 biblioteku `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* (pretprocesorskim naredbama posvećeno je poglavlje 17). 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 sadržanih u odgovarajućim bibliotekama. Jedna od osnovnih značajki (zli jezici će reći mana) jezika C++ jest vrlo oskudan broj funkcija ugrađenih u sam jezik. 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 razreda.

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" određeno područje imena (*imenik*, engl. *namespace*), a `std` je naziv imenika u kojem su obuhvaćane sve standardne funkcije, uključujući i funkcije iz gore opisane `iostream` biblioteke. Imenici su se prilično kasno pojavili u jeziku C++, a uvedeni su da se izbjegne kolizija istih imena funkcija ili varijabli iz različitih biblioteka. Na primjer, ako dvije različite funkcije iz različitih biblioteka imaju isto ime, prevoditelj će javiti pogrešku. Kada ne bismo imali na raspolaganju imenike, jedino rješenje u takvom slučaju bilo bi promijeniti ime funkcije u jednoj od biblioteka, što ponekad nije moguće jer proizvođači redovito te biblioteke isporučuju u već prevedenom obliku. Detaljnije o imenicima bit će govora u poglavlju 11, a do tada uzmite naredbu `using namespace std` "zdravo za gotovo".

Korisno je znati da je `iostream` novi (Standardom propisani) naziv za `iostream.h` zaglavlje. Do te promjene imena došlo je ne bez razloga. Naime, kada su uvedeni imenici, trebalo je sve standardne biblioteke uvrstiti u imenik `std`. Kako je već tada postojalo mnoštvo kôda koji je koristio `iostream.h` biblioteku, ona se zbog spojivosti sa tim starim kodom nije smjela mijenjati. Rješenje je bilo definirati novu biblioteku slična naziva u kojoj su definirane sve funkcije kao i u `iostream.h` zaglavlju, ali unutar imenika `std`. Zato je u bibliotekama većine današnjih prevoditelja zapravo unutar `iostream` datoteke samo uključena (pretprocesorskom naredbom `#include`) `iostream.h` datoteka, uz uključivanje te biblioteke u `std` imenik. Slično vrijedi za zaglavlja svih ostalih standardnih biblioteka. Inače Standard dozvoljava upotrebu starog nazivlja, ali se ono ne preporučuje.

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 15.

Međutim, to još nije sve! Iza znakovnog niza ponavlja se operator za ispis, kojeg slijedi `endl`. `endl` je konstanta u biblioteci `iostream` koja prebacuje ispis u novi redak, to jest vraća *značku* (ili *kurzor*, engl. *cursor*) na početak sljedećeg retka na zaslonu. Dovitljiva 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 gornji kôd 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. *Windows 95/98/2000, Windows NT*) i rezultirajući program ne pokrećete iz terminalskog (DOS) prozora već izravno iz grafičkog okruženja, tada se lako može dogoditi da se nakon pokretanja programa na trenutak otvori terminalski prozor, koji će se nestati prije nego što uspijete pročitati ijedno slovo teksta koji se ispisao. Osjećat ćete se nasamareni (“Iaaa!!! Zar sam zato platio punih 20 kuna lokalnom piratu za kopiju najnovijeg prevoditelja?”). Vjerojatno će odmah 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 100 megabitni) priključak na

računalo preko kojeg bi se sinkroniziralo okidanje aparata s otvaranjem prozora. Alternativa je video kamera s mogućnošću ubrzanog (10×) 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).

Mi vam nudimo sljedeća dva rješenja:
- otvorite komandni (DOS) 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 razredima 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!*

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`.

1.3. Moj treći C++ program

Sljedeći primjer je pravi mali dragulj interaktivnog programa:

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

int main() {
    int a, b, c;

    cout << "Upiši prvi broj:";
    cin >> a;                       // očekuje prvi broj

    cout << "Upiši i drugi broj:";
    cin >> b;                       // očekuje drugi broj

    c = a + b;                      // računa njihov zbroj
    // ispisuje rezultat (vidi sljedeću stranicu):
```

```
    cout << "Njihov zbroj je: " << c << endl;
    return 0;
}
```

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

```
int a, b, c;
```

u kojem se deklariraju tri varijable `a`, `b` i `c`. Ključnom riječi (*identifikatorom tipa*) `int` deklarirali smo ih kao cjelobrojne, tj. tipa `int`. Deklaracijom smo pridijelili simboličko, nama razumljivo i lako pamtljivo ime memorijskom prostoru u koji će se pohranjivati vrijednosti tih varijabli. Naišavši na te deklaracije, prevoditelj će zapamtiti njihova imena, te za njih rezervirati odgovarajući prostor u memoriji računala. Kada tijekom prevođenja ponovno sretne varijablu s nekim od tih imena, prevoditelj će znati da se radi o cjelobrojnoj varijabli i znat će gdje bi se ona u memoriji trebala nalaziti. Osim toga, tip varijable određuje raspon dozvoljenih vrijednosti te varijable, te definira operacije nad njima. 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 značka 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` (prvi broj), odnosno `b` (drugi broj). 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. Napomenimo da taj broj u našem primjeru ne smije biti veći od 32767 niti manji od –32768, jer je to dozvoljeni raspon cjelobrojnih `int` varijabli[†]. Unos većeg broja neće imati nikakve dramatične posljedice na daljnji rad vašeg računala, ali će dati krivi rezultat na kraju računa.

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.

[†] Opseg dozvoljenih vrijednosti ovisi o prevoditelju i nije strogo definiran Standardom. Za neke prevoditelje raspon je od –2.147.483.648 do 2.147.483.647. Detaljnije o rasponu dozvoljenih vrijednosti bit će riječi u odjeljku 2.4.1.

No, završimo s našim primjerom. Nakon unosa prvog broja, po istom principu se unosi drugi broj `b`, a zatim slijedi naredba za računanje zbroja

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

Ona kaže računalu da treba zbrojiti vrijednosti varijabli `a` i `b`, te njihov zbroj pohraniti u varijablu `c`, tj. u memoriju na mjesto gdje je prevoditelj rezervirao prostor za tu varijablu.

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.*

1.4. Komentari

Na nekoliko mjesta u gornjem kôdu 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 primjeru.

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 razrede definirane u datoteci te njihovo značenje, autor(i) kôda, uvjeti pod kojima je program preveden (operacijski sustav, ime i oznaka prevoditelja), zabilješke, te naknadne izmjene:

```
/*********************************************************
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++ 7.0 prevoditelj
Zabilješke: Znam Boris da si ti protiv ovakvih komentara,
            ali sam ih morao staviti
Izmjene:    15.07.1996. (jš) prva inačica
            21.08.1996. (bm) svi podaci su iz float
                             promijenjeni u int
            08.07.2000. (jš) iostream.h promijenjen u
                             iostream te ubačen namespace
*********************************************************/
```

S druge strane, s količinom komentara ne treba pretjerivati, jer će u protivnom izvorni kôd postati nepregledan. Naravno da ćete izbjegavati komentare poput:

```
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 [Strustrup91]:

- Na početku datoteke izvornog kôda opisati sadržaj datoteke.
- Kod deklaracije varijabli, razreda i objekata obrazložiti njihovo značenje i primjenu.
- Ispred funkcije 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.

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

1.5. 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 svoj kôd na 60-80 stupaca budući da je to broj znakova koji standardno stane u jedan redak ispisa na listu A4 formata. 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 – pisanje više naredbi u istom retku treba prakticirati samo u izuzetnim slučajevima.

2. 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, realne, logičke, pokazivačke podatke. U poglavlju koje slijedi upoznat ćemo se ugrađenim tipovima podataka i pripadajućim operatorima.

2.1. Identifikatori

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

1. Identifikator može biti sastavljen od kombinacije slova engleskog alfabeta (`A - Z`, `a - z`), brojeva (`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 2.1) ili nekoj od alternativnih oznaka operatora (tablica 2.2). 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 2.1. Ključne riječi jezika C++

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

a izbjegavati imena poput

```
Snjeguljica_i_7_patuljaka
MojaPrivatnaVarijabla
FrankieGoesToHollywood
RockyXXVII
```

Odmah uočimo da nije dozvoljena upotreba naših dijakritičkih znakova u identifikatorima. Čak i ako naletite na neki prevoditelj koji bi ih prihvatio, zbog prenosivosti kôda te znakove neizostavno izbjegavajte; program s varijablama `BežiJankec` ili `TeksaškiMasakrMotornjačom` na većini prevoditelja prouzročit će pogrešku prilikom prevođenja.

Valja uočiti da jezik C++ razlikuje velika i mala slova u imenima, tako da identifikatori

```
maliNarodiVelikeIdeje
MaliNarodiVelikeIdeje
malinarodiVELIKEIDEJE
```

predstavljaju tri različita naziva. Također, ne postoji teoretsko ograničenje na duljinu imena, no neki stariji prevoditelji razlikuju imena samo po prvih nekoliko znakova (pojam "nekoliko" je ovdje prilično rastezljiv). Stoga će prevoditelj koji uzima u obzir samo prvih 14 znakova, identifikatore

Tablica 2.2. Alternativne oznake operatora

and	bitand	compl	not_eq	or_eq	xor_eq
and_eq	bitor	not	or	xor	

```
Snjeguljica_i_7_patuljaka
Snjeguljica_i_sedam_patuljaka
```

interpretirati kao iste, iako se razlikuju od petnaestog znaka ('7' odnosno 's') na dalje. Naravno da dugačka imena treba izbjegavati radi vlastite komocije, posebice za varijable i funkcije koje se često ponavljaju.

Ponekad je pogodno koristiti složena imena zbog boljeg razumijevanja kôda. 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
```

(praznina umjesto znaka podcrtavanja između riječi označavala bi prekid imena, što bi uzrokovalo pogrešku prilikom prevođenja). U drugom pristupu riječi se pišu spojeno, s velikim početnim slovom:

```
SnjeguljicaTeSedamPatuljaka
```

Mi ćemo u primjerima preferirati potonji način samo zato jer tako označene varijable i funkcije zauzimaju ipak nešto manje mjesta u izvornom kôdu.

Iako ne postoji nikakvo ograničenje na početno slovo naziva varijabli, većina programera preuzela je iz programskog jezika FORTRAN naviku da imena cjelobrojnih varijabli započinje slovima `i`, `j`, `k`, `l`, `m` ili `n`.

2.2. Varijable, objekti i tipovi

Bez obzira na jezik u kojem je pisan, svaki program sastoji se od niza naredbi koje mijenjaju vrijednosti 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. *Tip* objekta, između ostalog, određuje raspored bitova prema kojem je taj objekt pohranjen u memoriju.

U programskom jeziku C++ pod objektima u užem smislu riječi obično se podrazumijevaju složeni tipovi podataka opisani pomoću posebnih "receptura" – razreda, što će biti opisano u kasnijim poglavljima. Jednostavni objekti koji pamte jedan cijeli ili realni broj se često nazivaju varijablama.

Da bi prevoditelj pravilno preveo naš izvorni C++ kôd u strojni jezik, svaku varijablu treba prije njena korištenja u kôdu *deklarirati*, odnosno jednoznačno odrediti

njen tip. U "Našem trećem C++ programu" varijable su deklarirane u prvom retku glavne funkcije:

```
int a, b, c;
```

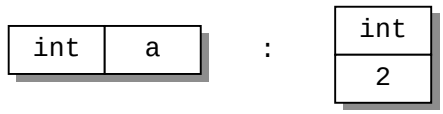
Tim deklaracijama je prevoditelju jasno dano do znanja da će varijable `a`, `b` i `c` biti cjelobrojne – prevoditelj će vrijednosti tih varijabli pohranjivati u memoriju prema svom pravilu za cjelobrojne podatke. Također će znati kako treba provesti operacije na njima. Prilikom deklaracije varijable treba također paziti da se u istom dijelu programa (točnije *bloku naredbi*, vidi poglavlje 3.1) ne smiju deklarirati više puta varijable s istim imenom, čak i ako su one različitih tipova. Zato će prilikom prevođenja sljedećeg kôda prevoditelj javiti pogrešku o višekratnoj deklaraciji varijable `a`:

```
int a, b, c;
int a;          // pogreška: ponovno korištenje naziva a
```

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 2.1. Lijevo od dvotočke je kućica koja simbolizira varijablu s njenim tipom i imenom.



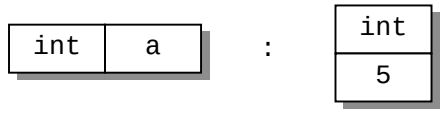
Slika 2.1. Deklaracija varijable i pridruživanje vrijednosti

Desno od dvotočke je kućica koja predstavlja objekt s njegovim tipom i konkretnom 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 2.2.

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:



Slika 2.2. Pridruživanje nove vrijednosti

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

Gornji oblik inicijalizacije naslijeđen je iz jezika C i uobičajen je za ugrađene tipove podataka. Međutim, postoji i drugi mogući oblik zapisa naredbi za inicijalizaciju, takozvanom *konstruktorskom sintaksom*( ):

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

Ovakav zapis redovito se koristi za inicijalizaciju složenih korisnički definiranih tipova koji iziskuju dva ili više parametra prilikom inicijalizacije. Primjerice, prilikom inicijalizacije kompleksnog broja treba inicijalizirati njegov realni i imaginarni. Detaljnije o ovakvom načinu inicijalizacije govorit ćemo u poglavlju o korisnički definiranim tipovima – *razredima*. Za razliku od programskog jezika C u kojem sve deklaracije moraju biti na početku programa ili funkcije, prije prve naredbe, u C++ jeziku ne postoji takvo ograničenje, pa deklaracija varijable može biti bilo gdje unutar programa. Štoviše, mnogi autori preporučuju da se varijable deklariraju neposredno prije njihove prve primjene.

2.3. 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 tip objekta prema definiranim pravilima pretvorbe, što će biti objašnjeno u poglavlju 2.4 o tipovima podataka. 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 *lvrijednosti* (engl. *lvalues*, kratica od *left-hand side values*). Pritom valja znati da se ne može svakoj lvrijednosti pridruživati nova vrijednost – neke varijable se mogu deklarirati kao konstantne (vidi

poglavlje 2.4.4) i pokušaj promjene njihove vrijednosti prevoditelj će naznačiti kao pogrešku. Stoga se posebno govori o *promjenjivim lvrijednostima*. S desne strane operatora pridruživanja mogu biti i lvrijednosti i konstante.

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 najdesnijeg znaka `=` moraju biti lvrijednosti:

```
a = b = c + d;      // OK!
a = b + 1 = c;      // pogreška: b + 1 nije lvrijednost
```

2.4. 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 na objektima, dok *pravila pretvorbe* 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.

2.4.1. Brojevi

U jeziku C++ ugrađena su u suštini dva osnovna tipa brojeva: cijeli brojevi (engl. *integers*) i realni 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 riječju `int` i njena će vrijednost u memoriji računala obično zauzeti dva *bajta*[†] (engl. *byte*), tj. 16 bitova. Prvi bit je rezerviran za predznak, tako da preostaje 15 bitova za pohranu vrijednosti. Stoga se varijablom tipa `int` mogu obuhvatiti svi brojevi od najmanjeg broja (najvećeg negativnog broja)[‡]

$$-2^{15} = -32768$$

do najvećeg broja

$$2^{15} - 1 = 32767$$

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

```
long int HrvataUDijasporiZaVrijemeIzbora = 3263456;
long ZrnacaPrasineNaMomRacunalu = 548234581;
            // mogao bih uzeti krpu za prašinu i svesti to na int!
```

Takva varijabla će zauzeti u memoriji više prostora i operacije s njom će dulje trajati.

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

- dekadskom,
- 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, 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 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. Na primjer, 11 u oktalnom sustavu odgovara broju 9 u dekadskom sustavu ($11_8 = 9_{10}$). 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

---

[†] Kod nas se često za bajt koristi autohtona hrvatska riječ *oktet*, budući da se byte sastoji od 8 bitova. U engleskom riječ byte nema nikakvo izvorno značenje, već je samo umjetna modifikacija riječi *bit* (engl. *komadić, trun, mrvica*).

[‡] Kao što smo već spomenuli, ove granične vrijednosti nisu uvijek iste, već ovise o implementaciji prevoditelja.

znamenki iznad 9 koriste se prva slova engleskog alfabeta, tako da je $A_{16} = 10_{10}$, $B_{16} = 11_{10}$, itd. (vidi tablicu 2.3). Oktalni i heksadekadski prikaz predstavljaju

Tablica 2.3. Usporedba brojeva u različitim sustavima

Dekadski	Oktalno	Heksadekadski
1	1	1
2	2	2
⋮	⋮	⋮
7	7	7
8	10	8
9	11	9
10	12	A
11	13	B
12	14	C
13	15	D
14	16	E
15	17	F
16	20	10
17	21	11
⋮	⋮	⋮
32	40	20

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) iziskivao bi puno prostora u programskom kôdu, te bi bio nepregledan. Oktalni i heksadekadski prikazi iziskuju manje prostora i iskusnom korisniku omogućuju brzu pretvorbu brojeva iz dekadskog u binarni oblik i obrnuto.

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

```
int SnjeguljicaIPatuljci = 010;   // odgovara dekadsko 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 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;                // oktalni prikaz broja 32
cout << i << endl;

i = 0x20;               // heksadekadski prikaz broja 32
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 isti broj, u dekadskom formatu.

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

```
float pi = 3.141593;
float brzinaSvjetlosti = 2.997925e8;
float nabojElektrona = -1.6E-19;
```

Prvi primjer odgovara uobičajenom načinu prikaza brojeva s decimalnim zarezom. U drugom i trećem primjeru brojevi su prikazani 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

```
float PlanckovaKonst = 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 što je ograničen raspon vrijednosti koje se mogu prikazati `float` tipom (vidi tablicu 2.4), treba znati da je i broj decimalnih znamenki u mantisi također ograničen na 7 decimalnih mjesta[†]. Čak i ako se napiše broj s više znamenki, prevoditelj će zanemariti sve niže decimalne znamenke. Stoga će ispis varijabli

[†] Broj točnih znamenki ovisi o prevoditelju i o računalu. U većini slučajeva broj točnih znamenki i dozvoljene pogreške pri osnovnim aritmetičkim operacijama ravnaju se po IEEE standardu 754 za binarni prikaz brojeva s pomičnim decimalnim zarezom.

```
float pi_tocniji    = 3.141592654;
float pi_manjeTocan = 3.1415927;
```

dati potpuno isti broj! Usput spomenimo da su broj π i ostale značajnije matematičke konstante definirani u biblioteci `cmath`, i to na veću točnost, tako da uključivanjem te datoteke možete prikratiti muke traženja njihovih vrijednosti po raznim priručnicima i srednjoškolskim udžbenicima.

Ako točnost na sedam decimalnih znamenki ne zadovoljava ili ako se koriste brojevi veći od 10^{38} ili manji od 10^{-38}, tada se umjesto `float` mogu koristiti brojevi s dvostrukom točnošću tipa `double` koji pokrivaju opseg vrijednosti od $1.7 \cdot 10^{-308}$ do $1.7 \cdot 10^{308}$, ili `long double` koji pokrivaju još širi opseg od $3.4 \cdot 10^{-4932}$ do $1.1 \cdot 10^{4932}$. Brojevi veći od 10^{38} vjerojatno će vam zatrebati samo za neke astronomske proračune, ili ako ćete se baviti burzovnim mešetarenjem. Realno gledano, brojevi manji od 10^{-38} neće vam gotovo nikada trebati, osim ako želite izračunati vjerojatnost da dobijete džekpot na lutriji ili da Milošević završi u Haagu (ovo je napisano u prosincu 2000.).

Iako `double` ima veću duljinu i zauzima više mjesta u memoriji, operacije s `double` podacima ne moraju nužno biti sporije nego s podacima tipa `float`. 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 ne treba previše ustručavati u korištenju podataka tipa `double`, naročito gdjegod nam je potrebna iole veća točnost.

U tablici 2.4 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. Duljina memorijskog prostora i opseg vrijednosti koje određeni tip varijable zauzima nisu standardizirani i 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 provjeriti veličinu pojedinog tipa, to možete učiniti i pomoću `sizeof` operatora:

```
cout << "Veličina cijelih brojeva: " << sizeof(int);
cout << "Veličina realnih brojeva: " << sizeof(float);
long double masaSunca = 2e30;
cout << "Veličina long double: " << sizeof(masaSunca);
```

[‡] Kod nekih prevoditelja navedeni podaci se nalaze u zaglavljima `limits.h` odnosno `float.h` (koja su naslijeđena iz jezika C), a ova su uključena u datoteke `climits`, odn. `cfloat` (glede imena vidi slično zapažanje vezano uz `iostream` biblioteku na str. 24)

Tablica 2.4. Ugrađeni brojevni tipovi, njihove tipične duljine i rasponi vrijednosti

tip konstante	bajtova	raspon vrijednosti	točnost
`char`	1	–128 do 127	
`short int`	2	–32768 do 32767	
`int`	2 (4)	–32768 do 32767 –2147483648 do 2147483647	
`long int`	4	–2147483648 do 2147483647	
`float`	4	$-3{,}4\cdot10^{38}$ do $-3{,}4\cdot10^{-38}$ i $3{,}4\cdot10^{-38}$ do $3{,}4\cdot10^{38}$	7 dec. znamenki
`double`	8	$-1{,}7\cdot10^{308}$ do $-1{,}7\cdot10^{-308}$ i $1{,}7\cdot10^{-308}$ do $1{,}7\cdot10^{308}$	15 dec. znamenki
`long double`	10	$-1{,}1\cdot10^{4932}$ do $-3{,}4\cdot10^{-4932}$ i $3{,}4\cdot10^{-4932}$ do $1{,}1\cdot10^{4932}$	18 dec. znamenki

Gornje naredbe će ispisati broj bajtova koliko ih zauzimaju podaci tipa `int`, `float`, odnosno `double`. Najveće i najmanje vrijednosti možete vidjeti ispisom odgovarajućih konstanti iz prije navedenih `climits` i `cfloat` datoteka:

```
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 << "Najmanji double: " << DBL_MIN;
```

Kada ćete isprobavati gornji kôd, nemojte zaboraviti uključiti zaglavlja `climits` i `cfloat`!

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

```
unsigned int i = 40000;
unsigned long int li;
unsigned long double BrojZvijezdaUSvemiru;
```

U tom slučaju će prevoditelj bit za predznak upotrijebiti 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 broj 65535 (ili 4294967295, ovisno o broju bajtova koji se koriste za pohranjivanje `int` brojeva). Uočimo da je taj broj za 1 manji od najvećeg dozvoljenog `unsigned int` broja 65536 (odnosno 4294967296). Zanimljivo da prevoditelj za gornji kôd neće javiti 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 čekovne iskaznice, a svoj bijes iskalite na računalu, ni krivom ni dužnom.

2.4.2. Aritmetički operatori

Da bismo objekte u programu mogli mijenjati, na njih treba primijeniti odgovarajuće operacije. Za ugrađene tipove podataka definirani su osnovni operatori, poput zbrajanja, oduzimanja, množenja i dijeljenja (vidi tablicu 2.5). 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;                          // uveća za 1
cout << i << endl;            // ispisuje 1
```

Tablica 2.5. 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

```
--i;                              // umanji za 1
cout << i << endl;                // ispisuje 0
```

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, a tek onda uveća
cout << i << endl;          // vidi, stvarno je uvećao!
```

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

```
float a = 2.;
float 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 kao rezultat vraća ostatak dijeljenja dva cijela broja:

```
int i = 6;
int j = 4;
cout << (i % j) << endl;    // ispisuje 2
```

On se vrlo često koristi za ispitivanje djeljivosti cijelih brojeva: ako su brojevi djeljivi, ostatak nakon dijeljenja će biti nula.

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

Valja uočiti dva suštinska problema vezana uz aritmetičke operatore. Prvi problem vezan je uz pojavu numeričkog preljeva, kada uslijed neke operacije rezultat nadmaši opseg koji dotični tip objekta pokriva. Drugi problem sadržan je u pitanju "kakvog će tipa biti rezultat binarne operacije s dva broja različitih tipova?".

Razmotrimo pojavu brojčanog preljeva. Donje naredbe će prvo ispisati najveći broj tipa `int`. Izvođenjem naredbe u trećem retku, na zaslonu računala će se umjesto tog broja uvećanog za 1 ispisati najveći negativni `int` (–32768 ili –2147483648)!

```
int i = INT_MAX;
cout << i << endl;
cout << (++i) << endl;
```

Uzrok tome je preljev podataka do kojeg došlo zbog toga što očekivani rezultat više ne stane u bitove predviđene za `int` varijablu. Podaci koji su se "prelili" ušli su u bit za predznak (zato je rezultat negativan), a raspored preostalih bitova daje broj koji odgovara upravo onom što nam je računalo ispisalo. Slično će se dogoditi oduzmete li od najvećeg negativnog broja neki broj. Ovo se može ilustrirati brojevnom kružnicom na kojoj zbrajanje odgovara kretanju po kružnici u smjeru kazaljke na satu, a oduzimanje odgovara kretanju u suprotnom smjeru (slika 2.3).

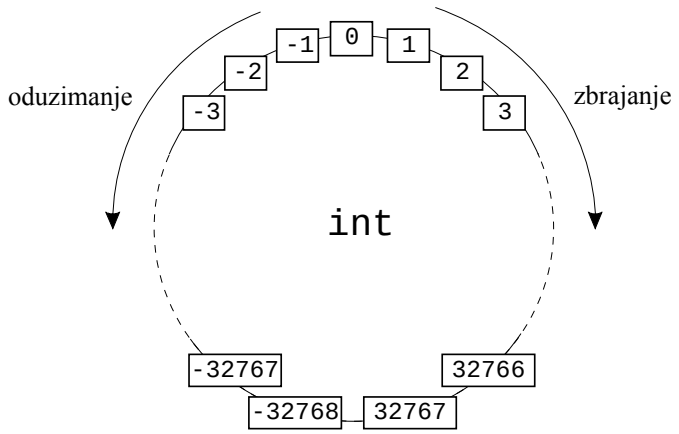
Ako postoji mogućnost pojave numeričkog preljeva, tada deklarirajte varijablu s većim opsegom – u gornjem primjeru umjesto `int` uporabite `long int`.

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 preljeva došlo "hardverskom" intervencijom prethodnog vlasnika, dok u programu do preljeva obično dolazi zbog nepažnje programera pri izradi programa.

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 = i - 2;
int AritSred = (prvi + drugi) / 2;
```

Zagrade oko zbroja su neophodne, jer dijeljenje ima viši prioritet od zbrajanja (o hierarhiji operacija govorit ćemo u odjeljku 2.7) – da zagrada nema, s 2 bi se dijelio



Slika 2.3. Prikaz preljeva na brojevnoj kružnici

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. No, prepišemo li zadnju naredbu na drugi način:

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

opasnost od preljeva ćemo otkloniti.

Slično vrijedi i za realne brojeve (tipa `float` i `double`). 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 na drugim prevoditeljima.

Drugo pitanje vezano na aritmetičke operatore koje se nameće jest kakvog će biti tipa rezultat binarne operacije na dva broja. 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/IEC1998]:

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. Potom, ako je jedan od operanada `unsigned long`, tada se i drugi operand pretvara u `unsigned long`.
6. U protivnom, ako je jedan od operanada tipa `long int`, a drugi operand tipa `unsigned int`, ako `long int` može obuhvatiti sve vrijednosti `unsigned int`, `unsigned int` se pretvara u `long int`; inače se oba operanda pretvaraju u `unsigned long int`.
7. Inače, ako je jedan od operanada tipa `long`, tada se i drugi operand pretvara u `long`.
8. Inače, ako je jedan od operanada `unsigned`, tada se i drugi operand pretvara u `unsigned`.

Na primjer, izvođenje kôda

```
int i = 3;
float a = 0.5;
cout << (a * i) << endl;
```

uzrokovat će ispis broja `1.5` na zaslonu, jer je od tipova `int` i `float` složeniji tip `float`. Cjelobrojnu varijablu `i` prevoditelj prije množenja pretvara u `float` (prema pravilu u točki 3), tako da se provodi množenje dva broja s pomičnom točkom, pa je i rezultat tipa `float`. 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;
float 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 `float` 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* `float` *na Dijeljenje stiže!*). Budući da su obje 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:

```
float 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 s pomičnom decimalnom točkom (`float` i `double`) pisati s decimalnom točkom, čak i kada nemaju decimalnih mjesta.

Evo pravilnog načina da se provede željeno dijeljenje:

```
float 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 `float` 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 `float` (ili još bolje, na `double`), na takve modificirane operande primijenio željeni operator, a dobiveni rezultat se pretvorio u zajednički tip. U tom slučaju programer uopće ne bi trebao paziti kojeg su tipa operandi – rezultat bi uvijek bio korektan (osim ako nemate procesor s pogreškom, što i nije nemoguće). Nedostatak ovakvog pristupa jest njegova neefikasnost. Zamislimo da treba zbrojiti dva broja tipa `int`! U gornjem "programer-ne-treba-paziti" pristupu, izvedbeni kôd dobiven nakon prevođenja trebao bi prvo jedan cjelobrojni operand pretvoriti u `float`. 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. Već sada uočavamo dvije operacije pretvorbe tipa koje kod izravnog zbrajanja cijelih brojeva ne postoje! Štoviše, samo zbrajanje se provodi na dva `float`-a, koji u memoriji zauzimaju veći prostor od `int`-a. Takvo zbrajanje iziskivat će puno više strojnih instrukcija i strojnog vremena, ne samo zbog veće duljine `float` brojeva u memoriji, već i zbog složenijeg načina na koji su oni pohranjeni. 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.

Zadatak. *Što će ispisati sljedeće naredbe:*

```
int a = 10;
float 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):*

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

2.4.3. 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;
float TocniKvocijent = static_cast<float>(Brojnik) /
                       static_cast<float>(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 `float` prije nego što se one međusobno podijele, tako da je krajni rezultat korektan. Naravno, bilo bi dovoljno operator dodjele tipa primijeniti samo na jedan operand – prema pravilima aritmetičke pretvorbe i drugi bi operand bio sveden na `float` tip. Da ne bi bilo zabune, same varijable `Brojnik` i `Nazivnik` i nadalje ostaju tipa `int`, tako da će njihovo naknadno dijeljenje

```
float OpetKriviKvocijent = Brojnik / Nazivnik;
```

opet kao rezultat dati kvocijent cjelobrojnog dijeljenja.

Korištenje operatora dodjele tipa je najčešće pokazatelj lošeg dizajna (*"Zašto su `Brojnik` i `Nazivnik` deklarirani kao `int` ako želimo s njima provesti `float` 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 4. poglavlju);
- kada želimo izbjeći standardne 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 – ovaj problem bit će obrađen u poglavlju 8 posvećenom nasljeđivanju)

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

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

```
float TocniKvocijent = (float)Brojnik / (float)Nazivnik;
```

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

```
float TocniKvocijent2 = float(Brojnik) / float(Nazivnik);
```

Iako su ova dva oblika dodjele tipa u potpunosti podržana standardom, njihova upotreba se ne preporučuje.

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

```
float a = 3.1415926
cout << "Cijeli dio broja a: " << static_cast<int>(a) << endl;
```

O operatorima dodjele tipa bit će još detaljno govora u 12. 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<float>(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:*

```
float a = 2.71;
float b = static_cast<int>a;
cout << b << endl;
```

2.4.4. 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
```

Češće se za specificiranje konstanti koriste *sufiksi*, posebni znakovi kojima se eksplicitno određuje tip brojčane konstante (vidi tablicu 2.6). Tako će sufiks `l`, odnosno `L` cjelobrojnu konstantu pretvoriti u `long`, a konstantu s decimalnom točkom u `double`:

```
long HrvatskiDugovi = 3457630455475571952525L;
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
```

Sufiks `f`, odnosno `F` će konstantu s decimalnom točkom pretvoriti u `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. Također, sufiksi `L` (`l`) i `U` (`u`) za cjelobrojne podatke mogu se međusobno kombinirati u bilo kojem redoslijedu (`LU`, `UL`, `Ul`, `uL`, `ul`, `lu`...) – rezultat će uvijek biti `unsigned long int`.

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

Tablica 2.6. Djelovanje sufiksa na brojčane konstante

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

2.4.5. 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...:

```
float Opseg = 2. * r * 3.14159265359;
float 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;
float Opseg = 2. * r * pi;
float 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, ako se (kojim slučajem, jednog dana) promijeni vrijednost broja π, bit će lakše ispraviti ga kada je definiran samo na jednom mjestu, nego raditi pretraživanja i zamjene brojeva po cijelom izvornom kôdu.

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` kojim se prevoditelju daje na znanje da varijabla mora biti 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 se ove zamjene neće odraziti u izvornom kôdu i on će nakon prevođenja ostati nepromijenjen.

Na prvi pogled nema razlike između pristupa – oba pristupa će osigurati prijavu pogreške pri pokušaju promjene konstante. Razlika postaje očita tek kada pokušate program ispraviti koristeći program za simboličko lociranje pogrešaka (engl. *debugger*, doslovni prijevod bio bi *istjerivač stjenica*, odnosno *stjeničji terminator*). 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. U protivnom ćemo dobiti poruku o pogreški:

```
const float 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:

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

2.4.6. Kvalifikator `volatile`

U prethodnom odsječku smo deklarirali konstantne objekte čime smo ih zaštitili od neovlaštenih promjena. Svi ostali objekti su promjenjivi (engl. *volatile*). 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, ako razvijamo sistemski program, vrijednost memorijske adrese se može promijeniti unutar obrade *prekida* (engl. *interrupt*) – time prekidna rutina može signalizirati programu da je određeni uvjet zadovoljen. U tom slučaju prevoditelj ne smije optimizirati pristup navedenoj varijabli. Na primjer:

```
int izadjiVan = 0;
while (!izadjiVan) {
    // čekaj dok se vrijednost izadjiVan ne postavi na 1
}
```

U gornjem primjeru prevoditelj analizom petlje može zaključiti da se varijabla `izadjiVan` ne mijenja unutar petlje, te će optimizirati izvođenje tako da se vrijednost varijable `izadjiVan` uopće ne testira. 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;
```

2.4.7. Pobrojenja

Ponekad su varijable u programu elementi pojmovnih skupova, tako da je takvim skupovima i njihovim elementima zgodno pridijeliti lakopamtljiva imena. Za takve slučajeve obično se koriste *pobrojani tipovi* (engl. *enumerated types, enumeration*):

```
enum dani {ponedjeljak, utorak, srijeda,
           cetvrtak, petak, subota, nedjelja};
```

Ovom deklaracijom uvodi se novi tip podataka `dani`, te sedam nepromjenjivih identifikatora (`ponedjeljak`, `utorak`,...) toga tipa. Prvom identifikatoru prevoditelj pridjeljuje vrijednost 0, drugom 1, itd. Sada možemo definirati varijablu tipa `dani`, te joj pridružiti neku od vrijednosti iz niza:

```
dani HvalaBoguDanasJe = petak;
dani ZakajJaNeVolim = ponedjeljak;
```

Naredbom

```
cout << HvalaBoguDanasJe << endl;
```

na zaslonu se ispisuje cjelobrojni ekvivalent za `petak`, tj. broj 4.

Varijable tipa `dani` mogli smo deklarirati i neposredno uz definiciju pobrojanog tipa:

```
enum dani{ponedjeljak, utorak, srijeda, cetvrtak,
          petak, subota, nedjelja} ThankGodItIs, SunnyDay;

ThankGodItIs = petak;
SunnyDay = nedjelja;
```

U pobrojanim nizovima podrazumijevana vrijednost prve varijable je 0. Želimo li da niz počinje nekom drugom vrijednošću, treba eksplicitno pridijeliti željenu vrijednost:

```
enum dani {ponedjeljak = 1, utorak, srijeda,
           cetvrtak, petak, subota, nedjelja};
```

Identifikator `utorak` poprima vrijednost 2, `srijeda` 3, itd. U ovom slučaju, kôd

```
dani ZecUvijekDolaziU = nedjelja;
cout << ZecUvijekDolaziU << endl;
```

na zaslonu ispisuje broj 7.

Eksplicitno pridjeljivanje može se primijeniti i na bilo koji od ostalih članova, s time da će slijedeći član, ako njegova vrijednost nije eksplicitno definirana, imati za 1 veću vrijednost:

```
enum likovi {kruznica = 0,
             trokut = 3,
             pravokutnik = 4,
             kvadrat = 4,
             cetverokut = 4,
             peterokut,
             sedmerokut = cetverokut + 3};

cout << kruznica << endl;           // ispisuje 0
cout << trokut << endl;             // ispisuje 3
cout << kvadrat << endl;            // ispisuje 4
cout << peterokut << endl;          // ispisuje 5
cout << sedmerokut << endl;         // ispisuje 7
```

Ako nam ne treba više varijabli tog tipa, ime tipa iza riječi `enum` može se izostaviti:

```
enum {NE = 0, DA} YesMyBabyNo, ShouldIStayOrShouldIGo;
enum {TRUE = 1, FALSE = 0};
```

Pobrojane vrijednosti mogu se koristiti u aritmetičkim izrazima. U takvim slučajevima se prvo provodi *cjelobrojna promocija* (engl. *integral promotion*) pobrojane vrijednosti – ona se pretvara u prvi mogući cjelobrojni tip naveden u slijedu: `int`, `unsigned int`, `long` ili `unsigned long`. Na primjer:

```
enum {DVOSTRUKI = 2, TROSTRUKI};
int i = 5;
float pi = 3.14159;
```

```
cout << DVOSTRUKI * i << endl;          // ispisuje 10
cout << TROSTRUKI * pi << endl;         // ispisuje 9.42477
```

Međutim, pobrojanim tipovima ne mogu se pridruživati cjelobrojne vrijednosti, pa će prevoditelj za sljedeći primjer javiti pogrešku:

```
dani VelikiPetak = petak;
dani DolaziZec = VelikiPetak + 2;       // pogreška
```

Prilikom zbrajanja pobrojeni tip `VelikiPetak` svodi se na tip zajednički sa cijelim brojem 2 (a to je `int`). Dobiveni rezultat tipa `int` treba pridružiti pobrojenom tipu, što rezultira pogreškom prilikom prevođenja[†]. Da bi gornja operacija "prošla", trebamo operatorom dodjele tipa taj int pretvoriti u tip dani:

```
dani DolaziZec = static_cast<dani>(VelikiPetak + 2);    // OK!
```

2.4.8. Logički tipovi i operatori

Logički podaci su takvi podaci koji 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*)[‡]:

```
bool JeLiDanasNedjelja = true;
bool SloboZeliIzZemlje = false;
bool NjofraMozeIzZemlje = 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. Ovakvo predstavljanje logičkih podataka je vrlo često, budući da je tip `bool` vrlo kasno uveden u standard jezika C++.

[†] Programski jezik C dozvoljava da se pobrojanom tipu pridruži `int` vrijednost. Zato, radi prenosivosti kôda pisanog u programskom jeziku C, te kôda pisanog u starijim varijantama jezika C++, ANSI/ISO C++ standard dozvoljava da se u nekim implementacijama prevoditelja omogući to pridruživanje, uz napomenu da to može prouzročiti neželjene efekte.

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

Tablica 2.7. Logički operatori

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

Za logičke podatke definirana su svega tri operatora: `!` (*logička negacija*), `&&` (*logički i*), te `||` (*logički ili*)† (tablica 2.7). 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 2.8 dani su rezultati logičkog *i* za sve moguće vrijednosti oba operanda. Logički *ili* daje istinu ako je bilo koji od operanada istinit (vidi tablicu 2.9).

Tablica 2.8. 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)

Razmotrimo primjenu logičkih operatora na sljedećem primjeru:

```
enum Logicki{NEISTINA, ISTINA, DRUGAISTINA = 124};
Logicki a = NEISTINA;
Logicki b = ISTINA;
Logicki c = DRUGAISTINA;

cout << "a = " << a << ", b = " << b
     << ", c = " << c << endl;
cout << "Suprotno od a = " << !a << endl;
cout << "Suprotno od b = " << !b << endl;
cout << "Suprotno od c = " << !c << endl;
```

Tablica 2.9. 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)

† | je znak koji je na tipkovnici označen prekinutom crtom ¦. Na *MS Windowsima* taj se znak, uz hrvatski raspored tipkovnice, dobiva istovremenim pritiskom na tipke <Ctrl>,<Alt> i <W>

```
cout << "a .i. b = " << (a && b) << endl;
cout << "a .ili. c = " << (a || c) << endl;
```

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*.

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

Budući da su logički podaci tipa bool dosta kasno uvršteni u standard C++ jezika, stariji prevoditelji ne podržavaju taj tip podataka. Želite li na starijem prevoditelju prevesti program koji je pisan u novoj verziji jezika, tip bool možete simulirati tako da na početak programa dodate sljedeće pobrojenje:

```
enum bool {false, true};
```

2.4.9. Poredbeni operatori

Osim aritmetičkih operacija, jezik C++ omogućava i usporedbe dva broja (vidi tablicu 2.10). 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

```
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 brojevi 1, 1, 0, 0, 0 i 1.

Tablica 2.10. 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

Poredbeni operatori (ponekad nazvani i *relacijski operatori* od engl. *relational operators*) se koriste pretežito 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 3.

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 efekta.

2.4.10. 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 2.11). Na primjer, kôd

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

Tablica 2.11. 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

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

pri čemu se u biti uspoređuju njihovi brojčani kôdovi u nekom od standarda. Najrašireniji je ASCII niz (kratica od *American Standard Code for Information Interchange*) u kojem su svim ispisivim znakovima, brojevima i slovima engleskog alfabeta pridruženi brojevi od 32 do 127, dok specijalni znakovi imaju kôdove od 0 do 31 uključivo. U prethodnom primjeru, 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. 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, na znakove se mogu primjenjivati i svi aritmetički operatori. Budući da se pritom znakovi cjelobrojnom promocijom pretvaraju u cijele brojeve, za njih vrijede ista pravila kao i za operacije sa cijelim brojevima. Ilustrirajmo to sljedećim primjerom:

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

Za pohranjivanje znakova čiji kôd ne stane u tip `char` može se koristiti tip `wchar_t`. Razjasnimo ukratko problem pohranjivanja znakova.

`char` tip podataka je 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 znakove engleskog alfabeta i specijalne znakove (interpunkcija, kontrolni kôdovi) – za nacionalne znakove neengleskih jezika preostali su kodovi od 127 do 255[†]. Taj skup je dovoljan da obuhvati jezično specifične znakove samo za nekoliko jezika. Tako primjerice prema ISO-8859-1 kôdiranju taj skup popunjavaju znakovi zapadnoeuropskih pisama, dok ga prema ISO-8859-2 kôdiranju popunjavaju znakovi srednjeeuropskih jezika (uključujući Hrvatski).

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 ISO-8859-2 standardu nekom Mađaru, on će sve naše znakove vidjeti korektno, jer se i u Mađarskoj koristi taj standard. Međutim, pošaljete li taj dokument nekome Dancu (ili bilo kome u zapadnoj Europi), njemu će se umjesto naših slova pojaviti potpuno drugačiji znakovi (npr. umjesto slova "ć", on će na ekranu imati slovo "æ"). Uspije li

[†] Doduše, postoje neki lokalni 7-bitni "standardi" koji znakove unutar ASCII skupa nadomještaju jezički specifičnim znakovima, no oni se danas uglavnom napuštaju.

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 su standard uključeni znakovi svih svjetskih pisama (uključujući ćirilicu, arapsko, japansko, kinesko pismo) pa će zbog toga bilo koji dokument pisan u Unicodu biti korektno prikazan u bilo kojem dijelu svijeta. Da bi se obuhvatili svi znakovi, kôd za pojedine znakove trebalo je proširit na 2 bajta (što daje mjesto za 65536 različitih znakova). Korištenjem Unicoda u programima olakšava se izrada programa za jezično različita područja.

Na žalost, da bi se Unicode mogao koristiti, treba ga podržavati operacijski sustav koji se vrti na računalu i koji je zadužen za pretvorbu brojčanih kôdova u grafički prikaz na zaslonu. U ovom trenutku praktički ne postoji operacijski sustav koji bi izravno podržavao Unicode, već se obavljaju translacije iz Unicoda u neki lokalni 8-bitni standard i onda se takvi znakovi prikazuju.

`wchar_t` tip omogućava pohranjivanje znakova po Unicode standardu pa tako možemo unutar istog programa kombinirati slova različitih pisama (naravno, uz pretpostavku operacijski sustav podržava Unicode):

```
wchar_t DomaceJeDomace = L'ć';
wchar_t OvakoToPisuPrekoDrine = L'ħ';
wchar_t NegdjeDalje = L'ŋ';
wchar t JosDalje = L'¥';
```

Prefix `L` naznačava da je znak unutar apostrofa tipa `wchar_t`. Duljina tipa `wchar_t` nije definirana standardom jezika C++; ona mora biti jednaka duljini nekog cjelobrojnog tipa. Na prevoditelju koji smo koristili pri pisanju knjige ona je iznosila dva bajta, tako da smo u jedan znak tipa `wchar_t` mogli smjestiti dva "obična" znaka (uočite da sada nema prefiksa `L` ispred apostrofa):

```
wchar_t podebljiZnak = 'ab';
```

2.4.11. Bitovni operatori

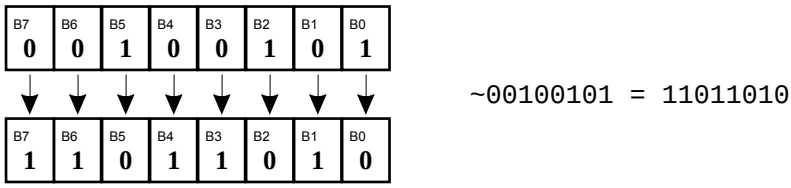
Velika prednost jezika C++ je što omogućava izravne operacije na pojedinim bitovima podataka. Na raspolaganju je šest operatora: bitovni komplement, bitovni *i*, bitovni *ili*, bitovni *isključivi ili*, bitovni pomak udesno i bitovni pomak ulijevo (tablica 2.12). Valja napomenuti da su bitovni operatori definirani samo za cjelobrojne (`int`, `long int`) operande.

Tablica 2.12. Bitovni operatori

`~i`	komplement
`i & j`	binarni i
`i | j`	binarni ili
`i ^ j`	isključivi ili
`i << n`	pomakni ulijevo
`i >> n`	pomakni udesno

Operator komplementiranja `~` (tilda) je unarni operator koji mijenja stanja pojedinih bitova, tj. sve bitove koji su jednaki nuli

postavlja u 1, a sve bitove koji su 1 postavlja u 0. Tako će binarni broj 00100101_2 komplementiranjem prijeći u 11011010_2, kako je prikazano na slici 2.4.



Slika 2.4. Bitovni komplement

Zbog toga će se izvođenjem kôda

```
unsigned char a = 0x25;             // 00100101 u heksadek.prikazu
unsigned char b = ~a;               // pridruži bitovni komplement

cout << hex << static_cast<int>(a) << endl;
                                    // ispisuje a u hex-formatu
cout << static_cast<int>(b) << endl;
                                    // ispisuje b u hex-formatu
```

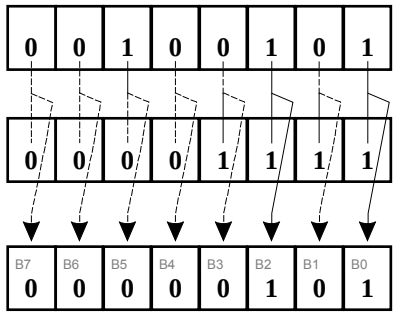
ispisati heksadekadski broj `25` i njegov bitovni komplement, heksadekadski broj `da` (tj. 11011010 u binarnom prikazu). U gornjem kôdu uočavamo `hex` manipulator† za izlazni tok, kojim smo osigurali da ispis svih brojeva koji slijede bude u heksadekadskom formatu. Manipulator `hex` definiran je u `iostream` biblioteci. Operator dodjele tipa `static_cast<int>` pri ispisu je neophodan, jer bi se bez njega umjesto heksadekadskih brojeva ispisali pripadajući ASCII znakovi.

Poneki čitatelj će se sigurno upitati zašto smo u gornjem primjeru varijable `a` i `b` deklarirali kao `unsigned char`. Varijabla tipa `char` ima duljinu samo jednog bajta, tj. zauzima 8 bitova, tako da će varijabla `a` nakon pridruživanja vrijednosti `0x25` u memoriji imati zaista oblik 00100101. Prilikom komplementiranja, ona se proširuje u `int`, tj. dodaje joj se još jedan bajt nula (ako `int` brojevi zauzimaju dva bajta), tako da ona postaje 00000000 00100101. Komplement tog broja je 11111111 11011010 ($FFDA_{16}$), ali kako se taj rezultat pridružuje `unsigned char` varijabli `b`, viši bajt (lijevih 8 bitova) se gubi, a preostaje samo niži bajt (DA_{16}). Da smo varijablu `b` deklarirali kao `int`, lijevih 8 bitova bi ostalo, te bismo na zaslonu dobili ispis heksadekadskog broja `ffda`. Pažljivi čitatelj će odmah primijetiti da je za varijablu `a` potpuno svejedno da li je deklarirana kao `char` ili kao `int` – ona se ionako prije komplementiranja pretvara u `int`. Nju smo deklarirali kao `unsigned char` samo radi dosljednosti s varijablom `b`.

Zadatak. *Razmislite (i eventualno provjerite) što se dobiva izvođenjem gornjeg kôda ako je* `int` *duljine 4 bajta, a varijable* `a` *i* `b` *se definiraju da su tipa* `int`*.*

† Manipulatori će biti detaljno obrađeni u poglavlju *15. Ulazni i izlazni tokovi*.

Operator & (bitovni *i*) postavlja u 1 samo one bitove koji su kod oba operanda jednaki 1 (slika 2.5); matematičkom terminologijom rečeno, traži se *presjek* bitova dvaju brojeva.




```
00100101 & 00001111 = 00000101
```

Slika 2.5. Bitovni operator *i*

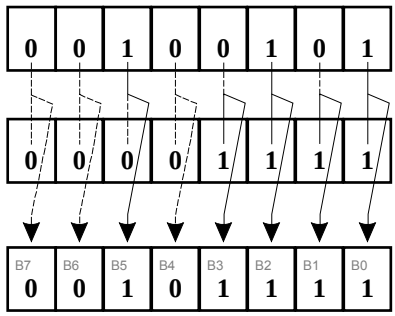
Bitovni *i* se najčešće koristi kada se žele pojedini bitovi broja postaviti u 0 (*obrisati*) ili ako se žele izdvojiti samo neki bitovi broja. Drugi operand je pritom maska koja određuje koji će se bitovi postaviti u nulu, odnosno koji će se bitovi izlučiti. Za postavljanje određenih bitova u nulu svi bitovi maske na tim mjestima moraju biti jednaki 0. Ako u gornjem primjeru uzmemo da je maska donji operand, tj. 00001111_2, tada možemo reći da smo kao rezultat dobili gornji broj u kojem su četiri najznačajnija (tj. lijeva) bita obrisana. Četiri preostala bita su ostala nepromijenjena, jer maska na tim mjestima ima jedinice, iz čega odmah slijedi zaključak da u maski za izlučivanje svi bitovi koje želimo izlučiti iz prvog broja moraju biti jednaki 1. Ako u gornjem primjeru uzmemo da je maska donji operand, kao rezultat smo dobili stanja četiri najniža bita gornjeg operanda. Zbog svojstva komutativnosti možemo reći i da je prvi broj maska pomoću koje brišemo, odnosno vadimo određene bitove drugog broja. Izvorni kôd za gornju operaciju mogli bismo napisati kao:

```
int a = 0x0025;
int maska = 0x000f;

cout << hex << (a & maska) << endl;
```

Izvođenjem se ispisuje broj 5.

Operator | (bitovni *ili*) postavlja u 1 sve one bitove koji su u bilo kojem od operanada jednaki 1 (slika 2.6); matematičkim rječnikom traži se *unija* bitova dvaju brojeva. Bitovni *ili* se najčešće koristi za postavljanje određenih bitova u 1. Drugi operand je pritom maska koja mora imati 1 na onim mjestima koja želimo u broju postaviti. U gornjem primjeru postavili smo sva četiri najniža bita lijevog broja u 1. Stoga će niz naredbi:




```
00100101 | 00001111 = 00101111
```

Slika 2.6. Bitovni operator *ili*

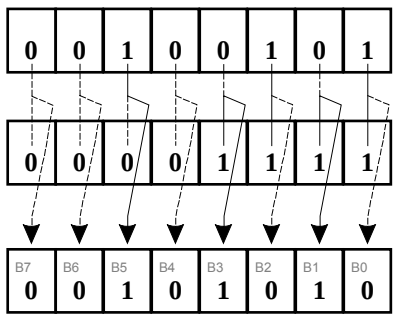
```
int a = 0x0025;
int maska = 0x000f;

cout << hex << (a | maska) << endl;
```

na zaslonu ispisati heksadekadski broj 2F.

Operator ^ (*isključivi ili*, ponekad nazvan i *ex-ili*, engl. *exclusive or*) postavlja u 1 samo one bitove koji su kod oba operanda međusobno različiti (slika 2.7). On se uglavnom koristi kada se žele promijeniti stanja pojedinih bitova. Odgovarajuća maska mora u tom slučaju imati 1 na mjestima bitova koje želimo promijeniti:

```
int a = 0x0025;
int maska = 0x000f;
cout << hex << (a ^ maska) << endl;         // ispisuje 0x2a
```




```
00100101 ^ 00001111 = 00101010
```

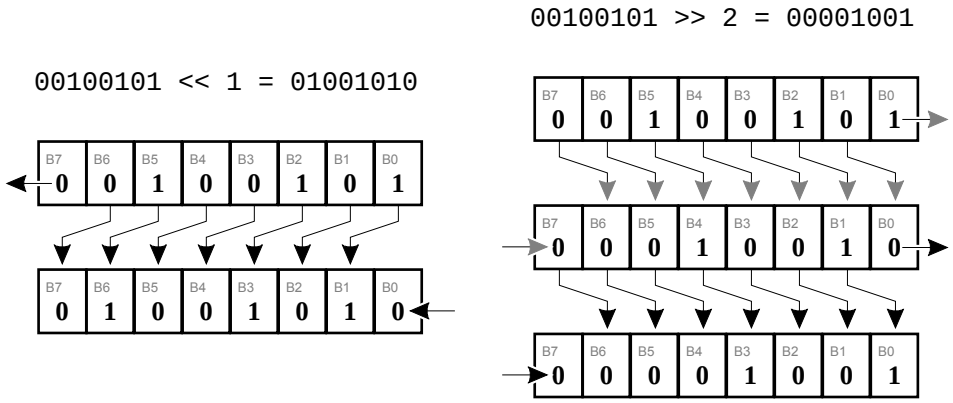
Slika 2.7. Bitovni operator *isključivo ili*

Zanimljivo je uočiti da ponovna primjena maske vraća broj u izvornu vrijednost:

```
cout << hex << (a ^ maska ^ maska) << endl;     // ispisuje 0x25
```

Ova činjenica često se koristi za jednostavnu zaštitu podataka ili kôda od nepoželjnog čitanja, odnosno korištenja: primjenom *isključivog ili* na podatke pomoću tajne maske podaci se šifriraju. Ponovnom primjenom *isključivog ili* s istom maskom podaci se vraćaju u izvorni oblik.

Operatori << (pomak ulijevo, engl. *shift left*) i >> (pomak udesno, engl. *shift right*) pomiču sve bitove lijevog operanda za broj mjesta određen desnim operandom, kako je prikazano na slici 2.8.



Slika 2.8. Bitovni pomaci ulijevo, odnosno udesno

Pri pomaku ulijevo najznačajniji (tj. krajnji lijevi) bitovi se gube, dok najmanje značajni bitovi poprimaju vrijednost 0. Slično, pri pomaku udesno, gube se najmanje značajni bitovi, a najznačajniji bitovi poprimaju vrijednost 0. Uočimo da pomak bitova u cijelim brojevima za jedno mjesto ulijevo odgovara množenju broja s 2 – situacija je potpuno identična dodavanju nula iza dekadskog broja, samo što je kod dekadskog broja baza 10, a u binarnoj notaciji je baza 2. Slično, pomak bitova za jedno mjesto udesno odgovara dijeljenju cijelog broja s 2:

```
int a = 20;
cout << (a << 1) << endl;       // pomak za 1 bit ulijevo -
                                //   ispisuje 40
cout << (a >> 2) << endl;       // pomak za 2 bita udesno -
                                //   ispisuje 5
```

Međutim, pri korištenju pomaka valja biti krajnje oprezan, jer se kod cjelobrojnih konstanti s predznakom najviši bit koristi za predznak. Ako je broj negativan, tada se pomakom ulijevo bit predznaka će se izgubiti, a na njegovo mjesto će doći najviša binarna znamenka. Kod pomaka udesno bit predznaka ulazi na mjesto najviše znamenke. Istina, neki prevoditelji vode računa o bitu predznaka, ali se zbog prenosivosti kôda ne treba previše oslanjati na to.

Mogućnost izravnog pristupa pojedinim bitovima često se koristi za stvaranje skupova logičkih varijabli pohranjenih u jednom cijelom broju, pri čemu pojedini bitovi tog broja predstavljaju zasebne logičke varijable. Takvim pristupom se štedi na memoriji, jer umjesto da svaka logička varijabla zauzima zasebne bajtove, u jednom je bajtu pohranjeno osam logičkih varijabli. Ilustrirajmo to primjerom u kojem se definiraju parametri za prijenos podataka preko serijskog priključka na računalu. Radi lakšeg praćenja, izvorni kôd ćemo raščlaniti na segmente, koje ćemo analizirati zasebno.

Za serijsku komunikaciju treba, osim brzine prijenosa izražene u baudima, definirati broj bitova po podatku (7 ili 8), broj stop-bitova (1 ili 2), te paritet (parni paritet, neparni paritet ili bez pariteta). Uzmimo da na raspolaganju imamo osam brzina prijenosa: 110, 150, 300, 600, 1200, 2400, 4800 i 9600 bauda. Svakoj toj brzini pridružit ćemo po jedan broj u nizu od 0 do 7:

```
enum {Baud110 = 0, Baud150, Baud300, Baud600,
      Baud1200, Baud2400, Baud4800, Baud9600};
```

Budući da imamo brojeve od 0 do 7, možemo ih smjestiti u tri najniža bita B0 - B2 (vidi sliku 2.9). Podatak o broju bitova po podatku pohranit ćemo u B3; ako se prenosi 7 bitova po podatku bit B3 je jednak 0, a ako se prenosi 8 bitova po podatku onda je 1. Binarni broj koji ima treći bit postavljen odgovara broju 08 u heksadekadskom prikazu ($0000\ 1000_2 = 08_{16}$):

```
enum {Bitova7 = 0x00, Bitova8 = 0x08};
```

Broj stop-bitova pohranit ćemo u B4 i B5. Ako se traži jedan stop-bit, tada će B4 biti 1, a za dva stop-bita B5 je jednak 1:

```
enum {StopB1 = 0x10, StopB2 = 0x20};
```

Konačno, podatke o paritetu ćemo pohraniti u dva najviša bita (B6 - B7):

```
enum {ParitetNjet = 0x00, ParitetNeparni = 0x40,
      ParitetParni = 0x80};
```

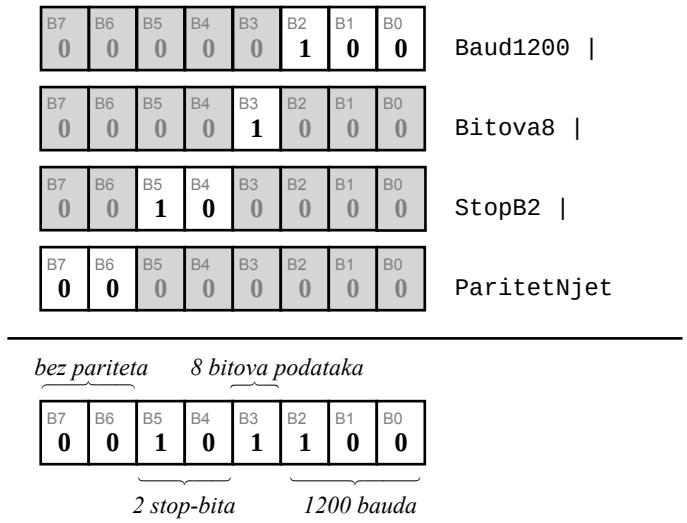
Uočimo da smo sva gornja pobrojenja mogli sažeti u jedno.

Želimo li sada definirati cjelobrojnu varijablu `SerCom` u kojoj će biti sadržani svi parametri, primijenit ćemo bitovni operator *ili*:

```
int SerCom = Baud1200 | Bitova8 | StopB2 | ParitetNjet;
```

kojim se određeni bitovi postavljaju u stanje 1. Rezultirajući raspored bitova prikazan je na slici 2.9.

Kako iz nekog zadanog bajta s parametrima izlučiti relevantne informacije? Za to trebamo na cijeli bajt primijeniti masku kojom ćemo odstraniti sve nerelevantne bitove. Na primjer, za brzinu prijenosa važni su nam samo bitovi B0 - B3, tako da ćemo



Slika 2.9. Primjer rasporeda bitovnih parametara za serijsku komunikaciju

bitovnim *i* operatorom s brojem koji u binarnom prikazu ima 1 na tim mjestima (tj. s 07 heksadekadsko), izlučiti podatak o brzini (vidi sliku 2.10):

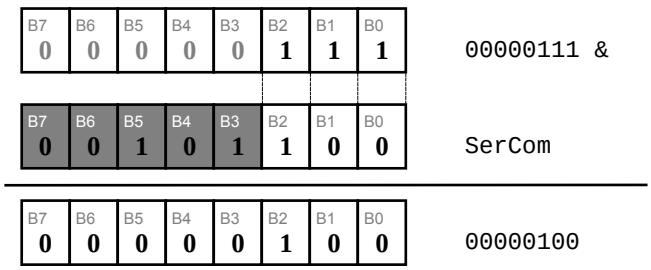
```
int Speed = SerCom & 0x07;              // dobiva se 4
```

Prenosi li se osam bitova podataka provjerit ćemo sljedećom bitovnom *ili* operacijom:

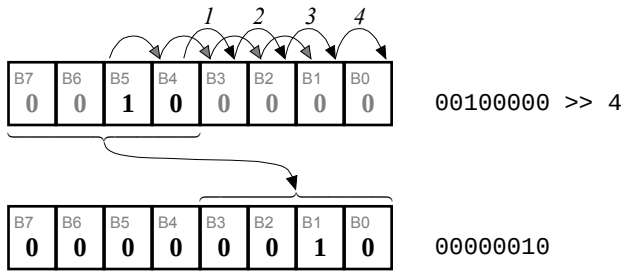
```
int JeLiOsamBita = SerCom & 0x08;       // dobiva se 8
```

budući da 8_{16} ima u binarnom prikazu 1 samo na mjestu B3. Ako je bit B3 postavljen, tj. ako ima vrijednost 1, varijabla `JeLiOsamBita` će poprimiti vrijednost dekadsko (i heksadekadsko!) 8; u protivnom će biti 0.

Analogno, za izlučivanje broja stop-bitova poslužit ćemo se maskom 30_{16} koja na mjestima B4 i B5 ima 1:



Slika 2.10. Primjena bitovne maske i operatora *i*



Slika 2.11. Pomak udesno za četiri bita

```
int BrojStopBita = SerCom & 0x30;       // dobiva se 32
```

U našem primjeru kao rezultat ćemo dobiti 32. Ako rezultantu bitovnu strukturu pomaknemo udesno za 4 mjesta tako da B4 i B5 dođu na B0 odnosno B1 (slika 2.11):

```
BrojStopBita = BrojStopBita >> 4;
```

dobit ćemo upravo broj stop-bitova, tj. broj 2.

Zadatak. *Napišite kôd u kojem će se* `int` *varijabla* `nekiBroj` *množiti s 4 pomakom bitova ulijevo. Prije pomaka pohranite bit predznaka u cjelobrojnu varijablu* `predznak`*, a nakon pomaka vratite predznak rezultatu! Napravite to isto i za dijeljenje s 2.*

2.4.12. 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 2.13 dani su svi operatori pridruživanja. Primjenu nekolicine operatora ilustrirat ćemo sljedećim kôdom

Tablica 2.13. 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.

2.4.13. 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 2.14). Uz to, za istu

Tablica 2.14. Alternativne oznake operatora

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

namjenu je u jeziku C++ dozvoljena i upotreba nizova od tri znaka (engl. *trigraph*), naslijeđenih iz programskog jezika C (tablica 2.15). Srećom, današnji operacijski sustavi za predstavljanje znakova koriste neki 8-bitni standard (npr. kodna stranica 852 pod DOS-om ili kodna stranica 1250 pod Windows-ima) pa omogućavaju istovremeno korištenje nacionalnih znakova i specijalnih znakova neophodnih za pisanje programa u jeziku C++, tako da većini korisnika tablice 2.14 i 2.15 neće nikada zatrebati. Ipak za ilustraciju pogledajmo kako bi izgledao neki program napisan pomoću alternativnih oznaka (čitatelju prepuštamo da "dešifrira" kôd):

```
%:include <iostream>
using namespace std;

int main() <%
    bool prva = true;
    bool druga = false;
    bool treca = prva and druga;
    cout << treca << endl;
    cout << not treca << endl;
    cout << prva or druga << endl;
    return 1;
%>
```

Tablica 2.15. Trigraf nizovi

trigraf	zamjena za	trigraf	zamjena za	trigraf	zamjena za
??=	#	??(	[	??<	{
??/	\	??)	]	??>	}
??'	^	??!	|	??-	~

2.4.14. Korisnički definirani tipovi i operatori

Osim standardno ugrađenih tipova podataka koje smo u prethodnim odjeljcima upoznali, 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 *razredi* koji omogućavaju uvođenje potpuno novih, *korisnički definirani tipova podataka*. Tako programer više nije sputan osnovnim tipovima koji su ugrađeni u jezik, već ih može po volji dopunjavati i definirati operacije na novostvorenim tipovima. Detaljnije o razredima bit će govora u poglavlju 6.

2.4.15. Deklaracija `typedef`

Ključna riječ `typedef` omogućava uvođenje novog imena za već postojeći ugrađeni ili korisnički definirani tip podataka. Na primjer, deklaracijom:

```
typedef float broj;
```

identifikator `broj` postaje sinonimom za tip `float`. Nakon gornje deklaracije "novopečeni" identifikator tipa `broj` može se ravnopravno koristiti u deklaracijama objekata:

```
broj pi = 3.14159;
```

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

```
float a = pi;
typedef float pliva;
pliva = pi;
```

Osnovni razlog za primjenu deklaracije `typedef` jesu jednostavnije 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 ponekad 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 bitno olakšati, ako u gornjem primjeru dodamo deklaraciju `typedef` za podatke čiji tip ćemo povremeno mijenjati:

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

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

```
typedef double brojevi;
```

Sada će svi podaci tipa `brojevi` biti prevedeni kao `double` podaci, bez potrebe daljnjih izmjena u izvornom kôdu.

Osim u ovakvim jednostavnim slučajevima, ključna riječ `typedef` se često koristi prilikom deklaracija pokazivača i referenci na objekte, pokazivača na funkcije i kod deklaracija polja. Naime, sintaksa kojom se ti tipovi opisuju može često biti vrlo složena. Zbog toga, ako često koristimo pokazivač na neki tip, programski kôd može postati vrlo nečitljiv. U tom slučaju, jednostavnije je pomoću ključne riječi `typedef` definirati novi tip, koji označava često korišteni tip "pokazivač na objekt". Takav tip postaje ravnopravan svim ostalim ugrađenim tipovima, te se može pojaviti na svim mjestima na kojima se može pojaviti ugrađeni tip: prilikom deklaracije objekata, specificiranja parametara funkcije, kao parametar `sizeof` operatoru i sl. Primjena `typedef` ključne riječi na takve tipove bit će objašnjena u kasnijim poglavljima.

2.5. 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, osim u smislu rezultata što ga daje `sizeof` operator; tako je `sizeof(char)` jednak 1. Naime, duljina bajta (broj bitova koji čine bajt) ovisi o arhitekturi računala. Mi ćemo u knjizi uglavnom podrazumijevati da bajt sadrži 8 bitova, što je najčešći slučaj u praksi.

Operand `sizeof` operatora 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
```

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

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

Operator `sizeof` se može primijeniti i na pokazivače, reference, polja, korisnički definirani razredi, strukture, unije 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.

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.

2.6. 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

```
i = 10, i + 5;
```

varijabli `i` 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 vrlo često koristi u sprezi s uvjetnim operatorom (poglavlje 3.3) te za razdvajanje više izraza u parametrima `for` petlje (poglavlje 3.5).

2.7. Hijerarhija i redoslijed izvođenja operatora

U matematici postoji 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. Ako dva susjedna operatora imaju isti prioritet, tada se operacije izvode prema slijedu izvođenja operatora. U tablici 2.16 na stranici 77 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;

    c = a---b;
    cout << "a = " << a << ", b = " << b << ", c = " << c << endl;
    return 1;
}
```

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).

Tablica 2.16. Hijerarhija operatora

operator	značenje	primjer
`::`	globalno područje	`::ime`
`::`	područje razreda	`razred::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(tip)`
`typeid`	identifikacija tipa tijekom izvođenja	`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)`
`++`	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) poljet	`delete[] pokazivac`
`()`	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)

Tablica 2.16 *(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*

Kao i u matematici, okruglim zagradama se može 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 množiti 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

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

Dobra je navada 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.

3. 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.

3.1. 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 21 i 25 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.

3.2. Grananje toka naredbom `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 nezavisna 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() {
    cout << "Upiši koeficijente pravca:" << endl;
    float a, b, c;                  // koeficijenti pravca
    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 ih

    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() {
    float a, b, c;

    cout << "Unesi koeficijente kvadratne jednadžbe:"
         << endl;
    cout << "a = ";
    cin >> a;
    cout << "b = ";
    cin >> b;
    cout << "c = ";
    cin >> c;

    float diskr = b * b - 4. * a * c;   // diskriminanta

    cout << "Jednadžba ima ";
    if (diskr == 0)
        cout << "dvostruki realni korijen." << endl;
    else if (diskr > 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() {
    float a, b, c;

    cout << "Unesi koeficijente kvadratne jednadzbe:"
         << endl;
    cout << "a = ";
    cin >> a;
    cout << "b = ";
    cin >> b;
    cout << "c = ";
    cin >> c;
    if (a) {
        float diskr = b * b - 4. * a * c;
        cout << "Jednadžba ima ";
        if (diskr == 0)
            cout << "dvostruki realni korijen." << endl;
        else if (diskr > 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 (pravilima pretvorbe vrijednost varijable svest će se na logičku istinu) i izvest će se 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.

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`, odnosno 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.

3.3. 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 = abs(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`). 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.*

3.4. Grananje toka naredbom `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
}
```

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`*. Nailaženjem 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 (algoritam je prepisan iz priručnika za jedan stari programirljivi kalkulator):

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

int main() {
    cout << "Upiši datum u formatu DD MM GGGG:";
    int dan, mjesec;
    long int godina;
    cin >> dan >> mjesec >> godina;

    long datum;
    if (mjesec < 3) {
        datum = 365 * godina + dan + 31 * (mjesec - 1)
                + (godina - 1) / 4
                - 3 * ((godina - 1) / 100 + 1) / 4;
    }
    else {
        // uočimo operator dodjele tipa static_cast<int>:
        datum = 365 * godina + dan + 31 * (mjesec - 1)
                - static_cast<int> (0.4 * mjesec + 2.3)
                + godina / 4 - 3 * (godina / 100 + 1) / 4;
    }

    cout << dan << "." << mjesec << "." << godina
         << ". pada u ";

    switch (datum % 7) {
      case 0:
        cout << "subotu." << endl;
        break;
      case 1:
        cout << "nedjelju." << endl;
        break;
      case 2:
        cout << "ponedjeljak." << endl;
        break;
      case 3:
        cout << "utorak." << endl;
        break;
      case 4:
        cout << "srijedu." << endl;
        break;
      case 5:
        cout << "četvrtak." << endl;
```

```
        break;
      default:
        cout << "petak." << endl;
    }
    return 0;
}
```

Algoritam se zasniva na cjelobrojnim dijeljenjima, tako da sve varijable treba deklarirati kao cjelobrojne. Štoviše, `godina` se mora definirati kao `long int`, jer se ona u računu množi s 365. U protivnom bi vrlo vjerojatno došlo do brojčanog preljeva, osim ako bismo se ograničili na datume iz života Kristovih suvremenika. Uočimo u gornjem kôdu operator dodjele tipa `static_cast<int>()`

```
/*...*/ static_cast<int>(0.4 * mjesec + 2.3)  /*...*/
```

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*).

U gornjem primjeru svaki `case` je zaključen `break` naredbom pa se netko može zapitati zašto je uopće neophodan `break` – zar ne bi bilo jednostavnije definirati da se program podrazumijevano izvršava samo do sljedeće naredbe `case`? Pogledajmo protuprimjer: u gornjem programu ćemo `switch` blok nadomjestiti sljedećim kôdom:

```
    switch (datum % 7) {
      case 0:
      case 1:
        cout << "dane weekenda." << endl;
        break;
      default:
        cout << "radne dane." << endl;
    }
```

Budući da odmah iza naredbe `case 0:` slijedi `case 1:`, blok naredbi do naredbe `break` će se izvršavati i za slučaj kada je rezultat izraza u `switch` jednak 0. `default` blok će se izvoditi za sve ostale slučajeve.

Zadatak. *U gornjem primjeru pomaknite `default` blok naredbi ispred `case` naredbi i pokrenite program. Nemojte zaboraviti dodati naredbu `break` na kraj `default` bloka! Što bi se dogodilo da taj `break` izostavite?*

Blok `default` može biti smješten bilo gdje unutar `switch` bloka, no zbog preglednosti je preporučljivo staviti ga na kraj. On se smije i izostaviti, ali ga je redovito zgodno imati da bi kroz njega program prošao za vrijednosti koje nisu obuhvaćene `case` blokovima. Ovo je naročito važno tijekom razvijanja programa, kada se u `default` blok može staviti naredba koja će ispisivati upozorenje da je *`cjelobrojni_izraz`* u `switch` poprimio neku nepredviđenu vrijednost.

Zadatak. *Zadnji primjer modificirajte tako da dodatno ispitujete da li zadani datum pada na neki blagdan (za početak ispitajte samo za 25.12.). Ako pada na blagdan ili*

ako pada u dane weekenda, ispišite da je taj dan neradni, inače da je radni. Uputa: na početku programa definirajte bool varijablu `DaLiJePraznik` *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.*

3.5. Petlja `for`

Često u programima treba ponavljati dijelove kôda. 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 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 računanje faktorijela (faktorijela od n je umnožak svih brojeva od 1 do n):

$$n! = 1 \cdot 2 \cdot \ldots \cdot (n-2) \cdot (n-1) \cdot n$$

Pogledajmo kôd:

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

int main() {
    int n;
    cout << "Upiši prirodni broj: ";        // manji od 13!?
    cin >> n;

    long int fjel = 1;
    for (int i = 2; 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. Na ulasku u petlju deklariramo brojač petlje, varijablu `i` tipa `int` te joj pridružujemo početnu vrijednost 2 (*`početni_izraz:`* `int i = 2`). 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`). Pri testiranju programa pazite da uneseni broj ne smije biti veći od 12, jer će inače doći do brojčanog preljeva varijable `fjel`.

Zanimljivo je uočiti što će se dogoditi ako za `n` unesemo brojeve manje od 2. Već pri prvom ulasku u petlju neće biti zadovoljen uvjet ponavljanja petlje te će odmah biti preskočeno tijelo petlje i ona se neće izvesti niti jednom! Ovo nam odgovara, jer je 1! = 1 i (po definiciji) 0! = 1. Naravno da smo petlju mogli napisati i na sljedeći način:

```
for (int i = n; i > 1; i--)
    fjel *= i;
```

Rezultat bi bio isti. Iskusni programer bi gornji program mogao napisati još sažetije (vidi poglavlje 3.11), no u ovom trenutku to nam nije bitno.

Može se dogoditi da je uvjet izvođenja uvijek zadovoljen, pa će se petlja izvesti neograničeni broj puta. Program će uletjeti u slijepu ulicu iz koje nema izlaska, osim pomoću tipki *Power* ili *Reset* na kućištu vašeg računala. Na primjer:

```
// ovaj primjer pokrećete na vlastitu odgovornost!
cout << "Beskonačna petlja";
for (int i = 5; i > 1; )
    cout << "a";
```

Varijabla `i` je uvijek veća od 1, tako da ako se i 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 bolje je definirati isključivo u izrazu prirasta.

U protivnom se lako može dogoditi da petlja postane beskonačna, poput sljedećeg primjera:

```
for (int 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 fjel;
    int i;
    for (i = 2, fjel = 1; i <= n; fjel *= i, i++) ;

    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 rezultata) obuhvatio u petlju. Sada je i deklaracija brojača prebačena ispred `for` naredbe, jer početni izraz ne trpi višestruke deklaracije. Da smo `for` naredbu napisali kao:

```
for (int i = 2, long fjel = 1; i <= n; fjel *= i, i++) ;
```

prevoditelj bi javio da je deklaracija varijable `i` okončana nepravilno, jer bi iza zareza, umjesto imena varijable naišao na ključnu riječ `long`.

Ako `for` naredba sadrži deklaraciju varijable, tada se područje te varijable prostire samo do kraja petlje[†] . Na primjer:

[†] Tijekom razvoja Standarda to pravilo se mijenjalo (zanimljivi članak o tome izašao je u časopisu *C++ Report* od veljače 1993, pod naslovom "Identifier Scope in C++ `for` statement" autora A. Koeniga). Tako će neki prevoditelji ostaviti varijablu `i` "živom" i iza `for` petlje.

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

int main() {
    int i = -1;
    for (int i = 1; i <= 10; i++)
        cout << i << endl;
    cout << i << endl;          // ispisuje -1
    return 0;
}
```

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.*

`for` petlje mogu biti ugniježđene jedna unutar druge. Ponašanje takvih petlji razmotrit ćemo na sljedećem programu:

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

int main() {
    for (int redak = 1; redak <= 10; redak++) {
        for (int 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 vanjsku petlju, inicijalizira se brojač redaka na vrijednost 1 te se s njom ulazi u 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 smo rabili operator za rukovanje (*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 prazninama. 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.*

3.6. 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. Sintaksa `while` bloka je

```
while ( uvjet_izvođenja )
    // blok_naredbi
```

`uvjet_izvođenja` je izraz čiji je rezultat tipa `bool`. Tok izvođenja `for`-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.dat`, ali uz promjene odgovarajućeg imena, to može biti i neka druga datoteka.

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

int main() {
    ifstream ulazniTok("brojevi.dat");
    cout << "Sadržaj datoteke:" << endl << endl;
    float broj;
    while ((ulazniTok >> broj) != 0)
        cout << broj << endl;
    return 0;
}
```

Brojevi u datoteci `brojevi.dat` 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).

Na početku programa se stvara objekt `ulTok` tipa (razreda) `ifstream`. Iako će razredi biti detaljnije objašnjene kasnije, za sada je dovoljno reći da je 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[†]. Želimo li koristiti tokove u našem programu, potrebno je uključiti datoteku zaglavlja `fstream` pomoću pretprocesorske naredbe `#include`.

Usredotočimo se na blok `while` naredbe. Na početku `while`-petlje, u uvjetu izvođenja se naredbom

```
ulTok >> broj
```

čita podatak. Ako je učitavanje bilo uspješno, `ulTok` će biti različit od nule (neovisno o vrijednosti učitanog broja), te se izvodi blok naredbi u `while`-petlji – ispisuje se broj na izlaznom toku `cout`. Nakon što se izvede naredba iz bloka, izvođenje se vraća na ispitivanje uvjeta petlje. Ako `ulTok` poprimi vrijednost 0, to znači da nema više brojeva u datoteci, petlja se prekida. Blok petlje se preskače, te se izvođenje 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 tada neće niti jednom izvesti, što u našem slučaju i trebamo.

Zadatak. *Dopunite gornji program tako da nakon sadržaja datoteke ispiše i broj iščitanih brojeva i njihovu srednju vrijednost.*

[†] Ulazni i izlazni tokovi te razred `fstream` detaljno su obrađeni u 15. poglavlju.

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 int fjel = 1;
    int i = 2;
    while (i <= n) {
        fjel *= i;
        i++;
    }
    cout << n << "! = " << fjel;
    return 0;
}
```

Trebalo je samo početni izraz (`int i = 2`) izlučiti ispred `while` naredbe, a izraz prirasta (`i++`) prebaciti u blok naredbi.

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. Ipak, 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.

3.7. 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, često je neophodno 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 programom-igricom u kojem treba pogoditi slučajno generirani trazeniBroj. Nakon svakog našeg pokušaja ispisuje se samo poruka da li je broj veći ili manji od traženog. Petlja se ponavlja sve dok je pokušaj (mojBroj) različit od traženog broja.

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

int main() {
    int raspon = 100;
    srand(time(0));     // inicijalizira generator sluč.brojeva
    // generira slučajni broj između 1 i raspon
    int trazeniBroj = static_cast<float>(rand()) / RAND_MAX
                      * (raspon - 1) + 1;
    cout << "Treba pogoditi broj između 1 i "
         << raspon << endl;
    int mojBroj;
    int brojPokusa = 0;

    do {
        cout << ++brojPokusa << ". pokušaj: ";
        cin >> mojBroj;
        if (mojBroj > trazeniBroj)
            cout << "MANJE!" << endl;
        else if (mojBroj < trazeniBroj)
            cout << "VIŠE!" << endl;
    } while(mojBroj != trazeniBroj);

    cout << "BINGO!!!" << endl;
    return 0;
}
```

Za generiranje slučajnih brojeva upotrijebljena je funkcija rand() iz zaglavlja cstdlib. Ona kao rezultat vraća slučajni broj između 0 i RAND_MAX, pri čemu je RAND_MAX broj također definiran u cstdlib biblioteci. Da bismo generirali broj u rasponu između 1 i raspon, broj koji vraća funkcija rand() podijelili smo s RAND_MAX (čime smo normirali broj na interval od 0 do 1), pomnožili s raspon - 1 i uvećali za 1. Prilikom dijeljenja primijenili smo operator dodjele tipa static_cast<float>, jer bi se u protivnom izgubila decimalna mjesta i rezultat bi bili samo brojevi 0 ili 1.

Uzastopni pozivi funkcije rand() uvijek generiraju isti slijed slučajnih brojeva, a prvi poziv te funkcije daje uvijek isti rezultat. Da bi se izbjegle katastrofalne posljedice takvog svojstva na neizvjesnost igre, prije poziva funkcije rand() pozvali smo funkciju srand() koja inicijalizira klicu generatora slučajnog broja, tj. postavlja početni "slučajno" generirani broj što ga daje rand() na neku drugu vrijednost. Funkciji srand() treba kao argument navesti neki cijeli broj na osnovu kojeg će se izračunati vrijednost klice. Iako bismo tu vrijednost mogli sami zadavati, radi *fair-playa* smo to prepustili računalu: pomoću funkcije time(), definirane u standardnoj biblioteci ctime,

učitava se trenutno sistemsko vrijeme (broj sekundi što ih odbrojava interni sat u računalu od referentnog trenutka[†]) i ono se koristi kao osnovica za klicu. Budući da nema šanse da program pokrenete više puta unutar iste sekunde (pa čak i na različitim računalima), tako generirani brojevi će biti doista slučajni.

Zadatak. *Izbacite iz gornjeg primjera naredbu* `srand()` *i provjerite "slučajnost" generiranih brojeva. Dodajte u gornji primjer provjeru je li broj pokušaja veći od 10 – u tom slučaju ispišite koliko je pokušaj "daleko" od traženog broja.*

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 realnu varijablu* `zbrojReda` *(početno 0), te cjelobrojne varijable* `red` *i* `preznak` *koje početno postavite na +1. Unutar* `do-while` *petlje definirajte realnu 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 izmnoženi* `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.*

3.8. Naredbe `break` i `continue`

Naredba `break` može se koristiti samo u petljama te u `switch` grananjima. Njenu funkciju u `switch` grananjima upoznali smo u odjeljku 3.4, dok se u petljama njome prekida 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. 94) 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.

[†] Na IBM kompatibilnim osobnim računalima referentno vrijeme je ponoć (0:00 sati) 1. siječnja 1970. godine.

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:

```
// ...
while ((znak = fgetc(ulazniTok)) != EOF) {
    if (znak < ' ' && znak != '\r')
        continue;       // preskače naredbe do kraja petlje
    cout << znak;
}
// ...
```

Nailaskom na znak čiji je kôd manji od kôda za prazninu i nije jednak znaku za novi redak, izvodi se 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 okolne 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.

Kao ilustraciju nesvrhovitog korištenja `break` naredbe pogledajmo primjer kakav se nerijetko može vidjeti kod samozvanih C++ gurua:

```
while (1) {         // beskonačna petlja!
    int i;
    cin >> i;
    if (i == 0)     // ako je učitani broj 0,
        break;      // tada prekini petlju
}
```

Budući da je uvjet izvođenja petlje uvijek 1 (tj. `true`), petlja je beskonačna; prekida se tek izvođenjem `break` naredbe. Ljepše napisana petlja bi izgledala ovako:

```
int i;
do {
    cin >> i;
} while(i != 0);
```

Za razliku od početnog rješenja, u ovako napisanom kôdu uvjet izvođenja petlje je odmah uočljiv.

3.9. Ostale naredbe za skok

Naredba `goto` omogućava bezuvjetni skok na neku drugu naredbu unutar iste funkcije. Opći 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 poglavlju 0.

U pravilno strukturiranom programu naredba `goto` uopće nije potrebna, te ju velika većina programera uopće ne koristi.

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 5 posvećenom funkcijama.

3.10. 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:

```
for ( /*...*/ )
    {
    // blok naredbi
    // ...
    }
```

2. početna zagrada izvučena, završna uvučena:

```
for ( /*...*/ )
{   // blok naredbi
    // ...
    }
```

3. zagrade izvučene, međusobno poravnate:

```
for ( /*...*/ )
{   // blok naredbi
    // ...
}
```

4. početna zagrada na kraju naredbe za kontrolu, završna izvučena:

```
for ( /*...*/ ) {
    // blok naredbi
    // ...
}
```

Pristupi 2. i 3. imaju još podvarijante u kojima se redak s početnom zagradom ostavlja prazan, bez naredbe. Međutim, taj prazan redak ne doprinosi bitno preglednosti i nepotrebno zauzima prostor. Ista zamjerka može se uputiti prvom pristupu.

Već letimičnim pogledom na četiri gornja pristupa čitatelj će uočiti da izvučena završna zgrada u 3. odnosno 4. pristupu bolje ističe kraj bloka. U 3. pristupu zagrade u paru otvorena-zatvorena vitičasta zagrada međusobno su poravnate, tako da je svakoj zagradi lako uočiti njenog partnera, što može biti vrlo korisno prilikom ispravljanja programa. Međutim, u mnogim knjigama koristi se pristup 4 kod kojeg je početna vitičasta zagrada smještena na kraj naredbe za kontrolu toka. Budući da ona započinje blok te je vezana uz njega, ovaj pristup je logičan. U ostalim pristupima povezanost bloka naredbi s naredbom za kontrolu toka nije vizualno tako očita i može se steći dojam da blok naredbi predstavlja potpuno samostalnu cjelinu. Zbog navedenih razloga (*Kud svi Turci, tuda i mali Mi*), u knjizi koristimo 4. pristup. Uostalom, pronalaženja para, odnosno provjera uparenosti vitičastih zagrada u urednicima teksta (*editorima*) koji se koriste za pisanje i ispravljanje izvornog kôda ne predstavlja problem, jer većina njih ima za to ugrađene funkcije.

Druga nedoumica jest koliko duboko uvlačiti blokove. Za uvlačenje blokova najprikladnije je koristiti tabulatore. Obično editori 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 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;
```

Koji ćete pristup preuzeti ovisi isključivo o vašoj odluci, ali ga onda svakako koristite dosljedno.

3.11. 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 editoru i 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 2.16). 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 `if` ispitivanju 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 uvjet za povećavanje kontrolne varijable `for` petlje:

```
// ver. 3.0
//...
for (int i = 1; i <= 100; zbroj += i, i++) ;
//...
```

Blok naredbi u petlji je sada ostao prazan, ali ne smijemo izbaciti znak `;`. U dosadašnjim realizacijama mogli smo brojač petlje `i` povećavati i prefiks-operatorom:

```
// ver. 3.1
//...
for (int i = 1; i <= 100; zbroj += i, ++i) ;
//...
```

Efekt je potpuno isti – u izrazu prirasta `for`-naredbe izrazi odvojeni znakom `,` (zarezom) se izračunavaju postupno, slijeva na desno. Naravno, da su izrazi napisani obrnutim redoslijedom

```
for (int i = 1; i <= 100; i++, zbroj += i)
```

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. Međutim, kada stopimo oba izraza za prirast

```
// ver. 4.0
//...
for (int i = 1; i <= 100; zbroj += i++) ;
//...
```

više nije svejedno koristi li se postfiks- ili prefiks-operator inkrementiranja, tj. da li se prvo dohvaća vrijednost brojača, dodaje zbroju i zatim povećava brojač, ili se prvo povećava brojač, a tek potom dohvaća njegova vrijednost:

```
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 `for`-naredbu, zakamuflirali smo osnovnu naredbu zbog koje je uopće petlja napisana. Budući da većina današnjih prevoditelja ima ugrađene 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).