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 jezikom C++? To je kompliciran jezik, spor, nedovoljno efikasan za primjenu u komercijalnim programima. Na kraju, ja to sve mogu napraviti u običnom C-u." Prije nego što počnemo objašnjavati pojedine značajke jezika, nalazimo važnim pokušati dati koliko-toliko suvisle odgovore na ta i slična pitanja.

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

0.1. Povijesni pregled razvoja programskih jezika

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

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

U asemblerskom jeziku svaka strojna instrukcija predstavljena je mnemonikom tj. nekom pridruženom kraticom koja je razumljiva ljudima koji čitaju program. Tako se za zbrajanje najčešće koristi mnemonik `ADD`, dok se za premještanje podataka s jednog

www.element.hr

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

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

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

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

Bjarne Stroustrup (rođen u Danskoj) je 1979. godine započeo rad na jeziku „C s klasama" (engl. *C with Classes*). Prije toga, on je radio na svom doktoratu u *Computing Laboratory of Cambridge* u Engleskoj te istraživao distribuirane sustave, granu računalne znanosti u kojoj se proučavaju modeli obrade podataka na više jedinica istodobno. Pri tome je koristio jezik Simula, koji posjedovao neka važna svojstva koja su ga činila prikladnim za taj posao. Na primjer, Simula je posjedovala pojam *klase* (engl. *class* - vrsta, razred, klasa)[1] – strukture koja objedinjava podatke i operacije nad podacima. Korištenje klasa omogućilo je da se koncepti problema koji se rješava izraze izravno 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

[1] Jezik Simula, koji su 1960-ih godina stvorili Kristeen Nygaard i Ole-Johan Dahl u *Norveškom računalnom centru* u Oslu, općenito se smatra prvim objektno orijentiranim programskim jezikom koji je uveo pojam klase te je imao veliki utjecaj na pojavu jezika C++ i Java.

www.element.hr

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

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

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

0.2. Osnovna svojstva jezika C++

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

1. *apstrakcija* (engl. *abstraction*)
2. *enkapsulacija* (engl. *encapsulation*)
3. *nasljeđivanje* (engl. *inheritance*)
4. *polimorfizam* (engl. *polymorphism*).

Sva ta svojstva doprinose ostvarenju takozvane *objektno orijentirane paradigme* programiranja. Da bismo to bolje razumjeli, pogledajmo koji su se programski modeli koristili u prošlosti. Pri tome je svakako najvažniji model proceduralno strukturiranog programiranja.

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

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

Princip kojim bismo mogli obilježiti proceduralno strukturirani model jest „podijeli-pa-vladaj”: cjelokupni program je presložen da bi ga se moglo razumjeti pa se zbog toga on rastavlja na niz manjih zadataka – procedura – koje su dovoljno jednostavne da bi se mogle izraziti pomoću naredbi programskog jezika. Pri tome, pojedina procedura također ne mora biti riješena monolitno: ona može svoj posao obaviti kombinirajući rad niza drugih procedura.

www.element.hr

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

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

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

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

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

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

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

Objedinjavanje podataka i operacija naziva se *enkapsulacija* (engl. *encapsulation*) – objekt se ne sastoji isključivo od podataka već sadrži i metode neophodne za operacije nad tim podacima. Pritom su podaci dostupni samo metodama unutar objekta, ali ne i ostalim dijelovima programa. Na taj su način podaci zaštićeni od neovlaštene promjene izvan objekta koja bi mogla narušiti njegovu cjelovitost. Ovakav mehanizam zaštite podataka naziva se *skrivanje podataka* (engl. *data hiding*). Svaki objekt svojoj okolini pruža

www.element.hr

isključivo podatke i operacije koji su neophodni da bi okolina objekt mogla koristiti. Ti javno dostupni podaci zajedno s operacijama koje ih prihvaćaju ili vraćaju čine *sučelje* (engl. *interface*) objekta. Programer koji će koristiti taj objekt više se ne mora zamarati razmišljajući o načinu na koji objekt iznutra funkcionira – on jednostavno preko sučelja traži od objekta određenu uslugu.

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

Kada PC preprodavač, vlasnik poduzeća „Taiwan/tavan-Commerce" sklapa računalo, on zasigurno treba kućište (iako se i to ponekad pokazuje nepotrebnim). To ne znači da će on morati početi od nule, miksajući atome željeza u čašici od *Kinderlade*. On će jednostavno otići kod susjednog dilera i kupiti gotovo kućište koje ima priključak na mrežni napon, napajanje, ladice za montažu diskova, USB utičnice. 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 novu liniju automobila, imaju dva izbora: ili mogu početi od nule i ponovo proračunavati svaki najmanji dio motora, šasije i ostalih dijelova, ili mogu jednostavno novi model bazirati na nekom starom modelu. Kompaniji je cilj što brže razviti novi model kako bi pretekla konkurenciju, pa će zasigurno jednostavno uzeti prethodni uspješni model automobila i samo izmijeniti neka njegova svojstva: promijenit će mu karoseriju, pojačati motor, dodati elektroniku za pouzdanije upravljanje. Slično je i s programskim komponentama: prilikom rješavanja nekog problema možemo uzdahnuti i početi kopati ili možemo uzeti neku već gotovu komponentu koja je blizu rješenja i samo dodati nove mogućnosti. To se zove *ponovna iskoristivost* (engl. *reusability*) i vrlo je važno svojstvo. Za novu programsku komponentu kaže se da je *naslijedila* (engl. *inherit*) svojstva komponente iz koje je izgrađena.

Korisnik koji kupuje auto sigurno neće biti presretan ako se njegov novi model razlikuje od starog po načinu korištenja, primjerice da se umjesto pritiskom na papučicu gasa auto ubrzava povlačenjem ručice na krovu vozila ili spuštanjem suvozačevog sjedala. On jednostavno želi pritisnuti papučicu plina a da pritom nova verzija automobila treba kraće vrijeme ubrzanja od 0 do 100 km/h. Slično je i s programskim komponentama: korisnik

www.element.hr

se ne treba opterećivati time koju verziju komponente koristi – on će jednostavno tražiti od komponente uslugu, a na njoj je da to obavi na adekvatan način. U primjeru s rotacijama tijela, korisnik će od svakog pojedinog tijela zatražiti istu operaciju – da se zakrene – a tijelo će operaciju izvesti na njen svojstven način. Svojstvo da različiti objekti istu operaciju obavljaju na različite načine zove se *polimorfizam* (engl. *polimorphism*).

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

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

Sada glavni program više ne mora voditi računa o tome koji se objekt rotira – on jednostavno samo zatraži od objekta da se zarotira. Sam objekt zna to učiniti ovisno o tome koji lik on predstavlja: kocka će se zarotirati na jedan način, a *kubični spline* na drugi. Također, ako se kasnije program proširi novim tijelima, nije potrebno mijenjati dio programa koji traži od pojedinih objekata da se zarotiraju – dovoljno je samo u novododanom objektu definirati operaciju rotacije. Prema tome, ono što jezik C++ čini vrlo pogodnim jezikom opće namjene za izradu složenih programa jest mogućnost jednostavnog uvođenja novih tipova te naknadnog dodavanja novih operacija.

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

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 dva jednoznamenkasta broja 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

www.element.hr

konkurencija, kvaliteta izvedbenog kôda koju su oni generirali se poboljšavala. Štoviše, današnji prevoditelji često daju efikasniji izvedbeni kôd iz C++ izvornog koda nego iz ekvivalentnog C kôda. Osim toga, jezik C++ sadrži neke konstrukcije, kao što su umetnute (engl. *inline*) funkcije, koje mogu znatno pospješiti brzinu konačnog izvedbenog koda. Ipak, u pojedinim slučajevima izvedbeni kôd dobiven iz C++ izvornog kôda doista može biti sporiji, a na programeru je da shvati kada je to dodatno usporenje prevelika smetnja da bi se toleriralo.

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

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

Virtualne funkcije i virtualne osnovne klase usporavaju program. Virtualne funkcije se izvode tako da se prije poziva konzultira posebna tablica, pa je jasno da će poziv svake takve funkcije biti nešto sporiji. Također, članovima virtualnih osnovnih klasa se redovito pristupa preko jednog pokazivača više. Međutim, usporenje je u većini realnih slučajeva neprimjetno i zanemarivo u odnosu na koristi koje virtualne funkcije donose. Na programeru je da se odluči hoće li koristiti „inkriminirana" svojstva jezika ili ne. Da bi se precizno ustanovilo kako djeluje pojedino svojstvo jezika na izvedbeni kôd, nije dobro nagađati i kriviti C++ za loše performanse, već izmjeriti vrijeme izvođenja te locirati problem.

Sagledajmo još jedan aspekt: asembler je jedini jezik u kojemu programer točno zna što se dešava u pojedinom trenutku prilikom izvođenja programa. Kako je programiranje u asembleru bilo složeno, razvijeni su viši programski jezici koji to olakšavaju. Prevedeni C kôd također nije maksimalno brz – posebno optimiran asemblerski kôd će sigurno dati bolje rezultate. No pisanje takvog kôda više nije moguće niti isplativo: problemi koji se rješavaju su previše složeni da bi se pazilo na svaki ciklus procesora.

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

www.element.hr

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

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

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

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

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).

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

www.element.hr

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 na neki način prisiljen razmišljati na objektno orijentirani način.

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

Nasuprot tome, Java je bila koncipirana kao potpuno novi programski jezik, neopterećen takvim nasljeđem. Zbog toga su neke stvari riješene elegantnije nego u jeziku C++. Uostalom, Javu je stvorila grupa C++ programera koji su bili frustrirani kompliciranošću jezika C++.

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

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

Java izvorni kôd se ne prevodi u strojni kôd matičnog procesora (*matični kôd*, engl. *native code*) nego u poseban, tzv. *Java binarni kôd* (engl. *Java bytecode*) i u takvom se obliku program distribuira korisnicima. Kada korisnik pokreće program, prvo se pokrene *Java virtualni stroj* (engl. *Java Virtual Machine*, *JVM*) na korisnikovom računalu, koji Java binarni kôd mora prevesti u strojni kôd i tek se onda taj strojni kôd izvodi. Prednost ovakvog pristupa jest da se prevedeni Java kôd (teoretski) može izvršavati na bilo kojem računalu s bilo kojim procesorom i operacijskim sustavom – uostalom, ovo je jedan od udarnih reklamnih slogana za Javu. Ipak, to vrijedi uz neke ograde.

Kao prvo, na odredišnom računalu mora biti instalirana odgovarajuća verzija Java virtualnog stroja. Za popularnije stolne (desktop) platforme, poput *Windows*, *Mac OS* ili *Linux* platformi postoji zadnja verzija virtualnog stroja, no za neke egzotične platforme mogu iskrsnuti problemi. Osim toga, da bi program pisan u Javi bio stvarno prenosiv na bilo koju platformu, ne smije sadržavati operacije specifične za određenu platformu. Na primjer, operacije vezane uz datotečni sustav ili grafičko sučelje mogu se značajno razlikovati od platforme do platforme.

Java binarni kôd se mora prvo prevesti u strojne instrukcije, što znači da se dio vremena namijenjenog za izvođenje programa zapravo troši na prevođenje iz Java binarnog u matični izvedbeni kôd. Zato pokretanje i izvođenje Java programa može biti osjetno

www.element.hr

sporije od ekvivalentnih programa prevedenih iz jezika C++. Osim toga, virtualni stroj iziskuje dodatne memorijske resurse tako da izvođenje programa pisanih u Javi troši nekoliko puta više memorije od istovjetnih programa prevedenih iz jezika C++. Zbog toga se Java programi često ne mogu izvoditi na ugradbenim računalnim sustavima (engl. *embedded systems*), koji se koriste u mnogim elektroničkim uređajima s ograničenom memorijom.

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

Programski kôd često poziva funkcije iz programskog sučelja operacijskog sustava što omogućava vrlo brzo izvođenje programa i maksimalno korištenje sklopovskih mogućnosti računala. Međutim, programska sučelja za pojedine platforme se međusobno razlikuju pa pisanje programa koji bi omogućio stvaranje izvedbenog kôda za različite platforme može biti vrlo mukotrpno. Srećom, postoje biblioteke (poput biblioteke Boost[1]) koje omogućavaju jednostavniju komunikaciju s različitim platformama. Za razliku od virtualnog stroja koji predstavlja „međusloj" tijekom izvođenja, multiplatformne biblioteke u jeziku C++ su međusloj koji se koristi tijekom prevođenja programa. Tijekom izvođenja su one potpuno transparentne pa je njihov utjecaj na brzinu zanemariv.

0.4.3. Java nema pokazivača

Davno su prošla vremena kada su se na sâm spomen riječi „pokazivač" (engl. *pointer*) zatvarali prozori i zaključavala vrata, a djecu tjeralo na spavanje ili 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 Babaroga kojom se plaše programeri-žutokljunci. Stoga će takvi aklamacijom prihvatiti istrjebljenje pokazivača.

Doduše, zavirimo li „pod haubu" vidjet ćemo da se u Javi svi objekti osim nekolicine ugrađenih, dohvaćaju preko pokazivača[2] – ono što je dobro jest da nema pretvorbi između pokazivača i objekata koje omogućavaju raznorazne (obično nenamjerne) „hakeraje" i redovito završavaju rušenjem ili blokiranjem programa. Iako su pokazivači jedan od najčešćih uzroka pogrešaka u početničkim C/C++ programima, iskusniji programeri znaju koliko je neke stvari jednostavnije i efikasnije napraviti koristeći pokazivače.

0.4.4. Java nema višestrukog nasljeđivanja

Nasljeđivanje omogućava kreiranje korisnički definiranih tipova podataka koristeći i proširujući osobine već postojećih tipova. Često je poželjno naslijediti osobine različitih tipova – tada se u jeziku C++ primjenjuje višestruko nasljeđivanje. Kao primjer, uzmimo tipku u nekoj aplikaciji koja umjesto standardnog oblika (tipka sa sjenčanjima i tekst) ima

[1] https://www.boost.org/

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

www.element.hr

oblik bitmapirane slike. Tipična primjena takve tipke bi bila u pregledniku datoteka sa slikama – umjesto da ispiše imena datoteka taj preglednik će prikazati male sličice, a pritiskom na tipku otvorit će se aplikacija u kojoj sliku možemo obrađivati. Naša tipka očito ima svojstva obične tipke (reagira na klik miša i u tom slučaju pokreće neku akciju), kao i svojstva bitmapirane slike (mora se znati iscrtati). Umjesto da prepišemo cjelokupan kôd oba tipa objekta, bit će dovoljno naslijediti njihova svojstva, eventualno modificirajući neka od njih.

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

0.4.5. Java ima ugrađeni sustav za automatsko upravljanjem memorijom

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

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

U programima s relativno malim brojem objekata sustav za automatsko upravljanje memorijom svoj posao će raditi prilično diskretno. Međutim, ako se tijekom izvođenja programa stvara veliki broj objekata, sakupljač smeća može drastično usporiti program. Java programeri će na takve slučajeve samo bespomoćno raširiti ruke uz komentar „to tako mora biti – ja ne mogu utjecati na *garbage collector*". U suštini, takvi programi iziskuju drugačiji pristup rješavanju problema. Mogli bismo reći da sustav za skupljanje smeća u takvim situacijama pomaže da se smeće gurne pod tepih i sakrije uzrok problema. Nasuprot tome, ekvivalentan program pisan u jeziku C++ će se najvjerojatnije zablokirati pa će programer biti prisiljen identificirati gdje je greškom dozvolio da memorija curi. U konačnici, i Java i C++ programer će se morati potruditi žele li da program radi brzo i korektno.

Budući da sustav za automatsko upravljanje memorijom sam procjenjuje kada treba to sakupljanje započeti, nerijetko se događa da to bude upravo u trenutku kada program obavlja neku zahtjevnu, vremenski kritičnu operaciju te će sakupljanje smeća usporiti izvođenje programa. Štoviše, budući da programer nema nadzora nad uništavanjem, ne

www.element.hr

može eksplicitno uništiti objekt, niti može kontrolirati redoslijed kojim se pojedini objekti uništavaju.

Ukratko, za vremenski kritične programe i programe koji se moraju izvoditi na sustavima s ograničenom memorijom, jezik C++ (u rukama discipliniranog programera) je u velikoj prednosti.

0.4.6. Java ne podržava preopterećenje operatora

Preopterećenje operatora omogućava da se funkcionalnost operatora može proširiti i za korisnički definirane podatke. Primjerice, operator zbrajanja + definiran je za ugrađene tipove podataka kao što su cijeli ili realni brojevi i znakovni nizovi. No, ako uvedemo kompleksne brojeve kao svoj novi tip podataka, realno je za očekivati da ćemo poželjeti zbrajanje kompleksnih brojeva izvoditi pomoću operatora +, na primjer:

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

Bez mehanizma preopterećenja operatora to nije moguće, već smo prisiljeni koristiti nečitljiviju sintaksu preko poziva funkcijskog člana:

```
c = a.add(b);
```

Zamislite samo kako bi u Javi trebao izgledati kôd za sljedeći matematički izraz:

```
e = (a + b) * 2 - (c - d) / 5;
```

0.4.7. U Javi su operacije s decimalnim brojevima lošije podržane

Izvorno je Java bila koncipirana uz pretpostavku da ju većina korisnika neće upotrebljavati za sofisticirane numeričke proračune. Budući da je osnovna misao vodilja autora Jave bila prenosivost kôda, nisu do kraja implementirani svi zahtjevi koje postavlja IEEE 754 standard za računanje s decimalnim brojevima tipa `float` te nisu do kraja iskorištene sve sklopovske mogućnosti procesora koji imaju ugrađene instrukcije za operacije s decimalnim brojevima[1]. Rezultat toga je i nešto sporije izvođenje takvih operacija. Ako točnost operacija s tipom `float` ne zadovoljava, korisnik ga može zamijeniti tipom `double`, no u slučaju da program istovremeno koristi veliki broj podataka to može stvoriti dodatne probleme, budući da podaci tipa `double` zauzimaju dvostruko više memorije. Ipak, kako je u specifikaciji jezika Java zapisano, prosječni korisnik gore spomenute nedostatke najvjerojatnije neće nikada primijetiti.

0.4.8. Java ima na raspolaganju opsežnu biblioteku

Java okruženje (engl. *Java Platform*) osim virtualnog stroja zaduženog za izvođenje programa, sadrži i vrlo bogatu biblioteku od nekoliko tisuća gotovih klasa i sučelja koji podržavaju čitav niz operacija uključujući rad s datotekama i mrežom, matematičke funkcije, grafičke operacije, rukovanje bazama podataka, obradu teksta, kriptografiju. Sve te

[1] W. Kahan, J. Darcy: *How Java's Floating-Point Hurts Everyone Everywhere*, ACM 1998 Workshop on Java for High–Performance Network Computing

www.element.hr

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

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

0.4.9. Java je u vlasništvu jedne tvrtke

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

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

0.5. Usporedba s jezikom C♯

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

0.6. Ima li smisla učiti jezik C++?

Zbog ponekad komplicirane sintakse, C++ prati glas vrlo nečitljivog i kompliciranog jezika. Jedan od glavnih uzroka tome je višestruki pristup podacima: podacima možemo pristupati izravno, preko pokazivača (odn. adrese) i preko reference. Ti različiti načini pristupa podacima pružaju fleksibilnost koja nije moguća u drugim jezicima, ali su nerijetko i uzrok pogreškama koje početnicima znaju zagorčati život. Java i C♯, koji su razvijeni na iskustvima jezika C++, nemaju tako kompliciranu sintaksu i dobrim dijelom podržavaju programske konstrukcije koje podržava jezik C++. Zbog toga je razvoj programa u tim i sličnim novijim programskim jezicima brži nego u jeziku C++. Međutim, C++ ima nekoliko bitnih prednosti od kojih ćemo spomenuti najvažnije.

www.element.hr

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

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

Zadnja, ali ne najmanje važna prednost jezika C++ jest njegova rasprostranjenost i popularnost. Po svim statistikama, jezik C++ je trenutno (rujan 2018.) među tri-četiri najpopularnija programska jezika, a zajedno s jezikom C (koji se u tim statistikama vodi zasebno) uvjerljivo vodi[1].

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

Primjeri kôda u knjizi su usklađeni s aktualnim standardom jezika C++ iz 2017. godine. Testirali smo ih pomoću najnovijih verzija popularnih prevoditelja koji su usklađeni sa standardom, … ali nikad se ne zna. Lako se može dogoditi da se poneki primjer ne može prevesti ili da ne radi ono što je napisano u knjizi zbog pogreške koju smo napravili u pripremi knjige.

Drugi razlog može biti neusklađenost prevoditelja kojeg koristite sa standardom. Naime, zahvaljujući velikoj popularnosti jezika C, programski jezik C++ se brzo proširio i prilično rano su se pojavili mnogi prevoditelji. Od svoje pojave, jezik C++ se dosta mijenjao, a prevoditelji su često „kaskali" za tim promjenama. Zbog toga se često događalo da je program koji se dao korektno prevesti na nekom prevoditelju, bio neprevodiv na novijoj inačici prevoditelja nekog drugog ili čak 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 standardizaciju (*International Organization for Standardization*, ISO). 14. studenog 1997. godine prihvaćen je ISO standard jezika C++ (ISO/IEC 14882, poznat pod nazivom *C++98*), koji je ratificiran 1998. 2003. godine prihvaćena je ažurirana verzija standarda (*C++03*) koja je uključivala samo ispravke i nije donijela nikakve značajne promjene u sam jezik.

Odbor za standardizaciju nastavio je raditi na sljedećoj verziji Standarda jezika, koja je prihvaćena u kolovozu 2011. pod nazivom *C++11*. Taj standard unosi u jezik niz novina opisanih u ovoj knjizi, ali održava kompatibilnost prema postojećem kôdu – sav postojeći kôd (uključujući i primjere iz prethodnih izdanja knjige) bi se i dalje, bez promjena, morao moći prevoditi na novijim prevoditeljima usklađenim sa standardom C++11. Verzija *C++14* objavljena je krajem 2014. godine, a uključivala je tek neznatna poboljšanja i ispravke pogrešaka.

[1] Npr. https://www.tiobe.com/tiobe-index/

www.element.hr

Krajem 2017. godine objavljen je *C++17*, koji unosi niz novosti, neke od kojih su spomenute u ovom izdanju knjige. Sljedeća verzija standarda, pod oznakom *C++20* trebala bi biti objavljena 2020. godine.

Većina najpopularnijih prevoditelja već u potpunosti podržava standard C++17, a dijelom i najavljene novine u standardu C++20, iako za to treba obično eksplicitno uključiti odgovarajuće opcije.

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

0.8. Literatura

Tijekom pisanja koristili smo literaturu navedenu na kraju knjige. Ponegdje se pozivamo na neku od tih knjiga, navodeći pripadajuću oznaku u uglatoj zagradi. Od svih navedenih knjiga, „najreferentniji" je trenutno aktualni standard jezika C++ iz 2017. godine; može se kupiti kod Američkog nacionalnog instituta za standarde (ANSI) ili kod Međunarodne organizacije za standardizaciju (ISO). Na Internetu se mogu naći besplatne radne verzije (engl. *draft*) standarda[1], dostupne u PDF (*Acrobat Reader*) formatu.

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

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

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

Nakon što je izašlo prvo izdanje knjige, jedan od najčešćih upita koje smo dobivali jest bio: „Može li se pomoću vaše knjige programirati u (*MS*) *Windowsima*?". Ova knjiga pruža osnove jezika C++ i vjerujemo da ćete se, kada nakon par mjeseci zaklopite zadnju stranicu, 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

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

www.element.hr

- Jeffrey Richter: *Windows via C/C++*, Microsoft Press, 2007, ISBN 978-0735663770.

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

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

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

0.9. Zahvale

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

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

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

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

0.9.1. Zahvale uz ostala izdanja

U prvom redu zahvaljujemo se svima koji su kupili prvo izdanje knjige koje je rasprodano u relativno kratkom vremenu – bez njih ne bi bilo novih izdanja knjige. A onima koji su fotokopirali ili skenirali knjigu ili preuzeli s neke stranice za dijeljenje datoteka želimo da Božićnicu doživotno dobivaju u fotokopiranim Konzumovim bonovima (naravno, svoje grijehe će okajati uništе li odmah ilegalni primjerak i kupe knjigu u knjižari).

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

www.element.hr