FAKULTET INŽENJERSKIH NAUKA

UNIVERZITETA U KRAGUJEVCU

P R O G R A M S K I J E Z I K

C++

Velibor Isailović

Autor:

dr Velibor M. Isailović, vanredni profesor, Fakultet inženjerskih nauka, Univerzitet u Kragujevcu

Naslov publikacije:

Programski jezik C++

Naziv izdavača:

Fakultet inženjerskih nauka Univerziteta u Kragujevcu, Sestre Janjić 6, Kragujevac

Za izdavača:

dr Slobodan Savić, redovni profesor, dekan, Fakultet inženjerskih nauka, Univerzitet u Kragujevcu

Urednik:

dr Blaža Stojanović, vanredni profesor, prodekan za nastavu, Fakultet inženjerskih nauka, Univerzitet u Kragujevcu

Recenzenti:
dr Nenad Filipović, redovni profesor, Fakultet inženjerskih nauka, Univerzitet u Kragujevcu
dr Boban Stojanović, redovni profesor, Prirodno-matematčki Fakultet, Univerzitet u Kragujevcu
dr Miloš Ivanović, redovni profesor, Prirodno-matematčki Fakultet, Univerzitet u Kragujevcu

Mesto i godina izdavanja: Kragujevac, 2024. godine.

Tiraž: 130 primeraka

Štampa: Grafički centar InterPrint, Kragujevac

ИСБН broj: 978-86-6335-085-4

Predgovor

Ova knjiga je univerzitetski udžbenik za kurseve tokom kojih se sluša programiranje na jezicima C i C++. Prvih osam poglavlja se uglavnom odnosi na programski jezik C, uz neznatno korišćenje elemenata jezika C++ za ulazno-izlazne operacije nad podacima. Preostalih osam poglavlja se u najvećoj meri odnose na objektno – orijentisano programiranje na jeziku C++.

Primeri koji prate obrađenu materiju nemaju veliku algoritamsku težinu. Oni su prevashodno namenjeni razumevanju koncepata proceduralnog i objektno-orijentisanog programiranja. U skladu sa tim, u prvih osam poglavlja je obrađena tema proceduralnog programiranja uz primenu struktura, kao složenog tipa podataka. Savladavanjem koncepta proceduralnog programiranja, uz proširenje korišćenjem struktura kao složenog tipa podataka, prelazi se na koncept objektno-orijentisanog programiranja.

Počev od devetog i zaključno sa dvanaestim poglavljem, obrađene su najznačajnije teme koncepta objektno-orijentisanog programiranja: apstrakcija, enkapsulacija, nasleđivanje, polimorfizam (virtuelne funkcije), preklapanje funkcija i operatora, itd. Neki od primera koji su obrađeni u delu koji se odnosi na proceduralno programiranje sa primenom struktura ponavljaju se u delu koji se odnosi na objektno-orijentisano programiranje. Ponovljeni primeri su prilagođeni principima objektno-orijentisanog programiranja i objektnom načinu razmišljanja, sa ciljem da se čitaocu prikažu razlike u načinu programiranja, kao i prednosti objektno-orijentisanog jezika C++ u odnosu na starije tehnike programiranja.

U trinaestom poglavlju obrađena je tema generičkog programiranja na jeziku C++. Ova tehnika je jedan od najmoćnijih alata za izgradnju kontejnerskih klasa, koje imaju mogućnost skladištenja velike količine podataka proizvoljnog tipa.

Četrnaesto poglavlje predstavlja kratak uvod u korišćenje kontejnerskih klasa, iteratora i generičkih algoritama sadržanih u standardnoj biblioteci jezika C++.

U petnaestom poglavlju je prikazan način primene mehanizma izbacivanja, hvatanja i obrade izuzetaka – alata koji definiše način reagovanja na nepredviđene okolnosti nastale u programu.

Poslednje, šesnaesto poglavlje daje osnovne smernice za rad sa ulazno-izlaznim tokovima podataka.

Nazivi klasa, funkcija i promenljivih korišćenih u primerima su na engleskom jeziku. Fond korišćenih engleskih reči je veoma mali, tako da ne predstavlja

prepreku u učenju programiranja. Razlog za ovakvu odluku leži u činjenici da poslovna praksa većine IT kompanija koje posluju u Srbiji podrazumeva pisanje koda korišćenjem engleskog jezika. Stil pisanja programa je takav da nazivi klasa, funkcija i promenljivih najčešće "govore" našoj intuiciji čemu služe ili na šta se odnose. Sve linije koda su numerisane kako bi se pojednostavila komunikacija sa studentima.

Na kraju, izrazio bih zahvalnost recenzentima na sugestijama, uočenim greškama i nedostacima. Zahvaljujući njihovom angažovanju na pregledu i izmenama originalnog teksta, ovaj udžbenik je dobio svoju konačnu formu.

Sadržaj

1. Uvod u programske jezike

Vrste programskih jezika

Kompjuterski program, koji se često naziva i aplikacija ili softver, predstavlja skup instrukcija koje govore računaru šta da radi. Komponente računara na kojima se fizički izvršavaju te instrukcije nazivaju se jednom rečju hardver.

Do danas je razvijen veliki broj programskih jezika. Jedna od osnovnih podela programskih jezika je na jezike **niskog nivoa**, kao sto su mašinski jezik i asembler, i jezike **visokog nivoa** kao što su: FORTRAN, Pascal, Basic, C, C++, Java, C#, Python, Ruby, JavaScript, PHP, itd.

Centralna procesorska jedinica (CPU) računara razume veoma ograničen skup instrukcija. Takav skup instrukcija se naziva **mašinski jezik**. Veoma bitna karakteristika ovog jezika jeste mogućnost direktne komunikacije sa hardverom. Instrukcije su šabloni bitova, gde svaki šablon odgovara određenoj komandi koja se zadaje mašini. Svaki model procesora ima sopstveni mašinski jezik ili skup instrukcija. Poboljšane verzije jednog procesora ili novi modeli zasnovani na prethodnim mogu koristiti sve instrukcije svog prethodnika kao i njima dodate instrukcije.

Asembler je programski jezik koji mašinski jezik određenog procesora predstavlja u formi koju je moguće pročitati. Kod napisan u asembleru se pretvara u mašinski kod, koji se može izvršiti na procesoru. Moguće je izvršiti i obrnuti proces, tj. pretvoriti mašinski kod u asemblerski, ali se tom prilikom gube određene informacije, kao sto su npr. komentari u kodu ili konstante, pa je tako dobijen kod prilično teško razumeti. Ovakav proces se naziva disasemblovanje.

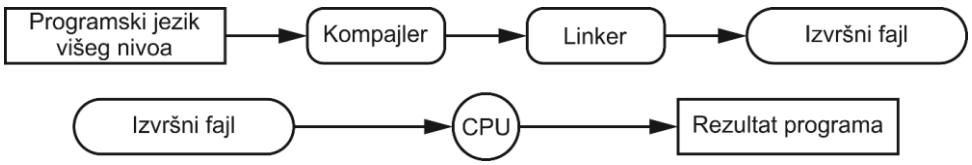
Korišćenje mašinskog jezika za pravljenje aplikacija bi bilo izuzetno komplikovano. Upravo zbog toga su tokom evolucije računara razvijeni mnogi programski jezici višeg nivoa, kao što su: FORTRAN, Pascal, Basic, C, C++, Java, C#, Python, Ruby, JavaScript, PHP, itd.

U zavisnosti od načina prevođenja, postoje dva načina izvršavanja programskih jezika višeg nivoa. U tom smislu, programski jezici se mogu podeliti na **kompajlerske** i **interpretatorske** jezike.

Kompajler je program koji čita izvorni kod i proizvodi samostalan izvršni program koji CPU može da razume. Kada se kod prevede u izvršni fajl, kompajler više nije potreban za pokretanje programa. Iako se intuitivno može zaključiti da su jezici

visokog nivoa manje efikasni od jezika niskog nivoa (asembler), savremeni kompajleri su veoma optimizovani i u stanju su da izvorni kod pisan u jeziku visokog nivoa pretvore u veoma brze izvršne fajlove.

Evo pojednostavljenog prikaza procesa kompajliranja izvornog koda i njegovog izvršavanja:



Slika 1: Proces kompajliranja programa

Interpretator (ili interpreter) je program koji direktno izvršava izvorni kod bez prethodnog prevođenja na mašinski kod. Interpretatori imaju veću fleksibilnost, ali su manje efikasni prilikom izvršavanja programa. Razlog tome leži u činjenici da je proces interpretacije koda potrebno uraditi svaki put kada se program pokreće. Drugim rečima, interpretator je potreban svaki put kada se program pokreće. Na slici Slika 2 je prikazan proces interpretacije izvornog koda i njegovog izvršavanja na računaru.

Slika 2: Proces interpretacije programa

Primeri programskih jezika koji se kompajliraju su C, C++, Pascal, FORTRAN, dok su primeri interpretatorskih programskih jezika Perl, JavaScript, Python. Neki jezici koriste i kombinaciju ova dva koncepta. Primer takvog programskog jezika je Java. Kod pisan na ovom jeziku se prevodi u takozvani bajt kod (bytecode), a zatim se takav kod izvršava na računaru uz pomoć Java Virtual Machine, i pritom je potpuno nezavisan od arhitekture računara na kome se pokreće.

Kompjuterski programi

Rešavanje problema korišćenjem kompjuterskih programa se može razložiti na više etapa:

Definisanje problema je postupak u kome naručilac programa i programer na prirodnom jeziku (srpski, engleski, itd.) definišu koje probleme program treba da

rešava. Iako nije neophodno, poželjno je da naručilac ima osnovno informatičko znanje, kako bi se lakše sporazumeo sa programerom i kako bi se izbegle eventualne greške usled različitog tumačenja istog problema.

Analiza problema obuhvata definisanje ulaznih i izlaznih podataka, moguća ograničenja njihovih vrednosti, kao i matematički model koji će biti korišćen za rešavanje datog problema.

Definisanje algoritma koji rešava problem podrazumeva definisanje konačnog uređenog niza pravila kojima se rešava određeni tip problema.

Projektovanje programa podrazumeva izbor platforme i programskog jezika, a zatim i definisanje arhitekture samog programa i načina čuvanja podataka.

Pisanje programa (kodiranje) je prevođenje pravila definisanih algoritmom na konkretan programski jezik. Dobro definisan algoritam znatno olakšava pisanje programa.

Testiranje je faza u razvoju kompjuterskih programa koja treba da osigura blagovremeno otkrivanje i uklanjanje grešaka. Testovi pomoću kojih se ispituje funkcionalnost programa treba da obuhvate sve opsege ulaznih promenljivih, kao i sve moguće grane u izvršavanju programa. Pored kontrole izvršavanja programa potrebno je obaviti i testiranje robusnosti programa u slučajevima unosa neodgovarajućih podataka od strane korisnika.

Analiza rezultata podrazumeva poređenje dobijenih rezultata sa teorijskim ili eksperimentalnim rezultatima (ukoliko je primenljivo), kao i modifikaciju modela u slučajevima kada dobijeni rezultati nisu u granicama dozvoljene tolerancije.

Isporuka programa je proces u kome se program putem različitih medija (CD, DVD, Flash memorija, FTP server, internet, ...) stavlja na raspolaganje naručiocu, tako da ga on može samostalno koristiti.

Održavanje programa je dugotrajan proces, koji podrazumeva obuku korisnika, ispravku uočenih nedostataka i prilagođavanje programa zahtevima korisnika.

U slučaju razvoja velikih programskih rešenja postoje različiti tipovi ciklusa kroz koje prolazi svaki program. Izbor najpogodnijeg tipa zavisi od namene programa, broja učesnika na projektu, strategije kompanije koja se bavi razvojem i mnogih drugih parametara.

Algoritmi

Ne postoji precizna definicija pojma algoritam. Sama reč potiče od prezimena persijskog matematičara Al Horezmija (Abu Abdulah Muhamed ibn Musa el Horezmi). Prvobitno je reč algoritam vezivana za opisivanje načina računanja pomoću decimalnih brojeva. Danas se reč algoritam gotovo isključivo vezuje za računarske nauke. Može se reći da je algoritam šematski opis rešavanja određenog problema. Algoritam je konačan i precizno definisan proces, koji sadrži niz jasno definisanih pravila, pomoću kojih se za zadate ulazne parametre izračunavaju vrednosti koje predstavljaju rešenje zadatog problema. Kako bi algoritam bio upotrebljiv, mora nedvosmisleno voditi do rešenja.

Jedan prost primer algoritma može biti neka rutinska radnja iz svakodnevnog života. Navedimo kao primer plaćanje parkinga.

- Pronaći saobraćajni znak sa brojem telefona i formatom poruke;
- Poslati poruku sa registracionom oznakom automobila u odgovarajućem formatu;
- Sačekati povratnu poruku;
- Ukoliko je povratna poruka potvrdna, parking je plaćen;
- Ukoliko je povratna poruka negativna, proveriti sadržaj poslate poruke i ponoviti slanje;
- Ukoliko prethodni uslov nije ispunjen, pozvati broj telefona parking servisa i obratiti se operateru.

Iz ovakvog jednostavnog primera može se videti postupnost, nedvosmislenost i konačnost algoritma. Algoritam koji se nikada ne završava ili algoritam čije se naredbe izvršavaju nepredvidivo nije upotrebljiv. Dakle, svrha algoritma je da precizno definiše naredbe pomoću kojih će se zadati problem rešiti.

2. Osnove jezika C++

Struktura programa

Računarski program je niz instrukcija koji računaru saopštava šta treba da radi.

Iskazi

Najčešća vrsta instrukcija u programu je iskaz. Iskaz je najmanja nezavisna jedinica u jeziku C++. Analogija sa jezikom kojim se sporazumevaju ljudi bila bi rečenica. Pišemo rečenice kako bi saopštili ideju ili neku misao. U C++ pišemo iskaze kako bismo saopštili kompajleru da želimo da obavimo određeni zadatak. Iskazi se najčešće završavaju simbolom tačka-zarez.

Postoji mnogo različitih tipova iskaza u jeziku C++. Slede neki od najčešćih tipova jednostavnih iskaza:

```
int x;
x = 5;
std::cout << x;
```

- `int x;` je deklaracija. Ovaj iskaz govori kompajleru da je x promenljiva koja čuva celobrojnu vrednost (`int`). U programiranju, promenljiva daje ime memorijskom bloku u kome je smeštena vrednost koja se može menjati. Sve promenljive u programu moraju biti deklarisane pre nego što se koriste. O tome će biti više reči kasnije.
- `x = 5;` je iskaz dodele vrednosti. Ovaj iskaz dodeljuje vrednost 5 promenljivoj x.
- `std::cout << x;` je izlazni iskaz, koji na ekranu prikazuje vrednost promenljive x (kojoj je vrednost dodeljena u prethodnom iskazu).

Izrazi

Izraz je matematički entitet u kome se izračunava neka vrednost. Kompajler je takođe sposoban za izračunavanje izraza. Na primer, jedan jednostavan izraz u matematici je: `2 + 3 = 5`. Izrazi mogu uključivati vrednosti (kao što je, na primer, bilo koji broj u prethodnom izrazu), promenljive (x), operatore (+,-,*,/,=,..) i funkcije (koje vraćaju neku vrednost izračunatu na osnovu ulaznih vrednosti funkcije). Generalno, izrazi mogu biti prosti ili složeni.

Na primer, izraz `x = 2 + 3;` je važeći izraz dodele vrednosti. Izraz `2 + 3` rezultuje vrednošću 5. Vrednost 5 se zatim dodeljuje promenljivoj x.

Funkcije

U jeziku C++, iskazi su grupisani u celine koje se nazivaju funkcije. Funkcija je kolekcija iskaza koja se izvršava sekvencijalno. Svaki C++ program mora sadržati posebnu funkciju pod nazivom `main`. Kada se program pisan na jeziku C++ pokreće, izvršenje počinje prvim iskazom unutar funkcije `main`. Funkcije su obično napisane kako bi obavile neki specifičan zadatak. Na primer, funkcija sa nazivom `max` može sadržati iskaze koji utvrđuju koji od dva prosleđena broja je veći. O funkcijama će biti više reči kasnije.

Dobra praksa je da `main()` funkcija bude smeštena u `.cpp` datoteci koja se zove ili `main.cpp` ili da ime `.cpp` fajla bude isto kao ime projekta. Na primer, `main` funkcija programa za igru "snake" može biti smeštena u datoteci sa imenom `snake.cpp`.

Biblioteke i C++ standardna biblioteka

Biblioteka je kolekcija prevedenog (kompajliranog) koda (npr. funkcija) koja je "zapakovana" i kao takva spremljena za ponovnu upotrebu u mnogim različitim programima. Biblioteke omogućavaju proširenje funkcionalnosti programa. Na primer, tokom razvoja video igara postojaće potreba da se programu dodaju biblioteke za zvuk i grafiku.

Jezik C++ je zapravo veoma mali i minimalistički. Međutim, C++ takođe dolazi sa standardnom bibliotekom koja pruža dodatnu funkcionalnost. Standardna biblioteka je podeljena na više delova (ponekad se i ti delovi takođe nazivaju bibliotekama, iako su zapravo samo delovi standardne biblioteke), pri čemu se svaka od njih fokusira na pružanje određene vrste funkcionalnosti. Jedan od najčešće korišćenih delova C++ standardne biblioteke je biblioteka `iostream`, koja sadrži funkcionalnost za pisanje na ekranu i dobijanje podataka sa konzole.

Jedan jednostavan C++ program

U nastavku je dat jedan jednostavan program napisan na jeziku C++.

```
1  #include <iostream>
2
3  int main()
4  {
5     std::cout << "Pozdrav!";
6     return 0;
7  }
```

Linija 1 je poseban tip iskaza koji se naziva pretprocesorska direktiva. Pretprocesorske direktive saopštavaju kompajleru (prevodiocu) da izvrši određeni zadatak. U ovom slučaju se saopštava kompajleru da doda sadržaj `iostream` zaglavlja u program. Zaglavlje `iostream` obezbeđuje pristup funkcionalnostima biblioteke `iostream`, što će omogućiti ispisivanje teksta na ekran.

Linija 2 je prazna i kompajler je ignoriše.

U liniji 3 se nalazi deklaracija `main()` funkcije, koja predstavlja minimum sadržaja bilo kog C++ programa. Svaki program mora imati `main()` funkciju.

Linije 4 i 7 predstavljaju granice unutar kojih je smeštena implementacija `main()` funkcije. Sve između otvorene vitičaste zagrade u liniji 4 i zatvorene vitičaste zagrade u liniji 7 smatra se `main()` funkcijom.

Linija 5 je prvi iskaz u programu (može se reći da je to iskaz jer se završava simbolom tačka-zarez). To je iskaz u kome se štampa tekst na izlazu. `std::cout` je poseban objekat koji predstavlja konzolu / ekran. Simbol `<<` je operator (kao što je operator `+` u matematici) koji se naziva operatorom izlaza. Objekat `std::cout` preuzima ono što je prosleđeno njegovom izlaznom operatoru i štampa na ekranu. U ovom slučaju prosleđen mu je tekst `"Pozdrav!"`.

Linija 6 je nova vrsta iskaza, koja se naziva povratni iskaz. Kada se program završi, funkcija `main()` vraća vrednost operativnom sistemu koja signalizira da li je funkcija uspešno izvršena ili ne.

U ovom slučaju funkcija vraća vrednost `0` operativnom sistemu, što znači da je korektno izvršena. Nenulti brojevi se obično koriste da bi se naznačilo da nešto nije u redu i da je prekinuto izvršavanje programa (ili neke funkcije). O ovome će kasnije biti više reči, kada tema budu funkcije.

Sintaksa i sintaksne greške

Ako posmatramo rečenice na srpskom, engleskom ili bilo kom drugom govornom jeziku, one se konstruišu prema specifičnim gramatičkim pravilima koje se uče tokom obrazovanja (ili čak intuitivno). Na primer, normalne rečenice se završavaju u određenom periodu, sklopljene su od određenih standardnih entiteta (npr. subjekat, objekat, predikat, itd.) predstavljenih određenim vrstama reči (imenice, zamenice, pridevi, glagoli, itd.). Pravila koja regulišu kako se rečenice konstruišu na nekom jeziku predstavljaju sintaksu jezika. Ako se rečenice sastavljaju kršeći ova pravila, ili preklapanjem dve rečenice, biće prekršena sintaksna pravila, pa će rečenice biti nejasne, dvosmislene, itd.

Jezik C++ takođe ima sintaksu, tj. pravila koja moraju biti ispoštovana kako bi program mogao da se prevede (kompajlira). Kompajler je odgovoran za proveru da li program prati osnovnu sintaksu jezika C++. Ako je neko pravilo prekršeno, kompajler će prilikom prevođenja prijaviti grešku i dati informaciju o sintaksnoj grešci.

U nastavku je dat primer sa namerno napravljenom sintaksnom greškom: izostavljen je simbol tačka-zarez na kraju iskaza u redu 5.

```
1  #include <iostream>
2
3  int main()
4  {
5     std::cout << "Pozdrav!"
6     return 0;
7  }
```

Razvojno okruženje Visual Studio će dati sledeću poruku o grešci:

```
c:\users\user\documents\visual studio
2022\projects\test\test\main.cpp(6,3): error C2143: syntax error:
missing ';' before 'return'
```

Ovo govori da postoji sintaksna greška u liniji 6: izostavljen je simbol tačka-zarez ispred povratnog iskaza. U ovom slučaju, greška je zapravo na kraju linije 5. Najčešće će kompajler tačno odrediti liniju gde se nalazi sintaksna greška. Međutim, ponekad je greška takva da se ne vidi do sledeće linije ili čak i dalje od nje.

Sintaksne greške su uobičajene prilikom pisanja programa. Na sreću, često ih je lako popraviti. Program se može u potpunosti prevesti i pokrenuti tek kada se otklone sve sintaksne greške.

Komentari

Tipovi komentara

Komentar predstavljaju jedna ili više linija teksta koje su umetnute u izvorni kod, kako bi se objasnilo šta kod radi. U jeziku C++ postoje dve vrste komentara: komentar u jednoj liniji ili komentar u više linija koda.

Uobičajeno je da se komentar u jednoj liniji koristi za kratko objašnjenje o jednoj ili više linija koda koje slede iza komentara. Simbol // označava liniju u kodu koja predstavlja komentar i govori kompajleru da ignoriše sve što se nalazi do kraja te linije koda.

```
1   // objekti cout i endl se nalaze u biblioteci iostream
2
3   std::cout << "Pozdrav!" << std::endl;
```

Takođe, komentar u jednoj liniji može da se nalazi posle završenog iskaza. Trebalo bi izbegavati ovakve komentare ukoliko je iskaz u pomenutoj liniji predugačak, pa komentar može da izlazi van okvira ekrana. Takav kod je manje čitljiv.

```
1   std::cout << "Pozdrav!" << std::endl; // Komentar
```

Par simbola `/*` i `*/` označavaju komentar koji može zauzimati više linija u kodu. Sve između pomenutih simbola kompajler ignoriše prilikom prevođenja programa.

```
1   /* Ovo je višelinijski komentar.
2      Ova linija koda će biti ignorisana. */
```

Greška može nastati pri pokušaju da se jedan komentar ugnezdi unutar drugog višelinijskog komentara:

```
1   /* Ovo je /* komentar */ ali ovaj tekst nije unutar komentara */
```

Pravilna upotreba komentara

Komentare bi trebalo koristiti za tri stvari. Za datu biblioteku, program ili funkciju, komentare bi trebalo napisati tako da precizno opisuju šta biblioteka, program ili funkcija radi. Na primer:

```
1   // Ovaj program računa prosečnu ocenu studenta
```

```
1   // Ova funkcija računa korene zadate kvadratne jednačine
```

```
1   // Sledeća linija generise random vrednost
```

Svi ovi komentari daju budućem korisniku koda informaciju o tome šta program, funkcija ili deo koda rade, bez potrebe da se pogleda stvarni kod. Korisnik na osnovu ovakvih komentara može brzo shvatiti kako se kod koristi. Ovo je naročito važno kod timskog rada, gde različiti članovi tima rade na različitim zadacima i ne poznaju detalje čitavog koda.

U okviru biblioteke, programa ili funkcije, komentari se mogu koristiti kako bi opisali način na koji će kod ostvariti svoj cilj.

```
1   /* Da bi se izračunala konačna ocena, sabiraju se ocene svih
2   kolokvijuma i domaćih zadataka i dele sa ukupnim brojem
3   predispitnih obaveza, kako bi se dobio procenat uspešnosti. Ovaj
4   procenat se koristi za izračunavanje konačne ocene.*/
```

Ovakvi komentari daju korisniku ideju o tome kako je u kodu rešen određeni problem, bez previše ulaženja u detalje oko implementacije.

```
// Za izbor slučajne stavke sprovodi se sledeća procedura:
// 1) Sve stavke se smeste u listu
// 2) Izračunava se verovatnoća za svaku stavku
// 3) Izabere se slučajni broj
// 4) Odredi se koja stavka odgovara slučajnom broju po verovatnoći
// 5) Vrati se odgovarajuća stavka
```

Na nivou iskaza, komentari treba da se koriste kako bi se opisalo zašto kod nešto radi, a ne kako radi. Ukoliko je potrebno da se komentariše iskaz kako bi se objasnilo kako radi, tada je iskaz najverovatnije loše napisan. Sledi nekoliko primera loših i dobrih komentara:

Loš komentar: iz iskaza se vidi ono što stoji u komentaru.

```
// Dodeljivanje vrednosti 0 promenljivoj counter
int counter = 0;
```

Dobar komentar: komentarom je objašnjeno čemu služi neka promenljiva.

```
// Promenljiva conuter se koristi za brojanje pozitivnih vrednosti
int counter = 0;
```

Loš komentar: iz iskaza se vidi da se računa cena, ali nije jasno zašto se vrednost promenljive `items` deli sa 2;

```
// Izračunavanje cena stavki
cost = items / 2 * storePrice;
```

Dobar komentar: komentarom je objašnjeno zašto se deli sa 2;

```
// Deli se sa 2 zato što se prodaju u paru
cost = items / 2 * storePrice;
```

Komentarisanje koda

Pretvaranje jedne ili više linija koda u komentar se naziva komentarisanje koda. Ovo omogućava (privremeno) isključivanje delova koda, kako se oni ne bi kompajlirali. Oba načina komentarisanja se mogu primeniti u ovu svrhu, komentarisanje u jednoj liniji ili u vise linija.

```
//    std::cout << 1;
//    std::cout << 2;
//    std::cout << 3;
```

```
/*    std::cout << 1;
      std::cout << 2;
      std::cout << 3;  */
```

Razlozi komentarisanja koda su sledeći:

- Kod na kome se radi još uvek ne može da se prevede, a neophodno je pokretanje programa. Kompajler neće dozvoliti pokretanje koda ako postoje greške u prevođenju. Komentarisanje koda koji kompajler ne prihvata će omogućiti da se program prevede, tako da se može pokrenuti. Kada su sve greške u kodu ispravljene, komentar se može ukloniti i nastaviti sa daljim radom na kodu.
- Novi kod koji je napisan može da se prevede, ali ne radi ispravno. U ovoj fazi ga je moguće komentarisati, kako ne bi izazivao probleme zbog nepravilnog izvršavanja. Kada se otklone nepravilnosti i testira novi kod, komentar se može ukloniti.
- Pronalaženje izvora greške. Ukoliko program ne daje željene rezultate (ili se jednostavno sruši tokom izvršavanja), u nekim slučajevima može biti korisno onemogućavanje izvršavanja delova koda, kako bi se izolovao deo koda koji predstavlja uzrok neispravnog funkcionisanja programa. Izolovanjem dela koda koji ne radi ispravno smanjuje se područje koje treba detaljno analizirati, kako bi se otkrile linije koda koje izazivaju problem.
- Potrebno je zameniti jedan deo koda novim kodom. Umesto brisanja izvornog koda, može se samo komentarisati i ostaviti. Kada se sa sigurnošću utvrdi da novi kod radi ispravno, komentarisani kod se može izbrisati. Dakle, na ovaj način je sačuvan ispravni deo koda, tako da se može veoma jednostavno vratiti na staro, provereno rešenje.

Promenljive, inicijalizacija i dodela vrednosti

Objekti

C++ programi stvaraju objekte, pristupaju im, manipulišu njima i uništavaju ih. Za početak, može se reći da je objekt deo memorije koji se može koristiti za čuvanje vrednosti. Ovo nije potpuna definicija objekta, jer u objektno-orijentisanom programiranju reč objekat ima prilično kompleksnije značenje. Objekti se mogu zamisliti kao bića ili stvari iz realnog sveta: čovek, automobil, pas, sto, stolica, itd. Svi računari imaju memoriju, nazvanu RAM (Random Access Memory), koja je programima dostupna za korišćenje. Kada se definiše neki objekat, deo te memorije se izdvaja za njegovo smeštanje. Većina objekata koji se koriste u jeziku C++ dolaze u formi promenljivih.

Promenljive

Iskaz kao što je k = 5; izgleda dovoljno očigledno. Kao što se može pretpostaviti, broju k dodeljena je vrednost 5. Ali šta je zapravo k? k je promenljiva.

Promenljiva u jeziku C++ je jednostavno deo memorijskog prostora kome je dodeljeno ime.

U ovom odeljku će biti razmatrane samo celobrojne promenljive. Primeri celih brojeva su: -12, -1, 0, 4, 27, itd. Celobrojna promenljiva je promenljiva koja sadrži celobrojnu vrednost.

Da bi se kreirala promenljiva, generalno, koristi se posebna vrsta iskaza koji se zove definicija (razlika između deklaracije i definicije biće objašnjena kasnije). Sledi primer definisanja celobrojne promenljive k (tj. promenljive u koju se smešta celobrojna vrednost):

```
1   int k;
```

Kada CPU računara izvršava ovaj iskaz, deo RAM memorije će biti rezervisan za promenljivu k.

Jedna od čestih operacija nad promenljivama je dodela vrednosti. Za to se koristi operator dodele vrednosti (=).

```
1   k = 5;
```

Inicijalizacija i dodela vrednosti

C++ podržava dva srodna koncepta koje programeri često pomešaju: dodeljivanje i inicijalizacija.

Nakon definisanja promenljive, vrednost joj može biti dodeljena preko operatora dodele (znak =):

```
1   int k;   // definicija promenljive
2   k = 5;   // dodela vrednosti 5 promenljivoj k
```

Pored toga, C++ omogućava definisanje promenljive i dodelu početne vrednosti u istom iskazu. Ovo se zove inicijalizacija.

```
1   int k = 5;   // inicijalizacija promenljive k vrednošću 5
```

Inicijalizacija promenljive se može obaviti samo prilikom definisanja promenljive.

Iako su ova dva koncepta slična po prirodi i često se mogu koristiti za postizanje sličnih ciljeva, kasnije će biti pokazani slučajevi u kojima neke vrste promenljivih

zahtevaju inicijalizaciju ili onemogućavaju dodelu vrednosti. Zbog toga je korisno napraviti razliku između ova dva koncepta.

Kada promenljiva ima početnu vrednost, inicijalizacija je poželjnija od dodele vrednosti.

Neinicijalizovane promenljive

Za razliku od nekih programskih jezika, C i C++ nemaju mogućnost automatske inicijalizacije promenljivih na neku vrednost (kao što bi, recimo, bila vrednost `0` za celobrojne promenljive). Stoga, kada je promenljivoj dodeljena memorijska lokacija od strane kompajlera, podrazumevana vrednost te promenljive je ono što je prethodno stajalo na toj memorijskoj lokaciji. Promenljiva kojoj nije dodeljena vrednost (kroz inicijalizaciju ili dodelu) naziva se neinicijalizovana promenljiva. Neki kompajleri, kao što je Visual Studio, inicijalizuju sadržaj memorije kada koristite Debug konfiguraciju kompajlera. Međutim, ovo se neće dogoditi kada koristite `Release` konfiguraciju. Stoga, upotreba neinicijalizovanih promenljivih može dovesti do neočekivanih rezultata. Korišćenje neinicijalizovane promenljive je prvi primer nedefinisanog ponašanja programa. U ovom slučaju, jezik C++ nema pravila koja određuju šta se dešava ako koristite vrednost promenljive kojoj nije dodeljena poznata vrednost. Shodno tome, ako se to učini, program će imati nedefinisano ponašanje.

Prvi pogled na cout, cin i endl

std::cout

Kao što je navedeno u prethodnim odeljcima, `std::cout` objekat, koji se nalazi u biblioteci `iostream`, može se koristiti za štampanje teksta na konzoli. Kao podsetnik, dat je ponovo jednostavan pozdravni program:

```
1  #include <iostream>
2
3  int main()
4  {
5     std::cout << "Pozdrav!";
6     return 0;
7  }
```

Za štampanje više od jedne stvari u istoj liniji koda, izlazni operator (<<) se može koristiti više puta. Na primer:

```
1  #include <iostream>
2
```

```
3   int main()
4   {
5      int x = 10;
6      std::cout << "x = " << x;
7      return 0;
8   }
```

Ovaj program će odštampati sledeći tekst:

```
x = 10
```

Sledi još jedan primer:

```
1   #include <iostream>
2
3   int main()
4   {
5      std::cout << "Zdravo!";
6      std::cout << "Moje ime je Jovana.";
7      return 0;
8   }
```

Na ekranu će biti odštampan ovakav tekst:

```
Zdravo!Moje ime je Jovana.
```

Rezultat je na prvi pogled neočekivan, ali objašnjenje sledi.

std::endl

Ako je potrebno odštampati više od jedne linije, to se može učiniti korišćenjem objekta `std::endl`. Kada se koristi sa `std::cout`, `std::endl` ubacuje znak nove linije, tj. uzrokuje da kursor skoči na početak sledeće linije. Sledi primer:

```
1   #include <iostream>
2
3   int main()
4   {
5      std::cout << "Zdravo!" << std::endl;
6      std::cout << "Moje ime je Jovana." << std::endl;
7      return 0;
8   }
```

Rezultat rada prethodnog programa će biti:

```
Zdravo!
Moje ime je Jovana.
```

std::cin

`std::cin` je objekat radi suprotno od `std::cout` - dok `std::cout` štampa podatke na konzoli pomoću izlaznog operatora (<<), `std::cin` čita ulaz sa

konzole koristeći ulazni operator (>>). Objekat `std::cin` se može koristiti za dobijanje korisničkog unosa i čuvanje te vrednosti u nekoj promenljivoj.

Kada se pokrene program koji sledi, na ekranu će biti odštampano `"Unesite jedan broj: "`, a zatim će program sačekati da se broj unese korišćenjem tastature. Kada se broj unese i pritisnite Enter, odštampaće se tekst `"Uneli ste broj "` a zatim i broj koji je upravo unet.

```
1  int main()
2  {
3      std::cout << "Unesite jedan broj: ";
4      int x;
5      std::cin >> x;
6      std::cout << "Uneli ste broj " << x << std::endl;
7
8      return 0;
9  }
```

Ovo je jednostavan način zahtevanja korisničkog unosa i biće često korišćen u mnogim primerima.

Uvod u funkcije, argumente, parametre i povratne vrednosti

Funkcija predstavlja niz iskaza dizajniranih za obavljanje određenog posla. Kao što je već napomenuto, svaki program mora imati funkciju pod nazivom `main()`. Međutim, svaki, bar malo kompleksniji program, sadržaće mnoštvo funkcija.

Veoma često program mora prekinuti ono što trenutno radi, kako bi privremeno uradio nešto drugo. Ovo se često događa i u stvarnom životu. Na primer, čitate knjigu, ali se setite da je potrebno da nekoga pozovete telefonom. Označićete u knjizi dokle ste stigli, obavićete telefonski poziv, a kada završite sa telefonskim pozivom, vratićete se u knjigu tamo gde ste stali.

C++ programi rade na isti način. Program će izvršavati iskaze sekvencijalno unutar jedne funkcije, i tom prilikom će naići na poziv neke druge funkcije. Poziv funkcije je izraz koji govori CPU da privremeno zaustavi izvršavanje tekuće funkcije i izvrši drugu funkciju. CPU "stavlja marker" na mestu gde je stao i poziva (izvršava) drugu funkciju. Kada se pozvana funkcija završi, CPU se vraća u funkciju u kojoj se zaustavio (na poziciju koju je prethodno označio) i nastavlja njeno izvršavanje.

Sledi primer malog programa u kome je pokazano kako se definiše neka funkcija i kako se poziva iz druge funkcije.

```
1  #include <iostream>
2
```

```
3   void doPrint()
4   {
5       std::cout << "U funkciji doPrint()" << std::endl;
6   }
```

```
1   int main()
2   {
3       std::cout << "Pokretanje funkcije main()" << std::endl;
4       doPrint(); // Pozivanje funkcije doPrint()
5       std::cout << "Kraj izvršavanja funkcije main()" << std::endl;
6
7       return 0;
8   }
```

Ovaj program će odštampati sledeći izlaz na konzoli:

```
Pokretanje funkcije main()
U funkciji doPrint()
Kraj izvršavanja funkcije main()
```

Program počinje izvršavanje pozivanjem funkcije `main()`. Prva linija koja će se izvršiti je štampanje teksta "`Pokretanje funkcije main()`" na konzoli. Druga linija u `main()` funkciji je poziv funkcije `doPrint()`. U ovom trenutku, izvršavanje iskaza u `main()` je privremeno zaustavljeno, a CPU pronalazi i poziva funkciju `doPrint()`. Prva (i jedina) linija u funkciji `doPrint()` je štampanje teksta "`U funkciji doPrint()`". Kada se `doPrint()` završi, funkcija pozivalac (u ovom slučaju funkcija `main()`) nastavlja izvršenje tamo gde je prethodno stala. Shodno tome, naredni iskaz koji se izvršava je štampanje teksta "`Kraj izvršavanja funkcije main()`" u `main()` funkciji.

Pozivanje funkcije u ovom slučaju se radi tako što se napiše naziv funkcije i par zagrada (otvorena i zatvorena). To znači da funkcija nema argumente. O ovome će biti više reči kasnije.

Povratne vrednosti

Kada se funkcija `main()` izvrši, operativnom sistemu vraća celobrojnu vrednost (najčešće 0) koristeći iskaz `return 0;`.

Prilikom definisanja ostalih funkcija, programer odlučuje da li će određena funkcija vratiti neku vrednost pozivaocu ili ne. Ovo se radi tako što se definiše tip povratne vrednosti funkcije. Tip povratne vrednosti se navodi ispred imena funkcije. On samo daje informaciju o tome koja vrsta vrednosti će biti vraćena a ne i o samoj vrednosti.

Unutar funkcije, koristi se iskaz `return` kako bi se vratila neka specifična vrednost programu ili funkciji koja poziva pomenutu funkciju. Vrednost koju funkcija vraća naziva se povratna vrednost funkcije.

Sledi primer jednostavne funkcije koja vraća celobrojnu vrednost i primer programa koji je poziva. U prvom pozivu `return5()`, funkcija vraća vrednost 5 pozivaocu – funkciji `main()`, koja tu vrednost predaje objektu `cout`, koji je zatim ispisuje na ekranu.

```
1   int return5()
2   {
3      return 5;
4   }
5
6   int main()
7   {
8      std::cout << return5() << std::endl;
9      std::cout << return5() + 2 << std::endl;
10
11     return5();
12     return 0;
13  }
```

U drugom pozivu funkcije `return5()`, funkcija vraća vrednost 5 pozivaocu. Izraz `5 + 2` se zatim izračunava i rezultat (vrednost 7) se štampa na konzoli.

U trećem pozivu funkcije `return5()`, funkcija vraća vrednost 5 pozivaocu. Međutim, `main()` ne radi ništa s povratnom vrednošću, pa se povratna vrednost odbacuje.

Sada bi trebalo da je jasno kako funkcija `main()` zapravo radi. Kada se program pokrene, operativni sistem poziva `main()` funkciju. Iskazi u `main()` funkciji se izvršavaju sekvencijalno. Na kraju, `main()` funkcija vraća celobrojnu vrednost operativnom sistemu (najčešće 0). Iz tog razloga je funkcija `main()` definisana kao `int main()`. Ova vrednost se zove statusni kod i ona govori operativnom sistemu (ili bilo kom drugom programu koji poziva ovakav program) da li je program uspešno izvršen ili ne. Po konvenciji, povratna vrednost `0` znači da je uspešno izvršen, a pozitivna (ili negativna) povratna vrednost signalizira na greške u izvršavanju.

C++ standard eksplicitno navodi da `main()` mora vratiti `int`. Međutim, ako se povratni iskaz u funkciji `main()` izostavi, kompajler će svakako vratiti vrednost `0`. Ipak, dobra je praksa da se iskaz `return` u funkciji `main()` ne izostavlja.

Nekoliko napomena o povratnim vrednostima

- Ako funkcija ima povratni tip (ne računajući `void`), ona mora vratiti vrednost tog tipa koristeći povratni iskaz. Jedini izuzetak od ovog pravila je funkcija `main()`, koja će vratiti vrednost od 0 čak i ako nije eksplicitno data.
- Kada se tokom izvršavanja neke funkcije dođe do iskaza `return`, program se vraća na liniju iz koje je funkcija pozvana. Sav preostali kod u funkciji se ignoriše.
- Funkcija može vratiti samo jednu vrednost pozivaocu. Međutim, funkcija može koristiti bilo koju logiku koja je na raspolaganju kako bi utvrdila koju specifičnu vrednost će vratiti. Može vratiti jedan broj (`return 5`). Ona takođe može vratiti vrednost promenljive, vrednost nekog izraza ili može izabrati jednu vrednost iz skupa mogućih vrednosti (što je opet samo jedna vrednost).
- Programer definiše da li funkcija ima povratnu vrednost i šta ona zapravo znači: Neke funkcije koriste povratne vrednosti u smislu statusnog koda, kako bi ukazale da li su uspešno završene ili ne; Neke funkcije vraćaju izračunatu ili izabranu vrednost; Neke ne vraćaju ništa. Zbog toga je značajno napisati komentar o funkcijama, šta vraćaju i koji je smisao povratne vrednosti.

Parametri i argumenti funkcija

Parametar funkcije je promenljiva koja se koristi u funkciji čiju vrednost daje pozivalac funkcije. Parametri funkcija se postavljaju između zagrada posle imena funkcije, i razdvajaju zarezom ukoliko ih ima više. Sledi nekoliko primera jednostavnih funkcija sa različitim brojem parametara.

```
1   void doPrint()
2   {
3      std::cout << "U funkciji doPrint()" << std::endl;
4   }
5
6   void printValue(int x)
7   {
8      std::cout << x  << std::endl;
9   }
10
11  int add(int x, int y)
12  {
13     return x + y;
14  }
```

Svaki parametar funkcije važi samo unutar te funkcije. Dakle, iako `printValue()` i `add()` imaju parametar sa imenom `x`, ovi parametri se smatraju zasebnim i nisu u sukobu.

Argument je vrednost koju pozivalac prenosi funkciji na mestu gde se funkcija poziva:

```
printValue(6); // broj 6 je argument funkcije
add(2, 3); // brojevi 2 i 3 su argumenti funkcije
```

Ukoliko funkcija ima više argumenata, takođe se odvajaju zarezom. Broj argumenata mora odgovarati broju parametara funkcije. U suprotnom, kompajler će prijaviti grešku.

Kada se pozove funkcija, svi parametri funkcije se kreiraju kao promenljive, a vrednost svakog od argumenata se kopira u odgovarajući parametar. Ovaj proces se zove prenošenje po vrednosti. Sledi primer:

```
#include <iostream>

void printValues(int x, int y)
{
   std::cout << x << std::endl;
   std::cout << y << std::endl;
}

int main()
{
   printValues(6,7);

   return 0;
}
```

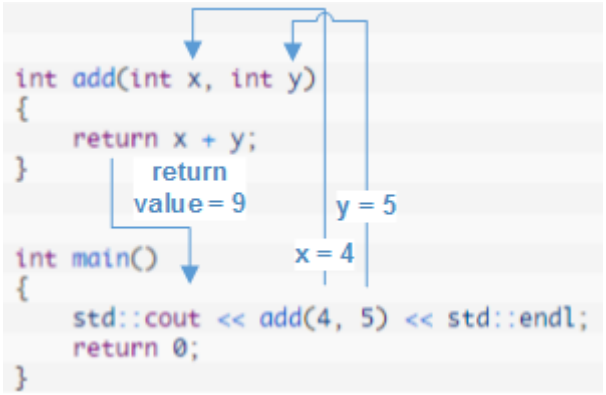
Kada se funkcija `printValues()` pozove sa argumentima `6` i `7`, stvara se parametar `x` i dodeljuje mu se vrednost `6`, a takođe i parametar `y` kome se dodeljuje vrednost `7`.

Sledi još jedan primer funkcije sa parametrima:

```
#include <iostream>

int add(int x, int y)
{
   return x + y;
}

int main()
{
   std::cout << add(4, 5) << std::endl;

   return 0;
}
```

U ovom primeru definisana je jednostavna funkcija kojoj se prosleđuju dva broja, a ona kao rezultat vraća njihov zbir. Kada se pozove funkcija `add()`, parametru x dodeljuje se vrednost 4, a parametru y dodeljuje se vrednost 5. Funkcija `add()` zatim izračunava izraz `x + y`, a to je vrednost 9, i vraća ovu vrednost natrag u funkciju `main()`. Ova vrednost 9 se zatim štampa na ekranu pomoću objekta `cout`.

Slikovito, to bi izgledalo ovako:



Sledi još jedan malo složeniji primer.

Prva dva iskaza u funkciji `main()` su jednostavna. Funkcijama `add()` i `multiply()` se prosleđuju po dva argumenta, čije se vrednosti izjednačavaju sa parametrima funkcija. Funkcije vraćaju rezultat (zbir ili proizvod), koji prihvata objekat `cout` i štampa ga na konzoli.

```
1   #include <iostream>
2
3   int add(int x, int y)
4   {
5       return x + y;
6   }
7
8   int multiply(int z, int w)
9   {
10      return z * w;
11  }
12
13  int main()
14  {
15     std::cout << add(4, 5) << std::endl;
16     std::cout << multiply(2, 3) << std::endl;
17
18     std::cout << add(1 + 2, 3 * 4) << std::endl;
19
```

```
20      int a = 5;
21      std::cout << add(a, a) << std::endl;
22
23      std::cout << add(1, multiply(2, 3)) << std::endl;
24      std::cout << add(1, add(2, 3)) << std::endl;
25      return 0;
26  }
```

U trećem iskazu, parametri su izrazi koji se izračunavaju pre nego što se prenesu funkciji. U ovom slučaju, izraz 1 + 2 se izračunava i njegova vrednost je 3. Tako izračunata vrednost predstavlja argument čija vrednost se dodeljuje prvom parametru funkcije – x. Drugi argument je takođe izraz, 3 * 4 čiji rezultat 12 predstavlja vrednost drugog argumenta. Ta vrednost se dodeljuje drugom parametru funkcije – y.

Sledeća dva iskaza su prilično jednostavna. Definisana je promenljiva a sa inicijalnom vrednošću 5. Zatim je ta promenljiva prosleđena funkciji add() dva puta, što znači da su oba parametra funkcije dobila istu vrednost 5. Rezultat koji funkcija vraća je 5+5=10.

Sledeća dva iskaza su malo složenija. U iskazu u liniji 23 je prvo pozvana funkcija add(). Njoj je kao prvi argument prosleđena vrednost 1, i to je vrednost prvog parametra funkcije. Na mestu drugog argumenta se nalazi poziv funkcije multiply() koja prihvata dva argumenta. To znači da će rezultat koji vrati funkcija multiply() zapravo biti drugi argument prethodno pozvane funkcije add().

Iskaz u liniji 24 je veoma sličan po strukturi, samo je drugi argument funkcije add() takođe funkcija add(), odnosno, rezultat koji ona vraća kada se pozove sa argumentima 2 i 3.

3. Promenljive i osnovni tipovi podataka

Osnovni tipovi promenljivih, inicijalizacija i dodela vrednosti

C++ jezik podržava određene tipove podataka. Ti tipovi podataka se nazivaju osnovni tipovi podataka, ali se često neformalno nazivaju i primitivni tipovi ili ugrađeni tipovi.

U tabeli koja sledi dat je pregled osnovnih tipova podataka.

Kategorija	Tip	Značenje
boolean	`bool`	true of false
character	`char`	ASCII karakter
floating point	`float`, `double`, `long double`	decimalni broj
integer	`short`, `int`, `long`, `long long`	ceo broj
void	`void`	ne postoji

U sledećem primeru definisano je pet različitih promenljivih korišćenjem pet različitih tipova.

```
1   bool bValue;
2   char chValue;
3   int nValue;
4   float fValue;
5   double dValue;
```

Inicijalizacija promenljivih

Kada je promenljiva definisana, može joj se istovremeno dodeliti vrednost. Ovo se naziva inicijalizacijom promenljive (ili kraće, inicijalizacija).

Jezik C++ podržava dva osnovna načina inicijalizacije promenljivih. Prvi način je inicijalizacija kopiranjem pomoću znaka jednakosti:

```
1   int nValue = 5; // Inicijalizacija kopiranjem
```

Drugi način je direktna inicijalizacija pomoću zagrada:

```
1   int nValue(5); // Direktna inicijalizacija
```

Iako direktna inicijalizacija podseća na poziv funkcije, kompajler vodi računa o tome šta su imena promenljivih a šta funkcije, tako da se ovakvi iskazi pravilno tumače.

Direktna inicijalizacija je za neke tipove podataka bolje rešenje od inicijalizacije kopiranjem. O ovome će biti više reči kasnije, kada tema budu klase.

Dodela vrednosti promenljivoj

Promenljivoj može biti dodeljena vrednost nakon što je prethodno definisana. Ovo se naziva dodelom vrednosti kopiranjem.

```
int nValue;
nValue = 5; // Dodela vrednosti kopiranjem
```

Definisanje više promenljivih

Moguće je definisati više promenljivih istog tipa u jednom iskazu tako što će imena promenljivih biti razdvojena zarezom. Definisanje dve promenljive u jednom iskazu:

```
int a, b;
```

Ovo je praktično isto kao da su definisane dve promenljive u dva odvojena iskaza:

```
int a;
int b;
```

Takođe, moguće je inicijalizovati više promenljivih u istom iskazu:

```
int a = 5, b = 6;
int c(7), d(8);
```

Gde treba definisati promenljive?

C++ kompajleri ne zahtevaju da se sve promenljive definišu na početku funkcije. Pravilan stil C++ programskog jezika je da se promenljive definišu što je moguće bliže njihovom prvom mestu upotrebe:

```
#include <iostream>
int main()
{
    std::cout << "Unesi prvi broj: ";
    int x;
    std::cin >> x;

    std::cout << "Unesi drugi broj: ";
    int y;
    std::cin >> y;

    std::cout << "Suma unetih brojeva je: " << x + y << std::endl;
    return 0;
}
```

Ovakav stil pisanja koda ima nekoliko prednosti:

- Jasna je namena promenljive koja je definisana na mestu gde se za njom prvi put javila potreba. Ako bi promenljiva x bila definisana na vrhu funkcije, ne bi bilo jasno čemu ona služi, sve dok se cela funkcija ne pregleda detaljno i pronađu mesta gde je korišćena.
- Definisanje promenljive na mestu gde je ona potrebna govori da ova promenljiva ne utiče na sve sto se nalazi iznad nje, što omogućava lakše razumevanje koda budućim korisnicima i zahteva mnogo manje kretanja kroz izvorni kod.
- Smanjuje se verovatnoća da promenljiva slučajno ostane neinicijalizovana, jer se može definisati i odmah inicijalizovati željenom vrednošću.

Tip `void`

`void` je najjednostavniji tip za objašnjavanje. U suštini, to znači da "nema tipa". Prema tome, ne mogu se definisati promenljive tipa `void`:

```
1   void a; //pogrešna deklaracija
```

`void` se obično koristi u sledećim situacijama:

(1) Kao način označavanja da funkcija ne vraća vrednost:

```
1   void writeValue(int x) // void znači da nema povratne vrednosti
2   {
3       std::cout << "Vrednost promenljive x je: " << x << std::endl;
4       // nema iskaza return zato što je povratni tip void
5   }
```

(2) U jeziku C, da se naznači da funkcija ne prihvata nikakve parametre:

```
1   int getValue(void) // void znači da nema parametara
2   {
3       int x;
4       std::cin >> x;
5       return x;
6   }
```

(3) Eksplicitna upotreba ključne reči `void` koja znači "nema parametara" je zadržana iz jezika C ali nije neophodna u jeziku C++. Sledeći kod je ekvivalentan prethodnom i poželjan je u C++:

```
1   int getValue() // prazna lista parametara (implicitno je void)
2   {
3       int x;
4       std::cin >> x;
5       return x;
6   }
```

Ključna reč `void` ima još jednu namenu u jeziku C++, koja će biti obrađena kasnije kada bude reči o pokazivačima (pointerima).

Veličina promenljivih i operator `sizeof`

Memorija na računarima je organizovana u jedinice veličine jednog bajta, pri čemu svaka jedinica ima jedinstvenu adresu. Većina promenljivih zapravo zauzima više od jednog bajta. Shodno tome, jedna promenljiva može koristiti 2, 4 ili čak 8 uzastopnih memorijskih adresa. Količina memorije koju koristi neka promenljiva zavisi od njenog tipa. Ovo je za sada samo korisna informacija, jer se memoriji najčešće pristupa preko imena promenljivih, a ne preko memorijskih adresa. Kompajler u velikoj meri skriva podatke o radu sa promenljivama različitih veličina.

Postoji nekoliko razloga zašto je korisno znati koliko memorije zauzima neka promenljiva.

- Što više memorije promenljiva zauzima, više informacija može zadržati. Pošto svaki bit može da ima vrednost 0 ili 1, kažemo da jedan bit može imati dve moguće vrednosti. Dva bita mogu imati četiri moguće vrednosti: 00, 01, 10, 11. Tri bita mogu imati osam mogućih vrednosti: 000, 001, 010, 011, 100, 101, 110, 111. Generalno posmatrano, može se reći da promenljiva sa n bitova može imati 2^n mogućih vrednosti. Jedan bajt (8 bitova) može sačuvati 2^8 (256) mogućih vrednosti. Veličina promenljive postavlja ograničenje na količinu informacija koje može da čuva - promenljive koje koriste više bajtova mogu imati širi raspon vrednosti.
- Računari imaju ograničenu količinu slobodne memorije. Svaki put kada se deklariše promenljiva, mali deo te slobodne memorije se zauzme sve dok promenljiva postoji. S obzirom da moderni računari poseduju veliku količinu memorije, ovo nije problem, posebno ukoliko je definisano samo nekoliko promenljivih. Međutim, za programe koji zahtevaju veliku količinu promenljivih (npr. 100,000,000,000), razlika između korišćenja jednobajtnih i osmobajtnih promenljivih može biti veoma značajna.

Veličina osnovnih tipova podataka u jeziku C++

Veličina osnovnih tipova podataka može da varira u različitim razvojnim okruženjima. Standard jezika C++ garantuje samo minimalne veličine osnovnih tipova podataka, koje su date u tabeli na narednoj strani.

Međutim, stvarna veličina promenljivih može biti drugačija na nekom računaru (naročito `int`, najčešće je 4 bajta). Da bi se odredila veličina tipova podataka na određenom računaru, C++ obezbeđuje operator `sizeof`. Operator `sizeof` je unarni operator koji uzima tip podataka ili promenljivu i vraća veličinu u bajtovima.

Kategorija	Tip	Minimalna veličina
boolean	`bool`	1 bajt
character	`char`	1 bajt

integer	`short`
	`int`
	`long`
	`long long`
		2 bajta
		2 bajta
		4 bajta
		8 bajtova

floating point	`float`
	`double`
	`long double`
		4 bajta
		8 bajta
		8 bajtova

Program koji sledi ispisuje veličinu osnovnih tipova podataka na računaru na kome je pokrenut.

```
1   #include <iostream>
2
3   int main()
4   {
5     std::cout << "bool:\t\t"      << sizeof(bool)        << std::endl;
6     std::cout << "char:\t\t"      << sizeof(char)        << std::endl;
7     std::cout << "wchar_t:\t"    << sizeof(wchar_t)     << std::endl;
8     std::cout << "char16_t:\t"   << sizeof(char16_t)    << std::endl;
9     std::cout << "char32_t:\t"   << sizeof(char32_t)    << std::endl;
10    std::cout << "short:\t\t"    << sizeof(short)       << std::endl;
11    std::cout << "int:\t\t"      << sizeof(int)         << std::endl;
12    std::cout << "long:\t\t"     << sizeof(long)        << std::endl;
13    std::cout << "long long:\t"  << sizeof(long long)   << std::endl;
14    std::cout << "float:\t\t"    << sizeof(float)       << std::endl;
15    std::cout << "double:\t\t"   << sizeof(double)      << std::endl;
16    std::cout << "long double:\t"<< sizeof(long double) << std::endl;
17    return 0;
18  }
```

Rezultati se mogu razlikovati zavisno od računara ili kompajlera.

Simbol `\t` u gore navedenom programu je poseban simbol koji dodaje tab u tekst koji se štampa. U ovom primeru je korišćen za poravnanje teksta koji se štampa na izlazu. Kasnije će biti reči i o drugim specijalnim simbolima, kada bude obrađivan tip podataka `char`.

Celobrojne promenljive

Celobrojni tip promenljivih, kao što i njegovo ime govori, može sadržati samo vrednosti iz skupa celih brojeva (npr. -2, -1, 0, 1, 2). U jeziku C++ postoji pet različitih tipova celobrojnih promenljivih: `char`, `short`, `int`, `long` i `long long`.

Tip `char` je poseban slučaj, pošto se svrstava i u slovne i celobrojne tipove. Za sada može biti tretiran kao normalan ceo broj.

Kategorija	Tip	Minimalna veličina
character	`char`	1 bajt
integer	`short`	2 bajta
	`int`	2 bajta (najčešće je 4)
	`long`	4 bajta
	`long long`	8 bajtova

Ključna razlika između različitih tipova `int` je u tome što imaju različite veličine - veći celobrojni tip može čuvati veću celobrojnu promenljivu. Kao što je već napomenuto, standard jezika C++ garantuje jedino minimalnu veličinu tipa `int` od dva bajta, iako je to gotovo uvek drugačije na današnjim računarima.

Definisanje celobrojnih promenljivih

Evo nekoliko definicija celobrojnih promenljivih:

```
1  char c;
2  short int si;         // važeća deklaracija
3  short s;              // poželjna deklaracija
4  int i;
5  long int li;          // važeća deklaracija
6  long l;               // poželjna deklaracija
7  long long int lli;    // važeća deklaracija
8  long long ll;         // poželjna deklaracija
```

Iako su `short int`, `long int` i `long long int` važeće definicije, poželjno je koristiti njihove kraće verzije: `short`, `long`, `long long`.

Znak celobrojnih promenljivih

Kao što je prethodno rečeno, promenljiva sa n bitova može imati 2^n različitih vrednosti. Skup svih specifičnih vrednosti koje neki tip podataka može imati naziva se opseg promenljive. Opseg celobrojnih promenljivih definišu dva faktora: veličina (u bitovima) i znak promenljive. U smislu znaka promenljive, mogu se podeliti na "označene" ili "neoznačene".

Označeni broj je promenljiva koja može imati i negativnu i pozitivnu vrednost. Za eksplicitno deklarisanje označenih promenljivih, može se upotrebiti ključna reč `signed`:

```
signed char c;
signed short s;
signed int i;
signed long l;
signed long long ll;
```

Na primer, jednobajtna označena celobrojna promenljiva ima opseg od -128 do 127.

Ponekad se unapred zna da neće biti potrebne negativne vrednosti. Ovo je uobičajeno kada se promenljive koristite za čuvanje veličine nečega (kao što je recimo visina - nema smisla imati negativnu visinu). Neoznačena celobrojna promenljiva može imati samo pozitivne vrednosti. Za eksplicitno deklarisanje promenljive kao neoznačene, može se upotrebiti ključna reč `unsigned`:

```
unsigned char c;
unsigned short s;
unsigned int i;
unsigned long l;
unsigned long long ll;
```

Na primer, jednobajtna neoznačena celobrojna promenljiva ima opseg od 0 do 255.

Značajna je činjenica da deklarisanje promenljive kao neoznačene dvostruko proširuje njen opseg vrednosti.

U sledećoj tabeli su dati opsezi za različite označene i neoznačene celobrojne tipove promenljivih.

Veličina [byte] – Tip	Opseg
1 – označen	-128 do 127
1 – neoznačen	0 do 255
2 – označen	-32768 do 32767
2 – neoznačen	0 do 65535
2 – označen	-2 147 483 648 do 2 147 483 647
4 – neoznačen	0 do 4 294 967 295
8 – označen	-9 223 372 036 854 775 808 do 9 223 372 036 854 775 807
8 – neoznačen	0 do 18 446 744 073 709 551 615

Pokušaj da se u promenljivu smesti broj koji je izvan njenog opsega prouzrokovao bi pogrešan zapis, jer u memorijskom bloku rezervisanom za tu promenljivu ne bi bilo dovoljno mesta.

Primer: smeštanje broja 21 u četvorobitnu promenljivu (koja inače prihvata brojeve od 0 do 15). Broj 21 u binarnom zapisu bi bio 10101. Prilikom smeštanja u memoriju, smeštale bi se redom cifre s desna na levo, tako da bi u promenljivu od 4 bita bile spakovane cifre 0101, koje odgovaraju broju 5. Sasvim je jasno je da ovakve stvari ne bi smele da se događaju u programu. Zato je veoma važno voditi računa o opsegu promenljivih i potencijalnoj mogućnosti da bude prekoračen.

Brojevi sa pokretnim zarezom

Promenljive sa pokretnim zarezom (floating point) su promenljive u koje mogu biti smešteni realni brojevi, kao što su 4320.0, -3.33 ili 0.01226. Termin pokretni zarez se odnosi na činjenicu da decimalni zarez može biti pokretan, odnosno, može imati promenljiv broj cifara pre i posle decimalnog zareza.

Postoje tri različita tipa podataka sa pokretnim zarezom: `float`, `double` i `long double`. Slično kao i kod celobrojnih promenljivih, C++ jezik definiše samo minimalnu veličinu ovih promenljivih. Današnja arhitektura računara skoro uvek prati IEEE 754 binarni format. U ovom formatu, `float` je 4 bajta, `double` je 8 bajtova, a `long double` može biti ekvivalentan `double` (8 bajtova), 12 bajtova ili 16 bajtova.

Tipovi podataka sa pokretnim zarezom su uvek označeni (mogu imati pozitivne i negativne vrednosti).

Kategorija	Tip	Minimalna veličina	Najčešća veličina

floating point	`float`
	`double`
	`long double`
		4 bajta
		8 bajtova
		8 bajtova
			4 bajta
			8 bajtova
			8, 12, 16 bajtova

Evo nekoliko definicija brojeva sa pokretnim zarezom:

```
1   float fValue;
2   double dValue;
3   long double dLongValue;
```

Kada se promenljivama sa pokretnim zarezom dodeljuje vrednost, postoji konvencija da se koristi najmanje jedno decimalno mesto. Ovo pravilo pomaže u razlikovanju vrednosti sa pokretnim zarezom od celobrojnih vrednosti.

```
1   int n(5);
```

```
2   double d(5.0);
3   float f(5.0f);
```

Kada se dodeljuje vrednost nekoj promenljivoj, podrazumevano je da je broj sa pokretnim zarezom tipa `double`. Sufiks f se koristi kako bi se eksplicitno naglasilo da je broj tipa `float`.

Naučna notacija

Naučna notacija je korisna za pisanje dugačkih brojeva na koncizan način. Brojevi u naučnoj notaciji imaju sledeći oblik: mantisa*10^{ekponent}. Na primer, u naučnoj notaciji 1.2*10^4, 1.2 je mantisa, 4 je eksponent. Ovaj broj je zapravo 12000. U jeziku C++ se umesto ovakvog zapisa dela sa eksponentom koristi slovo “e” ili “E” i eksponent. Zapis broja 12000 u C++ jeziku bi bio 1.2e+4 ili 1.2E+4.

Po dogovoru, brojevi u naučnoj notaciji se zapisuju tako da imaju jednu cifru pre zareza i ostatak cifara posle zareza.

Preciznost i opseg

Sa stanovišta elementarne matematike, često pri izračunavanju postoji potreba za razlomcima. Za rad sa razlomcima postoje precizno definisana matematička pravila. Međutim, sa stanovišta jezika C++, razlomke nije moguće sačuvati u memoriji onako kako bi bili zapisani na papiru. Na primer, razlomak 1/3 bi u jeziku C++ u decimalnoj reprezentaciji bio 0.33333333333333... Neograničena dužina broja zahteva beskonačnu memoriju za čuvanje, a obično je samo 4 ili 8 bajtova na raspolaganju. Brojevi sa pokretnim zarezom mogu da čuvaju samo određeni broj značajnijih cifara, a ostatak se gubi. Preciznost broja sa pokretnim zarezom definiše koliko se značajnih cifara može predstaviti bez gubitka informacija.

Kada se brojevi štampaju na konzoli, `std::cout` ima podrazumevanu preciznost od 6 - tj. pretpostavka je da promenljive sa pokretnim zarezom imaju do 6 značajnih cifara, pa će prema tome ostatak biti odsečen.

Sledeći program prikazuje kako `std::cout` štampa brojeve sa pokretnim zarezom sa šest decimalnih cifara:

```
1   #include <iostream>
2   int main()
3   {
4       float f;
5       f = 9.87654321f; // suffix f znači da je vrednost tipa float
6       std::cout << f << std::endl;
7       f = 987.654321f;
8       std::cout << f << std::endl;
```

```
9       f = 987654.321f;
10      std::cout << f << std::endl;
12      f = 9876543.21f;
13      std::cout << f << std::endl;
14      f = 0.0000987654321f;
15      std::cout << f << std::endl;
16      return 0;
17  }
```

Izlaz na konzoli bi izgledao ovako:

```
9.87654
987.654
987654
9.87654e+006
9.87654e-005
```

Može se primetiti da svaki odštampani broj ima samo 6 značajnih cifara. Takođe, u nekim slučajevima, `cout` će štampati brojeve u naučnoj notaciji. U zavisnosti od kompajlera, eksponent će obično takođe imati minimalan broj cifara. Minimalan broj cifara eksponenta zavisi od kompajlera (Visual Studio koristi 3, neki drugi koriste 2 prema standardu C99).

Podrazumevana preciznost objekta `cout` se može promeniti korišćenjem funkcije `std::setprecision()` koja je definisana u fajlu `iomanip`. O ovome će biti više reči kasnije.

Greške zaokruživanja

Postoji razlika u interpretaciji brojeva sa pokretnim zarezom u binarnom zapisu i načina na koji ih mi razumemo. Na primer, razlomak 1/10 u decimalnom zapisu se jednostavno predstavlja brojem 0.1. Međutim, u binarnom zapisu, broj 0.1 predstavljen je beskonačnom sekvencom: 0.00011001100110011... Zbog toga, kada se nekoj promenljivoj sa pokretnim zarezom dodeljuje vrednost 0.1, pojaviće se problem sa preciznošću takvog zapisa i greškom zaokruživanja. Efekti ovoga se mogu videti pokretanjem sledećeg koda.

```
1   #include <iostream>
2   int main()
3   {
4       double d(0.1);
5       std::cout << d << std::endl; // podrazumevana preciznost
6       std::cout << std::setprecision(17);
7       std::cout << d << std::endl; // štampa sa 17 decimalnih cifara
8       return 0;
9   }
```

Izlaz bi trebalo da bude:

```
0.1
0.10000000000000001
```

U prvoj liniji, `cout` štampa vrednost promenljive `d = 0.1`, što je i očekivano.

Međutim, u drugoj liniji, gde se pri štampanju koristi 17 značajnih cifara, promenljiva d zapravo nije jednaka tačno 0.1! Razlog tome je što je štampanje podešeno tako da se štampa 17 značajnih cifara, što je dovelo do toga da broj d nije jednak tačno 0.1 (17 značajnih cifara je izabrano zato što veličina memorijskog prostora rezervisana za promenljive tipa `double` dozvoljava toliko značajnih cifara). Ovo se zove greška zaokruživanja. U ovom momentu je značajno samo napomenuti ovu činjenicu. Kasnije će biti više reči o tome.

NaN i Inf

Postoje dve posebne kategorije brojeva u okviru tipa sa pokretnim zarezom. Prvi je `Inf` (Infinity), što predstavlja beskonačnost. `Inf` može biti pozitivan ili negativan. Drugi je NaN (Not a Number), što znači "Nije broj" ili IND (Indefinite Number). Postoji više tipova NaN-a, ali ovde neće biti reči o tome.

Sledi program koji prikazuje sva tri prethodno opisana tipa.

```
1   #include <iostream>
2   int main()
3   {
4       double zero = 0.0;
5       double posinf = 5.0 / zero; // pozitivna beskonačnost
6       std::cout << posinf << std::endl;
7
8       double neginf = -5.0 / zero; // negativna beskonačnost
9       std::cout << neginf << std::endl;
10
11      double nan = zero / zero; // nedefinisana vrednost
12      std::cout << nan << std::endl;
13
14      return 0;
15  }
```

Startovanjem programa trebalo bi da se dobije ovakav izlaz:

```
1.#INF
-1.#INF
1.#IND
```

`INF` predstavlja beskonačnu vrednost (deljenje nulom), a `IND` predstavlja nedefinisanu vrednost (deljenje nule nulom).

Dve stvari treba zapamtiti o brojevima sa pokretnim zarezom:

1) brojevi sa pokretnim zarezom su odlični za čuvanje veoma velikih ili veoma malih brojeva, uključujući i one sa razlomljenim komponentama, sve dok imaju ograničen broj značajnijih cifara.

2) Brojevi sa pokretnim zarezom veoma često imaju male greške zaokruživanja, čak i kada broj ima manje značajnih cifara od preciznosti računara. Često ovo prolazi nezapaženo, zato što su tako male, i zato što su brojevi skraćeni prilikom štampanja. Prema tome, upoređivanje brojeva sa pokretnim zarezom ne može uvek dati očekivane rezultate. Izvođenjem matematičkih operacija nad ovim vrednostima će dovesti do povećanja grešaka zaokruživanja.

`bool` vrednosti i uvod u `if` iskaze

U stvarnom životu, uobičajeno je postavljati pitanja (ili odgovarati na nečija pitanja) na koja se može odgovoriti sa "da" ili "ne". "Da li je danas utorak"? "Da"! "Da li je sutra četvrtak"? "Ne"!.

Ove vrste rečenica koje imaju samo dva moguća ishoda: da / tačno / istinito, ili ne / netačno / lažno su toliko uobičajene da većina programskih jezika uključuje poseban tip za rad sa njima. Taj tip se zove boolean tip.

Boolean promenljive

Boolean promenljive su promenljive koje mogu imati samo dve moguće vrednosti: `true` (1) i `false` (0).

Za deklaraciju boolean promenljive, koristi se ključna reč `bool`.

```
1   bool b;
```

Za inicijalizaciju ili dodelu vrednosti `bool` promenljivoj, koriste se ključne reči `true` i `false`.

```
1   bool b1 = true;       // inicijalizacija kopiranjem
2   bool b2(false);       // direktna inicijalizacija
3   b3 = false;           // dodela vrednosti
```

Slično kao što se unarni operator minus - može koristiti da bi celobrojna promenljiva dobila negativnu vrednost, logički operater NOT (!) se može koristiti za promenu `bool` vrednosti sa `true` na `false` ili obrnuto:

```
1   bool b1 = !true;      // b1 će imati vrednost false
2   bool b2(!false);      // b2 će imati vrednost true
```

Boolean vrednosti zapravo nisu smeštene u boolean promenljivama kao reči "true" ili "false". Umesto toga, one se čuvaju kao celobrojne vrednosti: true ima vrednost 1, a false vrednost 0. Shodno tome, kada štampamo boolean vrednosti sa std::cout, std::cout štampa 0 za false, a 1 za true:

```
#include <iostream>
int main()
{
    std::cout << true << std::endl;
    std::cout << !true << std::endl;

    bool b(false);
    std::cout << b << std::endl;
    std::cout << !b << std::endl;
    return 0;
}
```

Ovaj program štampa sledeći izlaz:

```
1
0
0
1
```

Boolean promenljive i if iskazi

Jedna od najčešćih upotreba boolean promenljivih su if iskazi. if iskazi najčešće imaju sledeći oblik:

```
if (izraz) iskaz1;
```

ili

```
if (izraz) iskaz1;
else iskaz2;
```

Izraz koji se nalazi u zagradi se naziva uslov ili uslovni izraz. U oba prethodno prikazana slučaja uslov se izračunava. Ako izraz ima nenultu vrednost, tada se izvršava iskaz1. U suprotnom slučaju, ako izraz izraz ima nultu vrednost, izvršava se iskaz2.

Slede dva primera koda.

```
if (true) // uslovni izraz ima vrednost true
   std::cout << "Uslov je tačan" << std::endl;
else
   std::cout << "Uslov je netačan" << std::endl;
```

Izlaz je:

```
Uslov je tačan
```

Sledeći primer je veoma sličan prethodnom:

```
1   bool b(false);
2   if (b) // uslovni izraz ima vrednost false
3      std::cout << "Uslov je tačan" << std::endl;
4   else
5      std::cout << "Uslov je netačan" << std::endl;
```

Izlaz je:

```
Uslov je netačan
```

U oba oblika uslov se izračunava. Ako uslov ima nenultu vrednost, tada se izvršava iskaz koji se nalazi neposredno ispod `if` iskaza, u suprotnom se izvršava iskaz koji se nalazi ispod `else` iskaza.

Char promenljive

Iako je tip podataka `char` zapravo celobrojni tip, podatke ovog tipa koristimo na drugačiji način. `char` promenljiva sadrži jednobajtni ceo broj. Međutim, umesto tumačenja `char` vrednosti kao celog broja, vrednost `char` promenljive se uobičajeno tumači kao ASCII simbol.

ASCII označava američki standardni kod za razmenu informacija i definiše poseban način predstavljanja simbola engleskog alfabeta (uz dodatak nekoliko drugih simbola) brojevima od 0 do 127 (nazvan ASCII kod). Na primer, znak 'a' ima kod 97. 'b' ima kod 98. Karakteri se uvek postavljaju između jednostrukih navodnika.

Na narednoj strani se nalazi tabela svih ASCII karaktera. Karakteri sa kodom od 0 do 31 su znakovi koji se ne mogu štampati, a najviše se koriste za formatiranje i kontrolu štampača. Većina ovih karaktera je danas zastarela.

Kodovi 32-127 su znakovi koji se mogu štampati i njih predstavljaju slova, brojevi i interpunkcijski znaci, koje većina računara koristi za prikaz osnovnog engleskog teksta.

Sledeći primer pokazuje dva različita načina inicijalizacije promenljivih tipa `char`.

```
1   char ch1(97);
2   char ch2('a');
```

Obe promenljive su inicijalizovane na vrednost karaktera `'a'`. Međutim, treba primetiti razliku u inicijalizaciji u sledećem primeru:

```
1   char ch1(8);
```

```
2   char ch2('8');
```

Kod	Simbol	Kod	Simbol	Kod	Simbol	Kod	Simbol
0	NUL (null)	32	space	64	@	96	`
1	SOH (start of header)	33	!	65	A	97	a
2	STX (start of text)	34	”	66	B	98	b
3	ETX (end of text)	35	#	67	C	99	c
4	EOT (end of transmission)	36	$	68	D	100	d
5	ENQ (enquiry)	37	%	69	E	101	e
6	ACK (acknowledge)	38	&	70	F	102	f
7	BEL (bell)	39	’	71	G	103	g
8	BS (backspace)	40	(	72	H	104	h
9	HT (horizontal tab)	41	)	73	I	105	i
10	LF (line feed/new line)	42	*	74	J	106	j
11	VT (vertical tab)	43	+	75	K	107	k
12	FF (form feed / new page)	44	,	76	L	108	l
13	CR (carriage return)	45	-	77	M	109	m
14	SO (shift out)	46	.	78	N	110	n
15	SI (shift in)	47	/	79	O	111	o
16	DLE (data link escape)	48	0	80	P	112	p
17	DC1 (data control 1)	49	1	81	Q	113	q
18	DC2 (data control 2)	50	2	82	R	114	r
19	DC3 (data control 3)	51	3	83	S	115	s
20	DC4 (data control 4)	52	4	84	T	116	t
21	NAK (negative acknowledge)	53	5	85	U	117	u
22	SYN (synchronous idle)	54	6	86	V	118	v
23	ETB (end of transmission block)	55	7	87	W	119	w
24	CAN (cancel)	56	8	88	X	120	x
25	EM (end of medium)	57	9	89	Y	121	y
26	SUB (substitute)	58	:	90	Z	122	z
27	ESC (escape)	59	;	91	[	123	{
28	FS (file separator)	60	<	92	\	124	|
29	GS (group separator)	61	=	93	]	125	}

30	RS (record separator)	62	>	94	^	126	~
31	US (unit separator)	63	?	95	_	127	DEL delete

Promenljiva ch1 je inicijalizovana celim brojem 8 koji u ASCII tabeli odgovara karakteru backspace, dok je promenljiva ch2 inicijalizovana brojem 8 numeričke tastature kome odgovara kod 88 iz ASCII tabele.

Ulazno – izlazne operacije sa karakterima

Štampanje i čitanje char promenljivih je veoma slično kao i kod prethodnih tipova promenljivih. Sledi primer štampanja karaktera na konzoli.

```
1   #include <iostream>
2   int main()
3   {
4      char ch('a');
5      std::cout << ch << std::endl;
6      return 0;
7   }
```

Slično prethodnom, moguće je sa tastature pročitati neku vrednost i smestiti je u neku char promenljivu.

Primer programa u kome se sa tastature učitava vrednost jedne promenljive tipa char dat je u nastavku. Izlaz iz programa će zavisiti od onoga što korisnik unese.

```
1   #include <iostream>
2   int main()
3   {
4      std::cout << "Unesite neki karakter na tastaturi: ";
5
6      char ch;
7      std::cin >> ch;
8      std::cout << "Uneli ste karakter: " << ch;
9      return 0;
10  }
```

Ako korisnik unese karakter a, izlaz iz programa će biti:

```
Unesite neki karakter na tastaturi: a
Uneli ste karakter: a
```

Escape sekvence

U jeziku C++ neki karakteri imaju posebno značenje. Ovi karakteri se zovu escape sekvence. Escape sekvence počinju karakterom '\' (backslash), koga prati slovo ili broj. Jedna od veoma često korišćenih escape sekvenci je '\n' koja se koristi za prelazak u novi red.

```
#include <iostream>
int main()
{
   std::cout << "Prva linija\nDruga linija" << std::endl;
   return 0;
}
```

Izlaz iz ovog programa je:

```
Prva linija
Druga linija
```

Druga veoma često korišćena escape sekvenca je '\t', koja štampa tab:

```
#include <iostream>
int main()
{
   std::cout << "Prvi deo\tDrugi deo" << std::endl;
   return 0;
}
```

Izlaz iz ovog programa je:

```
Prvi deo      Drugi deo
```

Sledi tabela svih escape sekvenci.

Ime	Simbol	Značenje
Alert	\a	Upozorenje, kao što je beep na zvučniku računara
Backspace	\b	Pomera kursor za jedno mesto unazad
Formfeed	\f	Pomera kursor na sledeću logičku stranu (danas zastarelo)
Newline	\n	Pomera kursor u sledeću liniju
Carriage return	\r	Pomera kursor na početak linije
Horizontal tab	\t	Štampa horizontalni tab
Vertical tab	\v	Štampa vertikalni tab
Single quote	\’	Štampa jednostruki navodnik
Double quote	\”	Štampa dvostruki navodnik
Backslash	\\	Štampa backslash
Question mark	\?	Štampa znak pitanja
Octal number	\(number)	Prevodi oktalni broj u char
Hex number	\x(number)	Prevodi heksadecimalni broj

U narednom primeru pokazano je kako se koristi nekoliko različitih escape sekvenci.

```
1  #include <iostream>
2  int main()
3  {
4    std::cout << "\"Tekst pod navodnicima\"\n";
5    std::cout << "Tekst sa karakterom backslash \\" << std::endl;
6    std::cout << "6F je heksadecimalni zapis karaktera \'\x6F\'";
7    return 0;
8  }
```

Izlaz je:

```
"Tekst pod navodnicima"
Tekst sa karakterom backslash \
6F je heksadecimalni zapis karaktera 'o'
```

Može se primetiti da `'\n'` i `std::endl` proizvode isti efekat, prelazak u novi red. Razlika je jedino u tome što prilikom korišćenja `'\n'` nije osigurano da će se naredba momentalno izvršiti zbog baferovanja podataka. Kod `std::endl` je to zagarantovano. Druga varijanta je za nijansu sporija, što može imati efekta prilikom zapisivanja ogromnih fajlova.

4. Operatori

Prioritet operatora i asocijativnost

U matematici, operacija je matematički račun koji na osnovu ulaznih vrednosti (operanada) proizvodi izlaznu vrednost. Uobičajene operacije (kao što su sabiranje, oduzimanje, itd.) koriste posebne simbole koji označavaju operaciju (kao što su recimo simboli +,-, itd.). Ovi simboli se nazivaju operatori. Operatori u programiranju funkcionišu na isti način, osim što imena operacija ne mogu uvek biti simboli. Operatori funkcionišu analogno funkcijama koje imaju ulazne parametre i povratnu vrednost, jedino im je "ime" skraćeno.

Kako bi pravilno izračunat izraz kao što je `4 + 2 * 3`, neophodno je znati kako operatori rade, i kojim redosledom ih treba primeniti. Redosled primene operatora u složenom izrazu se naziva prioritet operatora. Matematičkim pravilima za prioritet operatora (kojima je definisano da se množenje obavlja pre sabiranja, recimo), definisano je da pomenuti izraz treba izračunati kao `4 + (2 * 3)`, čime se dobija vrednost `10`.

U jeziku C++, kada kompajler prilikom prevođenja naiđe na neki složeni izraz, na sličan način analizira izraz i određuje kojim redosledom ga treba izvršiti. Kako bi se to izvelo, svim operatorima dodeljen je prioritet. Prvo se izvršavaju operatori sa najvišim prioritetom. Tako recimo množenje i deljenje imaju viši prioritet od sabiranja i oduzimanja.

Prema tome, `4 + 2 * 3` se izračunava kao `4 + (2 * 3)` jer množenje ima viši nivo prioriteta od sabiranja.

Ako su dva operatora sa istim nivoom prioriteta u jednom izrazu susedna jedan drugome, tada se redosled izvršavanja određuje na osnovu asocijativnosti operatora, koja govori da li se operatori izvršavaju s leva na desno ili s desna na levo. Na primer, u izrazu `3 * 4 / 2` operatori množenja i deljenja imaju isti prioritet. S obzirom da oba operatora imaju asocijativnost s leva na desno, izraz se rešava s leva na desno: `(3 * 4) / 2 = 6`.

Na sledećoj strani data je tabela najčešće korišćenih operatora u jeziku C++. U tabeli se nalaze podaci o prioritetu i asocijativnosti svih operatora.

Prioritet	Operator	Opis	Asocijativnost
1	::	Razrešenje opsega	Sleva nadesno
2	a++ a-- type() type{} a() a[] . ->	Operator inkrementiranja i dekrementiranja Operator konverzije (kastovanja) Poziv funkcije Operator indeksiranja Operator izbora (selekcije)	
3	++a --a +a -a ! ~ (type) *a &a sizeof new new[]	Operator inkrementiranja i dekrementiranja Unarni plus i minus Logičko NOT i bitwise NOT Operator kastovanja Operator dereferenciranja Adresni operator Operator sizeof Operator za dinamičko alociranje memorije	Zdesna nalevo
4	.* ->*	Pokazivač ka članu	Sleva nadesno
5	a*b a/b a%b	Množenje, deljenje, ostatak pri deljenju	
6	a+b a-b	Sabiranje, oduzimanje	
7	<< >>	Bitwise pomeranje levo i desno	
8	< <=	Relacioni operator < i <=	
	> >=	Relacioni operator > i >=	
9	== !=	Relacioni operator == i !=	
10	a&b	Bitwise I	
11	^	Bitwise ekskluzivno ILI	
12	|	Bitwise ILI	
13	&&	Logičko I	
14	||	Logičko ILI	
15	a?b:c = += -= *= /= %= <<= >>=	Ternarni uslovni operator Dodela vrednosti Sabiranje ili oduzimanje, množenje, deljenje i ostatak pri deljenju i dodela vrednosti Bitwise operator i dodela vrednosti	Zdesna nalevo
16	,	Zapeta	Sleva nadesno

Aritmetički operatori

Unarni aritmetički operatori

Unarni operatori su operatori koji imaju samo jedan operand. Postoje dva unarna aritmetička operatora: unarni plus (+) i unarni minus (-).

Operator	Simbol	Forma	Operacija
Unarni plus	+	+x	Vrednost x
Unarni minus	-	-x	Vrednost x sa suprotnim znakom

Operator unarni plus vraća vrednost operanda. Drugim rečima, +5 = 5, +k = k. Generalno posmatrano, ovaj operator je suvišan. Dodat je u velikoj meri da bi se obezbedila simetrija sa unarnim operatorom minus.

Operator unarni minus vraća operand pomnožen sa -1. Drugim rečima, ako je k = 5, -k = -5, ili ako je k = -3, -k = 3.

Bitno je razlikovati operator unarni minus od operatora oduzimanja, koji koriste isti simbol. Na primer, u izrazu k = 5 - -3; prvi minus je operator oduzimanja, a drugi operator je unarni minus.

Binarni aritmetički operatori

Binarni operatori su operatori koji imaju dva operanda. Postoji pet binarnih aritmetičkih operatora.

Operator	Simbol	Forma	Operacija
Sabiranje	+	x + y	x plus y
Oduzimanje	-	x - y	x minus y
Množenje	*	x * y	x puta y
Deljenje	/	x / y	x podeljeno y
Ostatak pri deljenju	%	x % y	ostatak pri deljenju x sa y

Operatori sabiranja, oduzimanja i množenja funkcionišu baš kao u matematici, bez ikakvih ograničenja.

Operacija deljenja i operacija ostatka pri deljenju zahtevaju dodatno objašnjenje.

Deljenje celobrojnih promenljivih i promenljivih sa pokretnim zarezom

Operator deljenja poseduje dva različita "načina" rada. Ako su oba operanda celi brojevi, operator deljenja izvršava celobrojno deljenje. Celobrojno deljenje

ignoriše ostatak pri deljenju. Na primer, `7 / 4 = 1` jer se ostatak pri deljenju ignoriše. Bitno je znati da celobrojno deljenje ne zaokružuje promenljivu.

Ako je bilo koji od dva operanda vrednost sa pokretnim zarezom, operator deljenja vrši deljenje sa pokretnim zarezom. Na primer: `7.0 / 3 = 2.333`, `7 / 3.0 = 2.333` i `7.0 / 3.0 = 2.333`.

Bitno je napomenuti da će pokušaj deljenja sa `0` (ili `0.0`) generalno dovesti do pada programa, jer su rezultati nedefinisani!

Operatori povećavanja i smanjivanja

Povećanje – inkrementiranje (dodavanje 1) i smanjenje – dekrementiranje (oduzimanje 1) vrednosti neke promenljive su uobičajene operacije u C++ jeziku. Za te operacije postoje posebni operatori. Postoje, zapravo, dve verzije ovih operatora: prefiksna verzija i postfiksna verzija.

Operator	Simbol	Forma	Operacija
Prefiksni inkrement	++	++x	Povećava vrednost x za 1, zatim koristi x
Postfiksni inkrement	--	--x	Smanjuje vrednost x za 1, zatim koristi x
Prefiksni inkrement	++	x++	Koristi x, zatim povećava vrednost x za 1
Prefiksni inkrement	--	x--	Koristi x, zatim smanjuje vrednost x za 1

Sledi primer koji pokazuje razliku između prefiksne i postfiksne verzije operatora.

```
#include <iostream>
int main()
{
   int x = 5, y = 5;
   cout << x << " " << y << endl;
   cout << ++x << " " << --y << endl; // prefiksni operator
   cout << x << " " << y << endl;
   cout << x++ << " " << y-- << endl; // postfiksni operator
   cout << x << " " << y << endl;

   return 0;
}
```

Izlaz iz ovakvog programa bi bio:

```
5 5
6 4
6 4
6 4
7 3
```

U liniji 6, promenljive x i y se povećavaju / smanjuju pre nego što se koriste u iskazu, tako da se na ekranu štampaju njihove nove vrednosti. U liniji 8, prvobitne vrednosti (x = 6, y = 4) se štampaju na konzoli, a zatim se povećavaju / smanjuju. Iz tog razloga se promene usled korišćenja postfiksnog operatora ne pojavljuju do sledeće linije koda.

Posledice korišćenja operatora inkrementiranja i dekrementiranja

Za funkciju ili iskaz se kaže da ima sporedni efekat ako menjaju neko stanje koje je izvan njihovog opsega, kao što je recimo dodela vrednosti promenljivoj, modifikovanje nekog objekta, pozivanje funkcija koje menjaju stanje neke globalne ili statičke promenljive, itd. Drugim rečima, sporedni efekat je rezultat operatora, izraza, iskaza ili funkcije koji ostaje čak i nakon što se operator, izraz, iskaz ili funkcija izvrše. Operatori inkrementiranja i dekrementiranja takođe imaju sporedne efekte.

U većini slučajeva sporedni efekti operatora inkrementiranja i dekrementiranja su korisni:

```
1  int x = 5;
2  int y = 0;
3  y = x++;
4  std::cout << x << std::endl;
5  std::cout << y << std::endl;
```

U prethodnom primeru operator ++ ima sporedni efekat trajne promene vrednosti x. Pošto se iskaz završi, y će imati vrednost 5 a x vrednost 6. Operator ++ ima sporedni efekat povećanja x. Štampanje promenljive x na konzoli ima sporedni efekat promene konzole. Izlaz iz ovakvog programa će biti:

```
6
5
```

Međutim, sporedni efekti mogu dovesti do neočekivanih rezultata:

```
#include <iostream>
int add(int x, int y)
{
   return x + y;
}

int main()
{
   int x = 5;
   int value = add(x, ++x);
   std::cout << value;
   return 0;
}
```

C++ jezik ne definiše redosled izjednačavanja vrednosti argumenata funkcije sa njenim parametrima. Ako se u prethodnom primeru prvo izjednači levi argument pa zatim desni, to znači da će vrednost parametara funkcije biti (5, 6), što daje rezultat 11. Međutim, ako se prvo izjednači desni argument pa zatim levi, to znači da su argumenti funkcije (6, 6), što daje rezultat 12! Ovaj problem se javlja samo zato što jedan od argumenata funkcije add(int x, int y) ima sporedni efekat. Izlaz iz prethodnog programa zavisiće od kompajlera.

Uslovni operator

Uslovni operator (? :) (poznat i kao operator "aritmetički if") je jedini ternarni operator u jeziku C++ (operator sa tri operanda).

Operator	Simbol	Forma	Operacija
Uslovni operator	?:	c ? x : y	Ako je c različito od 0, koristi x, u suprotnom y

Ovaj operator omogućava skraćivanje if/else iskaza. Sledeća forma if/else iskaza:

```
if (condition)
    expression;
else
    other_expression;
```

može biti napisana u drugačijoj formi:

```
(condition) ? expression : other_expression;
```

Operandi uslovnog operatora moraju biti samo izrazi a ne iskazi. Na primer, sledeći if/else iskaz:

```
if (condition)
```

```
    x = some_value;
else
    x = some_other_value;
```

može biti napisan u sledećoj formi:

```
x = (condition) ? some_value : some_other_value;
```

Uslovni operater ima vrlo nizak prioritet. Ako se radi bilo šta drugo osim dodele vrednosti rezultata nekoj promenljivoj, kompletan iskaz ? : treba staviti u zagrade. Takođe, dobra praksa je da se uslov (condition) stavlja u zagrade, zbog čitljivosti koda.

Sledi primer jedne konkretne upotrebe uslovnog operatora.

```
1   #include <iostream>
2   int main()
3   {
4      int x = 5, y = 6;
5
6      int z = (x < y) ? x : y;
7      std::cout << z;
8
9      z = (x > y) ? x : y;
10     std::cout << z;
11     return 0;
12  }
```

Izlaz će biti:

```
5
6
```

Uslovni operator daje mogućnost da se pojednostave jednostavni `if`/`else` iskazi, naročito prilikom dodele vrednosti promenljivama ili vraćanja rezultata kao povratne vrednosti funkcije. Ovakvih iskaza će biti u budućim primerima.

Relacioni operatori

Postoji šest relacionih operatora: >, <, >=, <=, ==, !=. Tabela sa svim relacionim operatorima se nalazi na narednoj strani. Sledi primer upotrebe svih šest operatora.

```
1   #include <iostream>
2   int main()
3   {
4      std::cout << "Unesite prvi ceo broj: ";
5      int x;
6      std::cin >> x;
```

```
7       std::cout << "Unesite drugi ceo broj: ";
8       int y;
9       std::cin >> y;
10
11      if(x==y)std::cout << x <<" je jednako "<< y <<"\n";
12      if(x!=y)std::cout << x <<" je različito od"<< y <<"\n";
13      if(x>y) std::cout << x <<" je veće od "<< y <<"\n";
14      if(x<y) std::cout << x <<" je manje od "<< y <<"\n";
15      if(x>=y)std::cout << x <<" je veće ili jednako "<< y <<"\n";
16      if(x<=y)std::cout << x <<" je manje ili jednako "<< y <<"\n";
17      return 0;
18  }
```

Primer rezultata programa:

```
Unesite prvi ceo broj: 4
Unesite drugi ceo broj: 5
4 je različito od 5
4 je manje od 5
4 je manje ili jednako 5
```

Operator	Simbol	Forma	Operacija
Veće	>	x > y	Tačno ako je x veće od y, u suprotnom netačno
Manje	<	x < y	Tačno ako je x manje od y, u suprotnom netačno
Veće ili jednako	>=	x >= y	Tačno ako je x veće ili jednako y, u suprotnom netačno
Manje ili jednako	<=	x >= y	Tačno ako je x manje ili jednako y, u suprotnom netačno
Jednako	==	x == y	Tačno ako je x jednako y, u suprotnom netačno
Različito	!=	x != y	Tačno ako je x različito od y, u suprotnom netačno

Upoređivanje vrednosti sa pokretnim zarezom

Direktno upoređivanje vrednosti sa pokretnim zarezom pomoću bilo kog od ovih operatora nije preporučljivo. Razlog tome je greška u zaokruživanju kod brojeva sa pokretnim zarezom, koja može izazvati neočekivane rezultate.

Sledi primer u kome greške zaokruživanja uzrokuju neočekivane rezultate:

```
1   #include <iostream>
2   int main()
3   {
4       double d1(100 - 99.99);  // trebalo bi da je jednako 0.01
5       double d2(10 - 9.99);    // trebalo bi da je jednako 0.01
6
7       if (d1 == d2)
8           std::cout << "d1 == d2" << "\n";
9       else if (d1 > d2)
10          std::cout << "d1 > d2" << "\n";
```

```
    else if (d1 < d2)
        std::cout << "d1 < d2" << "\n";
    return 0;
}
```

Ovaj program će prouzrokovati neočekivani rezultat:

```
d1 > d2
```

Oba broja, `d1` i `d2`, su blizu `0.01`, ali između njih ipak postoji razlika koja je uzrokovana greškom zaokruživanja.

Logički operatori

Relacioni operateri mogu da se koriste kako bi se testiralo da li je određeno stanje tačno ili netačno. Oni su ograničeni na testiranje samo jednog stanja u jednom iskazu. Međutim, često je potrebno proveriti da li su višestruki uslovi istovremeno istiniti.

Jezik C++ obezbeđuje tri logička operatora:

Operator	Simbol	Forma	Operacija
Logičko NE	!	!x	Tačno ako je x netačno ili netačno ako je x tačno
Logičko I	&&	x && y	Tačno ako su x i y tačno, u suprotnom netačno
Logičko ILI	||	x || y	Tačno ako su x ili y tačno, u suprotnom netačno

Logičko NE

Ako je operand na koga se primenjuje logičko NE tačan, rezultat operatora logično NE će biti netačan. Ako je operand na koga se primenjuje logičko NE netačan, rezultat operatora logično NE će biti tačan. Drugim rečima, logičko NE menja logičku vrednost od tačno do netačno, i obrnuto.

Desni operand	Simbol
Tačno	Netačno
Netačno	Tačno

Sledi primer koda u kome se koristi operator logičko NE.

```
    int x = 5;
    int y = 7;

    if ( !(x == y) )
        std::cout << "x nije jednako y";
    else
        std::cout << "x je jednako y";
    return 0;
```

Izlaz iz ovog programa bi bio:

```
"x nije jednako y";
```

Važno je napomenuti da po prioritetu operatora Logičko NE ima viši prioritet od relacionih operatora. Stoga, bi iskaz: `if(!x == y)` dao, na prvi pogled, neočekivan rezultat. Rezultat izraza u zagradi bi bio `false`, zato što bi se izvršila negacija od x, a pošto je x različito od nule, tj. "tačno", čitav izraz bi bio netačan. U tom slučaju program bi prošao kroz deo koda koji se nalazi unutar `else` iskaza.

Logičko ILI

Operator logičko ILI se koristi za testiranje da li je bilo koji od dva uslova tačan. Ako su levi ili desni operand tačni, operator logičko ILI vraća "tačno". Takođe, ako su oba operanda tačna, rezultat logičkog ILI će biti "tačno".

Levi operand	Desni operand	Rezultat
Netačno	Netačno	Netačno
Netačno	Tačno	Tačno
Tačno	Netačno	Tačno
Tačno	Tačno	Tačno

Sledi primer upotrebe operatora logičko ILI.

```
1   #include <iostream>
2   int main()
3   {
4       std::cout << "Unesite broj: ";
5       int value;
6       std::cin >> value;
7
8       if (value== 0 || value== 1)
9           std::cout << "Uneli ste 0 ili 1" << std::endl;
10      else
11          std::cout << "Niste uneli 0 ili 1" << std::endl;
12      return 0;
13  }
```

Zavisno od toga koje se vrednosti unesu prilikom startovanja ovakvog programa, program će štampati dve različite poruke. U slučaju da se unese 0 ili 1, program će štampati:

```
Uneli ste 0 ili 1
```

a u bilo kom drugom slučaju će štampati:

```
Niste uneli 0 ili 1
```

Logičko I

Operator logičko I se koristi za testiranje da li su dva uslova tačna. Ako su oba uslova tačna, logičko I vraća vrednost "tačno".

Levi operand	Desni operand	Rezultat
Netačno	Netačno	Netačno
Netačno	Tačno	Netačno
Tačno	Netačno	Netačno
Tačno	Tačno	Tačno

Sledi primer upotrebe operatora logičko I.

```
#include <iostream>
int main()
{
    std::cout << "Unesite broj: ";
    int value;
    std::cin >> value;

    if (value > 10 && value < 20)
        std::cout << "Uneli ste broj izmedju 10 i 20\n";
    else
        std::cout << "Niste uneli broj izmedju 10 and 20\n";
    return 0;
}
```

Operatori logičko ILI i logičko I se mogu povezati tako da se u jednom iskazu upoređuje mnogo veći broj uslova:

```
   if (value > 10 && value < 20 && value != 16)
    // uradi nešto
   else
    // uradi nešto drugo

   if (value == 1 || value == 2 || value == 3)
    // uradi nešto
   else
    // uradi nešto drugo
```

Kombinacija operatora logičko I i logičko ILI

Kombinovanje logičnih I i logičkih ILI operatora u istom izrazu često se ne može izbeći, ali kod takvih složenih iskaza postoji velika mogućnost da se nešto previdi i da se program ponaša suprotno očekivanjima. Razlog tome je postojanje razlike u prioritetu operatora logičko I i logičko ILI. Logičko I ima viši prioritet od logičkog ILI, pa će se, prema tome, operatori logičko I izvršavati pre operatora logičko ILI (osim ako nisu u zagradama).

Prilikom kombinovanja logičkog I i logičkog ILI operatora u istom izrazu, dobra praksa je da se svaki operator i njegovi operandi smeste u zagrade. Ovo pomaže u sprečavanju grešaka nastalih usled razlika u prioritetu operatora, a takođe olakšava čitanje koda i jasno definiše način na koji treba da se izračuna izraz. Na primer, umesto pisanja `value1 && value2 || value3 && value4`, bolje je napisati `(value1 && value2) || (value3 && value4)`.

5. Opseg važenja promenljivih i složeni tipovi podataka

Blokovi

Blok iskaza, takođe poznat pod nazivom složeni iskaz, je grupa iskaza koje kompajler tretira kao da je reč o jednom iskazu. Blokovi počinju simbolom “{“ a završavaju se simbolom “}”. Iskazi koje treba izvršiti nalaze se između ovih vitičastih zagrada. Blokovi se mogu koristiti na bilo kom mestu gde se može naći i običan iskaz. Na kraju bloka nije neophodan simbol tačka-zarez.

U prethodnim poglavljima su se već pojavljivali primeri blokova.

```
1   #include <iostream>
2
3   int add(int x, int y)
4   { // početak bloka
5       return x + y;
6   } // kraj bloka
7
8   int main()
9   {// početak bloka
10
11      // iskazi unutar bloka
12      int value(0);
13      add(3, 4);
14
15      return 0;
16  } // kraj bloka (nije potreban karakter ;)
```

Blokovi mogu biti ugnežđeni unutar drugih blokova. Blokovi se mogu koristiti na bilo kom mestu gde može da stoji i samo jedan iskaz. Na primer, `if` iskaz izvršava jedan iskaz ukoliko je uslov tačan. Shodno tome, `if` iskaz se može promeniti tako da se umesto jednog iskaza koji se izvršava ukoliko je uslov ispunjen, koristi ugnežđeni blok iskaza koji će se izvršiti ako je pomenuti uslov ispunjen. U nastavku je dat primer.

```
#include <iostream>

int main()
{
   std::cout << "Unesite ceo broj: ";
   int value;
   std::cin >> value;

   if (value >= 0)
   { // početak ugneždenog bloka
      std::cout << value <<" je pozitivan ceo broj (ili nula)";
      std::cout << std::endl;
      std::cout << "Njegova dvostruka vrednost je " << value * 2;
      std::cout << std::endl;
   } // kraj ugneždenog bloka
   else
   { // početak drugog ugneždenog bloka
      std::cout << value << " je negativan ceo broj";
      std::cout << std::endl;
      std::cout << "Njegova pozitivna vrednost je "<< -value;
      std::cout << std::endl;
   } // kraj drugog ugneždenog bloka

   return 0;
}
```

Ako korisnik unese broj 2, izlaz iz programa će biti:

```
Unesite ceo broj: 2
2 je pozitivan ceo broj (ili nula)
Njegova dvostruka vrednost je 4
```

Ako korisnik unese broj -3, izlaz iz programa će biti:

```
Unesite ceo broj: -3
-3 je negativan ceo broj
Njegova pozitivna vrednost je 3
```

Takođe je moguće staviti blok unutar bloka. Pritom, ne postoji praktično ograničenje koliko ugneždenih blokova može postojati. Međutim, uobičajeno je da postoji 3 do 4 ugneždena bloka po dubini. Ako se u funkciji javi potreba za većim brojem blokova ugneždenih jedan unutar drugog, verovatno bi je trebalo razdvojiti na više manjih.

U nastavku sledi primer ugneždenih blokova.

```
#include <iostream>

int main()
{
   std::cout << "Unesite ceo broj: ";
   int value;
   std::cin >> value;

   if (value >  0)
   {
      if ((value % 2) == 0)
      {
         std::cout << value << " je pozitivan paran broj";
         std::cout << std::endl;
      }
      else
      {
         std::cout << value << " je pozitivan neparan broj";
         std::cout << std::endl;
      }
   }
   return 0;
}
```

Lokalne promenljive, opseg i trajanje

Kada je reč o promenljivama, korisno je razdvojiti pojmove opsega i trajanja. Opseg promenljive određuje mesto gde je promenljiva dostupna. Trajanje promenljive je određeno mestom gde je stvorena i gde je uništena. Ova dva koncepta su često povezana.

Promenljive definisane unutar bloka nazivaju se lokalnim promenljivama. Opseg i trajanje lokalne promenljive su ograničeni mestom gde je definisana i završetkom bloka unutar koga se nalazi.

Sledi primer jedne jednostavne funkcije:

```
int main()
{
   int n(5); // promenljiva n je definisana i inicijalizovana
   double d(4.0); // promenljiva d je definisana i inicijalizovana

   return 0;
} // promenljive n i d izlaze iz opsega i bivaju uništene
```

Pošto su promenljive `i` i `d` definisane unutar bloka koji definiše `main()` funkciju, obe promenljive se uništavaju kada se funkcija `main()` završi.

Promenljive definisane unutar ugneždenih blokova uništavaju se čim se unutrašnji blok završi.

```
int main() // spoljašnji blok
{
   int n(5); // promenljiva n je definisana

   { // ugneždeni blok
       double d(4.0);//promenljiva d je definisana
   } // promenljiva d izlazi iz opsega i uništava se
   // promenljiva d se ne može koristiti ovde jer je već uništena
   return 0;
} // promenljiva n izlazi iz opsega i uništava se
```

Promenljive definisane unutar nekog bloka mogu se videti samo unutar tog bloka. S obzirom da svaka funkcija ima svoj blok, promenljive u jednoj funkciji ne mogu se videti iz druge funkcije:

```
void someFunction()
{
    int value(4); // promenljiva value je definisana

    // promenljiva value se ovde može koristiti

} // promenljiva value izlazi iz opsega i uništava se

int main() // spoljašnji blok
{
    // promenljiva value se ne može koristiti ovde

    someFunction();

    // promenljiva value se takođe ne može koristiti ovde

    return 0;
}
```

Ugneždeni blokovi se smatraju delom spoljašnjeg bloka, unutar koga su definisani. Shodno tome, promenljive definisane u spoljašnjem bloku mogu se videti unutar ugneždenog bloka:

```
#include <iostream>

int main()
{ // početak spoljašnjeg bloka
   int x(5);

   { // početak ugneždenog bloka
      int y(7);
      // promenljive x i y se mogu koristiti ovde
      std::cout << x << " + " << y << " = " << x + y;
```

```
    } // promenljiva y izlazi iz opsega i uništava se
    // promenljiva y se ne može koristiti ovde jer je već uništena
    return 0;
} // promenljiva x izlazi iz opsega i uništava se
```

Skrivanje imena

Promenljiva definisana unutar ugneždenog bloka može imati isto ime kao promenljiva unutar spoljašnjeg bloka. Kada se ovo desi, ugneždena promenljiva "sakriva" spoljašnju promenljivu. Ovo se naziva skrivanje ili maskiranje imena (eng. name hiding ili shadowing).

```
#include <iostream>

int main()
{ // spoljašnji blok
    int apples(5); // promenljiva apples u spoljašnjem bloku

    if (apples >= 5) // promenljiva outer iz spoljašnjeg bloka
    { // ugneždeni blok
        int apples;// maskira promenljivu outer iz spoljašnjeg bloka

        // promenljiva apples iz spoljašnjeg bloka nije dostupna
        // zbog identičnog imena


        apples = 10; // dodeljuje vrednost 10 promenljivoj apples
                     // iz ugneždenog bloka

        std::cout << apples << '\n';
    } // promenljiva apples iz ugneždenog bloka se uništava

    // promenljiva apples iz spoljašnjeg bloka je ponovo dostupna

    // štampanje vrednosti promenljive apples iz spoljašnjeg bloka
    std::cout << apples << '\n';

    return 0;
} // promenljiva apples iz spoljašnjeg bloka se uništava
```

Izlaz iz ovog programa je:

```
10
5
```

U prethodnom programu prvo je definisana promenljiva `apple` u spoljašnjem bloku. Zatim je definisana druga promenljiva, koja je takođe nazvana `apple`, u ugneždenom bloku. Dodela vrednosti `10` promenljivoj `apple` se u ovom slučaju odnosi na promenljivu definisanu u ugneždenom bloku. Posle štampanja ove

vrednosti, promenljiva apple iz ugneždenog bloka se uništava, tako da ostaje promenljiva apple iz spoljašnjeg bloka sa prvobitnom vrednošću (5), koja se zatim štampa. Program se, dakle, izvršava kao da su promenljive imale različita imena.

Međutim, da nije postojala definicija promenljive apple u ugneždenom bloku, promenljiva apple u ugneždenom bloku bi bila ona koja je definisana u spoljašnjem bloku, tako da bi se dodela vrednosti 10 promenljivoj apple odnosila na promenljivu koja je definisana u spoljašnjem bloku.

Iako je ovo korektno sa stanovišta sintakse jezika C++, ovakvo davanje imena promenljivama ne treba koristiti u praksi, jer ne donosi ništa korisno sa stanovišta preglednosti i razumljivosti koda.

Promenljive treba definisati u minimalnom mogućem opsegu. Na primer, ako se promenljiva koristi samo u ugneždenom bloku, nju treba definisati unutar tog bloka.

```
1   #include <iostream>
2
3   int main()
4   {
5      // ne treba ovde definisati promenljivu y
6      {
7         // promenljiva y se koristi samo u ugneždenom bloku.
8         // ovde je treba definisati
9         int y(5);
10        std::cout << y;
11     }
12
13     // u suprotnom, promenljiva y će i ovde biti dostupna
14
15     return 0;
16  }
```

Ograničavanjem opsega promenljive smanjuje se složenost programa, jer se smanjuje broj aktivnih promenljivih. Zatim, lakše se uočava gde se promenljiva koristi. Promenljiva definisana unutar bloka može se koristiti samo unutar tog bloka ili ugneždenih blokova.

Globalne promenljive

Promenljive definisane izvan blokova nazivaju se globalnim promenljivama. Globalne promenljive imaju statičko trajanje, što znači da se kreiraju kada se program pokreće i uništavaju tek kada se završi. Opseg globalnih promenljivih je

definisan fajlom u kome se nalaze, što znači da su vidljive do kraja fajla u kome su definisane.

Po konvenciji, globalne promenljive se definišu na vrhu datoteke, ispod naredbi za uključivanje dodatnih fajlova (naredba `#include`), ali iznad bilo kakvog koda u tom fajlu. Sledi primer definisanja nekoliko globalnih promenljivih:

```
#include <iostream>

// promenljiva deklarisane izvan blokova su globalne promenljive
int g_x;          // globalna promenljiva g_x
const int g_y(2); // globalna promenljiva g_y

void doSomething()
{
    // globalne promenljive su vidljive svuda
    g_x = 3;
    std::cout << g_y << "\n";
}
```

```
int main()
{
    doSomething();

    // globalne promenljive su vidljive svuda
    g_x = 5;
    std::cout << g_y << "\n";

    return 0;
}
```

U ovom primeru postoji i definicija jedne konstante, `const int g_y(2)`. Ovakvom definicijom ”promenljive”, tj. konstante onemogućava se promena njene vrednosti (postoji način da se čak i konstantama promeni vrednost, pomoću pointera i eksplicitnih konverzija, ali to neće biti prikazano u ovom kursu).

Lokalna promenljiva sa istim imenom kao globalna promenljiva sakriva globalnu promenljivu unutar bloka gde je lokalna promenljiva definisana. Međutim, operator razrešenja opsega (`::`) može se koristiti kako bi se pristupilo globalnoj promenljivoj na mestu gde postoji i lokalna promenljiva sa istim imenom.

```
#include <iostream>
int value(5); // globalna promenljiva

int main()
{
    int value = 7; // sakriva globalnu promenljivu value
    value++; // uvećava vrednost lokalne promenljive value
    ::value--; // umanjuje vrednost globalne promenljive value

    std::cout << "globalna promenljiva value: " << ::value << "\n";
    std::cout << "lokalna promenljiva value: " << value << "\n";
    return 0;
} // lokalna promenljiva value se uništava
```

Izlaz iz ovakvog koda je:

```
globalna promenljiva value: 4
lokalna promenljiva value: 8
```

Međutim, ovakav način davanja naziva promenljivama treba izbeći, jer bolji sigurno postoji. Davanjem istog imena ne dobija se ništa korisno već samo problemi.

Generalno posmatrano, globalne promenljive (posebno nekonstantne) treba izbegavati. Možda najznačajniji razlog zbog koga su nekonstantne globalne promenljive opasne je to što se njihove vrednosti mogu promeniti bilo kojom funkcijom koja se poziva, a ne postoji jednostavan način da programer zna da će se to dogoditi.

```
// deklaracija globalne promenljive
int g_mode;

void doSomething()
{
   g_mode = 2; // dodela vrednosti 2 globalnoj promenljivoj g_mode
}

int main()
{
   g_mode = 1; // dodela vrednosti 1 globalnoj promenljivoj g_mode

   doSomething();

// Programer očekuje da promenljiva g_mode ima vrednost 1
// Funkcija doSomething je dodelila vrednost 2 promenljivoj g_mode

   if (g_mode == 1) std::cout << "Nema detektovanih pretnji.\n";
   else             std::cout << "Pokretanje nuklearnih raketa!\n";

   return 0;
}
```

U prethodnom primeru, programer je dodelio vrednost 1 promenljivoj g_mode, a zatim pozvao funkciju doSomething(). Osim ako programer nije imao eksplicitno saznanje da funkcija doSomething() treba da promeni vrednost promenljive g_mode, verovatno ne očekuje da će funkcija doSomething() promeniti njenu vrednost! Zbog toga ostatak main() funkcije neće funkcionisati onako kako programer očekuje (a svet će već biti uništen!).

Nekonstantne globalne promenljive čine svaku funkciju potencijalno opasnom, a programer ne može lako saznati čija upotreba je opasna a čija ne. Lokalne promenljive su mnogo sigurnije jer druge funkcije ne mogu direktno uticati na njih.

Statičke promenljive

Korišćenjem promenljivih sa automatskim trajanjem (uobičajene promenljive), njihovo trajanje je ograničeno krajem bloka u kome su definisane. Dodavanjem ključne reči static kod lokalnih promenljivih menja se trajanje promenljive sa automatskog na statičko. Statička promenljiva zadržava svoju vrednost čak i nakon što je napušten opseg u kojem je definisana! Statičke promenljive se samo jednom stvaraju i inicijalizuju, a traju sve dok se program ne završi.

Najlakši način da se prikaže razlika između trajanja automatskih i statičkih promenljivih je sledeći primer. Svaki put kad se pozove funkcija incrementAndPrint(), definiše se promenljiva value i dodeljuje joj se vrednost 1. Funkcija incrementAndPrint() uvećava vrednost promenljive value za 1, a zatim odštampa njenu vrednost 2. Kada se funkcija incrementAndPrint() završi, promenljiva izlazi iz opsega i uništava se.

```
1   #include <iostream>
2
3   void incrementAndPrint()
4   {
5      int value = 1; // automatsko trajanje promenljive value
6      ++value;
7      std::cout << value << std::endl;
8   } // promenljiva value se uništava
9
10  int main()
11  {
12     incrementAndPrint();
13     incrementAndPrint();
14     incrementAndPrint();
15  }
```

Prema tome, izlaz iz ovog programa je:

```
2
2
2
```

U kodu koji sledi ponovljen je prethodni primer uz jednu veoma malu izmenu. Jedina razlika između ovog i prethodnog programa je lokalna promenljiva `value`, koja sada ima statičko trajanje, odnosno definisana je kao `static` promenljiva. U ovom programu, promenljiva `value` je deklarisana kao `static`, pa se stoga `value` definiše i inicijalizuje samo jednom. Kad izađe iz opsega, ona nije uništena. Svaki put kada se poziva funkcija `incrementAndPrint()`, vrednost `value` je ona koja je prethodno ostavljena.

```
#include <iostream>

void incrementAndPrint()
{
   static int value = 1; // statčko trajanje
   ++value;
   std::cout << value << std::endl;
} // promenljiva value se ne uništava

int main()
{
   incrementAndPrint();
   incrementAndPrint();
   incrementAndPrint();
}
```

Izlaz iz ovog programa će biti:

```
2
3
4
```

Jedna od čestih upotreba lokalnih promenljivih sa statičkim trajanjem je implementacija generatora jedinstvenih identifikatora u programu. Kada u programu postoji veliki broj sličnih objekata, često je korisno dodeliti svakom od njih jedinstven identifikacioni broj tako da se mogu identifikovati. Ovo je veoma lako realizovati pomoću lokalne promenljive sa statičkim trajanjem:

```
int generateID()
{
   static int itemID = 0;
   return itemID++;
}
```

Prvi put kada se ova funkcija pozove, vratiće vrednost `0`. Drugi put, ona vraća `1`. Svaki sledeći put kada se pozove, vratiće broj za jedan veći od onog u prethodnom

pozivu. Na ovaj način je obezbeđeno da će se svaki put kada se funkcija pozove vratiti broj koji je sigurno drugačiji od svih prethodnih. S obzirom da je `itemID` lokalna promenljiva, nju ne može promeniti bilo koja druga funkcija.

Prostor imena (namespace)

Konflikt imena se javlja kada su dva imena dovedena u isti opseg, i kompajler ne može da odluči koje od ta dva će koristiti (ovo se odnosi na nazive složenih tipova podataka, globalnih promenljivih, funkcija, itd.). Kada se to dogodi, kompajler ili linker će prijaviti grešku jer nema dovoljno informacija za rešavanje ovakve dvosmislenosti. Kako programi postaju sve veći i veći, broj imena se povećava, što zauzvrat dovodi do povećanja verovatnoće konflikta imena.

U sledećem primeru, `f.h` i `g.h` su datoteke zaglavlja (eng. header fajlovi) koje sadrže funkcije koje rade različite stvari, ali imaju isto ime i iste parametre.

`f.h`:

```
1  // Funkcija doSomething() vraća vrednost zbira svojih parametara
2  int doSomething(int x, int y)
3  {
4     return x + y;
5  }
```

`g.h`:

```
1  // Funkcija doSomething() vraća vrednost razlike svojih parametara
2
3  int doSomething(int x, int y)
4  {
5     return x - y;
6  }
```

`main.cpp`:

```
1  #include "f.h"
2  #include "g.h"
3  #include <iostream>
4
5  int main()
6  {
7     std::cout << doSomething(4, 3); // koja funkcija doSomething ???
8     return 0;
9  }
```

Ovakav kod će prouzrokovati grešku, jer kompajler nema informaciju koju od funkcija `doSomething()` da koristi. Kako bi se razrešile ovakve neodređenosti, uveden je pojam `namespace` (prostor imena).

Prostor imena definiše oblast koda u kojoj su svi identifikatori garantovano jedinstveni. Podrazumevano, globalne promenljive i funkcije su definisane u globalnom prostoru imena (global namespace).

```
1  int g_x = 5;
2
3  int f(int x)
4  {
5     return -x;
6  }
```

U prethodnom primeru, oba imena, ime globalne promenljive g_x i ime funkcije `f()`, definisana su u globalnom prostoru imena.

Kako bi se izbegli problemi u kojima dva nezavisna dela koda imaju konflikt imena kada se koriste zajedno, C++ dozvoljava definisanje novih prostora imena pomoću ključne reči `namespace`. Sve što je deklarisano unutar korisnički definisanog prostora imena pripada tom prostoru imena, a ne globalnom prostoru imena. Sledi primer prethodno definisanih zaglavlja, napisanih uz korišćenje prostora imena:

`f.h`:

```
1  namespace F
2  {
3     // Jedna verzija funkcije doSomething()
4     int doSomething(int x, int y)
5     {
6        return x + y;
7     }
8  }
```

`g.h`:

```
1  namespace G
2  {
3     // Još jedna verzija funkcije doSomething()
4     int doSomething(int x, int y)
5     {
6        return x - y;
7     }
8  }
```

Funkcija `doSomething()`, koja je definisana unutar fajla `f.h`, je sada unutar `F` prostora imena, a funkcija `doSomething()` definisana unutar fajla `g.h` je unutar `G` prostora imena. Na ovaj način se kompajleru obezbeđuje informacija koju od dve funkcije sa istim imenom da koristi. Međutim, sada nije moguće pozvati funkciju `doSomething()` onako kako je to prethodno učinjeno, već je neophodno naglasiti

kom prostoru imena ona pripada. Za ovo se koristi operator razrešenja opsega `::`. Sledi primer poziva različitih verzija funkcije `doSomething()`, smeštenih u dva odvojena prostora imena.

```
#include "f.h"
#include "g.h"
#include <iostream>

int main(void)
{
   std::cout << F::doSomething(4, 3) << '\n';
   std::cout << G::doSomething(4, 3) << '\n';
   return 0;
}
```

Ispravno je deklarisati blokove prostora imena na više lokacija (u više fajlova ili na više mesta unutar jednog fajla). Sve deklaracije unutar bloka prostora imena smatraju se delom prostora imena.

Takođe, moguće je praviti i prostore imena ugneždene unutar drugih prostora imena.

```
#include <iostream>

namespace F
{
   namespace G
   {
      const int g_x = 5;
   }
}

int main()
{
   std::cout << F::G::g_x;
   return 0;
}
```

Međutim, ovo treba izbegavati, jer je primarna namena prostora imena da se izbegne konflikt imena, a ne da se pomoću njih gradi programska hijerarhija.

Upotreba ključne reči `using`

Postoje dva načina upotrebe ključne reči `using`: `using` deklaracija i `using` naredba.

```
1  #include <iostream>
2
3  int main()
4  {
5     using std::cout;
6     cout << "Pozdrav!";
7
8     return 0;
9  }
```

Deklaracija `using std::cout;` govori kompajleru da će se koristiti objekat `cout` iz prostora imena `std`. Dakle, kompajler će, kad god vidi `cout`, pretpostaviti da se misli na `std::cout`. Ako postoji konflikt između imena `std::cout` i neke druge upotrebe `cout`-a, `std::cout` će imati prednost. Zbog toga u liniji 6 može da se piše `cout` umesto `std::cout`.

Ovo ne štedi mnogo vremena i truda u ovakvom trivijalnom primeru, ali ako se objekat `cout` intenzivno koristi unutar funkcija, deklaracija `using` može kod učiniti čitljivijim. Analogno tome, trebalo bi napraviti posebne deklaracije za svako ime koje se koristi (npr. jedna za `std::cout`, jedna za `std::cin`, a jedna za `std::endl`).

Drugi način upotrebe ključne reči `using` je `using` naredba. Sledi primer koda.

```
1  #include <iostream>
2
3  int main()
4  {
5     using namespace std;
6     cout << "Pozdrav!";
7
8     return 0;
9  }
```

Iskaz `using namespace std;` govori kompajleru će se koristiti sve što se nalazi u prostoru imena `std`. Ako kompajler pronađe ime koje ne prepoznaje, proveriće da li to ime postoji u `std` prostoru imena. Shodno tome, kada kompajler naiđe na liniju sa `cout` (koji ne prepoznaje), proveriće da li postoji u prostoru imena `std`. Ukoliko postoji konflikt imena između `std::cout` i neke druge upotrebe `cout`-a, kompajler će to označiti kao grešku.

Nabrojivi tip podataka

C++ sadrži veći broj ugrađenih tipova podataka. Međutim, ovi tipovi nisu uvek dovoljni za sve rešavanje problema na koje se nailazi prilikom razvoja softvera.

Jezik C++ daje mogućnost programerima da kreiraju sopstvene tipove podataka. Ovakvi tipovi podataka nazivaju se korisničkim tipovima podataka.

Jedan od najjednostavnijih korisničkih tipova je nabrojivi tip (enumerated type). To je takav tip podataka gde je svaka moguća vrednost podatka definisana kao simbolička konstanta, koja se naziva enumerator. Nabrojivi tipovi se deklarišu pomoću ključne reči `enum`. Sledi primer definicije nabrojivog tipa.

```
#include <iostream>

enum Color
{
    COLOR_BLACK,
    COLOR_RED,
    COLOR_BLUE,
    COLOR_GREEN,
    COLOR_WHITE,
    COLOR_CYAN,
    COLOR_YELLOW,
    COLOR_MAGENTA
}; // deklaracija nabrojivog tipa se mora završiti karakterom ;

// Definisanje promenljivih nabrojivog tipa
Color paint = COLOR_WHITE;
Color house(COLOR_BLUE);
```

Deklarisanjem tipa `enum` ne rezerviše se memorija u računaru. Kada se definiše promenljiva čiji je tip prethodno deklarisani `enum` (kao što je promenljiva `paint` u prethodnom primeru), za tu promenljivu se u trenutku njenog definisanja alocira memorijski prostor. Uobičajeno je da imena enumerisanih tipova počinju velikim slovom, a nazivi enumeratora najčešće imaju sva velika slova. Kada se enumeratori različitih nabrojivih tipova nalaze u istom prostoru imena, ime bilo kog enumeratora se ne može koristiti u više enumerisanih tipova:

```
#include <iostream>

enum Color
{
   RED,
   BLUE, // BLUE se nalazi u globalnom prostoru imena
   GREEN
};

enum Feeling
{
   HAPPY,
   TIRED,
   BLUE //greška, BLUE je već definisano u nabrojivom tipu Color
};
```

Zato je preporuka da imena enumeratora imaju neki standardni prefiks. U ovom slučaju to bi mogli biti COLOR_ (COLOR_RED, COLOR_BLUE, COLOR_GREEN) i FEELING_ (FEELING_HAPPY, FEELING_TIRED, FEELING_BLUE).

Svakom enumeratoru se automatski dodeljuje celobrojna vrednost na osnovu njegove pozicije u listi enumeratora. Podrazumevano je da se prvom enumeratoru dodeli celobrojna vrednost 0, a svaki sledeći ima vrednost za jedan veću od prethodnog.

```
1   #include <iostream>
2
3   enum Color
4   {
5       COLOR_BLACK,  // dodeljena vrednost 0
6       COLOR_RED,    // dodeljena vrednost 1
7       COLOR_BLUE,   // dodeljena vrednost 2
8       COLOR_GREEN,  // dodeljena vrednost 3
9       COLOR_WHITE,  // dodeljena vrednost 4
10      COLOR_CYAN,   // dodeljena vrednost 5
11      COLOR_YELLOW, // dodeljena vrednost 6
12      COLOR_MAGENTA // dodeljena vrednost 7
13  };
14
15  Color paint(COLOR_WHITE);
16  std::cout << paint;
```

Moguće je eksplicitno definisati vrednost enumeratora. Ove celobrojne vrednosti mogu biti pozitivne ili negativne i mogu imati istu vrednost kao neki drugi enumerator. Svaki nedefinisani enumerator ima vrednost koja je za jedan veća od prethodnog enumeratora.

```
1   enum Animal
2   {
3       ANIMAL_CAT = -3,
4       ANIMAL_DOG,          // dodeljena vrednost -2
5       ANIMAL_PIG,          // dodeljena vrednost -1
6       ANIMAL_HORSE = 5,
7       ANIMAL_GIRAFFE = 5, // ima istu vrednost kao ANIMAL_HORSE
8       ANIMAL_CHICKEN       // dodeljena vrednost 6
9   };
10
```

U prethodnom primeru enumeratorima ANIMAL_HORSE i ANIMAL_GIRAFFE dodeljene su iste vrednosti. Ovo je korektno sa stanovišta sintakse jezika C++, ali u tom slučaju ta dva enumeratora postaju međusobno zamenljiva. Iako je ovo dozvoljeno, ipak treba izbegavati dodelu iste vrednosti različitim enumeratorima istog nabrojivog tipa.

Nabrojivi tipovi su korisni za dokumentovanje i bolju čitljivost koda, kada se javi potreba za predstavljanjem određenog, unapred definisanog, skupa stanja.

Funkcije često vraćaju celobrojnu vrednost pozivaocu sa ciljem da daju informaciju o grešci, kada je nešto u funkciji izvršeno pogrešno. Uobičajeno je da se mali negativni brojevi koriste kako bi se predstavile različite moguće greške u kodu, kao u primeru koji sledi.

```
int readFileContents()
{
   if (!openFile())
      return -1;
   if (!readFile())
      return -2;
   if (!parseFile())
      return -3;

    return 0;
}
```

Međutim, korišćenje brojeva kao što je predstavljeno u ovom primeru ne opisuje šta se konkretno dogodilo. Alternativna metoda bi bila pomoću nabrojivog tipa.

```
enum ParseResult
{
    SUCCESS = 0,
    ERROR_OPENING_FILE = -1,
    ERROR_READING_FILE = -2,
    ERROR_PARSING_FILE = -3
};

ParseResult readFileContents()
{
    if (!openFile())
        return ERROR_OPENING_FILE;
    if (!readFile())
        return ERROR_READING_FILE;
    if (!parsefile())
        return ERROR_PARSING_FILE;

    return SUCCESS;
}
```

Ovakav kod je mnogo lakši za čitanje i razumevanje nego prethodni, u kome su provrtane vrednosti bili obični negativni brojevi. Pozivalac funkcije može testirati povratnu vrednost funkcije prema odgovarajućem enumeratoru, što je lakše razumeti u odnosu na upoređivanje rezultata sa nekom celobrojnom vrednošću.

```
    // Segment koda

    if (readFileContents() == SUCCESS)
    {
        // radi nešto
    }
    else
    {
        // štampanje poruke o grešci
    }
```

Upotreba ključne reči `typedef` i pseudonimi tipova podataka

Ključna reč `typedef` omogućava kreiranje pseudonima za neki tip podataka i njegovo korišćenje umesto stvarnog tipa. Da bi deklarisali pseudonim nekog tipa koristi se ključna reč `typedef`, zatim se navodi tip za koji je potrebno definisati pseudonim, i na kraju ime pseudonima.

```
typedef double distance_t; // distance_t je pseudonim za tip double

// Sledeća dva iskaza su ekvivalentna
double howFar;
distance_t howFar;
```

Jedna upotreba `typedef` naredbe je da pomogne u dokumentaciji i čitljivosti koda. Imena tipova podataka kao što su `char`, `int`, `long`, `double` i `bool` su dobra za opisivanje tipa promenljive koju neka funkcija vraća. Međutim, češće je korisno znati za koju svrhu se koristi povratna vrednost funkcije.

```
int GradeTest();
```

U prethodnom primeru iz deklaracije se vidi da je povratna vrednost ceo broj, ali se ne zna pouzdano šta znači: Da li je to broj pitanja u testu? Ili broj indeksa studenta? Neki kod greške? `int` sam po sebi ne govori ništa. Međutim, koristeći povratni tip `testScore_t` postaje očigledno da funkcija vraća vrednost promenljive koja predstavlja rezultat testa.

```
int GradeTest();
typedef int testScore_t;
testScore_t GradeTest();
```

`typedef` takođe omogućava da se promeni tip nekog podataka bez potrebe za promenama velikog dela koda. Na primer, ako je tip `short` korišćen za čuvanje broja indeksa studenta, a zatim se pojavila potreba za tipom `long` umesto `short`, moralo bi da se prođe kroz veliki deo koda kako bi se takva izmena primenila u kodu. Verovatno bi bilo teško razdvojiti `short` promenljive koje se koriste za

čuvanje promenljivih koje predstavljaju brojeve indeksa studenata od ostalih `short` promenljivih koje se koriste za nešto drugo.

Međutim, ukoliko je korišćena naredba `typedef`, sve što treba da se uradi je promena samo jednog iskaza. Iskaz:

```
1   typedef short student_ID;
```

treba promeniti u iskaz:

```
1   typedef long student_ID;
```

Jedna veoma velika prednost korišćenja `typedef` je to što se mogu koristiti za sakrivanje detalja specifičnih za neku platformu. Na nekim platformama, ceo broj je 2 bajta, a na drugim je 4. To znači da je korišćenje tipa `int` za čuvanje više od 2 bajta informacija potencijalno opasno kod pisanja koda koji treba da bude nezavisan od platforme za koju se kompajlira.

Strukture

Postoji mnogo situacija u programiranju gde je potrebno više od jedne promenljive kako neki entitet bio predstavljen. Na primer, za predstavljanje osobe neophodno je imati podatak o imenu i prezimenu, i opciono o rođendanu, visini, težini ili nekom drugom značajnom podatku. To bi se moglo uraditi na sledeći način:

```
1   std::string myName;
2   int myBirthYear;
3   int myBirthMonth;
4   int myBirthDay;
5   int myHeightInches;
6   int myWeightPounds;
```

Definisano je šest nezavisnih promenljivih koje nisu grupisane na bilo koji način. Ako je potrebno nekoj funkciji proslediti informacije o osobi, morale bi da se proslede sve promenljive pojedinačno. Zatim, ukoliko je potrebno sačuvati informacije o nekom drugom, neophodno bi bilo definisati još šest promenljivih za svaku novu osobu! Očigledno je da ovo nije bas najbolji način za čuvanje više informacija o jednom entitetu.

Jezik C++ omogućava kreiranje vlastitih, korisnički definisanih, agregatnih tipova podataka. Agregatni tip podataka je tip podataka koji zajedno grupiše više individualnih promenljivih. Jedan od najjednostavnijih agregatnih tipova podataka je struktura. Struktura omogućava grupisanje promenljivih različitih tipova u jednu složenu promenljivu.

Deklaracija i definisanje struktura

Strukture su korisnički definisan tip podataka. Da bi se koristile, neophodno je prvo saopštiti kompajleru kako izgleda struktura. Strukture se deklarišu pomoću ključne reči `struct`. Sledi primer deklaracije strukture.

```
1   struct Employee
2   {
3       short id;
4       int age;
5       double wage;
6   };
```

Ovo govori kompajleru da je deklarisana struktura pod imenom `Employee`. Struktura `Employee` sadrži tri promenljive: `short` promenljivu `id`, `int` promenljivu `age` i `double` promenljivu `wage`. Ove promenljive koje su deo strukture nazivaju se članovi (ili polja). Struktura `Employee` je samo deklaracija – time je kompajleru saopšteno da će struktura imati članove (promenljive) i u ovom trenutku se ne alocira memorijski prostor za bilo šta. Po konvenciji, imena struktura počinju velikim slovom da bi se razlikovale od imena promenljivih.

Kada je definisana struktura `Employee`, može se jednostavno definisati i promenljiva tipa `Employee`:

```
1   Employee joe;
```

Ovo definiše promenljivu tipa `Employee` čiji je naziv `joe`. Kao i kod standardnih promenljivih, definisanje promenljive strukturnog tipa alocira memorijski prostor za tu promenljivu.

Moguće je definisati više promenljivih istog strukturnog tipa:

```
1   Employee joe;
2   Employee frank;
```

Pristup članovima strukture

Kada se definiše promenljiva kao što je `Employee joe`, `joe` je podatak strukturnog tipa i sadrži sve promenljive – članove. Za pristup članovima strukture koristi se operator izbora (selekcije) člana. Sledi primer inicijalizacije članova korišćenjem operatora izbora članova.

```
Employee joe;
joe.id = 14;
joe.age = 32;
joe.wage = 24.15;

Employee frank;
frank.id = 15;
frank.age = 28;
frank.wage = 18.27;
```

Kao i kod običnih promenljivih, promenljive – članovi strukture nisu inicijalizovane, tako da ih je neophodno inicijalizovati ručno.

U prethodnom primeru, vrlo je lako reći koje promenljive – članovi pripadaju promenljivoj `joe` a koje promenljivoj `frank`. To obezbeđuje mnogo viši nivo organizacije u odnosu na situaciju kada se koriste pojedinačne promenljive. Osim toga, članovi promenljivih `joe` i `frank` imaju iste nazive, što obezbeđuje konzistentnost između više promenljivih istog strukturnog tipa.

Promenljive – članovi strukture se ponašaju isto kao obične promenljive, tako da je moguće obaviti uobičajene operacije nad njima:

```
Employee joe;
Employee frank;

int totalAge = joe.age + frank.age;

if (joe.wage > frank.wage)
    std::cout << "Joe makes more than Frank\n";
else if (joe.wage < frank.wage)
    std::cout << "Joe makes less than Frank\n";
else
    std::cout << "Joe and Frank make the same amount\n";

// Frank je dobio povišicu
frank.wage += 2.50;

// Na današnji dan je Joe rodjen
++joe.age; // starost se uvećava za 1
```

Inicijalizacija struktura

Inicijalizacija struktura dodeljivanjem vrednosti članu po članu je malo spora (u smislu pisanja koda), tako da jezik C++ podržava brži način inicijalizacije struktura pomoću liste za inicijalizaciju. Ovo omogućava inicijalizaciju nekih ili svih članova strukture u momentu definisanja promenljive.

```
struct Employee
{
    short id;
    int age;
    double wage;
};

// joe.id=1, joe.age = 32, joe.wage = 60000.0
Employee joe = {1,32,60000.0};
// frank.id=2, frank.age=28, frank.wage=0.0 (podrazumevano)
Employee frank = {2,28};
```

Strukture i funkcije

Velika prednost korišćenja struktura nad pojedinačnim promenljivama leži u činjenici da se funkciji mogu proslediti podaci strukturnog tipa, koja zatim može koristiti sve članove strukture.

U sledećem primeru, podatak strukturnog tipa `Employee` prosleđen je funkciji `printInformation()`. Ovo omogućava da se funkciji ne prosleđuje svaka promenljiva pojedinačno. Takođe, ako se ikada javi potreba za nekim novim podatkom unutar strukturnog tipa `Employee`, neće biti potrebe za promenama u deklaraciji funkcije ili njenih poziva u programu. Definicije funkcija (tela funkcija) koje rade sa ovakvim strukturnim podacima će možda pretrpeti neke izmene, ali je to daleko manja izmena u odnosu na situaciju kada je neophodno menjati i deklaracije funkcija – a posledično i sve njihove pozive.

```
struct Employee
{
    short id;
    int age;
    double wage;
};

void printInformation(Employee employee)
{
    std::cout << "ID:   " << employee.id << "\n";
    std::cout << "Age:  " << employee.age << "\n";
    std::cout << "Wage: " << employee.wage << "\n";
}
```

```
int main()
{
    Employee joe = { 14, 32, 24.15 };
    Employee frank = { 15, 28, 18.27 };

    // Štampanje informacija o zaposlenom “joe“
    printInformation(joe);

    std::cout << "\n";

    // Štampanje informacija o zaposlenom “frank“
    printInformation(frank);

    return 0;
}
```

Funkcija takođe može vratiti strukturni tip podataka, što je jedan od načina da funkcija vrati više promenljivih sadržanih unutar jedne promenljive.

```
#include <iostream>

struct Point3d
{
    double x;
    double y;
    double z;
};

Point3d getZeroPoint()
{
    Point3d temp = { 0.0, 0.0, 0.0 };
    return temp;
}

int main()
{
    Point3d zero = getZeroPoint();

    if (zero.x == 0.0 && zero.y == 0.0 && zero.z == 0.0)
        std::cout << "The point is zero\n";
    else
        std::cout << "The point is not zero\n";

    return 0;
}
```

Prethodni program štampa:

```
The point is zero
```

Ugneždene strukture

Promenljive strukturnog tipa mogu sadržavati i druge promenljive strukturnog tipa.

```
struct Employee
{
    short id;
    int age;
    double wage;
};

struct Company
{
    Employee CEO; // podatak tipa Employee u strukturi Company
    int numberOfEmployees;
};

Company myCompany;
```

U ovom slučaju, ako je potrebno pristupiti podatku o plati CEO, jednostavno se može upotrebiti operator za izbor članova strukture dva puta: `myCompany.CEO.wage;`

Ovim je izabran član generalni direktor (CEO) iz podatka `myCompany`, a zatim je izabran član plata direktora (`wage`).

Za inicijalizaciju ugneždenih struktura se mogu koristiti ugneždene liste:

```
struct Employee
{
    short id;
    int age;
    float wage;
};

struct Company
{
    Employee CEO;
    int numberOfEmployees;
};

Company myCompany = {{ 1, 42, 60000.0f }, 5 };
```

Veličina strukture predstavlja zbir veličina svih članova. Može se reći da je minimalna veličina strukture jednaka zbiru veličina njenih članova. Ponekad kompajler dodaje izvesne međuprostore zbog povećanja performansi.

S obzirom da deklaracije struktura ne zahtevaju alociranje memorijskog prostora, i ukoliko je potrebno deklarisani strukturni tip koristiti u više fajlova, deklaracija strukturnog tipa može biti stavljena u datoteku zaglavlja (eng. header file). Takvu datoteku je naredbom `#include` moguće uključiti gde god postoji potreba za podacima takvog strukturnog tipa.

Strukture su veoma značajne u jeziku C++ i predstavljaju prvi veliki korak ka objektno – orijentisanom konceptu programiranja. U poglavljima koja slede biće obrađene klase kao složeni tip podataka, koje su u osnovi bazirane na strukturama. Dobro razumevanje strukturnog tipa će učiniti mnogo lakšim prelazak na mnogo moćniji tip podataka – klase.

6. Kontrola toka programa

Uvod

Kada se program pokrene, CPU počinje izvršavanje od početka `main()` funkcije, izvršava iskaze koji se u njoj nalaze, i završava se završetkom `main()` funkcije. Sekvenca iskaza koje CPU izvršava se naziva putanja izvršavanja programa (ili, kraće, putanja). Većina programa koji su do sada prikazani bili su pravolinijski programi. Takvi programi imaju sekvencijalni tok – tj. imaju istu putanju (izvršavaju iste iskaze) svaki put kada se pokrenu (čak iako se ulazni podaci menjaju).

Međutim, programima ovakve strukture gotovo uvek nije moguće rešiti neki konkretan problem. Na primer, ako se od korisnika očekuje da izvrši neki izbor, a korisnik unese nevažeći izbor, u idealnom slučaju neophodno je da se od korisnika zahteva ponovni izbor. Ovo nije moguće u pravolinijskom programu. Zatim, postoje slučajevi u kojima je nešto potrebno učiniti više puta, ali je broj ponavljanja nepoznat u vreme kompajliranja programa. Na primer, ako je potrebno odštampati sve cele brojeve od 0 do nekog broja koji je uneo korisnik, to se ne može učiniti sve dok korisnik ne unese broj.

Jezik C++, slično kao i ostali viši programski jezici, sadrži naredbe kontrole toka programa, koje omogućavaju programeru da promeni putanju izvršavanja programa. Postoji nekoliko različitih tipova kontrola toka programa koji će biti obrađeni u okviru ovog poglavlja.

Iskaz `if`

Osnovna vrsta uslovnog grananja u C++ je `if` iskaz. Iskaz `if` ima oblik:

```
if (izraz)
    iskaz
```

ili:

```
if (izraz)
    iskaz
else
    iskaz2
```

`izraz` se naziva uslovni izraz. Ako `izraz` ima vrednost `true`, iskaz se izvršava. Ako `izraz` ima vrednost `false`, naredba `else` se izvršava, ukoliko ona postoji.

U nastavku sledi jednostavan program u kome je upotrebljen `if` iskaz.

```
#include <iostream>

int main()
{
   std::cout << "Unesite broj: ";
   int x;
   std::cin >> x;

   if (x > 10)
      std::cout << x << "je vece od 10\n";
   else
      std::cout << x << "nije vece od 10\n";

   return 0;
}
```

Iskaz `if` izvršava samo jedan iskaz ako je izraz tačan, a `else` izvršava samo jedan iskaz ako je izraz netačan. Da bi se izvršilo više iskaza, mogu se koristiti blokovi.

```
#include <iostream>

int main()
{
   std::cout << " Unesite broj: ";
   int x;
   std::cin >> x;

   if (x > 10)
   {
      std::cout << "Uneli ste " << x << "\n";
      std::cout << x << "je vece od 10\n";
   }
   else
   {
      std::cout << "Uneli ste " << x << "\n";
      std::cout << x << "nije vece od 10\n";
   }

   return 0;
}
```

Ulančani `if` iskazi

Moguće je napraviti povezani `if` – `else` iskaz.

```
#include <iostream>

int main()
{
   std::cout << "Unesite broj: ";
   int x;
   std::cin >> x;

   if (x > 10)
      std::cout << x << "je vece od 10\n";
   else if (x < 10)
      std::cout << x << "je manje od 10\n";
   else
      std::cout << x << "je jednako 10\n";

   return 0;
}
```

Ugneždeni `if` iskazi

Moguće je ugnezditi `if` iskaze u okviru drugih `if` iskaza:

```
#include <iostream>
int main()
{
   std::cout << "Unesite broj: ";
   int x;
   std::cin >> x;

   if (x > 10) // spoljašnji if iskaz
      // ovakav stil pisanja koda je loš
      if (x < 20) // unutrašnji if iskaz
         std::cout << x << "je izmedju 10 i 20\n";

      // na koji if iskaz se odnosi ovaj else iskaz?
      else
         std::cout << x << "je vece od 20\n";

   return 0;
}
```

Prethodni program može na prvi pogled izgledati dvosmisleno. Na koji od dva `if` iskaza se odnosi `else` iskaz? Iskaz `else` se uvek odnosi na poslednji neupareni iskaz `if` koji se nalazi u istom bloku. Dakle, u gore navedenom programu, iskaz `else` je uparen sa unutrašnjim `if` iskazom.

Da bi se izbegle takve nejasnoće kod složenih iskaza, uobičajeno je da se svaki iskaz, `if` ili `else`, ograniči posebnim blokovima.

Sledi izmenjena verzija programa napisanog bez dvosmislenosti.

```
#include <iostream>
int main()
{
   std::cout << "Unesite broj: ";
   int x;
   std::cin >> x;

   if (x > 10)
   {
      if (x < 20)
         std::cout << x << "je izmedju 10 i 20\n";
      else
         std::cout << x << "je vece od 20\n";
   }
   return 0;
}
```

Sada je mnogo jasnije da `else` iskaz pripada unutrašnjem `if` iskazu.

Smeštanje unutrašnjeg `if` iskaza u blok omogućava da se iskaz `else` dodeli spoljašnjem `if` iskazu

```
#include <iostream>
int main()
{
   std::cout << "Unesite broj: ";
   int x;
   std::cin >> x;

   if (x > 10)
   {
      if (x < 20)
         std::cout << x << "je izmedju 10 i 20\n";
   }
   else // ovaj else iskaz se odnosi na spoljašnji if iskaz
      std::cout << x << "je manje od 10\n";

   return 0;
}
```

Upotreba bloka govori kompajleru da će se `else` iskaz pridružiti `if` iskazu koji se nalazi ispred bloka koji se završava neposredno ispred `else` iskaza. Bez bloka, `else` iskaz bi se pridružio najbližem `if` iskazu, a to bi u ovom slučaju bio unutrašnji `if` iskaz.

Korišćenje logičkih operatora sa `if` iskazima

Takođe, `if` iskazi mogu proveravati više uslova zajedno, koristeći logičke operatore.

```
#include <iostream>

int main()
{
   std::cout << "Unesite ceo broj: ";
   int x;
   std::cin >> x;

   std::cout << "Unesite jos jedan ceo broj: ";
   int y;
   std::cin >> y;

   if (x > 0 && y > 0)
      std::cout << "Oba broja su pozitivna\n";
   else if (x > 0 || y > 0)
      std::cout << "Jedan od unetih brojeva je pozitivan\n";
   else
      std::cout << "Nijedan broj nije pozitivan\n";

   return 0;
}
```

Iskazi `if` se često koriste za proveru grešaka. Na primer, kod izračunavanja kvadratnog korena, vrednost prosleđena funkciji koja računa kvadratni koren bi trebalo da bude nenegativan broj:

```
#include <iostream>
#include <cmath> // u ovoj datoteci se nalazi funkcija sqrt()

void printSqrt(double value)
{
   if (value >= 0.0)
      std::cout << "Kvadratni koren broja "<< value
                <<" je "<< sqrt(value)<<"\n";
   else
      std::cout << "Greska: " << value << " je negativan broj\n";
}
```

Iskazi `if` se mogu koristiti i za rani izlazak iz funkcije, kada se funkcija ne izvršava do kraja usled neispunjavanja nekog zahtevanog uslova (primer je u nastavku).

```
#include <iostream>

enum ErrorCode
{
   ERROR_SUCCESS = 0,
   ERROR_NEGATIVE_NUMBER = -1
```

```
};
```

U ovom programu, ako je vrednost parametra `value` negativna, funkcija vraća šifru greške u obliku promenljive nabrojivog tipa. Postoji veliki broj situacija gde `if` iskazi nalaze primenu. To će se najbolje uočiti kroz rešavanje konkretnih zadataka.

```
ErrorCode doSomething(int value)
{
   // ako je value negativan broj
   if (value < 0)
     // raniji izlazak iz funkcije i vraćanje koda greške
      return ErrorCode::ERROR_NEGATIVE_NUMBER;

   // kod koji se izvršava kada value ima nenegativnu vrednost

   return ErrorCode::ERROR_SUCCESS;
}

int main()
{
   std::cout << "Unesite pozitivan broj: ";
   int x;
   std::cin >> x;

   if (doSomething(x) == ErrorCode::ERROR_NEGATIVE_NUMBER)
   {
      std::cout << "Uneli ste negativan broj!\n";
   }
   else
   {
      std::cout << "Program je uspesno izvrsen!\n";
   }

   return 0;
}
```

Jedna veoma česta greška koja se javlja pri korišćenju `if` iskaza je korišćenje operatora dodele vrednosti (=) umesto relacionog operatora (==).

```
#include <iostream>

int main()
{
   std::cout << "Unesite 0 ili 1: ";
   int x;
   std::cin >> x;
   if (x = 0) // dodela vrednosti umesto relacionog operatora
      std::cout << "Uneli ste 0";
   else
      std::cout << "Uneli ste 1";

```

```
13      return 0;
14  }
```

Prethodni primer će uvek štampati tekst "Uneli ste 1", zato što je promenljivoj x dodeljena vrednost 0. Iskaz `if` će tretirati promenljivu x kao uslov. S obzirom da je u jeziku C++ vrednost 0 ekvivalentna vrednosti `false`, program će uvek izvršavati `else` iskaz.

Iskaz `switch`

Iako je moguće povezati nekoliko `if` – `else` iskaza, za takav kod bi se moglo reći da nije baš čitljiv.

```
#include <iostream>

enum Colors
{
    COLOR_BLACK,
    COLOR_WHITE,
    COLOR_RED,
    COLOR_GREEN,
    COLOR_BLUE
};

void printColor(Colors color)
{
    if (color == COLOR_BLACK)
        std::cout << "Crna";
    else if (color == COLOR_WHITE)
        std::cout << "Bela";
    else if (color == COLOR_RED)
        std::cout << "Crvena";
    else if (color == COLOR_GREEN)
        std::cout << "Zelena";
    else if (color == COLOR_BLUE)
        std::cout << "Plava";
    else
        std::cout << "Nepoznata boja";
}
int main()
{
    printColor(COLOR_GREEN);

    return 0;
}
```

Potreba za ovakvim povezanim `if – else` iskazima je veoma česta. Jezik C++ pruža alternativni uslovni operator grananja koji se zove `switch`. Sledi kod za funkciju `printColor()`, napisanu pomoću `switch` iskaza.

```
1   void printColor(Colors color)
2   {
3      switch (color)
4      {
5         case COLOR_BLACK:
6            std::cout << "Crna";
7            break;
8         case COLOR_WHITE:
9            std::cout << "Bela";
10           break;
11        case COLOR_RED:
12           std::cout << "Crvena";
13           break;
14        case COLOR_GREEN:
15           std::cout << "Zelena";
16           break;
17        case COLOR_BLUE:
18           std::cout << "Plava";
19           break;
20        default:
21           std::cout << "Nepoznata boja";
22           break;
23     }
24  }
```

Iskazi `switch` rade veoma jednostavno: izraz koji se nalazi unutar zagrada `switch` iskaza se upoređuje sa vrednostima labela svih `case` – ova . U slučaju kada se ove vrednosti podudaraju, vrednost `switch` izraza i neke `case` labele, iskazi koji se nalaze unutar tog `case`-a će se izvršiti. Ukoliko ni u jednom slučaju ne dođe do podudaranja, izvršiće se oni iskazi koji se nalaze posle `default` labele (ukoliko oni postoje).

Zbog načina implementacije, `switch` iskazi su obično efikasnije rešenje od lanca `if – else` iskaza.

Struktura `switch` iskaza.

Iskaz `switch` počinje ključnom rečju `switch`, a zatim izrazom čija se vrednost izračunava. Obično je ovaj izraz samo jedna promenljiva, ali može biti i nešto složenije poput nX + 2 ili nX - nY. Jedino ograničenje za ovaj izraz je da on mora biti celobrojnog tipa (`char`, `short`, `int`, `long`, `long long`, ili `enum`).

Iza iskaza `switch` sledi blok. Unutar bloka, koriste se labele za definisanje svih vrednosti sa kojima treba da se upoređuje vrednost `switch` izraza. Postoje dve vrste labela: `case` i `default`.

Labela `case`

Lablela `case` se deklariše korišćenjem ključne reči `case` iza koje sledi konstantan izraz (izraz čija je vrednost konstanta). Drugim rečima, to može biti broj (kao što je recimo broj 2, 5, ili bilo koji drugi ceo broj), promenljiva nabrojivog tipa (kao što je COLOR_RED), ili konstanta (kao što je k, pri čemu je k promenljiva definisana kao `const int`). Konstantni izraz koji sledi iza labele `case` se testira na jednakost prema izrazu koji se nalazi iza ključne reči `switch`. Ako se podudaraju, izvršiće se iskazi koji se nalaze u okviru tog `case`-a. Vrednosti svih `case` labela moraju biti jedinstvene. Drugim rečima, nije moguće uraditi sledeće:

```
switch (x)
{
    case 4:
    case 4:  // nije dozvoljeno, već je korišćena vrednost 4!
    case COLOR_BLUE: // nije dozvoljeno, COLOR_BLUE ima vrednost 4!
};
```

Moguće je imati više različitih `case` labela koje se odnose na iste iskaze. Primer za to je sledeća funkcija koja koristi više `case`-ova da bi testirala da li je parametar `'c'` ASCII broj.

```
bool isDigit(char c)
{
    switch (c)
    {
        case '0': // ako c ima vrednost 0
        case '1': // ili ako c ima vrednost 1
        case '2': // ili ako c ima vrednost 2
        case '3': // ili ako c ima vrednost 3
        case '4': // ili ako c ima vrednost 4
        case '5': // ili ako c ima vrednost 5
        case '6': // ili ako c ima vrednost 6
        case '7': // ili ako c ima vrednost 7
        case '8': // ili ako c ima vrednost 8
        case '9': // ili ako c ima vrednost 9
            return true; // vraća true
        default:
            return false; // vraća false
    }
}
```

U slučaju da je `c` ASCII broj, prvi iskaz koji se nalazi iza `case`-a koji se poklapa sa `switch` izrazom biće izvršen, što je u ovom slučaju "`return true`".

Labela `default`

Druga vrsta labele je podrazumevana labela (često se naziva i "podrazumevani slučaj"). Podrazumevana labela se deklariše korišćenjem ključne reči `default`. Kod koji se nalazi ispod ove labele se izvršava ako nijedan od `case`-ova ne odgovara `switch` izrazu. Labela `default` nije obavezna. Može postojati samo jedna `default` labela unutar `switch` iskaza. Uobičajeno je da se ona nalazi na poslednjem mestu, iako sintaksom jezika to nije strogo definisano.

Iskaz `switch` i propadanje

Jedna od veoma značajnih osobina iskaza `switch` (koju početnici često zaborave) je način kako se izvršavaju iskazi kada dođe do podudaranja nekog `case`-a sa `switch` izrazom. Kada dođe do podudaranja, izvršavanje počinje od prvog iskaza koji sledi iza podudarenog `case`-a i nastavlja se sve dok se u kodu ne naiđe na neki od sledeća četiri uslova:

1) Došlo se do kraja `switch` bloka,

2) Pojavljuje se iskaz `return`,

3) Pojavljuje se iskaz `goto`, (ovo je najbolje zaboraviti da postoji!!![1])

4) Pojavljuje se iskaz `break`.

Shodno tome, ako je program ušao u neki `case`, a nijedan od ovih uslova za prekid ne bude ispunjen u okviru tog `case`-a, program će posle poslednjeg iskaza u tom `case`-u nastaviti da izvršava iskaze koji se nalaze u narednim `case`-ovima!

```
1   switch(2)
2   {
3      case 1: // Ne poklapa se
4          std::cout << 1 << '\n'; // preskače se
5      case 2: // Poklapa se!
6          std::cout << 2 << '\n'; // Izvršavanje se nastavlja ovde
7      case 3:
8          std::cout << 3 << '\n'; // Ovo se takođe izvršava
9      case 4:
10         std::cout << 4 << '\n'; // Ovo se takođe izvršava
11     default:
12         std::cout << 5 << '\n'; // Ovo se takođe izvršava
13  }
```

[1] U jezicima C i C++ postoji ključna reč `goto`. Međutim, kada god se ukaže “potreba” za korišćenjem ove naredbe, gotovo sigurno je postojalo elegantnije rešenje za taj problem. Upotrebu ove naredbe treba izbegavati jer “skakanje“ po programskom kodu predstavlja veoma lošu programersku praksu.

Izlaz iz ovakvog koda bi bio:

```
2
3
4
5
```

Ovo najčešće nije ono što stvarno treba da se uradi i skoro nikada nije želja programera. Ovakvo izvršavanje iskaza, koje prelazi iz jednog `case`-a u drugi, naziva se propadanjem. U retkim slučajevima gde za propadanjem postoji stvarna potreba, uobičajena je praksa ostaviti komentar u kojem je navedeno da je propadanje namerno urađeno.

Iskaz `break`

Iskaz `break` govori kompajleru da je završen iskaz `switch` u kome se program trenutno nalazi. Kada program dođe do `break` iskaza, izvršavanje će se nastaviti od prvog iskaza posle završetka bloka `switch` iskaza.

Sledi primer koji odgovara prethodnom primeru, ali sa pravilno umetnutim `break` iskazima.

```
1   switch(2)
2   {
3      case 1: // Ne poklapa se
4           std::cout << 1 << '\n'; // Preskače se
5           break;
6      case 2: // Poklapa se!
7           std::cout << 2 << '\n'; // Izvršavanje se nastavlja ovde
8           break; // iskaz break prekida swithc iskaz
9      case 3:
10          std::cout << 3 << '\n';
11          break;
12     case 4:
13          std::cout << 4 << '\n';
14          break;
15     default:
16          std::cout << 5 << '\n';
17          break;
18  }
19  // Izvršavanje se nastavlja ovde
```

Sada, kada se `case` 2 poklopi, broj 2 će se štampati na konzoli, a iskaz `break` će dovesti do prekida tekućeg `switch`-a. Ostali `case`-ovi će biti preskočeni.

U okviru jednog `case`-a je moguće imati više iskaza bez definisanja novog bloka.

```
switch(1)
{
    case 1:
        std::cout << 1;
        foo();
        std::cout << 2;
        break;
    default:
        std::cout << "podrazumevani slucaj\n";
        break;
}
```

Petlja `while`

Petlja `while` je najjednostavnija petlja u jeziku C++. Deklaracija je veoma slična deklaraciji `if` iskaza:

```
while (izraz)
    iskaz;
```

Ova petlja se deklariše korišćenjem ključne reči `while`. Kada se izvršava `while` petlja, izraz se izračunava. Ako je njegova vrednost različita od nule, odnosno `true`, iskaz se izvršava.

Međutim, za razliku od `if` iskaza, kada se iskaz izvrši program se vraća na vrh petlje `while` i proces se ponavlja.

Sledeći program štampa sve brojeve od `0` do 9 korišćenjem `while` petlje.

```
#include <iostream>

int main()
{
   int count = 0;
   while (count < 10)
   {
      std::cout << count << " ";
      ++count;
   }
   std::cout << "Uradjeno!";

   return 0;
}
```

Izlaz iz ovog programa je:

```
0 1 2 3 4 5 6 7 8 9 Uradjeno!
```

Inicijalno, brojač (`count`) je postavljen na vrednost 0. Ispituje se uslov da li je brojač manji od `10` (`0 < 10`), što je tačno, tako da se blok iskaza unutar `while`

petlje izvršava. Prvi iskaz odštampa 0, a u drugom se brojač uveća za 1, i dobije vrednost 1. Program se vraća na vrh petlje while. Uslov count < 10 je i sada zadovoljen (jer je 1 < 10), pa se blok unutar while petlje ponovo izvršava. Blok će se više puta izvršavati sve dok count ne dobije vrednost 10, kada uslov 10 < 10 više neće biti tačan (njegova vrednost biće false) i petlja će se završiti.

U programu se može dogoditi čak i da se while petlja ne izvrši ni jednom. To bi se dogodilo u prethodnom primeru da je inicijalna vrednost count bila 15. Tada iskaz 15 < 10 ne bi bio tačan, i program bi odmah došao do iskaza u liniji 11.

Beskonačne petlje

Ako je izraz u while deklaraciji uvek tačan, petlja će se izvršavati do beskonačnosti. Ovo se zove beskonačna petlja. Sledi primer beskonačne petlje.

```
1   #include <iostream>
2
3   int main()
4   {
5      int count = 0;
6      while (count < 10) // ovaj uslov neće nikada biti netačan
7         std::cout << count << " "; // ponavlja se izvršavanje
8
9      return 0; // ova linija se nikada neće izvršiti
10  }
```

Pošto se count nikada ne uvećava u ovom programu, izraz count < 10 će uvek biti tačan. Shodno tome, petlja nikada neće prestati, a program će zauvek štampati "0 0 0 0 0 ...".

Nekada se i namerno prave beskonačne petlje.

```
1   while(1) // ili while (true)
2   {
3     // ovakva petlja će se beskonačno izvršavati
4   }
```

Jedini način za izlazak iz beskonačne petlje su iskazi: return, break ili goto; poziv funkcije exit(); ili da korisnik nasilno prekine izvršavanje programa.

Promenljiva petlje

Često postoji potreba da se petlja izvrši tačno određeni broj puta. Uobičajeno je da se za ovu namenu koristiti promenljiva petlje, koja se često zove counter (brojač). Promenljiva petlje je celobrojna promenljiva koja je deklarisana isključivo za brojanje koliko puta je petlja izvršena. U prethodnom primeru, promenljiva count je promenljiva petlje.

Jedan prolazak kroz petlju u toku njenog izvršavanja naziva se iteracija.

Promenljivama petlje se najčešće daju jednostavna imena, kao što su `i`, `j` ili k. Za promenljive petlje najbolje je da se koriste označene celobrojne promenljive. Korišćenje neoznačenih celobrojnih promenljivih može dovesti do neočekivanih problema.

Na prvi pogled, prethodni program će izgledati potpuno ispravno. Međutim, ukoliko bi bio pokrenut, ispostaviće se da sadrži beskonačnu petlju.

```
1   #include <iostream>
2
3   int main()
4   {
5       unsigned int count = 10;
6
7       while (count >= 0)
8       {
9           if (count == 0)
10              std::cout << "Lansiranje!";
11          else
12              std::cout << count << " ";
13          --count;
14      }
15
16      return 0;
17  }
```

Izlaz iz programa počinje štampanjem "`10 9 8 7 6 5 4 3 2 1 Lansiranje!`", ali tada program "iskače iz koloseka" i počinje da štampa brojeve od 4294967295 unazad!? To se dešava zbog činjenice da uslov `count >= 0` nikada neće biti netačan (`false`)! Kada `count` dobije vrednost `0`, `0 >= 0` je tačno. U tom prolazu kroz petlju će se izvršiti iskaz `--count` i vrednost `count` će tada biti 4294967295. A pošto je `4294967295 >> 0`, program se nastavlja. Ovo se dešava zato što je promenljiva `count` deklarisana kao `unsigned int` i kao takva nikada ne može imati negativnu vrednost. To je razlog zašto će se petlja izvršavati do beskonačnosti.

Ugneždene petlje

Moguće je ugnezditi petlje unutar drugih petlji. U sledećem primeru unutrašnja petlja i spoljašnja petlja imaju svoje brojače. U ovom primeru, izraz čija se istinitost proverava u unutrašnjoj petlji koristi i promenljivu spoljašnje petlje!

```
#include <iostream>

int main()
{
   int outer = 1;
   while (outer <= 5)
   {
      int inner = 1;
      while (inner <= outer)
         std::cout << inner++ << " ";

      std::cout << "\n";
      ++outer;
   }

   return 0;
}
```

Izlaz iz ovog programa će biti:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

Petlja `do` – `while`

Jedna interesantna stvar u vezi petlje `while` je da ako je vrednost izraza petlje na početku netačna, petlja se uopšte neće izvršiti. Ponekad postoji potreba da se petlja izvrši bar jednom (na primer, kada na ekranu treba da se prikaže meni sa opcijama). U tom slučaju se može koristiti petlja `do-while`:

```
#include <iostream>
int main()
{
    int selection;
    do
    {
        std::cout << "Izaberite jednu od opcija: \n";
        std::cout << "1) Sabiranje\n" << "2) Oduzimanje\n;
        std::cout << "3) Mnozenje\n" << "4) Deljenje\n";
        std::cin >> selection;
    }
    while (selection != 1 && selection != 2 &&
           selection != 3 && selection != 4);

    // Uraditi nešto sa izabranom opcijom (npr. iskaz switch)

    std::cout << "Izabrali ste opciju #" << selection << "\n";
}
```

Petlja for

Najčešće korišćena petlja u jeziku C++ je for petlja. Ova petlja se koristi kada se zna koliko puta treba nešto da se ponovi. Ona omogućava lako definisanje, inicijalizaciju i promenu vrednosti promenljive petlje nakon svake iteracije.

Iskaz za for petlju izgleda prilično jednostavno:

```
for (init-statement; condition-expression; end-expression)
{
   Statement(s);
}
```

Promenljive definisane unutar petlje imaju posebnu vrstu opsega važenja koji se naziva i opseg petlje. Promenljive definisane unutar opsega petlje postoje samo unutar petlje i nisu dostupne van nje.

Iskaz for se sastoji iz tri dela:

1) Deo za inicijalizaciju. Najčešće se ovaj deo sastoji od definicije promenljive petlje i njene inicijalizacije. Ovaj deo for iskaza se izvršava samo jednom, prilikom prvog ulaska u petlju.

2) Uslov izvršavanja petlje. Petlja se izvršava dokle god je ovaj uslov ispunjen. Ako je njegova vrednost netačna, petlja se odmah prekida.

3) Pošto se petlja izvrši, treći deo iskaza for će se takođe izvršiti. Najčešće se ovaj izraz koristi za povećanje ili smanjenje vrednosti promenljivih deklarisanih u prvom delu for iskaza – delu za inicijalizaciju. Kada se izvrši ovaj poslednji iskaz, petlja se vraća na drugu iteraciju.

Sledi primer for petlje.

```
1   for (int count=0; count < 10; ++count)
2       std::cout << count << " ";
```

Prvo je deklarisana promenljiva petlje pod imenom count, i dodeljena joj je početna vrednost 0.

Zatim se izračunava vrednost uslova petlje, tj. ispituje se da li je vrednost brojača count < 10, a pošto je to za sada broj 0, izraz 0 < 10 je tačan. Shodno tome, iskaz unutar tela for petlje će se izvršavati. U ovom slučaju to je samo štampanje vrednosti promenljive petlje, odnosno inicijalna vrednost 0.

Na kraju prolaza kroz petlju izvršava se treći, poslednji, deo `for` iskaza. Vrednost promenljive petlje se uvećava za jedan (`++count`). Ovim je završena prva iteracija petlje, i petlja zatim prelazi na drugi korak.

Prvi deo iskaza `for` se više ne izvršava, jer kao što je rečeno, on se izvršava samo prvi put kada se uđe u petlju. Izračunava se vrednost uslova koji se nalazi u drugom delu, u ovom slučaju `1 < 10`, što je tačno, pa se petlja ponavlja. Iskaz u petlji štampa vrednost `count(1)`. Na kraju druge iteracije, promenljiva petlje, `count`, se ponovo uvećava za 1, dobija vrednost 2, itd.

Petlja će se tako izvršavati sve dok vrednost uslova petlje ne bude netačna (`false`), što bi u ovom slučaju bilo za vrednost `count = 10`. Tada bi vrednost iskaza `10 < 10` bila netačna, i petlja se više ne bi izvršavala.

Prema tome, ova petlja će štampati:

```
0 1 2 3 4 5 6 7 8 9
```

Sledi primer upotrebe `for` petlje.

```
1   // vraća vrednost nBase ^ nExp
2   int pow(int base, int exponent)
3   {
4      int total = 1;
5
6      for (int count=0; count < exponent; ++count)
7         total *= base;
8
9      return total;
10  }
```

Ova funkcija vraća vrednost $base^{exponent}$.

Promenljiva petlje uzima vrednosti od `0` do vrednosti `exponent - 1`.

Ako je eksponent `0`, `for` petlja se izvršava `0` puta, a funkcija će vratiti `1`.

Ako je eksponent 1, `for` petlja se izvršava 1 put, a funkcija će vratiti `1 * base`.

Ako je eksponent 2, `for` petlja se izvršava 2 puta, a funkcija će vratiti `1 * base * base`.

Najčešće se promenljiva petlje uvećava za 1 u svakom koraku petlje. Međutim, to ne mora biti pravilo. Može se uvećavati ili umanjivati za proizvoljnu celobrojnu vrednost.

```
1   for (int count = 9; count >= 0; count -= 2)
2      cout << count << " ";
```

Iako je uobičajeno za petlje da imaju samo jednu promenljivu petlje, ponekad petlje treba da rade sa više promenljivih petlje. Kada se ovo desi, programer može koristiti operator zarez, kako bi dodelio vrednost drugoj promenljivoj u delu za inicijalizaciju ili promenio vrednost u trećem delu `for` iskaza:

```
1   for (int i=0, j=9; i < 10; ++i, --j)
2      cout << i << " " << j << endl;
```

Kao i druge vrste petlji, `for` petlje se mogu ugnezditi unutar drugih petlji. U sledećem primeru su date dve `for` petlje gde se jedna nalazi unutar druge.

```
1   #include <iostream>
2
3   int main()
4   {
5      int counter = 0;
6      for (int i = 0; i < 10; i++) // spoljašnja petlja
7      {
8         for (int j = 0; j < 10; j++) // unutrašnja petlja
9         {
10           std::cout << ++counter << '\t';
11        }
12
13        std::cout << '\n';
14     }
15
16     return 0;
17  }
```

Ovaj program će štampati sve cele brojeve iz prvih deset dekada u deset odvojenih linija. Razmak između dva susedna broja odgovaraće veličini tabulatora. U programu se prvo ulazi u spoljašnju petlju, čija je promenljiva petlje `i`. Zatim se ulazi u drugu petlju, u kojoj je promenljiva petlje `j`. Unutrašnja petlja će se izvršavati ukupno 10 puta, za vrednosti `j` od `0` do 9, i štampaće se vrednost promenljive `counter`, koja se svaki put uvećava za 1 a zatim štampa. Prvi put kada se završi unutrašnja petlja, program ide na liniju 13, gde se samo prelazi u novi red za štampu, a zatim se promenljiva spoljašnje petlje `i` uvećava za 1. Sledi novi korak spoljašnje petlje, unutar koga se unutrašnja petlja ponovno izvršava 10 puta, jer sada promenljiva unutrašnje petlje `j` ponovo menja vrednosti od `0` do 9. S obzirom da je promenljiva `counter` inicijalizovana van spoljašnje petlje, ona će pri svakom prolazu kroz unutrašnju petlju uvećavati svoju vrednost za jedan, tako da će u drugom prolazu spoljašnje petlje i ponovnom prolasku kroz unutrašnju petlju imati vrednosti 11, 12, 13, itd. Izlaz iz ovog programa će biti:

```
1       2       3       4       5       6       7       8       9       10
11      12      13      14      15      16      17      18      19      20
21      22      23      24      25      26      27      28      29      30
31      32      33      34      35      36      37      38      39      40
41      42      43      44      45      46      47      48      49      50
51      52      53      54      55      56      57      58      59      60
61      62      63      64      65      66      67      68      69      70
71      72      73      74      75      76      77      78      79      80
81      82      83      84      85      86      87      88      89      90
91      92      93      94      95      96      97      98      99      100
```

Naredbe `break` and `continue`

Iskaz `break` se osim u `switch` iskazima koristi i za prekidanje petlji. U kontekstu petlje, iskaz `break` se može koristiti da bi se petlja prekinula pre nego što uslov petlje ne bude zadovoljen. Sledi primer prekida `for` petlje.

```
1   #include <iostream>
2
3   int main()
4   {
5      int sum = 0;
6
7      for (int count=0; count < 10; ++count)
8      {
9         std::cout << "Enter a number to add, or 0 to exit: ";
10        int num;
11        std::cin >> num;
12
13        // petlja se prekida ako korisnik unese 0
14        if (num == 0)
15           break;
16        sum += num;
17     }
18
19     std::cout << "Suma svih unetih brojeva je " << sum << "\n";
20
21     return 0;
22  }
```

Ovaj program omogućava korisniku da upisuje do 10 brojeva i na kraju prikazuje zbir svih unetih brojeva. Ako korisnik unese 0, iskaz `break` uzrokuje da se petlja ranije završi (pre unošenja svih 10 brojeva).

Takođe, iskaz `break` se može koristiti za izlazak iz beskonačne petlje:

```
#include <iostream>

int main()
{
   while (true) // beskonačna petlja
   {
      std::cout << "Unesite 0 za izlazak ili bilo sta za nastavak";
      int num;
      std::cin >> num;

      // petlja se prekida ako korisnik unese 0
      if (num == 0)
         break;
   }
   std::cout << "Izlazak iz petlje!\n";

   return 0;
}
```

Iskaz `continue` omogućava da program preskoči deo koda koji se nalazi ispod njega i kraja tela petlje, i da pređe na sledeću iteraciju petlje. Ovo je korisno kada treba ranije da se prekine trenutna iteracija petlje. U nastavku se može videti primer upotrebe gde se štampaju brojevi od 0 do 19, a pritom preskaču brojevi deljivi sa 4.

```
for (int count=0; count  < 20; ++count)
{
    // Ako je count deljiv sa 4, preskače se tekuća iteracija
    if ((count % 4) == 0)
        continue; // Skok na kraj tela petlje

    // Ako count nije deljiv sa 4, nastavlja se izvršavanje petlje
    cout << count << endl;

    // Iskaz continue skače ovde
}
```

Slično prethodnom, iskaz `continue` se može koristiti i u `while` i `do-while` petljama. Međutim, u ovom slučaju treba oprezno koristiti iskaz `break`. Kod ovakvih petlji, promenljive petlje se povećavaju unutar tela petlje, pa se upotrebom iskaza `continue` vrlo lako može dogoditi da petlja postane beskonačna!

```
int count(0);
while (count < 10)
{
   if (count == 5)
      continue; // Skok na kraj tela petlje
   cout << count << " ";
   count++;
   // Iskaz continue skače ovde
```

```
9   }
10
```

Cilj ovog programa je da štampa svaki broj između 0 i 9 osim 5. Ali, zapravo, odštampa sledeće:

```
0 1 2 3 4
```

Program zatim ulazi u beskonačnu petlju. Kada promenljiva count dobije vrednost 5, iskaz if je tačan, i program usled iskaza continue skače na kraj petlje! U sledećem, i svim ostalim koracima će situacija biti ista, tako da će program ostati do beskonačnosti u ovoj petlji. Sledi primer gde je iskaz continue pravilno upotrebljen.

```
1   int count(0);
2   do
3   {
4       if (count == 5)
5           continue; // Skok na kraj tela petlje
6       cout << count << " ";
7        // Iskaz continue skače ovde
8   } while (++count<10);
```

Ovakav program bi štampao željeni izlaz:

```
0 1 2 3 4 6 7 8 9
```

7. Nizovi, pokazivači i reference

Nizovi

U poglavlju 5 su objašnjene strukture kao složeni (agregatni) tipa podataka. Strukture obezbeđuju način smeštanja podataka različitog tipa unutar jednog agregatnog tipa. Ovo je veoma korisno u slučaju kada je potrebno modelirati jedan objekat koji ima više različitih osobina. Međutim, to nije dovoljno u slučaju kada je potrebno čuvati veliki broj podataka jednog tipa.

Strukture nisu jedini agregatni tip podataka u jeziku C++. Niz takođe predstavlja agregatni tip podataka koji omogućava pristup velikom broju promenljivih istog tipa preko zajedničkog identifikatora.

Primer problema koji se rešava: potrebno je zapisati rezultate testova za 30 učenika iz jednog razreda. Bez nizova, trebalo bi definisati 30 gotovo identičnih promenljivih!

```
1   // 30 celobrojnih promenljivih sa različitim imenom
2   int testScoreStudent1;
3   int testScoreStudent2;
4   int testScoreStudent3;
5   // ...
6   int testScoreStudent30;
```

Nizovi omogućavaju da se ovo uradi na mnogo jednostavniji način. Sledeća definicija niza je praktično ekvivalentna prethodnom:

```
1   int testScore[30]; // 30 celobrojnih promenljivih u nizu
```

U deklaraciji promenljive se koriste uglaste zagrade ([ ]) kako bi se kompajleru saopštilo da je neka promenljiva niz, kao i za koliko promenljivih treba alocirati memorijski prostor. Taj podatak se naziva dužina niza.

U prethodnom primeru, definisan je niz nazvan `testScore` koji sadrži 30 članova. Fiksni niz, ili niz fiksne veličine, je niz čija je dužina poznata u vreme kompajliranja. Kada se niz `testScore` definiše, kompajler alocira memoriju za 30 celobrojnih promenljivih.

Elementi nizova i indeksiranje

Svaka promenljiva u nizu se zove element niza. Elementi nemaju svoja jedinstvena imena. Umesto toga, za pristup pojedinačnim elementima nizova, koristi se ime

niza zajedno sa indeksnim operatorom [] i indeksom koji govori kompajleru kom elementu niza se pristupa. Ovaj proces se zove indeksiranje niza.

U prethodno navedenom primeru, prvi element u nizu je `testScore[0]`, drugi je `testScore[1]`, deseti je `testScore[9]`. Poslednji element u nizu `testScore` je `testScore[29]`. Ovo je veoma korisno, jer više nije potrebno voditi računa o velikom broju različitih, ali povezanih imena – dovoljno je samo promeniti indeks kako bi se pristupilo različitim elementima.

U jeziku C++, indeksi nizova uvek počinju od `0`. Za niz dužine `N`, elementi niza su numerisani (indeksirani) brojevima od `0` do `N-1`.

Primer programa sa nizom

Sledi primer programa u kome postoji definicija niza i indeksiranje niza.

```
1   #include <iostream>
2
3   int main()
4   {
5      int primeNumbers[5];
6
7      primeNumbers[0] = 2;
8      primeNumbers[1] = 3;
9      primeNumbers[2] = 5;
10     primeNumbers[3] = 7;
11     primeNumbers[4] = 11;
12
13     std::cout << "Najmanji prost broj je: " << prime[0] << "\n";
14     std::cout << "Suma prvih 5 prostih brojeva je: " <<
15               primeNumbers[0] + primeNumbers[1] +
16               primeNumbers[2] + primeNumbers[3] +
17               primeNumbers[4] << "\n";
18     return 0;
19  }
20
```

Izlaz iz ovog programa će biti:

```
Najmanji prost broj je: 2
Suma prvih 5 prostih brojeva je: 28
```

Nizovi i tipovi podataka

Nizovi se mogu napraviti za bilo koji tip podataka. U sledećem primeru definisan je niz promenljivih tipa `double`.

```
#include <iostream>

int main()
{
   double array[3];

   array[0] = 2.0;
   array[1] = 3.0;
   array[2] = 4.3;

   std::cout << "Aritmeticka sredina je "
             << (array[0]+array[1]+array[2])/3
             << "\n";
   return 0;
}
```

Izlaz iz ovog programa će biti:

```
Aritmeticka sredina je 3.1
```

Nizovi se mogu definisati i za strukturni tip podataka.

```
struct Rectangle
{
   int length;
   int width;
};
Rectangle rects[5];
```

Kako bi se pristupilo članu elementa niza koji predstavlja strukturu, prvo se izabere željeni element niza, a zatim se pomoću operatora za izbor člana strukture može pristupiti svakom članu strukture.

```
 rects[2].length = 24;
```

Indeksi niza

U jeziku C++, indeksi nizova moraju uvek biti celobrojnog tipa. Uobičajeno je da to budu promenljive tipa `int` (`short`, `int`, `long`, `long long`). Ovi indeksi mogu biti konstantna ili nekonstantna vrednost.

Sledi nekoliko primera:

```
int array[5];

// Korišćenje literala kao indeksa:
array[1] = 7; // ok

// Korišćenje konstante iz nabrojivog tipa kao indeksa:
enum Animals
{
    ANIMAL_CAT = 2
};
array[ANIMAL_CAT] = 4;

// Korišćenje promenljive kao indeksa:
short index = 3;
array[index] = 7;
```

Deklaracija niza fiksne dimenzije

Kada se deklariše niz fiksne dimenzije, dužina niza (broj između uglastih zagrada) mora biti konstanta definisana u vreme kompajliranja! Dakle, dužina fiksnog niza mora biti poznata u vreme kompajliranja. Sledi nekoliko različitih načina deklarisanja fiksnih nizova:

```
// Korišćenje literala za definisanje veličine niza
int array[5];

// Korišćenje simboličke konstante za definisanje veličine niza
#define ARRAY_LENGTH 5
int array[ARRAY_LENGTH];

// Korišćenje konstante za definisanje veličine niza
const int arrayLength = 5;
int array[arrayLength];

// Korišćenje nabrojivog tipa za definisanje veličine niza
enum ArrayElements
{
    MAX_ARRAY_LENGTH = 5
};
int array[MAX_ARRAY_LENGTH];

// Pokušaj sa nekonstantnom promenljivom
int length;           // Ne sme se koristiti. Mogu se koristiti
std::cin >> length;   // isključivo promenljive čija je vrednost
int array[length];    // poznata u vreme kompajliranja

// Korišćenje konstante definisane u vreme izvršavanja
int temp = 5;
const int length = temp; // Vrednost length nije poznata u vreme
                         // kompajliranja!
int array[length];       // Ne sme se koristiti
```

U poslednja dva slučaja kompajler bi prijavio grešku jer dužina nije konstanta u vreme kompajliranja. Neki kompajleri čak mogu dozvoliti ovakvo deklarisanje nizova, ali ovakve deklaracije nisu validne prema C++ standardu, i ne bi ih trebalo koristiti u C++ programima.

Pošto fiksni nizovi raspolažu memorijom alociranom u vreme kompajliranja, to uvodi dva ograničenja:

1. Fiksni nizovi ne mogu imati dužinu zasnovanu na korisničkom unosu ili nekoj drugoj vrednosti izračunatoj u vreme izvršavanja programa.
2. Fiksni nizovi imaju fiksnu dužinu koja se ne može promeniti.

U mnogim slučajevima ova ograničenja su problematična. Jezik C++ podržava i drugu vrstu nizova poznatijih kao dinamički nizovi. Dužina dinamičkog niza može se definisati u vreme izvršavanja. Dinamički nizovi su malo komplikovaniji za definisanje. Oni će biti obrađeni nešto kasnije, kada tema budu pokazivači (pointeri).

Inicijalizacija nizova

Elementi nizova se tretiraju baš kao obične promenljive, i kao takvi, oni se inicijalizuju kada se kreiraju.

Jedan od načina za inicijalizaciju nizova je da se to učini redom, element po element:

```
1    int prime[5];
2    prime[0] = 2;
3    prime[1] = 3;
4    prime[2] = 5;
5    prime[3] = 7;
6    prime[4] = 11;
```

Sledeći način je inicijalizacija pomoću liste inicijalizatora:

```
1    int prime[5] = { 2, 3, 5, 7, 11 };
```

Ako u listi ima više inicijalizatora nego što niz može prihvatiti, kompajler će prijaviti grešku. Međutim, ako u listi ima manje inicijalizatora nego što niz može prihvatiti, preostali elementi (članovi) se inicijalizuju na vrednost `0`.

```
#include <iostream>

int main()
{
    int array[5] = { 7, 4, 5 }; // inicijalizacija prva tri clana

    std::cout << array[0] << '\t';
    std::cout << array[1] << '\t';
    std::cout << array[2] << '\t';
    std::cout << array[3] << '\t';
    std::cout << array[4] << '\n';

    return 0;
}
```

Izlaz iz ovog programa će biti:

```
7       4       5       0       0
```

Prema tome, inicijalizacija svih elementa niza na vrednost `0` može se obaviti pomoću para vitičastih zagrada:

```
// Inicijalizacija svih elemenata vrednošću 0
int prime[5] = { };
```

Ukoliko se niz fiksne dužine inicijalizuje pomoću liste inicijalizatora, kompajler može odrediti dužinu niza i moguće je izostaviti eksplicitno deklarisanje dužine niza.

```
int array[5] = { 0, 1, 2, 3, 4 };// eksplicitno definisana dužina
int array[] = { 0, 1, 2, 3, 4 }; // implicitno definisana dužina
```

Ovo omogućava da se u budućnosti dužina niza ne mora ažurirati, ukoliko se doda ili izbriše neki element niza.

Prosleđivanje nizova funkcijama

Iako prosleđivanje niza funkciji na prvi pogled izgleda kao i prosleđivanje obične promenljive, jezik C++ nizove tretira drugačije.

Kada se obična promenljiva prenese po vrednosti, C++ kopira vrednost argumenta u parametar funkcije. Pošto je parametar kopija, promena vrednosti parametra ne menja vrednost argumenta.

Međutim, prosleđivanje niza funkciji kao argumenta funkcioniše drugačije. Jezik C++ ne kopira niz kada se on prenese nekoj funkciji. Umesto toga, prenosi se stvarni niz. Ovo ima sporedni efekat kojim je funkciji omogućeno da direktno menja vrednost elemenata niza!

Sledeći primer ilustruje ovaj koncept.

```
#include <iostream>

void passValue(int v)
{
   v = 99;
}
void passArray(int prime[5])
{
   prime[0] = 11;
   prime[1] = 7;
   prime[2] = 5;
   prime[3] = 3;
   prime[4] = 2;
}
int main()
{
   int value = 1;
   std::cout << "pre passValue: " << value << "\n";

   passValue(value);

   std::cout << "posle passValue: " << value << "\n";

   int prime[5] = { 2, 3, 5, 7, 11 };
   std::cout << "pre passArray: "
             << prime[0] << " "
             << prime[1] << " "
             << prime[2] << " "
             << prime[3] << " "
             << prime[4] << "\n";

   passArray(prime);

   std::cout << "posle passArray: "
             << prime[0] << " "
             << prime[1] << " "
             << prime[2] << " "
             << prime[3] << " "
             << prime[4] << "\n";
   return 0;
}
```

Izlaz iz ovog programa će biti:

```
pre passValue: 1
posle passValue: 1
pre passArray: 2 3 5 7 11
posle passArray: 11 7 5 3 2
```

U prethodnom primeru, promenljiva `value` se ne menja u `main()` funkciji, jer je vrednost parametra u funkciji `passValue(int v)` bila kopija promenljive `value` definisane u funkciji `main()`, a ne stvarna promenljiva `value`. Međutim, ovo nije slučaj i sa nizovima, jer funkcija može direktno menjati niz koji joj je prosleđen.

Ovo za sada treba samo imati u vidu, a zašto je to tako biće objašnjeno kasnije, kada budu obrađeni pokazivači u jeziku C++.

Ako je potrebno onemogućiti promenu elemenata nizova u funkciji koji je niz prosleđen, potrebno je takve parametre funkcije označiti kao konstantne pomoću ključne reči `const`:

```
1  void passArray(const int prime[5])
2  {
3     prime[0] = 11;
4     prime[1] = 7;
5     prime[2] = 5;
6     prime[3] = 3;
7     prime[4] = 2;
8  }
```

Operator `sizeof` i nizovi

Operator `sizeof` može da se koristi nad nizovima i vratiće ukupnu veličinu niza (dužina niza pomnožena veličinom elementa). Međutim, ako se niz prosledi funkciji, pa se unutar funkcije primeni operator `sizeof` na tako prenesen niz, rezultat će biti drugačiji. Ovo takođe ima veze sa pokazivačima i takođe će biti objašnjeno kada budu obrađeni pokazivači.

```
1  void printSize(int array[])
2  {
3     std::cout << sizeof(array) << '\n';
4  }
5
6  int main()
7  {
8     int array[] = { 1, 1, 2, 3, 5, 8, 13, 21 };
9     std::cout << sizeof(array) << '\n';
10    printSize(array);
12
13    return 0;
14 }
```

Izlaz iz ovog programa će biti (mada može zavisiti od kompajlera):

```
32
4
```

Veoma je bitno zapamtiti su indeksi elemenata niza dužine N brojevi od 0 do N-1! Dakle, pokušaj da se pristupi elementu niza sa indeksom koji je van tog opsega prouzrokovaće neočekivano ponašanje programa. Program će pokušati da upiše vrednost u memorijski blok koji se nalazi na mestu gde bi se nalazio element niza u slučaju da je veličina niza pravilno alocirana. To može izazvati različite efekte. Na primer, time može biti "pregažena" vrednost neke druge promenljive u programu ili će se program jednostavno srušiti. Ovo se događa zato što jezik C++ ne proverava da li je indeks niza u opsegu od 0 do veličine niza.

Nizovi i petlje

Recimo da je potrebno pronaći prosečnu ocenu na testu koji je radila grupa od pet studenata. To se može uraditi korišćenjem individualnih promenljivih:

```
1  int numStudents = 5;
2  int score0 = 84;
3  int score1 = 92;
4  int score2 = 76;
5  int score3 = 81;
6  int score4 = 56;
7  int totalScore = score0 + score1 + score2 + score3 + score4;
8  int averageScore = totalScore / numStudents;
```

Ovakav pristup nije efikasan jer stvara potrebu za velikim brojem promenljivih i obimno pisanje koda. Što je veći broj studenata, uvećava se i broj neophodnih promenljivih.

Pored toga, ako se doda novi student, mora se definisati i inicijalizovati nova promenljiva, koja zatim mora biti dodata i u izraz za izračunavanje promenljive totalScore.

Korišćenjem nizova moguće je napraviti elegantnije rešenje:

```
1  const int numStudents = 5;
2  int scores[numStudents] = { 84, 92, 76, 81, 56 };
3  int totalScore = scores[0] + scores[1] + scores[2] +
4                   scores[3] + scores[4];
5
   int averageScore = totalScore / numStudents;
```

Ovim je smanjen broj promenljivih koje se moraju deklarisati, ali promenljiva totalScore još uvek zahteva da svaki element niza bude naveden pojedinačno. I kao što je prethodno navedeno, promena broja studenata znači da se formula za izračunavanje svaki put mora ručno prilagođavati.

Kao što je već napomenuto, članu nekog niza se može pristupiti direktno preko indeksa (1,2,…,5,…10,…). To ne mora biti broj već može biti i neka promenljiva (može se koristiti promenljiva petlje za pristupanje članovima niza). Ovo se veoma često koristi gde god se koriste nizovi. U nastavku sledi prethodni primer rešen pomoću niza i petlje.

U programu koji sledi definisana je promenljiva `totalScore` pre petlje i inicijalizovana na vrednost 0, kako bi se unutar petlje na vrednost te promenljive iterativno dodavala vrednost ocena svih studenata. Ocene su smeštene u niz čija je dužina pet. U petlji se korišćenjem promenljive petlje (`student`) pristupa svakom pojedinačnom članu niza i njegova vrednost dodaje promenljivoj `totalScore`. Po izlasku iz petlje dobijena suma se deli sa brojem studenata. Na taj način se dobija vrednost prosečne ocene.

```
1   #include <iostream>
2
3   int main()
4   {
5      int scores[] = { 84, 92, 76, 81, 56 };
6      const int numStudents = sizeof(scores) / sizeof(scores[0]);
7
8      int totalScore = 0;
9      for (int student = 0; student < numStudents; ++student)
10        totalScore += scores[student];
11
12     int averageScore = totalScore / numStudents;
13
14     std::cout << "Prosecna ocena je " << averageScore;
15
16     return 0;
17  }
```

Petlje se često koriste zajedno sa nizovima. Sledi nekoliko primera gde se zajedno koriste petlje i nizovi:

- Izračunavanje neke vrednosti (npr. prosečna vrednost, ukupna vrednost),
- Traženje neke vrednosti (npr. najveća vrednost, najmanja vrednost),
- Reorganizovanje niza (npr. sortiranje u rastućem ili opadajućem redosledu elemenata niza).

Kada se izračunava vrednost, neka promenljiva se obično koristi za čuvanje međurezultata koji se koristi za izračunavanje konačne vrednosti. U prethodnom primeru gde izračunavamo prosečnu ocenu, promenljiva `totalScore` čuva zbirnu ocenu svih studenata.

Prilikom traženja vrednosti, neka promenljiva se obično koristi da čuva najbolju vrednost pronađenu do tekućeg koraka petlje (ili indeks najboljeg člana niza).

Sortiranje niza je malo komplikovanije, jer uključuje ugneždene petlje.

Višedimenzionalni nizovi

Pored nizova, u jeziku C++ je moguće definisati i višedimenzionalne nizove. Sledi primer deklaracije dvodimenzionalnog niza (odnosno matrice):

```
1   int array[3][5];
```

Kod dvodimenzionalnih nizova za prvi indeks se često kaže da je indeks vrste, a drugi indeks kolone. Konceptualno, prethodno definisan dvodimenzionalni niz izgleda ovako:

```
[0][0]  [0][1]  [0][2]  [0][3]  [0][4] // nulta vrsta
[1][0]  [1][1]  [1][2]  [1][3]  [1][4] // prva vrsta
[2][0]  [2][1]  [2][2]  [2][3]  [2][4] // druga vrsta
```

Za pristupanje elementima dvodimenzionalnog niza potrebna su dva indeksa: indeks vrste i indeks kolone.

```
1       array[2][3] = 7;
```

Inicijalizacija dvodimenzionalnih nizova

Inicijalizaciju dvodimenzionalnog niza najlakše je obaviti korišćenjem ugneždenih vitičastih zagrada, gde svaki set vrednosti predstavlja jednu vrstu.

```
1   int array[3][5] = {
2                         { 1, 2, 3, 4, 5 },     // nulta vrsta
3                         { 6, 7, 8, 9, 10 },    // prva vrsta
4                         { 11, 12, 13, 14, 15 } // druga vrsta
5                       };
```

Iako će neki kompajleri dozvoliti izostavljanje unutrašnjih zagrada, preporuka je da se uvek koriste, kako zbog čitljivosti koda, tako i zbog načina na koji C++ inicijalizuje nedostajuće članove vrednošću `0`.

```
1   int m[3][5] = {
2                     { 1, 2  },          // vrsta 0 =  1,  2,  0,  0, 0
3                     { 6, 7, 8 },        // vrsta 1 =  6,  7,  8,  0, 0
4                     { 11, 12, 13, 14 }  // vrsta 2 = 11, 12, 13, 14, 0
5                 };
```

Kao i obični jednodimenzionalni nizovi, članovi višedimenzionalnih nizova se mogu veoma jednostavno inicijalizovati na vrednost `0`:

```
int array[3][5] = {};
```

Pristupanje svim elementima dvodimenzionalnog niza zahteva dve petlje: jedna za vrstu i jedna za kolonu. Uobičajeno je da se indeks vrsta koristi kao spoljna petlja.

```
for(int row = 0; row < numRows; ++row)    // Vrste
   for(int col = 0; col < numCols; ++col) // Elementi u vrsti
        std::cout << array[row][col];
```

Višedimenzionalni nizovi mogu imati više od dve dimenzije. Sledi deklaracija trodimenzionalnog niza:

```
int array[5][4][3];
```

Trodimenzionalne nizove je teško inicijalizovati na intuitivan način pomoću inicijalizacionih lista, tako da ih je bolje inicijalizovati niz na vrednost 0, a zatim eksplicitno dodeliti vrednosti korišćenjem ugneždenih petlji. Pristup članovima trodimenzionalnog niza je potpuno analogan pristupu članovima dvodimenzionalnog niza:

```
std::cout << array[3][1][2];
```

Primer sa dvodimenzionalnim nizom

Sledi praktični primer sa dvodimenzionalnim nizom.

```
#include <iostream>

int main()
{
    const int numRows = 10;
    const int numCols = 10;
    int product[numRows][numCols] = { 0 };

    for (int row = 0; row < numRows; ++row)
        for (int col = 0; col < numCols; ++col)
            product[row][col] = row * col;

    for (int row = 1; row < numRows; ++row)
    {
        for (int col = 1; col < numCols; ++col)
            std::cout << product[row][col] << "\t";

        std::cout << '\n';
    }

    return 0;
}
```

Ovaj program izračunava i štampa tablicu množenja za sve vrednosti između 1 i 9. Prilikom štampanja tabele, petlje počinju od 1 a ne od 0. Ovo znači da se izostavlja

štampanje nulte kolone i nulte vrste, zato što bi to bila samo serija nula. Izlaz iz programa će biti:

```
1       2       3       4       5       6       7       8       9
2       4       6       8       10      12      14      16      18
3       6       9       12      15      18      21      24      27
4       8       12      16      20      24      28      32      36
5       10      15      20      25      30      35      40      45
6       12      18      24      30      36      42      48      54
7       14      21      28      35      42      49      56      63
8       16      24      32      40      48      56      64      72
9       18      27      36      45      54      63      72      81
```

Pokazivači

Kao što je ranije napomenuto, promenljiva u programu predstavlja naziv memorijskog bloka koji čuva neku vrednost. Kada program instancira promenljivu, slobodna memorijska adresa se automatski dodeljuje promenljivoj, a svaka vrednost dodeljena promenljivoj čuva se na ovoj memorijskoj adresi.

Sledi primer deklaracije jednostavne celobrojne promenljive.

```
1   int x;
```

Kada CPU izvrši ovaj iskaz, deo RAM memorije će biti izdvojen za promenljivu x. Promenljivoj x biće dodeljena određena memorijska lokacija koja poseduje vlastitu adresu. Kad god program vidi promenljivu x u izrazu ili iskazu, njenu vrednost pronalazi i preuzima preko njene adrese.

Veoma korisna stvar kod promenljivih je to što programer ne mora da vodi računa o tome koja specifična memorijska adresa im je dodeljena. Kada je potrebna neka promenljiva, vrednost promenljive se dobija preko njenog imena, a kompajler prevodi ovo ime na odgovarajuću memorijsku adresu.

Adresni operator ili operator “adresa od” (&)

Adresni operator (&) omogućava preuzimanje memorijske adrese koja je dodeljena određenoj promenljivoj, i prilično je jednostavan za korišćenje.

```
1   #include <iostream>
2
3   int main()
4   {
5      int x = 5;
6      std::cout << x <<'\n'; // štampa vrednost promenljive x
7      std::cout << &x <<'\n';// štampa memorijsku adresu promenljive x
```

```
8
9       return 0;
10  }
```

Izlaz iz ovakvog programa će zavisiti od računara, ali će izgledati nalik ovome što sledi.

```
5
0027FEA0
```

Operator dereferenciranja (*)

Dobijanje adrese promenljive nije mnogo korisno samo po sebi. Međutim, operator dereferenciranja (*) omogućava pristup vrednosti smeštenoj na određenoj memorijskoj adresi.

```
1   #include <iostream>
2
3   int main()
4   {
5       int x = 5;
6
7       std::cout << x << '\n';
8       std::cout << &x << '\n';
9       std::cout << *&x << '\n';
10
11      return 0;
12  }
```

U liniji 7 je odštampana vrednost promenljive x. U liniji 8 je odštampana adresa na kojoj se nalazi promenljiva x. U liniji 9 je odštampana vrednost koja se nalazi na memorijskoj adresi preuzetoj od promenljive x. Izlaz iz ovakvog programa će takođe zavisiti od računara, ali će izgledati nalik ovome što sledi.

```
5
0027FEA0
5
```

Pokazivači

Sada, kada su definisani adresni operator i operator dereferenciranja, mogu se definisati i pokazivači. Pokazivač je promenljiva u koju se smešta memorijska adresa **kao vrednost**. Pokazivači se obično smatraju jednim od najkonfuznijih elemenata jezika C++, ali su suštinski prilično jednostavni.

Pokazivači se deklarišu isto kao i normalne promenljive, sa dodatkom zvezdice (*) između tipa podataka i imena promenljive.

```
1   int *iPtr;
```

```
double *dPtr;
int* iPtr2;
int * iPtr3;
int *iPtr4, *iPtr5;
```

Gledano sa stanovišta sintakse, kompajleri jezika C++ će prihvatiti zvezdicu pored tipa podataka, pored imena promenljive, ili čak i na sredini. Ova zvezdica ne označava dereferenciranje! Ona je deo sintakse deklaracije pokazivača.

Međutim, prilikom deklarisanja više pokazivačkih promenljivih, zvezdica mora biti ispred svake promenljive. Zbog tog razloga, sintaksno je ispravnije stavljati zvezdicu ispred pokazivačke promenljive. U tom slučaju neće doći do previda koji je opisan u sledećem primeru.

```
int* iPtr6, iPtr7;
```

U prethodnom primeru, promenljiva `iPtr6` je pokazivač, ali promenljiva `iPtr7` nije! `iPtr7` je obična celobrojna promenljiva.

Međutim, kada je povratna vrednost funkcije pokazivač, jasnije je ako se zvezdica stavi pored povratnog tipa:

```
int* doSomething;
```

Ovo jasno pokazuje da funkcija vraća vrednost tipa `int*` a ne `int`.

Kao i kod običnih promenljivih, pokazivači nisu inicijalizovani kada su deklarisani. Ako se ne inicijalizuju vrednostima, oni će sadržavati proizvoljnu memorijsku adresu (koja nije alocirana!).

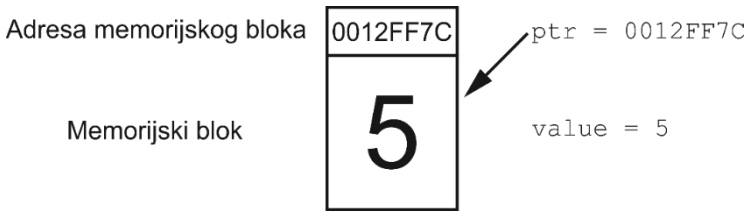
Dodeljivanje vrednosti pokazivaču

Pošto pokazivači čuvaju samo adrese, vrednost koja se dodeljuje pokazivaču mora biti adresa. Jedna od najčešćih stvari koje se rade sa pokazivačima jeste da oni čuvaju adresu druge promenljive.

Kao što je već pominjano, za dobijanje adrese promenljive koristi se adresni operator (operator “adresa od”):

```
int value = 5;
int *ptr = &value;
```

Konceptualno, to bi izgledalo kao na slici Slika 3.



Slika 3: Memorijski blok, promenljiva smeštena u njemu i pokazivač koji pokazuje na promenljivu (odnosno, na memorijski blok)

Sada je jasniji i način kako su pokazivači dobili ime. Pokazivač sadrži adresu bloka u kome je promenljiva smeštena, tako da se može reći da pokazivač "pokazuje" na vrednost promenljive sadržane u nekom memorijskom bloku.

Ovo se jednostavno može videti i iz koda primera koji sledi.

```
#include <iostream>

int main()
{
   int value = 5;
   int *ptr = &value;

   std::cout << &value << '\n';
   std::cout << ptr << '\n';

   return 0;
}
```

Izlaz iz programa će biti ista memorijska adresa, kao na primer:

```
0012FF7C
0012FF7C
```

Tip pokazivača mora odgovarati tipu promenljive na koju pokazuje.

```
int nValue = 5;
double dValue = 7.0;

int *nPtr = &nValue;      // Ispravno
double *dPtr = &dValue;   // Ispravno
nPtr = &dValue;           // Pogrešno
dPtr = &nValue;           // Pogrešno
```

Dereferenciranje pokazivača

Kada postoji pokazivačka promenljiva koja pokazuje na nešto, može se koristiti operator dereferenciranja kako bi se dobila vrednost promenljive na koju pokazuje. Dereferencirani pokazivač pokazuje na sadržaj memorijskog bloka čiju adresu čuva.

```
int value = 5;
std::cout << &value;// Štampa adresu promenljive value
std::cout << value; // Štampa vrednost promenljive value

int *ptr = &value;//ptr je pokazivač na promenljivu value
std::cout << ptr; //Štampa adresu koju čuva promenljiva ptr
std::cout << *ptr;//Štampa vrednost na koju pokazuje promenljiva ptr
```

Izlaz iz programa će biti nalik ovome što sledi:

```
0012FF7C
5
0012FF7C
5
```

Ovo je razlog zašto pokazivači moraju da imaju tip. Bez tipa, pokazivač ne bi znao kako da tumači sadržaj na koji pokazuje kada se dereferencira. Osim toga, tip pokazivača mora odgovarati adresi promenljive na koju pokazivač pokazuje. Ako adresa ne odgovara tipu pokazivača, kada se pokazivač dereferencira, pogrešno bi bili interpretirani bitovi koji se nalaze na toj adresi.

Pokazivači mogu menjati vrednost, pa mogu pokazivati na različite promenljive:

```
int value1 = 5;
int value2 = 7;

int *ptr;

ptr = &value1;     // ptr pokazuje na promenljivu value1
std::cout << *ptr; // Štampa broj 5

ptr = &value2;     // ptr sada pokazuje na promenljivu value2
std::cout << *ptr; // Štampa broj 7
```

Kada je adresa neke promenljive `value` dodeljena pokazivačkoj promenljivoj `ptr`, sledeće dve stvari su tačne:

- `ptr` je isto što i `&value`
- `*ptr` je isto što i `value`

Pošto se `*ptr` tretira isto kao promenljiva `value`, promenljivoj `value` se može dodeliti vrednost preko pokazivača! Sledeći program štampa broj 7.

```
int value = 5;
int *ptr = &value; // ptr pokazuje na promenljivu value

*ptr = 7; // *ptr je isto što i promenljiva value
std::cout << value; // Štampa broj 7
```

Upozorenje o dereferenciranju nevažećih pokazivača

Pokazivači u jeziku C++ su suštinski nebezbedni, a nepravilna upotreba pokazivača je jedan od najčešćih razloga za rušenje programa.

Kada se pokazivač dereferencira, program pokušava da dođe do memorijske lokacije koja je sačuvana u pokazivaču i da preuzme sadržaj memorije. Ako program pokuša da pristupi memorijskoj lokaciji koju mu nije dodelio operativni sistem, operativni sistem može zatvoriti aplikaciju.

Sledeći program to ilustruje i verovatno će se srušiti kada se pokrene.

```
1  #include <iostream>
2
3  int main()
4  {
5      int *p;         //Neinicijalizovani pokazivač
6      std::cout << *p;//Dereferenciranje neinicijalizovanog pokazivača
7      return 0;
8  }
```

Veličina pokazivača

Veličina pokazivača zavisi od platforme na kojoj se program izvršava. 32-bitni izvršni program koristi 32-bitne memorijske adrese, pa je, prema tome, pokazivač na 32-bitnoj mašini 32 bita (odnosno 4 bajta). Kod 64-bitnih programa, veličina pokazivača je 64 bita (8 bajtova). Ovo važi bez obzira na to na šta pokazivač pokazuje.

```
1  char *chPtr;   // char tip zauzima 1 bajt
2  int *nPtr;     // int tip uobičajeno zauzima 4 bajta
3  struct Something
4  {
5      int nX, nY, nZ;
6  };
7  Something *somethingPtr; // Tip Something zauzima 12 bajtova
8
9  std::cout << sizeof(chPtr) << '\n';         // Štampa 4
10 std::cout << sizeof(nPtr) << '\n';          // Štampa 4
11 std::cout << sizeof(somethingPtr) << '\n';  // Štampa 4
```

Kao što se može videti, veličina pokazivača je uvek ista. To je zato što je pokazivač samo memorijska adresa, a broj bitova potreban za pristup memorijskoj adresi na datoj mašini je uvek konstantan.

Gde su pokazivači korisni?

U ovom momentu pokazivači možda izgledaju beskorisno. Zašto koristiti pokazivač kad je jednostavnije koristiti originalnu promenljivu? Međutim, pokazivači su korisni u različitim slučajevima:

- Nizovi su implementirani pomoću pokazivača. Pokazivači se mogu koristiti za kretanje kroz niz (kao alternativa indeksiranju nizova).
- Jedino uz pomoć pokazivača je moguće dinamički alocirati memoriju u jeziku C++. Ovo je daleko najčešći slučaj korišćenja pokazivača.
- Mogu se koristiti za prenošenje velike količine podataka u funkciju na način koji ne uključuje kopiranje podataka (kopiranje može biti neefikasno).
- Pokazivači se mogu koristiti za prenošenje funkcije kao parametra drugoj funkciji.
- Mogu se koristiti za omogućavanje polimorfizma kada se radi o nasleđivanju.
- Mogu se koristiti unutar struktura kako bi se napravile složene strukture podataka, kao što su povezane liste i stabla.

Dakle, pokazivači su izuzetno korisni u jeziku C++. U ovom momentu mogu izgledati veoma konfuzno, ali će kroz primere u narednim poglavljima biti predstavljena pokazivačka aritmetika, što će značajno olakšati razumevanje pokazivača kao i potrebe za njihovim korišćenjem u jeziku C++.

Null pokazivači

Kao i obične promenljive, pokazivači nisu inicijalizovani kada se instanciraju. Ukoliko im nije dodeljena neka vrednost, pokazivači će podrazumevano pokazivati na neku adresu u memoriji.

Osim memorijskih adresa, pokazivač može imati i vrednost nula. Nulta vrednost znači da pokazivač ne pokazuje ni na šta. Pokazivač koji ima nultu vrednost naziva se nulti pokazivač.

Pokazivaču se može dodeliti nulta vrednost inicijalizacijom ili dodelom vrednosti `0`:

```
1   int *ptr(0);   // ptr je null pokazivač
2
3   int *ptr2;     // ptr2 je neinicijalizovani pokazivač
4   ptr2 = 0;      // ptr2 je sada null pokazivač
```

Nulti pokazivači se mogu koristiti u `if` uslovima kako bi se testiralo da li je pokazivač nulti pokazivač ili ne:

```
double *ptr(0);

if (ptr)
    cout << " ptr je pokazivač na promenljivu tipa double.";
else
    cout << "ptr je null pokazivač.";
```

Dereferenciranje nultog pokazivača takođe rezultira nedefinisanim ponašanjem programa.

Pokazivači i nizovi

Nizovi su usko povezani sa pokazivačima.

```
int array[5] = { 9, 7, 5, 3, 1 };
```

Ovo je niz od pet celih brojeva, ali za kompajler, promenljiva array je promenljiva tipa `int[5]`. Članovi niza su vrednosti: `array[0]`, `array[1]`, `array[2]`, `array[3]`, `array[4]` (sa vrednostima 9, 7, 5, 3 i 1 respektivno). Pitanje koje se nameće je: koju vrednost ima samo `array`?

Promenljiva `array` sadrži adresu početnog elementa niza, kao da je pokazivač! Ovo se može videti pokretanjem sledećeg programa.

```
#include <iostream>

int main()
{
   int array[5] = { 9, 7, 5, 3, 1 };

   std::cout << "Adresa niza je: " << array << '\n';

   std::cout << "Adresa nultog elementa je: " << &array[0] << '\n';

   return 0;
}
```

Izlaz iz ovog programa će zavisiti od računara, ali će izgledati nalik ovome:

```
Adresa niza je: 0042FD5C
Adresa nultog elementa je: 0042FD5C
```

Adresa koju čuva promenljiva `array` je adresa početnog elementa niza.

Jedna česta a principijelno pogrešna tvrdnja da je ime niza isto što i pokazivač na niz. Međutim, to nije u potpunosti tačno. Iako oba pokazuju na početni element u nizu, oni čuvaju različite tipove informacija. U gornjem slučaju, `array` je tipa `int[5]`, dok je pokazivač na niz tipa `int*`.

Konfuzija je prvenstveno uzrokovana činjenicom da prilikom korišćenja imena fiksnog niza, implicitno se pretvara u pokazivač na početni element niza. Svim elementima niza se i dalje može pristupiti preko pokazivača, ali se informacije koje se dobijaju iz promenljive tipa niza (`int[5]`) ne mogu dobiti iz pokazivača (kao što je, na primer, dužina niza pomoću operatora `sizeof`).

Međutim, ova osobina imena niza omogućava da u većini slučajeva bude tretiran kao pokazivač na niz. Na primer, može se dereferencirati niz kako bi se dobila vrednost početnog elementa. Takođe, može se deklarisati pokazivač koji pokazuje na niz, ili na određeni element niza.

```
#include <iostream>
int main()
{
   int array[5] = { 9, 7, 5, 3, 1 };
   std::cout << *array;   // Štampa vrednost 9

   int *ptr = array;
   std::cout << *ptr;     // Štampa vrednost 9

   int *ptr2 = &array[0];
   std::cout << *ptr2;    // Štampa vrednost 9

   return 0;
}
```

Razlike između pokazivača i fiksnih nizova

Primarna razlika se javlja kada koristite operator `sizeof`. Kada se koristi nad imenom fiksnog niza, `sizeof` operator vraća veličinu celog niza (dužina niza * veličina elementa). Kada se koristi nad pokazivačem, `sizeof` operator vraća veličinu memorijske adrese (u bajtovima). Sledeći program ilustruje prethodno.

```
#include <iostream>

int main()
{
   double array[5] = { 9.0, 7.0, 5.0, 3.0, 1.0 };

   std::cout << sizeof(array) << '\n';

   double *ptr = array;
   std::cout << sizeof(ptr) << '\n';

   std::cout << sizeof(*ptr) << '\n';

   return 0;
}
```

Program štampa sledeće:

```
40
4
8
```

U prvom redu je odštampana veličina niza. Ime fiksnog niza je promenljiva koja "zna" kolika je dužina niza.

U drugom redu je odštampana veličina pokazivača, tj. veličina memorijske adrese.

U trećem redu je odštampana veličina podatka koji se nalazi na adresi koju čuva pokazivač `ptr`.

Dinamičko alociranje memorije

Jezik C++ podržava tri osnovna tipa alociranja memorije.

Statičko alociranje memorije se dešava prilikom definisanja statičkih i globalnih promenljivih. Memorija za ove vrste promenljivih se alocira samo jednom, kada se program pokreće i takve promenljive traju tokom čitavog života programa (od momenta pokretanja do momenta prekida rada programa).

Automatsko alociranje memorije se dešava prilikom definisanja parametara funkcije i lokalnih promenljivih. Memorija za ove tipove promenljivih se alocira kada program uđe u odgovarajući blok, a dealocira se (oslobađa) kada program napusti blok.

Treći način alociranja memorije je dinamičko alociranje.

Statičko i automatsko alociranje memorije imaju dve zajedničke osobine:

- Veličina promenljive ili niza mora biti poznata u vreme kompajliranja.
- Alociranje i dealociranje memorije se dešava automatski (odnosno, kada je promenljiva definisana i uništena).

Ovo je u većini slučajeva u redu. Međutim, postoje situacije kada neko od ovih ograničenja (ili oba) uzrokuju probleme, najčešće kada postoji spoljašnji uticaj na program ili funkciju (npr. ulazni parametar čiju vrednost definiše korisnik).

Na primer, potrebno je u nizu karaktera smestiti nečije ime, ali se ne zna unapred koliko je ime dugačko. Zatim, može se javiti potreba za čitanjem nekog zapisa sa diska, ali se ne zna unapred kolika je veličina tog zapisa. Takođe, u razvoju kompjuterskih igara treba obezbediti mogućnost promenljivog broja aktivnih

entiteta u igri (npr. automobila nekoj Race video igri, ili raznih entiteta u nekoj strateškoj igri).

Ako je neophodno da se deklariše veličina svih promenljivih u trenutku kompajliranja, najbolje što se može uraditi je da se pretpostave maksimalne veličine promenljivih koje će biti potrebne:

```
1    char name[25];
2    Record record[500];
3    Car cars[40];
4    Polygon rendering[30000];
```

Ovo je loše rešenje iz najmanje četiri razloga:

1. Gubi se memorija ako se promenljive ne koriste. Na primer, ako se alocira 25 znakova za svako ime, a imena u proseku imaju 12 karaktera, tada će polovina alocirane memorije biti prazna ali rezervisana tako da je program ne može koristiti u druge svrhe.

2. Neki podaci vremenom prestanu da budu aktivni i trebalo bi ih izbrisati, tj. oslobodi memoriju. U primeru automobila u Race igri, tokom igre se može dogoditi da neki automobil izleti sa staze i nema ga više u trci. Ako su prethodno svi automobili bili smešteni u nekom nizu fiksne dimenzije, automobil koji više nije aktivan u igri će ostati u memoriji (i bespotrebno je zauzimati), iako više nije potreban.

3. Većina normalnih promenljivih (uključujući i nizove fiksne dimenzije) se alocira u delu memorije koja se zove stek (eng. **Stack**). Količina stek memorije rezervisana za program je obično sasvim mala - Visual Studio podrazumeva veličinu steka od 1MB za konzolne aplikacije. Ako se ova veličina prekorači, operativni sistem će verovatno zatvoriti program (stack overflow).

4. Možda i najvažnija činjenica je da može doći do veštačkih ograničenja dužine nizova. Na primer, šta će se dogoditi kada korisnik pokuša da pročita 600 zapisa sa diska, a u programu je predviđena memorija za maksimalno 500 zapisa? Korisnik mora biti obavešten o tome, tako što će program pročitati samo 500 zapisa ili, u najgorem slučaju kada se ovakvi nedostaci programa ignorišu, da program jednostavno nastavi da piše van granica alociranog prostora, što vodi ka nedefinisanom ponašanju programa.

Ovakvi problemi se lako rešavaju korišćenjem koncepta dinamičkog alociranja memorije. Dinamičkim alociranjem memorije je omogućeno da programi zahtevaju deo memorije od operativnog sistema u momentu kada je to potrebno.

Ova memorija ne dolazi iz ograničene memorije programa (stek memorije), već se ona izdvaja iz mnogo većeg memorijskog prostora kojom upravlja operativni sistem. Ta memorija se naziva hip (eng. **Heap**). Na savremenim mašinama, veličina hip memorije može se meriti gigabajtima.

Dinamičko alociranje pojedinačnih promenljivih

Za dinamičko alociranje memorije upotrebljava se operator `new`:

```
1   new int; // Dinamičko alociranje jedne celobrojne promenljive
```

U prethodnom primeru, od operativnog sistema se zahteva da alocira memoriju za jednu celobrojnu vrednost. Operator `new` vraća pokazivač koji sadrži adresu memorije koja je alocirana.

Povratna vrednost operatora se najčešće čuva u pokazivačkoj promenljivoj deklarisanoj u programu. Time je obezbeđen pristup alociranoj memoriji.

```
1   int *ptr = new int;
```

Sada se dereferenciranjem pokazivača može pristupiti alociranoj memoriji:

```
1   *ptr = 7;// Zapisivanje vrednosti 7 u dinamički alociranoj memoriji
```

Kako funkcioniše dinamičko alociranje memorije?

Računar ima radnu memoriju koja je dostupna za aplikacije koje će je koristiti. Kada se aplikacija pokrene, operativni sistem računara učitava aplikaciju u deo te memorije. Memorija koju koristi aplikacija podeljena je na različite oblasti, od kojih svaka ima drugačiju namenu. Jedna oblast sadrži kod. Druga oblast se koristi za normalne operacije (praćenje funkcija koje su pozvane, stvaranje i uništavanje globalnih i lokalnih promenljivih, itd.). Međutim, najveći deo memorije koja je na raspolaganju je slobodna i može biti dodeljena programima koji to zahtevaju.

Kada se memorija dinamički alocira, od operativnog sistema se zahteva da rezerviše deo te memorije za korišćenje u programu. Ako je ovaj zahtev moguće ispuniti, operativni sistem će aplikaciji vratiti adresu memorije. Aplikacija zatim može koristiti ovu memoriju na način kako je programer definisao. Kada aplikacija završi posao za koji joj je bila potrebna memorija, memorija se može vratiti operativnom sistemu kako bi je dodelio nekom drugom programu.

Za razliku od statičkog ili automatskog alociranja memorije, program (odnosno, programer) je odgovoran za dinamičko alociranje i dealociranje memorije.

Inicijalizacija dinamički alocirane promenljive

Dinamički alocirana promenljiva se takođe možete inicijalizovati direktnom inicijalizacijom.

```
1   int *ptr1 = new int (5); // Direktna inicijalizacija
```

Brisanje pojedinačnih promenljivih

Kad prestane potreba za dinamički alociranom promenljivom, potrebno je eksplicitno osloboditi memoriju kako bi se mogla ponovo koristiti. Za ovo se koristi operator `delete`.

```
1   int *ptr = new int (5);
2
3   // Uraditi nešto sa dinamički alociranom memorijom
4
5   delete ptr; // Vraćanje alocirane memorije operativnom sistemu
6   ptr = 0;    // Dodeljivanje vrednosti 0 pokazivaču ptr
```

Operator `delete` zapravo ne briše ništa. On jednostavno vraća operativnom sistemu prethodno alociranu memoriju. Operativni sistem tada može ponovo dodeliti tu memoriju drugoj aplikaciji (ili istoj aplikaciji).

Iako sintaksno izgleda kao da se promenljiva briše, to se zapravo ne događa. Pokazivač će i dalje imati isti opseg kao i ranije, i može mu se dodeliti nova vrednost baš kao i svakom drugom pokazivaču.

Bitno je napomenuti da korišćenje operatora `delete` nad pokazivačem koji ne pokazuje na dinamički alociranu memoriju može uzrokovati nedefinisano ponašanje programa.

Divlji pokazivači (eng. “wild“ ili “dangling“ pointers)

Jezik C++ ne garantuje šta će se desiti sa sadržajem dealocirane memorije ili vrednošću pokazivača koji se briše. U većini slučajeva, memorija koja se vraća operativnom sistemu će sadržati iste vrednosti kao i pre nego što je vraćena, a pokazivač će pokazivati na trenutno dealociranu memoriju.

Pokazivač koji pokazuje na dealociranu (ili nealociranu) memoriju naziva se “divlji” pokazivač. Dereferenciranje ili brisanje divljeg pokazivača prouzrokovaće nedefinisano ponašanje programa.

```
#include <iostream>

int main()
{
   int *ptr = new int;// Dinamičko alociranje promenljive
   *ptr = 7;          // Upisivanje vrednosti 7 u memoriji

   delete ptr;        // Vraćanje memorije operativnom sistemu
                      // ptr je sada divlji pokazivač!

   std::cout << *ptr; // Dereferenciranje divljeg pokazivača
                      // izaziva nedefinisano ponašanje programa.
   delete ptr;        // Pokušaj ponovnog oslobađanja memorije
                      // takođe izaziva nedefinisano ponašanje.
   return 0;
}
```

U prethodnom programu, vrednost 7, koja je prethodno dodeljena dinamički alociranoj promenljivoj, verovatno će i dalje biti u memoriji, ali je takođe moguće da se vrednost na toj memorijskoj adresi promenila. Takođe je moguće da je memorija dodeljena drugoj aplikaciji (ili je operativni sistem koristi za sebe). Pokušaj pristupa takvoj memoriji može dovesti do toga da operativni sistem zatvori program.

Dealociranje memorije može stvoriti više divljih pokazivača.

```
#include <iostream>

int main()
{
   int *ptr = new int;// Dinamičko alociranje celobrojne promenljive
   int *ptr1 = ptr;   // ptr1 pokazuje na istu memorijsku lokaciju

   delete ptr; // Vraćanje memorije operativnom sistemu.
               // ptr i ptr1 su sada divlji pokazivači.
   ptr = 0;    // ptr je sada null pokazivač.

   // ptr1 je još uvek divlji pokazivač!

   return 0;
}
```

Iz svega prethodno rečenog, izdvajaju se dve veoma korisne stvari koje bi trebalo usvojiti kao praksu u radu sa pokazivačima.

1. Izbegavati deklarisanje više pokazivača koji pokazuju na istu memorijsku lokaciju. Ako to ipak nije moguće, trebalo bi jasno naznačiti koji pokazivač "poseduje" memoriju (i odgovoran je za njeno oslobađanje) a koji pokazivač se koristi samo za pristup.

2. Kada se pokazivač oslobodi operatorom `delete`, ako taj pokazivač ne izlazi iz opsega odmah nakon toga, obavezno mu treba dodeliti vrednost `0`.

`null` pokazivači i dinamičko alociranje memorije

`null` pokazivači (pokazivači postavljeni na adresu `0`) su posebno korisni kada se koriste za dinamičko alociranje memorije. U kontekstu dinamičkog alociranja memorije, `null` pokazivač u osnovi znači: "ovom pokazivaču nije dodeljena memorija". Ovo omogućava dve korisne stvari:

1. Može se napraviti uslov za alociranje memorije:

```
1   if (!ptr)
2      ptr = new int;
```

2. Brisanje `null` pokazivača ne izaziva nikakav efekat. Prema tome, ne mora se ispitivati da li pokazivač pokazuje na nešto kako bi bilo moguće izbrisati ga, tj. dealocirati memoriju:

```
1   if (ptr)
2      delete ptr;
```

Ako pokazivač `ptr` nije `null` pokazivač, dinamički alocirana memorija će biti dealocirana. Ako je `null`, ništa se neće dogoditi.

Curenje memorije

Dinamički alocirana memorija nema opseg! Ona ostaje alocirana sve dok se eksplicitno ne dealocira ili dok se program ne završi. Međutim, za pokazivače koji pokazuju na dinamički alociranu memoriju važe ista pravila po pitanju opsega važenja kao i za obične promenljive. Ova neusklađenost može stvoriti probleme.

```
1   void doSomething()
2   {
3       int *ptr = new int;
4   }
```

Ova funkcija dinamički alocira prostor za celobrojnu promenljivu, ali nikada ne oslobađa memoriju korišćenjem operatora `delete`. Kada se funkcija završi, `ptr` će izaći iz opsega. Pošto je `ptr` jedina promenljiva koja čuva adresu dinamički alocirane celobrojne promenljive, kada je pokazivač `ptr` uništen, izgubljen je podatak o adresi dinamički alocirane memorije. Kao rezultat toga, ovaj dinamički alociran memorijski prostor za celobrojnu vrednost se ne može dealocirati. Ovaj fenomen se naziva curenje memorije. Curenje memorije se događa kada program izgubi adresu neke bitne dinamički alocirane memorije pre nego što je vrati operativnom sistemu. Kada se ovo desi, program više ne može izbrisati dinamički

dodeljenu memoriju, jer više nema informaciju o tome gde se takva memorija nalazi. Operativni sistem takođe ne može koristiti ovu memoriju, jer smatra da program još uvek koristi taj memorijski blok.

Curenje memorije “jede“ slobodnu memoriju računara dokle god je program pokrenut, čineći da je manje memorije dostupno ne samo ovom programu već i drugim trenutno pokrenutim programima. Programi sa ozbiljnim problemima curenja memorije mogu da “pojedu“ svu raspoloživu memoriju, što dovodi do toga da čitav računar radi sporo ili čak pada sistem. Tek kada se program završi, operativni sistem može očistiti i "povratiti" svu "pojedenu" (tj. iscurelu) memoriju.

Takođe, do curenja memorije može doći ako je pokazivaču koji čuva adresu dinamički alocirane memorije dodeljena druga vrednost:

```
1   int value = 5;
2   int *ptr = new int; // Alociranje memorije
3   ptr = &value;       // Adresa alocirane memorije je izgubljena!
```

Ovo se može izbeći brisanjem pokazivača pre ponovnog dodeljivanja:

```
1   int value = 5;
2   int *ptr = new int; // Alociranje memorije
3   delete ptr;         // Vraćanje memorije operativnom sistemu
4   ptr = &value;       // Dodela druge adrese pokazivaču
```

Takođe, dvostrukim alociranjem dolazi do curenja memorije:

```
1   int *ptr = new int;
2   ptr = new int; // Adresa prvobitno alocirane memorije je izgubljena!
```

Adresa koja se vraća drugim alociranjem se sada čuva, a adresa prvog alociranja je izgubljena! Kao posledica toga, memoriji koja je prvo alocirana se više ne može pristupiti, odnosno ne može se ni dealocirati.

Slično prethodnom, ovo se može izbeći brisanjem pokazivača pre ponovnog alociranja.

Zaključak

Operatori `new` i `delete` omogućavaju dinamičko alociranje memorije za pojedinačne promenljive u programu.

Dinamički alocirana memorija nema opseg i ostaje alocirana dok se ne dealocira ili se program završi.

Ne smeju se dereferencirati divlji i `null` pokazivači!

`void` pokazivači

Pokazivač `void`, koji se naziva i generički pokazivač, predstavlja poseban tip pokazivača koji može pokazivati na bilo koji tip podataka! Pokazivač `void` se deklariše na isti način kao i pokazivači na bilo koji drugi tip podataka.

```
void *ptr;
```

`void` pokazivač može pokazivati na objekat bilo kog tipa.

```
int nValue;
float fValue;

struct Something
{
    int n;
    float f;
};

Something sValue;

void *ptr;
ptr = &nValue; // Ispravno
ptr = &fValue; // Ispravno
ptr = &sValue; // Ispravno
```

Međutim, pošto pokazivač `void` ne zna na koju vrstu objekata pokazuje, ne može se direktno dereferencirati! Umesto toga, pokazivač `void` se mora prvo eksplicitno prevesti u tip pokazivača na koji zaista pokazuje, pre nego što se dereferencira.

```
int value = 5;
void *voidPtr = &value;

cout << *voidPtr << endl;      // Greška: Ne može se dereferencirati
                               // void pokazivač!
int *intPtr = (int*) voidPtr; // Moguće je konvertovati pokazivač
cout << *intPtr << endl;       // i njega zatim dereferencirati.
```

Ovaj kod će štampati broj 5.

S obzirom da pokazivač `void` ne zna na šta pokazuje, odnosno da za to ne postoji mehanizam u jeziku C++, odgovornost programera je da vodi računa o tome.

U kodu koji sledi dat je primer korišćenja `void` pokazivača.

```
#include <iostream>

enum Type
{
   INT,
   FLOAT,
   CSTRING
};

void printValue(void *ptr, Type type)
{
   switch (type)
   {
   case INT:
      std::cout << *(int*)ptr << '\n'; // cast to int pointer and
dereference
      break;
   case FLOAT:
      std::cout << *(float*)ptr << '\n'; // cast to float pointer
and dereference
      break;
   case CSTRING:
      std::cout << (char*)ptr << '\n'; // cast to char pointer (no
dereference)
      // std::cout knows to treat char* as a C-style string
      // if we were to dereference the result, then we'd just print
the single char that ptr is pointing to
      break;
   }
}

int main()
{
   int nValue = 5;
   float fValue = 7.5;
   char szValue[] = "Tekst";

   printValue(&nValue, INT);
   printValue(&fValue, FLOAT);
   printValue(szValue, CSTRING);

   return 0;
}
```

Izlaz iz programa će biti:

```
5
7.5
Tekst
```

Reference

Do sada su obrađena dva osnovna tipa promenljivih (ne misli se na tip podataka već na tip promenljivih):

1. Normalne promenljive, koje čuvaju vrednosti,
2. Pokazivači, koji čuvaju adresu neke druge promenljive (ili vrednost `null`).

Pokazivači koji čuvaju adresu neke promenljive mogu dereferenciranjem promeniti vrednost promenljive na koju pokazuju.

Reference su treći osnovni tip promenljivih u jeziku C++. Referenca je tip promenljive koji se ponaša kao pseudonim nekog drugog objekta ili vrednosti.

Reference se mogu odnositi na nekonstantne vrednosti (koje se nazivaju "reference" ili "nekonstantne reference") ili na konstantne vrednosti (često se nazivaju "konstantne reference").

Nekonstantne reference

Reference se deklarišu korišćenjem simbola & (ampersand) između referentnog tipa i imena promenljive:

```
1  int value = 5;
2  int &ref = value;       // Referenca na promenljivu value
```

U ovom kontekstu, znak & ne označava "adresu", već se odnosi na "reference".

Reference kao pseudonimi

Reference generalno rade identično kao i vrednosti čija su referenca. U tom smislu referenca radi kao pseudonim za objekat na koji se referencira. Sledi primer sa korišćenjem referenci.

```
1  #include <iostream>
2  int main()
3  {
4     int value = 5;      // Celobrojna promenljiva
5     int &ref = value;   // Referenca na celobrojnu promenljivu value
6
7     value = 6;          // value sada ima vrednost 6
8     ref = 7;            // value sada ima vrednost 7
9
10    std::cout << value; // Štampa 7
11    ++ref;
12    std::cout << value; // Štampa 8
13    return 0;
14 }
```

Ovaj kod će štampati sledeći izlaz:

```
7
8
```

U prethodnom primeru, `ref` i `value` se tretiraju kao sinonimi. Upotreba adresnog operatora nad referencama vraća adresu vrednosti na koju se referenca odnosi:

```
cout << &value;
cout << &ref;
```

Reference se moraju inicijalizovati u momentu kada se kreiraju:

```
int value = 5;
int &ref = value;

int &invalidRef; // Greška, neophodna je inicijalizacija reference
```

Za razliku od pokazivača koji mogu imati `null` vrednost, `null` reference ne postoje. Nekonstantne reference mogu se inicijalizovati samo nekonstantnim promenljivama.

```
#include <iostream>

int x = 5;
int &ref1 = x; // Ispravno, x je nekonstantna promenljiva

const int y = 7;
int &ref2 = y; // Pogrešno, y je konstanta

int &ref3 = 6; // Pogrešno, broj 6 nije promenljiva
```

U prvom slučaju inicijalizacija reference je ispravna. U drugom slučaju se ne može inicijalizovati nekonstantna referenca konstantnom promenljivom. Da je to tako, referencom bi bilo moguće promeniti vrednost konstantnog objekta na koje se referenca referencira, što bi narušilo konstantnost objekta.

Reference se ne mogu promeniti tako da se referenciraju na neki drugi objekat. Jednom inicijalizovana referenca se više ne može promeniti. Ovo je ilustrovano u kodu koji sledi.

```
#include <iostream>
int value1 = 5;
int value2 = 6;

int &ref = value1; // ref je sada referenca na promenljivu value1.
ref = value2;      // Dodeljuje vrednost 6 promenljivoj value1.
```

Drugi iskaz neće učiniti ono što je na prvi pogled možda očekivano. Umesto da preusmeri referencu `ref` na promenljivu `value2`, ovaj iskaz će dodeliti vrednost promenljive `value2` promenljivoj `value1`.

Reference kao parametri funkcije

Reference se najčešće koriste kod parametara funkcije. U ovom kontekstu, parametar funkcije koji je referenca omogućava funkciji da radi direktno sa argumentom koji joj je prosleđen, odnosno ne pravi se kopija argumenta čija se vrednost dodeljuje parametru. Ovo može biti veoma korisno u smislu performansi kada se radi sa veoma velikim argumentima u smislu zauzimanja memorijskog prostora i vremena potrebnog za kopiranje.

Pošto parametar referentnog tipa radi kao pseudonim za argument, funkcija koja ima parametre referentnog tipa može da izmeni argument koji joj je prosleđen.

```
1   #include <iostream>
2
3   void changeArgument(int &ref)
4   {
5      ref = 6;
6   }
7
8   int main()
9   {
10     int n = 5;
11     std::cout << n << '\n';
12
13     changeArgument(n);
14
15     std::cout << n << '\n';
16
17     return 0;
18  }
```

Izlaz iz prethodnog programa će biti:

```
5
6
```

Kada se argument prenese funkciji, parametar funkcije `ref` predstavlja referencu na argument `n`. Ovo omogućava funkciji da promeni vrednost `n` pomoću parametra `ref`. Kao što se vidi u primeru, argument koji se prosleđuje funkciji ne mora biti referenca (promenljiva `n` je tipa `int`).

Kada se nekonstantne reference koriste kao parametri funkcije, argument koji odgovara takvom parametru mora biti nekonstantna promenljiva.

Reference kao prečice

Reference se mogu koristiti kako bi se obezbedio lakši pristup ugneždenim podacima.

```
#include <iostream>

struct Something
{
    int value1;
    float value2;
};

struct Other
{
    Something something;
    int otherValue;
};

int main()
{
   Other other;
   int &ref = other.something.value1;


   other.something.value1 = 5;
   ref = 5; // potpuno analogno prethodnoj liniji koda
   return 0;
}
```

Kod veoma složenih struktura ovo može značajno poboljšati čitljivost koda.

Reference ili pokazivači?

Referenca radi slično kao pokazivač koji je implicitno dereferenciran prilikom pristupa.

```
   int value = 5;
   int *const ptr = &value;
   int &ref = value;
```

Dereferenciran pokazivač `*ptr` i referenca `ref` se odnose na istu promenljivu `value`. Kao rezultat ovoga, sledeća dva iskaza bi imala identičan efekat:

```
   *ptr = 5;
   ref = 5;
```

Pošto se reference moraju inicijalizovati na objekte koji postoje (ne mogu biti `null`) i ne mogu se kasnije menjati, reference je, u odnosu na pokazivače, mnogo sigurnije koristiti (nema rizika od dereferenciranja `null` pokazivača ili “divljeg“

pokazivača). Međutim, reference imaju ograničenu funkcionalnost u odnosu na pokazivače.

Ako se neki zadatak može rešiti i referencom i pokazivačem, trebalo bi izabrati referencu. Pokazivači treba da se koriste samo u situacijama kada nije moguće koristiti reference (kao što je, recimo, dinamičko alociranje memorije).

Konstantne reference

Baš kao što je moguće deklarisati pokazivač na konstantnu promenljivu, moguće je deklarisati i referencu na konstantu promenljivu.

```
1    const int value = 5;
2    const int &ref = value;
```

Reference na konstantnu vrednost se nazivaju i konstantnim referencama.

Za razliku od referenci na nekonstantnu vrednost, koje se mogu inicijalizovati samo nekonstantnim promenljivama, reference na konstantnu vrednost mogu se inicijalizovati i konstantnim i nekonstantnim promenljivama.

```
1    int x = 5;
2    const int &ref1 = x; // Ispravno, x je nekonstantna promenljiva.
3    const int y = 7;
4    const int &ref2 = y; // Ispravno, y je konstanta.
```

Slično kao i kod pokazivača na konstantnu vrednost, referenca na konstantnu vrednost može se odnositi na nekonstantnu promenljivu. Kada se referenca definiše kao konstantna, vrednost na koju se referencira se takođe smatra konstantnom, čak i kada originalna promenljiva nije konstantna.

```
1    int value = 5;
2    const int &ref = value; // ref je konstantna referenca
3
4    value = 6; // Ispravno, value je nekonstantna promenljiva
5    ref = 7;   // Pogrešno, ref je konstantna referenca
```

Konstantne reference kao parametri funkcije

Reference koje se koriste kao parametri funkcije takođe mogu biti konstantne. Ovim je obezbeđeno da se funkciji prosledi argument bez njegovog kopiranja, a takođe je onemogućeno da funkcija menja vrednost takvog argumenta.

```
1   #include <iostream>
2
3   void printIt(const int &x)
4   {
5       std::cout << x;
6   }
7   int main()
8   {
9      int a = 1;
10     printIt(a);
11     const int b = 2;
12     printIt(b);
13
14     return 0;
15  }
```

8. Funkcije

Funkcije

Parametri i argumenti funkcija

U okviru poglavlja o funkcijama često će biti pominjani termini **parametar** i **argument**, pa će ovde biti ponovljena njihova definicija.

Često se događa da se termini parametar i argument zamene jedan drugim. Međutim, postoji jasna razlika između ova dva termina.

Parametar funkcije je promenljiva deklarisana u deklaraciji funkcije:

```
1  void foo(int x) // x je parametar funkcije
2  {
3  }
```

Argument je vrednost koju pozivalac funkcije prenosi funkciji:

```
1  foo(6);   // 6 je vrednost argumenta funkcije
2  foo(y+1); // y+1 je vrednost argumenta funkcije
```

Kada se pozove funkcija, svi parametri funkcije se kreiraju kao promenljive, a vrednost argumenata se kopira u parametre.

```
1  void foo(int x, int y)
2  {
3  }
4
5  foo(6, 7);
```

Kada se funkcija `foo()` pozove sa argumentima 6 i 7, kreira se parametar funkcije x i dodeljuje mu se vrednost 6, a takođe i parametar y kome se dodeljuje vrednost 7.

Iako parametri nisu deklarisani unutar bloka funkcije, parametri funkcije imaju lokalni opseg. To znači da se kreiraju kada se funkcija pozove i uništavaju kada se napusti blok funkcije.

```
1  void foo(int x, int y) // Promenljive x i y su definisane ovde
2  {
3  } // Promenljive x i y se uništavaju ovde
```

Postoje tri načina prosleđivanja argumenata funkcijama: prosleđivanje po vrednosti, prosleđivanje po referenci i prosleđivanje po adresi.

Prenošenje argumenata po vrednosti

Kada se argument prenese po vrednosti, vrednost argumenta se kopira u vrednost parametra funkcije.

```
1   void foo(int y)
2   {
3      using namespace std;
4      cout << "y = " << y << endl;
5   }
6   int main()
7   {
8      foo(5);
9      int x = 6;
10     foo(x);
11     foo(x+1);
12
13     return 0;
14  }
```

U prvom pozivu funkcije `foo()`, argument je broj 5. Kada je funkcija pozvana, definisana je promenljiva y i dodeljena joj je vrednost argumenta 5. Promenljiva y se uništava po izlasku iz funkcije `foo`.

Zatim je definisana promenljiva x i inicijalizovana na vrednost 6. Ta vrednost je u sledećem pozivu funkcije `foo()` prosleđena kao argument i izjednačena sa parametrom funkcije y. Promenljiva y se uništava kada se završi funkcija `foo()`.

U trećem pozivu funkcije `foo()`, argument je izraz `x + 1`. Izraz `x + 1` se izračunava kako bi se dobila vrednost 7, koja se kopira u promenljivu y. Promenljiva će još jednom biti uništena kada se završi funkcija `foo()`.

Dakle, prethodni program štampa:

```
y = 5
y = 6
y = 7
```

Pošto se funkciji prenosi kopija argumenta, izvorni argument se ne može izmeniti unutar funkcije.

Prednosti prosleđivanja po vrednosti su:

- Argumenti koji se prosleđuju po vrednosti mogu biti promenljive (npr. x), literali (npr. 6), izrazi (npr. `x + 1`), strukture, klase i enumeratori. Drugim rečima, skoro bilo šta.

- Funkcija nikada ne menja argumente, što sprečava neželjene efekte.

Nedostatak prosleđivanja po vrednosti:

- Kopiranje struktura i klasa može značajno uticati na performanse programa, naročito ako se funkcija poziva više puta i ako argumenti zauzimaju veliki memorijski prostor.

U većini slučajeva, kada su parametri funkcije ugrađeni tipovi podataka, prosleđivanje po vrednosti je najbolji način za prihvatanje argumenata i izjednačavanje sa parametrima. Takođe, primenjuje se kada funkcija nema potrebu za menjanjem argumenta koji joj je prosleđen.

Prenošenje argumenata po referenci

Prenošenje po vrednosti je pogodno u mnogim slučajevima ali ima nekoliko ograničenja. Prvo, prilikom prenosa velikih struktura ili klasa funkcijama, prosleđivanje po vrednosti će kopirati argument u parametar funkcije. U mnogim slučajevima, ovo je nepotrebno narušavanje performansi programa, pošto bi izvorni argument bio sasvim dovoljan. Drugo, prilikom prenošenja argumenata po vrednosti, jedini način da funkcija vrati neku vrednost je putem povratne vrednosti funkcije. Iako je ovo često dovoljno, postoje slučajevi kada bi bilo jasnije i efikasnije kada bi funkcija modifikovala argument koji joj je prenesen. Prenošenje po referenci rešava oba ova problema.

Da bi se funkciji prosledile promenljive po referenci, jednostavno treba deklarisati parametre funkcije kao reference a ne kao obične promenljive.

```
1   void addOne(int &y)
2   {
3       y = y + 1;
4   }
```

Kada se funkcija pozove, y će postati referenca za argument koji je funkciji prosleđen. Pošto se referenca promenljive tretira potpuno isto kao i sama promenljiva, sve promene napravljene na referenci zapravo menjaju argument funkcije. U primeru koji sledi može se videti efekat prosleđivanja po referenci.

```
1   #include <iostream>
2
3   void foo(int &y)
4   {
5       y = 6;
6   }
```

```
int main()
{
   int x = 5;
   cout << "x = " << x << '\n';
   foo(x);
   cout << "x = " << x << '\n';
}
```

Parametar funkcije `foo()` je referenca umesto obične promenljive. Kada se funkcija pozove, parametar y postaje referenca za x. Izlaz iz ovakvog koda bi bio:

```
x = 5
x = 6
```

Vraćanje više vrednosti pomoću referenci

Često se u programima nailazi na potrebu da funkcija vrati više vrednosti. Međutim, funkcije mogu imati samo jednu povratnu vrednost. Jedan od načina za vraćanje više vrednosti je korišćenje parametara – referenci.

U sledećem primeru funkcija `getSinCos()` prihvata jedan parametar (po vrednosti) kao ulaz, a "vraća" dva parametra (po referenci) kao izlaz.

```
#include <iostream>
#include <math.h>    // za funkcije sin() i cos()

void getSinCos(double degrees, double &sinOut, double &cosOut)
{
   // Konverzija iz stepena u radijane
   const double pi = 3.14159265358979323846;
   double radians = degrees * pi / 180.0;
   sinOut = sin(radians);
   cosOut = cos(radians);
}

int main()
{
   double sinus(0.0);
   double cosinus(0.0);

   // getSinCos će vraćati vrednosti u promenljivama sin i cos
   getSinCos(30.0, sinus, cosinus);

   std::cout << "Sinus ugla je " << sinus << '\n';
   std::cout << "Cosinus ugla je " << cosinus << '\n';
   return 0;
}
```

Parametri koji se koriste samo za vraćanje vrednosti vraćaju se pozivaocu. Tim parametrima je u imenu dodat sufiks "out" kako bi bili označeni kao izlazni parametri funkcije. Ovo je podsetnik da početna vrednost argumenta nije bitna,

da treba očekivati da funkcija promeni njihove vrednosti i da ih na taj način "vrati" pozivaocu.

Funkcija `main()` stvara lokalne promenljive `sinus` i `cosinus` i njih prosleđuje funkciji `getSinCos()` po referenci. Ovo znači da funkcija `getSinCos()` ima pristup stvarnim `sinus` i `cosinus` promenljivama, a ne samo njihovim kopijama. Shodno tome, funkcija `getSinCos()` dodeljuje nove vrednosti promenljivama `sinus` i `cosinus` preko referenci `sinOut` i `cosOut` respektivno, koje prepisuju ("pregaze") stare vrednosti promenljivih `sinus` i `cosinus`. Funkcija `main()` zatim štampa ove ažurirane vrednosti.

Da su promenljive `sinus` i `cosinus` prosleđene po vrednosti umesto po referenci, funkcija `getSinCos()` bi promenila kopije argumenata `sinus` i `cosinus`, što bi dovelo do toga da promene napravljene nad parametrima funkcije apsolutno ne utiču na vrednosti argumenata funkcije. Ali, s obzirom da su argumenti `sinus` i `cosinus` prosleđeni po referenci, svaka promena napravljena nad tim promenljivama unutar funkcije utiče i na vrednosti koje su van funkcije (vrednosti prosleđenih argumenata). Stoga, prosleđivanje po referenci se može koristiti za vraćanje više od jedne vrednosti, što nije moguće uraditi preko povratne vrednosti.

Ovaj način prenošenja argumenata je veoma funkcionalan, ali ima nekoliko manjih nedostataka. Prvo, sintaksa je malo neprirodna, s obzirom da se i ulazni i izlazni parametri nalaze u istoj listi, tj. u "potpisu" funkcije. Drugo, pozivalac funkcije mora funkciji proslediti promenljive, a ne neke konkretne vrednosti (recimo, nije moguće proslediti neki broj: 2,3,4,.., ili `2.6`, `-3.8`, itd., izuzev ako je parametar funkcije konstantna referenca), što znači da je u programu neophodno definisati promenljive za argumente funkcije, iako se možda neće ni koristiti osim u pozivu funkcije. Na kraju, iz ugla funkcije – pozivaoca nije sasvim jasno da li će pozvana funkcija menjati argumente ili ne. Ovo je potencijalno najopasniji deo ovakvog načina prenošenja argumenata funkcijama, jer može dovesti do grešaka. Zato je uobičajeno u programiranju da se imenom parametara sugeriše na to da li će biti promenjeni ili ne. U suprotnom, ukoliko nije poželjno menjati vrednosti parametara funkcije koji su ipak reference (iz nekog drugog razloga je stavljeno da su reference), tada je dobra praksa da takvi parametri budu konstante reference.

```
1  void foo(const int &x)
2  {
3     x = 6;  // Greška: x je konstantna referenca!
4  }
```

Prednosti i nedostaci prenošenja argumenata po referenci

Prednosti prenošenja po referenci su:

- Reference omogućavaju funkciji da promeni vrednost argumenta, što je ponekad korisno. U suprotnom, konstante reference se mogu koristiti kao ograničenje kako bi se osiguralo da funkcija neće promeniti argument.
- Pošto se ne pravi kopija argumenta, prosleđivanje po referenci je brzo, čak i kada se koristi sa velikim strukturama ili klasama.
- Reference se mogu koristiti za vraćanje više vrednosti iz funkcije (preko parametara).
- Reference moraju biti inicijalizovane, tako da nije potrebno brinuti o nultim početnim vrednostima.

Nedostaci prenošenja po referenci su:

- Za parametre koji su nekonstantne reference uvek moraju postojati argumenti koji su stvarne promenljive.
- Može biti teško zaključiti da li će parametar koji je nekonstantna referenca biti promenjen unutar funkcije ili ne. Kako bi se ove stvari razdvojile, poželjno je imenom promenljive nagovestiti da li će biti promenjena ili ne, a takođe i onemogućiti promenu pomoću konstantne reference, ukoliko parametar nikako ne bi trebalo menjati.
- Iz poziva funkcije nije moguće znati da li se argument u funkciji menja ili ne. Argument prosleđen po vrednosti i prosleđen po referenci izgleda isto. Samo na osnovu deklaracije funkcije se može videti da li je neki parametar stvarna vrednost ili referenca.

Kada treba koristiti prenošenje po referenci:

- Kada se funkciji prenose strukture ili klase (koristiti `const` reference kada parametar služi samo za “čitanje”).
- Kada je potrebno da funkcija promeni vrednost argumenta.

Kada ne bi trebalo koristiti prenošenje po referenci:

- Pri prenošenju ugrađenih tipova (čak i ovo ne treba u potpunosti isključiti, nekada je korisno).
- Prilikom prenošenja nizova (koristi se prenošenje po adresi).

Prenošenje argumenata po adresi

Postoji još jedan način prenošenja argumenata funkciji, a to je prenošenje po adresi. Prenošenje argumenta po adresi podrazumeva prosleđivanje adrese promenljive – argumenta, a ne same promenljive – argumenta. Pošto je argument adresa promenljive, parametar funkcije mora biti pokazivač. Funkcija tada može dereferencirati pokazivač kako bi promenila vrednost na koju pokazuje.

Sledi primer funkcije koja prihvata parametar koji je prosleđen po adresi.

```
1   #include <iostream>
2
3   void foo(int *ptr)
4   {
5       *ptr = 6;
6   }
7
8   int main()
9   {
10     int value = 5;
11
12     std::cout << "value = " << value << '\n';
13     foo(&value);
14     std::cout << "value = " << value << '\n';
15     return 0;
16  }
```

Izlaz iz ovog koda će biti:

```
value = 5
value = 6
```

Kao što se može videti, funkcija `foo()` je promenila vrednost argumenta (vrednost promenljive) preko parametra koji je pokazivač.

Prenošenje po adresi obično se koristi sa pokazivačima koji se najčešće koriste da kako bi pokazivali na nizove koji se prosleđuju kao parametri. U sledećem primeru obe funkcije `printArray1` i `printArray2` će štampati sve vrednosti u nizu:

```
#include <iostream>

void printArray1(int array[], int length)
{
   for (int index=0; index < length; ++index)
      std::cout << array[index] << ' ';
}

void printArray2(int *array, int length)
{
   for (int index=0; index < length; ++index)
      std::cout << array[index] << ' ';
}

int main()
{
   int array[6] = { 6, 5, 4, 3, 2, 1 };
   printArray1(array, 6);
   printArray2(array, 6);
}
```

Ovaj program će štampati sledeće:

```
6 5 4 3 2 1
6 5 4 3 2 1
```

Fiksni nizovi se funkcijama prenose preko pokazivača (odnosno imena niza, koje je istovremeno i pokazivač na niz), tako da se funkciji mora proslediti i dužina niza kao poseban parametar. Kao što se moglo videti iz primera, postoje dva načina prihvatanja niza kao argumenta funkcije. Prvi je da parametar funkcije bude takođe niz (`int array[]`) a drugi je da parametar funkcije bude pokazivač (`int *array`). Oba načina su ispravna, jer ime niza predstavlja istovremeno i pokazivač na niz.

Prosleđivanje konstantne adrese

S obzirom da funkcija `printArray()` ne menja svoje argumenate, oni se mogu proglasiti konstantnim.

```
void printArray(const int *array, int length)
{
   if (!array) return;

   for (int index=0; index < length; ++index)
      std::cout << array[index] << ' ';
}

int main()
{
   int array[6] = { 6, 5, 4, 3, 2, 1 };
```

```
12      printArray(array, 6);
13  }
```

Ovim je onemogućeno da funkcija `printArray()` promeni bilo koji od članova niza preko pokazivača koji joj je prosleđen.

Međutim, adrese prosleđene funkciji su zapravo prosleđene po vrednosti! Kada se nekoj funkciji prosledi adresa, vrednost te adrese se kopira u vrednost pokazivača koji je parametar funkcije. Ako se vrednost parametra funkcije promeni, promeniće se samo kopija. Shodno tome, argument, odnosno u ovom slučaju adresa originalne promenljive neće biti promenjena. Sledi primer koji ilustruje ovo.

```
#include <iostream>

void setToNull(int *tempPtr)
{
   tempPtr = 0; // izjednačavanje tempPtr sa 0
}
int main()
{
   int five = 5;
   int *ptr = &five; // ptr je pokazivač na promenljivu five

   // Štampa vrednost 5
   std::cout << *ptr << std::endl;

   // tempPtr će dobiti kopiju pokazivača ptr
   setToNull(ptr);

   // pokazivač ptr i dalje ima adresu promenljive five!

   // Štampa vrednost 5
   if (ptr)  std::cout << *ptr;
   else      std::cout << "ptr je null pokazivac";

   return 0;
}
```

U promenljivu `tempPtr`, koja je pokazivač, se kopira vrednost adrese koja je prosleđena kao argument. Iako je u funkciji promenjen pokazivač `tempPtr`, tako da pokazuje na nešto drugo (`nullptr`), to ne menja vrednost na koju pokazivač `ptr` pokazuje. Shodno tome, ovaj program će odštampati sledeće:

```
5
5
```

Iako se sama adresa prenosi po vrednosti, i dalje se može dereferencirati kako bi se promenila vrednost argumenta.

Prilikom prenošenja argumenta po adresi, promenljiva – parametar funkcije dobija kopiju adrese koja je prosleđena kao argument. U ovom trenutku parametar funkcije i argument pokazuju na istu vrednost.

Ako se parametar funkcije dereferencira, kako bi se promenila vrednost na koju pokazuje, to će uticati na vrednost na koju ukazuje prosleđeni argument, jer i parametar funkcije i argument pokazuju na istu vrednost!

Ako je parametru funkcije dodeljena druga adresa, to neće uticati na argument, jer je parametar funkcije kopija, a promena kopije neće uticati na originalnu vrednost, tj. vrednost argumenta prosleđenog funkciji. Nakon promene vrednosti adrese koju čuva parametar funkcije, parametar funkcije i argument će pokazivati na različite vrednosti, tako da dereferenciranje takvog parametra i promena vrednosti više neće uticati na vrednost na koju pokazuje argument.

Prenošenje adrese po referenci

Sledeće logično pitanje je: "Šta ako je potrebno da funkcija promeni pokazivač koji joj je prosleđen kao argument?". Ovo se može uraditi na dva načina. Prvi je iznenađujuće jednostavan. Moguće je jednostavno proslediti adresu po referenci. Sintaksa za prenošenje reference na pokazivač izgleda malo čudno. Funkciji se prosleđuje referenca na pokazivač, tako da će funkcija promenom pokazivača – parametra funkcije menjati i pokazivač – argument funkcije.

```
1   #include <iostream>
2
3   // tempPtr je referenca na pokazivač.
4   // Bilo koja promena pokazivača tempPtr će uticati na argument!
5   void setToNull(int* & tempPtr)
6   {
7      tempPtr = 0;
8   }
9
10  int main()
11  {
12     int five = 5;
13     int *ptr = &five;
14
15     // Štampa vrednost 5
16     std::cout << *ptr << std::endl;
17
18     // tempPtr je referenca na pokazivač ptr
19     setToNull(ptr);
20     // ptr je sada null pokazivač!
21
22     if (ptr) std::cout << *ptr;
23     else     std::cout << "ptr je null pokazivac";
```

```
24
25      return 0;
26  }
```

Drugi način je da parametar funkcije bude pokazivač na pokazivač, kojoj će se kao argument poslati adresa pokazivača.

```
1   #include <iostream>
2   // tempPtr je pokazivač na pokazivač (ili adresa pokazivača).
3   // Promena dereferenciranog pokazivača tempPtr će menjati argument!
4   void setToNull(int** tempPtr)
5   {
6      *tempPtr = 0;
7   }
8
9   int main()
10  {
11     int five = 5;
12     int *ptr = &five;
13
14     // Štampa vrednost 5
15     std::cout << *ptr << std::endl;
16
17     // pokazivač ptr je prosleđen po adresi
18     setToNull(&ptr);
19     // ptr je sada null pokazivač!
20
21     if (ptr) std::cout << *ptr;
22     else     std::cout << "ptr je null pokazivac";
23     return 0;
24  }
```

I u ovom slučaju sintaksa izgleda pomalo čudno. Parametar funkcije je “pokazivač na pokazivač”, a odgovarajući argument bi bio adresa nekog pokazivača. Na taj način je omogućeno da se dereferenciranjem ovakvog pokazivača (tj. pokazivača na pokazivač) dobije stvarna vrednost onoga što je prosleđeno kao argument, a to je zapravo pokazivač prosleđen po adresi! Za nekoga ko se prvi put susreće sa pokazivačima ovo će verovatno i dalje delovati zbunjujuće, ali će vremenom, tj. vežbanjem pisanja koda u jeziku C++ postati sasvim jasno, kada pokazivačka aritmetika postane rutina u pisanju koda.

Izlaz iz oba prethodna primera će biti identičan:

```
5
ptr is null
```

Prednosti prenošenja argumenata po adresi su:

- Prenošenje po adresi dozvoljava funkciji da promeni vrednost argumenta, što je ponekad korisno. Inače, ključnom rečju `const` se može onemogućiti da funkcija promeni vrednost argumenta.
- Pošto se argument ne kopira, performanse računara pri radu sa ovakvom promenljivom su značajno bolje, pogotovo kada se radi sa velikim strukturama ili klasama.
- Moguće je vratiti više vrednosti iz funkcije preko parametara funkcije.

Nedostaci prenošenja po adresi:

- Budući da literali (konkretne vrednosti, kao što su brojevi, 2, 3, 5,... `2.4`, `-5.9`, ili slova ‘a’, ‘b’,…, i slično) kao i rezultati izraza nemaju adrese, argumenti funkcija čiji su parametri pokazivači moraju biti normalne promenljive.

Sve vrednosti moraju biti proverene kako bi se utvrdilo da li su `null` pokazivači. Pokušaj dereferenciranja nultog pokazivača rezultiraće rušenjem programa.

Dakle, prenošenje po adresi i prenošenje po referenci imaju gotovo identične prednosti i mane. Teško je reći kada bi trebalo primeniti neku od ove dve tehnike jer to uglavnom zavisi od trenutnih potreba. Međutim, može se reći da je prosleđivanje po referenci metoda koja bi trebalo da ima prednost, jer je bezbednija sa stanovišta stabilnosti programa (kod rada sa pokazivačima potencijalno postoji veća mogućnost greške).

Vraćanje povratne vrednosti funkcije po vrednosti, referenci i adresi

Vraćanje povratne vrednosti funkcije po vrednosti je najjednostavniji i najsigurniji povratni tip. Kada se promenljiva vraća po vrednosti, kopija te vrednosti se vraća pozivaocu funkcije. Kao i kod prenošenja po vrednosti, mogu se vratiti literali (npr. 5), vrednost promenljive (npr. x) ili vrednost izraza (npr. x + 1), što čini vraćanje po vrednosti veoma fleksibilnim.

Još jedna prednost vraćanja po vrednosti je to što se mogu vratiti promenljive (ili izrazi) koji uključuju lokalne promenljive deklarisane unutar funkcije, bez potrebe da se vodi računa o problemima u pogledu opsega važenja. Pošto se promenljive ili izrazi izračunavaju pre povratka iz funkcije, a kopija vrednosti vraća pozivaocu, ne postoji problem sa izlaskom promenljivih iz opsega kada se funkcija završi.

```
1   int doubleValue(int x)
2   {
3      int value = x * 2;
```

```
4       return value; // Kopija promenljive value će biti vraćena
5   } // Promenljiva value izlazi iz opsega
```

Vraćanje povratne vrednosti po adresi podrazumeva vraćanje adrese neke promenljive pozivaocu. Slično kao kod prosleđivanja po adresi, vraćanjem po adresi može se vratiti samo adresa neke promenljive, a ne literal ili izraz (zato što oni nemaju svoje adrese).

Međutim, vraćanje po adresi ima jedan nedostatak koji vraćanje po vrednosti nema. Ako se vrati adresa promenljive koja se nalazi unutar funkcije (lokalne promenljive koja važi na nivou funkcije), program će imati nedefinisano ponašanje. Sledi primer ovakvog koda.

```
1   int* doubleValue(int x)
2   {
3       int value = x * 2;
4       return &value; // vraća se adresa lokalne promenljive value
5   } // lokalna promenljiva value izlazi iz opsega!
```

Kao što se može videti, promenljiva `value` se uništava po izlasku iz funkcije `doubleValue()`. Adresa takve promenljive se vraća pozivaocu. Međutim, ta adresa će po izlasku iz funkcije biti adresa u memoriji koja nije alocirana. Korišćenje takve adrese, odnosno pristupanje memoriji na koju pokazuje pokazivač, koji bi eventualno dobio ovu adresu, izazvalo bi rušenje programa.

Vraćanje po adresi se često koristi za vraćanje pokazivača koji pokazuje na dinamički alociranu memoriju. Ovo može da radi jer dinamički alocirana memorija ne izlazi iz opsega na kraju bloka u kojem je deklarisana, tako da će memorija biti alocirana i kada program napusti funkciju u kojoj se alocira. U nastavku je dat takav primer.

```
1   int* allocateArray(int size)
2   {
3       return new int[size];
4   }
5
6   int main()
7   {
8       int *array = allocateArray(25);
9
10      // Uraditi nešto sa nizom
11
12      delete[] array;
13      return 0;
14  }
```

Vraćanje povratne vrednosti po referenci je slično prosleđivanju po referenci. Vrednosti koje se vraćaju po referenci moraju biti promenljive (ne može se vratiti

referenca na literal ili izraz). Kada se promenljiva vrati po referenci, referenca promenljive se vraća pozivaocu. Pozivalac zatim može koristiti ovu referencu i da pomoću nje menja stvarnu promenljivu. Međutim, baš kao i kod vraćanja po adresi, lokalne promenljive ne treba vraćati po referenci.

```
int& doubleValue(int x)
{
   int value = x * 2;
   return value; // Vraća se referenca na lokalnu promenljivu value
} // lokalna promenljiva value izlazi iz opsega!
```

U prethodnom programu, program vraća referencu na vrednost koja će biti uništena kada program izađe iz funkcije. To bi značilo da pozivalac dobija referencu ka promenljivoj koja je uništena, odnosno referencu ka memorijskom prostoru koji je prethodno dealociran. Kompajler će verovatno dati upozorenje ili grešku ako se napiše ovakav kod. Vraćanje po referenci se obično koristi za vraćanje podatka sadržanog unutar nekog argumenata koji je funkciji prosleđen po referenci ili adresi.

```
#include <iostream>
struct Rectangle
{
   int length;
   int width;
};
// Vraća referencu na člana strukture
int& getRectangleLength(Rectangle& rectangle)
{
   return rectangle.length;
}
// Vraća referencu na člana strukture
int& getRectangleWidth(Rectangle& rectangle)
{
   return rectangle.width;
}
```

```
int main()
{
   Rectangle rec = {0, 0};

   int& l = getRectangleLength(rec);
   l = 8;
   int& w = getRectangleWidth(rec);
   w = 5;

   std::cout << rec.length << '\n';
   std::cout << rec.width << '\n';
}
```

U prethodnom primeru, funkcije `getRectangleLength()` i `getRectangleWidth()` vraćaju reference na članove strukture koja im je prethodno prosleđena takođe po referenci. Promenljive koje prihvataju vrednosti vraćene iz ovih funkcija su takođe reference. Menjanjem ovih referenci, menjaće se i podaci promenljive `rec`, jer su promenljive `l` i `w` reference na članove `length` i `width` promenljive `rec`.

Zaključak

U većini slučajeva, vraćanje po vrednosti će biti dovoljno za rešavanje nekog problema. To je istovremeno najfleksibilniji i najsigurniji način vraćanja informacija iz funkcije pozivaocu. Međutim, postoje i situacije kada je neophodno vraćanje po referenci ili adresi, naročito kada se radi sa dinamički alociranom memorijom. Kada se koristiti vraćanje po referenci ili adresi, mora se osigurati da povratne vrednosti ne predstavljaju referencu ili adresu promenljive koja je izašla iz opsega prilikom izlaska iz funkcije.

Preklapanje funkcija

Preklapanje funkcija je karakteristika jezika C++ koja omogućava da mogu postojati više funkcija sa istim imenom, sve dok imaju različitu listu parametara.

```
1  int add(int x, int y)
2  {
3      return x + y;
4  }
```

Prethodni primer predstavlja trivijalnu funkciju koja sabira dva cela broja. Međutim, ako je potrebno sabirati i dva broja sa pokretnim zarezom, ova funkcija više neće biti odgovarajuća, jer će se argumenti sa pokretnim zarezom pretvarati u celobrojne parametre, što dovodi do toga da argumenti sa pokretnim zarezom gube svoje decimalne delove.

Jedan od načina za rešavanje ovog problema je definisanje više funkcija sa neznatno različitim imenima.

```
1  int addInteger(int x, int y)
2  {
3      return x + y;
4  }
5
6  double addDouble(double x, double y)
7  {
8      return x + y;
9  }
```

Međutim, ovo zahteva definisanje nekog standarda ili konvencije za imenovanje funkcija, te nazive je neophodno pamtiti, kako bi uvek bila pozvana odgovarajuća funkcija, itd.

Preklopljene funkcije su daleko bolje rešenje. Koristeći preopterećene funkcije, možemo jednostavno deklarisati još jednu funkciju `add()` koja kao parametre ima promenljive sa pokretnim zarezom.

```
1   double add(double x, double y)
2   {
3       return x + y;
4   }
```

Sada postoje dve verzije funkcije `add()`:

```
1   int add(int x, int y);
2   double add(double x, double y);
```

Iako je možda očekivano da ovo izazove konflikt imena, to se ipak neće dogoditi. Kompajler može da odredi koju verziju funkcije `add()` treba da pozove na osnovu argumenata koji se koriste u pozivu funkcije. Ako su u pozivu prosleđena dva cela broja, jezik C++ će znati da treba da pozove verziju funkcije koja prihvata takođe dva cela broja. Ako su u pozivu prosleđena dva brojeva sa pokretnim zarezom, C++ će znati da treba da pozove verziju koja prihvata takve brojeve. Zapravo, moguće je definisati neograničen broj funkcija sa istim imenom, sve dok je lista tipova parametara svake funkcije jedinstvena.

Takođe je moguće definisati funkciju `add()` sa različitim brojem parametara.

```
1   int add(int x, int y, int z)
2   {
3       return x + y + z;
4   }
```

Tip povratne vrednosti funkcije se NE uzima u obzir prilikom preklapanja funkcija. U sledećem kodu je dat primer u kome je trebalo napisati dve funkcije koje vraćaju slučajni broj, jedna koja vraća celobrojnu vrednost i jedna koja vraća broj sa pokretnim zarezom.

```
1   int getRandomValue();
2   double getRandomValue();
```

Kompajler će ovo označiti kao grešku. Ove dve funkcije imaju istu listu parametara (odnosno, u ovom slučaju su bez parametara), pa će se druga funkcija `getRandomValue()` tretirati kao ponovna deklaracija funkcije. Prema tome, ovakvim funkcijama potrebno je dati različita imena.

Pozivanje preklopljenih funkcija

Pozivanje preklopljenih funkcija može da ima jedan od sledeća tri moguće ishoda:

1) Pronađena je funkcija čija lista parametara odgovara prosleđenim argumentima;
2) Nije pronađeno nikakvo podudaranje argumenata i parametara;
3) Poziv je dvosmislen jer se lista argumenata poklapa se sa dve ili čak više funkcija.

Kada se pozove preklopljena funkcija, jezik C++ prolazi kroz sledeći proces kako bi se utvrdilo koja će se verzija funkcije pozvati:

1) Prvo, jezik C++ pokušava da pronađe potpuno podudaranje. Ovo je slučaj kada se stvarni argumenti apsolutno podudaraju s tipovima parametra jedne od preklopljenih funkcija.

   Primer:

```
1  void print(char *value);
2  void print(int value);
3
4  print(0); // potpuno poklapanje sa verzijom print(int)
```

   Iako bi argument `0` tehnički moglo da odgovara parametru `char*` (kao `null` pokazivač), ipak postoji tačno podudaranje sa verzijom funkcije čiji je argument `int` (jer bi za podudaranje sa tipom `char*` bila neophodna implicitna konverzija). Prema tome, biće pozvana funkcija koja prihvata celobrojnu vrednost.

2) Ako nije pronađeno tačno podudaranje argumenata i parametra, jezik C++ pokušava da pronađe odgovarajuću funkciju kroz implicitnu konverziju argumenata. Određeni tipovi podataka se mogu automatski konvertovati putem interne konverzije tipa u druge tipove. Recimo:

   - `char`, `unsigned char` i `short` se konvertuju u `int`,
   - `unsigned short` može se konvertovati u `int` ili `unsigned int`, u zavisnosti od veličine `int`,
   - `float` se može konvertovati u `double`,
   - `enum` se može konvertovati u `int`.

   Primer:

```
void print(char *value);
void print(int value);

print('a'); // implicitna konverzija char u int
```

U ovom slučaju, pošto nema funkcije sa parametrom `char`, `char` vrednost 'a' se konvertuje u celobrojnu vrednost, koja zatim odgovara verziji funkcije čiji je parametar `int`.

3) Ako konverzija nije moguća, jezik C++ pokušava da pronađe odgovarajuću funkciju kroz standardnu konverziju. Standardne konverzije uključuju:

- Svaki numerički tip će se podudarati s bilo kojim drugim numeričkim tipom, uključujući i neoznačene tipove (npr. `int` će se podudarati sa `float`)
- `enum` će se podudarati sa bilo kojim numeričkim tipom (npr. `enum` će se podudariti sa `float`)
- Nula će odgovarati tipu pokazivača i numeričkom tipu (npr. `0` će se poklapati sa `char*` ali takođe i sa `int`, `float`, `dobule`,...)
- Bilo koji pokazivač će se podudarati sa `void` pokazivačem.

Primer:

```
struct Employee;
void print(float value);
void print(Employee value);

print('a'); // implicitna konverzija char - int - float
```

U ovom slučaju, pošto ne postoji funkcija sa parametrom `char`, karakter 'a' će biti konvertovan u ceo broj, kome će zatim odgovarati verzija sa parametrom tipa `float`.

4) Najzad, jezik C++ pokušava da pronađe podudaranje kroz korisnički definisane konverzije. Iako se još uvek nisu obrađene klase kao složeni tip, unutar klasa (koje su slične strukturama) mogu biti definisane konverzije ka drugim tipovima koje se mogu implicitno primeniti na objekte te klase. Na primer, moguće je definisati klasu X i unutar nje korisnički definisanu konverziju u `int`.

Primer:

```
class X; // Klasa sa korisnički definisanom konverzijom u int

```

```
3   void print(float value);
4   void print(int value);
5
6   X value;
7   print(value); // promenljiva value će biti konverotovana u int
```

Iako je promenljiva `value` tipa klase `X`, pošto ova klasa ima konverziju definisanu od strane korisnika u `int`, poziv funkcije `print(value)` će pozvati `print(int)` verziju funkcije.

Preklapanje funkcija može značajno smanjiti složenost programa uz uvođenje veoma malo dodatnih rizika. Iako na prvi pogled može izgledati komplikovano (naročito pravila podudaranja), u stvarnosti preklopljene funkcije obično rade transparentno i bez ikakvih problema. Kompajler će označiti sve dvosmislene slučajeve, a generalno ih je moguće veoma lako rešiti.

Podrazumevani parametri

Podrazumevani parametar (ili opcioni parametar) je parametar funkcije koji ima podrazumevanu vrednost. Ako korisnik ne prosledi vrednost za ovaj parametar, funkcija će takvom parametru dodeliti podrazumevanu vrednost. Ako korisnik prosledi vrednost za podrazumevani parametar, koristiće se ta vrednost umesto podrazumevane. Sledi primer koda sa ovakvom funkcijom.

```
1   void printValues(int x, int y = 10)
2   {
3       std::cout << "x: " << x << '\n';
4       std::cout << "y: " << y << '\n';
5   }
6
7   int main()
8   {
9       printValues(1); // y ima podrazumevanu vrednost 10
10      printValues(3, 4); // y ima korisnički definisanu vrednost 4
11  }
```

Izlaz iz ovog programa će biti:

```
x: 1
y: 10
x: 3
y: 4
```

U prvom pozivu funkcije, pozivalac nije prosledio argument za parametar `y`, tako da je funkcija koristila podrazumevanu vrednost `10`. U drugom pozivu, pozivalac je prosledio vrednost za `y`, tako da je korišćena vrednost koju je definisao korisnik.

Podrazumevani parametri su odlična opcija kada je funkciji potrebna neka vrednost koju korisnik može ali ne mora da zameni. Sledi primer funkcije kojoj se prosleđuje ime fajla, koji se podrazumevano zove "default.log". Ukoliko korisnik funkcije pošalje drugačije ime, funkcija će prihvatiti to ime umesto podrazumevanog.

```
void openLogFile(char filename[]="default.log")
{
   // Kod za otvaranje datoteke
}

int main()
{
   openLogFile();
   openLogFile("errors.log");
   return 0;
}
```

Funkcija može imati više podrazumevanih parametara:

```
void printValues(int x = 10, int y = 20, int z = 30)
{
   std::cout << "Vrednosti: " << x << " " << y << " " << z << '\n';
}
int main
{
   printValues(1, 2, 3);
   printValues(1, 2);
   printValues(1);
   printValues();
   return 0;
}
```

Izlaz iz prethodnog programa će biti:

```
Vrednosti: 1 2 3
Vrednosti: 1 2 30
Vrednosti: 1 20 30
Vrednosti: 10 20 30
```

Ukoliko funkcija ima više parametara od kojih neki imaju podrazumevane vrednosti a neki ne, oni parametri koji imaju podrazumevane vrednosti moraju biti poređani s desna na levo u listi parametara, tako da u listi prvo budu parametri koji nemaju podrazumevane vrednosti a zatim parametri sa podrazumevanim vrednostima.

```
void printValues(int x=10, int y); // nije dozvoljeno
void printValues(int x, int y=10); // dozvoljeno
void printValues(int y, int x=10); // dozvoljeno
```

Važno je napomenuti da se podrazumevani parametri NE računaju kao parametri koji čine funkciju jedinstvenom. Prema tome, sledeće deklaracije nisu dozvoljene:

```
1   void printValues(int x);
2   void printValues(int x, int y=20);
```

Ako bi pozivalac pozvao funkciju `printValues()` sa argumentom `10`, kompajler ne bi mogao da utvrdi da li je korisnik želeo da pozove funkciju `printValues(int)` ili funkciju sa podrazumevanom vrednošću `printValues(int, 20)`.

Pokazivači na funkcije

Pokazivač je promenljiva koja čuva adresu neke druge promenljive. Pokazivači na funkcije su slični, osim što umesto pokazivanja na promenljive, oni pokazuju na funkcije.

Funkcije su kao i promenljive smeštene u memoriji računara. Slično kao kod nizova, ime funkcije istovremeno predstavlja i pokazivač na memorijsku lokaciju gde je funkcija smeštena. To se može videti u sledećem primeru.

```
#include <iostream>

int foo()
{
    return 5;
}

int main()
{
   std::cout << foo(); // Štampa povratnu vrednost funkcije
   std::cout << foo;   // Štampa adresu funkcije

   return 0;
}
```

U prvom pozivu `std::cout` štampa povratnu vrednost funkcije, a u drugom adresu na kojoj se funkcija nalazi (neki kompajler će možda praviti problem, ali u tom slučaju je dovoljno prilikom štampanja eksplicitno konvertovati ime `foo` u tip `(void*)`).

Sintaksa za deklarisanje pokazivača na funkciju izgleda ovako:

```
1   // fcnPtr je pokazivač na funkciju koja vraća int i nema argumente
2   int (*fcnPtr)();
```

U prethodnom kodu, `fcnPtr` je pokazivač ka funkciji koja nema parametre i vraća ceo broj. Pokazivač `fcnPtr` može pokazivati na bilo koju funkciju koja odgovara ovom tipu.

Zagrade unutar kojih je `*fcnPtr` su neophodne zbog prioriteta operatora, jer `int* fcnPtr()` bi se tumačio kao deklaracija funkcije pod nazivom `fcnPtr` koja nema parametre a vraća pokazivač na ceo broj (bez obzrira na poziciju zvezdice, da li je uz `int` ili uz `fcnPtr`).

Dodeljivanje funkcije pokazivaču na funkciju

Pokazivači na funkcije mogu biti inicijalizovani funkcijom:

```
1   int f()
2   {
3      return 5;
4   }
5   int g()
6   {
7      return 6;
8   }
9   int main()
10  {
11     int (*fcnPtr)() = f; // fcnPtr je pokazivač na funkciju f
12     fcnPtr = g; // fcnPtr je pokazivač na funkciju g
13
14     return 0;
15  }
```

Sledeći kod predstavlja jednu uobičajenu grešku:

```
1   fcnPtr = g();
```

Ovaj slučaj zapravo predstavlja pokušaj da se povratna vrednost funkcije dodeli pokazivaču na funkciju, a ne ono što je bila namera, tj. da se pokazivaču na funkciju dodeli adresa funkcije `g()`.

Pozivanje funkcije pomoću pokazivača funkciju

Pokazivačem na funkciju je moguće pozvati stvarnu funkciju na koju pokazivač i pokazuje.

```
1   int foo(int x)
2   {
3       return x;
4   }
5
6   int main()
7   {
```

```
8       int (*fcnPtr)(int) = foo; // fcnPtr je pokazivač na funkciju foo
9       fcnPtr(5); // poziv funkcije foo(5) pomoću pokazivača fcnPtr
10      (*fcnPtr)(5); // poziv funkcije foo(5) pomoću eksplicitno
11                    // dereferenciranog pokazivača fcnPtr.
12
13      return 0;
14  }
```

Neki od starijih kompajlera možda neće prihvatiti ovakvu sintaksu poziva u liniji 9, koja podrazumeva implicitno dereferenciranje, već se to mora uraditi eksplicitno. U tom slučaju poziv funkcije preko pokazivača bi izgledao kao u liniji 10.

Značajno je napomenuti da podrazumevani parametri ne rade u slučaju kada se funkcija poziva preko pokazivača. Razlog za to je što se pokazivači na funkcije razrešavaju u vreme izvršavanja a podrazumevani parametri u vreme kompajliranja. Zato sve vrednosti podrazumevanih parametara moraju biti definisane prilikom pozivanja funkcije preko pokazivača na funkciju.

Prosleđivanje funkcije kao argumenata drugoj funkciji

Jedna od najkorisnijih stvari sa pokazivačima na funkcije je da se funkcija prosledi kao argument nekoj drugoj funkciji.

U nastavku je dat primer gde treba da napišemo funkciju za obavljanje određenog zadatka (kao što je sortiranje niza), ali je potrebno omogućiti da korisnik može da definiše kako će se određeni deo tog zadatka izvršiti (recimo da odluči da li će niz biti sortiran u rastućem ili opadajućem redosledu). Ovaj primer se posebno odnosi na sortiranje, ali se može generalizovati i na druge slične probleme.

Svi algoritmi sortiranja rade na sličan način: algoritam sortiranja prolazi kroz listu brojeva, upoređuje parove brojeva i sortira brojeve na osnovu rezultata tih poređenja. Promenom načina poređenja može se promeniti način na koji funkcija sortira niz bez uticaja na ostatak koda.

Sledi kod jednostavne funkcije za sortiranje.

```
1   void selectionSort(int array[], int size)
2   {
3      for (int startIndex = 0; startIndex < size; ++startIndex)
4      {
5         int smallestIndex = startIndex;
6
7         for (int i = startIndex + 1; i < size; ++i)
8         {
9            if (array[smallestIndex] > array[i])
10              smallestIndex = i;
11        }
```

```
12
13          int tmp = array[startIndex];
14          array[startIndex] = array[smallestIndex];
15          array[smallestIndex] = tmp;
16      }
17  }
```

U prethodnoj funkciji se poređenje može zameniti funkcijom za poređenje. S obzirom da će funkcija upoređivanja upoređivati dva cela broja i vraćati logičku vrednost, kako bi saopštila da li elemente treba zameniti ili ne, kod funkcije za poređenje će izgledati ovako:

```
1   bool ascending(int x, int y)
2   {
3       return x > y;
4   }
```

Sada funkcija za sortiranje umesto uslova može da koristi funkciju `ascending()` za upoređivanje.

```
1   void selectionSort(int array[], int size)
2   {
3      for (int startIndex = 0; startIndex < size; ++startIndex)
4      {
5         int smallestIndex = startIndex;
6
7         for (int i = startIndex + 1; i < size; ++i)
8         {
9            if (ascending(array[smallestIndex], array[i]))
10              smallestIndex = i;
11        }
12
13        int tmp = array[startIndex];
14        array[startIndex] = array[smallestIndex];
15        array[smallestIndex] = tmp;
16     }
17  }
```

Kako bi se korisniku dala mogućnost da izabere način sortiranja, umesto da koristi funkciju poređenja, omogućeno mu je da sam definiše funkciju i prosledi je funkciji sortiranja preko pokazivača na funkciju.

Pošto funkcija upoređivanja upoređuje dve celobrojne vrednosti a vraća `bool` vrednost, pokazivač na takvu funkciju bi izgledao ovako:

```
1   bool (*comparisonFcn)(int, int);
```

Dakle, pozivalac funkcije ima mogućnost da prosledi pokazivač na željenu funkciju poređenja kao treći parametar. Funkcija sortiranja će zatim koristiti taj pokazivač za pozivanje odgovarajuće funkcije za upoređivanje.

Sledi primer funkcije za sortiranje koja za upoređivanje koristi parametar tipa pokazivač na funkciju. Ovom pokazivaču se dodeljuje adresa korisnički definisane funkcije za upoređivanje, koja će biti prosleđena kao argument. U programu su definisane dve funkcije za upoređivanje, jedna za upoređivanje u rastućem redosledu a druga u opadajućem.

```
#include <iostream>

bool ascending(int x, int y)
{
    return x > y;
}

bool descending(int x, int y)
{
    return x < y;
}

void selectionSort(int array[], int size, bool (*compFcn)(int,int))
{
   for (int startIndex = 0; startIndex < size; ++startIndex)
   {
      int smallestIndex = startIndex;

      for (int i = startIndex + 1; i < size; ++i)
      {
         if (compFcn (array[smallestIndex], array[i]))
            smallestIndex = i;
      }

      int tmp = array[startIndex];
      array[startIndex] = array[smallestIndex];
      array[smallestIndex] = tmp;
   }
}

void printArray(int *array, int size)
{
   for (int index=0; index < size; ++index)
      std::cout << array[index] << " ";
   std::cout << '\n';
}

int main()
{
   int array[9] = { 3, 7, 9, 5, 6, 1, 8, 2, 4 };

   // Sortiranje niza u opadajućem redosledu
   selectionSort(array, 9, descending);
   printArray(array, 9);

   // Sortiranje niza u rastućem redosledu
```

```
47      selectionSort(array, 9, ascending);
48      printArray(array, 9);
49  }
```

Prethodni program će štampati:

```
9 8 7 6 5 4 3 2 1
1 2 3 4 5 6 7 8 9
```

Prethodni primer je prilično jednostavan, i deluje zbunjujuće zašto su uopšte korišćeni pokazivači na funkcije. U ovom slučaju, oni bi bili korisni ako bi bilo potrebno definisati neki komplikovaniji način sortiranja, kao što je recimo sortiranje u rastućem redosledu, pri čemu će u sortiranom nizu prvo biti složeni parni brojevi pa zatim neparni.

```
1   bool evensFirst(int x, int y)
2   {
3      // ako je x paran a y neparan, x je ispred y (ne menjaju mesto)
4      if ((x % 2 == 0) && !(y % 2 == 0))
5         return false;
6
7      // ako je x neparan a y paran, y je ispred x (menjaju mesto)
8      if (!(x % 2 == 0) && (y % 2 == 0))
9         return true;
10
11     // inače se sortira u rastućem redosledu
12     return (x > y);
13  }
14
15  int main()
16  {
17     int array[9] = { 3, 7, 9, 5, 6, 1, 8, 2, 4 };
18
19     selectionSort(array, 9, evensFirst);
20     printArray(array, 9);
21
22     return 0;
23  }
```

Izlaz iz programa će biti:

```
2 4 6 8 1 3 5 7 9
```

Kao što se može videti, korišćenjem pokazivača na funkcije, u ovom kontekstu, moguće je dozvoliti pozivaocu da "doda" svoju vlastitu funkcionalnost u kod koji je prethodno napisan i testiran, što pomaže u ponovnom korišćenju koda.

Podrazumevani pokazivači na funkcije

Ako je pozivaocu omogućeno da funkciji prosledi funkciju kao argument, veoma je korisno obezbediti neke standardne funkcije koje će se koristiti u slučaju da pozivalac ne želi da bilo šta dodaje. Recimo, u prethodnom primeru sortiranja, pisanjem funkcija `ascending()` i `descending()` zajedno s funkcijom `selectionSort()`, biće omogućeno da korisnici jednostavnije pozovu funkciju za sortiranje, jer ne moraju da pišu svoje vlastite funkcije za rastući ili opadajući redosled. Takođe, moguće je u deklaraciji funkcije `selectionSort()` definisati podrazumevanu vrednost pokazivača na funkciju, tako da će funkcija uraditi sortiranje na podrazumevani način, ukoliko joj se ne prosledi neki konkretni pokazivač na funkciju.

```
1   // Podrazumevano sortiranje u rastućem redosledu
2   void selectionSort(int *array, int size, bool (*comparisonFcn)(int,
3   int) = ascending);
```

U ovom slučaju, sve dok korisnik poziva funkciju `selectionSort()` bez pokazivača na funkciju, parametar `comparisonFcn` će imati podrazumevanu vrednost, tj. pokazivaće na funkciju za sortiranje u rastućem redosledu (odnosno na funkciju `ascending()`).

Pokazivači na funkcije korisni su prvenstveno kada je potrebno skladištiti funkcije u nizu (ili nekoj drugoj strukturi) ili prosleđivati funkciju kao argument nekoj drugoj funkciji.

Rekurzivne funkcije

Rekurzivna funkcija u jeziku C++ je funkcija koja poziva samu sebe. Ovakve funkcije treba veoma pažljivo pisati. Sledi primer loše napisane rekurzivne funkcije.

```
#include <iostream>

void countDown(int count)
{
   std::cout << count << '\n';
   countDown(count-1); // funkcija poziva samu sebe rekurzivno
}

int main()
{
   countDown(5);

   return 0;
}
```

Kada se pozove funkcija `countDown()` sa argumentom 5, štampa se broj "5" a zatim poziva takođe funkcija `countDown()` ali ovoga puta sa argumentom 4. Tada će `countDown()` štampati broj "4" i pozvati `countDown` sa argumetnom `(3)`. `countDown(3)` štampa "3" i poziva `countDown(2)`. Sekvenca `countDown(n)` poziva `countDown(n-1)` i ponavlja se do beskonačnosti, formirajući rekurzivni ekvivalent beskonačne petlje.

Svaki poziv funkcije dovodi do upisivanja podataka u stek poziva (eng. call stack). Pošto se iz funkcije `countDown()` nikada ne vraća već ona neprestano iznova poziva samu sebe, informacije sa steka poziva se nikada ne uklanjaju. Ovo će u nekom trenutku napuniti stek poziva i rezultiraće prekoračenjem memorije steka poziva i rušenjem programa.

Uslovi prekidanja rekurzivnog pozivanja funkcije

Pozivi rekurzivnih funkcija obično rade kao i normalni pozivi. Međutim, prethodno navedeni program ilustruje najvažniju razliku između običnih i rekurzivnih funkcija: neophodno je uključiti uslov prekidanja rekurzije ili će se, u suprotnom, pokrenuti "zauvek" (odnosno, dok se stek poziva ne napuni i dođe do rušenja programa). Prekidanje rekurzije je uslov koji će, kada se ispuni, rekurzivnu funkciju zaustaviti.

Prekidanje rekurzije generalno podrazumeva korišćenje iskaza `if`. Sledi prethodna funkcija sa dodatim uslovom za prekidanje rekurzije.

```
#include <iostream>

void countDown(int count)
{
   std::cout << "push " << count << '\n';

   if (count > 1) // uslov za prekid rekurzije
```

```
8       countDown(count-1);
9
10      std::cout << "pop " << count << '\n';
11  }
12
13  int main()
14  {
15      countDown(5);
16      return 0;
17  }
```

Ako se sada pokrene program u debug modu, `countDown()` će započeti sledećim izlazom:

```
push 5
push 4
push 3
push 2
push 1
```

Stek poziva će u ovom momentu izgledati ovako:

```
countDown(1)
countDown(2)
countDown(3)
countDown(4)
countDown(5)
main()
```

Zbog uslova prekidanja, prilikom poziva `countDown()` sa argumentom 1 neće biti pozvana funkcija `countDown(0)` – umesto toga, `if` iskaz se ne izvršava, tako da će se štampati "pop 1", a zatim završiti. U ovom trenutku, poziv `countDown(1)` je završen i nestaje sa steka poziva, a program se vraća na prethodno pozvanu funkciju `countDown(2)`. `countDown(2)` nastavlja izvršavanje iza mesta gde je pozvana `countDown(1)`, tako da štampa "pop 2", i zatim se završi. Pozivi rekurzivne funkcije će redom nestajati sa steka, sve dok se ne završe svi prethodni rekurzivni pozivi.

Dakle, ovaj program će odštampati:

```
push 5
push 4
push 3
push 2
push 1
pop 1
pop 2
pop 3
```

```
pop 4
pop 5
```

Treba primetiti da se štampanje "push" i "pop" teksta dešava u obrnutim redosledima, jer je štampanje "push" pre poziva rekurzivne funkcije, a štampanje "pop" posle poziva rekurzivne funkcije. Odavde se može videti da funkcija koja je poslednja pozvana, tj. poslednja je na steku, prva odlazi sa steka (eng. Last In First Out - LIFO).

Jednostavni primeri upotrebe rekurzije

Jedan trivijalan primer rekurzije može biti rekurzivno računanje sume svih brojeva od 0 do unetog broja.

```
1   int sumTo(int value)
2   {
3      if (value <= 0)
4         return 0; // uslov za prekid rekurzije
5      else if (value == 1)
6         return 1; // uslov za prekid rekurzije
7      else
8         return value + sumTo(value - 1); // rekurzivni poziv funkcije
9   }
```

Rekurzivne programe je uglavnom teško shvatiti samo gledajući ih. Često je korisno videti šta se dešava kada je rekurzivna funkcija pozvana sa određenom vrednošću. U primeru koji sledi pokazano je šta se dešava kada je funkcija za izračunavanje sume pozvana sa argumentom 5.

```
sumTo(5) je pozvana, 5<=1 je netačno, vratiće vrednost sumTo(4)+5.
sumTo(4) je pozvana, 4<=1 je netačno, vratiće vrednost sumTo(3)+4.
sumTo(3) je pozvana, 3<=1 je netačno, vratiće vrednost sumTo(2)+3.
sumTo(2) je pozvana, 2<=1 je netačno, vratiće vrednost sumTo(1)+2.
sumTo(1) je pozvana, 1<=1 je tačno, vratiće vrednost 1.
```

Prethodni uslov `1<=1` je uslov prekidanja rekurzije.

Krećući se unazad, i sabirajući redom vrednosti koje predstavljaju povratne vrednosti funkcija koje se nalaze na steku, dobiće se sledeći rezultat:

```
sumTo(1) vraća 1.
sumTo(2) vraća sumTo(1) + 2, odnosno 1 + 2 = 3.
sumTo(3) vraća sumTo(2) + 3, odnosno 3 + 3 = 6.
sumTo(4) vraća sumTo(3) + 4, odnosno 6 + 4 = 10.
sumTo(5) vraća sumTo(4) + 5, odnosno 10 + 5 = 15.
```

Sada bi trebalo da je jasnije kako se sabiraju brojevi između 1 i vrednosti prosleđene rekurzivnoj funkciji `sumTo()`.

Rekurzivne funkcije najčešće rešavaju problem tako što prvo pronađu rešenje pod-problema (rekurzivno), a zatim modifikuju dobijeno medjurešenje kako bi se došlo do rešenja. U prethodno navedenom algoritmu, `sumTo(value)` prvo rešava `sumTo(value - 1)`, a zatim dodaje vrednost promenljive `value` kako bi se pronašlo rešenje za `sumTo(value)`.

U mnogim rekurzivnim algoritmima, za neke ulaze dobijaju se trivijalni izlazi. Na primer, `sumTo(1)` ima trivijalan izlaz `1`, i nema koristi od dalje rekurzije. Ulazi za koje algoritam trivijalno računa izlaz nazivaju se osnovni slučajevi. Osnovni slučajevi funkcionišu kao uslovi prekidanja rekurzije.

Još jedan primer rekurzije je rešavanje problema izračunavanja faktorijela.

```
1  int factorial(int value)
2  {
3     int fact;
4     if (value <= 1) return 1; // uslov za prekid rekurzije
5
6     fact = value * factorial(value - 1);
7
8     return fact;
9  }
```

Rekurzivno ili iterativno rešavanje problema?

Pitanje koje se često postavlja vezano za rekurzivne funkcije je: "Zašto koristiti rekurzivnu funkciju ako se isti zadatak može rešiti iterativno (pomoću petlji `for` ili `while`)?" Odgovor je da se rekurzivni problem uvek može rešiti iterativno, ali, za netrivijalne probleme, rekurzivna verzija algoritma je često mnogo jednostavnija za pisanje (čak i čitanje!). Iterativne funkcije (one koje koriste `for` ili `while` petlje) su skoro uvek efikasnije od ekvivalentnih rekurzivnih funkcija zato što ne opterećuju stek poziva. Međutim, to ne znači da su iterativne funkcije uvek bolji izbor. Ponekad je rekurzivna implementacija funkcije toliko čistija i lakša, pa je opterećivanje steka poziva zanemarljivo, posebno ako je algoritam takav da se ne mora mnogo puta ponavljati kako bi se pronašlo rešenje.

9. Osnove objektno orijentisanog programiranja

Uvod u objektno-orijentisano programiranje

U tradicionalnom programiranju (onom koje je predstavljeno u prethodnim poglavljima), programi su u osnovi liste instrukcija na računaru kojima se definišu podaci, a zatim radi sa tim podacima (pomoću iskaza i funkcija). Podaci i funkcije koje operišu nad tim podacima su odvojeni entiteti koji se kombinuju kako bi se dobio željeni rezultat. Zbog ovakve organizacije programa, tradicionalni programi često ne omogućavaju intuitivno predstavljanje stvarnosti. Na programeru je da na odgovarajući način poveže promenljive sa funkcijama. Ovo vodi do koda koji izgleda ovako:

```
1   driveTo(you, work);
```

Objektno orijentisano programiranje je najjednostavnije opisati analogijom sa životnim okruženjem. Sve što nas okružuje predstavlja objekte: knjige, zgrade, hrana, pa čak i mi sami. Objekti imaju dve glavne komponente:

1) Spisak relevantnih svojstava (npr. masa, boja, veličina, čvrstoća, oblik, itd.)

2) Određeni broj ponašanja koje mogu iskazati (npr. otvaranje, stvaranje nečega drugog, grejanje, hlađenje, itd.). Svojstva i ponašanje su nerazdvojni.

Objektno orijentisano programiranje (OOP) pruža mogućnost stvaranja objekata koji povezuju svojstva i ponašanja u celinu koja se može koristiti više puta. Ovakav koncept vodi ka kodu koji izgleda ovako:

```
1   you.driveTo(work);
```

Ovakav način programiranja ne samo da je jasniji, već je jasnije i ko je subjekat (određena osoba) i kakvo ponašanje se poziva (vožnja negde). Umesto fokusiranja na pisanje funkcija, pažnju treba usmeriti na definisanje objekata koji imaju dobro definisan skup ponašanja. To je razlog zbog koga se ovakva programerska paradigma naziva "objektno orijentisano programiranje".

Ovo omogućava da programi budu napisani na modularan način, što ih čini lakšim za čitanje i razumevanje, a takođe pruža i veći stepen ponovne upotrebljivosti koda. Objekti pružaju i intuitivniji način rada sa podacima tako što omogućavaju definisanje načina interakcije sa objektima i načina komunikacije sa drugim objektima.

Objektno orijentisano programiranje (OOP) ne zamenjuje tradicionalne metode programiranja. Umesto toga, OOP predstavlja dodatni alat za rešavanje složenih problema kada je to potrebno.

Objektno orijentisano programiranje uvodi i nekoliko drugih korisnih koncepata: apstrakcija, enkapsulacija, nasleđivanje i polimorfizam. Navedeni koncepti će detaljno biti opisani u narednih nekoliko poglavlja.

Termin "objekat" je malo drugačije definisan u odnosu na tradicionalno programiranje. U tradicionalnom programiranju, objekat je deo memorije koji se koristi za čuvanje neke vrednosti. U objektno orijentisanom programiranju, "objekat" podrazumeva i ono na šta se odnosi u tradicionalnom programiranju ali takođe predstavlja i kombinaciju osobina i ponašanja. Ubuduće, kada bude korišćen pojam "objekat", biće reč o objektima u objektno orijentisanom smislu.

Klase i članovi klasa

Iako jezik C++ pruža niz osnovnih tipova podataka (npr. `char`, `int`, `long`, `float`, `double`, itd.) koji su često dovoljni za rešavanje relativno jednostavnih problema, često može biti teško rešiti složene probleme koristeći samo ove jednostavne tipove podataka. Jedna od korisnijih funkcija jezika C++ je mogućnost definisanja sopstvenih tipova podataka koji odgovaraju problemu koji se rešava. Nabrojivi tipovi i strukture definisani u prethodnim poglavljima su primeri složenih korisnički definisanih tipova podataka.

U nastavku je dat primer definicije strukturnog tipa podataka koji se može upotrebiti za čuvanje podataka o datumima.

```
1  struct DateStruct
2  {
3      int year;
4      int month;
5      int day;
6  };
```

Nabrojivi tipovi i strukture podataka (strukture koje sadrže samo promenljive) predstavljaju tradicionalni ne-objektno orijentisan programerski koncept, jer oni mogu čuvati samo podatke. Ako je potrebno odštampati datum na ekranu, ima smisla napisati funkciju za to. Sledi kompletan program.

```
#include <iostream>

struct DateStruct
{
   int year;
   int month;
   int day;
};

void print(DateStruct &date)
{
   std::cout << date.year << "/" << date.month << "/" << date.day;
}

int main()
{
   DateStruct today = { 2020, 10, 14 };
   today.day = 16;
   print(today);

   return 0;
}
```

Izlaz iz programa će biti:

```
2020/10/16
```

Klase

U svetu objektno orijentisanog programiranja često je potrebno da korisnički definisani tipovi podataka ne poseduju samo podatke, već i funkcije koje rade sa podacima. U jeziku C++ se takav tip podataka definiše upotrebom ključne reči `class`. Korišćenjem ključne reči `class` definiše se novi korisnički definisani tip koji se naziva klasa. Klase su veoma slične strukturama, osim što klase pružaju mnogo veće mogućnosti i fleksibilnost. Sledeća struktura i klasa su praktično identične:

```
struct DateStruct
{
   int year;
   int month;
   int day;
};

class DateClass
{
public:
   int m_year;
   int m_month;
   int m_day;
};
```

Za sada je jedina značajna razlika ključna reč `public` u klasi. Kao i deklaracija strukturnog tipa, deklaracija klase takođe ne zauzima memoriju računara. Deklaracija klase definiše samo izgled klase.

Kao i sa strukturom, da biste koristili klasu, mora se deklarisati promenljiva tipa klase:

```
DateStruct today {2020, 10, 14};
```

Definisanje promenljive neke klase se naziva istanciranje klase. Sama promenljiva se naziva instanca klase. Promenljiva (podatak) čiji je tip neka klasa takođe se naziva i objektom. Definisanje promenljive ugrađenog tipa (npr. `int x`) alocira memoriju za tu promenljivu. Takođe, instanciranje objekta neke klase alocira memoriju za taj objekat.

Funkcije članice (metode)

Pored čuvanja podataka, klase mogu sadržati i funkcije. Funkcije definisane unutar klase se nazivaju funkcije članice (ili metode). Funkcije članice mogu se definisati unutar ili izvan definicije klase. Za početak biće pokazano kako se definišu unutar klase.

Sledi klasa `DateClass` sa funkcijom članicom za štampanje datuma:

```
class DateClass
{
public:
   int m_year;
   int m_month;
   int m_day;

   void print() // definicija funkcije članice print()
   {
      std::cout << m_year << "/" << m_month << "/" << m_day;
   }
};
```

Kao i članovima strukture, članovima klase (promenljivama i funkcijama) se pristupa pomoću operatora izbora (operatora selekcije) člana (.):

```
#include <iostream>

class DateClass
{
public:
   int m_year;
   int m_month;
   int m_day;

   void print()
   {
      std::cout << m_year << "/" << m_month << "/" << m_day;
   }
};

int main()
{
   DateClass today = { 2023, 10, 14 };

   today.m_day = 16;
   today.print();

   return 0;
}
```

Izlaz iz programa će biti:

```
2023/10/16
```

Ovaj program je veoma sličan strukturnoj verziji koja je prethodno pokazana.

Međutim, postoji nekoliko značajnih razlika. U `DateStruct` verziji funkcije `print()` iz prethodnog primera, funkciji `print()` se mora proslediti argument, odnosno podatak koji treba odštampati. U suprotnom, funkcija `print()` ne bi imala informaciju o tome koji podatak strukturnog tipa treba koristiti. Takođe, u funkciji je neophodno eksplicitno referenciranje na njen ulazni parametar (koji je prethodno izjednačen sa argumentom koji je funkciji prosleđen prilikom pozivanja).

Funkcije članice funkcionišu malo drugačije: svi pozivi funkcije članice moraju biti povezani sa nekim objektom klase. Kada se pozove "`today.print()`", kompajleru se saopštava da pozove funkciju članicu `print()`, koja je povezana s objektom `today`.

```
   void print()
   {
      std::cout << m_year << "/" << m_month << "/" << m_day;
   }
```

U prethodnom primeru promenljive `m_year`, `m_month` i `m_day` se odnose na objekat kome su pridružene (što je definisano pozivaocem).

Dakle, kada se pozove `today.print()`, kompajler promenljivu `m_day` tumači kao `today.m_day`, `m_month` kao `today.m_month` i `m_year` kao `today.m_year`. Prema tome, ukoliko se pozove `tomorrow.print()`, `m_day` će se odnositi na sutrašnji dan (tj. na objekat `tomorrow`).

Ključna razlika je da se funkcijama koje nisu članice klase moraju proslediti podaci sa kojima će raditi. U slučaju funkcija članica klase je drugačije, jer uvek postoji objekat klase sa kojim se radi.

Korišćenje prefiksa "`m_`" (tzv. mađarska notacija) za promenljive – članove klase pomaže u razlikovanju promenljivih – članova od parametara funkcija ili lokalnih promenljivih unutar funkcija članica. Ovo je korisno iz nekoliko razloga. Prvo, kada se uoči prefiks "`m_`" u nazivu promenljive, potpuno je jasno da se menjanjem tog podatka menja stanje objekta te klase. Drugo, za razliku od parametara funkcija ili lokalnih promenljivih, koje su deklarisane unutar funkcije, promenljive – članovi se deklarišu u deklaraciji klase. Dakle, ako je potrebno videti kako je deklarisana takva promenljiva, unapred se zna da bi trebalo pogledati deklaraciju klase a ne telo funkcije. Po dogovoru, imena klasa treba da počinju velikim slovom.

Sledi još jedan primer klase.

```
#include <iostream>
#include <string>
class Employee
{
public:
   std::string m_name;
   int m_id;
   double m_wage;

   void print()
   {
      std::cout << "Ime: " << m_name <<
                "  ID: " << m_id <<
                "  Zarada: $" << m_wage << '\n';
   }
};
int main()
{
   // Deklarisanje dva objekta klase Employee
   Employee aleksandra = { "Aleksandra", 1, 25.00 };
   Employee jovan = { "Jovan", 2, 22.25 };
   // Štampanje informacija o zaposlenima
   aleksandra.print();
```

```
24      jovan.print();
25       return 0;
26  }
```

Izlaz iz ovog programa je:

```
Ime: Aleksandra ID: 1  Zarada: $25
Ime: Jovan ID: 2  Zarada: $22.25
```

Zaključak

Ključna reč `class` omogućava kreiranje novog tipa podataka u jeziku C++, koji može sadržati i promenljive – članove i funkcije članice. Klase predstavljaju osnovu objektno orijentisanog programiranja. Naredna poglavlja će biti posvećena mogućnostima koje pruža ovaj koncept programiranja.

Specifikatori pristupa `public` i `private`

U programu koji sledi, deklarisana je promenljiva tipa `DateStruct`, a zatim je direktno pristupano njenim članovima kako bi se inicijalizovali. Ovo funkcioniše zato što su svi članovi strukture podrazumevano javni članovi. Javni članovi su članovi strukture ili klase kojima se može pristupiti izvan strukture ili klase. U ovom slučaju, funkcija `main()` je izvan strukture, ali može nesmetano pristupiti članovima `month`, `day` i `year`, jer su oni podrazumevano javni.

```
1   struct DateStruct
2   {
3      int month;
4      int day;
5      int year;
6   };
7
8   int main()
9   {
10     DateStruct date;
11     date.month = 10;
12     date.day = 14;
13     date.year= 2020;
14
15     return 0;
16  }
```

Po analogiji sa prethodnim primerom, napisan je sličan program. Jedina razlika je korišćenje klasnog umesto strukturnog tipa.

```
1   class DateClass  // članovi klase su podrazumevano privatni
2   {
3      int m_month; // privatni član dostupan na nivou klase
```

```
    int m_day;   // privatni član dostupan na nivou klase
    int m_year;  // privatni član dostupan na nivou klase
};
```

```
int main()
{
    DateClass date;
    date.m_month = 10;  // greška
    date.m_day = 14;    // greška
    date.m_year = 2020; // greška

    return 0;
}
```

Kompajliranjem prethodnog koda kompajler bi signalizirao da postoje greške prevođenja. Razlog je to što su svi članovi klase podrazumevano privatni. Privatni članovi su članovi klase kojima mogu pristupiti samo drugi članovi klase. Pošto funkcija `main()` nije član klase `DateClass`, ona nema pristup privatnim članovima promenljive `date`.

Specifikatori pristupa

Iako su članovi klase podrazumevano privatni, mogu se preinačiti u javne korišćenjem ključne reči `public`:

```
class DateClass
{
public:
    int m_month; // javni član, dostupan svuda preko imena objekta
    int m_day;   // javni član, dostupan svuda preko imena objekta
    int m_year;  // javni član, dostupan svuda preko imena objekta
};

int main()
{
    DateClass date;
    date.m_month = 10;  // Dostupno
    date.m_day = 14;    // Dostupno
    date.m_year = 2020; // Dostupno

    return 0;
}
```

Pošto su članovi klase `DateClass` sada javni, njima se može direktno pristupati iz funkcije `main()`.

Ključna reč `public` naziva se specifikatorom pristupa. Specifikatori pristupa definišu ko ima pristup članovima klase čija se deklaracija nalazi u produžetku

koda iza pomenutog specifikatora. Svaki od članova klase stiče nivo pristupa prethodnog specifikatora pristupa. Ako u kodu nije naveden nijedan specifikator pristupa, podrazumevani specifikator pristupa će biti `private`.

U jeziku C++ postoje tri ključne reči za obezbeđivanje različitih nivoa pristupa: `public`, `private` i `protected`. Specifikatori `public` i `private` se koriste kako bi članove koji su deklarisani iza njih proglasili javnim ili privatnim, respektivno. Treći specifikator pristupa, `protected`, funkcioniše slično kao privatni. Razlika između specifikatora `private` i `protected` biće objašnjena u temi "Nasleđivanje".

Kombinovanje specifikatora pristupa

Klase mogu imati više specifikatora pristupa, kako bi se definisao različit nivo pristupa različitim članovima klase. Generalno posmatrano, promenljive – članovi obično se proglašavaju privatnim, a funkcije članice javnim. Sledi primer klase koja koristi i privatni i javni pristup.

```
#include <iostream>
class DateClass  // članovi klase su podrazumevano privatni
{
   int m_month; // privatni član, dostupan samo unutar klase
   int m_day;   // privatni član, dostupan samo unutar klase
   int m_year;  // privatni član, dostupan samo unutar klase

public:
   // javna funkcija, dostupna svuda preko imena objekta
   void setDate(int month, int day, int year)
   {
      m_month = month;
      m_day = day;
      m_year = year;
   }
   // javna funkcija, dostupna svuda preko imena objekta
   void print()
   {
      std::cout << m_month << "/" << m_day << "/" << m_year;
   }
};

int main()
{
   DateClass date;
   date.setDate(10, 14, 2020); // Poziv javne funkcije
   date.print();               // Poziv javne funkcije
   return 0;
}
```

Prethodni program će štampati:

10/14/2020

Iako se promenljivama – članovima `m_month`, `m_day` i `m_year` ne može pristupiti direktno iz glavnog programa (jer su oni privatni), postoji mogućnost da im se indirektno pristupi pomoću javnih funkcija članica `setDate()` i `print()`.

Grupa javnih članova klase često se naziva javnim interfejsom. Pošto se javnim članovima može pristupiti izvan klase, javni interfejs definiše kakav će interfejs videti korisnici klase. Funkcija `main()` je ograničena na podešavanje datuma i štampanje datuma. Klasa štiti promenljive – članove od direktnog pristupa ili direktne promene.

Kontrola pristupa pomoću specifikatora pristupa radi na nivou klase!

```
1   #include <iostream>
2
3   class DateClass
3   {
4      int m_month;
5      int m_day;
6      int m_year;
7
8   public:
9      void setDate(int month, int day, int year)
10     {
11        m_month = month;
12        m_day = day;
13        m_year = year;
14     }
15
16     void print()
17     {
18        std::cout << m_month << "/" << m_day << "/" << m_year;
19     }
20
21     void copyFrom(const DateClass &d)
22     {
23        // Direktan pristup privatnim članovima objekta d!
24        m_month = d.m_month;
25        m_day = d.m_day;
26        m_year = d.m_year;
27     }
28  };
29
30  int main()
31  {
32     DateClass date;
33     date.setDate(10, 14, 2020);
34
35     DateClass copy;
36     copy.copyFrom(date);
37
```

```
38      copy.print();
39  }
```

Jedna od nijansi jezika C++ koja se često propusti ili pogrešno shvati je činjenica da kontrola pristupa radi na nivou klase, a ne na nivou objekta. To znači da kada funkcija ima pristup privatnim članovima klase, može pristupiti privatnim članovima bilo kojeg objekta te klase koji je u funkciji dostupan (kao parametar funkcije ili kao član klase).

U prethodnom primeru, `copyFrom()` je funkcija članica klase `DateClass`, koja pristupa privatnim članovima klase `DateClass`. Ovo znači da `copyFrom()` može direktno da pristupa privatnim članovima objekta na kome radi (odnosno u koga kopira vrednosti promenljivih), a takođe znači da ima direktan pristup privatnim članovima `DateClass` parametra d (objekta iz koga preuzima vrednosti njegovih članova)! Ako bi parametar d bio objekat neke druge klase, njegovim privatnim članovima se ne bi moglo direktno pristupiti. Ovo može biti naročito korisno kada je potrebno kopirati članove iz jednog objekta klase u drugi objekat iste klase. Ova osobina je veoma korisna kod preklapanja operatora. O tome će biti više reči kasnije.

Funkcije pristupa i enkapsulacija

U prethodnoj lekciji pomenuto je da su promenljive – članovi klase najčešće privatni. Programeri koji se prvi put susreću sa objektno orijentisanim pristupom često imaju teškoće da shvate zašto bi uopšte želeli da to bude tako.

U savremenom životu imamo pristup mnogim elektronskim uređajima. Televizor ima daljinski upravljač koji se može koristiti za uključivanje / isključivanje televizora. Automobil se koristi za prevoz do određene destinacije. Digitalni fotoaparat omogućava generisanje fotografija u digitalnoj formi. Sva tri primera imaju zajednički obrazac: sve pomenute sprave pružaju jednostavan interfejs za korišćenje (dugme, upravljač, itd.) kako bi se izvršila određena akcija. Međutim, način na koji ovi uređaji zapravo rade je sakriven od korisnika. Kada se pritisne dugme na daljinskom upravljaču, nije neophodno znati šta radi daljinski upravljač kako bi komunicirao sa televizorom. Kada se pritisne pedala gasa na automobilu, nije potrebno poznavati rad motora ili transmisije. Kada se koristi digitalni fotoaparat, nije potrebno znati kako objektiv aparata prikuplja svetlost i zapisuje sliku na CCD ili CMOS senzoru. Ovo razdvajanje interfejsa i implementacije je izuzetno korisno jer omogućava korišćenje objekata bez razumevanja kako oni rade. Time se znatno smanjuje složenost korišćenja objekata i povećava broj objekata s kojima mogu da komuniciraju.

Iz sličnih razloga, razdvajanje implementacije i interfejsa je korisno i u svetu programiranja.

Enkapsulacija

U objektno-orijentisanom programiranju, enkapsulacija (takođe nazvana skrivanje informacija ili učauravanje) je proces sakrivanja detalja o implementaciji od korisnika. Umesto toga, korisnici objekta pristupaju objektu putem javnog interfejsa. Na ovaj način korisnici mogu da koriste objekat bez potrebe da razumeju način kako je on implementiran.

U jeziku C++, enkapsulacija je implementirana preko specifikatora pristupa. Obično su sve promenljive – članovi klase privatni (sakrivaju se detalji implementacije), a većina funkcija članica je javna (javni interfejs za korisnika). Iako zahtev da korisnici klase koriste javni interfejs možda izgleda komplikovanije od direktnog pristupa javnim promenljivama – članovima, time se omogućavaju neke korisne stvari koje pomažu upotrebljivosti i održivosti klase.

Enkapsulirane klase se lakše koristite i smanjuju složenost programa. Sa potpuno enkapsuliranom klasom potrebno je znati samo koje su funkcije članice javno dostupne za korišćenje klase, koje argumente uzimaju i koje vrednosti vraćaju. Nije važno kako je klasa interno realizovana. Na primer, klasa koja sadrži listu imena mogla je biti implementirana korišćenjem dinamičkog niza, `std::array`, `std::vector`, `std::map`, `std::list` ili nekog drugog strukturnog podatka. Da bi se klasa koristila, nije potrebno znati šta je od prethodno navedenog korišćeno. Ovim se dramatično smanjuje složenost programa, a takođe i mogućnost pojave grešaka, i predstavlja ključnu prednost enkapsulacije.

Sve klase u standardnoj biblioteci C++ su enkapsulirane.

Enkapsulirane klase pomažu u zaštiti podataka i sprečavanju zloupotrebe. Globalne promenljive su opasne jer nema kontrole nad pristupom globalnoj promenljivoj (odnosno, ko može da je promeni). Klase sa javnim članovima imaju isti problem, samo u manjoj meri.

Problem sa javno vidljivim promenljivama se može ilustrovati na primeru klase za stringove – nizove karaktera.

```
class MyString
{
    char *m_string; // pokazivač za dinamičko alociranje memorije
    int m_length;   // veličina alociranog prostora
};
```

Ove dve promenljive imaju interesantnu vezu: `m_length` uvek treba da bude jednaka dužini niza koji se čuva u promenljivoj `m_string`. Ukoliko bi `m_length` bio javni član klase, svako bi mogao da promeni dužinu stringa bez promene `m_string` (ili obratno). To bi klasu dovelo u nekonzistentno stanje, što bi moglo dovesti do raznih problema. Promenom nivoa pristupa promenljivama `m_length` i `m_string` iz javnog u privatni, korisnici su prisiljeni da koriste samo javne funkcije za rad s klasom, a te funkcije članice mogu osigurati da se vrednosti promenljivih `m_length` i `m_string` uvek postavljaju na odgovarajući način.

Takođe, korisniku klase se može pomoći u zaštiti od pogrešnog korišćenja klase.

```
class IntArray
{
public:
    int m_array[10];
};
```

Ako korisnici mogu direktno pristupiti nizu definisanom unutar klase `IntArray`, imali bi mogućnost da pristupe članu niza sa nevažećim indeksom, što bi dovelo do neočekivanih rezultata.

```
int main()
{
    IntArray array;
    array.m_array[16] = 2; // indeks izvan granica niza [0]-[9]
}
```

Međutim, ako bi niz bio privatni član, korisnik klase će biti primoran da koristi funkciju koja potvrđuje da je indeks validan:

```
class IntArray
{
private:
   int m_array[10];//Privatni član, nije dozvoljen direktan pristup

public:
   void setValue(int index, int value)
   {
      // Ako indeks nije ispravan izlazi se iz funkcije
      if (index < 0 || index >= 10)
         return;

      m_array[index] = value;
   }
};
```

Na taj način se štiti integritet programa.

Enkapsulirane klase se lakše menjaju. Program koji je dat u nastavku za sada dobro funkcioniše. Međutim, šta bi se desilo ako bi se javila potreba za promenom imena ili tipa promenljive `m_value1`? To će se odraziti ne samo na program na kome se trenutno radi, već i na sve druge programe koji koriste ovu klasu.

```
#include <iostream>

class Something
{
public:
    int m_value1;
    int m_value2;
    int m_value3;
};
```

```
int main()
{
    Something something;
    something.m_value1 = 5;
    std::cout << something.m_value1 << '\n';
}
```

Enkapsulacija daje mogućnost da se promeni način na koji su klase implementirane, bez uticaja na programe koji ih koriste.

Sledi enkapsulirana verzija prethodne klase koja koristi funkcije za pristup promenljivoj `m_value1`:

```
#include <iostream>

class Something
{
private:
    int m_value[3];

public:
    void setValue1(int value) { m_value[0] = value; }
    int getValue1() { return m_value[0]; }
};

int main()
{
    Something something;
    something.setValue1(5);
    std::cout << something.getValue1() << '\n';
}
```

S obzirom da nisu izmenjeni potpisi funkcija u javnom interfejsu klase, programi koji koriste klase nastavljaju da rade bez ikakvih promena.

Slično tome, ako se softver na pametnom TV-u u toku noći ažurirao, sledećeg jutra se verovatno neće primetiti ikakva razlika u načinu menjanja kanala.

Enkapsulirane klase se lakše debagiraju. Na kraju, enkapsulacija pomaže da se programi jednostavnije debagiraju. Često je razlog zašto program loše radi pogrešna vrednost neke promenljive. Ako su svi u mogućnosti da direktno pristupaju promenljivama, otkrivanje dela koda koji je modifikovao vrednost neke promenljive može biti veoma teško. Međutim, ako svi pozivaoci moraju pozvati istu javnu funkciju kako bi modifikovali vrednost, onda se program može jednostavno zaustavljati u toj funkciji i posmatrati kako svaki pozivalac menja vrednost, dokle god se ne utvrdi gde je došlo do dodele vrednosti koja je izazvala grešku.

Funkcije pristupa

U zavisnosti od klase, može biti zgodno (u kontekstu onoga što klasa radi) da postoji mogućnost jednostavnog preuzimanja vrednosti nekog privatnog člana ili postavljanja njegove vrednosti.

Funkcija pristupa je kratka javna funkcija čiji je zadatak da preuzme ili promeni vrednost privatne promenljive – člana. Na primer, u klasi `MyString` može postojati funkcija koja pozivaocu vraća dužinu stringa:

```
class MyString
{
private:
    char *m_string;
    int m_length;

public:
    int getLength() { return m_length; }
};
```

Funkcija `getLength()` je funkcija pristupa koja jednostavno vraća vrednost `m_length`.

Obično postoje dve varijante pristupnih funkcija: funkcije koje vraćaju vrednost privatnog člana (getter) i funkcije koje postavljaju vrednost privatnog člana (setter).

Sledi primer klase sa ovakvim funkcijama.

```
class Date
{
private:
    int m_month;
    int m_day;
    int m_year;

public:
    int getMonth() { return m_month; }
    void setMonth(int month) { m_month = month; }

    int getDay() { return m_day; }
    void setDay(int day) { m_day = day; }

    int getYear() { return m_year; }
    void setYear(int year) { m_year = year; }
};
```

U ovoj klasi nema problema sa omogućavanjem korisniku da direktno dobije ili promeni bilo koji od članova, tako da je obezbeđen kompletan skup pristupnih funkcija. U prethodnom primeru klase `MyString` ne postoji funkcija pristupa koja bi menjala promenljivu `m_length` jer se ne sme omogućiti korisniku da direktno pristupa promenljivoj i na taj način podešava podatak o dužini. Dužina treba da se promeni samo kad se string promeni, a to bi radila neka druga funkcija koja se bavi alociranjem, realociranjem ili oslobađanjem memorijskog prostora.

Funkcije pristupa koje vraćaju vrednost nekog člana bi trebalo da vraćaju ili vrednost ili konstantnu referencu.

Konstruktori

Konstruktor je posebna vrsta funkcije članice klase koja se automatski poziva kada se objekat te klase instancira. Konstruktori se obično koriste za inicijalizaciju promenljivih – članova klase odgovarajućim podrazumevanim ili korisnički definisanim vrednostima, ili da izvrše sve korake podešavanja neophodne za upotrebu klase (npr. otvaranje datoteke ili baze podataka).

Za razliku od normalnih funkcija članica klase, konstruktori imaju specifična pravila kako se moraju nazvati:

1. Konstruktori uvek imaju isti naziv kao i klasa (računajući i velika i mala slova)
2. Konstruktori nemaju povratni tip (čak ni `void`)

Konstruktori su namenjeni samo za inicijalizaciju. Ne bi trebalo pozivati konstruktor kako bi se ponovo inicijalizovao postojeći objekat. Iako se to može kompajlirati, rezultati neće biti ono što je bila namera (umesto toga, kompajler će stvoriti privremeni objekat, a zatim ga odbaciti).

Podrazumevani konstruktori

Konstruktor koji nema parametre (ili ima parametre, pri čemu svi poseduju podrazumevane vrednosti) se zove podrazumevani konstruktor. Podrazumevani konstruktor se poziva ako inicijalne vrednosti nisu obezbeđene od strane korisnika klase.

Sledi primer klase koja ima podrazumevani konstruktor.

```cpp
#include <iostream>

class Fraction
{
private:
   int m_numerator;
   int m_denominator;

public:
   Fraction() // Podrazumevani konstruktor
   {
      m_numerator = 0;
      m_denominator = 1;
   }

   int getNumerator() { return m_numerator; }
   int getDenominator() { return m_denominator; }
   double getValue()
   {
      return (double) (m_numerator) / m_denominator;
   }
};

int main()
{
   Fraction f; // Poziv podrazumevanog konstruktora
   std::cout << f.getNumerator() << "/" << f.getDenominator();

   return 0;
}
```

Ova klasa je dizajnirana da čuva razlomak (`Fraction`) pomoću dva cela broja: brojilac i imenilac. Definisan je podrazumevani konstruktor pod nazivom `Fraction` (isto ime kao i ime klase).

S obzirom da se objekat tipa `Fraction` instancira pozivom konstruktora bez argumenata, podrazumevani konstruktor će biti pozvan odmah nakon što se alocira memorija za objekat, i objekat će biti inicijalizovan.

Izlaz iz ovog programa će biti:

```
0/1
```

Imenilac i brojilac su inicijalizovani vrednostima koje su postavljene u podrazumevanom konstruktoru. Bez podrazumevanog konstruktora, brojilac i imenilac bi imali proizvoljne vrednosti sve dok im se eksplicitno ne dodele konkretne vrednosti.

Ugrađeni (implicitno definisani) konstruktor

Ukoliko klasa nema konstruktore, jezik C++ će automatski napraviti javni podrazumevani konstruktor. Ovo se ponekad naziva implicitni konstruktor (ili implicitno generisan konstruktor).

Sledi primer klase bez eksplicitno definisanog konstruktora.

```
class Date
{
private:
   int m_year = 1900;
   int m_month = 1;
   int m_day = 1;
};
```

Klasa `Date` nema konstruktor. Prema tome, kompajler će generisati konstruktor koji se ponaša identično kao konstruktor iz primera koji sledi.

```
class Date
{
private:
   int m_year = 1900;
   int m_month = 1;
   int m_day = 1;

public:
   Date() { };
};
```

Ovaj konstruktor omogućava stvaranje objekata klase, ali ne obavlja nikakvu inicijalizaciju ili dodelu vrednosti samim članovima klase.

Iako se implicitno definisani konstruktor ne može videti, može se pokazati da on zaista postoji, jer da ga nema objekat ne bi bilo moguće instancirati.

```
class Date
{
private:
   int m_year = 1900;
   int m_month = 1;
   int m_day = 1;

   // Nema eksplicitno definisanog podrazumevanog konstruktora.
   // Jezik C++ će ga definisati implicitno.
};

int main()
{
   Date date; // Poziv implicitno definisanog konstruktora
   return 0;
```

```
16  }
```

Prethodni kod se kompajlira, jer objekat `date` koristi implicitni konstruktor (koji je javno dostupan).

Ako klasa ima bilo koji drugi konstruktor, implicitno definisani konstruktor neće biti obezbeđen.

```
1   class Date
2   {
3   private:
4      int m_year = 1900;
5      int m_month = 1;
6      int m_day = 1;
7
8   public:
9      Date(int year, int month, int day)
10     {
11        m_year = year;
12        m_month = month;
13        m_day = day;
14     }
15     // Ne postoji implicitno definisani podrazumevani konstruktor
16     // zato što je definisan konstruktor sa parametrima
17  };
18
19  int main()
20  {
21     Date date;   // Greška! Ne postoji podrazumevani konstruktor
22     Date today(2020, 10, 14); // Poziv konstruktora sa tri parametra
23
24     return 0;
25  }
```

Uopšteno posmatrano, dobra praksa je da uvek bude obezbeđen minimalno jedan konstruktor u klasi. Ovim se eksplicitno kontroliše način stvaranja objekata i sprečava da klasa kasnije izgubi neki deo funkcionalnosti kada se dodaju drugi konstruktori.

Klase koje sadrže objekte drugih klasa

Klasa može sadržavati objekte drugih klasa kao članove. Podrazumevano, kada se instancira objekat ovakve klase, konstruktori članova će takođe biti pozvani. Ovo se dešava pre nego što izvrši telo konstruktora klase koja sadrži objekte drugih klasa.

```
#include <iostream>

class A
{
public:
    A() { std::cout << "A\n"; }
};

class B
{
private:
    A m_a; // Klasa B sadrži objekat klase A kao član
public:
    B() { std::cout << "B\n"; }
};

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

Prethodni program će odštampati sledeće:

```
A
B
```

Kada se instancira promenljiva b, poziva se konstruktor klase B. Pre nego što se izvrši telo konstruktora, promenljiva m_a se inicijalizuje, pozivajući podrazumevani konstruktor klase A. Konstruktor klase A štampa "A". Tada se kontrola programa vraća nazad u konstruktor klase B, i izvršava se telo konstruktora klase B.

Sve ovo ima zaista ima smisla, jer u konstruktoru klase B će možda biti korišćena vrednost promenljive m_a, i zato je važno da ona pre toga dobije početnu vrednost.

Destruktori

Destruktor je još jedna posebna vrsta funkcije članice klase koja se izvršava kada se objekat te klase uništava. Budući da su konstruktori dizajnirani da inicijalizuju klasu, destruktori su dizajnirani da pomognu u čišćenju klase.

Kada objekat izađe iz opsega ili se eksplicitno obriše korišćenjem ključne reči `delete`, poziva se destruktor klase (ako postoji) da obavi "čišćenje" klase, ukoliko je to potrebno, pre nego što se objekat ukloni iz memorije. Za jednostavne klase (one koje samo inicijalizuju vrednosti članova), destruktor nije potreban jer će jezik C++ automatski očistiti memoriju.

Međutim, ako objekat klase drži bilo koji resurs (npr. dinamički alociranu memoriju ili datoteku), ili ako je neophodno učiniti bilo kakvu vrstu održavanja pre nego što se objekat uništi, destruktor je savršeno mesto za to, s obzirom da je to poslednja funkcija koja će biti pozvana pre nego što se objekat uništi.

Ime destruktora

Kao i konstruktori, destruktori imaju specifična pravila imenovanja:

1) destruktor mora imati isti naziv kao klasa, kome prethodi karakter tilda (~),

2) destruktor ne može imati argumente,

3) destruktor nema povratni tip.

Klasa može imati samo jedan destruktor. Kao i konstruktori, destruktori se ne pozivaju eksplicitno. Međutim, destruktori mogu da pozivaju druge funkcije članice, pošto objekat nije uništen sve dok se ne izvrši telo destruktora. Sledi primer jednostavne klase koja koristi destruktor.

Program koji sledi će odštampati sledeći tekst:

```
The value of element 5 is: 6
```

U prvoj liniji `main()` funkcije je instanciran novi objekt klase `IntArray` čije je ime `ar`, i čija dužina treba da bude `10`. Ovo poziva konstruktor koji dinamički alocira memoriju za članove niza. Ovde se mora koristiti dinamičko alociranje memorije jer u vreme kompajliranja nije poznato kolika će biti dužina niza (onaj ko koristi klasu će to definisati).

Na kraju `main()` funkcije promenljiva `ar` izlazi iz opsega. Ovo dovodi do pozivanja destruktora `~IntArray()`, koji oslobađa memoriju alociranu za niz koji je prethodno alociran u konstruktoru.

```
1   #include <iostream>
2
3   class IntArray
4   {
5   private:
6      int *m_array;
7      int m_length;
8
9   public:
10     IntArray(int length) // Konstruktor
11     {
12        m_array = new int[length];
13        m_length = length;
```

```
14      }
15      ~IntArray() // Destruktor
16      {
17         delete[] m_array; // Oslobađanje dinamički alocirane memorije
18      }
19      void setValue(int index, int value) { m_array[index] = value; }
20      int& getValue(int index) { return m_array[index]; }
21      int getLength() { return m_length; }
22  };
23  int main()
24  {
25      IntArray ar(10); // alociranje prostora za 10 celih brojeva
26      for (int count=0; count < 10; ++count) ar.setValue(count,
27  count+1);
28
29      std::cout << "The value of element 5 is: " << ar.getValue(5);
30      return 0;
31  } // objekat ar izlazi iz opsega i uništava se.
32    // Implicitno se poziva destruktor!
```

Pozivanje konstruktora i destruktora

Kao što je ranije pomenuto, konstruktor se poziva kada se kreira objekat, a destruktor se poziva kada se objekat uništava. Sledeći primer demonstrira pozive konstruktora i destruktora štampanjem odgovarajućeg teksta pomoću objekta `cout` unutar tela konstruktora i destruktora.

```
#include <iostream>
class Simple
{
private:
   int m_nID;

public:
   Simple(int nID)
   {
      std::cout << "Constructing Simple " << nID << '\n';
      m_nID = nID;
   }
   ~Simple()
   {
      std::cout << "Destructing Simple" << m_nID << '\n';
   }
   int getID() { return m_nID; }
};

int main()
{
   // Instanciranje objekta klase Simple u stack memoriji
   Simple simple(1);
   std::cout << simple.getID() << '\n';

   // Dinamičko instanciranje objekta klase Simple u heap memoriji
   Simple *pSimple = new Simple(2);
   std::cout << pSimple->getID() << '\n';
   delete pSimple;

   return 0;
} // Objekat simple izlazi iz opsega
```

Prethodni program daje sledeći rezultat kao izlaz:

```
Constructing Simple 1
1
Constructing Simple 2
2
Destructing Simple 2
Destructing Simple 1
```

Objekat `simple` (koji ima vrednost 1) je uništen nakon objekta na koji pokazuje pokazivač `pSimple` (koji ima vrednost 2) jer je pokazivač `pSimple` izbrisan pre kraja funkcije, dok objekat `simple` postoji sve do izlaska iz funkcije `main()`.

Globalne promenljive se kreiraju pre funkcije `main()` i uništavaju nakon izvršavanja funkcije `main()`.

Skriveni pokazivač `this`

Sledi primer jednostavne klase koja sadrži samo jednu celobrojnu promenljivu.

```
1  #include <iostream>
2
3  class Simple
4  {
5  private:
6     int m_id;
7
8  public:
9     Simple(int id)
10    {
11       setID(id);
12    }
13
14    void setID(int id) { m_id = id; }
15    int getID() { return m_id; }
16 };
17
18 int main()
19 {
20    Simple simple(1);
21    simple.setID(2);
22    std::cout << simple.getID() << '\n';
23
24    return 0;
25 }
```

Poziv funkcije `setID(int id)` zahteva prosleđivanje jednog argumenta.

```
1     simple.setID(2);
```

Iako poziv funkcije `setID` izgleda kao da funkcija ima samo jedan parametar (odnosno argument), ona zapravo ima dva! Kada se kompajlira, kompajler pretvara liniju `simple.setID(2);` u sledeći kod:

```
1     setID(&simple, 2);
```

Ovo je sada samo standardni poziv funkcije, a objekat `simple` (koji je prethodno bio prefiks) sada se prihvata po adresi kao argument funkcije. Pošto poziv funkcije sada ima dodatni argument, definicija funkcije članice mora biti modifikovana tako da prihvati (i koristi) ovaj argument kao parametar. Shodno tome, sledeću funkciju članicu:

```
1  void simple.setID(2){m_id = id;};
```

kompajler će konvertovati u:

```
1  void setID(Simple* const this, int id) { this->m_id = id; }
```

Kada kompajler kompajlira funkciju članicu, on implicitno dodaje novi parametar funkciji pod nazivom `this`. Ovaj pokazivač je skriveni konstantni pokazivač koji čuva adresu objekta za koji je funkcija članica pozvana.

Unutar funkcije članice, svi članovi klase (funkcije i promenljive) takođe moraju biti ažurirani tako da se odnose na objekat za koji je funkcija članica pozvana. To se lako može uraditi dodavanjem prefiksa "`this->`" svakom od njih. Dakle, u telu funkcije `setID`, promenljiva `m_id` (koja je član klase) pretvorena je u `this->m_id`. Prema tome, kada `*this` pokazuje na objekat `simple`, `this->m_id` će se odnositi na `simple.m_id`.

Dakle:

1) Kada se poziva `simple.setID(2)`, kompajler zapravo poziva `setID(&simple, 2)`.

2) Unutar funkcije `setID`, `*this` pokazivač čuva adresu objekta `simple`.

3) Svi članovi unutar funkcije `setID` imaju prefiks "`this->`". Dakle, iskaz `m_id = id`, kompajler zapravo tretira kao iskaz `this->m_id = id`, koji u ovom slučaju ažurira `simple.m_id` vrednošću `id`.

Dobra vest je da se sve ovo radi u pozadini, i da za programera nije naročito značajno da li zna i da li se seća kako ovo funkcioniše ili ne. Sve što treba znati je da sve normalne funkcije – članice imaju pokazivač `*this`, koji se odnosi na objekat pomoću koga je funkcija pozvana.

Izvorni kod klase i datoteke zaglavlja

Sve klase koje su do sada predstavljene bile su veoma jednostavne (npr. klasa `Date`), pa je kompletna njihova implementacija bila smeštena unutar same definicije klase.

Međutim, klase mogu biti mnogo veće i komplikovanije. Stavljajući sve definicije funkcija članica i promenljivih – članova unutar definicije klase, kod klase postaje sve veći i rad sa takvim kodom sve teži. Osim toga, upotreba već napisane klase zahteva poznavanje samo njenog javnog interfejsa (javne funkcije članice), a ne kako klasa funkcioniše “ispod haube”. Detalji o implementaciji funkcija članica klase su jednostavno suvišni, ukoliko se klasa samo koristi.

```
class Date
{
private:
   int m_year;
   int m_month;
   int m_day;

public:
   Date(int year, int month, int day)
   {
      setDate(year, month, day);
   }

   void setDate(int year, int month, int day)
   {
      m_year = year;
      m_month = month;
      m_day = day;
   }

   int getYear() { return m_year; }
   int getMonth() { return m_month; }
   int getDay()  { return m_day; }
};
```

Jezik C++ omogućava jednostavan način se deklaracija klase odvoji od njene implementacije. Ovo je omogućeno definisanjem funkcija članica klase izvan same definicije klase. Da bi se ovo uradilo, jednostavno treba definisati funkcije članice klase kao da su normalne funkcije, ali im kao prefiks treba dodati ime klase na koju se odnose i operator razrešavanja opsega (`::`) (isto kao i za prostor imena).

Sledi primer prethodno definisane klase Date sa konstruktorom i funkcijom `setDate()` definisanom izvan definicije klase. Prototipovi (deklaracije) ovakvih funkcija i dalje postoje unutar definicije klase, ali je stvarna implementacija premeštena izvan klase.

```
class Date
{
private:
   int m_year;
   int m_month;
   int m_day;
public:
   Date(int year, int month, int day);

   void SetDate(int year, int month, int day);

   int getYear() { return m_year; }
   int getMonth() { return m_month; }
   int getDay()  { return m_day; }
};
```

```
// Konstruktor klase Date
Date::Date(int year, int month, int day)
{
   SetDate(year, month, day);
}

// Funkcija članica klase Date
void Date::SetDate(int year, int month, int day)
{
   m_month = month;
   m_day = day;
   m_year = year;
}
```

S obzirom da su pristupne funkcije često samo jedna linija, one se obično ostavljaju u definiciji klase, iako se i one mogu pomerati izvan klase.

Postavljanje definicije klase u datoteku zaglavlja

Definicije klasa se mogu staviti u datoteke zaglavlja kako bi se olakšala ponovna upotreba u više datoteka ili više projekata. Tradicionalno, definicija klase se stavlja u datoteku zaglavlja istog imena kao i klasa, a funkcije članice definisane izvan klase stavljaju se u `*.cpp` datoteku sa istim imenom kao i klasa.

Sledi kod klase `Date`, podeljen u datoteke `*.h` i `*.cpp`.

`Date.h`:

```
#ifndef DATE_H
#define DATE_H

class Date
{
private:
   int m_year;
   int m_month;
   int m_day;

public:
   Date(int year, int month, int day);

   void SetDate(int year, int month, int day);

   int getYear() { return m_year; }
   int getMonth() { return m_month; }
   int getDay()  { return m_day; }
};

#endif
```

Date.cpp:

```
#include "Date.h"

// Konstruktor klase Date
Date::Date(int year, int month, int day)
{
   SetDate(year, month, day);
}

// Funkcija članica klase Date
void Date::SetDate(int year, int month, int day)
{
   m_month = month;
   m_day = day;
   m_year = year;
}
```

Sada je moguće klasu Date koristiti u bilo kojoj drugoj datoteci jednostavnim uključivanjem datoteke pomoću pretprocesorske direktive `#include "Date.h"`. Datoteka Date.cpp takođe treba biti uključena u projekat koji koristi datoteku Date.h, tako da linker zna kako je implementirana klasa Date.

Konstantni objekti klase

Instancirani objekti klase mogu biti deklarisani kao konstantni korišćenjem ključne reči const. Inicijalizacija se obavlja preko konstruktora klase.

```
const Date date1; // Podrazumevani konstruktor
const Date date2(2020, 10, 16); // Konstruktor sa tri parametra
```

Kada se konstantan objekat neke klase inicijalizuje preko konstruktora, svaki pokušaj izmene članova objekta je onemogućen, jer bi to kršilo konstantnost objekta. Ovo uključuje i direktno menjanje članova klase (ako su javni), ili pozivanje funkcija članica koje menjaju vrednost članova.

```
class Something
{
public:
   int m_value;

   Something(): m_value(0) { }

   void setValue(int value) { m_value = value; }
   int getValue() { return m_value ; }
};
```

```
1  int main()
2  {
3     const Something something; // Podrazumevani konstruktor
4
5     something.m_value = 5; // Greska! Konstantan objekat!
6     something.setValue(5); // Greska! Konstantan objekat!
7
8     return 0;
9  }
```

Kod u linijama 5 i 6 prouzrokuje grešku kompajliranja, jer one krše konstantnost, pokušavajući da direktno promene promenljivu – člana (linija 5) ili pozivanjem funkcije članice (linija 6) koja takođe pokušava da promeni promenljivu – člana.

Konstantne funkcije članice

Objekat `something`, u kodu koji sledi, je prethodno deklarisan kao konstantan.

```
1     std::cout << something.getValue();
```

Ovakav kod će takođe uzrokovati grešku kompajliranja, iako funkcija `getValue()` ne čini ništa kako bi promenila promenljive – članove. Konstantni objekti mogu eksplicitno da pozivaju samo konstantne funkcije članice, a `getValue()` nije deklarisana kao konstantna funkcija. Konstantna funkcija članica je funkcija koja garantuje da neće promeniti objekat ili pozvati bilo koju drugu nekonstantnu funkciju koja može modifikovati objekat.

Da bi funkcija `getValue()` bila konstantna potrebno je dodati ključnu reč `const` na prototip funkcije, nakon liste parametara, ali pre tela funkcije.

```
1  class Something
2  {
3  public:
4     int m_value;
5
6     Something() { m_value= 0; }
7
8     void resetValue() { m_value = 0; }
9     void setValue(int value) { m_value = value; }
10
11    int getValue() const { return m_value; }
12 };
13
```

Sada je funkcija `getValue()` deklarisana kao konstantna, što znači da može biti pozvana iz bilo kog konstantnog ili nekonstantnog objekta, jer je zagarantovano da funkcija neće menjati stanje objekta.

Za funkcije članice definisane izvan definicije klase, ključna reč `const` mora se koristiti na oba mesta: u prototipu funkcije u deklaraciji klase, ali i u definiciji same funkcije:

```
class Something
{
public:
   int m_value;

   Something() { m_value= 0; }

   void resetValue() { m_value = 0; }
   void setValue(int value) { m_value = value; }

   int getValue() const;
};

int Something::getValue() const
{
   return m_value;
}
```

Svaka konstantna funkcija članica koja pokušava da promeni promenljivu – člana ili pozove funkciju članicu koja nije konstantna će prouzrokovati grešku kompajliranja.

```
class Something
{
public:
    int m_value ;

    void resetValue() const { m_value = 0; }
    // Greka! Konstanta funkcija ne može da menja članove klase.
};
```

U ovom primeru, funkcija `resetValue()` je deklarisana kao konstantna funkcija članica. U njoj je učinjen pokušaj promene vrednosti člana `m_value`, što će prouzrokovati grešku kompajliranja.

Konstruktori ne mogu biti označeni kao `const`. Razlog je činjenica da konstruktori moraju biti sposobni da inicijalizuju promenljive – članove klase, a konstantan konstruktor to ne bi mogao da uradi.

Statički članovi klase i statičke funkcije članice

U jeziku C++ postoje još dve mogućnosti upotrebe ključne reči `static`: definisanje statičke promenljive – člana i definisanje statičke funkcije članice. Ove definicije su prilično jednostavne.

Statički članovi klase

Kada se instancira objekat klase, svaki objekat dobija sopstvenu kopiju svih promenljivih – članova.

```
class Something
{
public:
   int m_value;
   Something () {m_value = 1;}
};

int main()
{
    Something first, second;

    first.m_value = 2;

    std::cout << first.m_value << '\n' << second.m_value << '\n';
}
```

U ovom slučaju deklarisana su dva objekta klase `Something`, pa će postojati i dve različite promenljive `m_value`: `first.m_value` i `second.m_value`. Promenljiva `first.m_value` se razlikuje od promenljive `second.m_value` jer pripadaju različitim objektima. Prema tome, prethodno navedeni program će štampati:

```
2
1
```

Članovi klase mogu biti deklarisani kao statički korišćenjem ključne reči `static`. Za razliku od običnih članova, statičke članove dele svi objekti klase. Sledeći program je veoma sličan prethodnom.

```
class Something
{
public:
    static int s_value;
};

int Something::s_value = 1;

int main()
{
    Something first, second;

    first.s_value = 2;

    std::cout << first.s_value << '\n' << second.s_value << '\n';
}
```

Program štampa sledeći izlaz:

```
2
2
```

S obzirom da je `s_value` statička promenljiva – član, promenljivu `s_value` dele svi objekti klase. Shodno tome, promenljiva `first.s_value` je ista promenljiva kao i `second.s_value`. Navedeni primer pokazuje da se promenljivoj čija je vrednost postavljena korišćenjem objekta `first` može pristupiti i korišćenjem objekta `second`!

Iako se statičkim članovima neke klase može pristupiti preko objekata te klase, statički članovi postoje čak i kada nijedan objekat te klase nije instanciran. Slično kao i kod globalnih promenljivih, oni se kreiraju kada se program pokreće i uništavaju kada se program završi.

S obzirom da statičke promenljive – članovi klase nisu deo objekata klase, moraju biti eksplicitno definisani izvan klase, u globalnom opsegu.

U prethodnom primeru je to urađeno pomoću ove linije:

```
1   int Something::s_value = 1;
```

Prethodna linija ima dve svrhe: ona instancira statičku promenljivu – člana (baš kao i globalna promenljiva), a opciono je i inicijalizuje. U ovom slučaju obezbeđena je inicijalna vrednost `1`. Ako inicijalna vrednost nije obezbeđena, jezik C++ inicijalizuje na vrednost `0`.

Statičke funkcije članice

Kao i statičke promenljive – članovi klase, statičke funkcije članice nisu povezane sa nekim određenim objektom.

```
1   class Something
2   {
3   private:
4      static int s_value;
5   public:
6      // Statička funkcija članica
7      static int getValue() { return s_value; }
8   };
9
10  int Something::s_value = 1; // Inicijalizacija statičkog člana
11
12  int main()
13  {
14      std::cout << Something::getValue() << '\n';
15  }
```

S obzirom da statičke funkcije članice nisu povezane sa određenim objektom, one se mogu pozvati direktno koristeći ime klase i operator razrešenja opsega. Kao statične promenljive – članovi, one se takođe mogu pozvati preko objekata ove klase, ali se to ne preporučuje.

Statičke funkcije članice nisu povezane sa objektom i nemaju pokazivač `this`! Pokazivač `this` uvek pokazuje na objekat iz koga je pozvana funkcija članica. Statičke funkcije članice ne rade nad objektom, tako da im ovaj pokazivač nije potreban.

Statičke funkcije članice mogu pristupiti samo statičkim članovima. Jasan je i razlog zašto je to tako: nestatičke promenljive – članovi moraju pripadati nekom objektu klase, a statičke funkcije članice nisu vezane za objekte klase (instance) već za samu klasu.

Sledi primer za generisanje jedinstvenih brojeva koji je već korišćen. Sada je malo izmenjen i napisan u skladu sa principima objektno orijentisanog programiranja.

```
1   #include <iostream>
2
3   class IDGenerator
4   {
5   private:
6      static int s_nextID;
7
8   public:
9      static int getNextID()
10     {
11         return s_nextID++;
12     };
13  };
14
15  // Inicijalizacija statičkog člana
16  int IDGenerator::s_nextID = 1;
17
18  int main()
19  {
20     for (int count = 0; count < 5; ++count)
21        std::cout << "Sledeći ID: "
22                  << IDGenerator::getNextID()
23                  << '\n';
24     return 0;
25  }
```

Program štampa sledeći izlaz:

```
Sledeći ID: 1
Sledeći ID: 2
Sledeći ID: 3
Sledeći ID: 4
Sledeći ID: 5
```

S obzirom da su svi podaci i funkcije u ovoj klasi statički, nije potrebno instancirati objekat klase kako bi se iskoristila njenu funkcionalnost. Klasa koristi statički član da čuva vrednost sledećeg ID-a koji će biti dodeljen, i obezbeđuje statičku funkciju članicu da bi vratila taj ID i uvećala njegovu vrednost za 1.

10. Preklapanje operatora

Uvod u preklapanje operatora

U jeziku C++, princip preklapanja se ne primenjuje samo na funkcije, već i na operatore. To znači da se operatori mogu definisati tako da rade ne samo sa ugrađenim tipovima, već i sa klasama. Programer može da napiše svoj sopstveni operator u nekoj klasi preklapanjem ugrađenog operatora kako bi izvršio određeno računanje kada se operator koristi nad objektima te klase. Preklapanje operatora može biti veoma korisno, jer se njihovim definisanjem može napisati programski kod koji izgleda sasvim prirodno. S druge strane, preklapanje operatora, kao i sve napredne funkcije, čini jezik komplikovanijim. Pored toga, operatori mogu da imaju specifičnu funkcionalnost, a većina programera ne očekuje da operatori obavljaju mnogo posla. Takođe, preklopljeni operatori mogu biti i loše definisani tako da kod postane nečitljiv.

Programer može redefinisati ili preklopiti većinu ugrađenih operatora dostupnih u jeziku C++. Shodno tome, programer može koristiti operatore nad korisnički definisanim tipovima – klasama.

Preklopljeni operatori su funkcije sa posebnim imenima koja se sastoje od ključne reči `operator` i simbola operatora koji treba da bude preklopljen. Kao i svaka druga funkcija, preklopljeni operator ima povratni tip i listu parametara.

Postoji više načina definisanja preklopljenih operatora. Moguće ih je definisati korišćenjem običnih funkcija, ukoliko su članovi klase javni. Drugi način je pomoću `friend` funkcija ukoliko postoje privatni članovi. Treći, i najčešće korišćeni način preklapanja operatora je korišćenjem operatora kao funkcija članica klase.

U okviru ove knjige biće opisan poslednji od navedenih načina definisanja preklopljenih operatora, koji se i najviše slaže sa objektno orijentisanim konceptom programiranja.

Rešavanje preklopljenih operatora

Kada izračunava neki izraz koji sadrži operator, kompajler koristi sledeća pravila:

- Ako su svi operandi osnovni tipovi podataka, kompajler će pozvati ugrađenu rutinu, ukoliko ona postoji. Ukoliko ne postoji, kompajler će prijaviti grešku.
- Ako je neki od operanda korisnički definisan tip podataka (npr. neka klasa), kompajler će pokušati da pronađe odgovarajući preklopljeni operator koji

može pozvati. Ako ne pronađe, pokušaće da konvertuje jedan ili više operanda korisnički definisanih tipova u osnovne tipove podataka tako da može koristiti ugrađene operatore.

Prilikom definisanja preklopljenih operatora postoje izvesna ograničenja.

1. Većina postojećih operatora u jeziku C++ može biti preklopljena. Postoji nekoliko izuzetaka, npr. uslovni operator (`?:`), `sizeof` operator, operator razrešenja opsega (`::`), operator izbora članova (`.`), itd.
2. Mogu biti preklopljeni jedino operatori koji postoje. Nije moguće kreirati nove operatore ili preimenovati postojaće.
3. Minimalno jedan operand u preklopljenom operatoru mora biti korisnički definisan tip. To znači da se ne može preklopiti operator plus koji radi sa celobrojnim promenljivama ili promenljivama sa pokretnim zarezom.
4. Nije moguće promeniti broj operanda koje operator podržava.

Svi operatori čuvaju svoj podrazumevani prioritet i asocijativnost (bez obzira na to za šta se koriste) i to se ne može promeniti.

Kada se preklapaju operatori, najbolje je zadržati funkciju operatora što je moguće bliže prvobitnoj nameni operatora.

Pored toga, pošto operatori nemaju opisna imena, nije uvek jasno šta će oni uraditi. Na primer, operator + bi mogao biti razuman izbor za klasu `String` čijom primenom bi se obavilo spajanje dva stringa. Ali šta bi u slučaju stringa bio operator “?“ ? Nejasno je šta bi ovakav operator radio. Ako značenje operatora, kada se primenjuje na neku klasu, nije jasno i intuitivno, trebalo bi umesto operatora napisati funkciju sa imenom koje bi intuitivno govorilo šta funkcija radi.

Pored ovih ograničenja, i dalje postoji puno korisnih funkcionalnosti koje donose preklopljeni operatori. Recimo, može se preklopiti operator +, koji bi spajao dva stringa ili sabrao dva objekta klase `Fraction` (razlomak). Može se preklopiti operator << kako bi se olakšalo štampanje podataka iz klase na konzoli. Može se preklopiti relacioni operator (==) da bi se uporedila dva objekta iste klase. Sve ovo čini preklopljene operatore jednom od veoma korisnih funkcija u jeziku C++. Oni omogućavaju rad sa klasama na veoma intuitivan način.

U okviru ove knjige biće obrađen veći broj preklopljenih operatora.

Unarni operatori

Unarni operatori, kao što im i samo ime govori, imaju samo jedan operand. Primeri unarnih operatora su:

- Unarni minus -,
- Unarni plus +,
- Operator inkrementiranja ++ (prefiksni ili postfiksni),
- Operator dekrementiranja -- (prefiksni ili postfiksni),
- Logički operator negacije !

Unarni operatori funkcionišu nad objektom za koji su pozvani. Neki od njih se pojavljuje na levoj strani objekta, kao što su !, -, prefiksni operator inkrementiranja ili dekrementiranja (++, --), ali nekada se mogu koristiti kao postfiksni, kao sto su postfiksni operatori ++ i --.

Unarni operator –

U sledećem primeru je pokazano kako unarni operator minus (-) može biti preklopljen kao prefiksni operator.

```
class Complex
{
private:
   double real;
   double imag;
public:
   Complex(){}
   Complex(double realPart, double imagPart)
   {
      real = realPart;
      imag = imagPart;
   }

   Complex operator-()
   {
      return Complex(-real, -imag);
   }

   Complex operator+()
   {
      return Complex(real, imag);
   }
};
```

```
1   int main()
2   {
3      Complex n1(1,-2);
4      Complex n3 = -n1;
5   }
```

Preklopljene operatore je najjednostavnije razumeti upoređivanjem njihove uloge u klasi kojoj pripadaju sa njihovom ulogom kod prostih tipova podataka.

```
1   int a = 5;
2   int b = -a;
```

Prvi iskaz će definisati promenljivu a i dodeliti joj vrednost 5, dok će drugi iskaz definisati promenljivu b i dodeliti joj vrednost promenljive a ali sa suprotnim znakom. Za ovakav unarni operator je karakteristično da uvek dolazi sa leve strane operanda na koji se primenjuje (broj a u ovom slučaju). Ukoliko je potrebno ovakav operator primeniti nad kompleksnim brojem, tako da njegovi realni i imaginarni deo dobiju vrednosti od nekog drugog kompleksnog broja, ali suprotnih znakova, neophodno je za to definisati poseban unarni operator -. Takav operator se takođe poziva iz objekta i nalazi se sa njegove leve strane (linija 31 u prethodnom primeru).

Sama definicija ovakvog operatora je data unutar klase `Complex`. Ovakav unarni operator nema argument jer je on unarni, i radi samo nad podacima objekta iz koga je pozvan. On zapravo treba da generiše novi objekat sa vrednostima realnog i imaginarnog dela suprotnog znaka od znaka koji imaju u objektu iz koga se operator poziva, i da ga zatim (kao i obična funkcija) vrati po vrednosti. Dakle, ako se uporedi sa unarnim operatorom primenjenim nad celobrojnim vrednostima, može se napraviti sledeća analogija:

```
1   int a = 5;
2   int b = -a;
3
4   Complex z1(1,-2);
5   Complex z2 = -z1;
```

Broj b će dobiti vrednost broja a sa suprotnim znakom, a realan i imaginaran deo kompleksnog broja z2 će dobiti vrednosti realnog i imaginarnog dela kompleksnog broja z1, ali sa suprotnim znakovima.

Unarni operator +

Unarni operator + takođe može biti preklopljen. Kao što je ranije napomenuto, pri radu sa ugrađenim tipovima podataka u jeziku C++ on nema naročit značaj, jer je

dodat uglavnom zbog simetrije sa unarnim operatorom -. Prema tome, ako se napravi analogija sa unarnim operatorom - nad kompleksnim brojevima, takav operator bi recimo mogao da radi analognu stvar kao i unarni operator + nad ugrađenim tipovima – odnosno da ne menja ništa. Evo i primera:

```
1   int a = 5;
2   int b = +a;   // Izjednačavanje celih brojeva b i a
3
4   Complex z1(1,-2);
5   Complex z2 = +z1; // Izjednačavanje kompleksnih brojeva z2 i z1
```

Pritom, unarni operator + bi izgledao onako kako je definisan u klasi `Complex` u liniji 19. Unarni operator + može biti definisan da radi i drugačije, ali onda intuitivno značenje takvog operatora nije usklađeno sa onim što radi, i bespotrebno komplikuje program.

Unarni operator inkrementiranja ++

Unarni operatori inkrementiranja (++) i dekrementiranja (--) su takođe dva važna operatora dostupna u jeziku C++.

U sledećem primeru je objašnjeno kako operator inkrementiranja (++) može biti preklopljen kao prefiksni ili kao postfiksni. Slično se može preklopiti i operator dekrementiranja (--).

Jedina razlika u deklaraciji prefiksnog i postfiksnog operatora inkrementiranja je ključna reč `int` unutar zagrada koje predstavljaju listu argumenata operatora. To u slučaju ovog operatora ne znači da ima celobrojni argument već samo da se takav operator primenjuje kao postfiksni, odnosno da dolazi sa desne strane objekta na koji se odnosi.

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

class Time
{
private:
   int hours;             // od 0 do 23
   int minutes;           // od 0 do 59
public:
   // Podrazumevani konstruktor
   Time()
   {
      hours = 0;
      minutes = 0;
   }
   // Konstruktor sa dva parametra
   Time(int h, int m)
   {
      hours = h;
      minutes = m;
   }
   // Funkcija članica za prikaz vremena na ekranu
   void displayTime()
   {
      cout << "H: " << hours << " M:" << minutes <<endl;
   }
   // preklopljeni operator ++ (prefiksni)
   Time operator++()
   {
      ++minutes;          // uvećavanje minuta tekućeg objekta
      if(minutes >= 60)
      {
         ++hours;
         minutes -= 60;
      }
      return Time(hours, minutes);
   }
   // preklopljeni operator ++ (postfiksni)
   Time operator++( int )
   {
      Time T(hours, minutes); // čuvanje originalnog stanja objekta

      // Uvećavanje minuta tekućeg objekta
      ++minutes;

      if(minutes >= 60) // Ažuriranje sati – ako je potrebno
      {
         ++hours;
         minutes -= 60;
      }
      return T; // vraćanje originalnog stanja objekta
   }
};
```

```
int main()
{
   Time T1(11, 59), T2(10,40);

   ++T1;               // Uvećavanje T1
   T1.displayTime();   // Prikaz T1
   ++T1;               // Još jedno uvećavanje T1
   T1.displayTime();   // Prikaz T1

   T2++;               // Uvećavanje T2
   T2.displayTime();   // Prikaz T2
   T2++;               // Još jedno uvećavanje T2
   T2.displayTime();   // Prikaz T2

   T1 = T2++;          // Izjednačavanje T2 sa T1 i uvećavanje T2
   T1.displayTime();   // Prikaz T1
   T2.displayTime();   // Prikaz T2
   T1 = ++T2;          // Uvećavanje T2 i te dodela te vrednosti T1
   T1.displayTime();   // Prikaz T1
   T2.displayTime();   // Prikaz T2

   return 0;
}
```

Iz prikazanog primera se može videti kako rade prefiksna i postfiksna verzija preklopljenog operatora.

Prefiksna verzija uvećava broj minuta za 1 u objektu iz koga je pozvan, proverava i radi ažuriranje broja sati ukoliko je potrebno, a zatim vraća kao vrednost objekat sa ažuriranim vrednostima (što je takođe moguće uraditi i pomoću pokazivača `this`, o čemu će biti reči kasnije).

Postfiksna verzija čuva trenutno stanje objekta u posebnom objektu, menja stanje objekta na koji se odnosi, a vraća po vrednosti pređašnje stanje. Rezultat ovakvog načina rada ove dve verzije operatora se vidi u izlazu koji štampa funkcija `main()`:

```
H: 12 M:0
H: 12 M:1
H: 10 M:41
H: 10 M:42
H: 10 M:42
H: 10 M:43
H: 10 M:44
H: 10 M:44
```

Unarni operator dekrementiranja `--`

Slično prethodnom, mogu se definisati i unarni operatori dekrementiranja (prefiksni i postfiksni) unutar klase:

```
   // Preklopljeni operator – (prefiksni)
   Time operator--()
   {
      --minutes;          // umanjivanje minuta tekućeg objekta
      if(minutes < 0)
      {
         --hours;
         minutes += 60;
      }
      return Time(hours, minutes);
   }
```

```
   // Preklopljeni operator – (postfiksni)
   Time operator--( int )
   {
      // čuvanje originalnog stanja objekta
      Time T(hours, minutes);

      // Umanjivanje minuta tekućeg objekta
      --minutes;

      if(minutes < 0)
      {
         --hours;
         minutes += 60;
      }

      // vraćanje originalnog stanja objekta
      return T;
   }
```

Sledeći program ilustruje rad operatora dekrementiranja.

```
int main()
{
   Time T1(23,0);
   Time T2 = T1--;      // Izjednačavanje T2 sa T1 i umanjivanje T2
   T1.displayTime();    // Prikaz T1
   T2.displayTime();    // Prikaz T2
   Time T3 = --T1;      // Umanjivanje T1 i dodela te vrednosti T3
   T1.displayTime();    // Prikaz T1
   T3.displayTime();    // Prikaz T3
   return 0;
}
```

Izlaz iz prethodnog programa će biti:

```
H: 22 M:59
H: 23 M:0
H: 22 M:58
H: 22 M:58
```

Unarni operator !

Iz ranijih primera se moglo videti kako funkcioniše operator logičke negacije: za svaki iskaz koji je različit od 0 primenom ovog operatora dobija se logička vrednost `false`, a u suprotnom `true`. Ovakav operator takođe je moguće preklopiti.

U primeru koji sledi, definisana je klasa koja reprezentuje stanje nekog uređaja. Stanje uređaja koji je uključen može se smatrati "tačnim" stanjem, a stanje isključenog uređaja "netačnim". Za takvu klasu se može napisati operator ! koji bi u ovom slučaju ispitivao stanje uređaja, da li je uključen ili isključen. Dakle, ovakav operator, pozvan nad određenim objektom, vratio bi vrednost `true` ili `false` zavisno od toga da li je stanje uređaja uključeno ili isključeno.

```
1   #include <iostream>
2   using namespace std;
3
4   class DeviceState
5   {
6   public:
7      enum State
8      {
9         STATE_TURNED_OFF,
10        STATE_TURNED_ON
11     };
12  private:
13     State m_State;
14  public:
15     DeviceState(State s)
16     {
17        m_State = s;
18     }
19     bool operator!()
20     {
21        if (m_State == State::STATE_TURNED_OFF) return true;
22        else                                    return false;
23     }
24     void TurnOn()
25     {
26        m_State = State::STATE_TURNED_ON;
27     }
28     void TurnOff()
29     {
30        m_State = State::STATE_TURNED_OFF;
31     }
32  };
```

```
int main()
{
   DeviceState deviceState(DeviceState::State::STATE_TURNED_OFF);

   if (!deviceState)
   {
      cout << "The device is turned off!" << endl;
   }
   else
   {
      cout << "The device is turned on!" << endl;
   }
}
```

Binarni operatori

Binarni operatori su operatori koji imaju dva operanda. Primeri binarnih operatora su aritmetički operatori sabiranja (+), oduzimanja (-), množenja (*) i deljenja (/). Pored ovih osnovnih binarnih operatora postoji veći broj takođe binarnih operatora, kao što su relacioni operatori (<, >, <=, >=, ==, !=), sabiranje i dodela vrednosti (+=), oduzimanje i dodela vrednosti (-=), itd.

Binarni aritmetički operatori

Način implementacije aritmetičkih operatora biće prikazan na primeru kompleksnih brojeva. U sledećem primeru prikazana je klasa koja predstavlja kompleksne brojeve i ima implementirane osnovne aritmetičke operatore.

```
class Complex
{
private:
   double m_real, m_imag;
public:
   Complex(){}
   Complex(double realPart, double imagPart)
   {
      m_real = realPart;   m_imag = imagPart;
   }
   Complex operator+(const Complex& other)
   {
      double real = m_real + other.m_real;
      double imag = m_imag + other.m_imag;
      return Complex(real, imag);
   }
   Complex operator-(const Complex& other)
   {
      double real = m_real - other.m_real;
      double imag = m_imag - other.m_imag;
      return Complex(real, imag);
   }
   Complex operator*(const Complex& other)
   {
      double real = m_real * other.m_real - m_imag * other.m_imag;
      double imag = m_real * other.m_imag + m_imag * other.m_real;
      return Complex(real, imag);
   }
   Complex operator/(const Complex& other)
   {
      double denominator = other.m_real*other.m_real +
                           other.m_imag*other.m_imag;
      double real = m_real * other.m_real + m_imag * other.m_imag;
      double imag = m_imag * other.m_real - m_real * other.m_imag;
      return Complex(real/denominator, imag/denominator);
   }
   void operator=(Complex& other)
   {
      m_real = other.m_real;
      m_imag = other.m_imag;
   }
   void Print()
   {
      cout << "real: " << m_real << " imag: " << m_imag <<endl;
   }
};
```

```
#include <iostream>
#include "complex.h"

using namespace std;

int main()
{
   Complex n1( 1,-2);
   Complex n2(-3, 4);
   Complex n3 = n1+n2;
   Complex n4 = n1-n2;
   Complex n5 = n1*n2;
   Complex n6 = n1/n2;

   n3.Print();
   n4.Print();
   n5.Print();
   n6.Print();
}
```

Operatori sabiranja i oduzimanja rade veoma slično. Može se napraviti potpuna analogija sa sabiranjem i oduzimanjem ugrađenih tipova, kao što je recimo celobrojni tip.

```
   int a = 1;
   int b = 2;
   int c = a + b;

   Complex z1(1,2);
   Complex z2(3,4);
   Complex z3 = z1 + z2;
```

Ako se celobrojne promenljive a, b i c posmatraju kao objekti, operacije nad njima se mogu interpretirati na sledeći način: objektu c se dodeljuje vrednost zbira objekata a i b. Dakle, objekat c bi trebalo da pozove svoj operator = kako bi sebi dodelio vrednost. Sa druge strane, iz objekta a se poziva operator sabiranja (+) i njemu prosleđuje objekat b. Operator + objekta a vraća vrednost zbira sebe i drugog sabirka (tj. objekta b). Tu povratnu vrednost prihvata operator = objekta c.

Analogno prethodnom, mogu se implementirati operatori klase za komplekse brojeve, tako da rade po istoj logici kao i kod ugrađenih tipova, s tim da se mora napisati kod za takve operatore, koji će biti realizovani korišćenjem matematičkih definicija operacija nad kompleksnim brojevima.

Iz deklaracije operatora + definisanog u klasi `Complex` se vidi da je njegov parametar konstantna referenca na drugi objekat tipa `Complex`, a povratna vrednost takođe objekat tipa `Complex`. Ovo je potpuno analogno operatoru sabiranja celobrojnih promenljivih. Dakle, ako se iz nekog objekta – kompleksnog

broja pozove ovaj operator (u ovom slučaju objekat z1), njemu je potrebno proslediti drugi sabirak (objekat z2), takođe tipa Complex, a operator će kao povratnu vrednost vratiti zbir ova dva broja, koji je takođe tipa Complex. Unutar definicije operatora + se može videti kako se generiše novi kompleksan broj koji predstavlja zbir: njegov realni deo jednak je zbiru realnog dela prvog sabirka, odnosno objekta iz koga je operator pozvan (z1), i realnog dela drugog sabirka, odnosno objekta koji je operatoru prosleđen kao argument (z2). Isto važi i za imaginarni deo. Rezultat je novi kompleksan broj, generisan od prethodno izračunatog realnog i imaginarnog dela zbira.

Novi objekat koji je vratio operator + pozvan iz objekta z1 predstavlja zbir objekata z1 i z2. Taj objekat je sada argument operatora = pozvanog iz objekta z3. Operator = izjednačava članove objekta iz koga je pozvan (objekat z3) sa članovima objekta koji mu je prosleđen kao argument (objekat koji predstavlja rezultat sabiranja objekata z1 i z2, odnosno objekat koji je vratio operator + pozvan iz objekta z1). Njegova povratna vrednost u ovom slučaju nije potrebna, pa stoga operator = ne vraća nikakvu vrednost. Međutim, često postoje situacije kada operator = ipak treba da vrati neku vrednost. Sledi primer analogije sa prostim tipovima podataka i njihovim operatorom =.

```
1    int a = 1, b, c;
2    c = b = a;
3
4    Complex z1(1,2), z2, z3;
5    z3 = z2 = z1;
```

Izraz dat u liniji 2 je sasvim regularan. Vrednost objekta a je prvo dodeljena objektu b, a zatim je njegova vrednost dodeljena objektu c. Dakle, operator = pozvan iz objekta b **vraća svoju vrednost** i prosleđuje je objektu c iz koga je još jednom pozvan operator =.

Posmatrajući izraz u liniji 5, uz analogiju sa prethodno rečenim, uočava se nedostatak prethodno definisanog operatora =. Gledajući izraz s desna na levo, objekat z1 će se naći u ulozi argumenta operatora = pozvanog iz objekta z2. Objekat z2 će dobiti vrednost objekta z1, ali povratni tip operatora = pozvanog iz objekta z2 je void! To znači da će argument koji prihvata operator = pozvan iz objekta z3 biti void, što je neizvodljivo. Iz tog razloga, operator = bi trebalo da **vrati objekat iz koga je pozvan**, odnosno da vrati samog sebe.

```
class Complex
{
private:
   double m_real;
   double m_imag;
public:
   Complex(){}
   Complex(double realPart, double imagPart)
   {
      m_real = realPart;
      m_imag = imagPart;
   }
   Complex operator=(Complex& other)
   {
      m_real = other.m_real;
      m_imag = other.m_imag;
      return Complex(m_real, m_imag);
   }
};
```

Ovako definisan operator bi bio sposoban da izvrši iskaz z3=z2=z1. Ipak, i ovakav operator ima jedan manji nedostatak, a to je bespotrebno generisanje kopija jednog istog objekta. Zato je bolje da operator = vrati objekat dobijen dereferenciranjem pokazivača na samog sebe (this), a da povratni tip bude referenca na takav objekat.

```
class Complex
{
public:
   Complex& operator=(Complex& other)
   {
      m_real = other.m_real;
      m_imag = other.m_imag;
      return *this;
   }
};
```

Analogno logici operatora + rade i preostali aritmetički operatori: -, *, / tako da nije neophodno posebno objašnjavati način njihovog rada.

Binarni relacioni operatori

Postoji nekoliko različitih relacionih operatora koje podržava jezik C++: <, >, <=, >=, ==, !=. Ovi operatori se mogu koristiti za upoređivanje ugrađenih tipova u jeziku C++.

Bilo koji od ovih operatora može biti preklopljen u korisnički definisanim klasama, tako da se mogu koristiti za upoređivanje objekata iste klase.

U sledećem primeru je objašnjeno kako se može preklopiti operator > unutar neke klase, a na sličan način je moguće preklopiti i bilo koji drugi relacioni operator. Karakteristično za sve relacione operatore je da vraćaju vrednost tipa `bool`.

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

class Money
{
private:
   int m_OneEuroCoins;
   int m_TwoEuroCoins;
   int m_FiveEuroBanknotes;
   int m_TenEuroBanknotes;
public:
   Money(){}
   Money(int oneEuro=0,int twoEuro=0,int fiveEuro=0,int tenEuro=0)
   {
      m_OneEuroCoins = oneEuro;
      m_TwoEuroCoins = twoEuro;
      m_FiveEuroBanknotes = fiveEuro;
      m_TenEuroBanknotes = tenEuro;
   }

   Money operator+(const Money& other)
   {
      return Money(m_OneEuroCoins      + other.m_OneEuroCoins,
                   m_TwoEuroCoins      + other.m_TwoEuroCoins,
                   m_FiveEuroBanknotes + other.m_FiveEuroBanknotes,
                   m_TenEuroBanknotes  + other.m_TenEuroBanknotes);
   }

   bool operator>(const Money& other)
   {
      int sum = 1 * m_OneEuroCoins + 2 * m_TwoEuroCoins +
                5 * m_FiveEuroBanknotes + 10 * m_TenEuroBanknotes;
      int sumOther = 1 * other.m_OneEuroCoins +
                     2 * other.m_TwoEuroCoins +
                     5 * other.m_FiveEuroBanknotes +
                    10 * other.m_TenEuroBanknotes;
      return sum > sumOther;
   }

   Money& operator=(Money& other)
   {
      m_OneEuroCoins = other.m_OneEuroCoins;
      m_TwoEuroCoins = other.m_TwoEuroCoins;
      m_FiveEuroBanknotes = other.m_FiveEuroBanknotes;
      m_TenEuroBanknotes = other.m_TenEuroBanknotes;
      return *this;
   }
};
```

```
1   int main()
2   {
3      Money amount1(3,2,5,0);
4      Money amount2(4,0,2,10);
5
6      if (amount1 > amount2)
7      {
8         cout << "amount1 is greater then amount2" << endl;
9      }
10     else
11     {
12        cout << "amount1 is less then amount2" << endl;
13     }
14  }
```

U prethodnom primeru definisana su dva objekta tipa Money, odnosno dve gomile novca, u apoenima od 1, 2, 5 i 10 evra. Pravljenjem preklopljenog operatora > (ili bilo kog drugog relacionog operatora) omogućava se direktno upoređivanje dva objekta tipa Money, kako bi se dobila informacija o tome u kom objektu (odnosno na kojoj gomili) ima više novca. Kao što se može videti, preklopljeni operator > računa sumu novca objekta iz koga je pozvan i sumu novca drugog objekta sa kojim se upoređuje. Rezultat koji operator vraća je logička vrednost `true` ili `false`.

Takođe, u prethodnom primeru je definisan i operator +, tako da je moguće pisati izraze kao što je navedeno u liniji 7 u sledećem programu:

```
1   int main()
2   {
3      Money amount1(3,2,5,0);
4      Money amount2(4,0,2,10);
5      Money amount3(4,0,2,30);
6
7      if ((amount1 + amount2) > amount3)
8      {
9         cout << "amount1 + amount2 > amount3" << endl;
10     }
11     else
12     {
13        cout << "amount1 + amount2 < amount3" << endl;
14     }
15  }
```

Na gotovo istovetan način se mogu definisati i ostali relacioni operatori za ovakvu klasu (<, <=, >=, !=, ==).

Indeksni operator [ ]

Indeksni operator [ ] se obično koristi za pristup elementima nizova. Ovaj operator može biti preklopljen kako bi se poboljšala funkcionalnost klasa u jeziku C++ koje predstavljaju dinamičke nizove.

Sledi primer preklopljenog indeksnog operatora.

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

class Array
{
private:
   int* m_pArray;
   int m_nSize;

public:
   Array()
   {
      m_nSize = 0;
      m_pArray = 0;
   }
   Array(int size)
   {
      if (size > 0)
      {
         m_nSize = size;
         m_pArray = new int[size];
      }
   }
   ~Array()
   {
      if (m_pArray)
      {
         delete [] m_pArray;
      }
   }
   int GetSize() { return m_nSize; }

   int& operator[](int i)
   {
      if( i > m_nSize )
      {
         cout << "Index out of bounds" << endl;
         return m_pArray[0]; // vraća početni član niza.
      }
      return m_pArray[i];
   }
};
```

```
1   int main()
2   {
3      Array A(10);
4      for (int i = 0; i < A.GetSize(); i++) A[i] = i+1;
5
6      cout << "Value of A[2]  : " << A[2]  <<endl;
7      cout << "Value of A[5]  : " << A[5]  <<endl;
8      cout << "Value of A[12] : " << A[12] <<endl;
9      return 0;
10  }
```

U prethodnom primeru je definisana klasa koja sadrži dva člana: pokazivač na celobrojni tip, koji se koristi za dinamičko alociranje memorije i jednu celobrojnu promenljivu koja čuva podatak o veličini alociranog niza. U konstruktoru klase se alocira memorija za određeni broj članova koje će klasa čuvati, a takođe i postavlja vrednost promenljive koja čuva podatak o veličini niza.

Novina u ovoj klasi u odnosu na ono što je do sada viđeno je preklopljeni indeksni operator [ ]. Ovaj operator ima jedan parametar koji predstavlja indeks člana "niza" kome se pristupa. Ako se napravi analogija sa običnim nizom fiksne dimenzije, to bi izgledalo ovako:

```
1      int a[10];
2      a[0] = 5;
3
4      Array A(10);
5      A[0] = 5;
```

Ako je potrebno pristupiti nultom članu niza a, to se može jednostavno postići kao u liniji 2. Takođe, ako je potrebno sličnu stvar uraditi sa objektom A klase `Array`, koja predstavlja dinamički niz, moguće je preklopiti njen indeksni operator, tako da bude omogućena sintaksa kao u liniji 5. Dakle, potpuno isto kao i kod niza fiksnih dimenzija. Da bi to bilo moguće, potrebno je da preklopljeni indeksni operator ima parametar koji predstavlja indeks člana "niza" kome se pristupa, a povratna vrednost je referenca na taj član. Neophodno je da bude referenca, kako bi bilo omogućeno menjanje člana niza, a ne njegove kopije koja bi bila vraćena iz operatora u slučaju da povratni tip operatora nije referentni tip, već običan tip (u ovom slučaju `int`). Međutim, ovde postoji i jedan problem koji je rešen na loš način. U slučaju da korisnik upotrebom indeksnog operatora zahteva član niza sa indeksom koji nije unutar granica niza, biće vraćena referenca na početni član niza! Ovo bi trebalo rešiti izbacivanjem izuzetaka koji će biti obrađeni u poglavlju 15.

11. Nasleđivanje

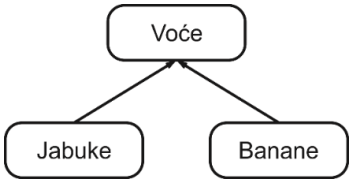
Uvod u nasleđivanje

Prilikom definisanja klasa moguće je objekte neke klase smestiti unutar definicije neke druge klase. Primer za ovo bi bila klasa Automobil, koja unutar sebe sadrži četiri objekta tipa Točak. Ovo je jedan od načina kako je moguće implementirati neku složenu klasu. Međutim, složenu klasu je moguće projektovati i na drugi način, tako da klasa nasleđuje podatke i funkcionalnost neke jednostavnije klase. Ovaj način projektovanja složenih klasa se naziva nasleđivanje, i predstavlja jedan od najvažnijih elemenata koncepta objektno orijentisanog programiranja.

Nasleđivanje podrazumeva stvaranje novih objekata direktnim nasleđivanjem atributa i ponašanja drugih objekata, a zatim njihovo proširivanje i specijalizaciju. Primeri nasleđivanja se mogu videti svuda u stvarnom životu. Pođimo od sebe, nasledili smo gene svojih roditelja i dobili fizičke atribute od oboje - ali je onda do izražaja došla naša ličnost. Tehnološki proizvodi (računari, mobilni telefoni itd.) nasleđuju karakteristike svojih prethodnika. Na primer, Intel Pentium procesor je nasledio mnoge funkcije koje je imao procesor Intel 486, koji je i sam nasledio funkcije ranijih procesora. Jezik C++ je nasledio mnoge osobine jezika C, na kojem je baziran, a jezik C je nasledio mnoge funkcionalnosti programskih jezika koji su se pojavili pre njega.

Analiziranjem različitih biljnih ili životinjskih vrsta mogu se uočiti neke njihove sličnosti. Na primer, iako su jabuke i banane različite vrste voća, za obe vrste je zajedničko da su voće. A pošto su i jabuke i banane voće, jednostavna logika govori da sve što se odnosi na voće važi i za jabuke i za banane. Na primer, sve vrste voća imaju ime, boju i veličinu. Dakle, i jabuke i banane takođe imaju ime, boju i veličinu. Može se reći da jabuke i banane nasleđuju sva svojstva voća jer su i same voće. Takođe, poznato je da voće prolazi kroz proces sazrevanja, čime postaje jestivo. Pošto su jabuke i banane voće, i jabuke i banane će naslediti “funkciju” sazrevanja.

Odnos između jabuka, banana i voća bi izgledao ovako:

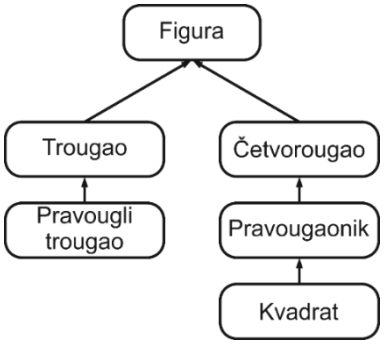


Slika 4: Dijagram hijerarhije

Hijerarhija

Hijerarhija predstavlja dijagram koji pokazuje kako su različiti objekti povezani. Hijerarhija najčešće pokazuje progresiju tokom vremena (386 -> 486 -> Pentium) ili kategorizuje stvari na način koji se ide od opšteg ka specifičnom (voće -> jabuka -> crvena ukusna).

Evo još jednog primera hijerarhije: kvadrat je vrsta pravougaonika, koji je četvorougao, a koji takođe predstavlja geometrijsku figuru. Jednakostranični trougao je takođe trougao, a takođe je i geometrijska figura. Dijagram hijerarhije za ovaj primer bi izgledalo ovako:



Slika 5: Hijerarhija geometrijskih figura

Ovaj dijagram ide od opšteg (vrh) ka specifičnom (dno), pri čemu svaka stavka u hijerarhiji nasleđuje osobine i ponašanja stavke iznad nje.

U ovom poglavlju biće obrađene osnove nasleđivanja u jeziku C++.

Osnove nasleđivanja u jeziku C++

Nasleđivanje u jeziku C++ se dešava na nivou klasa. Kada se govori o nasleđivanju, klasa koja je nasleđena naziva se roditeljska klasa, osnovna klasa ili nadklasa, a klasa koja je dobijena nasleđivanjem se naziva dete – klasa, izvedena klasa ili podklasa.

Dakle, u prethodnim dijagramima hijerarhije:

- Voće je roditelj, a jabuke i banane su deca.
- Trougao je dete (roditelja Figura) i roditelj (Pravouglog trougla)

Izvedena klasa nasleđuje i ponašanja (funkcije članice) i svojstva (promenljive – članove) od roditelja.

Promenljive – članovi i funkcije članice postaju članovi izvedene klase.

Pošto su izvedene klase apsolutno ravnopravne baznim klasama, one mogu imati svoje članove koji su specifični za novu izvedenu klasu.

Sledi primer jednostavne klase koja predstavlja osobu.

```
#include <string>

class Person
{
public:
   std::string m_name;
   int m_age;

   Person(std::string name = "", int age = 0)
      : m_name(name), m_age(age)
   {
   }

   std::string getName() const { return m_name; }
   int getAge() const { return m_age; }
};
```

S obzirom da je ova klasa dizajnirana da predstavlja generičku osobu, definisani su samo članovi koji bi bili uobičajeni za bilo koju osobu. Svaka osoba (bez obzira na pol, profesiju, itd.) ima ime i godine starosti, pa su iz tog razloga u klasu smešteni podaci koji mogu sačuvati takve informacije.

U prethodnom primeru sve promenljive i funkcije su proglašene javnim. Ovo je urađeno samo zbog jednostavnosti pisanja. Uobičajeno je da promenljive budu privatne i da za njih postoje funkcije pristupa, koje su ranije već objašnjene.

Recimo da se javlja potreba za programom koji čuva informacije o košarkašima. Košarkaši moraju imati informacije koje su specifične za njih – na primer, možda treba čuvati broj trojki, ili ukupan broj poena u toku utakmice.

Sledi primer klase koja bi predstavljala košarkaša:

```
class BasketballPlayer
{
public:
   int m_ThreePoints;
   int m_TotalScore;

   BasketballPlayer(int threePoints = 0, int totalScore = 0)
      : m_ThreePoints(threePoints), m_TotalScore(totalScore)
   {
   }
};
```

Međutim, takođe je potrebno čuvati ime i starost košarkaša, a te informacije već postoje kao članovi klase `Person`. Postoje tri načina kako se može dodati ime i starost košarkaša:

1) Dodavanjem novih članova u klasu `BasketballPlayer`: ime i starost. Ovo je verovatno najgori izbor, jer se duplira kod koji već postoji u klasi `Person`. Ako se javi potreba za bilo kakvim ažuriranjem klase `Person`, verovatno će se pojaviti i u klasi `BasketballPlayer`.

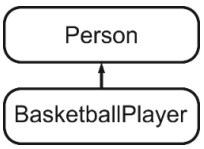
2) Dodavanjem objekta klase `Person` kao član `BasketballPlayer` klase. Ni ovo nije adekvatno rešenje, jer bi podrazumevalo da košarkaš ima osobu, kao što ima starost ili godine.

3) Nasleđivanjem atributa klase `Person` u klasi `BasketballPlayer`. Ovo je najlogičniji izbor, s obzirom da je `BasketballPlayer` takođe osoba.

Sintaksa nasleđivanja je prilično jednostavna. Nakon deklaracije klase `BasketballPlayer` sledi simbol “:”, ključna reč `public` i ime klase koju treba naslediti. Ovakvo nasleđivanje se zove javno nasleđivanje. O različitim tipovima nasleđivanja će biti više reči kasnije.

```
class BasketballPlayer : public Person
{
public:
   int m_ThreePoints;
   int m_TotalScore;

   BasketballPlayer(int threePoints = 0, int totalScore = 0)
      : m_ThreePoints(threePoints), m_TotalScore(totalScore)
   {
   }
};
```

Korišćenjem klasnog dijagrama, nasleđivanje bi izgledalo ovako:



Slika 6: Hijerarhija klasa

Kada klasa `BasketballPlayer` nasleđuje klasu `Person`, klasa `BasketballPlayer` nasleđuje funkcije članice i promenljive – članove iz klase `Person`. Pored toga, u klasi `BasketballPlayer` su definisana dva nova člana: `m_ThreePoints` i `m_TotalScore`. Ovo ima smisla, jer su ova svojstva specifična za klasu `BasketballPlayer`, a ne za bilo koju osobu (odnosno klasu `Person`).

Tako će objekti klase `BasketballPlayer` imati četiri člana: `m_ThreePoints` i `m_TotalScore` definisanih unutar klase `BasketballPlayer`, i `m_name` i `m_age` iz klase `Person`.

Ovo je veoma lako dokazati:

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

class Person
{
public:
   std::string m_name;
   int m_age;

   Person(std::string name = "", int age = 0)
      : m_name(name), m_age(age)
   {
   }

   std::string getName() const { return m_name; }
   int getAge() const { return m_age; }
};

// Klasa BasketballPlayer javno nasleđuje klasu Person
class BasketballPlayer : public Person
{
public:
   int m_ThreePoints;
   int m_TotalScore;

   BasketballPlayer(int threePoints = 0, int totalScore = 0)
     : m_ThreePoints(threePoints), m_TotalScore(totalScore)
   {
   }
};

```

```
32  int main()
33  {
34     BasketballPlayer jokic;
35     jokic.m_name = "Nikola";
36     std::cout << jokic.getName() << '\n';
37
38     return 0;
39  }
40
```

Prethodni kod će štampati:

```
Nikola
```

Ovakav program se može kompajlirati zato što je `jokic` objekat klase `BasketballPlayer`, a svi objekti klase `BasketballPlayer` imaju promenljivu – člana `m_name` i funkciju članicu `getName()`, nasleđene iz klase `Person`.

Ulančano nasleđivanje

Moguće je naslediti klasu koja je i sama izvedena iz druge klase. Logika je sasvim analogna prethodnom.

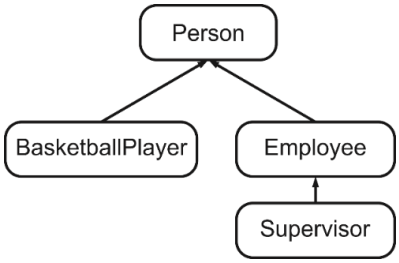
Primer za ovakvo nasleđivanje bi bila klasa `Employee` (zaposleni), koja je izvedena iz klase `Person`.

```
1   class Employee: public Person
2   {
3   public:
4      double m_hourlySalary;
5      long m_employeeID;
6
7      Employee(double hourlySalary = 0.0, long employeeID = 0)
8         : m_hourlySalary(hourlySalary), m_employeeID(employeeID)
9      {
10     }
11
12     void printNameAndSalary() const
13     {
14        std::cout << m_name << ": " << m_hourlySalary << '\n';
15     }
16  };
```

Sada se može, na primer, definisati klasa `Supervisor`. `Supervisor` je takođe zaposleni, a istovremeno je i osoba. Klasa `Employee` već postoji, pa se ona može koristiti kao osnovna klasa iz koje je izvedena klasa `Supervisor`:

```
class Supervisor: public Employee
{
public:
   long m_overseesIDs[5];

   Supervisor()
   {
   }
};

```

Sada klasni dijagram izgleda kao na slici Slika 7.



Slika 7: Klasni dijagram

Svi objekti klase `Supervisor` nasleđuju funkcije i promenljive iz klasa `Employee` i `Person` i dodaju svoju vlastitu promenljivu `m_nOverseesIDs`.

Konstrukcijom ovakvih lanaca nasleđivanja može se kreirati skup upotrebljivih klasa koje su vrlo opšte (na vrhu) i postaju progresivno specifičnije na svakom nivou nasleđivanja.

Zašto je ova vrsta nasleđivanja korisna?

Nasleđivanje od osnovne klase omogućava da se ne moraju redefinisati informacije iz osnovne klase u izvedenim klasama. Nasleđivanjem se dobijaju funkcije članice i promenljive - članovi bazne klase, a zatim se jednostavno dodaju dodatne funkcije ili promenljive koje su potrebne. Ovo ne samo da štedi rad, već takođe znači da ako se ikada ažurira ili modifikuje osnovna klasa (npr. dodaju se nove funkcije ili se ispravi neka greška), sve izvedene klase će automatski naslediti promene.

Na primer, ako se ikada doda nova funkcija klasi `Person`, klase `Employee` i `Supervisor` bi automatski dobile pristup toj funkciji. Ako se doda nova promenljiva klasi `Employee`, klasa `Supervisor` bi takođe dobila pristup toj promenljivoj. Ovo omogućava konstruisanje novih klasa na jednostavan, intuitivan način i njihovo jednostavno održavanje.

Redosled konstruisanja izvedenih klasa

Sada će biti prikazan redosled pozivanja konstruktora koji se dešava kada se izvedena klasa instancira.

Za početak će biti definisane klase koji će pomoći u ilustraciji veoma značajnih karakteristika klasa.

U primeru koji sledi, klasa `Derived` je izvedena iz klase `Base`. S obzirom da klasa `Derived` nasleđuje funkcije i promenljive iz klase `Base`, na prvi pogled može izgledati da su članovi klase `Base` kopirani u `Derived`. Međutim, to nije tako. Klasa `Derived` se može posmatrati kao da sadrži dva dela: jedan deo je `Derived` a drugi deo je `Base`.

```
1   class Base
2   {
3   public:
4      int m_id;
5
6      Base(int id=0)
7         : m_id(id)
8      {
9      }
10
11     int getId() const { return m_id; }
12  };
13
14  class Derived: public Base
15  {
16  public:
17     double m_cost;
18
19     Derived(double cost=0.0)
20        : m_cost(cost)
21     {
22     }
23
24     double getCost() const { return m_cost; }
25  };
```

Već je u nekoliko primera pokazano šta se događa prilikom instanciranja obične (neizvedene) klase:

```
1   int main()
2   {
3      Base base;
4      return 0;
5   }
```

`Base` je klasa koja nije izvedena jer ne nasleđuje ni jednu drugu klasu. Jezik C++ alocira memoriju za objekat klase `Base`, a zatim poziva podrazumevani konstruktor klase `Base` da obavi inicijalizaciju.

Na prvi pogled, instanciranje objekata izvedenih klasa izgleda identično prethodnom.

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

Međutim, stvari se odvijaju nešto drugačije. Kao što je već pomenuto, klasa `Derived` se stvarno sastoji iz dva dela: deo `Base` i deo `Derived`. Jezik C++ konstruiše objekte izvedenih klasa u fazama. Prvo se konstruiše klasa koja se nalazi na vrhu stabla nasleđivanja, a zatim sve izvedene klase redosledom njihovog izvođenja, dok se ne dođe do poslednje klase na dnu stabla nasleđivanja.

Dakle, kada se instancira objekat klase `Derived`, prvo se konstruiše deo klase `Derived` koji se odnosi na klasu `Base` (koristeći podrazumevani konstruktor klase `Base`). Kada se taj deo završi, konstruiše se deo koji se odnosi na klasu `Derived` (koristeći podrazumevani konstruktor klase `Derived`).

Ovaj proces se može jednostavno prikazati.

```
#include <iostream>

class Base
{
public:
   int m_id;

   Base(int id=0)
      : m_id(id)
   {
      std::cout << "Base\n";
   }

   int getId() const { return m_id; }
};

class Derived: public Base
{
public:
   double m_cost;

   Derived(double cost=0.0) : m_cost(cost)
   {
```

```
24          std::cout << "Derived\n";
25      }
26
27      double getCost() const { return m_cost; }
28  };
29
30  int main()
31  {
32      cout << "Instanciranje objekta klase Base" << endl;
33      Base cBase;
34
35      cout << "Instanciranje objekta klase Derived" << endl;
36      Derived cDerived;
37  }
```

Izlaz iz programa će biti:

```
Instanciranje objekta klase Base
Base
Instanciranje objekta klase Derived
Base
Derived
```

Dakle, instanciranjem objekta izvedene klase `Derived`, prvo se poziva konstruktor bazne klase `Base`, a zatim konstruktor izvedene klase `Derived`.

Redosled konstruisanja prilikom ulančanog nasleđivanja

Ponekad se javlja potreba da se klase izvode iz drugih klasa, koje su i same dobijene nasleđivanjem drugih klasa.

```
1   class A
2   {
3   public:
4       A()
5       {
6           cout << "A" << endl;
7       }
8   };
9
10  class B: public A
11  {
12  public:
13      B()
14      {
15          cout << "B" << endl;
16      }
17  };
18
19  class C: public B
20  {
21  public:
```

```
22      C()
23      {
24          cout << "C" << endl;
25      }
26  };
27
28  class D: public C
29  {
30  public:
31      D()
32      {
33          cout << "D" << endl;
34      }
35  };
```

Jezik C++ uvek prvo konstruiše "najosnovnije" klase. Zatim prolazi kroz stablo nasleđivanja i konstruiše redom svaku nasleđenu klasu. Sledi program koji ilustruje redosled konstruisanja kod ulančanog nasleđivanja.

```
1   int main()
2   {
3       cout << "Instanciranje objekta klase A: " << endl;
4       A a;
5       cout << "Instanciranje objekta klase B: " << endl;
6       B b;
7       cout << "Instanciranje objekta klase C: " << endl;
8       C c;
9       cout << "Instanciranje objekta klase D: " << endl;
10      D d;
11  }
```

Prethodni kod će odštampati sledeći izlaz:

```
Instanciranje objekta klase A:
A
Instanciranje objekta klase B:
A
B
Instanciranje objekta klase C:
A
B
C
Instanciranje objekta klase D:
A
B
C
D
```

Konstruktori i inicijalizacija izvedenih klasa

Do sada su pokazane osnove nasleđivanja u jeziku C++ i redosled kojim se izvedene klase inicijalizuju. Sada će detaljnije biti objašnjena uloga konstruktora u inicijalizaciji izvedenih klasa. Za tu namenu će se koristi jednostavne klase `Base` i `Derived`, definisane u prethodnim primerima.

```
class Base
{
public:
   int m_id;

   Base(int id=0) : m_id(id) { }

   int getId() const { return m_id; }
};
```

```
class Derived: public Base
{
public:
   double m_cost;

   Derived(double cost=0.0) : m_cost(cost) { }

   double getCost() const { return m_cost; }
};
```

Sa neizvedenim klasama, konstruktori klase moraju da brinu samo o svojim članovima. Na primer, objekat klase Base može biti jednostavno instanciran.

```
int main()
{
   Base base(5); // Poziv konstruktora Base(int)
   return 0;
}
```

Prilikom instanciranja objekta klase `Base` događa se sledeće:

1. Alocira se memorija za objekat,
2. Poziva se odgovarajući konstruktor,
3. Inicijalizaciona lista inicijalizuje promenljive,
4. Izvršava se telo konstruktora.

Kod jednostavnih neizvedenih klasa ovo prilično jednostavno funkcioniše. Međutim, kod izvedenih klasa stvari su nešto složenije.

```
int main()
{
```

```
3       Derived derived(1.3); // Poziv konstruktora Derived(double)
4       return 0;
5   }
```

Prilikom instanciranja objekta klase `Derived` događa se sledeće:

1. Alocira se memorija za objekat klase `Derived` (dovoljna za sve članove, i `Base` i `Derived`),
2. Poziva se odgovarajući konstruktor klase `Derived`,
3. Objekat klase `Base` se prvo konstruiše pomoću odgovarajućeg konstruktora klase `Base`. Ako nijedan bazni konstruktor nije naveden, koristiće se podrazumevani konstruktor,
4. Inicijalizaciona lista inicijalizuje promenljive,
5. Izvršava se telo konstruktora,

Jedina stvarna razlika između ovog slučaja i nenasleđenog slučaja je da se konstruktor bazne klase poziva pre nego što konstruktor izvedene klase uradi bilo šta značajno.

Inicijalizacija članova bazne klase

Novi programeri često pokušavaju da reše problem inicijalizacije izvedenih klasa na sledeći način:

```
1   class Derived: public Base
2   {
3   public:
4       double m_cost;
5
6       Derived(double cost=0.0, int id=0) : m_cost(cost), m_id(id)
7       {
8           // Greška!
9       }
10
12      double getCost() const { return m_cost; }
13  };
```

Ovo je dobar pokušaj ali ne funkcioniše. Jezik C++ ne dozvoljava da se nasleđene klase inicijalizuju u inicijalizacionoj listi konstruktora. Drugim rečima, vrednost neke promenljive – člana može se postaviti samo u inicijalizacionoj listi konstruktora koji pripada istoj klasi kao i sama promenljiva – član.

Jedan od razloga zašto je to tako leži u načinu inicijalizacije `const` članova. `const` promenljive se moraju inicijalizovati nekom vrednošću u vreme kreiranja, pa konstruktor bazne klase mora postaviti njegovu vrednost u momentu kada se kreira promenljiva. Međutim, kada se završi konstruktor bazne klase, izvršavaju su

inicijalizacione liste konstruktora izvedenih klasa. Svaka izvedena klasa bi onda imala mogućnost da inicijalizuje tu promenljivu, potencijalno menjajući njenu vrednost. Ovo se ne bi smelo dogoditi.

Krajnji rezultat je da gornji primer ne funkcioniše jer je promenljiva `m_id` nasleđena iz klase Base, a samo nenasleđene promenljive se mogu promeniti u inicijalizacionoj listi.

Međutim, nasleđene promenljive se mogu i dalje promeniti u telu konstruktora koristeći dodelu vrednosti. Kao posledica toga, novi programeri često pokušavaju i to.

Iako bi u primeru koji sledi ovo funkcionisalo, pojavio bi se problem ako bi promenljiva `m_id` bila `const` (zato što se `const` vrednosti inicijalizuju u inicijalizacionoj listi konstruktora). Takođe je i neefikasno, jer `m_id` dobija vrednost dva puta: jednom u inicijalizacionoj listi konstruktora bazne klase, a zatim ponovo u telu konstruktora izvedene klase. I na kraju, ako je baznoj klasi potreban pristup ovoj promenljivoj tokom svoje izgradnje, ona nema načina da joj pristupi, jer joj nije dodeljena vrednost sve dok se konstruktor klase `Derived` ne izvrši.

```
1   class Derived: public Base
2   {
3   public:
4      double m_cost;
5
6      Derived(double cost=0.0, int id=0) : m_cost(cost)
7      {
8         m_id = id;
9      }
10
11     double getCost() const { return m_cost; }
12  };
```

U svim dosadašnjim primerima, kada se instancira objekat klase `Derived`, deo klase `Base` je kreiran korišćenjem podrazumevanog baznog konstruktora. Razlog zašto je uvek korišćen podrazumevani konstruktor bazne klase je činjenica da nikada nismo rekli da klasa uradi drugačije.

Jezik C++ daje mogućnost da se eksplicitno izabere koji će konstruktor bazne klase biti pozvan. Kako bi se ovo uradilo, jednostavno treba dodati poziv konstruktora bazne klase u inicijalizacionoj listi izvedene klase:

```
1   class Derived: public Base
2   {
3   public:
```

```
4       double m_cost;
5
6       Derived(double cost=0.0, int id=0)
7          : Base(id), // Poziv konstruktora Base(int)
8             m_cost(cost)
9       {
10      }
11
12       double getCost() const { return m_cost; }
13  };
```

Sada, kada se izvrši ovakav kod, konstruktor bazne klase `Base(int)` će se koristiti za inicijalizaciju promenljive `m_id` na `5`, a konstruktor izvedene klase će se koristiti za inicijalizaciju promenljive `m_cost` na vrednost `1.3`.

```
1   int main()
2   {
3      Derived derived(1.3, 5);
4      std::cout << "Id: " << derived.getId() << '\n';
5      std::cout << "Cost: " << derived.getCost() << '\n';
6      return 0;
7   }
```

Program će štampati sledeći izlaz:

```
Id: 5
Cost: 1.3
```

Detaljnije posmatrano, dešava se sledeće:

1. Alocira se memorija za objekat klase `Derived`,
2. Poziva se konstruktor klase `Derived(double, int)`, sa argumentima `1.3` i `5`,
3. Kompajler gleda da li je tražen određeni konstruktor bazne klase `Base`, i poziva ga ako je zahtevano,
4. Inicijalizaciona lista bazne klase postavlja `m_id` na `5`,
5. Izvršava se telo konstruktora bazne klase, što u ovom slučaju ne radi ništa,
6. Program se vraća iz konstruktora bazne klase,
7. Inicijalizaciona lista konstruktora izvedene klase `Derived` dodeljuje promenljivoj `m_cost` vrednost `1.3`
8. Izvršava se telo konstruktora izvedene klase, što u ovom slučaju ne radi ništa,
9. Program se vraća iz tela konstruktora izvedene klase.

Ovo može izgledati donekle složeno, ali je zapravo vrlo jednostavno. Sve što se dešava jeste da konstruktor izvedene klase poziva određeni konstruktor bazne klase da inicijalizuje bazni deo objekta. Budući da se promenljiva `m_id` nalazi u

baznom objektu, konstruktor kazne klase je jedini konstruktor koji može inicijalizovati tu vrednost.

Svi do sada prikazani primeri su imali članove sa javnim pristupom. Ukoliko se promene tako da budu privatni, neophodno je obezbediti funkcije pristupa (`get` i `set`) za sve promenljive, s obzirom da izvedene klase ne mogu da pristupe privatnim članovima bazne klase. O ovome će biti više reči kasnije.

Konstruktori kod ulančanog nasleđivanja

Klase nastale ulančanim nasleđivanjem rade na potpuno analogan način.

```
#include <iostream>

class A
{
public:
   A(int a) { std::cout << "A: " << a << '\n'; }
};

class B: public A
{
public:
   B(int a, double b) : A(a) { std::cout << "B: " << b << '\n'; }
};

class C: public B
{
public:
   C(int a , double b , char c) : B(a, b)
   {
      std::cout << "C: " << c << '\n';
   }
};

int main()
{
   C c(5, 4.3, 'R');

   return 0;
}
```

U ovom primeru klasa C je izvedena iz klase B, koja je izvedena iz klase A. Kada se instancira objekat klase C, konstruktor C poziva `B(int, double)`. Konstruktor klase B poziva konstruktor `A(int)`. S obzirom da klasa A ne nasleđuje ništa, ovo je prva klasa koja će biti konstruisana. U konstruktoru klase A biće odštampana vrednost 5, a zatim će se kontrola programa vratiti na konstruktor klase B. Konstruktor klase B štampa vrednost `4.3`, i vraća kontrolu u konstruktor klase C. Konstruiše se klasa C, štampa vrednost 'R' i vraća kontrola `main()` funkciji.

Dakle, ovaj program štampa:

```
A: 5
B: 4.3
C: R
```

Destruktori izvedenih klasa

Kada se objekat izvedene klase uništava, svaki destruktor se poziva u redosledu obrnutom od redosleda izvršavanja konstruktora. U prethodnom primeru, kada se objekat c uništava, prvo se poziva `C` destruktor, zatim `B` destruktor, i na kraju `A` destruktor.

Nasleđivanje i specifikatori pristupa

Do sada je pokazano kako funkcioniše nasleđivanje na jednostavnim primerima. U svakom primeru je korišćen princip javnog (`public`) nasleđivanja, tj. izvedena klasa je javno nasleđivala osnovnu klasu.

Sada će malo temeljnije biti obrađeno `public` nasleđivanje, kao i druga dva tipa nasleđivanja: `private` i `protected`. Takođe će biti pokazano kako različite vrste nasleđivanja utiču na kontrolu pristupa nasleđenih članova.

U primerima koji su do sada obrađeni korišćeni su `public` i `private` specifikatori pristupa, koji određuju ko može pristupiti članovima klase. Javnim članovima može pristupiti bilo ko. Privatnim članovima mogu pristupiti samo funkcije članice iste klase ili prijateljske klase (`friend` klase). Ovo znači da izvedene klase ne mogu direktno pristupiti privatnim članovima osnovne klase!

Specifikator pristupa `protected`

U jeziku C++ postoji i treći specifikator pristupa, koji se koristi samo u kontekstu nasleđivanja. Specifikator pristupa `protected`, odnosno zaštićeni pristup, dozvoljava pristup članovima klase: klasi kojoj član pripada, prijateljskim klasama i izvedenim klasama. Zaštićeni članovi nisu dostupni izvan klase.

U primeru koji sledi, može se videti da je zaštićeni član `m_protected` klase `Base` direktno dostupan i od strane izvedene klase, ali nije javno dostupan.

Izvedene klase mogu direktno pristupiti zaštićenom članu bazne klase. To znači da, ako se promeni bilo šta vezano za zaštićeni član (tip, šta njegova vrednost znači, itd.), verovatno će morati da se promeni i bazna klasa i sve izvedene klase.

Zbog toga je korišćenje `protected` specifikatora pristupa najkorisnije kada se dizajniraju vlastite klase koje su izvedene iz klasa koje ste takođe vi napravili, pri čemu je broj izvedenih klasa razuman. Na taj način, ako se napravi promena u implementaciji bazne klase, najverovatnije će izmene pretrpeti i izvedene klase, ali

to verovatno možete učiniti sami, s obzirom da broj izvedenih klasa ne bi trebalo da je velik.

Ako su članovi klasa privatni, biće ostvarena bolja enkapsulacija i izvedena klasa će biti izolovana od promena napravljenih nad osnovnom klasom.

```
class Base
{
public:
   int m_public;    // Dostupan svuda
private:
   int m_private;   // Dostupan samo unutar klase
protected:
   int m_protected; // Dostupan unutar klase i izvedenih klasa
};

class Derived: public Base
{
public:
   Derived()
   {
      m_public = 1;      // Dozvoljeno
      m_private = 2;     // Nije dozvoljeno
      m_protected = 3;   // Dozvoljeno (Derived nasleđuje Base)
   }
};

int main()
{
   Base b;
   b.m_public = 1;       // Dozvoljeno
   b.m_private = 2;      // Nije dozvoljeno izvan klase
   b.m_protected = 3;    // Nije dozvoljeno izvan klase
}
```

Različite vrste nasleđivanja i njihov uticaj na pristup

Postoje tri različita načina nasleđivanja: javno, privatno i zaštićeno nasleđivanje (`public`, `private` i `protected`). Sledi primer sva tri tipa nasleđivanja.

```
class Pub: public Base    // Javno nasleđivanje
{ // Telo klase };

class Pri: private Base   // Privatno nasleđivanje
{ // Telo klase };

class Pro: protected Base // Zaštićeno nasleđivanje
{ // Telo klase };

class Def: Base           // Podrazumevano privatno nasleđivanje
{ // Telo klase };
```

Ako tip nasleđivanja nije eksplicitno naveden, jezik C++ podrazumeva privatno nasleđivanje (baš kao što su i članovi klase podrazumevano privatni).

Javno nasleđivanje

Javno nasleđivanje je daleko najčešće korišćena vrsta nasleđivanja. Zapravo, druge vrste nasleđivanja se veoma retko koriste, tako da bi primarni fokus trebalo da bude na razumevanju ovog odeljka. Javno nasleđivanje je najlakše razumeti. Kada se osnovna klasa nasledi javno, nasleđeni javni članovi ostaju javni, a nasleđeni zaštićeni članovi ostaju zaštićeni. Nasleđeni privatni članovi nisu dostupni jer su bili privatni u baznoj klasi.

Specifikator pristupa u baznoj klasi	Specifikator pristupa u izvedenoj klasi
`public`	`public`
`private`	nedostupan
`protected`	`protected`

```
1   class Base
2   {
3   public:
4      int m_public;
5   private:
6      int m_private;
7   protected:
8      int m_protected;
9   };
10
11  class Pub: public Base // Javno nasleđivanje
12  {
13  public:
14     Pub()
15     {
16        m_public = 1;   // Dozvoljeno
17        m_private = 2;  // Nije dozvoljeno, privatni član bazne klase
18        m_protected = 3;// Dozvoljeno, zaštićeni član bazne klase
19     }
20  };
21
22  int main()
23  {
24     Base b;
25     b.m_public = 1;    // Dozvoljeno
26     b.m_private = 2;   // Nije dozvoljeno, privatni član
27     b.m_protected = 3; // Nije dozvoljeno, zaštićeni član
28     Pub p;
29     p.m_public = 1;    // Dozvoljeno
30     p.m_private = 2;   // Nije dozvoljeno, privatni član
31     p.m_protected = 3; // Nije dozvoljeno, zaštićeni član
32  }
```

Ovaj primer je isti kao i prethodni primer gde je predstavljen `protected` specifikator pristupa, osim što je instancirana i izvedena klasa, kako bi se pokazalo da sa javnim nasleđivanjem stvari funkcionišu identično u baznoj i izvedenoj klasi.

Kao što je već napomenuto, javno nasleđivanje je najčešće korišćeni tip nasleđivanja i koristi se gotovo uvek, osim ako postoji neki bitan razlog da to ne bude tako.

Privatno nasleđivanje

Kod privatnog nasleđivanja, svi članovi iz osnovne klase nasleđeni su kao privatni. To znači da su privatni članovi nedostupni, a zaštićeni i javni članovi postaju privatni u izvedenoj klasi.

Ovo ne utiče na način na koji izvedena klasa pristupa članovima nasleđenim od svog roditelja! To utiče samo na kod koji spolja pokušava pristupiti tim članovima kroz objekat izvedene klase.

Specifikator pristupa u baznoj klasi	Specifikator pristupa u izvedenoj klasi
`public`	`private`
`private`	nedostupan
`protected`	`private`

Sledi primer privatnog nasleđivanja. Klasa `Base` data je u prethodnom primeru.

```
class Pri: private Base // Privatno nasleđivanje
{
public:
   Pri()
   {
      m_public = 1;   // Dozvoljeno
      m_private = 2;  // Nije dozvoljeno, privatni član klase Base
      m_protected = 3;// Dozvoljeno, privatni član klase Pri
   }
};

int main()
{
   Base b;
   b.m_public = 1;    // Dozvoljeno
   b.m_private = 2;   // Nije dozvoljeno, privatni član klase Base
   b.m_protected = 3; // Nije dozvoljeno: zaštićeni član klase Base

   Pri p;
   p.m_public = 1;    // Nije dozvoljeno, privatni član klase Pri
   p.m_private = 2;   // Nije dozvoljeno, privatni član klase Base
   p.m_protected = 3; // Nije dozvoljeno, privatni član klase Pri
}
```

Privatno nasleđivanje može biti korisno kada izvedena klasa nema očigledan odnos sa baznom klasom, ali koristi baznu klasu za internu implementaciju. U tom slučaju verovatno ne postoji potreba da javni interfejs bazne klase bude dostupan kroz objekte izvedene klase (što bi inače bilo, da je nasleđivanje javno).

U stvarnosti se ovo veoma retko koristi.

Zaštićeno nasleđivanje

Zaštićeno nasleđivanje je poslednji tip nasleđivanja. Ova vrsta nasleđivanja se veoma retko koristi. Kod zaštićenog nasleđivanja, javni i zaštićeni članovi postaju zaštićeni, a privatni članovi ostaju nedostupni.

U nastavku je data tabela promene specifikatora pristupa kod zaštićenog nasleđivanja.

Specifikator pristupa u baznoj klasi	Specifikator pristupa u izvedenoj klasi
`public`	`protected`
`private`	nedostupan
`protected`	`protected`

Pozivanje nasleđenih funkcija

Pozivanje funkcije osnovne klase

Kada se funkcija članica poziva iz objekta izvedene klase, kompajler prvo traži funkciju u izvedenoj klasi. Ako je ne pronađe, počinje da se kreće kroz lanac nasleđivanja i proverava da li je funkcija definisana u bilo kojoj od nadklasa, i koristi prvu koju pronađe (sledi primer).

```
1   class Base
2   {
3   protected:
4      int m_value;
5
6   public:
7      Base(int value) : m_value(value) { }
8
9      void identify() { std::cout << "Objekat klase Base\n"; }
10  };
11
12  class Derived : public Base
13  {
14  public:
15     Derived(int value) : Base(value) { }
16  };
```

```
int main()
{
   Base b(5);
   b.identify();

   Derived d (7);
   d.identify();

   return 0;
}
```

Izlaz iz ovog koda je:

```
Objekat klase Base
Objekat klase Base
```

Kada se poziva `derived.identify()`, kompajler će pogledati da li je funkcija `identify()` definisana u klasi `Derived`. S obzirom da to ovde nije slučaj, kompajler traži dalje u nasleđenim klasama (što je u ovom slučaju klasa `Base`). U klasi `Base` je definisana funkcija `identify()`, tako da će ona biti pozvana. Drugim rečima, funkcija `Base::identify()` je korišćena jer funkcija `Derived::identify()` ne postoji.

To znači da ako je neka funkcija koju obezbeđuje bazna klasa dovoljna, može se jednostavno koristiti i u izvedenim klasama.

Redefinisanje ponašanja

Međutim, da je postojala definicija funkcije `Derived::identify()` u klasi `Derived`, ona bi bila korišćena u prethodnom primeru.

To znači da mogu da se napišu funkcije sa istim imenom (i potpisom) u izvedenim klasama koje rade drugačije u odnosu na bazne klase.

U prethodnom primeru, bilo bi tačnije ako bi poziv `derived.identify()` ispisao "`Objekat klase Derived`". U skladu sa tim, dodata je i ta funkcija u klasi `Derived`. Kako bi se funkcija definisana u baznoj klasi promenila tako da radi drugačije u izvedenoj klasi, jednostavno je treba redefinisati u izvedenoj klasi. Sledi prethodni primer, ali sada koristeći novu `Derived::identify()` funkciju.

```
class Derived: public Base
{
public:
   Derived(int value) : Base(value) { }
   int getValue() { return m_value; }
   void identify() { std::cout << "Objekat klase Derived\n"; }
};
```

```
int main()
{
   Base b(5);
   b.identify();

   Derived d(7);
   d.identify();

   return 0;
}
```

```
Objekat klase Base
Objekat klase Derived
```

Kada se redefiniše funkcija u izvedenoj klasi, izvedena funkcija ne nasleđuje specifikator pristupa koji je imala u baznoj klasi već ga dobija u izvedenoj klasi. Stoga, funkcija koja je definisana kao privatna u osnovnoj klasi, može biti redefinisana tako da bude javna u izvedenoj klasi ili obrnuto.

```
class Base
{
private:
   void print()
   {
      std::cout << "Base";
   }
};

class Derived : public Base
{
public:
   void print()
   {
      std::cout << "Derived ";
   }
};
```

```
int main()
{
   Derived d;
   d.print(); // Poziv javne funkcije derived::print()
   return 0;
}
```

Dodavanje u postojeću funkcionalnost

Ponekad ne treba u potpunosti zameniti funkciju osnovne klase, već joj treba dodati dodatnu funkcionalnost. U prethodnom primeru, funkcija `Derived::identify()` u potpunosti sakriva (maskira) funkciju

`Base::identify()`. Ovo, međutim, često nije ono što je potrebno uraditi. Moguće je da izvedena funkcija poziva osnovnu verziju funkcije istog imena (da bi ponovo koristila kod), a zatim dodaje neku dodatnu funkcionalnost.

Izvedena funkcija može da pozove osnovnu funkciju istog imena, jednostavnim pozivanjem zajedno sa nazivom bazne klase (odnosno klase u kojoj je funkcija definisana, ukoliko se radi o ulančanom nasleđivanju) i operatorom razrešenja opsega (naziv osnovne klase + `::`). U sledećem primeru je redefinisana funkcija `Derived::identify()` tako da prvo poziva `Base::identify()`, a zatim dodaje svoju vlastitu funkcionalnost.

```
class Derived: public Base
{
public:
   Derived(int value) : Base(value)
   {
   }

   int GetValue() { return m_value; }

   void identify()
   {
      Base::identify(); // Poziv funkcije Base::identify()
      // Dodatna funkcionalnost funkcije Derived::identify()
      std::cout << "Objekat klase Derived\n";
   }
};

int main()
{
   Base b(5);
   b.identify();

   Derived d(7);
   d.identify();

   return 0;
}
```

Izlaz iz ovog koda će biti:

```
Objekat klase Base
Objekat klase Base
Objekat klase Derived
```

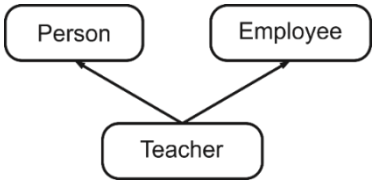
Kada se izvršava funkcija `derived.identify()`, prva stvar koju funkcija `Derived::identify()` radi je poziv funkcije `Base::identify()`, koja štampa tekst "`Objekat klase Base`". Kada se kontrola programa vrati iz

`Base::identify()`, funkcija `Derived::identify()` nastavlja da se izvršava i štampa tekst "`Objekat klase Derived`".

Višestruko nasleđivanje

Svi primeri nasleđivanja koji su do sada prikazani predstavljaju jednostruko nasleđivanje - to jest, svaka nasleđena klasa ima samo jednog roditelja. Međutim, jezik C++ pruža i mogućnost višestrukog nasleđivanja. Višestruko nasleđivanje omogućava izvedenoj klasi da nasledi članove od više roditelja – odnosno više baznih klasa.

Recimo da je potrebno napisati program koji će čuvati informacije o nastavnicima. Nastavnik je osoba. Pored toga, nastavnik je i zaposleni. Višestruko nasleđivanje se može koristiti za kreiranje klase `Teacher` koja nasleđuje osobine klasa `Person` i `Employee`. Da bi se koristilo višestruko nasleđivanje, jednostavno treba navesti svaku baznu klasu sa znakom "`,`" između pojedinih klasa.



Slika 8: Primer višestrukog nasleđivanja

```
1   #include <string>
2
3   class Person
4   {
5   private:
6      std::string m_name;
7      int m_age;
8   public:
9      Person(std::string name, int age) : m_name(name), m_age(age) { }
10
11     std::string getName() { return m_name; }
12     int getAge() { return m_age; }
13  };
```

```
#include <string>

class Employee
{
private:
   std::string m_employer;
   double m_wage;
public:
   Employee(std::string employer, double wage)
      : m_employer(employer), m_wage(wage) { }

   std::string getEmployer() { return m_employer; }
   double getWage() { return m_wage; }
};
```

```
// Klasa Teacher javno nasleđuje klase Person i Employee
class Teacher: public Person, public Employee
{
private:
   int m_teachesGrade;
public:
   Teacher(std::string name, int age, std::string employer,
                double wage, int teachesGrade)
      : Person(name, age),
        Employee(employer, wage),
        m_teachesGrade(teachesGrade)
   {
   }
};
```

Problemi sa višestrukim nasleđivanjem

Iako višestruko nasleđivanje izgleda kao jednostavno proširivanje jednostrukog nasleđivanja, višestruko nasleđivanje donosi izvesne probleme koji mogu značajno povećati složenost programa i učiniti ih teškim za održavanje.

Prvo, može se pojaviti dvosmislenost kada više baznih klasa sadrži funkciju sa istim imenom.

```
class USBDevice
{
private:
   long m_id;
public:
   USBDevice(long id) : m_id(id) { }

   long getID() { return m_id; }
};
```

```
class NetworkDevice
{
private:
   long m_id;
public:
   NetworkDevice(long id) : m_id(id) { }

   long getID() { return m_id; }
};
```

```
class WirelessAdapter: public USBDevice, public NetworkDevice
{
public:
   WirelessAdapter(long usbId, long networkId)
      : USBDevice(usbId), NetworkDevice(networkId)
   {
   }
};
```

```
#include <iostream>

int main()
{
   WirelessAdapter c54G(5442, 181742);
   std::cout << c54G.getID(); // Koja funkcija getID() se poziva?

   return 0;
}
```

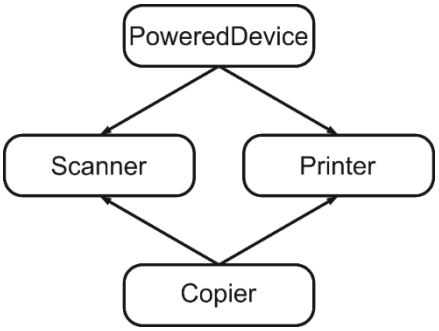
Kada se kompajlira linija `c54G.getID()`, kompajler će videti da `WirelessAdapter` nema funkciju nazvanu `getID()`. Prevodilac potom razmatra da li bilo koja od nadklasa ima funkciju pod nazivom `getID()`. Obe nadklase, `USBDevice` i `NetworkDevice` imaju svoje funkcije `getID()`. Ovaj poziv je dvosmislen i kompajler će prijaviti grešku pri pokušaju kompajliranja ovakvog koda.

Postoji način da se reši ovaj problem: može se eksplicitno navesti koja verzija funkcije treba da se pozove.

```
std::cout << c54G.USBDevice::getID();
```

Upotrebom operatora razrešenja opsega je moguće pozvati odgovarajuću funkciju. Međutim, ako neka klasa nasleđuje veći broj klasa, koje su takođe izvedene iz nekih drugih klasa, jasno je da se na ovaj način program u značajnoj meri komplikuje.

Drugi ozbiljan problem se javlja kada klasa nasleđuje dve klase, pri čemu one nasleđuju istu baznu klasu.



Slika 9: Primer problematičnog nasleđivanja

```
class PoweredDevice
{
};

class Scanner: public PoweredDevice
{
};
class Printer: public PoweredDevice
{
};

class Copier: public Scanner, public Printer
{
};
```

U prethodnom primeru i skeneri i štampači koriste napajanja, tako da su izvedeni iz klase `PoweredDevice`. Međutim, mašina za kopiranje uključuje funkcionalnost i skenera i štampača. Postoji mnogo pitanja koja se pojavljuju u ovom kontekstu, uključujući i to da li mašina za kopiranje treba da ima jednu ili dve kopije objekta `PoweredDevice` i kako da reši dvosmislenosti koje se javljaju kao posledica ovakvog nasleđivanja. Iako se većina ovih problema može rešiti putem eksplicitnog razrešenja opsega, ovo se u praksi uglavnom ne koristi. Razlozi bi trebalo da su očigledni.

Većina problema koji se mogu rešiti korišćenjem višestrukog nasleđivanja mogu se rešiti korišćenjem jednostrukog nasleđivanja. Mnogi objektno orijentisani jezici čak ne podržavaju višestruko nasleđivanje. Mnogi relativno moderni jezici kao što su Java i C# ograničavaju klase na jednostruko nasleđivanje, ali omogućavaju višestruko nasleđivanje (odnosno implementaciju) interfejsa.

Višestruko nasleđivanje čini jezik previše kompleksnim i najčešće pravi više štete nego koristi.

12. Virtuelne funkcije i polimorfizam

Pokazivači i reference na baznu klasu objekata izvedene klase

U prethodnom poglavlju je obrađena tema nasleđivanja, gde je prikazano kako se nasleđivanjem izvode nove klase iz postojećih klasa. U ovom poglavlju fokus će biti na jedanom od veoma važnih elemenata nasleđivanja - virtuelnim funkcijama.

Kada je bilo reči o nasleđivanju, pokazano je da se izvedena klasa sastoji iz više delova: po jedan deo za svaku nasleđenu klasu, i jedan deo za sebe (tj. za vlastite članove i funkcije).

```
1   class Base
2   {
3   protected:
4
5      int m_value;
6
7   public:
8      Base(int value) : m_value(value)
9      {
10     }
11
12     const char* getName() { return "Base"; }
13     int getValue() { return m_value; }
14  };
15
16  class Derived: public Base
17  {
18  public:
19
20     Derived(int value) : Base(value)
21     {
22     }
23
24     const char* getName() { return "Derived"; }
25     int getValueDoubled() { return m_value * 2; }
26  };
```

Kada se kreira objekat klase `Derived`, on sadrži deo koji se odnosi na klasu `Base`, i deo koji se odnosi na klasu `Derived`.

Pokazivači, reference i izvedene klase

Pokazivač i referenca na klasu `Derived` se veoma jednostavno deklarišu:

```
#include <iostream>

int main()
{
   Derived d(5);

   Derived &r = derived;
   Derived *p = &derived;

   std::cout << d.getName()  << "=>" << d.getValue()  << '\n';
   std::cout << r.getName()  << "=>" << r.getValue()  << '\n';
   std::cout << p->getName() << "=>" << p->getValue() << '\n';
}
```

Izlaz iz ovog programa će biti:

```
Derived=>5
Derived=>5
Derived=>5
```

Međutim, pošto je klasa `Derived` izvedena iz klase `Base`, interesantnije pitanje je da li će jezik C++ dozvoliti da se pokazivaču ili referenci na klasu `Base` dodeli adresa ili referenca na objekat klase `Derived`? Kompajliranjem sledećeg koda ispostaviće se da je to izvodljivo.

```
#include <iostream>

int main()
{
   Derived d(5);

   // These are both legal!
   Base &rBase = d;
   Base *pBase = &d;

   std::cout << d.getName()     << "=>" << d.getValue()     <<'\n';
   std::cout << rBase.getName() << "=>" << rBase.getValue() <<'\n';
   std::cout << pBase->getName()<< "=>" << pBase->getValue()<<'\n';
}
```

Izlaz će biti:

```
Derived=>5
Base=>5
Base=>5
```

Ovaj rezultat možda nije baš očekivan. Ispostavlja se da su `rBase` i `pBase` referenca i pokazivač na klasu `Base` i da oni mogu videti samo članove klase `Base` (ili bilo koje klase koju je klasa `Base` nasledila). Dakle, iako `Derived::getName()` sakriva `Base::getName()`, pokazivač ili referenca na klasu `Base` ne mogu videti `Derived::getName()`. Shodno tome, oni će pozivati `Base::getName()`. Takođe, pokazivač i referenca na klasu `Base` ne mogu pozvati `Derived::getValueDoubled()` koristeći promenljive `rBase` ili `pBase`. One ne mogu videti bilo šta definisano u klasi `Derived`.

U primeru koji sledi definisana je bazna klasa `Animal`, koja ima jednu promenljivu `m_name`, konstruktor i dve funkcije: `getName()` i `speak()`. Zatim su iz te klase izvedene klase `Cat` i `Dog`, koje imaju redefinisane funkcije `speak()`.

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

class Animal
{
protected:

   std::string m_name;

   Animal(std::string name) : m_name(name) {}

public:

   std::string getName() { return m_name; }

   const char* speak() { return "???"; }
};

class Cat: public Animal
{
public:

   Cat(std::string name) : Animal(name) { }

   const char* speak() { return "Mijaukanje"; }
};

class Dog: public Animal
{
public:

   Dog(std::string name) : Animal(name) { }

   const char* speak() { return "Lavez"; }
};
```

```
int main()
{
    Cat cat("Tom");
    std::cout << cat.getName() << " => " << cat.speak() << '\n';

    Dog dog("Snupi");
    std::cout << dog.getName() << " => " << dog.speak() << '\n';

    Animal *p = &cat;
    std::cout << p->getName()  << " => " << p->speak()  << '\n';

    p = &dog;
    std::cout << p->getName()  << " => " << p->speak()  << '\n';

    return 0;
}
```

Izlaz iz ovog koda će biti:

```
Tom => Mijaukanje
Snupi => Lavez
Tom => ???
Snupi => ???
```

Pošto je `pAnimal` pokazivač na `Animal`, on može videti samo deo klase koji se odnosi na `Animal`. Prema tome, `pAnimal->speak()` će pozvati `Animal::speak()` umesto `Dog::speak()` ili `Cat::speak()`.

Korišćenje pokazivača i referenci na baznu klasu

Postoji više razloga za korišćenje pokazivača ili referenci na baznu klasu.

Prvo, recimo da treba napisati funkciju koja štampa ime i zvuk životinje. Bez upotrebe pokazivača na baznu klasu, neophodne bi bile dve preklopljene funkcije.

```
void report(Cat &cat)
{
    std::cout << cat.getName() << " => " << cat.speak() << '\n';
}

void report(Dog &dog)
{
    std::cout << dog.getName() << " => " << dog.speak() << '\n';
}
```

Ovo ne izgleda previše teško, ali šta bi se desilo ako bi trebalo uraditi isto ali za 30 različitih vrsta životinja umesto dve? Postojalo bi 30 skoro identičnih funkcija! Takođe, ako se ikada doda nova vrsta životinja, uz nju bi išla i jedna nova funkcija. Ovo je ogromno gubljenje vremena s obzirom da je jedina stvarna razlika između svih tih funkcija samo tip parametra.

S obzirom da su Cat i Dog klase izvedene iz klase Animal, Cat i Dog imaju deo koji se odnosi na Animal. Prema tome, smisleno bi bilo uraditi nešto nalik ovome:

```
void report(Animal &rAnimal)
{
   std::cout << rAnimal.getName() <<" => "<< rAnimal.speak() <<'\n';
}
```

Ovo bi omogućilo prosleđivanje objekta bilo koje klase izvedene iz Animal, čak i one klase koje su nastale nakon što je napisana funkcija. Umesto jedne funkcije po svakoj izvedenoj klasi, napisana je jedna funkcija koja radi sa svim klasama izvedenim iz Animal!

Problem je, naravno, što je rAnimal referenca na Animal, tako da će rAnimal.speak() pozvati Animal::speak() umesto izvedene verzije speak().

Drugo, recimo da postoje tri mačke i tri psa koje treba držati u nizu kako bi im se lakše pristupalo. Pošto se u nizove mogu smestiti samo objekti jedne klase, bez pokazivača ili reference na baznu klasu, neophodno bi bilo definisati poseban niz za svaki izvedeni tip.

```
#include <iostream>

int main()
{
   Cat cats[] = { Cat("Tom"), Cat("Silvester"), Cat("Garfild") };
   Dog dogs[] = { Dog("Snupi"), Dog("Pluton"), Dog("Toto") };

   for (int i = 0; i < 3; i++)
      std::cout << cats[i].getName() << " => "
                << cats[i].speak()   << '\n';

   for (int i = 0; i < 3; ++i)
      std::cout << dogs[i].getName() << " => "
                << dogs[i].speak()   << '\n';

   return 0;
}
```

Ako bi postojalo 30 različitih vrsta životinja, trebalo bi deklarisati 30 različitih nizova, po jedan za svaku vrstu životinja.

Međutim, klase Cat i Dog su izvedene iz Animal, tako da je moguće deklarisati objekte ovih klasa, a zatim napraviti niz pokazivača u kome će biti smeštene njihove adrese.

```
#include <iostream>

int main()
{
   Cat tom("Tom"), silvester("Silvester"), garfild("Garfild");
   Dog snupi("Snupi"), pluton("Pluton"), toto("Toto");

   Animal *animals[] = {     &tom,   &snupi,  &silvester,
                          &garfild,  &pluton,       &toto };
   for (int i=0; i < 6; i++)
      std::cout << animals[i]->getName() << " => "
                << animals[i]->speak()   << '\n';

   return 0;
}
```

Iako ovo može da se kompajlira i izvršava, nažalost, činjenica da je svaki element niza `animals[]` pokazivač na `Animal` znači da će `animals[i]->speak()` pozvati `Animal::speak()` umesto funkcije u izvedenim klasama koje zapravo treba pozvati.

Pokazivač ili referenca na baznu klasu pozivaju osnovnu verziju funkcije, a ne izvedenu, i to je jedini nedostatak koji se primećuje u prethodnim primerima. Ovaj problem rešavaju virtuelne funkcije.

Virtuelne funkcije i polimorfizam

Virtuelna funkcija je poseban tip funkcije koja, kada se poziva, poziva najizvedeniju verziju funkcije koja postoji između osnovne i izvedene klase. Ova sposobnost je poznata kao polimorfizam. Izvedena funkcija smatra se podudarnom ako ima isti potpis (ime, tipovi parametara, i da li je `const`) i povratni tip kao osnovna verzija funkcije (ili povratni tip izveden iz povratnog tipa bazne verzije funkcije). Takve funkcije se zovu "nadglasane" funkcije (eng. overrides).

Da bi se definisala funkcija kao virtuelna, jednostavno treba dodati ključnu reč `virtual` pre deklaracije funkcije. Sledi primer s početka poglavlja sa virtuelnom funkcijom.

```
1   #include <iostream>
2
3   class Base
4   {
5   public:
6      virtual const char* getName() { return "Base"; }
7   };
8
9   class Derived: public Base
10  {
11  public:
12     virtual const char* getName(int number) { return "Derived"; }
13  };
14
15  int main()
16  {
17     Derived derived;
18     Base &rBase = derived;
19     std::cout << "rBase je referenca na klasu " << rBase.getName();
20     return 0;
21  }
```

Prethodni primer će štampati:

```
rBase je referenca na klasu Derived
```

Pošto je `rBase` referenca za deo objekta `Derived` koji se odnosi na nasleđenu klasu `Base`, kada se poziva `rBase.getName()`, uobičajeno bi bila pozvana verzija `Base::getName()`. Međutim, `Base::getName()` je virtuelna, i to govori programu da proveri da li postoje izvedene verzije funkcije između klasa `Base` i `Derived`. U ovom slučaju, biće pozvana verzija `Derived::getName()`.

Sledi nešto složeniji primer.

```cpp
#include <iostream>

class A
{
public:
   virtual const char* getName() { return "A"; }
};

class B: public A
{
public:
   virtual const char* getName() { return "B"; }
};

class C: public B
{
public:
   virtual const char* getName() { return "C"; }
};

class D: public C
{
public:
   virtual const char* getName() { return "D"; }
};

int main()
{
   C c;
   A &rBase = c;
   std::cout << "rBase je referenca na klasu " << rBase.getName();

   return 0;
}
```

U prethodnom primeru prvo se instancira objekat klase C. `rBase` je referenca na klasu A, kojoj je dodeljena referenca dela A objekta klase C. Zatim se poziva funkcija `rBase.getName()`. Program pronalazi funkciju `A::getName()`. Međutim, `A::getName()` je virtuelna funkcija, tako da će program pronaći najizvedeniju verziju između klasa A i C. U ovom slučaju, to je `C::getName()`. Treba primetiti da program neće pozvati `D::getName()`, iako je i ona virtuelna, jer je izvorni objekat bio C, a ne D, tako da se razmatraju samo verzije funkcija između A i C.

Kao rezultat, program štampa:

```
rBase je referenca na klasu C
```

Sledi primer sa klasama Dog i Cat, koji je već analiziran. Međutim, u ovom slučaju funkcije `speak()` su virtuelne.

```
#include <iostream>
#include <string>
class Animal
{
protected:
   std::string m_name;

   // Zaštićeni konstruktor onemogućava instanciranje objekata
   // klase Animal ali dozvoljava izvedenim klasama da ga pozovu
   Animal(std::string name) : m_name(name) { }

public:
   std::string getName() { return m_name; }
   virtual const char* speak() { return "???"; }
};

class Cat: public Animal
{
public:
   Cat(std::string name) : Animal(name) { }

   virtual const char* speak() { return "Mijaukanje"; }
};

class Dog: public Animal
{
public:
   Dog(std::string name) : Animal(name) { }

   virtual const char* speak() { return "Lavez"; }
};

void report(Animal &animal)
{
    std::cout << animal.getName() <<" => "<< animal.speak() <<'\n';
}

int main()
{
    Cat cat("Tom");
    Dog dog("Snupi");

    report(cat);
    report(dog);
}
```

Izlaz iz prethodnog programa će biti:

```
Tom => Mijaukanje
Snupi => Lavez
```

Kada se pozove `animal.speak()`, program će imati informaciju da je ta funkcija virtuelna. U slučaju kada se funkciji prosledi objekat tipa `Cat`, program će

pregledati sve klase između `Animal` i `Cat` kako bi pronašao odgovarajuću funkciju u najizvedenijoj klasi. U ovom slučaju pronaći će funkciju `Cat::speak()`. U slučaju da se funkciji prosledi objekat tipa `Dog`, program će pronaći funkciju `Dog::speak()`.

Funkcija `Animal::GetName()` nije virtuelna. Ovo je zato što nije postojala potreba za preklapanjem funkcije `GetName()`, jer se podrazumeva da svaka životinja ima ime.

Slično prethodnom, primer sa nizom koji je opisan ranije će sada funkcionisati na očekivani način.

```
#include <iostream>

int main()
{
   Cat tom("Tom"), silvester("Silvester"), garfild("Garfild");
   Dog snupi("Snupi"), pluton("Pluton"), toto("Toto");

   Animal *animals[] = {     &tom,   &snupi,  &silvester,
                          &garfild,  &pluton,       &toto };
   for (int i=0; i < 6; i++)
      std::cout << animals[i]->getName() << " => "
                << animals[i]->speak()   << '\n';
}
```

Izlaz će biti:

```
Tom => Mijaukanje
Snupi => Lavez
Silvester => Mijaukanje
Garfild => Mijaukanje
Pluton => Lavez
Toto => Lavez
```

Virtuelni destruktori, nadglasavanje virtuelnih funkcija

Virtuelni destruktori

U jeziku C++ postoji podrazumevani destruktor za sve klase, ukoliko nije eksplicitno definisan. Međutim, ponekad postoji potreba za vlastitim destruktorom, naročito ako klasa treba da dealocira memoriju. U slučaju nasleđivanja, uvek bi trebalo da se definišu virtuelni destruktori.

```
#include <iostream>
class Base
{
public:
   ~Base() // Destrutor nije virtuelni
   {
      std::cout << "Poziv destruktora ~Base()" << std::endl;
   }
};

class Derived: public Base
{
private:
   int* m_array;

public:
   Derived(int length)
   {
      m_array = new int[length];
   }

   ~Derived() // Destrutor nije virtuelni
   {
      std::cout << "Poziv destruktora ~Derived()" << std::endl;
      delete[] m_array;
   }
};

int main()
{
   Derived *derived = new Derived(5);
   Base *base = derived;
   delete base;

   return 0;
}
```

Budući da je base pokazivač na klasu Base, kada se base briše, program će pogledati da li je destruktor klase Base virtuelan. U ovom slučaju nije, pa će pretpostaviti da je potrebno pozvati samo destruktor klase Base. To se može videti u izlazu prethodnog primera, koji štampa:

```
Poziv destruktora ~Base()
```

Međutim, u ovom slučaju je potrebno da operator delete pozove destruktor klase Derived (koja će pozvati i destruktor klase Base). U suprotnom, dinamički niz m_array neće biti oslobođen. Da bi se ovo obezbedilo, destruktor klase Base treba da bude virtuelan:

```
#include <iostream>
class Base
{
public:
   virtual ~Base() // Virtuelni destruktor
   {
      std::cout << "Poziv destruktora ~Base()" << std::endl;
   }
};

class Derived: public Base
{
private:
   int* m_array;

public:
   Derived(int length)
   {
      m_array = new int[length];
   }

   virtual ~Derived() // Virtuelni destruktor
   {
      std::cout << "Poziv destruktora ~Derived()" << std::endl;
      delete[] m_array;
   }
};

int main()
{
   Derived *derived = new Derived(5);
   Base *base = derived;
   delete base;

   return 0;
}
```

Izlaz iz programa će biti:

```
Poziv destruktora ~Derived()
Poziv destruktora ~Base()
```

Kao što je već napomenuto, kad god u programu postoji nasleđivanje, za takve klase bi trebalo definisati virtuelne destruktore.

Nadglasavanje virtuelnih funkcija

Nekada se javlja potreba, mada veoma retko, da je potrebno ignorisati virtuelizaciju.

```
class Base
{
public:
    virtual const char* getName() { return "Base"; }
};
class Derived: public Base
{
public:
    virtual const char* getName() { return "Derived"; }
};
```

Postoje slučajevi u kojima je potrebno da pokazivač čiji je tip `Base`, a koji pokazuje na objekat tipa `Derived`, pozove `Base::getName()` umesto `Derived::getName()`. To se jednostavno može uraditi pomoću operatora razrešenja opsega:

```
#include <iostream>
int main()
{
    Derived derived;
    Base &base = derived;
    // Poziv Base::GetName() umesto virtuelne Derived::GetName()
    std::cout << base.Base::getName() << std::endl;
}
```

Čiste virtuelne funkcije

Čiste virtuelne (apstraktne) funkcije i apstraktne bazne klase

Sve virtuelne funkcije koje su do sada analizirane imale su telo (tj. definiciju). Međutim, jezik C++ omogućava kreiranje posebne vrste virtuelnih funkcije koje se nazivaju čistim virtuelnim funkcijama (ili apstraktnim funkcijama). Takve funkcije uopšte nemaju telo. Čista virtuelna funkcija je jednostavno funkcija za kojom će postojati potreba u izvedenim klasama, i u njima će biti definisana.

Čista virtuelna funkcija se deklariše slično kao i bilo koja druga virtuelna funkcija. Jedina razlika je što takva funkcija nema telo i što joj se dodeljuje vrednost `0`.

```
class Base
{
public:
   const char* sayHi() { return "Hi"; }
   virtual const char* getName() { return "Base"; }

   virtual int getValue() = 0; // Čista virtuelna funkcija

   int doSomething() = 0; // Greška: Ne može se dodeliti vrednost 0
                          //         funkciji koja nije virtuelna
};
```

Kada se definiše čista virtuelna funkcija u nekoj klasi, omogućeno je da se takva funkcija implementira u izvedenim klasama.

Korišćenje čiste virtuelne funkcije ima dve važne posledice.

1. Svaka klasa sa jednom ili više čistih virtuelnih funkcija postaje apstraktna bazna klasa, što znači da se ne mogu instancirati objekti te klase.
2. U svakoj izvedenoj klasi se mora definisati telo ovakve funkcije, ili će ta izvedena klasa takođe biti apstraktna klasa.

Primer čiste virtuelne funkcije

U primeru sa klasama koje opisuju životinje, koji je već analiziran, postojala je bazna klasa `Animal` i dve izvedene klase `Cat` i `Dog`. Moguće je redefinisati klasu `Animal`, tako da njena funkcija `speak()` bude čista virtuelna funkcija. U tom slučaju klasa `Animal` postaje apstraktna klasa, što je generalno posmatrano i korektno, s obzirom da instanciranje objekta klase `Animal` nema adekvatnu paralelu u stvarnom životu. Klasa `Animal` treba da bude apstraktna i da sadrži neke osnovne podatke koje imaju sve vrste životinja.

Dakle, klasa `Animal` bi trebalo da izgleda ovako:

```
#include <string>
class Animal // Apstraktna klasa
{
protected:
   std::string m_name;

public:
   Animal(std::string name)
      : m_name(name) { }

   std::string getName() { return m_name; }

   virtual const char* speak() = 0; // Čista virtuelna funkcija
};
```

U ovom primeru treba primetiti nekoliko stvari.

Prvo, `speak()` je sada čista virtuelna funkcija. To znači da je `Animal` sada apstraktna bazna klasa, i ne može se instancirati.

Drugo, pošto je klasa `Cow` izvedena iz `Animal`, ali u njoj nije definisana funkcija `Cow::speak()`, `Cow` će takođe biti apstraktna klasa. Pokušaj kompajliranja prethodnog koda prijaviće poruku o grešci, zato što nije moguće instancirati objekat apstraktne klase.

```
#include <iostream>
class Cow: public Animal
{
public:
   Cow(std::string name) : Animal(name) { }

    // Nije definisana funkcija speak()
};

int main()
{
    Cow cow("Milka"); // Greška! Klasa Cow je apstraktna!
    std::cout << cow.getName() << " => " << cow.speak() << '\n';
}
```

Da bi se omogućilo instanciranje objekata klase Cow, neophodno je unutar klase definisati telo funkcije speak().

```
#include <iostream>
class Cow: public Animal
{
public:
   Cow(std::string name) : Animal(name) { }

   virtual const char* speak() { return "Mukanje"; }
};

int main()
{
   Cow cow("Milka");
   std::cout << cow.getName() << " => " << cow.speak() << '\n';
}
```

Sada se program može kompajlirati i štampaće sledeći izlaz:

```
Milka => Mukanje
```

Čiste virtuelne funkcije su korisne kada postoji potreba za funkcijom koja po svojoj prirodi pripada baznoj klasi, ali samo izvedene klase znaju šta ona treba da radi. Definisanjem čiste virtuelne funkcije onemogućava se instanciranje objekta klase u kojoj je ona definisana, a izvedene klase su prisiljene da definišu ovu funkciju kako bi se mogli instancirati njihovi objekti.

13. Uvod u generičko programiranje

Uvod u generičko programiranje

Generičko programiranje je tehnika koja dozvoljava da jedna promenljiva može da čuva različite tipove podataka (takozvana višeobličnost ili polimorfizam) sve dok su zadovoljeni određeni uslovi kao što su odgovarajuća podklasa i pravilna deklaracija.

U prethodnim poglavljima je objašnjeno kako napraviti funkcije i klase, koje pomažu da programi budu lakši za pisanje, sigurniji za korišćenje i jednostavni za održavanje. Iako su funkcije i klase moćni i fleksibilni alati za efikasno programiranje, u određenim slučajevima oni takođe mogu biti ograničavajući, zbog ograničenja jezika C++ da tipovi svih parametara budu definisani.

Na primer, recimo da treba napisati funkciju za izračunavanje maksimuma dva broja. To bi moglo da izgleda ovako:

```
1  int max(int x, int y)
2  {
3     return (x > y) ? x : y;
4  }
```

Ova funkcija bi funkcionisala za celobrojne promenljive. Međutim, šta bi se dogodilo kasnije kada se javi potreba za sličnom funkcijom koja radi sa promenljivama sa pokretnim zarezom? Tradicionalno, odgovor bi bio da treba preklopiti funkciju `max()` kako bi radila sa `double` promenljivama:

```
1  double max(double x, double y)
2  {
3     return (x > y) ? x : y;
4  }
```

Kod za implementaciju ove nove verzije funkcije `max()` je potpuno isti kao i u verziji koja radi sa celobrojnim promenljivama. Jedina razlika je tip povratne vrednosti i tip parametara. Ovakva implementacija bi funkcionisala za sve tipove promenljivih: `char`, `int`, `double`, `float`, pa čak i za klase, ukoliko poseduju preklopljeni operator >. Međutim, ovakvom logikom svaki novi tip promenljive iziskivao bi definisanje jedne posebne verzije funkcije `max()`.

Ovakav način programiranja, gde se javlja veći broj različitih funkcija koje suštinski rade istu stvar ali sa različitim tipovima podataka, vremenom postaje sve teži za održavanje i najčešće predstavlja bespotrebno gubljenje vremena. Takođe, nije ni

u skladu sa opštim preporukama za programiranje, da dupliranje koda treba svesti na minimum. Bilo bi mnogo elegantnije napisati jednu funkciju `max()` koja može da radi sa parametrima bilo kog tipa.

U jeziku C++, generičke funkcije (koje se još nazivaju “template“ funkcije ili šablonske funkcije) su funkcije koje služe kao šablon za kreiranje drugih sličnih funkcija. Osnovna ideja generičkih funkcija je stvaranje funkcije bez potrebe za definisanjem tačnih tipova nekih ili svih parametara ili promenljivih. Umesto toga, definiše se funkcija korišćenjem promenljivih generičkog tipa. Ovakva funkcija, napisana korišćenjem generičkih tipova, naziva se generička ili šablonska funkcija.

Kada se generička funkcija pozove, kompajler zamenjuje generičke tipove stvarnim tipovima promenljivih koje su funkciji prosleđene kao argumenti. Upotrebom ove metodologije, kompajler može kreirati više "različitih" funkcija iz jednog šablona.

Generičke funkcije u jeziku C++

Kreiranje generičkih funkcija u C++ - u je prilično jednostavno.

```
1   int max(int x, int y)
2   {
3      return (x > y) ? x : y;
4   }
```

Postoje tri mesta gde se u funkciji `max()` koriste određeni tipovi: parametri x i y, i povratna vrednost funkcije moraju biti celi brojevi. Generičku funkciju je veoma jednostavno napisati zamenom ovih specifičnih tipova generičkim tipom. U ovom slučaju, pošto postoji samo jedan tip koji treba zameniti (`int`), potreban je samo jedan generički tip. Naziv generičkog tipa može biti bilo šta osim ključnih reči jezika C++. Međutim, u C++-u je uobičajeno da imena generičkih tipova budu jednoslovne oznake (kao, na primer, slovo T – skraćeno od "Tip").

Sledi nova funkcija sa generičkim tipom:

```
1   T max(T x, T y)
2   {
3      return (x > y) ? x : y;
4   }
```

Da bi ovakav kod radio, kompajleru se moraju saopštiti dve stvari: prvo, da je ovo definicija generičke funkcije (šablona), i drugo, da je T generički tip. Ove stvari se mogu učiniti u jednoj liniji, koristeći ono što se u jeziku C++ naziva deklaracija parametra šablona, koja se sastoji iz nekoliko delova. Prvi je ključna reč `template`: ovo govori kompajleru da će ono što sledi biti lista generičkih tipova

šablona. Svi generički tipovi se stavljaju između znakova < i > (nalik na zagrade). Kako bi se kreirao generički tip, navodi se ključna reč `typename` ili `class`. U ovom kontekstu ne postoji razlika između ove dve ključne reči, pa je jednostavno stvar vlastitog izbora koja od njih će biti korišćena. Ako se koristi ključna reč `class`, tip koji je prosleđen ne mora biti klasa (može biti i ugrađeni tip, pokazivač ili bilo šta drugo što se podudara). Iza neke od ove dve ključne reči sledi naziv generičkog tipa (najčešće slovo "T").

```
template <typename T>
T max(T x, T y)
{
   return (x > y) ? x : y;
}
```

Ako generička funkcija koristi više generičkih tipova, oni se takođe mogu definisati unutar "zagrada" <> tako što će biti razdvojeni zarezom:

```
template <typename T1, typename T2>
```

Za klase koje koriste više od jednog generičkog tipa, uobičajena imena su T1, T2, itd. ili druga velika slova, kao što je S, U, itd.

S obzirom da tip argumenta koji se prosleđuje može biti klasa, bilo bi bolje parametre i povratne vrednosti generičkih funkcija definisati kao reference (u velikom broju slučajeva i kao konstantne reference).

```
template <typename T>
T& max(T& x, T& y)
{
   return (x > y) ? x : y;
}
```

Korišćenje generičkih funkcija

Korišćenje generičkih funkcija je izuzetno jednostavno – mogu se koristiti kao i bilo koja druga funkcija. Sledi kompletan program koji koristi šablonsku funkciju:

```
#include <iostream>
int main()
{
   int i1 = 3, i2 = 7;
   int i = max(i1, i2);            // Vraća 7
   std::cout << i << '\n';

   double d1 = 6.34, d2 = 18.523;
   double d = max(d1, d2);         // Vraća 18.523
   std::cout << d << '\n';

   char ch1 = 'a', ch2 = '6';
```

```
13      char ch = max(ch1, ch2);        // Vraća 'a'
14      std::cout << ch << '\n';
15  }
```

Izlaz iz prethodnog koda će biti:

```
7
18.523
a
```

Sva tri poziva funkcije `max()` imaju argumente različitih tipova. Pošto je funkcija pozvana sa tri različita tipa, kompajler će koristiti definiciju šablonske funkcije da bi kreirao tri različite verzije ove funkcije: jednu sa parametrima tipa `int`, jednu sa parametrima tipa `double` i jednu sa parametrima tipa `char`.

Kao što se može videti, generičke funkcije mogu uštedeti vreme, jer je dovoljno napisati samo jednu funkciju a ona će raditi sa različitim tipovima podataka. Generičke funkcije znatno olakšavaju održavanje koda, jer se količina dupliranog koda značajno smanjuje. Na kraju, generičke funkcije su i sigurnije, jer nema potrebe za kopiranjem funkcija i ručnom promenom tipova gde god se javi potreba za funkcijom sa novim tipom podataka.

Generičke funkcije imaju i nekoliko nedostataka. Prvo, pronalaženje grešaka u kodu generičkih funkcija je dosta teže u odnosu na regularne funkcije. Drugo, generičke funkcije mogu povećati vreme kompajliranja i veličinu koda, jer se pojedinačne verzije funkcija mogu ponovo kompajlirati u mnogim datotekama. Ovo se, međutim, relativno jednostavno rešava upotrebom nestandardne pretprocesorske direktive `#pragma once` na početku fajla u kome je generička funkcija definisana.

Navedeni nedostaci su prilično zanemarljivi u poređenju sa prednostima i fleksibilnošću koje generičke funkcije, kao alat za programiranje, donose.

Operatori, pozivanje funkcija i generičke funkcije

Korisno je pogledati kako su generičke funkcije implementirane u jeziku C++. Kompajler ne kompajlira generičku funkciju direktno. Umesto toga, u vreme kompajliranja, kada se kompajler susretne sa pozivom generičke funkcije, on zamenjuje generičku funkciju sa parametrima generičkog tipa funkcijom sa stvarnim tipovima. Funkcija sa stvarnim tipovima se zove instanca generičke funkcije.

Generičke funkcije će raditi i sa ugrađenim tipovima (npr. `char`, `int`, `double`, itd.) i sa klasama, ali sa jednim ograničenjem. Kada kompajler kompajlira instancu

generičke funkcije, kompajlira je kao normalnu funkciju. U normalnoj funkciji, svi operatori ili pozivi funkcija sa generičkim tipovima moraju biti definisani. Prema tome, svi operatori ili pozivi funkcija u generičkoj funkciji moraju biti definisani za sve tipove za koje je generička funkcija instancirana.

Sledi primer jednostavne klase.

```
class Cents
{
private:
   int m_cents;
public:
   Cents(int cents) : m_cents(cents)
   {
   }
};
```

Jezik C++ će prilikom kompajliranja koda koji sadrži funkciju `max()` sa parametrima tipa `Cents` napraviti jednu njenu instancu.

```
template <typename T>
T& max(T& x, T& y)
{
   return (x > y) ? x : y;
}

class Cents
{
private:
   int m_cents;
public:
   Cents(int cents) : m_cents(cents)
   {
   }
};

int main()
{
   Cents nickle(5);
   Cents dime(10);

   Cents bigger = max(nickle, dime);

   return 0;
}
```

Instanca generičke funkcije `max()` će izgledati ovako:

```
Cents& max(Cents &x, Cents &y)
{
    return (x > y) ? x : y;
}
```

Kompajler će zatim pokušati da kompajlira ovu funkciju. Međutim, kompajler neće znati kako da izračuna vrednost izraza `x > y`! Kao posledica, kompajler će prijaviti grešku.

Kako bi se rešio ovaj problem, jednostavno treba preklopiti operator `>` za svaku klasu čiji se objekti prosleđuju funkciji `max()` kao argumenti.

```
class Cents
{
private:
   int m_cents;
public:
   Cents(int cents) : m_cents(cents) { }

   bool operator>( const Cents &other)
   {
      return (m_cents > other.m_cents);
   }
};
```

Sada će jezik C++ znati kako da uporedi promenljive x i y u izrazu `x > y`, u slučaju kada su x i y objekti klase `Cents`. Kao rezultat, funkcija `max()` će raditi sa dva objekta tipa `Cents`.

Primer funkcije za izračunavanje srednje vrednosti

Sledeća generička funkcija se koristi za izračunavaće srednje vrednosti svih objekata u nizu:

```
template <class T>
T average(T *array, int length)
{
   T sum = 0;
   for (int count=0; count < length; ++count)
      sum += array[count];

   sum /= length;
   return sum;
}

int main()
{
   int array1[] = { 5, 3, 2, 1, 4 };
   std::cout << average(array1, 5) << '\n';

   double array2[] = { 3.12, 3.45, 9.23, 6.34 };
   std::cout << average(array2, 4) << '\n';

   return 0;
}
```

Izlaz iz ovog koda će biti:

```
3
5.535
```

Kao što se vidi, ovaj primer radi sa ugrađenim tipovima.

Međutim, ako bi ovakva funkcija bila pozvana sa nizom objekata tipa `Cents` kao argumentom, nedostajala bi još dva operatora u klasi `Cents`: +=, /=.

```
class Cents
{
private:
   int m_cents;
public:
   Cents(int cents) : m_cents(cents) { }

   bool operator>(const Cents &other)
   {
      return (m_cents > other.m_cents);
   }

   Cents& operator+=(Cents cents)
   {
      m_cents += cents.m_cents;
      return *this;
   }

   Cents& operator/=(int value)
   {
      m_cents /= value;
```

```
22          return *this;
23      }
24  };
25
26  template <class T>
27  T average(T *array, int length)
28  {
29      T sum = 0;
30      for (int count=0; count < length; ++count)
31          sum += array[count];
32
33      sum /= length;
34      return sum;
35  }
36
37  int main()
38  {
39      Cents array3[] = { Cents(5), Cents(10), Cents(15), Cents(14) };
40      Cents averageValue = average(array3, 4);
41
42      return 0;
43  }
```

Generičke klase u jeziku C++

Kao deo generičkog koncepta programiranja, do sada su prikazane samo generičke funkcije. One omogućavaju generalizaciju funkcija kako bi bile sposobne da rade sa različitim vrstama podataka. Međutim, one predstavljaju samo deo generičkog programiranja.

Generičke klase (šabloni) i kontejnerske klase

U okviru predavanja o osnovama objektno orijentisanog programiranja, pokazano je kako se pomoću klasa može napraviti dinamički niz sa adekvatnom kontrolom pristupa članovima niza. Sledi jednostavan primer takve klase.

```
1   class IntArray
2   {
3   private:
4       int m_length;
5       int *m_data;
6
7   public:
8       IntArray()
9       {
10          m_length = 0;
11          m_data = null;
12      }
13
14      IntArray(int length)
```

```
15      {
16          m_data = new int[length];
17          m_length = length;
18      }
19
20      ~IntArray()
21      {
22          delete[] m_data;
23      }
24
25      void Erase()
26      {
27          delete[] m_data;
28          m_data = null;
29          m_length = 0;
30      }
31
32      int& operator[](int index) { return m_data[index]; }
33
34      int getLength() { return m_length; }
35  };
```

Ova klasa pruža jednostavan način za kreiranje dinamičkih nizova celih brojeva. Međutim, ukoliko bi se ukazala potreba za sličnom klasom koja bi se koristila kao dinamički niz za promenljive tipa `double`, koristeći tradicionalne metode programiranja, neophodno bi bilo napraviti potpuno novu klasu. Sledi primer takve klase.

```
class DoubleArray
{
private:
   int m_length;
   double *m_data;

public:
   DoubleArray()
   {
      m_length = 0;
      m_data = null;
   }

   DoubleArray(int length)
   {
      m_data = new double[length];
      m_length = length;
   }

   ~DoubleArray()
   {
      delete[] m_data;
   }

   void Erase()
   {
      delete[] m_data;
      m_data = nullptr;
      m_length = 0;
   }

   double & operator[](int index)
   {
      return m_data[index];
   }

   int getLength() { return m_length; }
};
```

Analizom prethodne dve klase može se primetiti da su one skoro identične. Zapravo, jedina suštinska razlika jeste tip podataka koji klasa čuva (`int` ili `double`). Ovo već intuitivno navodi na razmišljanje kako bi mogle da izgledaju generičke klase, koje omogućavaju stvaranje klasa koje nisu vezane za određeni tip podataka.

Pisanje generičkih klasa je slično pisanju generičkih funkcija. Sledi primer generičke klase koja zamenjuje prethodne dve klase.

```
template <class T>//Template, korisnik će definisati tip podataka T
class Array
{
private:
   int m_length; // Dužina niza
   T *m_data;    // Pokazivač na dinamički alocirani niz

public:
   Array()
   {
      m_length = 0;
      m_data = nullptr;
   }
   Array(int length)
   {
      m_data = new T[length];
      m_length = length;
   }
   ~Array()
   {
      delete[] m_data;
   }
   void Erase()
   {
      delete[] m_data;
      m_data = nullptr;
      m_length = 0;
   }
   T& operator[](int index)
   {
      return m_data[index];
   }
   int getLength();
   // Funkcija getLength() je definisana izvan deklaracije klase
};

template <class T> // Definicija funkcije getLength() izvan klase
int Array<T>::getLength() // Array<T> je ime klase
{
   return m_length;
}
```

Kao što se može videti, ova klasa je skoro identična prethodno definisanim klasama `IntArray` i `DoubleArray`, osim što je dodata deklaracija `template` i promenjen tip podataka koje klasa čuva iz `int` (ili `double`) u generički tip `T`.

Funkcija `getLength()` je definisana izvan deklaracije klase. Ovo nije neophodno, ali je korisno znati kako radi. Svaka funkcija članica generičke klase definisana izvan deklaracije klase zahteva svoju `template` deklaraciju. Takođe, naziv generičke klase niza je `Array<T>`, a ne samo `Array`. Naziv `Array` bi se odnosio na negeneričku verziju klase `Array`.

Sledi kratki primer sa upotrebom prethodno definisane generičke klase.

```
#include <iostream>
#include "Array.h"

int main()
{
   Array<int> intArray(12);
   Array<double> doubleArray(12);

   for (int count = 0; count < intArray.getLength(); ++count)
   {
      intArray[count] = count;
      doubleArray[count] = count + 0.5;
   }

   for (int count = intArray.getLength()-1; count >= 0; --count)
      std::cout << intArray[count] << "\t"
                << doubleArray[count] << '\n';

   return 0;
}
```

Ovaj program će štampati sledeći izlaz:

```
11      11.5
10      10.5
9        9.5
8        8.5
7        7.5
6        6.5
5        5.5
4        4.5
3        3.5
2        2.5
1        1.5
0        0.5
```

Generičke klase se instanciraju na isti način kao i generičke funkcije - kompajler generiše kopiju po potrebi, a generički parametar klase zamenjuje se stvarnim tipom podataka koji je korisniku potreban, a zatim kompajlira tu kopiju. Ako se nikada ne koristi generička klasa, kompajler je nikada neće ni kompajlirati.

Generičke klase su idealne za implementaciju kontejnerskih klasa, jer je veoma poželjno da kontejneri rade sa različitim tipovima podataka. Generičke klase upravo to omogućavaju bez dupliranja koda. Iako je sintaksa pomalo neobična, a otklanjanje grešaka može biti veoma komplikovano, generičke klase su ipak jedna od najboljih i najkorisnijih funkcionalnosti jezika C++.

Generičke klase pored generičkih tipova podataka mogu da imaju još jednu vrstu parametra poznatih pod imenom "`expression parameters`" (doslovni prevod bi bio parametri izraza).

Ovi parametri predstavljaju poseban tip parametara koji ne zamenjuje tip, već zamenjuje vrednost. Parametar izraza može biti:

- Vrednost celobrojnog ili nabrojivog tipa,
- Pokazivač ili referenca na objekat klase,
- Pokazivač ili referenca na funkciju,
- Pokazivač ili referenca na funkciju članicu klase.

U sledećem primeru je definisana generička klasa za statički niz u kojoj se koristi i generički tip i parametar izraza. Generički tip definiše tip podataka koje statički niz čuva, a parametar izraza definiše veličinu statičkog niza.

```
1   #include <iostream>
2
3   template <class T, int size> // size je "expression parameter"
4   class StaticArray
5   {
6   private:
7      // expression parameter se koristi za definisanje veličine niza
8      T m_array[size];
9
10  public:
11     T* getArray();
12
13     T& operator[](int index)
14     {
15        return m_array[index];
16     }
17  };
18
19  // Showing how a function for a class with an expression parameter
20  is defined outside of the class
21  template <class T, int size>
22  T* StaticArray<T, size>::getArray()
23  {
24      return m_array;
25  }
```

```
int main()
{
   // declare an integer array with room for 12 integers
   StaticArray<int, 12> intArray;

   // Fill it up in order, then print it backwards
   for (int count=0; count < 12; ++count)
      intArray[count] = count;

   for (int count=11; count >= 0; --count)
      std::cout << intArray[count] << " ";
   std::cout << '\n';

   // declare a double buffer with room for 4 doubles
   StaticArray<double, 4> doubleArray;

   for (int count=0; count < 4; ++count)
      doubleArray[count] = 4.4 + 0.1*count;

   for (int count=0; count < 4; ++count)
      std::cout << doubleArray[count] << ' ';

   return 0;
}
```

Ovaj program će štampati:

```
11 10 9 8 7 6 5 4 3 2 1 0
4.4 4.5 4.6 4.7
```

Generičke klase u standardnoj biblioteci

Sada kada su obrađene generičke klase (template klase, šablonske klase ili šabloni), trebalo bi da je jasno šta predstavlja deklaracija `std::vector<int>`. `std` je naziv prostora imena, a `vector<int>` je zapravo generička klasa u kojoj se generički parametar zamenjuje celobrojnim tipom. Standardna biblioteka sadrži veći broj predefinisanih generičkih klasa koje se mogu jednostavno koristiti.

Prethodno prikazani primer generičke klase `StaticArray` je sličan generičkoj klasi `std::array` koja se takođe nalazi u standardnoj biblioteci jezika C++.

14. Standard Template Library - STL

STL kontejneri

Standard Template Library (STL) je kolekcija klasa koja obezbeđuje kontejnere, algoritme i iteratore. Veoma korisna stvar u jeziku C++ je to što se kontejneri iz standardne biblioteke mogu koristiti prilično jednostavno, bez potrebe za pisanjem i debagiranjem novih kontejnerskih klasa. STL biblioteka pruža veoma efikasne verzije kontejnerskih klasa. Kao nedostatak, može se navesti složenost STL-a, čije korišćenje može biti malo komplikovano s obzirom da je biblioteka generička.

Daleko najčešće korišćena funkcionalnost STL biblioteke su STL kontejnerske klase. STL sadrži veći broj kontejnerskih klasa koje se mogu koristiti u različitim situacijama. Uopšteno govoreći, kontejnerske klase se mogu svrstati u tri osnovne kategorije: sekvencijalni kontejneri, asocijativni kontejneri i adaptivni kontejneri.

Sekvencijalni kontejneri

Sekvencijalni kontejneri su kontejnerske klase koje održavaju redosled elemenata u kontejneru. Kod sekvencijalnih kontejnera korisnik definiše poziciju u kontejneru gde želi da smesti određeni element. Najčešći primer sekvencijalnog kontejnera je klasa `Array`: ako se nekoliko elemenata smesti u ovakav kontejner, biće poređani u redosledu kojim su smeštani.

Standardom C++ 11 definisano je šest sekvencijalnih kontejnera u biblioteci STL: `std::vector`, `std::deque`, `std::array`, `std::list`, `std::forward_list`, i `std::basic_string`.

Kada se kaže vektor, obično se misli na vektor kao veličinu koja je definisana pravcem, smerom i intenzitetom. Međutim, naziv klase `vector` u STL-u nema nikakve veze sa tim. Klasa `vector` je dinamički niz koji je sposoban da se uvećava kako bi bio u mogućnosti da prihvati nove elemente. Klasa `vector` omogućava pristup elementima preko indeksnog operatora [ ], kao i smeštanje i uklanjanje elemenata sa kraja kontejnera.

Sledeći program ubacuje šest brojeva u vektor i koristi preklopljeni indeksni operator [ ] za pristup elementima kako bi bili odštampani na konzoli.

```
#include <vector>
#include <iostream>
int main()
{
   using namespace std;

   vector<int> vect;
   for (int nCount=0; nCount < 6; nCount++)
      vect.push_back(10 - nCount); // Dodavanje na kraju niza

   for (int nIndex=0; nIndex < vect.size(); nIndex++)
      cout << vect[nIndex] << " ";

   cout << endl;
}
```

Ovaj program će štampati sledeći izlaz:

```
10 9 8 7 6 5
```

Klasa `deque` (čita se dek) je klasa koja predstavlja niz sa dva kraja. Klasa je implementirana kao dinamički niz koji može da se proširuje na oba kraja.

```
#include <iostream>
#include <deque>
int main()
{
   using namespace std;

   deque<int> deq;
   for (int nCount=0; nCount < 3; nCount++)
   {
      deq.push_back(nCount); // Dodavanje na kraju niza
      deq.push_front(10 - nCount); // Dodavanje na početku niza
   }

   for (int nIndex=0; nIndex < deq.size(); nIndex++)
      cout << deq[nIndex] << " ";

   cout << endl;
}
```

Ovaj program će štampati sledeći izlaz:

```
8 9 10 0 1 2
```

Klasa `list` je posebna vrsta sekvencijalnih kontejnera koja je realizovana kao dvostruko povezana lista, gde svaki element u kontejneru sadrži pokazivače koji pokazuju na sledeći i prethodni element liste. Liste pružaju pristup elementima na početku i na kraju liste. Ako je potrebno pronaći vrednost negde u listi,

neophodno je početi na jednom kraju i "kretati se po listi" dok se ne dođe do elementa koji je tražen. Za kretanje kroz listu se najčešće koriste iteratori.

Klasa `string` se uglavnom ne posmatra kao sekvencijalni kontejner, ali ona to u suštini jeste, jer se može posmatrati kao `vektor` sa elementima tipa `char`.

Asocijativni kontejneri

Asocijativni kontejneri su kontejneri koji automatski sortiraju elemente prilikom njihovog umetanja u kontejner. Podrazumevano je da asocijativni kontejneri upoređuju elemente pomoću operatora <.

Klasa `set` je kontejner koji čuva jedinstvene elemente, pri čemu je onemogućena pojava duplikata elemenata. Elementi su sortirani prema njihovim vrednostima.

Klasa `multiset` je skup gde su dozvoljeni duplikati elementa.

Klasa `map` (koja se naziva i asocijativni niz) je skup u kome je svaki element par koji sadrži ključ i vrednost. Ključ se koristi za sortiranje i indeksiranje podataka i mora biti jedinstven, a vrednost je zapravo stvarni podatak koji se čuva.

Klasa `multimap` (koja se takođe naziva i rečnik) je mapa koja dozvoljava duplikate ključeva. Rečnici u realnom životu su analogija klasi `multimap`: ključ je reč, a vrednost je značenje reči. Svi ključevi su sortirani u rastućem redosledu. Podaci (vrednosti) iz kontejnera se mogu uzimati po ključu (ključ praktično predstavlja indeksiranje pri čemu tip “indeksa“ ne mora biti broj). S obzirom da neke reči mogu imati višestruka značenja, za rečnik je analogna klasa `multimap` a ne `map`.

Adaptivni kontejneri

Adaptivni kontejneri su posebni predefinisani kontejneri koji su prilagođeni specifičnim upotrebama. Zanimljivo je da se kod njih može izabrati koji sekvencijalni kontejner treba da koriste.

Klasa `stack` je kontejner u kome elementi rade u kontekstu LIFO (Last In – First Out), gde se elementi dodaju na kraju i uklanjaju sa kraja kontejnera. Podrazumevano je da stekovi koriste klasu `deque` kao njihov sekvencijalni kontejner, ali takođe mogu koristiti i kontejnere `vector` ili `list`.

Klasa `que` je kontejner u kome elementi funkcionišu u kontekstu FIFO (First In – First Out), gde se elementi ubacuju (potiskuju) na zadnju stranu kontejnera a uklanjaju (ispadaju) s prednje strane. Oni podrazumevano koriste klasu `deque` za smeštanje podataka, ali takođe mogu koristiti i klasu `list`.

Kontejnerska klasa `priority_que` je zapravo klasa `que` u kojoj su elementi sortirani (pomoću operatora <). Kada se element dodaje, smešta se na odgovarajuću poziciju, tako da kontejner ostane sortiran.

STL iteratori

Iterator je objekat koji se može kretati (iterisati) kroz kontejner bez potrebe da korisnik zna kako je kontejner implementiran. Kod mnogih klasa (posebno listi i asocijativnih kontejnera), iteratori su primarni način pristupa elementima ovih klasa.

Iterator predstavlja pokazivač na dati element u kontejneru, sa skupom preklopljenih operatora koji obezbeđuje skup dobro definisanih funkcija:

- Operator * - Dereferenciranje iteratora, vraća element na koji iterator trenutno pokazuje.
- Operator ++ - Pomera iterator na sledeći element u kontejneru. Većina iteratora takođe obezbeđuje operator -- koji pomera iterator na prethodni element.
- Operator == i Operator != - Osnovni operatori upoređivanja za utvrđivanje da li dva iteratora pokazuju na isti element. Da bi se uporedile vrednosti na koje pokazuju dva iteratora, prvo ih treba dereferencirati, a zatim primeniti operator poređenja.
- Operator = - Dodela vrednosti iteratoru (obično početni ili krajnji element kontejnera). Kako bi se dodelila vrednost elementu na koji iterator pokazuje, prvo treba dereferencirati iterator, a zatim koristiti operator dodele vrednosti.

Svaki kontejner sadrži četiri osnovne funkcije članice koje se koriste zajedno sa operatorom =:

- `begin()` - vraća iterator koji predstavlja početni element u kontejneru.
- `end()` - vraća iterator koji predstavlja element koji se nalazi neposredno iza poslednjeg u kontejneru.
- `cbegin()` - vraća `const` iterator (samo za čitanje) koji predstavlja početni element u kontejneru.
- `cend()` - vraća `const` iterator (samo za čitanje) koji predstavlja element koji se nalazi neposredno iza poslednjeg u kontejneru.

Možda izgleda čudno što `end()` ne pokazuje na poslednji element u listi, ali to je tako pre svega kako bi se lako omogućilo ciklično ponavljanje u petljama:

iterisanje po elemenata se nastavlja dok iterator ne dobije vrednost koju vraća funkcija `end()`.

Svi kontejneri obezbeđuju najmanje dve vrste iteratora:

- `container::iterator`, koji omogućava i čitanje i pisanje,
- `container::const_iterator`, koji obezbeđuje samo čitanje.

Sledi nekoliko primera korišćenja iteratora.

Kretanje kroz kontejner `vector`

```
1   #include <iostream>
2   #include <vector>
3   using namespace std;
4
5   int main()
6   {
7      vector<int> vect;
8      for (int nCount = 0; nCount < 6; nCount++)
9         vect.push_back(nCount);
10
11     vector<int>::const_iterator it; // Iterator za čitanje
12     it = vect.begin(); // Iterator pokazuje na početak niza
13     while (it != vect.end()) // Radi dok se ne dostigne kraj niza
14     {
15        cout << *it << " "; // Vrednost koju pokazuje iterator
16        ++it; // Pomeranje iteratora na sledeći član niza
17     }
18
19     cout << endl;
20  }
```

Ovaj kod štampa sledeći izlaz:

```
0 1 2 3 4 5
```

Kretanje kroz kontejner `list`

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

int main()
{
   list<int> li;
   for (int nCount = 0; nCount < 6; nCount++)
      li.push_back(nCount);

   list<int>::const_iterator it; // Iterator za čitanje
   it = li.begin(); // Iterator pokazuje na početak liste
   while (it != li.end()) // Radi dok se ne dostigne kraj liste
   {
      cout << *it << " "; // Vrednost koju pokazuje iterator
      ++it; // Pomeranje iteratora na sledeći član liste
   }

   cout << endl;
}
```

Ovaj kod štampa identičan izlaz:

```
0 1 2 3 4 5
```

Kod je skoro isti kao i u prethodnom slučaju, iako su implementacije vektora i liste sasvim različite.

Kretanje kroz kontejner `set`

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

int main()
{
   set<int> myset;
   myset.insert(7);
   myset.insert(2);
   myset.insert(-6);
   myset.insert(8);
   myset.insert(1);
   myset.insert(-4);

   set<int>::const_iterator it; // Iterator za čitanje
   it = myset.begin(); // Iterator pokazuje na početak seta
   while (it != myset.end()) // Radi dok se ne dostigne kraj seta
   {
      cout << *it << " "; // Vrednost koju pokazuje iterator
      ++it; // Pomeranje iteratora na sledeći član seta
   }

   cout << endl;
}
```

Ovaj kod štampa sledeći izlaz:

```
-6 -4 1 2 7 8
```

Kretanje kroz kontejner map

Ovaj kontejner je malo drugačiji. Kontejneri map i multimap prihvataju elemente koji predstavljaju parove ključ/vrednost (definisanih kao std::pair). Postoji pomoćna funkcija make_pair() za jednostavno stvaranje parova. std::pair omogućava pristup elementima para preko članova first i second. Član first predstavlja ključ, a second vrednost.

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

int main()
{
   map<int, string> mymap;
   mymap.insert(make_pair(4, "apple"));
   mymap.insert(make_pair(2, "orange"));
   mymap.insert(make_pair(1, "banana"));
   mymap.insert(make_pair(3, "grapes"));
   mymap.insert(make_pair(6, "mango"));
   mymap.insert(make_pair(5, "peach"));

   map<int, string>::const_iterator it; // Iterator za čitanje
   it = mymap.begin(); // Iterator pokazuje na početak mape
   while (it != mymap.end()) // Radi dok se ne dostigne kraj mape
   {
      // Vrednosti koje pokazuje iterator
      cout << it->first << "=" << it->second << " ";
      ++it; // Pomeranje iteratora na sledeći par u mapi
   }
   cout << endl;
}
```

Izlaz iz ovog koda će biti:

```
1=banana 2=orange 3=grapes 4=apple 5=peach 6=mango
```

Iteratori pružaju jednostavan način za prolazak kroz elemente kontejnerske klase bez potrebe za razumevanjem načina implementacije same klase. Iteratori moraju biti implementirati na osnovu klase, jer iterator mora da poznaje detalje implementacije kontejnerske klase kojoj pripada. Zato su iteratori uvek vezani za specifične kontejnerske klase.

STL algoritmi

Osim kontejnerskih klasa i iteratora, STL takođe pruža niz generičkih algoritama za rad sa elementima kontejnerskih klasa. Oni omogućavaju pretraživanje, sortiranje, umetanje, preuređivanje, brisanje i kopiranja elemenata kontejnerskih klasa.

Algoritmi su implementirani kao globalne funkcije koje rade pomoću iteratora. To znači da je svaki algoritam implementiran samo jednom, i obično automatski radi sa svim kontejnerima koji obezbeđuju vlastiti set iteratora. Iako je ovo veoma moćno i daje mogućnost da se veoma brzo napiše složeni programski kod, ima i svoje nedostatke: neka kombinacija algoritama i kontejnera možda neće raditi, može prouzrokovati beskonačne petlje ili može raditi sa veoma lošim performansama.

STL pruža dosta različitih algoritama. Ovde će biti prikazani samo neki osnovni algoritmi.

Da bi se omogućilo korišćenje bilo kog STL algoritama, treba u kodu uključiti datoteku zaglavlja "`algorithm.h`".

min_element i max_element

Ova dva algoritma pronalaze minimalni i maksimalni element u kontejnerskoj klasi.

```
1   #include <iostream>
2   #include <list>
3   #include <algorithm>
4   int main()
5   {
6      using namespace std;
7
8      list<int> li;
9      for (int nCount = 0; nCount < 6; nCount++)
10        li.push_back(nCount);
11
12     list<int>::const_iterator it;
13     it = min_element(li.begin(), li.end());
14     cout << *it << " ";
15     it = max_element(li.begin(), li.end());
16     cout << *it << " ";
17
18     cout << endl;
19  }
```

Izlaz je:

```
0 5
```

find i list::insert

U ovom primeru se koristi `find()` algoritam kako bi se pronašla vrednost u klasi list, a zatim funkcija `list::insert()` kako bi se dodala nova vrednost u listi na toj poziciji.

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

int main()
{
   list<int> li;
   for (int nCount = 0; nCount < 6; nCount++)
      li.push_back(nCount);

   list<int>::iterator it;
   it = find(li.begin(), li.end(), 3);
   li.insert(it, 8);

   for (it = li.begin(); it != li.end(); it++)
      cout << *it << " ";

   cout << endl;
}
```

Izlaz je:

```
0 1 2 8 3 4 5
```

sort i reverse

U primeru koji sledi prvo je sortiran objekat kontejnerske klase `vector` a zatim je u njemu obrnut redosled elemenata.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main()
{
   vector<int> vect;
   vect.push_back(7);
   vect.push_back(-3);
   vect.push_back(6);
   vect.push_back(2);
   vect.push_back(-5);
   vect.push_back(0);
   vect.push_back(4);

   sort(vect.begin(), vect.end()); // sortiranje liste

   vector<int>::const_iterator it;
   for (it = vect.begin(); it != vect.end(); it++)
      cout << *it << " ";

   cout << endl;

   reverse(vect.begin(), vect.end()); // okretanje liste

   for (it = vect.begin(); it != vect.end(); it++)
      cout << *it << " ";

   cout << endl;
}
```

Izlaz je:

```
-5 -3 0 2 4 6 7
7 6 4 2 0 -3 -5
```

Iako je ovo samo delić algoritama koje nudi STL, dovoljno je da bi se shvatilo koliko ih je lako koristiti zajedno sa iteratorima i osnovnim kontejnerskim klasama.

15. Uvod u obradu izuzetaka

Uvod u obradu izuzetaka

Izuzeci obezbeđuju način reagovanja na nepredviđene okolnosti nastale u programu (kao što su greške izvršavanja), prenošenjem kontrole programa posebnim funkcijama koje se zovu rukovaoci izuzecima.

Kako bi se izuzetak u programu "uhvatio", deo koda u kome je očekivano da se izuzetak pojavi se stavlja "pod nadzor". Ovo se radi tako što se taj deo koda stavlja u `try` – blok. Kada se izuzetak (nepredviđena okolnost) pojavi unutar tog bloka, "izbacuje" se izuzetak koji prenosi kontrolu delu koda koji se naziva rukovalac izuzetkom. Ako se izuzetak ne izbaci, kod nastavlja normalno da se izvršava i svi rukovaoci se ignorišu.

Izuzeci se izbacuju korišćenjem ključne reči `throw` iz unutrašnjosti bloka `try`. Rukovaoci izuzecima se deklarišu ključnom rečju `catch`, i moraju se definisati odmah nakon `try` bloka:

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

int main () {
   try
   {
      throw 20;
   }
   catch (int e)
   {
      cout << "Pojavio se izuzetak sa brojem: " << e << '\n';
   }
   return 0;
}
```

Kod u kome treba obraditi izuzetak uokviren je `try` blokom. U primeru koji sledi, kod jednostavno izbacuje izuzetak:

```
   throw 20;
```

Izraz `throw` prihvata jedan parametar (u ovom slučaju celobrojna vrednost `20`), koji se prenosi kao argument rukovaocu izuzetkom.

Rukovalac izuzetkom se deklariše ključnom rečju `catch` odmah nakon zatvaranja bloka `try`. Sintaksa za `catch` blok je slična sintaksi regularne funkcije sa jednim parametrom. Tip ovog parametra je veoma značajan, jer se tip argumenta koji je

prosleđen izrazom `throw` upoređuje sa njim, i samo u slučaju da se podudaraju, izuzetak će biti uhvaćen od strane tog rukovaoca.

Višestruki rukovaoci (tj. `catch` izrazi) mogu biti ulančani, svaki sa drugačijim tipom parametara. Izvršiće se samo rukovalac čiji tip parametra odgovara tipu izuzetaka specificiranom u `throw` iskazu koji je izvršen.

Ako se kao parametar rukovaoca stave tri tačke (`...`), takav rukovalac će uhvatiti bilo koji izuzetak bez obzira na tip izuzetka. Ovo se može koristiti kao podrazumevani rukovalac koji hvata sve izuzetke koje nisu uhvatili drugi rukovaoci.

```
try {
  // Kod pod nadzorom
}
catch (int param) { cout << "int izuzetak"; }
catch (char param) { cout << "char izuzetak "; }
catch (...) { cout << "podrazumevani izuzetak"; }
```

U ovom slučaju, poslednji rukovalac bi uhvatio bilo koji izuzetak koji je izbačen čiji tip nije ni `int` ni `char`.

Nakon što je izuzetak obrađen, program nastavlja da se izvršava nakon bloka `try`–`catch`, a ne nakon iskaza `throw`.

Moguće je i ugnezditi `try` blok unutar više spoljašnjih `try` blokova. U ovakvim slučajevima postoji mogućnost da interni blok za hvatanje predaje izuzetak njegovom spoljašnjem nivou. Ovo se radi pomoću izraza `throw;` bez argumenata. Sledi primer.

```
try
{
   try
   {
      // Kod pod nadzorom
   }
   catch (int n)
   {
      throw;
   }
}
catch (...)
{
   cout << "Izuzetak se pojavio!";
}
```

Standardni izuzeci

Standardna biblioteka C++ obezbeđuje osnovnu klasu koja je specijalno dizajnirana za deklarisanje objekata koji se izbacuju kao izuzeci. Ta klasa se zove `std::exception` i definisana je u `<exception>` zaglavlju. Ova klasa ima virtuelnu funkciju članicu `what()` koja vraća sekvencu karaktera koja se završava karakterom `'\0'`. Ova funkcija se može preklopiti u izvedenim klasama kako bi sadržala neku vrstu opisa izuzetka.

```
#include <iostream>
#include <exception>
using namespace std;
class myexception: public exception
{
   virtual const char* what() const throw()
   {
      return "Dogodio se izuzetak!";
   }
} myex;
int main ()
{
   try
   {
      throw myex;
   }
   catch (exception& e)
   {
      cout << e.what() << '\n';
   }
   return 0;
}
```

U prethodnom primeru rukovalac izuzecima prihvata objekte izuzetke po referenci. Takav rukovalac će moći da uhvati izuzetke poput izuzetka `myex`, jer je on objekat klase `myexception` koja je izvedena iz bazne klase `exception`.

Svi izuzeci koje izbacaju komponente standardne C++ biblioteke su izvedeni iz klase `exception`.

Primeri obrade izuzetaka

Primer 1

U sledećem primeru od korisnika se traži unos broja. Ukoliko korisnik unese pozitivan broj, iskaz `if` se neće izvršiti i neće biti izbačen izuzetak, tako da će kvadratni koren broja biti odštampan u konzolnom prozoru.

```
1   #include "math.h" // za funkciju sqrt()
2   #include <iostream>
3
4   int main()
5   {
6      std::cout << "Unesite broj: ";
7      double x;
8      std::cin >> x;
9
10     try
11     {
12        if (x < 0.0)
13           throw "Ne moze se izračunati koren negativnog broja!";
14        std::cout << "Kvadratni koren od " << x << " je "
15                  << sqrt(x) << '\n';
16     }
17     catch (const char* exception)
18     {
19        std::cerr << "Error: " << exception << std::endl;
20     }
21  }
```

Međutim, ako korisnik unese negativni broj, biće izbačen izuzetak tipa `const char*`. Pošto je kontrola programa u okviru bloka `try`, a pronađeno je podudaranje izuzetka sa rukovaocem izuzecima, kontrola se odmah prebacuje na pronađeni rukovalac. Rezultat će biti:

```
Enter a number: -4
Error: Can not take sqrt of negative number
```

Primer 2

Sledeći primer je klasa koja predstavlja dinamički niz u kome se čuvaju celobrojne vrednosti. Klasa izbacuje izuzetak ako se pristupa članu sa indeksom koji je van opsega.

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

class ArrayException: public std::exception
{
private:
   std::string m_error;
public:
   ArrayException(std::string error) : m_error(error) { }
   const char* what() { return m_error.c_str(); }
};

class IntArray
{
private:

   int* m_data;
   int m_length;
public:
   IntArray()
   {
      m_data = 0; m_length = 0;
   }
   IntArray(int size)
   {
      m_data = new int[size];
      m_length = size;
   }
   ~IntArray()
   {
      if(m_data)
      {
         delete [] m_data;
         m_data = 0;
         m_length = 0;
      }
   }
   int getLength() { return m_length; }
   int& operator[](const int index)
   {
      if (index < 0 || index >= getLength())
         throw ArrayException("Invalid index");
      return m_data[index];
   }
```

```
46  };
1   int main()
2   {
3      std::cout << "Unesite velicinu niza: ";
4      int size;
5      std::cin >> size;
6
7      IntArray array(size);
8
9      for (int i = 0; i < size; i++)
10     {
11        std::cout << "Unesite član " << i + 1 << "\t";
12        std::cin >> array[i];
13     }
14
15     std::cout << "Koji član želte da vidite? ";
16     int index;
17     std::cin >> index;
18
19     try
20     {
21        std::cout << array[index] << std::endl;
22     }
23     catch (ArrayException &exception)
24     {
25        std::cout <<"Dogodio se izuzetak: "<< exception.what() <<")";
26     }
27  }
```

Primeri koji su pokazani predstavljaju elementarne primere obrade izuzetaka. Tema obrada izuzetaka je mnogo šira od onoga što je ovde prikazano i prevazilazi obim ove knjige.

16. Ulazno – izlazni tokovi

Ulazno – izlazni tokovi

Funkcije ulaza i izlaza nisu definisane kao sastavni deo jezika C++, već su date kao deo standardne C++ biblioteke, i stoga se nalaze u prostoru imena `std`. Uključivanjem zaglavlja `iostream` omogućen je pristup čitavoj hijerarhiji klasa odgovornih za pružanje ulazno – izlaznih funkcionalnosti.

Tokovi (strimovi)

Reč strim (eng. stream) se veoma često koristi kada se govori o ulazno – izlaznim operacijama u jeziku C++. Tok ili strim predstavlja samo niz znakova kojima se može pristupiti sekvencijalno. Tokom vremena, tok može proizvesti ili prihvatiti potencijalno neograničene količine podataka.

Obično se radi sa dve različite vrste tokova. **Ulazni tokovi** drže podatke koje neki "generator" podataka šalje (npr. tastatura, datoteka, mreža, itd.). Na primer, korisnik može pritisnuti taster na tastaturi u momentu kada program ne očekuje unos. Umesto da ignoriše pritisnuti taster, podatak o tome se unosi u ulazni tok, gde će biti sačuvan sve dok program ne bude spreman da ga preuzme.

Nasuprot tome, **izlazni tokovi** drže izlazne podatke, koje određeni izlazni "uređaj" koristi (monitor, štampač, datoteka, mreža, itd.). Kada se podaci ispisuju na nekom izlaznom uređaju, uređaj možda neće biti spreman da prihvati te podatke. Na primer, štampač se možda još uvek zagreva dok program piše podatke u njegov izlazni tok. Podaci će se nalaziti u izlaznom toku dok štampač ne počne da ih koristi.

Neki "uređaji", kao što su datoteke ili računarske mreže, mogu biti i ulazni i izlazni tokovi.

Lepa stvar kod tokova je da programer mora samo naučiti kako da radi sa tokovima, kako bi čitao i pisao podatke na različitim vrstama uređaja. Korisnik ne mora poznavati detalje o povezivanju toka sa stvarnim uređajima. O tome se stara radno okruženje ili operativni sistem.

Ulaz i izlaz u jeziku C++

Klasa `istream` je osnovna klasa koja se koristi kada se radi o ulaznim tokovima. Kod ulaznih tokova se operator >> koristi za uzimanje vrednosti iz toka. Na primer,

kada korisnik pritisne taster na tastaturi, ASCII kod tastera se stavlja u ulazni tok. Program uzima vrednost iz toka i zatim je može koristiti.

Klasa `ostream` je osnovna klasa koja se koristi kada se radi o izlaznim tokovima. Kod izlaznih tokova se operator << koristi za stavljanje vrednosti u tok. Na primer, program šalje neke vrednosti u izlazni tok, a neki izlazni "uređaj" ih koristi (monitor, štampač, datoteka, mreža).

Klasa `iostream` može da radi i sa ulazom i sa izlazom, omogućavajući dvosmerni ulaz/izlaz.

Standardni tokovi u jeziku C++

Standardni tok je tok koji okruženje obezbeđuje računarskom programu. Jezik C++ dolazi sa četiri predefinisana standardna toka u obliku objekata koji su već podešeni za korišćenje:

- `cin` – objekat klase `istream_withassign` – komunikacioni kanal povezan sa standardnim ulazom (najčešće tastatura),
- `cout` – objekat klase `ostream_withassign` – komunikacioni kanal povezan sa standardnim izlazom (najčešće monitor),
- `cerr` – objekat klase `ostream_withassign` – komunikacioni kanal za standardne greške (najčešće monitor), sa nebaferisanim izlazom,
- `clog` - klasa `ostream_withassign` – komunikacioni kanal za standardne greške (najčešće monitor), sa baferisanim izlazom.

Nebaferisanim izlazom se obično rukuje odmah, dok se baferisani izlaz najčešće čuva i ispisuje kao blok. Tok `clog` se ne koristi često, pa se obično izostavlja iz liste standardnih tokova.

Primer sa tokovima

```
#include <iostream>
#include <cstdlib> // Za funkciju exit()
using namespace std;

int main()
{
   cout << "Unesite pozitivan broj: " << endl;

   int number;
   cin >> number;

   if (number <= 0)
   {
      cerr << "Pogresan unos!" << endl;
      exit(1);
   }

   cout << "Uneli ste broj " << number << endl;

   return 0;
}
```

Unos podataka sa istream

Biblioteka `iostream` je prilično složena. Međutim, ovde će biti pokazane samo osnovne funkcionalnosti ove biblioteke.

Operator >>

Kao što je pokazano u prethodnim primerima, operator >> se koristi za čitanje podataka sa ulaznog toka. Jezik C++ ima unapred definisane operatore >> za sve ugrađene tipove podataka, a kod korisnički definisanih tipova je moguće preklopiti ove operatore, kako bi se koristili za rad sa standardnim tokovima.

Kada se čitaju nizovi karaktera, jedan čest problem sa operatorom >> je kako sprečiti da ulaz prekorači veličinu bafera.

```
char buf[10];
cin >> buf;
```

Ovakav kod će dobro funkcionisati do momenta kada korisnik pokuša da unese niz karaktera koji prevazilazi granicu bafera. Tada će doći do prekoračenja veličine bafera i program će se ponašati neočekivano. Uopšteno govoreći, loše je napraviti bilo kakvu pretpostavku o tome koliko će karaktera korisnik programa uneti.

Jedan od načina za rešavanje ovog problema je korišćenje manipulatora. Manipulator je objekat koji se koristi za modifikovanje toka kada se primenjuje sa operatorima >> ili <<. Jedan manipulator sa koji je korišćen u dosadašnjim primerima je `endl`, koji štampa znak nove linije. U jeziku C++ takođe postoji manipulator poznat kao `setw` (u zaglavlju "`iomanip.h`") koji se može koristiti za ograničavanje broja znakova koji se čitaju iz toka (strima).

```
#include <iomanip.h>
char buf[10];
cin >> setw(10) >> buf;
```

U ovom slučaju će se pročitati samo prvih devet znakova iz strima (ostavljajući prostor za '\0'). Svi preostali karakteri će biti ostavljeni u toku do sledećeg čitanja.

Značajno je napomenuti da operator >> radi sa formatiranim podacima, tj. preskače prazna mesta, tabove i nove redove.

```
#include "iostream"
using namespace std;
int main()
{
    char ch;
    while (cin >> ch)
        cout << ch;

    return 0;
}
```

Ukoliko se sa tastature učita neki tekst sa razmacima, tabovima ili novim redovima, prethodni program će odštampati taj tekst, ali bez navedenih karaktera.

Međutim, često postoji potreba da se ne odbacuju prazna mesta. Klasa `istream` pruža mnoge funkcije koje se mogu iskoristiti za to.

Jedna od najkorisnijih je funkcija `get()`, koja jednostavno uzima karakter iz ulaznog toka. Sledi program nalik prethodnom, ali sa korišćenjem funkcije `get()`.

```
#include "iostream"
using namespace std;
int main()
{
   char ch;
   while (cin.get(ch))
      cout << ch;

   return 0;
}
```

Ovakav program neće ignorisati prazna mesta.

Funkcija `get()` takođe ima svoju verziju koja radi sa nizovima karaktera, u kojoj je neophodno definisati maksimalan broj znakova koji se čitaju.

```
#include "iostream"
using namespace std;
int main()
{
   char strBuf[11];
   cin.get(strBuf, 11);
   cout << strBuf << endl;

   return 0;
}
```

Ako se u prethodnom programu unese tekst:

```
Hello my name is Alex
```

izlaz će biti:

```
Hello my n
```

Jedna važna stvar koju treba napomenuti je da funkcija `get()` ne čita '\n' karakter. Ovo može prouzrokovati neočekivane rezultate.

```
#include "iostream"
using namespace std;
int main()
{
   char strBuf[11];

   cin.get(strBuf, 11);
   cout << strBuf << endl;

   cin.get(strBuf, 11);
   cout << strBuf << endl;

   return 0;
}
```

Ako korisnik unese:

```
Hello!
```

Program će štampati:

```
Hello!
```

i završiće se. Pitanje je: zašto nije tražio još 10 karaktera? Odgovor je zato što je prvi poziv funkcije `get()` pročitao simbol za novu liniju i zaustavio se. Drugi poziv

get() je pogledao ponovo u tok cin i pokušao da pročita iz njega. Međutim, prvi karakter je opet bio nova linija, pa se program odmah zaustavio.

Postoji još jedna funkcija koja se zove getline() koja funkcioniše isto kao get(), ali čita i karakter nova linija.

```
#include "iostream"
using namespace std;
int main()
{
   char strBuf[11];

   cin.getline(strBuf, 11);
   cout << strBuf << endl;

   cin.getline(strBuf, 11);
   cout << strBuf << endl;

   return 0;
}
```

Ovaj kod će se izvršiti kako je očekivano, čak i ako korisnik unese niz karaktera sa novom linijom u njemu.

Ako je potrebno znati koliko je karaktera dobijeno poslednjim pozivom getline(), može se upotrebiti funkcija gcount().

```
#include "iostream"
using namespace std;
int main()
{
   char strBuf[100];
   cin.getline(strBuf, 100);
   cout << strBuf << endl;
   cout << cin.gcount() << " karaktera je procitano" << endl;

   return 0;
}
```

Takođe, postoji i posebna verzija getline(), izvan klase istream, koja se koristi za čitanje u promenljivu tipa std::string. Ova specijalna verzija nije član ni istream ni ostream klase, već se nalazi u zaglavlju <string>. Sledi primer upotrebe:

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

int main()
{
   using namespace std;
   string strBuf;
   getline(cin, strBuf);
   cout << strBuf << endl;

   return 0;
}
```

Postoji još nekoliko korisnih funkcija koje će ovde biti samo navedene:

- `ignore()` – odbacuje prvi karakter u strimu.
- `ignore(int nCount)` – odbacuje prvih nCount znakova u toku (strimu).
- `peek()` – omogućava čitanje karaktera iz toka bez njegovog uklanjanja iz toka.
- `unget()` – vraća poslednji pročitani karakter nazad u tok, tako da ga sledeći poziv može ponovo pročitati.
- `putback(char ch)` – omogućava vraćanje karaktera po sopstvenom izboru nazad u tok, kako bi ga bilo moguće pročitati sledećim pozivom funkcija za čitanje.

Ispisivanje podataka sa `ostream`

Za slanje podataka u izlazni tok koristi se operator <<. Ovaj operator je u jeziku C++ unapred definisan za sve ugrađene tipove podataka, a za korisnički definisane tipove (klase) ga je moguće preklopiti.

Formatiranje

Postoje dva načina za promenu načina formatiranja: pomoću maski (eng. flag) i manipulatora. Maske su promenljive koje se mogu zamisliti kao logičke promenljive, koje se mogu uključiti ili isključiti. Manipulatori su objekti smešteni u tok, i utiču na način na koji stvari ulaze u tok ili izlaze iz njega.

Za uključivanje maski se koristiti funkcija `setf()`, sa odgovarajućom maskom kao parametrom. Na primer, jezik C++ podrazumevano ne štampa oznaku + ispred pozitivnih brojeva. Međutim, korišćenjem maske `std::showpos` može se promeniti ovo ponašanje.

```
std::cout.setf(std::showpos); // Uključivanje maske std::showpos
std::cout << 27 << '\n';
```

Rezultat ovakvog koda će biti:

```
+27
```

Moguće je uključiti više maski istovremeno pomoću operatora OR (|).

```
// Uključivanje maski std::showpos i std::uppercase
std::cout.setf(std::showpos | std::uppercase);
std::cout << 27 << '\n';
```

Za isključivanje maski koristi se funkcija `unsetf()`.

```
std::cout.setf(std::showpos); // Uključivanje maske std::showpos
std::cout << 27 << '\n';
std::cout.unsetf(std::showpos); // Isključivanje maske std::showpos
std::cout << 28 << '\n';
```

Rezultat ovakvog koda će biti:

```
+27
28
```

Postoji još jedan detalj, na prvi pogled neočekivan, kada se koristi funkcija `setf()`. Mnoge maske pripadaju grupama maski. To su grupe maski koje obavljaju slične (ponekad međusobno isključive) opcije formatiranja. Na primer, grupa pod nazivom `basefield` sadrži oznake `oct`, `dec` i `hex`, koja kontroliše način ispisivanja celobrojnih vrednosti. Podrazumevano je postavljena maska `dec`.

```
std::cout.setf(std::hex);
std::cout << 27 << '\n';
```

Prethodni kod neće odštampati broj u heksadecimalnom formatu, što je na prvi pogled bilo očekivano da će uraditi. Razlog tome je što funkcija `setf()` samo uključuje ili isključuje određenu masku, ali nema mehanizam da prepozna da li je neka druga maska uključena ili ne. Prema tome, u prethodnom primeru, iako je maska `std::hex` uključena, maska `std::dec` je takođe bila uključena. S obzirom da ona ima izvesnu prednost, broj je štampan u decimalnom formatu. Kako bi se broj odštampao u heksadecimalnom formatu, treba prvo isključiti masku `std::dec`, a zatim uključiti `std::hex`.

```
std::cout.unsetf(std::dec);
std::cout.setf(std::hex);
std::cout << 27 << '\n';
```

Sada će izlaz biti:

```
1b
```

Drugi način je upotreba drugog oblika funkcije `setf()`, koja ima dva parametra: prvi parametar je maska koju treba postaviti, a drugi je grupa kojoj maska pripada. Kada se koristi ovaj oblik `setf()`, sve maske koje pripadaju grupi se isključuju a uključuje se samo ona koja je navedena u pozivu funkcije.

```
1  // Uključivanje maske std::hex i isključivanje grupe std::basefield
2  std::cout.setf(std::hex, std::basefield);
3  std::cout << 27 << '\n';
```

Izlaz će takođe biti:

```
1b
```

Kao što je već napomenuto, u jeziku C++ postoji još jedna mogućnost promene načina formatiranja, pomoću manipulatora. Manipulatori imaju mogućnost da uključe i isključe odgovarajuće maske. Evo primera korišćenja nekih manipulatora:

```
1  std::cout << std::hex << 27 << '\n';
2  std::cout << 28 << '\n';
3  std::cout << std::dec << 29 << '\n';
```

Izlaz iz ovog primera je:

```
1b
1c
29
```

Uopšteno govoreći, upotreba manipulatora je mnogo lakša od uključivanja i isključivanja maski. Mnoge opcije je moguće postaviti na oba načina. Međutim, postoje i opcije koje je moguće postaviti samo preko maski ili samo preko manipulatora.

Ovde su navedene samo neke od osnovnih opcija vezanih za formatiranje. Postoji veliki broj opcija, koji izlaze iz okvira ove knjige. Primere upotrebe je moguće pronaći na internet sajtovima koji se bave temom učenja programiranja na jeziku C++.

Osnove ulazno – izlaznih operacija sa datotekama

Ulazno – izlazne operacije sa datotekama funkcionišu veoma slično kao i druge ulazno – izlazne operacije. Postoje tri osnovne datoteke zaglavlja u kojima se nalaze ulazno – izlazne klase u jeziku C++:

1. `ifstream` (izvedeno iz `istream`),

2. `ofstream` (izvedeno iz `ostream`),
3. `fstream` (izvedeno iz `iostream`).

Ove klase, respektivno, omogućavaju: pisanje podataka, čitanje podataka i čitanje i pisanje podataka istovremeno. Za korišćenje klasa za čitanje iz datoteke i pisanje u datoteku potrebno je uključiti zaglavlje `<fstream>`.

Za razliku od `cout`, `cin`, `cerr` i `clog` tokova, koji su već spremni za upotrebu, tokovi za datoteke se moraju eksplicitno podesiti kako bi ih bilo moguće koristiti. Kako bi se datoteka otvorila za čitanje i/ili pisanje, jednostavno treba instancirati objekat odgovarajuće klase za rad sa datotekama, a naziv datoteke treba proslediti konstruktoru klase kao argument. Zatim je moguće koristiti operatore `<<` i `>>` za čitanje iz datoteke ili pisanje u nju. Kada se završi rad sa nekom datotekom, postoje dva načina za njeno zatvaranje: eksplicitnim pozivanjem funkcije `close()` ili se jednostavno dozvoli da objekat za rad sa datotekom izađe iz opsega, pri čemu se poziva destruktor klase unutar koga je takođe realizovano zatvaranje datoteke.

Pisanje u datoteku

Za pisanje u datoteku se koristi klasa `ofstream`.

```
1   #include <fstream>
2   #include <iostream>
3   #include <cstdlib>
4
5   using namespace std;
6
7   int main()
8   {
9      ofstream outf("Sample.dat");
10
11     if (!outf)
12     {
13        cerr << "Problem sa otvaranjem datoteke!" << endl;
14        exit(1);
15     }
16
17     outf << "Ovo je linija 1" << endl;
18     outf << "Ovo je linija 2" << endl;
19
21     return 0;
22
23     // Kada objekat outf izađe iz opsega
24     // njegov destruktor će zatvoriti datoteku
25  }
```

Nakon izvršavanja prethodnog koda, u direktorijumu u kome se nalazi projekat, trebalo bi da postoji datoteka sa nazivom "Sample.dat" sa sadržajem koji odgovara unetom tekstu.

Za upisivanje pojedinačnih karaktera u datoteku može se koristiti funkciju put().

Čitanje iz datoteke

U narednom primeru biće pročitan sadržaj datoteke koja je zapisana u prethodnom primeru. Objekat klase ifstream će prilikom čitanja vratiti vrednost 0 ako se dođe do kraja datoteke (tj. do End of File). Ova činjenica se često koristi kako bi se odredilo dokle treba čitati.

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

int main()
{
   ifstream inf("Sample.dat");
   if (!inf)
   {
      cerr << "Problem sa otvaranjem datoteke!" << endl;
      exit(1);
   }
   while (inf)
   {
      std::string strInput;
      inf >> strInput;
      cout << strInput << endl;
   }

   return 0;

   // Kada objekat inf izađe iz opsega
   // njegov destruktor će zatvoriti datoteku
}
```

Izlaz će biti:

```
Ovo
je
linija
1
Ovo
je
linija
2
```

Izlaz iz programa ne odgovara u potpunosti očekivanjima (tekst ne izgleda onako kako je prethodno zapisan). Razlog tome je činjenica da operator >> radi sa formatiranim izlazom, i da izostavlja prazna mesta. Za čitanje celih linija moguće je koristiti funkciju `getline()`.

```
#include <fstream>
#include <iostream>
#include <string>
#include <cstdlib>

int main()
{
   using namespace std;

   ifstream inf("Sample.dat");

   if (!inf)
   {
      cerr << "Problem sa otvaranjem datoteke!" << endl;
      exit(1);
   }

   while (inf)
   {
      std::string strInput;
      getline(inf, strInput);
      cout << strInput << endl;
   }

   return 0;

   // Kada objekat inf izađe iz opsega
   // njegov destruktor će zatvoriti datoteku
}
```

Izlaz će sada biti:

```
Ovo je linija 1
Ovo je linija 2
```

Baferovani izlaz

Izlaz u jeziku C++ može biti baferovan. To znači da sve što se stavlja u izlazni tok datoteke ne mora odmah biti zapisano na disk. Umesto toga, nekoliko izlaznih operacija može biti sakupljeno i realizovano zajedno. Ovo se radi uglavnom zbog performansi. Proces zapisivanja bafera na disk se naziva pražnjenje bafera. Jedan od načina da se isprazni bafer je zatvaranje datoteke - sadržaj bafera će biti zapisan na disk, a zatim će se datoteka zatvoriti.

Baferovanje podataka obično nije problem, ali u određenim okolnostima može izazvati komplikacije. Problem može nastati kada podaci postoje u baferu, a izvršavanje programa se iznenadno prekine. U ovim slučajevima se ne izvršavaju destruktori klasa za rad sa datotekama, što znači da se datoteke ne zatvaraju i da se baferi nikada ne isprazne. U ovom slučaju podaci koji su se nalazili u baferu nisu zapisani na disk i trajno su izgubljeni. Zbog toga je veoma značajno da se sve otvorene datoteke eksplicitno zatvore pre nego što se program završi.

Bafer je moguće ručno isprazniti korišćenjem funkcije `ostream::flush()` ili slanjem maske `std::flush` na izlazni tok. Bilo koja od ovih metoda se može koristiti kako bi se osiguralo da se sadržaj bafera odmah zapiše na disk.

Korisno je napomenuti da maska `std::endl` takođe prazni bafer izlaznog toka. Prema tome, prekomerna upotreba `std::endl` može imati negativan uticaj na performanse. Iz tog razloga programeri koji se bave poboljšanjem performansi programa često koriste `'\n'` umesto `std::endl`, kako bi se izbeglo često pražnjenje bafera.

Režimi rada sa datotekama

Ponovnim pokretanjem primera sa pisanjem u datoteku može se primetiti da je originalna datoteka potpuno prepisana svaki put kada se program pokrene. Međutim, konstruktor klase za pisanje u datoteku ima opcioni drugi parametar koji omogućava da se navede informacija o tome kako se datoteka otvara (za čitanje, pisanje, pisanje od početka, nastavljanje na kraj fajla, pisanje u binarnu ili tekstualnu datoteku, itd.). Ovaj parametar se zove mod (način ili režim rada), a maske koje je moguće koristiti se nalaze u klasi `ios`.

Moguće je definisati višestruke maske pomoću operatora OR (`|`). Maske `ios::in` i `ios::out` su podrazumevane za klase `ifstream` i `ofstream`, respetivno. Ako se koristi klasa `fstream` (koja može i čitati iz datoteke i pisati u datoteku), neophodno je eksplicitno definisati šta objekat treba da radi, koristeći `ios::in` i/ili `ios::out`.

Sledi program koji dodaje još dve linije u datoteku "`Sample.dat`" koja je zapisana u jednom od prethodnih primera.

```
#include <cstdlib>
#include <iostream>
#include <fstream>

int main()
{
   using namespace std;

   ofstream outf("Sample.dat", ios::app);

   if (!outf)
   {
      cerr << "Problem sa otvaranjem datoteke!" << endl;
      exit(1);
   }

   outf << "Ovo je linija 3" << endl;
   outf << "Ovo je linija 4" << endl;

      return 0;

   // Kada objekat outf izađe iz opsega
   // njegov destruktor će zatvoriti datoteku
}
```

Sada će datoteka “Sample.dat“ imati sledeći sadržaj:

```
Ovo je linija 1
Ovo je linija 2
Ovo je linija 3
Ovo je linija 4
```

17. Literatura

[1] The C++ Programming Language (4th Edition), Bjarne Stroustrup, Addison-Wesley, 2013.
[2] Misliti na jeziku C++, prevod drugog izdanja, Bruce Eckel, Mikroknjiga 2003.
[3] Rešeni zadaci iz programskog jezika C++ (peto izdanje), Laslo Kraus, Akademska misao, 2016.
[4] www.learncpp.com

CIP - Каталогизација у публикацији - Народна библиотека Србије, Београд

004.432.2C++(075.8)(076)

004.42.045(075.8)(076)

ИСАИЛОВИЋ, Велибор, 1980-

Збирка задатака из програмског језика C++ / Велибор Исаиловић. - Крагујевац : Факултет инжењерских наука Универзитета, 2021 (Крагујевац : ИнтерПринт). - 130 стр. : илустр. ; 30 cm

Тираж 130. - Библиографија: стр. 130.

ISBN 978-86-6335-085-4

a) Програмски језик "C++" - Задаци b) Објектно оријентисано програмирање - Задаци

COBISS.SR-ID 44995849