
nim jezikom jasnim
zikom jasnim jezik
jezikom jasnim jez
nim jezikom jasnim
zikom jasnim jezik
jezikom jasnim jez
nim jezikom jasnim
zikom jasnim jezik
jezikom jasnim jez
nim jezikom jasnim
zikom jasnim jezik
jezikom jasnim jez
im jezikom jasnim
zikom jasnim jezik
jezikom jasnim jez
nim jezikom jasnim
zikom jasnim jezik
jezikom jasnim jez
im jezikom jasnim
zikom jasnim jezik

Rečnik termina

abstract class – apstraktna klasa
Apstraktna klasa je klasa koja ne može da se koristi za pravljenje objekata, ali može biti korisna za pravljenje porodice povezanih klasa. Na primer, apstraktna klasa može definisati skup opštih usluga koje ostale klase realizuju definisanjem funkcija.

S tehničke strane, apstraktna klasa je svaka klasa s jednom *potpuno virtuelnom funkcijom* (engl. *pure virtual function*) ili više takvih funkcija. Potpuno virtuelna funkcija nije definisana u klasi u kojoj je deklarisana, ali može biti realizovana u izvedenim klasama. Pogledajte i *potpuno virtuelna funkcija* (pure virtual function).

abstract data types – apstraktni tipovi podataka
Jedan od zbunjujućih termina teorije objektno orijentisanog programiranja jeste izraz *apstraktni tipovi podataka*. Teoretičari ga vole, ostali lude od njega.

Pojednostavljeno, apstraktan tip podatka je onaj u kome detalji nisu definisani; oni mogu biti skriveni ili naknadno uneti. Filozofski gledano, pojam seže do biti objektne orijentisanosti: prvo definišete veze među objektima i klasama, a kasnije se pobrinete za detalje. U praksi, on nije uvek važan, osim u jednom smislu: jezik kao što je C++ omogućuje sakrivanje detalja realizacije, tako da tip podatka bude definisan onim što *radi* a ne prema tome kako je napravljen.

aggregates – agregati

Agregati su liste konstanti koje se koriste za inicijalizaciju složenih tipova podataka, a pojavljuju se u dva oblika. Prvi vid je skupovni zapis. Skup sadrži elemente koji mogu biti proste vrednosti ili drugi agregati (kao što je znakovni niz). Naredna sintaksa znači da se skup definiše vitičastim zagradama unutar kojih se može navesti proizvoljan broj *elemenata*.

```
{element,[element,]...}
```

Drugi oblik agregata je znakovni literal i on se koristi za inicijalizaciju niza znakova. Sintaksa za neposredno zadavanje vrednosti znakovnog niza je:

```
"tekst"
```

Na primer, sledeći iskaz inicijalizuje niz pokazivača tipa `char*`:

```
char *macke[3] = {"Cica", "Cuca", "Ceca"};
```

U ovom slučaju, svaki od tri pokazivača je inicijalizovan tako da ukazuje na različit znakovni niz. U sledećem primeru nisu inicijalizovani svi pokazivači.

```
char *macke[5] = {"Cica", "Cuca", "Ceca"};
```

To će napraviti niz `macke` od pet pokazivača tipa `char*`; prva tri biće inicijalizovana tako da pokazuju na znakovne nizove a poslednja dva elementa niza podrazumevano će dobiti vrednost `NULL`.

Kada se ugnežđeni skupovi koriste za inicijalizaciju složenih struktura, svaki ugnežđeni skup inicijalizuje jednu logičku celinu. Primera radi, sledeća deklaracija inicijalizuje sva tri reda matrice `matrica1`, ali samo prvu polovinu svakog reda. Neinicijalizovani elementi podrazumevano dobijaju vrednost 0.

```
int matrica1[3][6] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
```

Agregati imaju posebnu ulogu u inicijalizaciji nizova bez indeksa. Tada agregat određuje koliko prostora treba dodeliti nizu. Na primer, naredna deklaracija zauzima memorijski prostor za šest elemenata:

```
int brojevi[] = {1, 5, 21, 27, 33, 40};
```

anonymous unions – anonimne unije
Anonimna je ona unija kojoj nije dodeljeno ime. ANSI standard za C++ dopušta njihovo korišćenje. Kada pristupate članu takve unije, ime člana označava da je on njen deo. Na primer, u deklaraciji:

```
struct paket {
    char ime[20];
    union{
          char    znaci[20];
          double  broj;
    };
} pak1;
```

može se direktno pristupati elementima `pak1.znaci` i `pak1.broj`, bez navođenja imena unije.

Ograničenje ove mogućnosti je što korišćenje anonimne unije ne sme da dovede do bilo kakve dvosmislenosti. U navedenom slučaju nema dvosmislenosti, jer je `ime` različito i od `znaci` i od `broj`. Detaljnija objašnjenja o unijama potražite na strani 130.

argument – argument
Argument je vrednost prosleđena funkciji ili šablonu. Na primeru funkcije `fact`, `fact(4)` vraća faktorijel broja 4, a `fact(5)` faktorijel broja 5. Drugim rečima, brojevi 4 i 5 su argumenti. U definiciji funkcije `fact`, argument može biti definisan kao `n`:

```
long fact (int n)  {
//...
}
```

U nekim knjigama o programiranju različito se koriste termini *parametar* i *argument*, da bi se napravila razlika između deklaracije argumenta (u ovom slučaju `n`) i vrednosti prosleđene funkciji (u istom primeru 4 ili 5). U drugima se za to koriste termini *formalni argument* i *stvarni argument*. U ovoj knjizi se, radi pojednostavljenja, u oba slučaja koristi termin *argument*. Razlikovanje deklarisanog argumenta i vrednosti argumenta ponekad jeste korisno, ali ima slučajeva kada ta razlika sve nepotrebno komplikuje.

array – niz
Niz je skup elemenata istoga tipa, u kome se svakom *elementu* pristupa po indeksu. Deklarisanjem niza možete i više puta da ponovite isti tip. Indeksiranje nizova je primer dobre upotrebe petlji.

Deklaracija niza u obliku `tip ime[n]` pravi niz od `ime[0]` do `ime[n-1]`. Na primer, ako je deklarisano:

```
int brojevi[3];
```

to će definisati sledeće elemente u memoriji, od kojih će svaki biti ekvivalentan pojedinačnoj promenljivoj tipa `int`:

```
brojevi[0]
brojevi[1]
brojevi[2]
```

U jeziku C++, kao i u C-u, između nizova i pokazivača postoji bliska veza. Više informacija o tome pronaći ćete u poglavlju 3.

Zapamtite da je ime niza konstanta. U prethodnom primeru, `brojevi` je konstanta jednaka adresi elementa `brojevi[0]`. (Ime niza je konstanta, jer se adresa ne može menjati, iako se podaci u nizu mogu izmeniti. Ime niza se izjednačava sa ovom adresom i stoga ne može biti promenjeno. Međutim, pokazivač na podatke u nizu može se proizvoljno menjati.)

Pogledajte i *multidimensional array – višedimenzionalni niz*.

assignment operators – operatori dodele

Dodela smešta vrednost u promenljivu (kao u izrazu `x=1`). U jezicima C i C++ postoji veliki broj operatora dodele. Većina njih, poput `+=` i `-=`, izvršavaju neke druge operacije pre dodele. Izuzetak je uobičajeni operator dodele (`=`), koji se, u nedostatku boljeg imena, može nazvati jednostavna dodela.

C i C++ nemaju posebnu vrstu iskaza koja se koristi isključivo za dodelu, već je dodeljivanje samo još jedna vrsta izraza i može se pojaviti unutar većeg izraza. Od dodeljivanja, kao i od svih izraza, možete da napravite iskaz dodavanjem tačke i zareza.

```
x = y;
```

Svim operatorima dodele zajedničko je to što menjaju vrednost levog operanda. Zato taj operand mora biti *lvrednost*: to je izraz, poput zasebne promenljive, koji ima posebnu, važeću adresu u memoriji. (Pogledajte odrednicu *lvalue – lvrednost*.) Lvrednost je obično promenljiva, ali može biti i element niza ili podatak član objekta.

U poglavlju 2 pronaći ćete više informacija o dodeli. U tabeli 2 na strani 44 nabrojani su svi operatori dodele.

association – asocijativnost

Asocijativnost operatora određuje redosled izvršavanja operacija kada su operatori istoga prioriteta. (Pogledajte objašnjenje termina *precedence – prioritet*.)

Na primer, pošto sabiranje (+) i oduzimanje (-) imaju isti prioritet, zapitaćete se kako da izračunate sledeći izraz:

```
kolicina = 10 - 5 + n
```

Po sintaksi, sabiranje i oduzimanje su asocijativni sleva nadesno (kraće: levo asocijativni), pa je stoga ovaj izraz ekvivalentan izrazu:

```
kolicina = (10 - 5) + n   //  kolicina = 5 + n
```

što daje drugačiji rezultat od izraza:

```
kolicina = 10 - (5 + n)   //  kolicina = 5 - n
```

Asocijativnost ćete lako i sami utvrditi. Svi operatori u C++-u su levo asocijativni, osim operatora dodele, unarnih operatora i operatora uslovljavanja (?:). Kada se dvoumite, pogledajte tabele 1 i 2 (str. 44), u kojima su prikazani prioritet i asocijativnost operatora.

Korisno je da zapamtite da su asocijativnost i prioritet u računarskim jezicima isti kao u aritmetici.

base class – osnovna klasa

Osnovna klasa prenosi članove *izvedenoj klasi*. Primera radi, ako je klasa B osnovna klasa klase C, onda su svi članovi klase B automatski i članovi klase C. Ova veza je primer nasleđivanja u C++-u. Osnovna klasa se ponekad zove natklasa (engl. *superclass*) ili roditeljska klasa (engl. *parent class*). Više informacija o tome potražite u poglavlju 5.

base-class constructor – konstruktor osnovne klase

Konstruktor osnovne klase (ako postoji) može da se pozove iz konstruktora potklase. To je posebno korisno ako u osnovnoj klasi postoje privatni članovi, jer oni drugačije ne bi mogli ni da budu inicijalizovani. Sintaksa je, uopšteno, ovakva:

```
klasa::klasa(argumenti) : osnovna_klasa(argumenti) {
      iskazi
}
```

Pretpostavimo da je klasa `CSportsCar` izvedena iz klase `CAuto`. Sledeći primer pokazuje kako se poziva konstruktor osnovne klase. U njemu su argumenti `h` i `m` prosleđeni konstruktoru klase `CAuto`.

```
CSportsCar::CsportsCar(double h, CStr m, double a) :
      CAuto(h, 2, m) {

      accel_0_60 = a;
      stripes = 0;
}
```

Inicijalizacione vrednosti `h` i `m` zajedno s konstantom 2 prosleđene su konstruktoru osnovne klase `CAuto`. Potpuniji opis ovog primera nalazi se u poslednjem odeljku poglavlja 6.

binary mode – binarni režim

Binarni režim je jedan od dva režima ulazno-izlaznih operacija u kojima možete otvoriti datoteku. Prilikom čitanja ili pisanja u binarnom režimu, podaci se ne prevode, nego im pristupate kakvi su u datoteci. Suprotno tome, u tekstualnom režimu se znakovni nizovi upisuju uz prevođenje svake oznake za kraj reda (jedan znak u memoriji) u par znakova: kraj reda/prelazak u novi red (engl. *carriage-return/linefeed*).

Druga mogućnost binarnog režima koju programeri obično (ali ne uvek) koriste jeste direktno čitanje i pisanje numeričkih vrednosti, a ne kao niza tekstualnih znakova koji predstavljaju vrednost. Kada koristite funkcije biblioteke `stdio`, vrednosti čitate i pišete direktno koristeći funkcije `fread` i `fwrite` (str. 174, 175). Kada koristite klase tokova, vrednosti čitate i pišete direktno, pomoću funkcije članice `read` i `write` (str. 288 i 299).

Na primer, ako upisujete vrednost 255 kao tekst, program snima ASCII kodove za „2“, „5“ i „5“ u datoteku: te vrednosti su 50, 53 i 53. Ako upisujete vrednost 255 direktno, program snima 255 u datoteku (heksadecimalno 0xFF). Ukoliko je vrednost 255 kratkog celobrojnog tipa (`short`), u datoteku se snimaju bajtovi 0x00 0xFF.

binary operator – binarni operator

Binarni operator kombinuje dva operanda da bi formirao veći izraz. U jeziku C++, kao i u većini drugih, skoro svi operatori su binarni. Uobičajeni primeri su sabiranje, oduzimanje i množenje. Na primer:

```
x + y
količina * faktor
```

Dodela je takođe binarna operacija, jer ima levi i desni operand.

```
x = y
```

Pogledajte i *unary operator – unarni operator*.

bit field – polje bitova

Polje bitova je C/C++ tip podataka koji predstavlja podskup skupa bitova u okviru celog broja. Takvo polje može da čini samo jedan bit, ali ono može biti široko i nekoliko bitova. Kada prevodi kôd koji radi s poljima bitova, prevodilac pravi instrukcije koje izvlače, menjaju i upisuju vrednosti pojedinih bitova. Iako je ishod isti kao da ste direktno koristili operacije nad bitovima, polja

bitova se lakše čitaju i razumeju, a u dosta slučajeva zahtevaju manje redova izvornog koda za istu operaciju. U poslednjem odeljku poglavlja 2 nalazi se više detalja o tome.

Boolean values – logičke vrednosti

Logički izraz (engl. *Boolean*) može imati dve vrednosti: `true` i `false`. Tipovi podataka po pravilu imaju mnogo više od dve vrednosti. Ako bi smeštaj bio optimalan, za čuvanje logičke vrednosti bio bi dovoljan jedan bit. (Pogledajte prethodno objašnjenje). Međutim, pristupanje bitu zahteva dodatne instrukcije, zbog čega se logičke promenljive obično čuvaju kao celi brojevi.

U predašnjim verzijama C++-a nije postojao zaseban tip logičke vrednosti; nula je predstavljala vrednost *netačno*, a sve različito od nule vrednost *tačno*. Nepažnja je mogla da dovede do pogrešnih rezultata. Na primer, konjunkcija 6 i 9 nad bitovima daje 0, što je vrednost *netačno*, iako 6 i 9 predstavljaju vrednost *tačno*. Jedno rešenje je da umesto operatora nad bitovima koristite logičke. Logički operatori C/C++-a na isti način rade sa svim vrednostima različitim od 0.

ANSI C++ ima drugo rešenje: korišćenje novog logičkog tipa `bool`. Svaka vrednost različita od 0 dodeljena promenljivoj tipa `bool` automatski se pretvara u 1. Više informacija o ovom tipu podataka potražite na strani 71.

U literaturi na engleskom jeziku, reč *Boolean* se piše velikim slovom jer vodi poreklo od prezimena Džordža Bula (George Boole), oca moderne simboličke logike. (To je jedna od zanimljivosti koje ste možda čuli na fakultetu, a onda zaboravili.)

callback function – funkcija povratnog poziva

Neki pozivi funkcija traže adresu druge funkcije koju ste već napisali, funkcije *povratnog poziva.* U suštini, tako bar privremeno, omogućujete tuđem kodu da vrati kontrolu vašem kodu. Iako ovo možda zvuči nejasno, to je jedini način da dovoljno elastično realizujete funkcije `qsort` i `bsearch`. Kada pozivate neku od ovih funkcija, prosleđujete adresu vaše funkcije povratnog poziva.

Značaj ovih funkcija je i u tome što deklarišu adrese funkcija kao argumente. Tu se primenjuje sintaksa pokazivača na funkcije. Više informacija potražite u objašnjenjima funkcija `qsort` i `bsearch` na str. 213 i 170.

cast – konverzija tipova

Konverzija tipova uzima vrednost izraza, menja mu tip i rezultat prosleđuje dalje. (Pri konverziji tipova nema posledica po tip izvorne promenljive; utičete samo na tip izraza.) U nekim slučajevima, konverzija tipova menja i način

predstavljanja vrednosti u memoriji. Na primer, konverzija celobrojne vrednosti u realnu menja podatke s kojima se radi:

```
double d = static_cast<double>(i);
```

Ova operacija se može izraziti i staromodnom konverzijom tipova u C-u:

```
double d = (double) i;
```

U ostalim slučajevima, konverzija tipova ne utiče na podatke, nego menja samo način na koji se izraz tumači. Na primer, konverzija pokazivača tipa `int*` u `float*` ne utiče na vrednost pokazivača, već samo na format koji se koristi za interpretaciju bitova. Prilikom konverzije tipova pokazivača bitno je obratiti pažnju na razliku između formata podataka.

Standard ANSI C++ definiše nove operatore konverzije tipova `const_cast`, `dynamic_cast`, `reinterpret_cast` i `static_cast`, ali podržava i staromodnu konverziju iz jezika C. Pravila i upotreba operatora moderne konverzije tipova razmatrani su u odgovarajućem pregledu u prvom delu knjige. Pogledajte i *promotion – unapređenje tipova*.

class – klasa

Klasa je jedan od osnovnih pojmova jezika C++. Sažeto, klasa je svaki tip koji definiše korisnik, osim niza ili tipa definisanog rezervisanom rečju `typedef`; tu spadaju sve strukture podataka definisane rezervisanim rečima `class`, `struct` ili `union`. Kada deklarišete klasu, definisanjem nove strukture tipa podataka proširujete jezik C++.

Deklarisanu klasu možete koristiti da napravite brojne instance, ili *objekte*, od kojih svaka ima atribute definisane u klasi. Poglavlje 5 vas upućuje u klase i objekte.

Klase su napravljene po uzoru na strukture jezika C, koje mogu da sadrže polja podataka, ali im je dodata podrška za funkcije. To su *funkcije članice*: one definišu operacije na objektu klase. Možete da definišete i kako će operatori (kao što su `+`,`*`,`/` itd.) raditi s objektima (poglavlje 7). Druga važna mogućnost klasa jeste privatni pristup, koji skriva odabrane članove klase od ostatka sveta. (Pogledajte *encapsulation – kapsuliranje*.)

class instance – instanca klase

Instanca je isto što i objekat. Pogledajte objašnjenje termina *instance – instanca* i *object – objekat*.

comments – komentari

Komentar je tekst koji prevodilac ignoriše. Iako nije preveden, komentar ostaje deo izvornog koda. Teoretski, komentar može biti bilo šta. Ako hoćete,

možete upisati „Biti ili ne biti“. Uobičajeno je, naravno, da programer upiše komentar koji će pomoći onima što čitaju izvorni kôd.

Sintaksa C++-a podržava dve vrste komentara. Najčešći je jednoredni komentar; prevodilac ignoriše sav tekst od dve kose crte (`//`) do kraja reda. (Ovo je jedna od retkih situacija kada je u C++-u bitan prelom reda.)

```
x = y; // ovo je komentar; x je dobio vrednost y
```

Druga vrsta komentara, višeredni komentar, nasleđen je iz jezika C. Prevodilac ignoriše sav tekst između `/*` i `*/` bez obzira na prelom reda, na primer:

```
/* ovo je višeredni komentar
u jeziku C. */
```

Zapamtite da u višerednim komentarima simboli za komentare ne funkcionišu kao zagrade. Komentare ne možete da ugnezdite, jer prevodilac ignoriše sav tekst od simbola za početak komentara (`/*`) do *prvog* simbola za kraj komentara (`*/`) na koji naiđe. Zbog toga sledeći primer izaziva sintaksnu grešku.

```
/* Ovo je spoljašnji komentar, a
    /* Ovo je umetnuti komentar. */
    Ovo je nastavak spoljašnjeg komentara. */
```

Prevodilac će pročitati simbol za kraj komentara (`*/`) na kraju drugog reda i tumačiće ga kao kraj komentara, a treći red će pokušati da prevede kao običan kôd. Da biste privremeno blokirali kôd koristite pretprocesorske komande `#if` i `#endif`, koje mogu biti ugnežđene na bilo kom nivou. Više informacija o ovim pretprocesorskim komandama pronaći ćete u odeljku 5 prvog dela knjige.

complex data types – složeni tipovi podataka
Složen tip podataka je sačinjen od drugih tipova. U njega spadaju svi nizovi, klase, unije i strukture, kao i pokazivači. Nisu složeni samo prosti tipovi podataka kao što su `int`, `bool` i `float`.

compound statements – složeni iskazi
Složeni iskaz se sastoji od jednog ili više iskaza unutar vitičastih zagrada (`{}`). Iskazi se posmatraju kao grupa – izvršavaju se ili svi ili se ne izvršava nijedan. Sintaksa za složeni iskaz je:

```
{
iskazi
}
```

Složeni iskazi imaju brojne primene. Prvo, oni definišu iskazni blok. Oblast važenja lokalnih promenljivih prestaje posle završne zagrade (}). Druga njihova važna primena je s kontrolnim strukturama. Gde god možete da stavite iskaz, možete da stavite i složeni iskaz. Stoga je moguć sledeći `if` iskaz, u kome se izvršavaju ili svi iskazi ili nijedan:

```
if (zameni_sad) {
        privremeni = a;
        a = b;
        b = privremeni;
}
```

U jezicima C i C++, kraj iskaza označava tačka i zarez (;), za razliku od Pascala, gde taj znak razdvaja iskaze. Zbog toga je sintaksa C++-a jednostavnija. Tačkom i zarezom ne završava se jedino složeni iskaz; otuda se tačka i zarez ne koristi posle završne vitičaste zagrade (}) osim u slučaju deklarisanja klase.

conditional compilation – uslovno prevođenje

Uslovno prevođenje je tehnika za čuvanje više verzija programa. Koristite je da ne biste morali ponovo da pišete program svaki put kada ga prevedete za drugačiju ciljnu platformu. Treba samo da upišete redove specifične za platforme unutar blokova `#if`...`#endif`. Na taj način, uz vrlo malo truda, možete promeniti ono što će biti prevedeno u kodu. U odeljku 5 prvog dela naći ćete više informacija o komandi `#if` i drugim pretprocesorskim komandama.

control structure – kontrolna struktura

Kontrolna struktura je iskaz koji može da menja tok izvršavanja programa ili da donosi odluke. (Iako je računar Deep Blue pobedio Kasparova, još uvek je sporno mogu li računari stvarno donositi odluke. Kontrolna struktura, u stvari, samo izvršava jednostavne numeričke testove i sledi instrukcije.)

U kontrolne strukture spadaju naredbe `if-else`, `while`, `do`, `for` i `switch`. Funkcije i naredba `goto` takođe menjaju tok izvršavanja. Više informacija o tome potražite u odeljku 4 prvog dela.

constant – konstanta

Konstanta je vrednost koja se ne menja. U C++-u postoji nekoliko vrsta konstanti.

- Numeričke konstante kao što su 100, 4.5, 0xFF i 018. (Poslednje dve koriste heksadecimalni i oktalni format.)
- Simboličke konstante definisane komandom `#define`. Zovu se i *makroi* (engl. *macros*).

- Konstante definisane rezervisanom rečju `enum`.
- Znakovni literali, na primer `"Ovo je znakovni niz"`.
- Ime niza, koje je konstanta jednaka prvom elementu niza. (Imena nizova su konstante, ali pokazivači i elementi niza nisu.)
- Bilo koja promenljiva deklarisana rezervisanom rečju `const`.

Poslednja kategorija se bitno razlikuje od ostalih. Promenljiva deklarisana kao `const` i dalje je promenljiva, iako je i konstanta. Ona zauzima prostor u toku rada programa, poput svake druge promenljive, a može biti bilo kog osnovnog tipa, imati bilo kakvu oblast važenja i biti smeštena u bilo koju klasu.

Prve dve kategorije razlikuju se od promenljivih sa oznakom `const` po tome što njihove konstante postoje samo u vreme prevođenja. Te konstante se, zavisno od prevodilaca, mogu zameniti tokom prevođenja. Na primer, izraz 2 + 2 verovatno će biti zamenjen sa 4.

Više informacija o numeričkim konstantama potražite u poglavlju 2, a o znakovnim nizovima – u poglavlju 3. Komanda `#define` je opisana na strani 145, a rezervisane reči `enum` i `const` na stranama 84 i 76.

constructor – konstruktor

Konstruktor je funkcija za inicijalizaciju klase. Kad god pravite objekat izvrši se konstruktor klase tog objekta. Konstruktor je logično mesto za inicijalizovanje vrednosti podataka članova i izvršavanje drugih inicijalnih zadataka. On se uvek zove jednako kao klasa i nema povratni tip, čak ni `void`.

Ista klasa može imati više konstruktora, ali svaki od njih mora da ima jedinstvenu listu argumenata. Lista argumenata konstruktora mora da odgovara argumentima korišćenim za inicijalizaciju objekta te klase (ako ih je bilo). Ukoliko nikakve vrednosti nisu korišćene za inicijalizaciju objekta, izvršava se podrazumevani konstruktor klase. (Pogledajte odrednicu *default constructor – podrazumevani konstruktor.*) Više detalja o konstruktorima predočeno je u poglavlju 6. Pogledajte i *copy constructor – konstruktor za kopiranje.*

conversion functions – funkcije konverzije

C++ dozvoljava konverziju između tipova na dva načina. Klasa C može definisati ulazne konverzije tipa T putem korišćenja konstruktora oblika C(T). Ulazna konverzija definiše kako sadržaj tipa T može biti dodeljen objektu klase C (kao kod `c_obj = t_obj`).

Operatori konverzije definišu izlazne konverzije. Na primer, ako klasa C ima funkciju konverzije za tip T, ta funkcija će definisati kako se instance klase C dodeljuju instancama klase T (kao kod `t_obj = c_obj`). Sintaksa operatora konverzije razmatrana je u odeljku „Pisanje funkcije konverzije“, na strani 436 poglavlja 7.

Konstruktori i funkcije konverzije ne definišu samo kako će se izvršiti dodela već definišu i posredne konverzije. Primera radi, ako klasa `CStr` definiše konverziju u `char*`, onda možete da prosledite instancu klase `CStr` gde god funkcija zahteva argument tipa `char*`.

copy constructor – konstruktor za kopiranje

Konstruktor za kopiranje poziva se svaki put kada program inicijalizuje nov objekat pomoću drugog objekta iste klase. Kada se objekat prosleđuje funkciji po vrednosti, program automatski pravi kopiju argumenta ovim konstruktorima i prosleđuje je dalje.

Drugi slučaj kada se ovaj konstruktor koristi jeste funkcija čija povratna vrednost je objekat, ali se ne koristi ni pokazivač ni referenca. Konstruktor za kopiranje automatski pravi trajnu kopiju koju funkcija vraća pozivaocu. (Pošto je to neefikasno, često se koriste pokazivači i reference.)

Svaka klasa ima jedan konstruktor za kopiranje. Ako ga sami ne napišete, prevodilac će automatski definisati podrazumevani konstruktor za kopiranje. On pojedinačno kopira podatke članove objekta. To je prihvatljivo u mnogim situacijama, ali može biti neprikladno ako klasa sadrži pokazivače. Parametar konstruktora za kopiranje je referenca na objekat iste klase. (Pogledajte objašnjenje termina *reference – referenca*.)

```
C(const C&)
```

Konstruktor za kopiranje ima sličnosti s *operatorom dodele* (dodela vrednosti jednog objekta drugom objektu iste klase), ali oni nisu istovetni. Konstruktor za kopiranje inicijalizuje potpuno nov objekat.

Više detalja o konstruktorima potražite u poglavlju 6.

data abstraction – apstrakcija podataka

Apstrakcija podataka je umetnost definisanja tipova na osnovu onoga što rade, a ne prema njihovoj unutrašnjoj strukturi. Dok učite C++, nemojte obraćati mnogo pažnje na ovaj termin – pomalo je nejasan. Pogledajte i *abstract data types – apstraktni tipovi podataka*.

data member – podatak član objekta

Podaci članovi su promenljive u deklaraciji klase. Po pravilu, oni definišu polja podataka za svaki objekat klase. Postoji jedan izuzetak: podatak član deklarisan rezervisanom rečju `static` nije povezan s poljem podataka unutar pojedinačnog objekta, već statične podatke članove dele svi objekti iste klase – što znači da nemaju mnogo veze s pojedinačnim objektima.

data type – tip podataka
Pogledajte objašnjenje termina *type – tip*.

declaration – deklaracija
Deklaracija je iskaz koji opisuje promenljivu, klasu ili funkciju. Takav opis daje informacije o tipu, tj. saopštava programu sa čime se radi. Neke deklaracije definišu promenljivu ili kôd funkcije; one se zovu *definicije*. Na primer:

```
int x,y,z;        // Deklariše x, y i z kao celobrojne
                  // promenljive.
```

Ostale vrste deklaracija pružaju informacije za tumačenje isturenih i spoljašnjih referenci (engl. *forward reference* i *external reference*). To uključuje prototipe funkcija i deklaracije rezervisanom rečju `extern`. C++ je stroži od mnogih drugih jezika. Funkcija mora biti deklarisana – pomoću definicije ili putem prototipa – pre nego što je pozovete.

Da biste razumeli složene deklaracije C/C++-a, morate da koristite obrnutu logiku. Zapitajte se: šta znači kada se ovaj izraz pojavi u izvršnom kodu? Primera radi, razmotrite sledeću deklaraciju:

```
char *niz[100];
```

Da biste je razumeli, primenite pravila prioriteta i asocijativnosti. Budući da su unarni operatori asocijativni zdesna nalevo (desno asocijativni), deklaracija je ekvivalentna ovoj:

```
char *(niz[100]);
```

Pretpostavite da se izraz `*(niz[n])` pojavio u kodu, u kome `n` dobija vrednosti od 0 do 99. To znači da je element uzet s indeksom od 0 do 99, a onda dereferenciran da dobije vrednost tipa `char`. Stoga se `niz` sastoji od 100 pokazivača na `char`. Suprotno tome,

```
char (*pokazivac)[100];
```

pokazivač pokazuje na niz od 100 elemenata tipa `char`.

Sličnom logikom možete shvatiti zašto se i po čemu razlikuju naredna dva izraza. Prvi je prototip funkcije, a drugi pokazivač na funkciju:

```
int (*pfn1)(void);    // funkcija vraća pokazivač
                      // na celobrojni tip
int *pfn2(void);      // pokazivač na funkciju
                      // koja vraća celobrojni tip
```

Da biste razumeli ove deklaracije, važno je da zapamtite da `pfn2` mora biti dereferencirano u izvršnom kodu pre nego što se pozove funkcija.

decorating – obeležavanje imena
Obeležavanje imena je postupak koji koristi prevodilac C++-a da bi uz ime napravio informaciju o tipu. Na primer, ako deklarišete promenljivu `Pera` tipa `int`, u datoteke .obj ili .o neće se samo snimiti ime `Pera`, već i dodatne oznake uz ime, pa je rezultat duže ime koje identifikuje `Pera` kao promenljivu tipa `int`.

Dva su glavna razloga za obeležavanje imena. Jedan je podržavanje *povezivanja bezbednog za tipove (*engl*. type-safe linkage)*. Program za povezivanje razlikuje promenljive različitog tipa, čak i ako imaju isto ime. Time se sprečavaju neke greške pri povezivanju. Druga svrha obeležavanja imena jeste omogućavanje preklapanja funkcija. Postupak obeležavanja imena razlikuje se od prevodioca do prevodioca. Pogledajte i *overloading, function – preklapanje funkcija.*

default argument value – podrazumevana vrednost argumenta
Svakom argumentu u definiciji funkcije može biti dodeljena podrazumevana vrednost. Ako funkcija očekuje argument, a pozvana je bez argumenta, koristiće se podrazumevana vrednost. Primera radi, pretpostavite da ste definisali sledeću funkciju, koja ima jedan traženi argument i jedan sa podrazumevanom vrednošću:

```
void postavi(char* ime, int iznos = 0);
```

Ako se `postavi` poziva sa samo jednim argumentom, pretpostavlja se da je `iznos` 0.

```
 postavi("Mika Peric");
```

Međutim, ako se `postavi` poziva s oba argumenta, `iznos` će dobiti zadatu vrednost.

```
postavi( "Mika Peric", 5)
```

C++ nameće neka ograničenja podrazumevanim vrednostima. Prvo, svi argumenti s podrazumevanom vrednošću nalaze se na kraju liste. Ovo ograničenje je neophodno jer se vrednosti dodeljuju argumentima sleva nadesno. Na primer, ako pozovete funkciju i navedete tri argumenta, prva tri argumenta imaće zadate vrednosti; preostali argumenti dobiće podrazumevane vrednosti.

Sledeće ograničenje se odnosi na preklapanje funkcija. (Pogledajte odrednicu *overloading, function – preklapanje funkcija.*) Kada je ime funkcije preklopljeno, potpisi funkcija se moraju razlikovati po bar jednom argumentu koji *nema* podrazumevanu vrednost.

default constructor – podrazumevani konstruktor

Podrazumevani konstruktor je funkcija inicijalizacije koja nema argumente. Kad god za pravljenje objekta nisu korišćene vrednosti za njegovu inicijalizaciju, poziva se podrazumevani konstruktor klase tog objekta. Potpis podrazumevanog konstruktora za klasu C je:

```
C();
```

Ako ne napišete konstruktore za svoju klasu, prevodilac će napraviti skriveni podrazumevani konstruktor koji će svim podacima članovima dodeliti vrednost 0; međutim, ako definišete *bilo kakav* konstruktor, to se neće desiti. Zbog toga bi bilo dobro da napišete podrazumevani konstruktor za svaku klasu, iako on ništa neće raditi.

Možda vam izgleda neobično što prevodilac sam pravi podrazumevani konstruktor, ali ga uklanja čim vi napišete neki drugi. Premda na to liči, ovo nije deo zavere da bi vam se život otežao. C++ pruža automatske podrazumevane konstruktore da bi bio kompatibilan sa strukturama preuzetim iz jezika C. Kada počnete da pišete klase, ne bi trebalo da se oslanjate na konstruktor prevodioca.

Više informacija o podrazumevanom konstruktoru i ostalim vrstama konstruktora potražite u poglavlju 6.

definition – definicija

Definicija formira promenljivu ili kôd funkcije. U velikom i složenom projektu, funkcija ili promenljiva može biti deklarisana mnogo puta, ali samo jednom definisana. (Izuzetak: virtuelna funkcija može biti definisana drugačije u različitim klasama.) Definicija promenljive je mesto u kodu programa gde se promenljiva pravi i gde može biti inicijalizovana. Primer:

```
int i;
```

Definicija funkcije sadrži iskaze koji treba da se izvrše po pozivu funkcije. Evo kratkog primera. (Da, to je još jedan primer funkcije faktorijel!)

```
long fakt(int n) {
     long vrednost;
     for (vrednost = 1; n > 1; n--)
          vrednost *= n;
     return vrednost;
}
```

Definicije su vrsta deklaracije, ali postoje i deklaracije koje nisu definicije. Na primer, deklaracije `extern` ne definišu promenljivu.

dereference – dereferenciranje
Dereferenciranje s adresnog izraza daje sadržaj sa te adrese. Primera radi, ako je vrednost adresnog izraza numerička vrednost 1055, onda će se dereferenciranjem izraza dobiti sadržaj s adrese 1055. Drugačije bi se to reklo ovako: dereferenciranjem pokazivača dobija se ono na šta pokazivač ukazuje. Na primer:

```
int n = 5;
int *p = &n;    // Stavlja adresu od n u p.
cout « *p       // Prikazuje ono na šta p pokazuje (n).
```

Zvezdica (*), primenjena kao unarni operator, jeste operator dereferenciranja ili *operator indirekcije* (engl. *indirection operator*). Kada se koristi za definisanje podataka, kao gore u drugom iskazu, ona pravi pokazivač. Kada se, pak, koristi u izvršnom iskazu, dereferencira pokazivač.

Pokazivači koji ukazuju na druge pokazivače i višedimenzionalni nizovi malo komplikuju stvari. Adresni izraz mora biti potpuno dereferenciran da bi se dobili podaci s njegove adrese. Na primer, ako pokazivač `pp` ima vrednost 1055 ali je tipa `int**`, on mora biti dereferenciran dvaput da bi se dobila celobrojna vrednost:

```
int matrica[2] [2] = {{1, 2}, {3,4}};
int **pp = matrica;     // Pokazuje na početak matrice.
cout « **pp;            // Prikazuje prvi element matrice.
```

Zapamtite da primena indeksa dereferencira pokazivač, isto što i * radi.

derived class – izvedena klasa
Izvedena klasa nasleđuje neke ili sve svoje članove od druge klase, koja se zato zove *osnovna klasa*. Na primer, ako je klasa C izvedena iz osnovne klase B, onda su svi članovi klase B automatski i članovi klase C. (U izvedenu klasu C mogu da se dodaju i novi članovi.) Izvedena klasa se ponekad imenuje kao *potklasa (*engl. *subclass)* ili *klasa potomak (*engl. *child class).*Više informacija o tome pronaći ćete u poglavlju 5.

destructor – destruktor
Destruktor je suprotnost konstruktoru. Konstruktor se izvršava čim se objekat napravi, a destruktor pre nego što se objekat ukloni iz memorije. Destruktor se skoro nikada ne poziva direktno. On se automatski poziva kada neka akcija, kao što je korišćenje operatora `delete`, izaziva uništenje objekta. (Ne zaboravite da se sav kôd objekta nalazi u njegovoj klasi.)

Iako se destruktor ne mora napisati za svaku klasu, takva funkcija može biti korisna prilikom čišćenja objekata i izvršavanja zadataka vezanih za

njihovo uništavanje. Na primer, destruktor možete upotrebiti da zatvorite datoteke povezane s objektom neke klase. Suprotno od konstruktora, destruktori su jedinstveni: postoji samo po jedan za svaku klasu. Za svaku zadatu klasu C, deklaracija destruktora je:

```
~C ();
```

Više detalja o tome potražite u poglavlju 5, u odeljku „Život i smrt objekta: konstruktori“ na strani 402. Uprkos surovom imenu (*destruktor*) i sintaksi (*može postojati samo jedan*), destruktori su ponekad korisni.

directive – pretprocesorska komanda

Pretprocesorska komanda je specijalna komanda prevodiocu koja se izvršava pre nego što je ostatak programa preveden i pokrenut. Za razliku od iskaza, pretprocesorska komanda se nikada ne prevodi u izvršni kôd, pa sve što postignete ovim komandama nimalo ne smanjuje brzinu izvršavanja. Jedna od korisnijih pretprocesorskih komandi jeste `#define`, koja definiše simboličku konstantu:

```
#define PI 3.14159265
```

U odeljku 5 prvog dela knjige naći ćete potpun spisak pretprocesorskih komandi propisanih ANSI specifikacijom, zajedno s njihovim opisima.

empty statement – prazan iskaz

U jezicima C i C++ dozvoljeno je korišćenje praznog iskaza, tj. nepopunjenog iskaza koji se završava tačkom i zarezom (;). Dobra strana te mogućnosti je što obično ne bivate kažnjeni ako vam prst sklizne, pa otkucate tačku i zarez više. Sledeći kôd je, na primer, potpuno ispravan:

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

Tehnički, ovo nije jedan iskaz, već deklaracija za kojom slede dva prazna iskaza.

Prazni iskazi su korisni bar u jednom slučaju: kao pomoć pri obezbeđivanju odredišta iskaza `goto`. Ako biste hteli da skočite na kraj procedure, morali biste da skočite na oznaku praznog iskaza. U protivnom, oznaka ne bi prethodila nijednom iskazu i bila bi nedozvoljena.

```
i = m * 2;
if (i == j)
   goto kraj;
//...
kraj:
    ;
}
```