Dodatak A
Jezički detalji

Ovaj dodatak ima dve namene: sadrži dodatne detalje osnove jezika i u kratkim crtama opisuje sve izraze i iskaze jezika, uključujući i neke koje nismo koristili u knjizi. Materijal o osnovama jezika pre svega se odnosi na neke složene elemente sintakse deklaracija u jeziku C++, kao i na detalje ugrađenih aritmetičkih tipova (sve to je nasleđeno od programskog jezika C). Ovi detalji nisu neophodni za razumevanje programa iz ove knjige. I bez njih ćete moći da pišete dobre programe na jeziku C++. Ali ovakva znanja su neophodna za pisanje mnogih programa, pa će biti korisno da vam predstavimo i tu stranu jezika C++.

U ovom dodatku ćemo koristiti sledeću sintaksu: simboli u `neproporcionalnom` stilu označavaju sami sebe, reči napisane *kurzivnim* stilom predstavljaju sintaksičke kategorije, tri tačke (...) označavaju proizvoljan broj (može i nula) ponavljanja stavke koja neposredno prethodi, a fraze unutar kurzivnih uglastih zagrada *[]* nisu obavezne. Pored toga, za grupisanje stavki koristimo vitičaste zagrade napisane u kurzivnom stilu *{}*, a uspravnu crtu | stavljamo između dve opcije. Evo primera:

deklaracija:specifikatori-deklaracije [deklarator [inicijalizator]] [, deklarator [inicijalizator]]...;

Ovim smo saopštili da se *deklaracija* sastoji od *specifikatora-deklaracije*, zatim proizvoljnog broja (može i nula) *deklaratora*, iza kojih može da sledi *inicijalizator*. Deklaracija se završava tačkom i zarezom (`;`).

A.1 Deklaracije

Deklaracije su ponekad teško razumljive, posebno ako se u okviru njih deklariše nekoliko imena s različitim tipovima, ili ako imaju veze s funkcijama koje vraćaju pokazivače na funkcije. Na primer, u odeljku 10.1.1 videli smo da definicija

```
int* p, q;
```

definiše `p` kao podatak tipa „pokazivač na tip `int`“, a `q` kao objekat tipa `int`. Vratimo se deklaraciji iz odeljka 10.1.2:

```
double (*get_analysis_ptr())(const vector<Student_info>&);
```

Ovim smo deklarisali da je `get_analysis_ptr` funkcija bez argumenata, koja vraća pokazivač na funkciju sa argumentom tipa `const vector<Student_info>&` i povratnim tipom `double`.

Ako razložimo prethodne deklaracije, biće jasnije o čemu se radi:

```
int* p;
int q;
```

odnosno

```
// definiše anaysis_fp kao tip funkcije sa argumentom tipa
// const vector<Student_info>& i povratnim tipom double
typedef double (*analysis_fp)) const vector<Student_info>&);
analysis_fp get_analysis_ptr();
```

Nažalost, to što se mi trudimo da pojednostavimo deklaracije ne znači da će deklaracije u drugim programima biti lako razumljive.

U osnovi, deklaracije imaju ovakav oblik:

deklaracija:specifikatori-deklaracije [deklarator [inicijalizator]] [, deklarator [inicijalizator]]...;

Deklarišu se imena svih *deklaratora*. Ova imena važe od mesta deklaracije do kraja oblasti važenja deklaracije. Neke deklaracije su istovemeno i definicije. Imena se mogu deklarisati više puta, ali smeju da se definišu samo jednom. Deklaracija je definicija ukoliko rezerviše skladište u memoriji, definiše klasu ili telo funkcije.

Jezik C++ je sintaksu deklaracija nasledio od jezika C. Svaka deklaracija počinje sekvencom *specifikatora-deklaracije* koji predstavljaju tip i ostale atribute onoga što se deklariše. Nakon toga, piše se proizvoljan broj (može i nula) *deklaratora*, kojima se pojedinačno može pridružiti *inicijalizator*.

Prva stvar koju treba da pronađete u deklaraciji jeste granica između specifikatora i deklaratora. To je iznenađujuće lako: svi specifikatori su rezervisane reči ili imena tipova, pa postoje samo do prvog simbola koji ne odgovara pomenutim odrednicama. Na primer, u iskazu

```
const char * const * const * cp;
```

prvi simbol, `*`, nije ni rezervisana reč ni ime tipa, pa su specifikatori reči `const char`, a (jedini) deklarator je `*const * const * cp`.

Drugi primer je nešto misterioznija definicija (odeljak 10.1.2):

```
double (*get_analysis_ptr())(const vector<Student_info>&);
```

Ovde je granicu lako naći: `double` je ime tipa, a otvorene zagrade napisane nakon njega nisu ni rezervisana reč niti ime tipa. To znači da se *specifikatori-deklaracije* sastoje samo od rezervisane reči `double`, a ostatak deklaracije (izuzev tačke i zareza) jesu deklaratori.

A.1.1 Specifikatori

Specifikatore deklaracije možemo podeliti u tri kategorije: specifikatori tipa, specifikatori klasa za skladištenje i ostali specifikatori:

specifikatori-deklaracije: { specifikator-tipa | specifikator-klase-za-skladištenje | ostali-specifikatori } ...

Iako ovakva podela može pomoći da razumete deklaracije, nema veći značaj jer, konkretno, u deklaracijama se ne može uočiti odgovarajuća podela: *specifikatori-deklaracije* mogu imati bilo koji redosled.

Specifikatori tipa određuju tip onoga što se deklariše. O ugrađenim tipovima govorićemo u odeljku A.2.

specifikator-tipa:`char` | `wchar_t` | `bool` | `short` | `int` | `long` | `signed` | `unsigned` | `float` | `double` | `void` | *ime-tipa* | `const` | `volatile`
ime-tipa:ime-klase | *ime-nabrojivog-tipa* | *definisano-ime-tipa*

Specifikator `const` označava da se sadržaj objekta tog tipa ne sme menjati. Rezervisana reč `volatile` ukazuje prevodiocu da na vrednost promenljivih mogu uticati i spoljni faktori da treba izbegavati optimizaciju ovih promenljivih.

Obratite pažnju na to da rezervisana reč `const` može biti deo i specifikatora (tada menja značenje tipa) i deo deklaratora, kada zadaje konstantni pokazivač. Nećete imati nikakve nedoumice u vezi s rezervisanom rečju `const` jer joj uvek prethodi zvezdica * kada je deo deklaratora.

Specifikatori klase za skladištenje određuju gde se nalazi promenljiva:

specifikatori-klase-za-skladištenje: `register` | `static` | `extern` | `mutable`

Specifikator `register` ukazuje prevodiocu da pokuša objekat smestiti u registar kako bi poboljšao performanse.

Uobičajeno je da se lokalne promenljive uništavaju na kraju bloka u kome su deklarisane; statičke promenljive opstaju i van oblasti važenja.

Specifikator `extern` označava da tekuća deklaracija nije definicija, što ukazuje na to da se odgovarajuća definicija nalazi na nekom drugim mestu.

Specifikator `mutable` se koristi samo za podatke članove, dopuštajući da se ti podaci menjaju, čak i ako su članovi konstantnih objekata.

Ostali specifikatori deklaracije definišu svojstva koja nemaju veze s tipovima:

ostali-specifikatori-deklaracije: `friend` | `inline` | `virtual` | `typedef`

Specifikatorom `friend` (odeljci 12.3.2 i 13.4.2) zaobilaze se mehanizmi zaštite podataka u klasi.

Specifikator `inline` se javlja u definicijama funkcija i ukazuje prevodiocu da, po mogućstvu, ugradi funkciju u kôd. Definicija takve funkcije mora biti dostupna kada se funkcija pozove, pa je korisno smestiti telo ugrađene funkcije u isto zaglavlje u kome je i njena deklaracija.

Specifikator `virtual` (odeljak 13.2.1) može se upotrebiti samo za funkcije članice. On označava funkciju čiji se pozivi mogu dinamički povezivati.

Specifikator `typedef` (odeljak 3.2.2) definiše sinonim tipa.

A.1.2 Deklaratori

Svaki deklarator određuje svojstva jednog entiteta. Entitet dobija ime i implicitno mu se dodeljuju klasa za skladištenje i ostali atributi određeni specifikatorima. Specifikatori i deklarator zajedno određuju da li se imenuje objekat, niz, pokazivač, referenca ili funkcija. Evo primera:

```
int *x, f();
```

Ovim smo deklarisali da je `x` pokazivač na tip `int` i da funkcija `f` vraća celobrojnu vrednost. Upravo deklaratori `*x` i `f()` prave razliku između `x` i `f`.

*deklarator:[* [* const *]* | & *]* ... *direktni-deklarator*

direktni-deklarator:identifikator-deklaratora | *(deklarator)* |
direktni-deklarator (*deklaraciona-lista-parametara*) |
direktni-deklarator [*konstantni-izraz*]

identifikator-deklaratora je *identifikator, koji može biti kvalifikovan*:

identifikator-deklaratora: [specifikator-ugnežđenog-imena] *identifikator*

specifikator-ugnežđenog-imena:{ *ime-klase-ili-imenskog-prostora* :: } ...

Direktni-deklarator koji se sastoji samo od *identifikatora-deklaratora* deklariše identifikator sa svojstvima opisanim *specifikatorima-deklaratora*, bez daljih izmena. Na primer, u deklaraciji

```
int n;
```

`n` je *direktni-deklarator* koji se sastoji samo od *identifikatora-deklaratora*, pa mu se pridružuje tip `int`.

Ako je oblik deklaratora drugačiji, njegov tip se može odrediti na sledeći način: prvo, neka je `T` tip određen na osnovu *specifikatora-deklaratora* (pri čemu se ignorišu rezervisane reči koje se ne odnose na tip, na primer `friend` ili `static`) i neka je `D` deklarator. Pratite sledeće korake dok ne svedete `D` na *identifikator-deklaratora*; `T` će, onda, biti tip koji tražite:

1. Ako `D` ima oblik (`D1`), zamenite `D` sa `D1`.
2. Ako `D` ima oblik `*D1` ili `*const D1`, zamenite `T` sa „pokazivač na tip `T`“ ili „konstantni pokazivač na tip `T`“ (*ne* sa „pokazivač na tip `const T`“), zavisno od toga da li je upotrebljena rezervisana reč `const`. Tada `D` zamenite sa `D1`.
3. Ako `D` ima oblik `D1` (*deklaraciona-lista-parametara*), `T` zamenite sa „funkcijom koja vraća tip `T`“, čiji su argumenti određeni *deklaracionom-listom-parametara*, a zatim `D` zamenite sa `D1`.
4. Ako `D` ima oblik `D1` [*konstantni-izraz*], `T` zamenite sa „niz elemenata tipa `T`“, koji sadrži onoliko elemenata koliko je određeno *konstantnim-izrazom*, a `D` zamenite sa `D1`.
5. Na kraju, ako deklarator ima oblik `&D1`, `T` zamenite sa „referenca na tip `T`“, a `D` sa `D1`.

Evo primera:

```
int *f();
```

Tip `T` je `int`, a `D` je pokazivač na funkciju `f*()`, pa `D` ima oblik `*D1`, a `D1` je `f()`.

Mogli biste pomisliti da deklarator `D` ima ili oblik `D1()` ili `*D1`. Međutim, kada bi `D` imao oblik `D1()`, onda bi `D1` morao da bude pokazivač `*f`, a `D1` bi takođe morao biti *direktni-deklarator* (jer gramatika utvrđena na početku ovog poglavlja dozvoljava da paru običnih zagrada `()` prethodi samo *direktni-deklarator*). Ali, ako pogledamo definiciju *direktnog-deklaratora*, videćemo da on ne može da sadrži operator `*`. Zato deklarator `D` može biti samo pokazivač `*f()`, koji ima oblik `*D1`, gde je `D1` operacija `f()`.

Pošto smo rešili da je `D1` funkcija `f()`, znamo da `T` moramo zameniti sa „pokazivač na tip `T`“, što je, u stvari „pokazivač na tip `int`“, kao i da `D` treba da zamenimo sa `f()`.

`D` još nismo sveli na *identifikator-deklaratora*, pa moramo sve ponoviti. Ovoga puta, `D1` može biti samo `f`, pa `T` postaje „funkcija koja vraća tip `T`“, što je, u stvari, „funkcija bez argumenata koja vraća pokazivač na tip `int`“, a deklarator `D` menjamo sa `f`.

Sada smo deklarator `D` sveli na *identifikator-deklaratora* i završili posao. Utvrdili smo da deklaracija

```
int *f();
```

određuje da `f` predstavlja „funkciju bez argumenata koja vraća pokazivač na tip `int`".

Kao drugi primer, pogledajmo sledeću deklaraciju:

```
int* p, q;
```

Ova deklaracija ima dva deklaratora, `*p` i `q`. Za svaki od njih važi da je tip `T`, u stvari, `int`. Za prvi deklarator, `D` je pokazivač `*p`, pa `T` menjamo sa „pokazivač na tip `int`", a `D` sa `p`. Ova deklaracija, znači, kaže da je `p` „pokazivač na tip `int`".

Analizirajmo posebno drugi deklarator. `T` je ponovo tip `int`, a `D` je `q`. Trebalo bi da je već jasno da je promenljiva `q` tipa `int`.

Analizirajmo, na kraju, tajanstveni primer iz odeljka 10.1.2:

```
double (*get_analysis_ptr())(const vector<Student_info>&);
```

Analiza ima pet ključnih tačaka:

1. `T:double; D: (*get_analysis_ptr())(const vector<Student_info>&);`
2. `T:`funkcija sa povratnim tipom `double` i argumentom tipa `const vector<Student:info>&; D: (*get_analysis_ptr())`
3. `T:`funkcija s povratnim tipom `double`... (kao i pre); `D:*get_analysis_ptr()`
4. `T:`pokazivač na funkciju s povratnim tipom `double`...; `D:get_analysis_ptr()`
5. `T:`funkcija koja vraća pokazivač na funkciju sa povratnim tipom `double`...; `D:get_analysis_ptr`.

Drugim rečima, utvrdili smo da je `get_analysis_ptr` funkcija koja vraća pokazivač na funkciju s povratnim tipom `double` i čiji je jedini argument konstantni vektor (`const vector<Student_info>&`). Kao vežbu, „razložite" tip `const vector<Student_info>&`.

Na sreću, mali je broj funkcija sa tako zbunjujućom deklaracijom, koja uglavnom ima ovakav oblik:

deklarator:identifikator-deklaratora (*deklaratorska_lista-parametara*)

Deklaracije funkcija koje vraćaju pokazivač na funkcije su, bez premca, najkomplikovanije.

A.2 Tipovi

Tipovi su neizostavni u svim programima na jeziku C++. Svaki objekat, izraz i funkcija imaju tip, koji određuje njihovo ponašanje. Uz jedan izuzetak (tip objekta iz neke hijerarhije nasleđivanja kome se pristupa preko pokazivača ili reference), tip svakog entiteta se određuje u vreme prevođenja.

U jeziku C++, tipove možemo posmatrati kao sredstvo za strukturiranje i pristupanje memoriji, kao i za definisanje operacija koje se mogu izvoditi nad objektima datog tipa. Drugim rečima, tipovi određuju i svojstva podatka i operacije koje se mogu izvoditi nad tipom.

Iako je ova knjiga namenjena savladavanju složenijih struktura podataka, veoma je važno da prvo shvatite proste tipove pomoću kojih se grade komplikovaniji oblici podataka. Ti prosti tipovi predstavljaju jednostavne apstraktne oblike koji su zasnovani na hardveru računara, kao što su brojevi (celi i realni), znakovi (uključujući široke znakove za neke zahtevnije jezike), logičke vrednosti i mašinske adrese (pokazivači,

reference i nizovi). Literali, često zvani konstante, mogu biti celi brojevi, brojevi u formatu pokretnog zareza, logičke vrednosti, znakovi ili znakovni nizovi. U narednom odeljku se podsećamo ugrađenih tipova i proširujemo znanja o načinima njihovog korišćenja.

A.2.1 Integralni tipovi

C++ je od jezika C nasledio pomalo zbunjujuću grupu integralnih tipova, koji uključuju cele brojeve, logički tip i znakovne tipove. Pošto je predviđeno da jezik C++ efikasno radi na različitim hardverskim podlogama, ovaj jezik mnoge detalje osnovnih tipova prepušta izvršnom okruženju umesto da te tipove precizno definiše.

A.2.1.1 Celi brojevi

Postoje tri zasebna označena celobrojna tipa i isto toliko neoznačenih celobrojnih tipova:

`short int`	`int`	`long int`
`unsigned short int`	`unsigned int`	`unsigned long int`

Umesto `short int` i `long int` možete skraćeno pisati `short` i `long`. Ako se ime jednog tipa sastoji od više rezervisanih reči, one se mogu pisati bilo kojim redom.

Pomoću ovih tipova možete predstaviti svaki ceo broj iz opsega koji definiše izvršno okruženje. Svaki od ovih tipova (izuzev prvog) mora da pokriva opseg jednak ili veći od onog koji odgovara prethodnom tipu. Opsezi celobrojnih vrednosti za tip `short` moraju da budu najmanje ±32767 ($\pm(2^{15}-1)$), dok je minimalan opseg koji mora da zadovolji tip `long int` ±2.147.483.647($\pm(2^{31}-1)$).

Svakom označenom integralnom tipu odgovara po jedan neoznačeni. Neoznačeni tipovi predstavljaju nenegativne cele brojeve iz opsega određenog vrednošću 2^n, gde n zavisi od tipa i izvršnog okruženja. Kao i za neoznačene tipove, vrednost n koja odgovara neoznačenom tipu (izuzev tipa `unsigned char`) mora biti veća ili jednaka vrednosti n pridruženoj prethodnom tipu. Osim toga, svaki neoznačeni tip mora obuhvatiti svaku nenegativnu vrednost iz opsega pridruženog odgovarajućem označenom tipu. Svaki označeni tip nenegativne vrednosti interno mora da predstavlja na isti način kao i odgovarajući neoznačeni tip. Iz ovoga se može zaključiti da četiri neoznačena tipa moraju da imaju dodatan bit koji odgovara bitu znaka odgovarajućeg označenog tipa. To znači da vrednost n za četiri neoznačena tipa mora biti najmanje 8, 16, 16 i 32, redom.

Prevodioci mogu vrednosti označenih tipova da predstavljaju i u potpunom i u nepotpunom komplementu.

Standardna biblioteka definiše tip `size_t` kao sinonim za neoznačene tipove. On je garantovano dovoljno veliki da sadrži podatak o veličini najvećeg mogućeg objekta, uključujući tu i nizove. Ovaj tip je definisan u zaglavlju `<cstddef>`.

Celobrojni literali: Celobrojni literal je sekvenca cifara, kojoj može da prethodi indikator osnove, a može da je sledi indikator veličine. Osim toga, celobrojni literali nemaju predznak, pa je `-3` izraz, ne literal.

Ako literal počinje sa `0x` ili `0X`, ceo broj je predstavljen u heksadecimalnom zapisu, u kome cifre, pored uobičajenih decimalnih, mogu biti i `AaBbCcDdEeFf`. Ako se na početku literala nalazi `0` bez `x` ili `X`, ceo broj je predstavljen u oktalnom zapisu, a dozvoljene cifre su `01234567`.

Indikatori veličine su `u`, `l`, `ul` ili `lu` (slovo `u` može biti i malo i veliko). Indikatori veličine `lu` ili `ul` ukazuje na to da je tip literala neoznačeni `long` (`unsigned long`). Ako je indikator veličine `u`, literalu je dodeljen neoznačeni tip (pod uslovom da data vrednost odgovara). Indikator veličine `l` označava da je u pitanju tip `long` (vrednost mora da bude prihvatljiva za ovaj tip); u suprotnom, literal je tipa `unsigned long`.

Ako je naveden indikator osnove, ali ne i indikator veličine, tip je prvi u skupu `int`, `unsigned`, `long` i `unsigned long` koji odgovara datoj vrednosti. Ukoliko nije naveden nijedan indikator, tip vrednosti je `int`, ako je to moguće; u suprotnom, u pitanju je tip `long`.

Ova pravila ukazuju na to da tip celobrojnog literala često zavisi od izvršnog okruženja. Na sreću, celobrojni literali u dobro napisanim programima obično su mali, pa ovakvi detalji uglavnom i nisu važni. Pominjemo ih za svaki slučaj, ako nekada budete morali da ih primenite.

A.2.1.2 Logičke vrednosti

Rezultati izraza koji predstavljaju uslove su logičke vrednosti (tipa `bool`). Logička vrednost može biti `true` (tačno) i `false` (netačno). Broj ili pokazivač mogu se upotrebiti kao logička vrednost. U tom slučaju, nuli se pridružuje vrednost `false`, dok se sve ostale vrednosti pretvaraju u `true`.

Kada se logička vrednost pretvori u broj, `false` je `0`, a `true` je `1`.

Literali logičkih vrednosti: Jedini literali logičkih vrednosti su `true` i `false`, sa očiglednim značenjem.

A.2.1.3 Znakovi

U jeziku C++, znakovi (engl. *characters*) su samo mali celi brojevi. Mogu se koristiti u aritmetičkim izrazima na isti način kao i celi brojevi.

Znakovi mogu, kao i celi brojevi, biti označeni ili neoznačeni, tako da razlikujemo tip `signed char` i `unsigned char`. Od svakog izvršnog okruženja se očekuje da za označene znakove definiše opseg vrednosti koji će sadržavati bilo koji znak iz osnovnog mašinskog skupa znakova i koji će u najmanju ruku biti veličine ± 127 ($\pm 2^7 - 1$)).

Osim toga, postoji i običan tip `char`, koji, i ako je odvojen od prethodna dva tipa, mora da se predstavlja na isti način kao i oni. Izvršno okruženje odlučuje koji će od tih tipova da primeni. Naravno, to treba da bude najprirodniji izbor za računar.

Postoje i „široki znakovi“ (`wchar_t`) koji moraju da sadrže najmanje 16 bita. Ovaj tip je namenjen predstavljanju znakova u jezicima kao što je japanski (odnosno u jezicima sa mnogo većim brojem znakova u odnosnu na abecedu). Tip `wchar_t` mora da se ponaša kao i bilo koji drugi integralni tip. Drugi tip koji se koristi zavisi od izvršnog okruženja i obično se bira tako da na najbolji način predstavi datu vrednost.

Znakovni literali: Znakovni literal, na primer `'a'`, obično je pojedinačan znak napisan pod polunavodnicima. Tip ovakvog literala je `char`, koji je, znamo od ranije (odeljak A2.1.3), celobrojni tip. Svako izvršno okruženje definiše kako su znakovi, na primer `a`, vezani za odgovarajuće celobrojne vrednosti. Većina programa ne zavisi od te veze jer kada programeri napišu literal, u našem slučaju `'a'`, on automatski znači „ceo broj koji odgovara znaku `a`“. Pošto veza između znaka i celog broja zavisi od izvršnog okruženja, programeri ne treba da se oslanjaju na aritmetička svojstva znakova. Na primer, ništa ne garantuje da će rezultat izraza `'a' + 1` biti literal `'b'`. S druge strane,

kada radite sa ciframa, sigurni ste da će rezultat ovakvog izraza biti susedna cifra. Na primer, izraz `'3' + 1` uvek će imati vrednost `'4'` (ali brojna vrednost obično nije 4).

Literali znakovnog niza: Literal znakovnog niza, na primer `"Zdravo, svete!"`, jeste sekvenca proizvoljnog broja znakova (može i nula) pod navodnicima. Tip literala znakovnog niza je `const char*`. Na kraj svakog ovakvog literala, prevodilac stavlja znak null.

Dva literala znakovnog niza razdvojena samo razmakom automatski se nadovezuju u nov, duži literal iste vrste. To omogućava da literale znakovnog niza napišete u više redova i povećate preglednost programa.

A.2.1.4 Predstavljanje znakova

Rekli smo da je literal znaka obično usamljen znak pod polunavodnicima, a da je literal znakovnog niza sekvenca znakova pod navodnicima. Naravno, postoje izuzeci među obe vrste literala:

- Da biste predstavili navodnike iste vrste kao i granice literala, morate pre njih dopisati obrnutu kosu crtu, kao u `'\'` ili `"ovo su \"navodnici\""` i time označiti da se literal ne završava tim navodnicima.
- Da biste predstavili obrnutu kosu crtu, morate pre nje dopisati istu takvu crtu, kao u `'\\'`, čime ukazujete prevodiocu da ta obrnuta kosa crta nije u funkciji označavanja nekog od specijalnih znakova.
- Neka pravila vezana za ovu tematiku odnose se na skupove internacionalnih znakova, što prevazilazi sadržaj ove knjige. Treba samo da znate da postoje pravila u kojima veći broj susednih znakova ima specijalno značenje. Da bi se omogućilo da nekoliko znakova pitanja stoje jedan do drugog u znakovnim nizovima, jezik C++ dozvoljava da se znak pitanja predstavi pomoću kose crte `\?`. Na taj način, literale kao što je `"Šta?\?"` možete pisati bez susednih znakova pitanja.
- Mnogobrojni kontrolni znaci, koji na razne načine utiču na rezultat programa, mogu se u vidljivom obliku predstaviti unutar literala: znak za novi red (`\n`), horizontalni tabulator (`\t`), vertikalni tabulator (`\v`), znak za brisanje (`\b`), znak za povratak na početak reda (`\r`), znak za kraj strane (`\f`) i alarm (`\a`). Kakve su posledice ispisivanja ovih znakova na izlaznom uređaju, zavisi od izvršnog okruženja.
- Ako vam treba znak koji se interno predstavlja na poseban način, možete ga napisati kao \x koje slede heksadecimalne cifre (dozvoljena su i mala i velika slova), ili ga možete predstaviti pomoću obrnute kose crte \ iza koje su tri oktalne cifre. Tako, na primer, `'\x20'` i `'\40'` predstavljaju znak odgovara decimalnoj cifri 32 (20 u heksadecimalnom sistemu i 40 u oktalnom sistemu). U skupu znakova ASCII, taj znak je razmak. Ovakvo predstavljanje se uglavnom (a u mnogim programima i jedino) primenjuje kada se literalom `'\0'` navodi znak sa vrednošću nula.

A.2.2 Realni brojevi

Jezik C++ ima tri tipa realnih brojeva u formatu pokretnog zareza: `float`, `double` i `long double`. Od svakog izvršnog okruženja se zahteva da tipu `float` dodeli najmanje 6 značajnih (decimalnih) cifara, a tipovima `double` i `long double` najmanje 10 ovakvih cifara. Većina izvršnih okruženja dodeljuje samo 6 značajnih cifara tipu `float`, a tipu `double` petnaest.

Literali u formatu pokretnog zareza: Literal u formatu pokretnog zareza je sekvenca od najmanje jedne cifre, sa eksponentom na kraju i/ili decimalnom tačkom unutar sekvence. Kao i celobrojni literali, ni ovi literali nisu označeni: `-3.1` je izraz, ne literal. Decimalna tačka može biti na početku, u sredini ili na kraju sekvence cifara, ili se može izostaviti ako se koristi eksponent. Eksponent predstavlja slovo `e` ili `E` i, (ako treba), znak za polaritet i jedna ili više cifara. Eksponent se uvek predstavlja u decimalnom sistemu.

Na primer, `312E5` i `31200000.` predstavljaju isti broj, ali `31200000` je celobrojni literal, ne literal u formatu pokretnog zareza. Drugi primer je literal `1.2E-3`, koji predstavlja isti broj kao i `.0012`, što se može napisati i kao `0.000012e+2`.

Tip literala u formatu pokretnog zareza obično je `double`. Ako hoćete da ovakvom literalu pridružite tip `float`, možete mu dopisati indikator `f`, odnosno `F`, a ako želite tip `long double`, dadajte oznaku `l` ili `L`.

A.2.3 Konstantni izrazi

Konstantan-izraz je izraz čija je vrednost integralnog tipa i određuje se prilikom prevođenja. Operandi ovakvog izraza mogu da sadrže samo literale, nabrojive konstante, konstantne promenljive ili statičke podatke članove integralnog tipa inicijalizovane pomoću *konstantnih-izraza* ili izraza `sizeof`. Konstantni izrazi ne smeju da sadrže funkcije, objekte klasa, pokazivače ili reference, niti da koriste operator dodele, uvećanja, umanjenja, pozive funkcija ili operator zarez.

Konstantan-izraz se može pojaviti gde god se očekuje konstanta. To podrazumeva dimenziju u deklaraciji niza (odeljak 10.1.3), oznake u iskazima *switch* (odeljak A.4) i inicijalizatore za nabrojive tipove (odeljak A.2.5).

A.2.4 Konverzije

Do konverzija dolazi kada je potrebno da operandi operatora budu istog tipa. Kada je moguće, uvek će se izabrati ona konverzija prilikom koje se informacije zadržavaju, odnosno ne gube. Osim toga, prednost imaju konverzije u neoznačene tipove (u odnosu na konverzije u označene tipove). Prilikom aritmetičkih operacija tip `short` ili znakovi pretvaraju se u tip `int` (ili neki tip većeg kapaciteta), a prilikom operacija u formatu pokretnog zareza dolazi do implicitne konverzije u tip `double` ili neki s većim opsegom vrednosti.

Najjednostavnije konverzije su unapređenja (engl. *promotions*). Prilikom unapređenja, vrednosti tipova manjeg kapaciteta (npr. `char`) konvertuju se u srodan tip većeg kapaciteta (npr. `int`), pri čemu se znak početne vrednosti ne menja. Unapređenja integralnih tipova podrazumevaju konverziju vrednosti tipa `char`, `signed char`, `unsigned char`, `short` ili `unsigned short` u tip `int` ako on može da ih prihvati, ili u tip `unsigned int`. Široki znakovi i nabrojivi tipovi (odeljak A.2.5) unapređuju se u najmanji celobrojni tip koji može da predstavi sve vrednosti početnog tipa (redom se proba konverzija u tip `int`, `unsigned int`, `long` i `unsigned long`), logičke vrednosti (`bool`) unapređuju se u tip `int`, a tip `float` u tip `double`.

Prilikom konverzije integralnog tipa u vrednost u formatu pokretnog zareza, rezultujuća vrednost ima maksimalnu preciznost koja se na datom računaru može postići.

Konverziju nekog dužeg označenog tipa (npr. `short`) u tip manjeg kapaciteta (npr. `long`) određuje izvršno okruženje. Konverzija dužeg neoznačenog tipa u manji tip predstavlja operaciju modulo od 2^n, gde je n broj bita koji zauzima ovaj drugi tip. Pri konverziji vrednosti u formatu pokretnog zareza u integralni tip, cifre iza decimalne

tačke se odbacuju. Ako je rezultat prevelik za integralni tip, nije definisano šta će se desiti.

Pokazivači, integralni tipovi i vrednosti u formatu pokretnog zareza mogu se konvertovati u logičke vrednosti. Ako je originalna vrednost `0`, odgovarajuća logička vrednost je `false`; u suprotnom, rezultat je `true`. Logičke vrednosti se mogu konvertovati u druge tipove: vrednost `true` postaje `1`, a vrednosti `false` pridružuje se vrednost `0`.

Konstantan-izraz (odeljak A.2.3) koji vraća vrednost `0` može se konvertovati u pokazivač.

Svaki pokazivač se može pretvoriti u tip `void*`. Osim toga, pokazivač na vrednost koja nije konstanta može se konvertovati u pokazivač na konstantu. Slično je i sa referencama na vrednosti koje nisu konstantne. Pokazivač ili referenca na objekat klase koja javno nasleđuje neku drugu klasu može se konvertovati u pokazivač ili referencu na objekte osnovne klase.

Aritmetičke konverzije: Određivanje tipa rezultata aritmetičkih operacija nije uvek jednostavno, samim tim što operandi mogu biti označeni ili neoznačeni, integralnog tipa ili u formatu pokretnog zareza. Pri ovome pomažu pravila koja zovemo ***uobičajene aritmetičke konverzije*:**

- Ako je bar jedan operand u formatu pokretnog zareza i rezultat će biti takav i nasledиće veću preciznost od operanada.
- U suprotnom, ako je tip nekog operanda `unsigned long`, to će biti tip rezultata.
- U suprotnom, ako je jedan operand tipa `long int`, a drugi je neoznačen (izuzev tipa `unsigned long` jer bi u skladu s prethodnim pravilom rezultat bio istog tipa), rezultat bi zavisio od izvršnog okruženja: ako opseg vrednosti za tip `long int` sadrži opseg vrednosti `unsigned int`, tip rezultata je `long int`; inače, rezultat je tipa `unsigned int`.
- U suprotnom, ako je bilo koji operand tipa `long int`, takav je i rezultat.
- U svim ostalim situacijama, operandi su ili označene celobrojne vrednosti tipa `int` ili nekog kraćeg tipa, pa je i tip rezultata `int`.

Posledica ovih pravila je da tip rezultata aritmetičke operacije nikada nije `short` ili `char` (bilo označeni ili neoznačeni). Ove operacije rade samo sa tipovima `int` ili većim.

A.2.5 Nabrojivi tipovi

Nabrojivi tip definiše skup celobrojnih vrednosti. Objekti tog tipa mogu da imaju samo one vrednosti koje određuje dati tip:

```
enum ime-nabrojivog-tipa {
                nabrojiva-vrednost [ , nabrojiva-vrednost ]...
};
```

ime-nabrojivog-tipa može se koristiti gde god se očekuje *ime-tipa*.

Promenljive nabrojivog tipa mogu imati samo one vrednosti koje su navedene u listi *nabrojivih vrednosti*:

nabrojiva-vrednost:identifikator [=konstantni-izraz]

Ako nije drugačije definisano, vrednostima nabrojivih tipova redom se pridružuju celobrojne vrednosti, počev od nule. Takođe, moguće je izričito dodeliti vrednosti. Inicijalizatori moraju biti integralnog tipa (odeljak A.2.1), a njihova vrednost određena prilikom prevođenja (odeljak A.2.3). Vrednost nabrojivog tipa se, po potrebi, automatski konvertuje u celobrojnu vrednost.

A.2.6 Preklapanje

Funkcije mogu imati isto ime ako se razlikuju u broju ili tipovima parametara.

Kada se pozove preklopljena funkcija, prilikom prevođenja se proverava koja se funkcija iz skupa preklopljenih funkcija tačno poziva. Prevodilac određuje koju će od tih funkcija pozvati tako što poredi argumente s tipovima parametara funkcija i bira onu funkciju čiji su parametri najbolje usklađeni sa argumentima. Bira se ona funkcija čijim parametrima odgovara najviše argumenata u odnosu na ostale funkcije (pri čemu je obavezno da tipovi barem jednog argumenta i parametra budu isti).

Najbolje podudaranje za određeni argument se definiše na sledeći način:

- Najpoželjnije je potpuno podudaranje (tip argumenta i parametra je isti).
- Podudaranje do kojeg dolazi nakon unapređenja (odeljak A.2.4) bolje je od onog koje podrazumeva ugrađene konverzije, što je prihvatljivije od podudaranja ostvarenog pomoću konverzija definisanih tipom klase (odeljci 12.2 i 12.5).

Ako se parametri više funkcija istovetno podudaraju s datim argumentima, u pitanju je greška.

A.3 Izrazi

C++ je od jezika C nasledio bogatu sintaksu za izraze, koju je unapredio preklapanjem operatora (odeljak A.3.1). Preklapanje operatora dozvoljava programerima da definišu argument i povratni tip, kao i značenje operatora, ali ne i njihove prioritete, broj operanda ili asocijativnost. Nije moguće promeniti značenje ugrađenih operatora nad podacima ugrađenih tipova. U ovom odeljku ćemo opisati (i neke nove) operatore koji rade sa ugrađenim tipovima.

Svaki operator vraća vrednost, čiji tip zavisi od tipa operanada. U osnovi, ako operandi imaju isti tip, to je i tip rezultata. U suprotnom, standardnom konverzijom se operandi svode na isti tip (odeljak A.2.4).

Lvrednost (engl. *lvalue*) je vrednost objekta koji nije privremen (što znači da ima trajnu adresu). Neke operacije podržavaju samo lvrednosti, dok ih neke vraćaju. Svaki izraz vraća neku vrednost; neki od njih vraćaju lvrednosti.

Redosled izračunavanja operanada samo četiri operatora tačno je definisan:

`&&` Desni operand se računa samo ako je vrednost levog operanda `true`.

`||` Desni operand se računa samo ako je vrednost levog operanda `false`.

`? :` Računa se samo jedan izraz iza uslova. Izraz iza znaka pitanja se računa ako uslov ima vrednost `true`; u suprotnom, računa se izraz iza dvotačke `:`. Rezultat je izraz koji se računao; to je lvrednost ako su oba izraza bila lvrednosti istog tipa.

`,` Prvo se računa levi operand, a rezultat se zanemaruje; rezultat izraza je desni operand.

U slučaju ostalih operatora, osim pravilima prioriteta, redosled računanja operanada nije definisan. Drugim rečima, prevodilac može da računa operande bilo kojim redom. Podrazumevani prioriteti se mogu izmeniti pomoću zagrada, ali se redosled izračunavanja može potpuno odrediti samo pomoću privremenih objekata kojima se izričito zadaju vrednosti.

Prioritet i asocijativnost svakog operatora strogo su definisani i ne mogu se promeniti. U narednoj tabeli smo predstavili sve operatore, poređane po prioritetu. Svi operatori u grupi imaju isti prioritet. Ova tabela je potpunija od tabele iz poglavlja 2 i obuhvata sve operatore:

simbol	Identifikator ili literal; identifikatori su lvrednosti, literali nisu.
`C::m`	Član `m` klase `C`.
`N::m`	Član `m` imenskog prostora `N`.
`::m`	Ime `m` iz globalne oblasti važenja.
x[y]	Element objekta `x` sa indeksom `y`. Vraća lvrednost.
`x->y`	Član `y` objekta na koji pokazuje `x`. Ako je `x` lvrednost, takav je i rezultat.
`x.y`	Član `y` objekta `x`. Vraća lvrednost ako je i `x` lvrednost.
`f(args)`	Poziva funkciju i prosleđuje joj argumente navedene u listi *args*.
`x++`	Uvećava lvrednost `x`. Vraća originalnu vrednost operanda `x`.
`x--`	Umanjuje vrednost `x`. Vraća originalnu vrednost operanda `x`.
`*x`	Dereferencira pokazivač `x`. Vraća podatak na koji je pokazivač pokazivao, ali kao lvrednost.
`&x`	Adresa objekta `x`. Vraća pokazivač na podatak `x`.
`-x`	Unarni minus. Može se koristiti samo u numeričkim izrazima.
`!x`	Logička negacija. Ako je vrednost `x` nula, rezultat izraza `!x` je `true`; u suprotnom je rezultat `false`.
`~x`	Nepotpuni komplement operanda `x`, koji mora biti integralnog tipa.
`++x`	Uvećava lvrednost `x`. Vraća uvećanu vrednost kao lvrednost.
`--x`	Umanjuje lvrednost `x`. Vraća umanjenu vrednost kao lvrednost.
`sizeof(e)`	Broj bajtova (tipa `size_t`) koji koristi izraz `e`.
`sizeof(T)`	Broj bajtova (tipa `size_t`) koji zauzima objekat tipa `T`.
`T(arg)`	Pomoću argumenata iz liste `arg` pravi objekat tipa `T`.
`new T`	Pravi nov, podrazumevano inicijalizovan objekat tipa `T`.
`new T(args)`	Pravi nov objekat tipa `T` inicijalizovan pomoću argumenata iz liste *arg*.
`new T[n]`	Pravi niz od `n` podrazumevano inicijalizovanih objekata tipa `T`.
`delete p`	Uništava objekte na koje pokazuje pokazivač `p`.
`delete [] p`	Uništava niz objekata na koji pokazuje pokazivač `p`.
`x * y`	Proizvod operanada `x` i `y`.
`x / y`	Količnik operanada `x` i `y`. Ako su oba operanda celobrojna, izvršno okruženje odlučuje da li da rezultat zaokružuje ka `0` ili ka $-\infty$.
`x % y`	`x - ((x / y) * y)`.
`x + y`	Zbir operanada `x` i `y` ako su u pitanju numerički operandi. Ako je jedan operand pokazivač, a drugi je integralnog tipa, rezultat je pokazivač na poziciju koja je od `x` udaljena `y` mesta.

`x - y`	Razlika između vrednosti operanada `x` i `y`, ako su oba operanda numerička. Ako su `x` i `y` pokazivači, ovaj izraz vraća rastojanje (broj elemenata) između pozicija na koje oni ukazuju.
`x >> y`	Ako su operandi `x` i `y` integralnog tipa, bitovi vrednosti `x` se pomeraju udesno za `y` mesta(vrednost `y` ne sme biti negativna). Ukoliko je `x` objekat klase `istream`, izraz učitava sadržaj iz `x` u objekat `y` i vraća lvrednost `x`.
`x << y`	Ako su operandi `x` i `y` integralnog tipa, bitovi vrednosti `x` se pomeraju udesno za `y` mesta (vrednost `y` ne sme biti negativna). Ukoliko je `x` objekat klase `ostream`, izraz upisuje sadržaj `y` u objekat `x` i vraća lvrednost `x`.
`x relop y`	Operatori poređenja vraćaju logičku vrednost poređenja. Značenje operatora `<`, `>`, `<=` i `>=` je očigledno. Ako su `x` i `y` pokazivači, moraju da pokazuju na isti objekat ili niz.
`x == y`	Vraća logičku vrednost koja pokazuje da li su `x` i `y` jednaki.
`x != y`	Vraća logičku vrednost koja pokazuje da li su `x` i `y` različiti.
`x & y`	Konjunkcija (engl. *and*) nad bitovima. I `x` i `y` moraju biti integralnog tipa.
`x ^ y`	Isključiva disjunkcija (engl. *xor*) nad tipovima. I `x` i `y` moraju biti integralnog tipa.
`x | y`	Disjunkcija nad bitovima. I `x` i `y` moraju biti integralnog tipa.
`x && y`	Vraća logičku vrednost koja ukazuje na to da li je vrednost oba operanda `true`. Računa vrednost operanda `y` samo ako `x` ima vrednost `true`.
`x || y`	Vraća logičku vrednost koja pokazuje na to da li je vrednost bar jednog operanda `true`. Računa operand `y` samo ako `x` ima vrednost `false`.
`x = y`	Operandu `x` dodeljuje vrednost operanda `y`. Vraća `x` kao lvrednost.
`x` *`op`*`= x`	Složeni operator dodele. Izraz je ekvivalentan izrazu `x = x op y`, gde je *op* aritmetički operator, operator nad bitovima ili operator pomeranja.
`x ? y1 : y2`	Vraća `y1` ako `x` ima vrednost `true`; u suprotnom, vraća operand `y2`. Računa se vrednost ili operanda `y1` ili operanda `y2`, koji moraju biti istog tipa. Ako su `y1` i `y2` lvrednosti, i rezultat je takav. Ovaj operator je desno asocijativan.
`throw x`	Ukazuje na grešku tako što izaziva izuzetak koristeći vrednost operanda `x`. Tip operanda `x` određuje proceduru za obradu greške.
`x, y`	Računa vrednost operanda `x`, zanemaruje rezultat, zatim računa vrednost operanda `y`. Vraća `y`.

A.3.1 Operatori

Većina ugrađenih operatora može se preklapati. To ne važi za operator za izazivanje izuzetka, operator oblasti važenja, operator tačke, kao i uslovni operator (`? :`). Svi ostali operatori se mogu preklopiti. U odeljku 11.2.4 opisali smo kako se definišu preklopljeni operatori.

Sufiksna verzija operatora uvećanja i umanjenja razlikuje se od prefiksne verzije po tome što definišemo da ima (lažan) parametar koji se ne koristi. Tačnije, da bismo preklopili sufiksni operator, pišemo:

```
class Number {
public:
  Number operator++(int) { /* telo funkcije */ }
  Number operator--(int)   { /* telo funkcije */ }
};
```

Najčešće se preklapaju operatori dodele i indeksiranja, operatori pomeranja za upisivanje i ispisivanje vezani za objekte klasa `ostream` i `istream`, kao i operatori koje koriste iteratori, čiji pregled možete naći u odeljku B2.5.

A.4 Iskazi

Kao svi programski jezici, C++ pravi razliku između deklaracija, izraza i iskaza. U nekim situacijama, deklaracije i iskazi mogu se ugnezditi u druge deklaracije i iskaze, ali ne i unutar izraza. Svaki iskaz se, na kraju, pojavi u definiciji funkcije, gde predstavlja deo onoga što se dešava kada se ta funkcija pozove.

Ako se ne navede drugačije, iskazi u okviru funkcije izvršavaju se po redu kojim se pojavljuju. Izuzeci su petlje; pozivi funkcija; iskazi `goto`, `break` i `continue`, kao i iskazi `try` i `throw` vezani za obradu izuzetaka.

Iskazi se pišu u slobodnoj formi. Prelazak u novi red unutar iskaza ne menja smisao tog iskaza. Većina iskaza se završava tačkom i zarezom, ali i tu postoje izuzeci, od kojih je najznačajniji blok iskaza (počinje levom vitičastom zagradom {, a završava desnom takvom zagradom }).

`;` Prazan iskaz; njegovo izvršavanje nema nikakav efekat.

`e;` Iskazni izraz; računa izraz `e` radi sporednog efekta.

`{ }` Blok iskaza; iskazi unutar bloka se izvršavaju redom. Promenljive definisane u bloku uništavaju se na kraju bloka.

`if` (*uslov*) *iskaz1*

Računa uslov ako je rezultat tog ispitivanja `true`, izvršava *iskaz1*.

`if` *(uslov) iskaz1* `else` *iskaz2*

Računa uslov, i ako je rezultat tog ispitivanja `true`, izvršava *iskaz1*; u suprotnom, izvršava *iskaz2*. Svaki iskaz `else` se povezuje s najbližim slobodnim iskazom `if`.

`while` *(uslov) iskaz*

Ispituje uslov i ako je rezultat `true`, izvršava *iskaz*.

`do` *iskaz* `while` (*uslov*);

Izvršava *iskaz*, a zatim ispituje *uslov*. Nastavlja da izvršava *iskaz* dok rezultat računanja uslova ne bude `false`.

`for` (*inicijalni-iskaz uslov; izraz*) *iskaz*

Izvršava jednom *inicijalni-iskaz* na početku petlje, zatim računa *uslov*. Ako je vrednost uslova `true`, izvršava *iskaz*, zatim računa *izraz*. Nastavlja da proverava vrednost *uslova*, da izvršava *iskaz* i računa *izraz*, dok vrednost *uslova* ne bude `false`. Ako je *inicijalni-iskaz* deklaracija, oblast važenja deklarisane promenljive je *iskaz* `for`.

`switch` (*izraz*) *iskaz*

U praksi, *iskaz* je skoro uvek blok koji sadrži iskaze sa oznakama oblika

```
case vrednost:
```

gde svaka *vrednost* mora da bude drugačiji konstantni izraz vrednosti integralnog tipa (odeljak A.2.3). Pored toga, može se pojaviti i oznaka

```
default:
```

ali samo jednom.

U okviru izvršavanja iskaza `switch` izračunava se *izraz* i prelazi na oznaku `case` čija vrednost odgovara rezultatu *izraza*. Ako takva oznaka ne postoji, izvršava se iskaz sa oznakom `default:` ukoliko postoji. Ako oznaka `default` ne postoji, prelazi se na prvi iskaz iza bloka `switch`.

Pošto su oznake `case` samo oznake, program će prelaziti s jedne na drugu, dok programer to izričito ne promeni. To se najčešće radi pomoću iskaza `break` koji se, nakon prve oznake, dopisuje ispred svih ostalih.

`break;`

Prelazi na tačku programa koja se nalazi odmah iza najbližeg iskaza `while`, `for`, `do` ili `switch` čijem telu pripada.

`continue;`

Prelazi na početak narednog koraka (uključujući i uslov) najbližeg iskaza `while`, `for` ili `do` čijem telu pripada. Ako je najbliži takav iskaz `for`, naredni njegov korak će obuhvatiti i *izraz*. Evo primera:

```
for (int i = 0; i < 10; ++i) {
     if (i % 3 == 0)
         continue;
     cout << i << endl;
}
```

Ova petlja ispisuje vrednosti `1`, `2`, `4`, `7` i `8`, i to svaku u posebnom redu.

`goto` *oznaka;*

Ponaša se slično kao i na drugim jezicima. Odredište iskaza `goto` je oznaka, koja predstavlja identifikator iza kojeg su dve tačke. Oznake mogu da imaju isto ime kao i drugi entiteti. Oblast važenja oznake je funkcija u kojoj se pojavljuje, što znači da je moguće ući u blok s mesta izvan bloka. Međutim, takav skok se ne može iskoristiti da bi se izbegla inicijalizacija promenljive.

`try` (*iskazi*) `catch` (*parametar1*) { *iskazi1* }
`[ catch` (*parametar2*) { *iskazi2* } `] ...`

Izvršava *iskaze* koji mogu da izazovu izuzetak, koji treba obraditi pomoću jednog ili više narednih iskaza `catch`.

Iskaz `catch` obrađuje izuzetke čija je vrednost tipa sličnog tipu *parametra*, tako što izvršava odgovarajuće *iskaze*. „Slično“ ovde podrazumeva da je tip vrednosti izuzetka isti kao i tip parametra ili tip izveden iz tipa parametra.

Ako iskaz `catch` ima oblik `catch (...)`, onda obrađuje svaki izuzetak.

Ukoliko za dati izuzetak ne postoji odgovarajući iskaz `catch`, izuzetak će napredovati izvan funkcije do prvog sledećeg iskaza `try` u čijoj je oblasti važenja. Ako ni takav iskaz ne postoji, pogram se prekida.

`throw` *izuzetak*;

Okončava program ili kontrolu prepušta iskazu `catch` čije je izvršavanje u toku. Prosleđuje izuzetak na osnovu čijeg tipa se određuje blok `catch` koji može da ga obradi. Ako se u tom trenutku nijedan odgovarajući blok `try` ne izvršava, program se prekida.

Izuzeci su često objekti klasa, i obično se pojavljuju u jednoj funkciji, a obrađuju u drugoj.