Predavanje 1_b

Jezik C++ omogućava mnogo veću slobodu u deklaraciji i definiciji promjenljivih u odnosu na jezik C (pogotovo u odnosu na *starije dijalekte* jezika C, jer su noviji dijalekti jezika C, poput C99, preuzeli neke ideje iz jezika C++). Na primjer, razmotrimo sljedeći program, koji nalazi rješenja kvadratne jednačine, vodeći računa o tome da li su ona realna ili kompleksna (pri čemu se kompleksna rješenja ispisuju u obliku "*re* + *im* * `i`" odnosno "*re* - *im* * `i`"). U ovom primjeru, deklaracija promjenljivih je izvedena "u duhu" starijih dijalekata jezika C (podsjetimo se da funkcija "`fabs`" vraća apsolutnu vrijednost realnog broja zadanog kao argument):

```
#include <iostream>
#include <cmath>

int main() {
  double a, b, c, d, x1, x2, re, im;
  std::cout << "Unesite koeficijente a, b i c kvadratne jednacine:";
  std::cin >> a >> b >> c;
  d = b * b - 4 * a * c;
  if(d >= 0) {                                          /* realna rješenja */
    x1 = (-b - std::sqrt(d)) / (2 * a);
    x2 = (-b + std::sqrt(d)) / (2 * a);
    std::cout << "x1 = " << x1 << "\nx2 = " << x2 << std::endl;
  }
  else {                                                /* kompleksna rješenja */
    re = -b / (2 * a);
    im = std::fabs(std::sqrt(-d) / (2 * a));
    std::cout << "x1 = " << re << "+" << im << "*i\n"
      << "x2 = " << re << "-" << im << "*i\n";
  }
  return 0;
}
```

Naime, u starijim dijalektima jezika C se sve promjenljive unutar nekog bloka moraju deklarirati isključivo *prije prve izvršne naredbe bloka*. Ovo ograničenje ne postoji u jeziku C++. Štaviše, u C++-u se smatra veoma lošom praksom deklarirati promjenljivu prije nego što se za njom zaista pojavi potreba u programu. Dalje, u starijim dijalektima C-a se promjenljive prilikom deklaracije mogu inicijalizirati *isključivo konstantnim vrijednostima*, dok se u C++-u za inicijalizaciju mogu upotrijebiti *proizvoljni izrazi ispravnog tipa*. Slijedi ponovo isti primjer, ovaj put napisan u skladu sa stilom jezika C++:

```
#include <iostream>
#include <cmath>

int main() {
  double a, b, c;
  std::cout << "Unesite koeficijente a, b i c kvadratne jednacine:";
  std::cin >> a >> b >> c;
  double d = b * b - 4 * a * c;
  if(d >= 0) {                                          // realna rješenja
    double x1 = (-b - std::sqrt(d)) / (2 * a);
    double x2 = (-b + std::sqrt(d)) / (2 * a);
    std::cout << "x1 = " << x1 << "\nx2 = " << x2 << std::endl;
  }
  else {                                                // kompleksna rješenja
    double re = -b / (2 * a);
    double im = std::fabs(std::sqrt(-d) / (2 * a));
    std::cout << "x1 = " << re << "+" << im << "*i\n"
      << "x2 = " << re << "-" << im << "*i\n";
  }
  return 0;
}
```

Treba napomenuti da su ovakve slobodnije deklaracije promjenljivih ušle u novije standarde jezika C, kao što su C99, C11, C17 i C23, ali o toj činjenici se vrlo rijetko govori u udžbenicima posvećenim jeziku C (odnosno, udžbenici se uglavnom drže starih standarda koji su vrijedili prije standarda C99). U ovom programu, pored slobodnije deklaracije promjenljivih vidimo i novi način pisanja komentara koji je uveden u jeziku C++ i novijim dijalektima jezika C. Naime, pored načina za pisanje komentara naslijeđenog iz jezika C, po kojem se komentari omeđuju znakovima "`/*`" i "`*/`", u C++-u je komentar moguće započeti znakom "`//`" pri čemu se smatra da je komentar sve iza tog znaka pa do kraja reda u

kojem je on napisan. Ovakav način pisanja je jako praktičan za kratke jednolinijske komentare, dok je za dugačke komentare, koji se protežu u nekoliko redova, pogodnije koristiti način naslijeđen iz C-a.

Slično kao u C-u, opseg važenja (vidokrug) svake deklarirane promjenljive, završava se na kraju bloka unutar kojeg je promjenljiva deklarirana. U prethodnom primjeru, to na primjer znači da promjenjive "`x1`" i "`x2`" vrijede samo unutar bloka koji slijedi iza "**`if`**" naredbe, dok je opseg važenja promjenljivih "`re`" i "`im`" ograničen samo na blok koji slijedi iza "**`else`**". Pravilo dobre upotrebe promjenljivih u jeziku C++ nalaže da svaka promjenljiva treba da ima vidokrug koji ne treba da bude veći od opsega u kojem se ta promjenljiva zaista i koristi (tj. da joj vidokrug bude najmanji mogući). Na taj način se sprečava mogućnost da se neka promjenljiva greškom upotrijebi na mjestu gdje to nije bilo planirano.

Radi politike sužavanja opsega važenja promjenljivih na najmanju moguću mjeru, u jeziku C++ je omogućeno da se promjenljive mogu deklarirati unutar *prvog argumenta naredbe* "**`for`**". Opseg važenja takvih promjenljivih ograničen je samo na tijelo pripadne *for*-petlje. Drugim riječima, svaka takva promjenljiva *prestaje postojati* čim se petlja završi (tako da je istoimenu promjenljivu moguće kasnije ponovo deklarirati i upotrijebiti za nešto drugo). Na primjer, pogledajmo sljedeći C++ fragment, koji izračunava i ispisuje zbir recipročnih vrijednosti svih brojeva od 1 do 1000:

```
double suma = 0;
for(int i = 1; i <= 1000; i++) suma += 1. / i;
std::cout << suma;
```

U ovom primjeru, promjenljiva "`i`", deklarirana unutar *for*-petlje, postoji samo dok se petlja ne završi. U C++-u je preporuka da se brojačke promjenljive koje upravljaju radom *for*-petlji deklariraju isključivo na ovakav način. Na taj način se sprečava mogućnost da se brojačka promjenljiva nehotice neispravno upotrijebi, ili čak zloupotrijebi izvan tijela petlje. Recimo, pogledajmo ovaj (neispravni) isječak, pisan u duhu jezika C, s deklaracijom brojačke promjenljive "`i`" ispred petlje:

```
double suma = 0;
int i;
for(i = 1; i <= 1000; i++);                          // Podmukla greška...
  suma += 1. / i;
std::cout << suma;
```

Vidite li logičku grešku u ovom isječku, zbog kojeg on ispisuje pogrešan rezultat (0.000999001)? Pogrešan je tačka-zarez na kraju reda u kojem je *for*-petlja, zbog kojeg se smatra da je tijelo ove petlje *prazno*, tj. da mu naredba "`suma += 1. / i`" ne pripada. Ova naredba je zapravo *izvan tijela petlje* i izvršiće se tek nakon što se čitava petlja "izvrti naprazno". Nakon završetka petlje, promjenljiva "`i`" će imati vrijednost 1001, tako da će izračunata suma sadržavati samo član 1/1001. Da je promjenljiva "`i`" bila deklarirana da vrijedi lokalno samo unutar petlje, kompajler bi nam prijavio grešku da u naredbi "`suma += 1. / i`" promjenljiva "`i`" više nije dostupna, tako da bismo odmah shvatili da ova naredba zapravo nije unutar petlje (iako smo je napisali "uvučeno"), i da trebamo "skinuti" suvišni tačka-zarez (ili uklopiti tijelo petlje u vitičaste zagrade). U svakom slučaju, daleko je bolje da nam kompajler prijavi grešku, pa da odmah znamo da nešto nije u redu, nego da mislimo da nam program radi, a da daje pogrešan rezultat (za koji nećemo uvijek ni prepoznati da je pogrešan).

Naredbe "**`if`**", "**`while`**" i "**`switch`**" također dozvoljavaju da se unutar njihovog argumenta deklarira promjenljiva koja će vrijediti samo unutar tijela tih naredbi, ali je takva mogućnost mnogo manje korisna, jer je deklaraciju teško interpretirati kao uvjet (u tom slučaju, deklaracija se tumači kao "tačna" ukoliko je vrijednost pridružena promjenljivoj koja se deklarira različita od nule, a u suprotnom je "netačna"). Prilično su rijetke situacije u kojima ovo može biti korisno, ali postoje (na jednu od njih naići ćemo kasnije u ovom kursu). Od standarda C++17 nadalje, uvedene su i verzije naredbi "**`if`**" i "**`switch`**" s dva argumenta razdvojena tačka-zarezom (kao u naredbi "**`for`**"), pri čemu se u prvom argumentu obavi neka lokalna deklaracija, a u drugom argumentu navede se uvjet. Ovo je već daleko korisnije. Recimo, neka želimo testirati da li je $(\sqrt{x}+1)^2-3/(\sqrt{x}+1)$ pozitivno. Jedan način je da napišemo test poput "`if((std::sqrt(x) + 1) * (std::sqrt(x) + 1) - 3/((std::sqrt(x) + 1) > 0) …`", naravno uz pretpostavku da se vrijednost x nalazi u promjenljivoj "`x`", ali nedostatak ovog rješenja je što se isti podizraz "`std::sqrt(x) + 1`" računa tri puta. Bolje rješenje je sačuvati vrijednost tog podizraza u nekoj pomoćnoj promjenljivoj, recimo "`y`". Međutim, ukoliko tu pomoćnu promjenljivu deklariramo prije same naredbe "**`if`**", ona će postojati i nakon njenog tijela, što nije poželjno. Standard C++17 omogućava da to obavimo "lokalno", konstrukcijom poput "`if(double y = std::sqrt(x) + 1; y * y – 3 / y > 0) …`". Slična mogućnost je podržana i u naredbi "**`switch`**", ali ne i u naredbi "**`while`**", s obzirom da se nepostojeća

konstrukcija poput "`while(deklaracija; uvjet)`" svakako može simulirati koristeći naredbu "`for`" s izostavljenim trećim argumentom (tj. koristeći konstrukciju "`for(deklaracija; uvjet;)`").

Obratimo pažnju da smo u prethodnom primjeru pisali "`1.`" umjesto "`1`". Da to nismo uradili, imali bismo pogrešan rezultat. Naime, kako je "`1`" cjelobrojnog tipa i kako je promjenljiva "`i`" također cjelobrojna, operator "`/`" u izrazu poput "`1 / i`" interpretirao bi se kao *cjelobrojno djeljenje*. Ovako, kako je "`1.`" realnog tipa, operator "`/`" se interpretira kao *klasično dijeljenje*. Ovo je dobar povod da kažemo nešto o *konverziji tipova* u jeziku C++. C++ podržava skoro sve automatske konverzije tipova koji se dešavaju i u jeziku C (uz rijetke izuzetke koji će biti posebno istaknuti). Na primjer, cjelobrojni izrazi se automatski konvertiraju u realne ukoliko se upotrijebe u kontekstu u kojem se očekuje realan izraz, npr. ukoliko želimo dodijeliti neki cjelobrojni izraz realnoj promjenljivoj (inače, konverzija nekog cjelobrojnog tipa u drugi širi cjelobrojni tip ili nekog realnog tipa u širi realni tip često se naziva *promocija*, dok se taj naziv ne koristi za konverziju cjelobrojnih u realne tipove). S druge strane, realan izraz često možemo upotrijebiti u kontekstu u kojem se očekuje cjelobrojni izraz, npr. kao argument neke funkcije koja očekuje cjelobrojni argument ili pri dodjeli realnog izraza cjelobrojnoj promjenljivoj. Takve konverzije praćene su *gubitkom informacija* (u konkretnom primjeru *odsjecanjem decimala*) i obično ih nazivamo *degradacija* (engl. *demotion*). C++ kompajleri imaju pravo da nailaskom na degradirajuću automatsku konverziju emitiraju *poruku upozorenja*, ali *ne smiju odbiti da prevedu program*. Dalje, podsjetimo se da u jeziku C možemo izvršiti *eksplicitnu konverziju* proizvoljnog izraza "*izraz*" u neki tip "*tip*", pomoću tzv. *operatora konverzije tipa* (engl. *type-cast*) koji ima sintaksu

(tip)izraz

Ovaj operator se može koristiti i u jeziku C++. Pogledajmo, na primjer, sljedeći programski isječak koji ispisuje količnik dvije cjelobrojne vrijednosti unesene s tastature:

```
int broj_1, broj_2;
std::cin >> broj_1 >> broj_2;
std::cout << (double)broj_1 / broj_2;
```

Konverzijom zapravo na osnovu jednog objekta kreiramo *novi objekat*, koji ima istu vrijednost kao objekat koji konvertiramo, ali koji je zadanog tipa, koji je tipično drugačiji od tipa onog što konvertiramo (originalni objekat pri tome ostaje isti kakav je bio i ranije). U ovom isječku, pomoću eksplicitne konverzije na osnovu promjenljive "`broj_1`" koja je cjelobrojnog tipa kreiramo novi objekat koji je realnog tipa (tj. tipa "`double`"), ali koji ima *istu vrijednost* kakvu ima i promjenljiva "`broj_1`". Svrha ove konverzije je da izbjegnemo da operator "`/`" bude interpretiran kao cjelobrojno dijeljenje. Mada ovakva sintaksa i dalje radi u jeziku C++, ona se više *ne preporučuje* (ali ipak nije "pokuđena", nego je prosto "odraz lošeg stila"). C++ uvodi dvije nove sintakse za konverziju tipova. Prva je takozvana *funkcijska* ili *konstruktorska* notacija, koja izgleda ovako:

tip(izraz)

Dakle, u ovoj sintaksi, ime tipa se tretira kao *funkcija koja svoj argument prevodi u rezultat iste vrijednosti, ali odgovarajućeg tipa*. Odatle i potiče naziv *funkcijska notacija*, dok naziv *konstruktorska notacija* potiče od činjenice da se istom sintaksom jeziku C++ vrši konstrukcija ma kakvih objekata na osnovu određenih parametara, o čemu ćemo kasnije detaljno govoriti. Prethodni primjer bi se, u skladu s ovom sintaksom, napisao ovako:

```
int broj_1, broj_2;
std::cin >> broj_1 >> broj_2;
std::cout << double(broj_1) / broj_2;
```

Ova sintaksa može biti mnogo praktičnija kod složenijih izraza. Naime, zbog vrlo visokog prioriteta operatora konverzije tipa u C stilu, često su svakako potrebne dodatne zagrade. Na primjer, neka želimo izračunati i ispisati cijeli dio produkta "`2.541 * 3.17`". Koristeći C notaciju, to bismo uradili ovako:

```
std::cout << (int)(2.541 * 3.17);
```

Dodatne zagrade su neophodne, jer bi se u suprotnom operator konverzije "`(int)`" odnosio samo na podatak "`2.541`" (odnosno, da smo pisali samo "`(int)2.541 * 3.17`", kao rezultat bismo dobili vrijednost produkta "`2 * 3.17`"). Korištenjem funkcijske notacije, istu stvar možemo uraditi ovako:

```
std::cout << int(2.541 * 3.17);
```

Na ovaj način, ime tipa "`int`" se ujedno može interpretirati i kao *funkcija* koja vraća kao rezultat cijeli dio svog argumenta.

Funkcijska notacija posjeduje dva ograničenja – ime tipa u koji se vrši konverzija može se sastojati *samo od jedne riječi* i nije podržana konverzija u tzv. *anonimne tipove* (poput "`int *`", što predstavlja tip pokazivača na cijele brojeve). Na primjer, ukoliko želimo na osnovu promjenljive "`a`" kreirati novi objekat iste vrijednosti, ali tipa "`unsigned long int`", to *ne možemo* uraditi ovako:

```
unsigned long int(a)                    // SINTAKSNA GREŠKA!
```

ali možemo ovako (u C stilu):

```
(unsigned long int)a
```

Druga sintaksa za konverziju tipa uvedena u jeziku C++, koja ne posjeduje navedene nedostatke, koristi novu ključnu riječ "`static_cast`", a izgleda ovako:

```
static_cast<tip>(izraz)
```

U skladu s ovom sintaksom, prethodno napisani primjeri izgledali bi ovako:

```
int broj_1, broj_2;
std::cin >> broj_1 >> broj_2;
std::cout << static_cast<double>(broj_1) / broj_2;
```

odnosno

```
std::cout << static_cast<int>(2.541 * 3.17);
```

Ova sintaksa je dosta rogobatna, ali ona ima svoje prednosti. Prvo, mjesta gdje se vrše eksplicitne konverzije tipa su potencijalno opasna mjesta, koja često prave probleme kada se program treba prenijeti s jedne računarske arhitekture na drugu ili s jednog na drugi operativni sistem. Korištenjem ove sintakse, takva mjesta u programu je lako locirati prostim traženjem ključne riječi "`static_cast`". Drugo, konverzija tipa se za različite tipove u C++-u često vrši na suštinski različite načine. Pri upotrebi pretvorbe u stilu C-a, programer na to nema nikakvog uticaja, nego se prosto bira način pretvorbe za koji kompajler smatra da je najpogodniji u upotrijebljenom kontekstu. Postoje pravila kako se vrši taj odabir, ali na to programer nema nikakvog utjecaja. Stoga su u jeziku C++ uvedene četiri ključne riječi za konverziju tipova (pored već spomenute "`static_cast`", imamo i "`const_cast`", "`reinterpret_cast`" i "`dynamic_cast`"), čime programer može eksplicitnije da iskaže tačnu namjeru kakvu konverziju želi. U detalje na ovom mjestu nećemo ulaziti.

U vezi s tipovima i konverzijama tipova, treba istaći jednu razliku u tretmanu matematičkih funkcija između jezika C i C++. Naime, poznato je da u jezicima C i C++ postoje *tri tipa* realnih promjenljivih, nazvani "`float`", "`double`" i "`long double`", koji se međusobno razlikuju po dozvoljenom opsegu i preciznosti (broju tačnih cifara koje se mogu zapamtiti). U jeziku C, sve standardne matematičke funkcije iz biblioteke sa zaglavljem "`math.h`" poput "`sqrt`", "`sin`", "`log`", itd. očekuju isključivo argumente tipa "`double`" i vraćaju rezultat tipa "`double`". To znači da ukoliko im se ponudi argument tipa "`float`", vrši se njegova konverzija u tip "`double`", a ukoliko je argument tipa "`long double`", vrši se njegova degradacija u tip "`double`". Ovo nije posve dobra ideja. Naime, na taj način se ne može iskoristiti činjenica da se matematičke operacije s tipom "`float`" mogu obavljati brže (s obzirom da su promjenljive tipa "`float`" manje tačnosti i zauzimaju duplo manji prostor u memoriji), niti iskoristiti povećana tačnost koju pružaju promjenljive tipa "`long double`" (jer će doći do njihove degradacije u tip "`double`" čim se proslijede kao argument nekoj funkciji). Od standarda C99 jezika C nadalje ovaj problem je riješen tako je većina matematičkih funkcija dobila svoje dvojnike sa sufiksima "`f`" odnosno "`l`" (poput "`sqrtf`", "`sinl`" itd.) koje primaju argumente i vraćaju rezultate tipa "`float`" odnosno "`long double`". Međutim, u jeziku C++ ovaj problem je riješen na drugačiji način. Naime, uskoro ćemo vidjeti da C++ dozvoljava da je moguće imati više funkcija istog imena, ali koje se razlikuju po tome kakve argumente primaju. To je upravo iskorišteno u biblioteci "`cmath`", tako što svaka matematička funkcija iz biblioteke "`cmath`" dobila *tri istoimene verzije*, koje daju rezultat tipa "`float`", "`double`" ili "`long double`", u zavisnosti da li je njihov argument tipa "`float`", "`double`" ili "`long double`" respektivno. Tako će "`sin(a)`" dati rezultat tipa "`float`" ukoliko je promjenljiva "`a`" tipa "`float`", rezultat tipa "`double`" ukoliko je promjenljiva "`a`" tipa "`double`", itd. Na taj način preciznost računanja funkcije zavisi od preciznosti njenog argumenta.

Može se postaviti pitanje *koja će verzija matematičkih funkcija biti pozvana ukoliko im se proslijedi argument cjelobronog tipa*, npr. ukoliko se izvrši izraz poput "`sqrt(3)`" ili "`sqrt(a)`" gdje je "`a`" neka *cjelobrojna promjenljiva*. Drugim riječima, pitanje je da li će cjelobrojna vrijednost "`3`" odnosno "`a`" biti pretvorena u realnu vrijednost tipa "**`float`**", "**`double`**" ili "**`long double`**" (ovo je bitno, s obzirom da od toga zavisi kakve će tačnosti biti rezultat). U jeziku C++ uvedena je konvencija da se u tom slučaju koristi *ista konvencija kakvu koristi i jezik C*, tj. cjelobrojne vrijednosti biće pretvorene u tip "**`double`**" (odnosno, pozvaće se verzija funkcije "`sqrt`" koja očekuje argument tipa "**`double`**" i daje rezultat tipa "**`double`**"). Ukoliko nam to ne odgovara, uvijek možemo izvršiti *eksplicitnu pretvorbu* cjelobrojnog argumenta u željeni realni tip i na taj način utjecati na tačnost računanja, kao recimo u sljedećim konstrukcijama (druga konstrukcija daje isti efekat kao i da nismo zadali eksplicitnu pretvorbu, jedino što se iz nje jasnije vidi kakvu tačnost očekujemo):

```
std::cout << std::sqrt(float(a));
std::cout << std::sqrt(double(a));                    // Podrazumijevana varijanta...
std::cout << std::sqrt(static_cast<long double>(a));
```

U posljednjem slučaju, ključnu riječ "**`static_cast`**" smo morali upotrijebiti, jer se ime tipa "**`long double`**" sastoji od *dvije riječi* (kao alternativu, mogli smo koristiti i notaciju za konverziju u C stilu).

Bitno je naglasiti da se za klasično napisane realne brojeve (tj. realne brojne konstante) smatra po konvenciji da su tipa "**`double`**". Tako se pri pozivu funkcije "`sqrt`" u primjeru

```
std::cout << std::sqrt(3.42);
```

poziva verzija funkcije "`sqrt`" koja očekuje argument tipa "**`double`**" i vraća rezultat tipa "**`double`**". Ova konvencija se može izmijeniti dodavanjem sufiksa "`F`" (ili "`f`") odnosno "`L`" (ili "`l`") na broj, tako da je broj "`3.42F`" (ili "`3.42f`") tipa "**`float`**", dok je broj "`3.42L`" (ili "`3.42l`") tipa "**`long double`**". Isto tako, treba praviti razliku između brojčane konstante poput "`3`" i poput "`3.`". Konstanta "`3`" je cjelobrojna konstanta tipa "**`int`**", dok je "`3.`" realna konstanta tipa "**`double`**" (a "`3.F`" ili "`3.f`" realna konstanta tipa "**`float`**"). Za razliku od jezika C, jezik C++ posjeduje daleko više operacija koje su ovisne od tačnog tipa operanada (tj. koje daju različite rezultate ovisno od toga kakvog su tipa operandi), tako da je u jeziku C++ potrebno dobro voditi računa o tipovima podataka, odnosno naučiti praviti razliku između podataka koji imaju istu vrijednost, ali različit tip (kao što su "`3`", "`3.`" i "`3.F`" odnosno "`3.f`").

Jedno od ne baš omiljenih svojstava jezika C i C++ je činjenica da standard ovih jezika ne propisuje koliki je tačno dozvoljeni opseg brojčanih vrijednosti koje mogu stati u određene brojčane tipove. Na primjer, dok se u jezicima Java i C# tačno zna da najveća vrijednost koja može stati u tip "**`int`**" iznosi 2147483647, u jezicima C i C++ to ovisi od konkretne izvedbe kompajlera. Da bi se prevazišle ove razlike, u jeziku C su podržane razne bibliotečki definirane konstante koje predstavljaju maksimalne i minimalne vrijednosti koje se mogu smjestiti u pojedine brojčane tipove. Recimo, konstante nazvane "`CHAR_MAX`", "`INT_MAX`", "`UINT_MAX`" i "`DBL_MAX`" predstavljaju maksimalne vrijednosti koje se mogu smjestiti u tipove "**`char`**", "**`int`**", "**`unsigned int`**" i "**`double`**" respektivno. Ove konstante su definirane u bibilotekama sa zaglavljem "`limits.h`" za slučaj cjelobrojnih tipova, odnosno "`float.h`" za slučaj realnih tipova. Načelno je ove konstante moguće koristiti i u C++ programima, samo što bi se umjesto zaglavlja "`limits.h`" i "`float.h`" trebala koristiti zaglavlja "`climits`" i "`cfloat`" (bez obzira na to, zbog nekih tehničkih razloga ove konstante nisu u imeniku "`std`", tako da bi upotreba prefiksa "`std::`" ispred njih bila pogrešna). Međutim, C++ nudi jednoobrazniji i fleksibilnihi način za saznavanje raznih informacija o pojedinim numeričkim tipovima podataka, koje uključuju i informacije o maksimalnoj i minimalnoj dozvoljenoj vrijednosti. Za tu svrhu potrebno je u program uključiti biblioteku sa zaglavljem "`limits`" (bez "c" na početku). Sâm pristup ovim informacijama vrši se pomoću konstrukcija za koje je karakteristično da sadrže frazu "`numeric_limits<`*tip*`>`" gdje je "*tip*" ime tipa za kojeg nam trebaju informacije. Recimo, konstrukcija "`std::numeric_limits<`*tip*`>::max()`" daje kao rezultat najveću vrijednost koja može stati u tip "*tip*" (naravno, prefiks "`std::`" nije potreban ukoliko smo prethodno imali "**`using`**" deklaraciju kojom smo "uvezli" ime "`numeric_limits`" iz imenika "`std`", ili čak i cijeli imenik "`std`"). Tako će, na primjer, sljedeći programski isječak ispisati najveće vrijednosti koje se mogu smjestiti u tipove "**`unsigned int`**" i "**`double`**" respektivno (na većini današnjih kompajlera ispis će biti "**`4294967295`**" i "**`1.79769e+308`**", pri čemu ovo posljednje znači $1.79769 \cdot 10^{308}$):

```
std::cout << std::numeric_limits<unsigned int>::max() << " "
  << std::numeric_limits<double>::max() << std::endl;
```

Postoji i konstrukcija "`std::numeric_limits<`*tip*`>::min()`", ali se ponaša pomalo neočekivano. Za *cjelobrojne tipove*, ona zaista daje najmanju dozvoljenu vrijednost koja se može smjestiti u navedeni tip

(za tip "`int`" to je najčešće −2147483648), dok za *realne tipove* ona daje *najmanju pozitivnu vrijednost* koju je moguće predstaviti u navedenom tipu (za tip "`double`" to je najčešće $2.22507 \cdot 10^{-308}$). Također postoji i konstrukcija "`std::numeric_limits<tip>::lowest()`" koja *za sve brojčane tipove podataka* daje najmanju dozvoljenu vrijednost koja se može smjestiti u navedeni tip (tako da za cjelobrojne tipove ona radi isto kao i "`std::numeric_limits<tip>::min()`").

Nije na odmet reći da postoji i standardna biblioteka s zaglavljem "`stdint.h`" (za C) odnosno "`cstdint`", koja rješava pomenuti nedostatak vezan za opsege cjelobrojnih tipova. Naime, ova biblioteka definira cjelobrojne tipove čiji je opseg *tačno poznat* (npr. tip "`int32_t`" predstavlja cjelobrojni tip za koji se rezervira tačno 32 bita, što daje tačan opseg od −2147483648 do 2147483647. U detalje nećemo ulaziti, s obzirom da promjenljive tipa "`int`" praktično uvijek imaju dovoljan opseg da potrebe većine primjena u kojima se cjelobrojne promjenljive uopće koriste (nešto više informacija o ovim tipovima moći će se naći u jednom od dodataka koji će kasnije biti priloženi uz predavanja).

Za ispravno razumijevanje nekih složenijih koncepata jezika C++, bitno je na samom početku dobro razumjeti suštinsku razliku između *inicijalizacije* (engl. *initialization*) i *dodjele* (engl. *assignment*). Naime, treba dobro razlikovati konstrukciju poput

```
int a = 3;                                          // Ovo je INICIJALIZACIJA
```

u kojoj se promjenljiva "`a`" *inicijalizira* na vrijednost "`3`", i konstrukcije poput

```
int a;
a = 3;                                              // Ovo je DODJELA
```

u kojoj se promjenljivoj "`a`" *dodjeljuje* vrijednost "`3`". Naime, u prvoj konstrukciji, u memoriji se stvara promjenljiva "`a`" koja se odmah pri stvaranju inicijalizira na vrijednost "`3`". U drugom slučaju, prvo se stvara *neinicijalizirana* promjenljiva "`a`" (slučajnog i nepredvidljivog sadržaja), kojoj se kasnije *dodjeljuje* vrijednost "`3`". Za razliku od inicijalizacije, koja se vrši nad objektom koji se *tek stvara* i koji *prije toga nije postojao*, u procesu dodjele se postavlja vrijednost objekta *koji već postoji* (i koji pri tome od ranije ima neku vrijednost, makar i nedefiniranu), pri čemu novopostavljena vrijednost *zamjenjuje prethodno postojeću vrijednost*.

Početnicima ukazivanje na ovu razliku može djelovati kao nepotrebno sitničarenje. I zaista, u ovako jednostavnom primjeru, kada se radi o jednostavnim tipovima poput "`int`", razlika između inicijalizacije i dodjele je zaista minorna, tako da mnogi na nju neće obraćati pažnju. Ta razlika također nije mnogo bitna kod svih tipova podataka naslijeđenih iz jezika C. Međutim, kod rada sa složenijim objektima, koji ne postoje u C-u, razlike postaju mnogo izražajnije. U slučaju složenijih objekata ćemo vidjeti da dodjela može biti znatno neefikasnija od inicijalizacije (s obzirom da *uklanjanje* prethodnog sadržaja kod složenih objekata može biti prilično zahtjevna operacija). Pored toga, postoje i takvi objekti nad kojima se uopće ne može vršiti dodjela, već samo inicijalizacija, odnosno dodjela se ne može ni primijeniti. Složeniji koncepti jezika C++ zahtijevaju od programera da jasno vodi računa o razlici između ova dva pojma, kao što ćemo vidjeti kasnije kada upoznamo rad s objektima mnogo složenije strukture nego što su cjelobrojne promjenljive, odnosno složenijim korisnički definiranim tipovima podataka.

Da bi se jasnije istakla razlika između inicijalizacije i dodjele (na koju se u jeziku C++ treba dobro navići), u C++-u je uvedena i alternativna sintaksa za inicijalizaciju, u kojoj se ne koristi znak dodjele "`=`", nego se početna vrijednost promjenljive navodi unutar zagrada. Ova sintaksa naziva se *konstruktorska sintaksa*, zbog činjenice da se inicijalizacija složenijih objekata pomoću tzv. konstruktora koje ćemo upoznati kasnije obavlja upravo ovom sintaksom (dok se klasična sintaksa u duhu jezika C naziva i *kopirajuća sintaksa*, zbog razloga koji će postati jasni kasnije). Na primjer, umjesto deklaracije

```
int a = 3, b = 2;                   // Sintaksa u C stilu (kopirajuća sintaksa)
```

u C++-u također možemo koristiti i deklaraciju

```
int a(3), b(2);                     // Konstruktorska sintaksa
```

Ova nova sintaksa uvedena je iz nekoliko razloga. Prvo, ona nas odmah podstiče da se navikavamo na razliku između inicijalizacije i dodjele. Drugo, C++ je uveo i neke složenije tipove podataka koji se ne mogu inicijalizirati jednom vrijednošću, već traže više vrijednosti za inicijalizaciju, tako da je sintaksa koja koristi znak "`=`" neprikladna (npr. objekti koji predstavljaju kompleksne brojeve inicijaliziraju se s dvije vrijednosti, koje predstavljaju realni i imaginarni dio kompleksnog broja). Treće, postoje izvjesni

tipovi objekata u jeziku C++, koje ćemo kasnije upoznati, kod kojih bi upotreba znaka "=" za inicijalizaciju bila vrlo zbunjujuća, te je stoga i zabranjena. Vidjećemo da prilikom dizajniranja vlastitih korisničkih tipova podataka, programer može odlučiti da li će inicijalizacija pomoću znaka dodjele "=" uopće biti dozvoljena ili ne. Konačno, vidjećemo kasnije da kod nekih složenijih tipova podataka mogu postojati izvjesne razlike u interpretaciji konstruktorske i kopirajuće inicijalizacije (kod jednostavnijih tipova podataka, recimo svih onih naslijeđenih iz jezika C, ove razlike se ne javljaju).

Bez obzira na plemenite namjere zbog kojih je konstruktorska sintaksa uvedena, mora se naglasiti da se ona nije "dobro primila" među programerima u slučajevima kada se vrši inicijalizacija jednostavnih objekata kakvi su, recimo, brojčane promjenljive (tipa "int", "double", itd.), tako da je, u tim slučajevima, nećemo ni mi koristiti (da budemo u skladu s globalnim trendovima). Jedan od najčešćih argumenata za to je da takva sintaksa izgleda "ružno" i "neprirodno" kada se radi o brojčanim objektima (iako je to jako subjektivno mišljenje). Također, mnogima smeta i činjenica da takva sintaksa ne postoji u srodnim jezicima kao što su Java i C#, tako da ona zbunjuje programere koji pored jezika C++ koriste i ove jezike. Međutim, postoje i mnogo objektivnije kritike upućene konstruktorskoj sintaksi za inicijalizaciju. Na prvom mjestu, formiranje nekih složenijih objekata u jeziku C++ vrši se koristeći istu sintaksu, ali pri tome podatak koji se navodi u zagradi nema nikakve veze s početnom vrijednošću objekta (npr. vidjećemo da kod tzv. vektora broj koji se navodi u zagradi nije početna vrijednost vektora, nego broj elemenata vektora). Ovo može ponekad biti iznimno zbunjujuće. Dalje, konstruktorska sintaksa za inicijalizaciju prilično liči na sintaksu za deklaraciju funkcija, tako da se u nekim (doduše prilično rijetkim) slučajevima može desiti da kompajler pogrešno zaključi da se umjesto inicijalizacije radi o deklaraciji funkcije, što zna biti izuzetno frustrirajuće. Recimo, u konstrukciji poput

```
double a(int(b));                    // Ovo je (neočekivano) prototip funkcije!
```

namjera programera je vrlo vjerovatno bila da kreira realnu promjenljivu "a" i inicijalizira je cijelim dijelom promjenljive "b" (koja je, recimo, također realnog tipa). Međutim, ova konstrukcija se također može shvatiti i kao *prototip funkcije* koja se zove "a", koja ima cjelobrojni parametar nazvan "b" i koja vraća realni rezultat (malo je čudno što je parametar "b" unutar zagrada, ali interesantno je da sintaksa jezika C i C++ to dopušta). U jeziku C++ postoji kontroverzno pravilo po kojem u slučaju dvojbe, *sve što se može shvatiti kao prototip funkcije biće zaista shvaćeno kao prototip funkcije*, tako da će ova konstrukcija biti shvaćena kao prototip funkcije a ne kao deklaracija promjenljive. Ovo kontroverzno pravilo jedan od C++ "gurua" Scott Meyers nazvao je "*most vexing parse*" (u slobodnom prevodu, najneugodnije moguće raščlanjivanje) i vrlo je poznato među C++ programerima baš pod tim imenom.

Konačno, još jedan argument protiv konstruktorske sintakse za inicijalizaciju je to što ona nije saglasna s načinom kako se u jezicima C i C++ inicijaliziraju neki tipovi podataka kao što su nizovi i strukture, koji se inicijaliziraju uz pomoć *vitičastih zagrada*. Recimo, niz "niz" od 5 cijelih brojeva čije su početne vrijednosti redom 3, 4, 2, 8 i 1, u jezicima C i C++ može se definirati i inicijalizirati ovako:

```
int niz[5] = {3, 4, 2, 8, 1};
```

Međutim, isti niz se *ne može* inicijalizirati koristeći konstruktorsku sintaksu, odnosno ovako:

```
int niz[5](3, 4, 2, 8, 1);           // Ovo NE RADI!
```

Da bi se donekle riješili ovi problemi, naknadno je u jezik C++ uvedena još jedna sintaksa za inicijalizaciju nazvanu *jednoobrazna* (*uniformna*) *inicijalizacija*, kod koje se vrijednosti koje se koriste za inicijalizaciju promjenljive navode u *vitičastim zagradama* i ova vrsta inicijalizacije je predviđena da radi za *sve vrste podataka*. Tako se sada cjelobrojna promjenljiva "a" može inicijalizirati na vrijednost 3 pomoću sljedeće četiri ekvivalentne sintakse:

```
int a = 3;                           // Kopirajući stil
int a(3);                            // Konstruktorski stil
int a{3};                            // Jednoobrazni stil
int a = {3};                         // Uvedeno samo zbog konzistencije
```

Posljednja od ove četiri sintakse je uvedena samo radi konzistencije s načinom kako se u jeziku C inicijaliziraju nizovi (znak jednakosti ispred vitičaste zagrade) i ne preporučuje se. S druge strane, nizovi se u jeziku C++ mogu inicijalizirati na sljedeća dva načina, od kojih je prvi naslijeđen iz jezika C, dok je drugi karakterističan za jezik C++ i sada se upravo on preporučuje:

```
int niz[5] = {3, 4, 2, 8, 1};        // Stari stil
int niz[5]{3, 4, 2, 8, 1};           // Novi stil - preporučeno
```

Danas su česte preporuke da se uvijek koristi jednoobrazna inicijalizacija. Međutim, postoje izvjesne dvojbe po tom pitanju (pokazalo se da u nekim situacijama to i nije najbolja odluka), tako da će se u ovim materijalima za inicijalizaciju brojčanih promjenljivih koristiti klasična odnosno kopirajuća sintaksa (u duhu jezika C), dok će se konstruktorska i jednoobrazna sintaksa koristiti samo u onim slučajevima kada je to *neophodno*, ili je prosto *prirodnije* (takve situacije će biti posebno istaknute). Također treba napomenuti da postoje izvjesni složeniji tipovi podataka kod kojih inicijalizacija pomoću okruglih i vitičastih zagrada, tj. konstruktorska i jednoobrazna inicijalizacija, *nemaju isto značenje*. Na takve slučajeve ćemo također posebno ukazati kada za to dođe vrijeme.

Kod jednoobrazne inicijalizacije, u mnogim slučajevima je dopušteno da se vitičaste zagrade ostave *prazne*. Tada se vrši tzv. *vrijednosna inicijalizacija* (engl. *value initialization*) odnosno inicijalizacija koristeći *podrazumijevanu vrijednost za odgovarajući tip* (uz pretpostavku da je tip varijable koja se deklarira takav da za njega postoji definirana podrazumijevana vrijednost). Recimo, za sve *brojčane tipove* (cjelobrojne ili realne), podrazumijevana vrijednost je *nula*. Ovo ilustrira sljedeći primjer:

```
int a;                      // Promjenljiva "a" je neinicijalizirana
int b{};                    // Vrijednosna inicijalizacija - poput "int b{0};"
```

Napomenimo još da jednoobrazna inicijalizacija *ne dopušta degradaciju*, tj. takvu pretvorbu kod koje bi došlo do *gubljenja informacija* (npr. inicijalizaciju cjelobrojne promjenljive realnim brojem). Tako će, recimo, deklaracija poput "`int a{3.5};`" dovesti do prijave sintaksne greške (bez obzira što su deklaracije "`int a = 3.5;`" odnosno "`int a(3.5);`" posve legalne).

Bitno je napomenuti da se i konstruktorska i jednoobrazna sintaksa mogu koristiti *samo za potrebe inicijalizacije*, odnosno za postavljanje početne vrijednosti promjenljivoj koja se *upravo kreira*. Za *dodjelu* vrijednosti promjenljivoj, odnosno za mijenjanje vrijednosti *već postojeće promjenljive*, mora se *isključivo koristiti znak dodjele* "`=`" (već smo rekli da je jedan od motiva za uvođenje posebne sintakse za inicijalizaciju bio da se programeri naviknu da vode računa o razlici između inicijalizacije i dodjele). Recimo, ukoliko već imamo cjelobrojnu promjenljivu "`a`" i želimo da joj promijenimo vrijednost na recimo 5, to nipošto *ne smijemo* uraditi konstrukcijom poput

```
a(5);                                       // Ovo NIJE ISPRAVNO!
```

nego isključivo pomoću operatora dodjele:

```
a = 5;                                      // Ovo je dobro
```

Recimo sada nešto o tretmanu konstantnih vrijednosti u jeziku C++. U jeziku C, konstantne vrijednosti se obično definiraju pomoću preprocesorske direktive "`#define`", na primjer kao

```
#define BrojStudenata 50
#define PI 3.141592654
```

Ovaj način posjeduje mnoge nedostatke. Najveći nedostatak je što preprocesorski elementi ne podliježu sintaksnoj kontroli i tipizaciji jezika, tako da eventualna greška u njihovoj definiciji često ostaje neotkrivena, ili bude otkrivena tek prilikom njihove upotrebe. Posebno, direktiva "`#define`" radi vrlo primitivno, na principu *proste zamjene teksta*, tj. simbol koji definiramo zamijenjuje se bukvalno tekstom koji slijedi iza definicije. Na primer, ako napišemo nešto poput

```
#define x 2 + 3
std::cout << x * x;
```

nećemo dobiti kao rezultat 25, nego 11, jer će "`x`" bukvalno biti zamijenjeno s "`2 + 3`", tako da će izraz "`x * x`" postati "`2 + 3 * 2 + 3`", a vrijednost ovog izraza je 11. Ovo bismo mogli riješiti tako da u definiciji umjesto "`2 + 3`" napišemo "`(2 + 3)`", ali ovo već upućuje da su definicije pomoću direktive "`#define`" vrlo nepouzdane. Dalje, svi preprocesorski elementi uvijek se tretiraju *globalno*, odnosno preprocesorski element uvijek vrijedi u čitavom programu od mjesta definiranja nadalje, čak i ukoliko je definiran recimo unutar tijela neke funkcije. Što je još gore, definiranje nekog imena putem direktive "`#define`" potpuno onemogućava da se isto ime koristi *lokalno* za neku drugu svrhu (na primjer, uz gornju definiciju, potpuno je onemogućeno da neka funkcija koristi ime "`BrojStudenata`" kao ime svoje lokalne promjenljive). Ovo su dovoljni razlozi da u jeziku C++ ovaj način definiranja konstanti treba izbjegavati. Generalno, preprocesor "nije u duhu" C++-a, tako da njegovu upotrebu u C++-u treba smanjiti na najnužniji minimum, po mogućnosti, samo na direktivu "`#include`", osim ukoliko nemamo vrlo jak razlog za suprotno (još uvijek postoje neke stvari koje se mogu izvesti samo uz pomoć preprocesora).

Umjesto toga, u C++-u konstante deklariramo kao i promjenljive, uz dodatak ključne riječi "**const**" (za istu svrhu moguće je koristiti i ključnu riječ "**constexpr**", pri čemu ćemo uskoro objasniti u čemu je razlika, s obzirom da postoje značajne razlike između ove dvije ključne riječi):

```
const int BrojStudenata = 50;
const double PI = 3.141592654;
```

S obzirom da se kod konstanti uvijek radi o *inicijalizaciji* (s obzirom da se već definiranoj konstanti ne može ništa *dodjeljivati*), legalno je koristiti i sintaksu koja ne koristi znak dodjele, nego okrugle zagrade, pa čak i vitičaste zagrade (u duhu jednoobrazne inicijalizacije):

```
const int BrojStudenata(50);              // ili "BrojStudenata{50}"
const double PI(3.141592654);             // ili "PI{3.141592654}"
```

Nadalje, konstante moraju biti inicijalizirane, tako da nije dozvoljena deklaracija poput

```
const int broj;                           // Greška: NEINICIJALIZIRANA KONSTANTA!
```

Ključna riječ "**const**" postoji i u jeziku C, ali u njemu ima slabiju težinu nego u C++-u. Naime, u C-u ključna riječ "**const**" i dalje deklarira *promjenljive* (koje imaju svoju *adresu* i zauzimaju *prostor u memoriji*), ali čiji se sadržaj nakon inicijalizacije više ne može mijenjati. Ovakve kvazi-konstante obično se zovu nesretnim imenima poput *konstantne promjenljive* (iako je, s lingvističkog aspekta, jasno da je ovaj termin klasični oksimoron) ili, još gore, *neizmjenljive promjenljive* (engl. *unmodifable variables*). Možda je bolje koristiti termin *neprave konstante*. One se ne mogu koristiti u svim kontekstima u kojima se mogu koristiti prave konstante. Na primjer, u starijim dijalektima jezika C nije dozvoljena sljedeća konstrukcija (ovo ograničenje je uklonjeno u dijalektu C99):

```
const int BrojStudenata = 50;
int ocjene[BrojStudenata];                // Neispravno u C-u do verzije C99
```

dok sljedeća konstrukcija jeste:

```
#define BrojStudenata 50
int ocjene[BrojStudenata];
```

S druge strane, u C++-u su obje konstrukcije ispravne, a samo je prva preporučena. Naime, u C++-u je s "**const**" moguće deklarirati *prave konstante* (engl. *true constants*), pod uvjetom da se njihova inicijalizacija izvrši ili *brojem* ili *konstantnim izrazom* (engl. *constant expression*). Tako, je u prethodnom primjeru "`BrojStudenata`" prava konstanta. Sve prave konstante imaju osobinu da se njihova vrijednost može utvrditi *već prilikom kompajliranja programa*, tj. *prije nego što se program uopće pokrene*. Precizna pravila koja govore šta je tačno konstantni izraz dosta su složena, ali ugrubo možemo reći da je to izraz koji se sastoji isključivo od brojeva, osnovnih aritmetičkih operacija i drugih pravih konstanti.

Sljedeći primjer ilustrira *neprave konstante*. U njemu je definirana neprava konstanta "`starost`", koja zbog načina inicijalizacije, očito nije prava konstanta:

```
int godina_rodjenja, tekuca_godina;
std::cout << "Unesite godinu rodjenja: ";
std::cin >> godina_rodjenja;
std::cout << "Unesite tekucu godinu: ";
std::cin >> tekuca_godina;
const int starost = tekuca_godina - godina_rodjenja;
```

U ovom primjeru se od korisnika prvo traže podaci o godini rođenja i tekućoj godini, a zatim se na osnovu ovih podataka računa starost, koja se koristi za inicijalizaciju (neprave) konstante "`starost`". Ovdje je upotrijebljena (neprava) konstanta, a ne promjenljiva, s obzirom da se starost, nakon što je izračunata na osnovu ulaznih podataka, više neće mijenjati do kraja programa (glavna svrha konstanti, bilo pravih, bilo nepravih, je da omoguće kompajleru prijavu greške u slučaju da kasnije u programu nehotično probamo promijeniti vrijednost nekog podatka koji bi, po prirodi stvari, trebao da bude nepromjenljiv). Međutim, jasno je da vrijednost ove "konstante" ne možemo znati prije početka rada programa, odnosno prije nego što korisnik unese podatke na osnovu kojih će biti izračunata njena vrijednost. Stoga ona ne može biti prava konstanta.

S obzirom na prilično kompleksna pravila šta se smatra konstantnim izrazom a šta ne, mnogi programeri često nisu sigurni da li je konstanta koju su deklarirali prava ili ne. To je jedan od razloga zbog kojih je uvedena i ključna riječ "**constexpr**". Za razliku od ključne riječi "**const**", ova ključna riječ

dozvoljava definiranje *samo pravih konstanti*, odnosno pokušaj da pomoću nje definiramo nešto što nije prava konstanta dovodi do *prijave greške* (stoga se prethodni primjer *neće kompajlirati* ukoliko "**const**" zamijenimo s "**constexpr**"). Ipak, uloga ove ključne riječi *nije samo sigurnosne prirode*, nego ona *omogućava da se pomoću nje tretiraju kao prave konstante i mnogi izrazi koji se u skladu s pravilima jezika C++ ne smatraju pravim konstantama*, *iako bi logički gledano mogli biti*. Recimo, u normalnim okolnostima, jezik C++ nikada neće smatrati da je konstantni izraz ikakav izraz koji u sebi sadrži poziv ma kakve funkcije, čak i ukoliko u tom izrazu i u implementaciji funkcije učestvuju samo prave konstante (odnosno, rezultat ma koje funkcije u normalnim okolnostima prema standardu jezika C++ ne smatra se pravom konstantom. čak i u slučajevima kad bi logički gledano mogao biti). Recimo, pretpostavimo da imamo sljedeću vrlo jednostavnu funkciju:

```
int Kvadrat(int x) {
  return x * x;
}
```

Uz ovakvu funkciju, u sljedećem programskom isječku konstanta "a" neće biti prava konstanta, iako bi logički gledano mogla biti (sasvim je jasno da će joj vrijednost biti 25), tako da se definicija konstante "b" iz istog razloga uopće neće moći kompajlirati:

```
const int a = Kvadrat(5);           // Kompajlira se, ali "a" nije prava konstanta
constexpr int b = Kvadrat(5);       // Ovo daje grešku pri kompajliranju!
```

Ovakav tretman može se promijeniti upravo pomoću ključne riječi "**constexpr**". Naime, ukoliko se uz povratni tip neke funkcije doda ova ključna riječ, tada će se rezultat te funkcije smatrati za pravu konstantu kad god je pozvana s argumentima koji su prave konstante i kad god njeno izvršavanje ne pravi neke propratne efekte vidljive izvan nje poput izmjene neke globalne promjenljive. Recimo, ukoliko definiciju funkcije "Kvadrat" napišemo kao

```
constexpr int Kvadrat(int x) {
  return x * x;
}
```

tada se će obje prethodne definicije konstanti "a" i "b" korektno kompajlirati (i obje će biti prave konstante). Ipak, implementacija funkcije mora zadovoljavati izvjesna ograničenja da bi se uz njen povratni tip mogla pisati specifikacija "**constexpr**". Doduše, u novijim verzijama jezika C++ ta ograničenja su u velikoj relaksirana u odnosu kakva su bila u vrijeme kada je ključna riječ "**constexpr**" uvedena. Glavna prednost funkcija označenih "**constexpr**" specifikacijom je što kompajler kada vidi da je negdje u programu naveden poziv takve funkcije s argumentima koji su prave konstante i koji neće izazvati nikakve propratne efekte, kompajler može *već za vrijeme kompajliranja utvrditi šta će biti rezultat te funkcije* (tj. *obaviti njeno računanje za vrijeme kompajliranja*) i u programu *zamijeniti poziv te funkcije njenim rezultatom* (tako da se pri izvršavanju programa neće trošiti nikakvo vrijeme za računanje funkcije), što može znatno ubrzati izvršavanje programa. Štaviše, to će uvijek biti urađeno kadgod se poziv funkcije iskoristi u kontekstu u kojem se zahtijeva prava konstanta, recimo na mjestu gdje inicijaliziramo neku pravu konstantu s "**constexpr**" deklaracijom (jer drugačije i ne može), a da li će to biti urađeno i u nekim drugim kontekstima, ovisi od procjene kompajlera. Ovo je vrlo zanimljivo, ali u više detalja o tome se u ovom kursu ne možemo upuštati.

Na kraju, završimo ovo izlaganje napomenama o formatiranom ispisu podataka. Poznato je da funkcija "printf" naslijeđena iz jezika C posjeduje obilje mogućnosti za formatiranje ispisa. Slične mogućnosti mogu se ostvariti pomoću objekata izlaznog toka, samo na drugi način. Funkcija "width" primijenjena nad objektom "cout" postavlja željenu širinu ispisa podataka (širina se zadaje kao argument), dok funkcija "precision" postavlja željenu preciznost ispisa (tj. maksimalni broj tačnih cifara, što uključuje kako cifre cijelog dijela broja tako i decimale, ne računajući vodeće nule u brojevima poput 0.00025) prilikom ispisa realnih podataka. Pri tome se eventualne suvišne nule na kraju broja iza decimalnog zareza ne ispisuju. Tako, na primjer, ukoliko izvršimo sljedeću sekvencu naredbi:

```
std::cout << "10/7 = ";
std::cout.width(20);
std::cout.precision(10);
std::cout << 10. / 7;
```

ispis će izgledati ovako:

```
10/7 =          1.428571429
```

Ovdje treba obratiti pažnju da "20" *nije broj razmaka* koji se ispisuju ispred podatka, nego *broj mjesta koje će zauzeti podatak*. Kako u našem slučaju podatak zauzima ukupno 11 mjesta (10 cifara i decimalna tačka), on se dopunjuje s 9 dodatnih razmaka ispred, tako da će ukupna širina ispisa biti 20 mjesta. Razmaci se podrazumijevano dodaju s *lijeve strane*, odnosno podatak koji se ispisuje se podrazumijevano *poravnava udesno* unutar predviđenog prostora za ispis (vidjećemo uskoro kako se ova konvencija može izmijeniti). Isto tako, treba imati u vidu da sama funkcija "width" *ne vrši nikakav ispis razmaka*, nego samo *signalizira* da oni trebaju biti ispisani prilikom ispisa narednog podatka. Za slučaj kada je zadana širina ispisa manja od minimalne neophodne širine potrebne da se ispiše rezultat, funkcija "width" se ignorira.

Nažalost, funkcija "width" djeluje samo na prvu sljedeću stavku koja se ispisuje, nakon čega se opet sve vraća na normalu (za razliku od funkcije "precision" čiji efekat vrijedi za *sve naredne ispise realnih podataka*, sve dok eventualnim novim pozivom ove funkcije ne zadamo neku novu preciznost). Pretpostavimo da želimo uljepšati ispis iz nekog programa koji računa i prikazuje podatke o prihodima, oporezovanom prihodu, porezu i čistom prihodu. Ovo uljepšavanje bi moralo rezultirati dosadnim ponavljanjem konstrukcija "cout.width", kao u sljedećem isječku:

```
std::cout << "Prihod:             "
std::cout.width(5);
std::cout << prihod << std::endl << std::endl;
std::cout << "Oporezovani prihod: ";
std::cout.width(5);
std::cout << oporezovani_prihod << std::endl;
std::cout << "Poreska dazbina:   ";
std::cout.width(5);
std::cout << porez << std::endl;
std::cout << "Cisti prihod:       ";
std::cout.width(5);
std::cout << prihod - porez << std::endl;
```

Ovaj isječak, uz pretpostavku da promjenljive "prihod", "oporezovani_prihod" i "porez" imaju redom vrijednosti 11000, 2000 i 666, proizvodi ispis poput sljedećeg:

```
Prihod:             11000

Oporezovani prihod:  2000
Poreska dažbina:      666
Čisti prihod:       10334
```

Ovim smo, bez sumnje, uljepšali ispis time što smo poravnali ispis udesno (primijetimo da smo, radi lakše procjene neophodne širine polja za ispis podataka, sve tekstove proširili razmacima na istu dužinu). Međutim, ovo uljepšavanje smo "skupo platili" dosadnim ponavljanjem funkcije "width" i potrebom da "prekidamo" tok na objekat "cout". Srećom, biblioteka "iomanip" (skraćeno od engl. *input-output manipulators*) posjeduje tzv. *manipulatore* (odnosno *manipulatorske objekte*). Manipulatori su specijalni objekti koji se *nikad ne koriste samostalno*, nego se *šalju* na izlazni tok (pomoću operatora "<<") s ciljem podešavanja osobina izlaznog toka (u suštini, i objekat "endl", koji smo već upoznali, također predstavlja manipulator, iako za njegovu upotrebu ne treba uključivati biblioteku "iomanip", jer je on definiran već u biblioteci "iostream"). Pomoću upotrebe manipulatora "setw" širinu ispisa možemo postavljati "u hodu", bez prekidanja toka, pomoću naredbi poput:

```
std::cout << "Prihod:             " << std::setw(5) << prihod << std::endl << std::endl;
std::cout << "Oporezovani prihod: " << std::setw(5) << oporezovani_prihod << std::endl;
std::cout << "Poreska dazbina:    " << std::setw(5) << porez << std::endl;
std::cout << "Cisti prihod:       " << std::setw(5) << prihod - porez << std::endl;
```

ili čak poput sljedeće *jedne jedine* naredbe (bez ikakvog prekidanja toka):

```
std::cout
  << "Prihod:             " << std::setw(5) << prihod << std::endl << std::endl
  << "Oporezovani prihod: " << std::setw(5) << oporezovani_prihod << std::endl
  << "Poreska dazbina:    " << std::setw(5) << porez << std::endl
  << "Cisti prihod:       " << std::setw(5) << prihod - porez << std::endl;
```

Naravno, za tu svrhu treba pomoću direktive "#include" uključiti zaglavlje biblioteke "iomanip" u program.

Objekti tokova poput "cin" i "cout" podržavaju veliki broj manipulatora, od kojih ćemo mi koristiti samo neke. Interesantno je da su mnogi manipulatori definirani već u biblioteci "iostream", odnosno *ne zahtijevaju* da se uključuje i biblioteka "iomanip". Srećom, pravilo kada treba koristiti biblioteku "iomanip" je vrlo jednostavno. Naime, ona je potrebna jedino u slučaju kada se koriste manipulatori koji *imaju parametre* (kao što je već spomenuti manipulator "setw").

Primijetimo da smo u svim navedenim primjerima unutar navodnika umetali razmake, bez obzira na to što ćemo kasnije dodatno podešavati širinu ispisa brojčanih rezultata upotrebom funkcije "width" ili manipulatora "setw". Ovo smo učinili da bismo lakše mogli proizvesti ispis koji je poravnat uz posljednju cifru rezultata. Naravno da smo isti efekat mogli postići i bez umetanja razmaka uz prethodno pažljivo proračunavanje širina za svaki od brojčanih podataka. Tako smo isti ispis mogli postići pomoću sljedeće naredbe, u kojoj stringovne konstante ne sadrže razmake:

```
std::cout << "Prihod:" << std::setw(18) << prihod << std::endl << std::endl
  << "Oporezovani prihod:" << std::setw(6) << oporezovani_prihod << std::endl
  << "Poreska dažbina:" << std::setw(9) << porez << std::endl
  << "Cisti prihod:" << std::setw(12) << prihod - porez << std::endl;
```

Očigledno je ovakvo rješenje mnogo manje elegantno od prethodnog rješenja u kojem se može smatrati da "tekstualni dio" ispisa i "brojčani dio" ispisa u svakom redu zauzimaju istu širinu, pri čemu je ta širina u tekstualnom dijelu ispisa ostvarena dopunjavanjem stringovnih konstanti do iste fiksne dužine, a u brojčanom dijelu ispisa slanjem manipulatora "setw(5)" na izlazni tok.

Već je rečeno da se prilikom zadavanja željene širine ispisa podaci ispisuju *poravnati udesno* u predviđeni prostor za ispis. Ukoliko zbog nekog razloga želimo poravnavanje *ulijevo*, to možemo postići slanjem manipulatora "left" na izlazni tok. Na podrazumijevano ravnanje udesno vraćamo se slanjem manipulatora "right". Tako će, na primjer, naredba

```
std::cout << std::left << std::setw(10) << 2 + 3 << "<"
  << std::right << std::setw(10) << 2 * 3 << std::endl;
```

dovesti do sljedećeg ispisa (9 razmaka iza broja 5 i 9 razmaka ispred broja 6):

```
5         <         6
```

Uz pomoć ovih manipulatora, ranije navedeni primjer formatiranog ispisa skupine računovodstvenih podataka mogli smo napisati i ovako:

```
std::cout << std::left << std::setw(20) << "Prihod:"
  << std::right << std::setw(5) << prihod << std::endl << std::endl
  << std::left << std::setw(20) << "Oporezovani prihod:"
  << std::right << std::setw(5) << oporezovani_prihod << std::endl
  << std::left << std::setw(20) << "Poreska dazbina:"
  << std::right << std::setw(5) << porez << std::endl
  << std::left << std::setw(20) << "Cisti prihod:"
  << std::right << std::setw(5) << prihod - porez << std::endl;
```

Manipulator "setprecision" omogućava isti efekat kao i primjena funkcije "precision" nad objektom izlaznog toka, ali bez potrebe za prekidanjem toka. Tako bismo umjesto naporne konstrukcije

```
std::cout << "10/7 = ";
std::cout.width(20);
std::cout.precision(10);
std::cout << 10. / 7;
```

mogli prosto pisati

```
std::cout << "10/7 = " << std::setw(20) << std::setprecision(10) << 10. / 7;
```

bez potrebe za prekidanjem toka. Jedina cijena koju za to trebamo "platiti" je uključivanje zaglavlja biblioteke "iomanip" u program (s obzirom da je "setprecision" manipulator koji *zahtijeva parametar*).

Pomoću funkcije "fill" koja se poziva nad objektom izlaznog toka, ili manipulatora "setfill" koji se direktno šalje na izlazni tok, moguće je postići da se prazan prostor u prostoru rezerviranom pomoću funkcije "width" ili manipulatora "setw" popunjava nečim drugim, a ne razmacima. Funkcija "fill" i manipulator "setfill" kao parametar primaju *znak* koji se koristi za popunjavanje. Na primjer, ukoliko izvršimo skupinu naredbi

```
std::cout.width(10);
std::cout.fill('*');
std::cout << 15 << std::endl;
```

ili naredbu (ovo traži upotrebu biblioteke "`iomanip`")

```
std::cout << std::setw(10) << std::setfill('*') << 15 << std::endl;
```

na ekranu ćemo dobiti sljedeći ispis:

```
********15
```

Treba naglasiti da dejstvo funkcije "`fill`" odnosno manipulatora "`setfill`" ostaje aktivno sve dok se ponovnom njihovom upotrebom ne postavi drugi znak za ispunu (koji naravno može biti i razmak, kako je podrazumijevano na početku).

Funkcije koje mijenjaju neko stanje toka poput "`width`", "`precision`", "`fill`" itd. također kao rezultat *vraćaju kao rezultat vrijednost trenutno važećeg odgovarajućeg stanja*. Ovo nam omogućava da prije neke promjene stanja očitamo trenutno stanje i zapamtimo ga u nekoj promjenljivoj, da bismo kasnije mogli obnoviti prvobitno stanje. Također, vrijednost trenutno važećeg stanja možemo dobiti i *bez njegove izmjene* pozivom nad objektom toka *bez parametara* (npr. pozivom "`cout.precision()`").

Na kraju, spomenimo i manipulatore "`fixed`" i "`scientific`". Upotreba manipulatora "`fixed`" forsira da se ispis realnih podataka vrši s *fiksnim brojem decimala*, pri čemu se broj decimala zadaje funkcijom "`precision`" ili manipulatorom "`setprecision`" (njihov parametar tada nije broj tačnih cifara, nego broj decimalnih mjesta). Manipulator "`scientific`" forsira ispis realnih podataka u tzv. *naučnoj* odnosno *eksponencijalnoj* notaciji (notaciji oblika $x\texttt{e}y$ sa značenjem $x \cdot 10^y$), pri čemu se broj decimala također podešava pomoću "`precision`" ili "`setprecision`". Na primjer, naredba

```
std::cout << std::fixed << std::setprecision(5) << 1. / 7 << " "
  << 1. / 5 << " " << 132 * 211 << " " << 132. * 211 << std::endl
  << std::scientific << std::setprecision(3) << 1. / 7 << " "
  << 1. / 5 << " " << 132 * 211 << " " << 132. * 211 << std::endl;
```

dovešće do sljedećeg ispisa:

```
0.14286 0.20000 27852 27852.00000
1.429e-001 2.000e-001 27852 2.785e+004
```

Treba obratiti pažnju da je rezultat izraza "`132 * 211`" (za razliku od izraza "`132. * 211`") *cjelobrojnog tipa* te se na njega manipulatori "`setprecision`", "`fixed`" i "`scientific`" *ne odnose*.

Napomenimo da manipulatori "`fixed`" i "`scientific`" imaju dejstvo na sve naredne ispise realnih podataka sve dok se njihovo dejstvo ne prekine pomoću jedne prilično rogobatne naredbe u čiji tačan smisao ne možemo ovdje ulaziti te je treba prihvatiti "zdravo za gotovo" (nakon nje se tretman ispisa realnih podataka vraća na podrazumijevani tretman):

```
std::cout.unsetf(std::ios::floatfield);
```