Predavanje 2_a

Jezik C++ poznaje znatno više tipova podataka u odnosu na C. Neki od novododanih tipova podataka ugrađeni su u samo jezgro jezika C++ kao nove ključne riječi, poput logičkog tipa "`bool`", dok su drugi definirani kao izvedeni tipovi podataka u standardnoj biblioteci jezika C++, poput tipova "`complex`", "`vector`", "`string`" itd. čija upotreba zahtijeva uključivanje zaglavlja odgovarajuće biblioteke u program i upotrebu prefiksa "`std::`" (izuzev ukoliko pomoću naredbe "`using`" nisu "uvezeni" odgovarajuća imena tipova iz imenika "`std`", ili eventualno čitav imenik "`std`").

Recimo nekoliko stvari o tipu "`bool`". U jeziku C vrijedi konvencija da je vrijednost svakog tačnog uvjeta jednaka jedinici, dok je vrijednost svakog netačnog uvjeta jednaka nuli (drugim riječima, vrijednosti "tačno" i "1" odnosno vrijednosti "netačno" i "0" su poistovjećene). S druge strane, jezik C++ posjeduje dvije nove ključne riječi "`true`" i "`false`" koje respektivno predstavljaju vrijednosti "tačno" odnosno "netačno". Tako je vrijednost svakog tačnog izraza "`true`", a vrijednost svakog netačnog izraza "`false`". Uvedena je i ključna riječ "`bool`" kojom se mogu deklarirati promjenljive koje mogu imati samo vrijednosti "`true`" odnosno "`false`". Na primjer, ako imamo sljedeće deklaracije:

```
bool u_dugovima, punoljetan, polozio_ispit;
```

tada su sasvim ispravne sljedeće dodjele (uz pretpostavku da također imamo deklarirane brojčane promjenljive "`stanje_kase`" i "`starost`"):

```
u_dugovima = stanje_kase < 0;
punoljetan = starost >= 18;
polozio_ispit = true;
```

Za logičke izraze kažemo da su logičkog tipa, odnosno tipa "`bool`". Međutim, kako u jeziku C vrijedi konvencija da su logički izrazi numeričkog tipa, preciznije cjelobrojnog tipa "`int`" (s obzirom da im je pripisivana cjelobrojna vrijednost 1 ili 0), radi kompatibilnosti uvedena je automatska pretvorba logičkih vrijednosti u numeričke i obratno, koja se vrši po potrebi i koja omogućava miješanje aritmetičkih i logičkih operatora u istom izrazu, na isti način kao u jeziku C. Pri tome vrijedi pravilo da se, u slučaju potrebe za pretvorbom logičkog tipa u numerički, vrijednost "tačno" (tj. "`true`") konvertira u cjelobrojnu vrijednost "1", dok se vrijednost "netačno" (tj. "`false`") konvertira u cjelobrojnu vrijednost "0". U slučaju potrebe za obrnutom konverzijom, nula se konvertira u vrijednost "netačno" dok se *svaka numerička vrijednost* (*cjelobrojna ili realna*) *različita od nule* (a ne samo jedinica) konvertira u vrijednost "tačno". Na primjer, razmotrimo sljedeći programski isječak:

```
bool a;
a = 5;                                  // Ovo se interpretira kao "a = true;"
std::cout << a;
```

Ovaj isječak će ispisati na ekran vrijednost "1". Pri dodjeli "`a = 5`" izvršena je automatska konverzija cjelobrojne vrijednosti "5" u logičku vrijednost "tačno". Konverzija je izvršena zbog činjenice da je odredište (promjenljiva "`a`") u koju smještamo vrijednost logičkog tipa. Prilikom ispisa na ekran, logička vrijednost "tačno" konvertira se u cjelobrojnu vrijednost "1", tako da pri ispisu dobijamo jedinicu. Naravno, ovaj primjer je potpuno vještački formiran, ali pomaže da se lakše shvati šta se zapravo dešava.

Iz prethodnog primjera, može se vidjeti da se promjenljive logičkog tipa pretvaraju u cjelobrojne vrijednosti prilikom ispisa na ekran. Slanjem manipulatora "`boolalpha`" na izlazni tok može se postići da se promjenljive logičkog tipa (i ne samo promjenljive, nego i svi logički izrazi čija je vrijednost "tačno" odnosno "netačno") ispisuju na ekran kao riječi "true" odnosno "false". Tako će, na primjer, naredbe

```
bool a = true;
std::cout << std::boolalpha << a << " " << false << " " << (2 < 3) << std::endl;
```

na ekranu proizvesti ispis

```
true false true
```

Ovakav režim "slovnog" (alfanumeričkog) ispisivanja vrijednosti logičkih izraza i promjenljivih ostaje aktivan sve dok ga ne deaktiviramo slanjem manipulatora "`noboolalpha`" na izlazni tok.

Zbog činjenice da je podržana automatska dvosmjerna konverzija između numeričkih tipova i logičkog tipa, ostvarena je praktično stoprocentna kompatibilnost između tretmana logičkih izraza u C-u i C++-u, bez obzira na činjenicu da su oni različitih tipova u ova dva jezika (cjelobrojnog odnosno logičkog tipa respektivno). Na primjer, u C++-u je i dalje moguće kombinirati aritmetičke i logičke operatore u istom izrazu, zahvaljujući automatskim pretvorbama. Recimo, ukoliko se kao neki od operanada operatora sabiranja "+" upotrijebi logička vrijednost (ili logički izraz), ona će automatski biti konvertirana u cjelobrojnu vrijednost, s obzirom da operator sabiranja nije prirodno definiran za operande koji su logičke vrijednosti. Stoga je izraz

```
5 + (2 < 3) * 4
```

potpuno legalan, i ima vrijednost "9", s obzirom da se vrijednost izraza "`2 < 3`" koja iznosi "tačno" konvertira u vrijednost "1" prije nego što se na nju primijeni operacija množenja.

U nekim slučajevima na prvi pogled nije jasno da li se treba izvršiti pretvorba iz logičkog u numerički tip ili obrnuto. Na primjer, neka je "`a`" logička promjenljiva čija je vrijednost "tačno", a "`b`" cjelobrojna promjenljiva čija je vrijednost "5". Postavlja se pitanje da li je uvjet "`a == b`" tačan. Odgovor zavisi od toga kakva će se pretvorba izvršiti. Ako se vrijednost promjenljive "`a`" pretvori u cjelobrojnu vrijednost "1", uvjet neće biti tačan. S druge strane, ako se vrijednost promjenljive "`b`" pretvori u logičku vrijednost "tačno", uvjet će biti tačan. U jeziku C++ uvijek vrijedi pravilo da se u slučaju kada je moguće više različitih pretvorbi, uvijek *uži tip* (po opsegu vrijednosti) pretvara u *širi tip*. Dakle, ovdje će logička vrijednost promjenljive "`a`" biti pretvorena u cjelobrojnu vrijednost, tako da uvjet neće biti tačan.

S obzirom na automatsku konverziju koja se vrši između logičkog tipa i numeričkih tipova, promjenljive "`u_dugovima`", "`punoljetan`" i "`polozio_ispit`" iz prethodnog primjera mogle su se deklarirati i kao obične cjelobrojne promjenljive (tipa "**`int`**"). U jeziku C, tako se i mora raditi, s obzirom da tip "**`bool`**" ne postoji u jeziku C. Međutim, takvu praksu u jeziku C++ treba strogo izbjegavati, jer se na taj način povećava mogućnost zabune i program čini nejasnijim. Stoga, ukoliko je neka promjenljiva zamišljena da čuva samo logičke vrijednosti, nju treba deklarirati upravo kao takvu.

Radi planiranja memorijskih resursa, korisno je znati da promjenljive tipa "**`bool`**" zauzimaju *jedan bajt*, a ne *jedan bit* kako bi neko mogao pomisliti. Ovo je urađeno iz razloga efikasnosti, s obzirom da procesori ne mogu dovoljno efikasno pristupati individualnim bitima u memoriji, tako da bi rad s promjenljivim tipa "**`bool`**" bio isuviše spor kada bi se one pakovale tako da zauzimaju svega jedan bit. Naravno, razlika između utroška jednog bita i jednog bajta za individualne promjenljive nije toliko bitna, ali treba imati na umu da niz od 10000 elemenata čiji su elementi tipa "**`bool`**" zauzima 10000 bajta memorije, a ne 1250 bajta, koliko bi bilo da se za jedan element tipa "**`bool`**" troši samo jedan bit (ipak, vidjećemo i da je moguće 10000 elemenata čiji su elementi tipa "**`bool`**" zaista smjesti u 1250 bajta).

Na ovom mjestu dobro je reći nešto o *enumeracijama*, odnosno *pobrojanim* (engl. *enumerated*) *tipovima*. Pobrojani tipovi postoje i u jeziku C, ali oni u njemu predstavljaju samo manje ili više prerušene cjelobrojne tipove. Njihov tretman u jeziku C++ je kompletno izmijenjen, pri čemu su, nažalost, morale nastati izvjesne nekompatibilnosti s jezikom C. U jeziku C++ pobrojani tipovi predstavljaju prvi korak ka *korisnički definiranim tipovima*. Pobrojani tipovi opisuju *konačan skup vrijednosti koje su imenovane i uređene* (tj. *stavljene u poredak*) *od strane programera*. Definiramo ih pomoću ključne riječi "**`enum`**", iza koje slijedi *ime tipa koji definiramo* i *popis mogućih vrijednosti tog tipa unutar vitičastih zagrada* (kao moguće vrijednosti mogu se koristiti proizvoljni *identifikatori* koji nisu već iskorišteni za neku drugu svrhu). Na primjer, pomoću deklaracija

```
enum Dani {Ponedjeljak, Utorak, Srijeda, Cetvrtak, Petak, Subota, Nedjelja};
enum Rezultat {Poraz, Nerijeseno, Pobjeda};
```

definiramo dva nova *tipa* nazvana "`Dani`" i "`Rezultat`". Promjenljive pobrojanog tipa možemo deklarirati na uobičajeni način, na primjer:

```
Rezultat danasnji_rezultat;
Dani danas, sutra;
```

Za razliku od jezika C++, u jeziku C se ključna riječ "**`enum`**" mora ponavljati prilikom svake upotrebe pobrojanog tipa. Na primjer:

```
enum Rezultat danasnji_rezultat;          // C stil - nepreporučljivo u C++-u
enum Dani danas, sutra;                   // C stil - nepreporučljivo u C++-u
```

Ova sintaksa i dalje radi u C++-u, ali se smatra izrazito nepreporučljivom.

Obratite pažnju na tačka-zarez iza završne vitičaste zagrade u deklaraciji pobrojanog tipa, s obzirom da je u jeziku C++ prilična rijetkost da se tačka-zarez stavlja neposredno iza zatvorene vitičaste zagrade. Razlog za ovu prividnu anomaliju leži u činjenici da sintaksa jezika C++ (ovo vrijedi i za C) dozvoljava da se odmah nakon deklaracije pobrojanog tipa definiraju i konkretne promjenljive tog tipa, navođenjem njihovih imena iza zatvorene vitičaste zagrade (sve do završnog tačka-zareza). Na primjer, prethodne deklaracije tipova "`Dani`" i "`Rezultat`" i promjenljivih "`danasnji_rezultat`", "`danas`" i "`sutra`" mogle su se pisati skupa, na sljedeći način:

```
enum Dani {Ponedjeljak, Utorak, Srijeda, Cetvrtak, Petak, Subota,
  Nedjelja} danas, sutra;
enum Rezultat {Poraz, Nerijeseno, Pobjeda} danasnji_rezultat;
```

Stoga je tačka-zarez iza zatvorene vitičaste zagrade prosto signalizator da ne želimo odmah deklarirati promjenljive odgovarajućeg pobrojanog tipa, nego da ćemo to obaviti naknadno. Treba napomenuti da se deklaracija promjenljivih pobrojanog tipa istovremeno s deklaracijom tipa, mada je dozvoljena, u jeziku C++ smatra *lošim stilom*, jer se ne uklapa u filozofiju jezika C++ po kojoj se ni jedna promjenljiva ne treba definirati prije nego što nam zaista i zatreba .

Promjenljive ovako definiranog tipa "`Dani`" mogu uzimati samo vrijednosti "`Ponedjeljak`", "`Utorak`", "`Srijeda`", "`Cetvrtak`", "`Petak`", "`Subota`" i "`Nedjelja`", i ništa drugo. Ove vrijednosti nazivamo *pobrojane konstante tipa* "`Dani`" (u slučaju potrebe, treba znati da pobrojane konstante spadaju u *prave konstante*). Slično, promjenljive tipa "`Rezultat`" mogu uzimati samo vrijednosti "`Poraz`", "`Nerijeseno`" i "`Pobjeda`" (pobrojane konstante tipa "`Rezultat`"). Slijede primjeri legalnih dodjeljivanja s pobrojanim tipovima:

```
danas = Srijeda;
danasnji_rezultat = Pobjeda;
```

Promjenljive pobrojanog tipa u jeziku C++ posjeduju mnoga svojstva pravih korisnički definiranih tipova, o kojima ćemo govoriti kasnije. Tako je, recimo, moguće definirati *kako će djelovati pojedini operatori* kada se primijene na promjenljive pobrojanog tipa, o čemu ćemo detaljno govoriti kada budemo govorili o preklapanju (preopterećivanju) operatora. Na primjer, moguće je definirati šta će se tačno desiti ukoliko pokušamo sabrati dvije promjenljive ili pobrojane konstante tipa "`Dani`", ili ukoliko pokušamo ispisati promjenljivu ili pobrojanu konstantu tipa "`Dani`". Međutim, ukoliko ne odredimo drugačije, svi izrazi u kojima se upotrijebe promjenljive pobrojanog tipa koji *ne sadrže propratne efekte koji bi mijenjali vrijednosti tih promjenljivih* (poput primjene operatora "`++`" itd.), izvode se tako da se promjenljiva (ili pobrojana konstanta) pobrojanog tipa *automatski konvertira u cjelobrojnu vrijednost* koja odgovara *rednom broju odgovarajuće pobrojane konstante u definiciji pobrojanog tipa* (pri čemu numeracija počinje od nule). Tako se, na primjer, pobrojana konstanta "`Ponedjeljak`" konvertira u cjelobrojnu vrijednost "0" (isto vrijedi za pobrojanu konstantu "`Poraz`"), pobrojana konstanta "`Utorak`" (ili pobrojana konstanta "`Nerijeseno`") u cjelobrojnu vrijednost "1", itd. Stoga će rezultat izraza

```
5 * Srijeda - 4
```

biti cjelobrojna vrijednost "6", s obzirom da se pobrojana konstanta "`Srijeda`" konvertira u cjelobrojnu vrijednost "2" (preciznije, ovo vrijedi samo ukoliko postupkom preklapanja operatora nije dat drugi smisao operatoru "`*`" primijenjenom na slučaj kada je drugi operand tipa "`Dani`"). Ista će biti i vrijednost izraza "`5 * danas - 4`" s obzirom da je promjenljivoj "`danas`" dodijeljena pobrojana konstanta "`Srijeda`". Iz istog razloga će naredba

```
std::cout << danas;
```

ispisati na ekran broj "2", a ne tekst "Srijeda", kako bi neko mogao brzopleto pomisliti (osim ukoliko postupkom preklapanja operatora operatoru "`<<`" nije dat drugačiji smisao). Također, zahvaljujući automatskoj konverziji u cjelobrojne vrijednosti, između pobrojanih konstanti definiran je i *poredak*, na osnovu njihovog redoslijeda u popisu prilikom deklaracije. Na primjer, poređenje "`Srijeda < Petak`" je tačno, kao i poređenje "`Pobjeda > Nerijeseno`".

Moguće je deklarirati i promjenljive *bezimenog pobrojanog tipa* (engl. *anonymous enumerations*). Na primjer, deklaracijom

```
enum {Poraz, Nerijeseno, Pobjeda} danasnji_rezultat;
```

deklariramo promjenljivu "danasnji_rezultat" koja je sigurno pobrojanog tipa i koja sigurno može uzimati samo vrijednosti "Poraz", "Nerijeseno" i "Pobjeda", ali tom pobrojanom tipu nije dodijeljeno nikakvo ime koje bi se kasnije moglo iskoristiti za definiranje novih promjenljivih istog tipa.

Prilikom deklariranja pobrojanih tipova moguće je zadati *u koju će se cjelobrojnu vrijednost* konvertirati odgovarajuća pobrojana konstanta, prostim navođenjem znaka jednakosti i odgovarajuće vrijednosti iza imena pripadne pobrojane konstante. Na primjer, ukoliko izvršimo deklaraciju

```
enum Rezultat {Poraz = 3, Nerijeseno, Pobjeda = 7};
```

tada će se pobrojane konstante "Poraz", "Nerijeseno" i "Pobjeda" konvertirati respektivno u vrijednosti "3", "4" i "7". Pobrojane konstante kojima nije eksplicitno pridružena odgovarajuća vrijednost konvertiraju se u vrijednost koja je za 1 veća od vrijednosti u koju se konvertira prethodna pobrojana konstanta.

Činjenica da se pobrojane konstante, u slučajevima kada nije određeno drugačije, automatski konvertiraju u cjelobrojne vrijednosti, daje utisak da su pobrojane konstante samo prerušene cjelobrojne konstante. Naime, ukoliko postupkom preklapanja operatora nije eksplicitno određeno drugačije, pobrojane konstante "Ponedjeljak", "Utorak", itd. u deklaraciji

```
enum Dani {Ponedjeljak, Utorak, Srijeda, Cetvrtak, Petak, Subota, Nedjelja};
```

ponašaju se u izrazima praktički identično kao cjelobrojne konstante iz sljedeće deklaracije:

```
const int Ponedjeljak = 0, Utorak = 1, Srijeda = 2, Cetvrtak = 3, Petak = 4,
  Subota = 5, Nedjelja = 6;
```

U jeziku C je zaista tako. Pobrojane konstante u jeziku C nisu ništa drugo nego prerušene cjelobrojne konstante, dok su promjenljive pobrojanog tipa samo prerušene cjelobrojne promjenljive. Međutim, u jeziku C++ dolazi do bitne razlike. U prethodna dva primjera, u prvom slučaju konstante "Ponedjeljak", "Utorak", itd. su tipa "Dani", dok su u drugom slučaju tipa "int". To ne bi bila toliko bitna razlika, da u jeziku C++ nije *strogo zabranjena* dodjela cjelobrojnih vrijednosti promjenljivim pobrojanog tipa, *bez eksplicitnog navođenja konverzije tipa*. Drugim riječima, sljedeća naredba nije legalna (uz pretpostavku da je "danas" promjenljiva tipa "Dani"):

```
danas = 5;                          // OVO NE RADI U C++-U (MADA RADI U C-U)!
```

Ukoliko je baš neophodna dodjela poput prethodne, možemo koristiti eksplicitnu pretvorbu tipa:

```
danas = Dani(5);                    // Nakon ovoga, "danas" dobija vrijednost "Subota"
```

Za razliku od jezika C++, u jeziku C je dozvoljena dodjela cjelobrojne vrijednosti promjenljivoj pobrojanog tipa, bez eksplicitnog navođenja konverzije tipa, što predstavlja jednu od malobrojnih nekompatibilnosti jezika C++ sa C-om. Osnovni razlog zbog kojeg su autori jezika C++ zabranili ovakvu dodjelu je *sigurnost*. Naime, u protivnom, bile bi moguće opasne dodjele poput dodjele "danas = 17" nakon koje bi promjenljiva "danas" imala vrijednost koja izlazi izvan skupa dopuštenih vrijednosti koje promjenljive tipa "Dani" mogu imati. Također, bile bi moguće besmislene dodjele poput "danas = Poraz" (ovakve dodjele su u jeziku C nažalost moguće), bez obzira što je konstanta "Poraz" tipa "Rezultat", a promjenljiva "danas" tipa "Dani". Naime, imenovana konstanta "Poraz" konvertirala bi se u cjelobrojnu vrijednost, koja bi mogla biti legalno dodijeljena promjenljivoj "danas". Komitet za standardizaciju jezika C++ odlučio je da ovakve konstrukcije zabrani. S druge strane, posve je legalno (mada ne uvijek i previše smisleno) dodijeliti pobrojanu konstantu ili promjenljivu pobrojanog tipa cjelobrojnoj promjenljivoj (ovdje dolazi do automatske konverzije tipa), s obzirom da ovakva dodjela nije rizična. Spomenimo i to da je u jeziku C moguće bez konverzije dodijeliti promjenljivoj pobrojanog tipa vrijednost druge promjenljive nekog drugog pobrojanog tipa, dok je u jeziku C++ takva dodjela također striktno zabranjena.

Sigurnosni razlozi su također bili motivacija za zabranu primjene operatora poput "++", "--", "+=" itd. nad promjenljivim pobrojanih tipova (što je također dozvoljeno u jeziku C), kao i za zabranu čitanja promjenljivih pobrojanih tipova s tastature pomoću operatora ">>". Naime, ukoliko bi ovakve operacije bile dozvoljene, postavlja se pitanje šta bi trebala da bude vrijednost promjenljive "danas" nakon izvršavanja izraza "danas++" ukoliko je njena prethodna vrijednost bila "Nedjelja", odnosno kakva bi trebala da bude vrijednost promjenljive "danasnji_rezultat" nakon izvršavanja izraza "danasnji_rezultat--" ukoliko je njena prethodna vrijednost bila "Poraz". U jeziku C operatori "++" i

"`--`" prosto povećavaju odnosno smanjuju pridruženu cjelobrojnu vrijednost za 1 (npr. ukoliko je vrijednost promjenljive "`danas`" bila "`Srijeda`", nakon "`danas++`" nova vrijednost postaje "`Cetvrtak`"), bez obzira da li novodobijena cjelobrojna vrijednost zaista odgovara nekoj pobrojanoj konstanti. U jeziku C++ ovakve nesigurne konstrukcije nisu dopuštene.

Razumije se da je u jeziku C++ moglo biti uvedeno da operatori "`++`" odnosno "`--`" uvijek prelaze na *sljedeću* odnosno *prethodnu* cjelobrojnu konstantu, i to na *kružnom principu* (po kojem iza konstante "`Nedjelja`" slijedi ponovo konstanta "`Ponedjeljak`", itd.). Međutim, na taj način bi se ponašanje operatora poput "`++`" *razlikovalo* u jezicima C i C++, što se ne smije dopustiti. Pri projektiranju jezika C++ postavljen je striktan zahtjev da sve ono što istovremeno radi u jezicima C i C++ *mora raditi isto u oba jezika*. U suprotnom, postojali bi veliki problemi pri prenošenju programa iz jezika C u jezik C++. Naime, moglo bi se desiti da program koji radi ispravno u jeziku C počne raditi neispravno nakon prelaska na C++. Mnogo je manja nevolja ukoliko nešto što je radilo u jeziku C *potpuno prestane da radi* u jeziku C++. U tom slučaju, kompajler za C++ će prijaviti grešku u prevođenju, tako da uvijek možemo ručno izvršiti modifikacije koje su neophodne da program proradi (što je svakako bolje u odnosu na program koji se *ispravno prevodi*, *ali ne radi ispravno*). Tako, na primjer, ukoliko želimo da simuliramo ponašanje operatora "`++`" nad promjenljivim cjelobrojnog tipa iz jezika C u jeziku C++, uvijek možemo umjesto "`danas++`" pisati nešto poput

```
danas = Dani(danas + 1);
```

Doduše, istina je da na taj način nismo logički riješili problem na koji smo ukazali, već smo samo postigli da se program može ispravno prevesti. Alternativno, moguće je postupkom preklapanja operatora *dati značenje* operatorima poput "`++`", "`+=`", "`>>`" itd. čak i kada se primijene na promjenljive pobrojanog tipa. Pri tome je moguće ovim operatorima dati značenje *kakvo god želimo* (npr. moguće je simulirati njihovo ponašanje iz jezika C ili definirati neko inteligentnije ponašanje). O ovome ćemo detaljno govoriti kada budemo govorili o preklapanju operatora.

Osnovni razlog za uvođenje pobrojanih tipova leži u činjenici da njihovo uvođenje, mada ne doprinosi povećanju *funkcionalnosti* samog programa (u smislu da pomoću njih nije moguće uraditi ništa što ne bi bilo moguće uraditi bez njih), znatno povećava *razumljivost* samog programa, pružajući mnogo prirodniji model za predstavljanje pojmova iz stvarnog života, kao što su dani u sedmici ili rezultati nogometne utakmice. Ukoliko rezultat utakmice može biti samo poraz, neriješen rezultat ili pobjeda, znatno prirodnije je definirati poseban tip koji može imati samo vrijednosti "`Poraz`", "`Nerijeseno`" i "`Pobjeda`", nego koristiti neko šifrovanje pri kojem ćemo jedan od ova tri moguća ishoda predstavljati nekim cijelim brojem. Također, uvođenje promjenljivih pobrojanog tipa može često ukloniti neke dileme sociološke prirode. Zamislimo, na primjer, da nam je potrebna promjenljiva "`pol_zaposlenog`" koja čuva podatak o polu zaposlenog radnika (radnice). Pošto pol može imati samo dvije moguće vrijednosti ("muško" i "žensko", ako ignoriramo eventualne medicinske fenomene), neko bi mogao ovu promjenljivu deklarirati kao promjenljivu tipa "`bool`", s obzirom da takve promjenljive mogu imati samo dvije moguće vrijednosti ("`true`" i "`false`"). Međutim, ovakva deklaracija otvara ozbiljne sociološke dileme da li vrijednost "`true`" iskoristiti za predstavljanje muškog ili ženskog pola. Da bismo izbjegli ovakve dileme, mnogo je prirodnije izvršiti deklaraciju tipa

```
enum Pol {Musko, Zensko};
```

a nakon toga prosto deklarirati promjenljivu "`pol_zaposlenog`" kao

```
Pol pol_zaposlenog;
```

Prethodno opisani pobrojani tipovi, koliko god mogu biti korisni i doprinositi čitiljivosti programa, posjeduju neke ozbiljne nedostatke. Na prvom mjestu, *nije dozvoljeno da dva različita pobrojana tipa koriste istu pobrojanu konstantu* (s obzirom da bi tada bilo nejasno u koju bi se ona brojčanu vrijednost trebala pretvoriti). Na primjer, sljedeće deklaracije nisu dozvoljene, jer se pobrojana konstanta "`Crvena`" definira u dva različita pobrojana tipa:

```
enum BojeSemafora {Crvena, Zuta, Zelena};
enum BojeKarata {Crna, Crvena};                    // OVO DAJE GREŠKU!
```

Drugi problem je činjenica da automatska konverzija pobrojanih tipova u cjelobrojni tip nije uvijek poželjna, jer može imati neočekivane posljedice. Recimo, uz prethodno definirane tipove "`Dani`" i "`Rezultat`", izraz "`Srijeda == Pobjeda`" je *tačan* (s obzirom da se obje pobrojane konstante "`Srijeda`" i "`Pobjeda`" konvertiraju u istu cjelobrojnu vrijednost "2"), mada je ovo s logičkog aspekta *posve besmisleno*. Da bi se ovi problemi (i još neki) riješili, u jezik C++ uvedena je još jedna vrsta pobrojanih

tipova, nazvanih *enumeracije s ograničenim vidokrugom* (engl. *scoped enumerations*), a koje se umjesto samo s "`enum`" deklariraju pomoću konstrukcije "`enum class`". Na primjer:

```
enum class BojeSemafora {Crvena, Zuta, Zelena};
enum class BojeKarata {Crna, Crvena};                    // Sad je ovo OK...
```

U odnosu na klasične pobrojane tipove, ova nova vrsta pobrojanih tipova posjeduje neke značajne razlike. Na prvom mjestu, pobrojane konstante koje se definiraju ovakvim tipovima ne mogu se koristiti *samostalno*, nego se uvijek kao prefiks mora naznačiti *pobrojani tip kojem one pripadaju* (npr. ne može se pisati samo "`Zelena`" nego "`BojeSemafora::Zelena`"). Na ovaj način riješena je dilema kojem pobrojanom tipu pripada neka pobrojana konstanta. Drugo, ovakve pobrojane vrijednosti se *ne pretvaraju automatski u cjelobrojne vrijednosti* (nego samo eksplicitnom pretvorbom), čime se izbjegavaju razna neugodna iznenađenja. Treće, poređenje ovakvih pobrojanih vrijednosti je moguće samo ukoliko pripadaju istom pobrojanom tipu (što je zapravo posljedica nepostojanja automatske pretvorbe u cjelobrojni tip). Tako, ako bismo recimo imali sljedeće deklaracije:

```
enum class Dani {Ponedjeljak, Utorak, Srijeda, Cetvrtak, Petak, Subota, Nedjelja};
enum class Rezultat {Poraz, Nerijeseno, Pobjeda};
```

tada bi bila sasvim korektna i smislena poređenja poput "`Dani::Petak > Dani::Utorak`" (koje je tačno) i "`Rezultat::Pobjeda < Rezultat::Poraz`" (koje je netačno), dok bi, s druge strane, poređenje poput "`Dani::Srijeda > Rezultat::Pobjeda`" bilo besmisleno i sintaksno neispravno.

Jezik C++ podržava i rad s *kompleksnim brojevima*, što je veoma korisno u inženjerskim primjenama (pogotovo u elektrotehnici). Za deklariranje kompleksnih tipova podataka koristi se identifikator "`complex`". Kako ovi tipovi nisu ugrađeni u samo jezgro C++ nego su definirani u standardnoj biblioteci jezika C++, za njihovo korištenje potrebno je uključiti u program zaglavlje biblioteke koje se također zove "`complex`". Pored toga, njihovo korištenje također zahtijeva upotrebu "`std::`" prefiksa ispred riječi "`complex`", osim ukoliko pomoću naredbe "`using`" nismo "uvezli" ovaj identifikator iz imenika "`std`" (ili čitav imenik "`std`").

U matematici je kompleksni broj formalno definiran kao uređeni par realnih brojeva, ali kako u jeziku C++ imamo nekoliko tipova za opis realnih brojeva ("`float`", "`double`" i "`long double`"), kompleksne brojeve je moguće izvesti iz svakog od ovih tipova. Sintaksa za deklaraciju kompleksnih promjenljivih je

```
std::complex<osnovni_tip>  popis_promjenljivih
```

gdje je "*osnovni_tip*" tip koji određuje kakvi će biti realni i imaginarni dio promjenljive (navođenje prefiksa "`std::`" moguće je izbjeći korištenjem naredbe "`using`"). Na primjer, deklaracijom oblika

```
std::complex<double> z;
```

deklariramo kompleksnu promjenljivu "`z`" čiji su realni i imaginarni dio tipa "`double`". Standard jezika C++ predviđa da "*osnovni_tip*" smije biti samo neki od tipova "`float`", "`double`" i "`long double`", inače je ponašanje *nepredvidljivo* (kompajler neće prijaviti grešku, ali je nespecificirano kako će se tačno ponašati tako deklarirane promjenljive). Neki kompajleri podržavaju da "*osnovni_tip*" može biti i neki drugi tip (npr. neki od cjelobrojnih tipova poput "`int`"), ali takvo ponašanje *nije podržano standardom*.

Poput svih drugih promjenljivih, i kompleksne promjenljive se mogu inicijalizirati prilikom deklaracije. Kompleksne promjenljive se tipično inicijaliziraju parom vrijednosti u okruglim ili vitičastim zagradama, koje predstavljaju realni odnosno imaginarni dio broja. Na primjer, deklaracija

```
std::complex<double> z1(2, 3.5);            // Može i "z1{2, 3.5}"
```

deklarira kompleksnu promjenljivu "`z1`" i inicijalizira je na kompleksnu vrijednost "(2, 3.5)", što bi se u algebarskoj notaciji moglo zapisati i kao "$2 + 3.5\,i$". Pored toga, moguće je inicijalizirati kompleksnu promjenljivu realnim izrazom odgovarajućeg realnog tipa iz kojeg je razmatrani kompleksni tip izveden (npr. promjenljiva tipa "`complex<double>`" može se inicijalizirati izrazom tipa "`double`", npr. realnim brojem), kao i kompleksnim izrazom istog tipa (npr. drugom kompleksnom promjenljivom istog tipa). Tako su, na primjer, uz pretpostavku da je promjenljiva "`z1`" deklarirana pomoću prethodne deklaracije, također legalne i sljedeće deklaracije:

```
std::complex<double> z2(12.7), z3(z1);      // Ili "z2{12.7}, z3{z1}"
```

U ovakvim slučajevima, inicijalizaciju je moguće izvršiti i pomoću znaka "=". Na primjer:

```
std::complex<double> z2 = 12.7, z3 = z1;      // Sintaksa u duhu jezika C...
```

Potrebno je naglasiti da tipovi "complex<float>", "complex<double>" i "complex<long double>" međusobno predstavljaju *bitno različite tipove* i automatske konverzije između njih su *prilično ograničene*. Stoga je izrazito nepreporučljivo u istom izrazu miješati kompleksne promjenljive izvedene iz različitih osnovnih tipova. Mada je takvo miješanje moguće uz poštovanje izvjesnih pravila koja moraju biti zadovoljena (u čiji opis nećemo ulaziti), najbolje je takve "vratolomije" u potpunosti izbjeći. U daljem, isključivo ćemo koristiti kompleksni tip izveden iz realnog tipa "double".

Kompleksne promjenljive mogu se koristiti u aritmetičkim izrazima u kojima se koriste četiri osnovne računske operacije "+", "-", "*" i "/". Recimo, ukoliko su "a", "b" i "c" kompleksne promjenljive, smisleni su izrazi poput "a + b * c" ili "(a + b) / (a - c)". Također, za kompleksne promjenljive definirani su i operatori "+=", "-=", "*=" i "/=", ali ne i operatori "++" i "--". Pored toga, gotovo sve matematičke funkcije, poput "sqrt", "sin", "log" itd. definirane su i za kompleksne vrijednosti argumenata (ovdje nećemo ulaziti u to šta zapravo predstavlja sinus kompleksnog broja, itd.). Kao dodatak ovim funkcijama, postoje neke funkcije koje su definirane isključivo za kompleksne vrijednosti argumenata. Na ovom mjestu vrijedi spomenuti sljedeće funkcije:

real(z)	Realni dio kompleksnog broja *z*; rezultat je *realan* broj (tj. realnog je tipa)
imag(z)	Imaginarni dio kompleksnog broja *z*; rezultat je *realan* broj
abs(z)	Apsolutna vrijednost (modul) kompleksnog broja *z*; rezultat je *realan* broj
arg(z)	Argument kompleksnog broja *z* (u *radijanima*); rezultat je *realan* broj
conj(z)	Konjugovano kompleksna vrijednost kompleksnog broja *z*

Također, pri izvođenju svih aritmetičkih operacija, dozvoljeno je miješati operande kompleksnog tipa i onog tipa iz kojeg je odgovarajući kompleksni tip izveden. Recimo, moguće je miješati operande tipa "complex<double>" i tipa "double". Stoga, ukoliko su "a", "b" i "c" promjenljive deklarirane kao promjenljive tipa "complex<double>", a "x" i "y" tipa "double", sljedeći izrazi predstavljaju primjere nekih legalnih izraza u kojima se javljaju kompleksne vrijednosti:

```
a + b / std::conj(c)
a + b * y
std::real(a + b) / std::imag(a - c)                    // Rezultat ovog izraza je realan!
3.5 * a - 2.3 * b / (x + std::conj(a + b))
a * c.real() - b * c.imag()                            // Pogledati pasus ispod...
std::log(a + std::sqrt(b + x) / 2.5) * std::sin(a + c)
```

Nije još na odmet napomenuti da u slučaju da nam treba realni ili imaginarni dio *neke kompleksne promjenljive* (a ne proizvoljnog kompleksnog izraza), recimo "z", umjesto "real(z)" odnosno "imag(z)" može se pisati i "z.real()" odnosno "z.imag()" (što je iskorišteno u petom redu u prethodnom primjeru). Mada je izbor između varijanti poput "real(z)" i "z.real()" manje-više stvar ukusa, treba napomenuti da se ove druge varijante *nikada ne koriste sa* "std::" *prefiksom* (dakle, piše se uvijek samo "z.real()" a nikad "z.std::real()"). Generalno, prefiks "std::" se nikada ne koristi s funkcijama koje se ne pozivaju *samostalno*, nego isključivo *nad nekim objektom*.

Bez obzira na gore opisanu fleksibilnost, koja omogućava djelimično miješanje tipova u izrazima, zbog izvjesnih tehničkih razloga nije dozvoljeno da se u istom izrazu miješaju operand kompleksnog tipa i operand nekog tipa *različitog od tipa iz kojeg je posmatrani kompleksni tip izveden*. Na primjer, ukoliko je "a" promjenljiva tipa "complex<double>" a "b" promjenljiva tipa "int", izraz poput "a + b" *nije legalan*. Problem se lako rješava *eksplicitnom konverzijom tipa*. Konkretno, u ovom slučaju, možemo izvršiti eksplicitnu konverziju tipa promjenljive "b" u tip "double" ili čak direktno u kompleksni tip "complex<double>". Stoga će bilo koji od sljedeća dva izraza raditi korektno:

```
a + double(b)
a + std::complex<double>(b)
```

Zapravo, nisu legalni čak ni tako banalni izrazi poput "a + 1" ili "1 / a" (s obzirom da je broj "1" također tipa "int"), ali se u ovim slučajevima problem lako rješava pisanjem "a + 1." odnosno "1. / a" (po konvenciji su brojevi koji sadrže decimalnu tačku tipa "double"). Naravno, radili bi i izrazi poput "a + double(1)" ili "double(1) / a". Napomenimo da bismo u slučaju da je "a" tipa "complex<float>" a ne "complex<double>", umjesto "1." morali pisati "1.F" (ili "1.f"), jer je "1." tipa "double", a "1.F" (ili "1.f") tipa "float" (alternativno bismo mogli koristiti konstrukcije poput "float(1)").

Prirodno se postavlja pitanje kako bismo u nekom izrazu mogli upotrijebiti neku kompleksnu vrijednost poput "(2.5, 3.12)" (odnosno "$2.5 + 3.12\,i$" u algebarskoj notaciji), a da pri tome *ne kreiramo* posebnu promjenljivu za čuvanje te vrijednosti. Jedna ideja je da kreiramo neku kompleksnu promjenljivu, recimo "`I`", koja bi čuvala kompleksnu vrijednost "(0, 1)" (što odgovara imaginarnoj jedinici "*i*"), i da onda koristimo izraze poput "`2.5 + 3.12 * I`". Nažalost, nedostatak tog rješenja je što bi se takvi izrazi morali izračunavati, odnosno pri njihovom korištenju moralo bi se zaista izvršiti jedno množenje i jedno sabiranje, što je gubitak vremena. Srećom, jezik C++ nudi fleksibilnije i efikasnije rješenje za to. Naime, konstrukcijom poput

```
std::complex<osnovni_tip>(realni_dio, imaginarni_dio)
```

kreira se kompleksni broj odgovarajućeg tipa (bez njegovog smještanja u neku imenovanu promjenljivu), sa zadanim realnim i imaginarnim dijelom, koji se može dalje upotrebljavati u izrazima kompleksnog tipa (standardi od C++11 nadalje dozvoljavaju i vitičaste zagrade umjesto okruglih). Pri tome, realni i imaginarni dio moraju biti izrazi realnog tipa ili nekog tipa koji se može pretvoriti u realni tip (npr. cjelobrojnog tipa). Na primjer, ukoliko su "`a`", "`b`", "`c`" promjenljive tipa "`complex<double>`" a "`x`" i "`y`" promjenljive tipa "`double`", legalni su i sljedeći izrazi:

```
a + std::complex<double>(3.2, 2.15)
std::log(a + std::sqrt(b) / std::complex<double>(0, 3)) * std::sin(a + c)
std::sin(x) + c - std::complex<double>(x + y, x - y)
(a + b / x) * (c - std::complex<double>(y, 3)) / std::real(a)
```

Isto tako, ukoliko želimo nekoj *već postojećoj* kompleksnoj promjenljivoj *dodijeliti* neku drugu kompleksnu vrijednost, tj. *promijeniti joj vrijednost* (recimo, promijeniti promjenljivoj "`z`" vrijednost na "(2.5, 3.12)", to možemo uraditi pomoću dodjele poput

```
z = std::complex<double>(2.5, 3.12);
```

Opisani način je univerzalan, mada ćemo uskoro vidjeti da je u nekim kontekstima moguće koristiti i *nešto jednostavniju sintaksu*, odnosno određene *sintaksne olakšice*. Također, nije na odmet napomenuti da je kompleksnim promjenljivim moguće dodjeljivati i realne vrijednosti, pa čak i cjelobrojne vrijednosti, pri čemu dolazi do automatske konverzije tipa.

Na ovom mjestu treba odmah uočiti sličnosti i razlike između sljedeće dvije, sintaksno vrlo slične konstrukcije:

```
std::complex<double> z(2.5, 3.12)          // Kreira promjenljivu
std::complex<double>(2.5, 3.12)            // Kreira bezimenu vrijednost
```

Prva od ove dvije konstrukcije kreira *imenovani objekat* (*promjenljivu*) imena "`z`" čija je (početna) vrijednost kompleksan broj "(2.5, 3.12)" i koji se eventualno kasnije može koristiti za razne potrebe. S druge strane, druga konstrukcija kreira *neimenovanu* (*bezimenu*, *anonimnu*) kompleksnu vrijednost koja predstavlja kompleksan broj "(2.5, 3.12)", koja sama za sebe nema nikakvog smisla ukoliko se odmah ne upotrijebi u okviru nekog složenijeg izraza. Ove dvije konstrukcije ilustriraju nešto što je u jeziku C++ manje-više *opće pravilo*: sintaksa kojom se kreiraju razni *bezimeni objekti* (raznih tipova) inicijalizirani na odgovarajući način, a čija je jedina svrha da budu upotrijebljeni unutar nekih složenijih izraza, praktično je *identična* konstruktorskoj ili jednoobraznoj sintaksi kojom se kreiraju *imenovani objekti* (tj. promjenljive) istog tipa i inicijalizirani na isti način, jedino što se *izostavlja* ime odgovarajućeg objekta (promjenljive). Primijetimo također da, u ovom kontekstu, frazu "`std::complex<double>`" možemo posmatrati ne samo kao ime tipa, nego i kao funkciju s dva parametra, koja od dva realna broja pravi kompleksni broj, pri čemu su ti parametri realni i imaginarni dio tog kompleksnog broja. Ovo je u potpunosti u duhu ranije konstatacije da se u jeziku C++ imena tipova često mogu posmatrati i kao funkcije. Konstrukciju poput "`std::complex<double>(2.5, 3.12)`" možemo također posmatrati i kao *konverziju*, koja par realnih brojeva pretvara u jedan kompleksni broj.

Treba obratiti pažnju na jednu čestu početničku grešku. Izraz čiji je oblik "`(x, y)`" gdje su "x" i "y" neki izrazi sintaksno je *ispravan* u jeziku C++, ali *ne predstavlja* kompleksni broj čiji je realni dio "x", a imaginarni dio "y". Stoga je naredba poput

```
z = (2, 3.5);                              // Ovo ne radi u skladu s očekivanjima!
```

sintaksno *posve ispravna*, ali ne radi ono što bi korisnik mogao očekivati (tj. da izvrši dodjelu kompleksne vrijednosti "(2, 3.5)" promjenljivoj "`z`"). Šta će se dogoditi? Izraz "`(2, 3.5)`", zahvaljujući već ranije

opisanoj interpretaciji tzv. *zarez-operatora* (koja je nasljeđena još iz jezika C), ima izvjesnu *realnu* vrijednost (koja, u konkretnom slučaju, iznosi "3.5"), koja će, nakon odgovarajuće pretvorbe, biti dodijeljena promjenljivoj "z". Drugim riječima, vrijednost promjenljive "z" nakon ovakve dodjele biće broj "3.5", tačnije kompleksna vrijednost "(3.5, 0)"! Isto tako, trebamo se čuvati izraza poput

```
a + (3.2, 2.15)                    // Ni ovo ne daje očekivanu vrijednost!
```

koji, mada su sintaksno ispravni, ne rade ono što bi korisnik mogao očekivati, s obzirom na već spomenuti (za neupućene neočekivani) tretman izraza oblika "(x, y)".

Kompleksni brojevi se mogu ispisivati slanjem na izlazni tok pomoću operatora "<<", pri čemu se ispis vrši u vidu uređenog para "(x, y)", a ne u nešto uobičajenijoj algebarskoj notaciji "$x + y\,i$". Također, kompleksne promjenljive se mogu učitavati iz ulaznog toka (npr. s tastature) koristeći isti format zadavanja kompleksnog broja. Pri tome, kada se očekuje učitavanje kompleksne promjenljive iz ulaznog toka, moguće je unijeti realni ili cijeli broj, koji će biti automatski konvertiran u kompleksni broj.

Nije na odmet napomenuti da neki kompajleri *pogrešno dopuštaju* da se realni i imaginarni dio kompleksne promjenljive koji se dobiju pozivom funkcija "real" i "imag" koriste *kao individualne promjenljive*, odnosno da se koriste u kontekstima u kojima bi se smjele koristiti samo promjenljive, kao recimo s lijeve strane znaka jednakosti ili za prihvatanje podataka s ulaznog toka. Recimo, ako je "z" neka *kompleksna promjenljiva*, neki kompajleri će pogrešno prihvatiti neke (ili čak i sve) od konstrukcija poput sljedećih:

```
std::real(z) = 3.5;                // Ovo nije ispravno, mada negdje radi...
z.real() = 3.5;                    // Niti ovo...
std::cin >> std::imag(z);          // Nije ispravno ni ovo...
std::cin >> z.imag();              // Kao ni ovo...
```

Kad bi ovo bilo legalno, prirodno se postavlja pitanje šta bi onda trebala uraditi konstrukcija poput

```
std::real(a + b - c) = 3.5;        // Ovo je očito besmisleno...
```

Razlog zbog kojeg prethodne konstrukcije nisu ispravne identičan je razlogu zbog kojeg ako je recimo "x" neka realna promjenljiva nisu ispravne konstrukcije poput

```
std::sqrt(x) = 2;                  // Jasno je da ovo nije ispravno...
std::cin >> std::sin(x);           // Kao ni ovo...
```

Naime, rezultati funkcija se u općem slučaju ne mogu koristiti poput promjenljivih (iza njih u općem slučaju ne stoji nikakva memorijska lokacija koja bi mogla prihvatiti vrijednost), a "real" i "imag" su klasične funkcije, isto kao što su "sqrt" i "sin". Stoga, ukoliko iz nekog razloga želimo *posebno unijeti* s tastature realni i imaginarni dio neke kompleksne promjenljive (recimo "z"), to *ne smijemo* uraditi na sljedeći način, bez obzira što neki kompajleri prihvataju ovakve konstrukcije:

```
std::cin >> z.real() >> z.imag();  // Ovo se ne smije raditi!!!
```

Legalan način da postignemo tako nešto je da posebno unesemo realni i imaginarni dio u dvije realne promjenljive, koje ćemo kasnije iskoristiti da na osnovu njih konstruiramo traženu kompleksnu vrijednost, odnosno da učinimo nešto poput sljedećeg:

```
double re, im;
std::cin >> re >> im;
z = std::complex<double>(re, im);
```

Alternativno, realni odnosno imaginarni dio već postojeće kompleksne promjenljive moguće je promijeniti pozivom funkcija "real" odnosno "imag" nad tom promjenljivom, zadajući kao njihov argument novu vrijednost realnog odnosno imaginarnog dijela (npr. "z.real(5)"). Stoga bismo umjesto prethodnog isječka za postizanje istog efekta mogli koristi i sljedeći isječak:

```
double re, im;
std::cin >> re >> im;
z.real(re); z.imag(im);
```

Vrijedi napomenuti još i funkciju "polar". Ova funkcija ima formu

```
polar(r, φ)
```

koja daje kao rezultat kompleksan broj čiji je modul "r", a argument "φ" (u *radijanima*). Ovdje su "r" i "φ" neki izrazi realnog tipa. Ova funkcija je veoma korisna za zadavanje kompleksnih brojeva u trigonometrijskom ili Eulerovom obliku. Na primjer,

```
z = std::polar(5.12, PI / 4);
```

Ovaj primjer podrazumijeva da imamo prethodno definiranu realnu konstantu "`PI`". Treba znati da funkcija "`polar`" daje rezultat onog kompleksnog tipa koji odgovara tipu njenih argumenata. Tako, ako se kao argumenti zadaju vrijednosti tipa "**`double`**", funkcija "`polar`" daje rezultat tipa "`complex<double>`". Ova informacija je korisna zbog prethodno navedenih ograničenja vezanih za miješanje tipova prilikom inicijalizacija i upotrebe aritmetičkih operatora.

Rad s kompleksnim tipovima ilustriraćemo kroz analizu sljedećeg programa za rješavanje kvadratne jednačine, uz upotrebu kompleksnih promjenljivih (objašnjenje slijedi odmah nakon prikaza):

```
#include <iostream>
#include <complex>
typedef std::complex<double> Kompleksni;                  // Korisna olakšica...
int main() {
  double a, b, c;
  std::cout << "Unesi koeficijente:\n";
  std::cin >> a >> b >> c;
  double d = b * b - 4 * a * c;
  if(d >= 0) {
    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 {
    Kompleksni x1 = (-b - std::sqrt(Kompleksni(d))) / (2 * a);
    Kompleksni x2 = (-b + std::sqrt(Kompleksni(d))) / (2 * a);
    std::cout << "x1 = " << x1 << "\nx2 = " << x2 << std::endl;
  }
  return 0;
}
```

U ovom programu iskoristili smo jednu interesantnu deklarativnu naredbu koja je naslijeđena još iz jezika C, a to je "**`typedef`**". Ona se može koristiti na razne načine, a ovdje smo je koristili u najčešće korištenom obliku koji ima sintaksu

`typedef` *ime_tipa alternativno_ime*

Ova naredba omogućava da se tipovima koja imaju nezgrapna imena daju jednostavnija *alternativna imena* (tzv. *aliasi*). Tako je u prethodnom programu iskorištena upravo "**`typedef`**" naredba da uvede alternativno ime (alias) "`Kompleksni`" za rogobatno ime tipa "`std::complex<double>`".

Mogući scenario izvršavanja gore napisanog programa je sljedeći (uz takve ulazne podatke za koje rješenja nisu realna, tako da bi se sličan program koji koristi samo realne promjenljive podatke ili "srušio" ili bi dao *ne-brojeve* kao rezultat):

```
Unesi koeficijente:
3 6 15
x1 = (-1,-2)
x2 = (-1,2)
```

Čak i u ovako jednostavnom programu ima mnogo stvari koje treba da se pojasne. Prvo, primijetimo da su promjenljive "`x1`" i "`x2`" deklarirane kao realne u slučaju kada su rješenja realna (diskriminanta nenegativna), a kao kompleksne samo u slučaju kada rješenja nisu realna. Zbog čega smo uopće provjeravali znak diskriminante, odnosno, zbog čega nismo uvijek ove promjenljive posmatrali kao kompleksne? Nezgoda je u tome što se kompleksne promjenljive uvijek ispisuju kao *uređeni parovi*, čak i kad je imaginarni dio jednak nuli. Drugim riječima, da smo koristili isključivo kompleksne promjenljive

(bez ispitivanja znaka diskriminante, odnosno bez razdvajanja slučajeva kada su rješenja realna odnosno kompleksna), mogli bismo imati scenario poput sljedećeg, što nije prijatno:

```
Unesi koeficijente:
2 10 12
x1 = (-3,0)
x2 = (2,0)
```

Alternativni način da riješimo ovaj problem je da uvijek koristimo kompleksne promjenljive "x1" i "x2", ali da u slučaju kada su rješenja realna, ispisujemo samo realni dio ovih promjenljivih (koji možemo dobiti pomoću funkcije "real". Ova ideja prikazana je u sljedećem programu:

```
#include <iostream>
#include <complex>

typedef std::complex<double> Kompleksni;

int main() {
  double a, b, c;
  std::cout << "Unesi koeficijente:\n";
  std::cin >> a >> b >> c;
  double d = b * b - 4 * a * c;
  Kompleksni x1 = (-b - std::sqrt(Kompleksni(d))) / (2 * a);
  Kompleksni x2 = (-b + std::sqrt(Kompleksni(d))) / (2 * a);
  if(d >= 0)
    std::cout << "x1 = " << x1.real() << "\nx2 = " << x2.real() << std::endl;
  else
    std::cout << "x1 = " << x1 << "\nx2 = " << x2 << std::endl;
  return 0;
}
```

Druga stvar na koju treba obratiti pažnju u ovom programu je prilično rogobatna konstrukcija "sqrt(Kompleksni(d))" (ne zaboravimo da je, zahvaljujući uvedenoj "typedef" deklaraciji, ova konstrukcija zapravo ekvivalentna još goroj konstrukciji "sqrt(std::complex<double>(d))"). Zbog čega ne možemo samo pisati "sqrt(d)"? Problem je u tome što ova funkcija, poput gotovo svih matematičkih funkcija, daje rezultat *onog tipa kojeg je tipa njen argument*. Tako, ukoliko je njen argument tipa "double", rezultat bi također trebao da bude tipa "double". Slijedi da funkcija "sqrt" može kao rezultat dati kompleksan broj samo ukoliko je njen argument kompleksnog tipa. U suprotnom se prosto smatra da je vrijednost funkcije "sqrt" za negativne realne argumente *nedefinirana* (bez obzira što se krajnji rezultat smješta u promjenljivu kompleksnog tipa). Stoga nam je neophodna eksplicitna pretvorba tipa "Kompleksni(d)" (odnosno "std::complex<double>(d)") da obezbijedimo da argument funkcije bude kompleksan.

U suštini, na funkciju "sqrt" s *realnim argumentom* (i *realnim rezultatom*) i funkciju "sqrt" s *kompleksnim argumentom* (i *kompleksnim rezultatom*) treba gledati kao na *dvije posve različite funkcije*, koje se slučajno *isto zovu*. Naime, i u matematici je poznato da neka funkcija nije određena samo preslikavanjem koje obavlja, već i svojim *domenom* i *kodomenom* (npr. funkcija $x \to x^2$ definirana za operande x iz skupa $\mathbb{N}$ i funkcija $x \to x^2$ definirana za operande x iz skupa $\mathbb{R}$ *nisu ista funkcija*). Vidjeli smo da čak i za slučaj realnih operanada, postoje *tri verzije* funkcije "sqrt" (za argumente tipa "float", "double" i "long double"). Pojava da može postojati više funkcija istog imena za različite tipove argumenata (koje mogu raditi čak i posve različite stvari ovisno od tipa argumenta) naziva se *preklapanje* (ili *preopterećivanje*) *funkcija* (engl. *function overloading*). O ovoj mogućnosti ćemo detaljnije govoriti kasnije.

Konstrukcija "sqrt(d)" bi svakako dala kompleksne rezultate kada bi sama promjenljiva "d" bila kompleksna. Međutim, ukoliko uradimo tako nešto, nećemo moći izvršiti test poput "d >= 0", s obzirom da se kompleksni brojevi *ne mogu porediti po veličini*. Doduše, da li su rješenja realna u tom slučaju bismo mogli testirati na neki drugi način (recimo testom poput "std::imag(x1) == 0" odnosno "x1.imag() == 0"), ali nije nimalo mudro deklarirati kao kompleksne one promjenljive koje to zaista ne moraju biti. Naime, kompleksne promjenljive troše više memorije, rad s njima je daleko sporiji nego rad s realnim promjenljivim, i što je najvažnije, pojedine operacije sasvim uobičajene za realne brojeve (poput poređenja po veličini) nemaju smisla za kompleksne brojeve.

Interesantnu mogućnost dobijamo ukoliko u prethodnom programu sve promjenljive, uključujući "a", "b" i "c", deklariramo kao kompleksne promjenljive. To će nam omogućiti da možemo rješavati i kvadratne jednačine čiji su koeficijenti kompleksni brojevi (ne zaboravimo da kompleksne brojeve unosimo s tastature kao uređene parove). Probajte sami izvršiti izmjene u prethodnom programu koje će omogućiti i rješavanje takvih jednačina. Drugim riječima, program nakon izmjene treba da prihvati i scenario poput sljedećeg, u kojem je prikazano rješavanje jednačine $(2+3\,i)\,x^2 - 5\,x + 7\,i = 0$ (probajte riješiti ovu jednačinu "pješke", vidjećete da nije baš jednostavno, pogotovo zbog relativno komplikovanog postupka računanja kvadratnog korijena iz kompleksnih brojeva):

```
Unesi koeficijente:
(2,3) -5 (0,7)
x1 = (0.912023,-2.01861)
x2 = (-0.142792,0.864761)
```

Naravno, ukoliko u tako modificiranom programu želimo omogućiti da se realna rješenja ne ispisuju kao uređeni parovi, ne možemo koristiti uvjet poput "`d >= 0`" (s obzirom da je "`d`" kompleksno) nego uvjet "`x1.imag() == 0`" ili neki srodan uvjet.

Rekli smo ranije da jezik C++ nudi i neke olakšice pri radu s kompleksnim promjenljivim. Prvo ćemo razmotriti neke olakšice koje doduše nisu direktno vezane baš za kompleksne promjenljive, nego su nuspojava nekih drugih stvari koje postoje u jeziku C++, ali bez obzira na to mogu biti korisne. Da bismo objasnili te olakšice, moramo uvesti pojam tzv. *liste inicijalizatora* (engl. *initializer list*). Lista inicijalizatora je skupina podataka unutar vitičastih zagrada koji su međusobno razdvojeni zarezima (npr. "`{2.5, 3.17, 4, -2.25}`" ili "`{x, y, z}`"). Pri tome, u listama inicijalizatora zarez *zaista služi kao separator*, dakle za *razdvajanje* stavki. Liste inicijalizatora postoje i u jeziku C, ali u njemu one se mogu koristiti samo za inicijalizaciju nizova i struktura prilikom njihove deklaracije i nizašta drugo. S druge strane, u jeziku C++, mogućnosti listi inicijalizatora su izuzetno proširene i one se sada mogu koristiti i na mnogim drugim mjestima. O ovome ćemo detaljnije govoriti kasnije. Međutim, u ovom trenutku za nas je interesantno da se liste inicijalizatora u pojedinim kontekstima mogu *automatski pretvarati u razne druge tipove podataka* (zavisno od konkretnih okolnosti). Tako se, u izvjesnim kontekstima, liste inicijalizatora koje se sastoje od dva realna podatka mogu pretvarati u kompleksne vrijednosti (pri tome ta dva realna podatka predstavljaju realni i imaginarni dio dobijene kompleksne vrijednosti). Spomenućemo neke najvažnije kontekste u kojima se ta pretvorba dešava. Na prvom mjestu, ona se dešava pri pokušaju dodjele liste inicijalizatora kompleksnoj promjenljivoj. Stoga su, uz pretpostavku da je "`z`" kompleksna, a "`x`" i "`y`" realne promjenljive, legalne sljedeće konstrukcije:

```
z = {3.2, 2.15};
z = {y + 3, x - 2 * y};
```

Nažalost, ova konstrukcija se ne može koristiti u proizvoljnim izrazima koji sadrže kompleksne veličine (npr. ne možemo umjesto "`a + std::complex<double>(2, 3)`" pisati samo "`a + {2, 3}`"), jer u takvim situacijama nije posve jasno kako se lista inicijalizatora treba tretirati. Još dva interesantna konteksta u kojima se može koristiti ova olakšica je kada šaljemo neku kompleksnu vrijednost nekoj funkciji koja prima parametar kompleksnog tipa, kao i kad vraćamo kompleksnu vrijednost kao rezultat funkcije. Neka na primjer imamo funkciju "`Fun`" poput sljedeće, koja prima kao parametar kompleksnu vrijednost i vraća kompleksni rezultat (ova funkcija vraća kao rezultat kompleksni broj čiji je realni dio dva puta veći, a imaginarni tri puta veći od kompleksnog broja koji joj je proslijeđen kao argument):

```
std::complex<double> Fun(std::complex<double> z) {
  return std::complex<double>(2 * z.real(), 3 * z.imag());
}
```

S ovako napisanom funkcijom, umjesto da pišemo "`Fun(std::complex<double>(10, 5))`", može se pisati prosto "`Fun({10, 5})`", u skladu s gore opisanom konvencijom o prenosu parametara. Isto tako, i sama funkcija "`Fun`" se može jednostavnije napisati ovako, u skladu s gore opisanom konvencijom o vraćanju kompleksnih vrijednosti kao rezultata iz funkcija:

```
std::complex<double> Fun(std::complex<double> z) {
  return {2 * z.real(), 3 * z.imag()};
}
```

Postoji i još jedna olakšica, specifično vezana upravo za kompleksne brojeve. Ukoliko u program uključimo imenik nazvan `std::complex_literals`, tj. dodamo li negdje u program naredbu oblika

```
using namespace std::complex_literals;
```

dobijamo pravo da neposredno pišemo *čisto imaginarne brojeve*, koristeći nastavak (sufiks) "`i`", uz pretpostavku da koristimo tip "`complex<double>`" (za tipove "`complex<float>`" i "`complex<long double>`", odgovarajući nastavci su "`if`" odnosno "`il`"). Recimo, sljedeće konstrukcije su legalne, uz pretpostavku da smo prethodno uključili imenik "`std::complex_literals`":

```
std::complex<double> z1 = 3.25i, z2(2i), z3, z4;
z3 = -2.17i; z4 = 1i;                                   // Mora "1i" (samo "i" ne može)!
```

Indirektno, ovo omogućava i *zadavanje kompleksnih brojeva u algebarskom obliku*, tj. pisanje nečega poput "`z3 = 4.235 + 2.17i`", mada treba znati da se ovo zapravo interpretira kao sabiranje realnog broja "`4.235`" i imaginarnog broja "`2.17i`". Stoga je ova konstrukcija ipak manje efikasna od konstrukcije "`z3 = std::complex<double>(4.235, 2.17)`", koja odmah kreira čitav kompleksni broj. Naravno, moguće su i znatno složenije konstrukcije poput "`z3 = z1 + (2.5 - 3.12i) * (6.41 - 5i)`". Ipak, treba obratiti pažnju na jednu "zvrčku". Naime, suprotno očekivanjima, konstrukcija poput

```
z3 = 2 + 3i;                                            // ILEGALNO!!!
```

nije legalna. Problem je u tome što se izraz "`2 + 3i`" tumači kao zbir *cijelog broja* "`2`" (tipa "`int`") i *čisto imaginarnog broja* "`3i`" (tipa "`complex<double>`"), a već smo ranije ukazali da aritmetičke operacije između cjelobrojnih i kompleksnih operanada *nisu dozvoljene*. Naravno, problem se veoma lako rješava: sve što je potrebno je da umjesto "`2 + 3i`" napišemo "`2. + 3i`".