Predavanje 1_a

Ovi materijali namijenjeni su onima koji su prethodno odslušali kurs "Osnove računarstva" na Elektrotehničkom fakultetu u Sarajevu, kao i svima onima koji već poznaju osnove programiranja u programskom jeziku C, a žele da pređu na programski jezik C++ (dakle, izvjesno predznanje iz programskog jezika C je *nužan preduvjet* za praćenje ovih materijala, odnosno oni *nisu pogodni* za one koji se prvi put susreću s programiranjem). S obzirom da je programski jezik C++ isprva bio zamišljen kao nadgradnja jezika C, gotovo svi programi pisani u jeziku C ujedno predstavljaju ispravne C++ programe. Ipak, kompatibilnost između C-a i C++-a nije stoprocentna, tako da postoje neke konstrukcije koje su ispravne u jeziku C, a nisu u jeziku C++. Takvi primjeri će biti posebno istaknuti. S druge strane, bitno je napomenuti da mnoge konstrukcije iz jezika C, mada rade u jeziku C++, treba izrazito izbjegavati u jeziku C++, s obzirom da u jeziku C++ postoje mnogo bolji, efikasniji i sigurniji načini da se ostvari isti efekat. Treba shvatiti da je C++ danas ipak posve novi programski jezik, bitno drugačiji od C-a, mada u principu visoko kompatibilan s njim. Rješenja koja su dobra (a ponekad i jedina moguća) u C-u, u C++-u mogu biti veoma loša. Stoga biti vrlo dobar programer u C-u najčešće ne znači biti dobar C++ programer. Vrijedi čak obrnuto: ispočetka su nabolji C programeri obično katastrofalno loši C++ programeri, sve dok se ne naviknu na filozofiju novog jezika i na novi način razmišljanja. Stoga će uvodna poglavlja uglavnom biti posvećena onim aspektima jezika C koje nije preporučljivo koristiti u C++-u, dok će specifičnosti načina razmišljanja u C++-u doći kasnije.

Verzija (standard) jezika C++ koja će se koristiti u ovim materijalima poznata je kao ISO C++ 2014 ili skraćeno kao C++14, koja je objavljena sredinom 2014. godine, tako da je za uspješno kompajliranje primjera iz ovih materijala potreban kompajler koji podržava barem C++14 standard i eventualno novije standarde (takvi su uglavnom praktično svi današnji kompajleri). Inače, posljednji važeći standard jezika C++ je C++20, koji se pojavio krajem 2020. godine, ali je za njega podrška još uvijek dosta slaba u današnjim kompajlerima, a pored toga, inovacije koje taj standard uvodi uglavnom izlaze daleko izvan okvira ovog kursa. Treba istaći da standard C++14 uglavnom sadrži samo neka minorna poboljšanja standarda C++14, tako da razlika između standarda C++11 i C++14 nije velika. U međuvremenu, između standarda C++14 i C++20 pojavio se i standard C++17, koji uvodi i neke radikalnije inovacije, ali koje većinom izlaze izvan okvira ovog kursa. Na neke od značajnijih, ali ne previše komplikovanih inovacija koje uvode standardi C++17 i C++20, osvrnućemo se na nekoliko mjesta tokom ovog kursa.

U nekim kompajlerima (uglavnom starijim), podrška za C++14 standard nije automatski uključena, tako ju je potrebno *posebno aktivirati* podešavajući opcije kompajlera (isto vrijedi i za C++17 ili C++20 ako ih konkretan kompajler podržava). Recimo, u starijim verzijama CodeBlocks razvojnog okruženja, potrebno je aktivirati odgovarajuću opciju na kartici "Compiler" u meniju "Settings", mada je podrška za C++14 automatski uključena u novijim verzijama ovog razvojnog okruženja.

Bitno je naglasiti da je u standardu C++11 uvedeno toliko mnogo inovacija u odnosu na standard C++03 (iz 2003. godine) koji mu je prethodio, da programi koji koriste mnogo svojstava standarda počev od C++11 nadalje, starijim poznavaocima jezika C++ (koji u njih nisu upućeni) djeluju kao da su pisani u potpuno novom programskom jeziku, a ne u C++-u. Zapravo, standard C++11 je jezik C++ učinio gotovo dvostruko obimnijim u odnosu na standard C++03 (što je zaista mnogo, jer je već i standard C++03 bio izuzetno obiman). U ovom kursu, radi nedostatka prostora, nećemo moći ulaziti u sve inovacije koje su uveli standardi C++11 i C++14 (zapravo, najbitnije inovacije poput podrške višenitnom programiranju, varijadičke funkcije i klase, itd. uopće se neće ni spomenuti), nego će se uglavnom koristiti inovacije koje su lako objašnjive, a koje omogućavaju znatno elegantnije izvođenje nekih programskih konstrukcija nego što je to bilo moguće u ranijim standardima.

Krenućemo od prvog C programa koji se uglavnom viđa u većini udžbenika o jeziku C – tzv. "Hello world"/ "Vozdra raja" programu:

```
#include <stdio.h>

main() {                                    /* NEISPRAVNO OD C++98 NADALJE! */
  printf("Vozdra raja!\n");
  return 0;
}
```

Čak ni ovaj posve jednostavan C program nije ispravan u C++-u počev od standarda C++98 nadalje (inače, u ovim materijalima, svi neispravni kodovi prikazivaće se *crveno*, uz oznaku bombe "  " na

margini, a ispravni *plavo*). Naime, u ovom primjeru, funkcija "`main`" nema povratni tip. U C-u se tada podrazumijeva da je povratni tip "`int`", dok se u C++-u povratni tip *uvijek mora navoditi* (doduše, i u jeziku C su počev od stanarda C99 uveli da se povratni tip mora navoditi, ali standardi jezika C stariji od C99 su još uvijek u širokoj upotrebi, pogotovo kada se pišu programi za podršku raznih ugradbenih sistema). Dakle, u C++-u (i novijim dijalektima jezika C) ispravna bi bila sljedeća varijanta:

```
#include <stdio.h>

int main() {
  printf("Vozdra raja!\n");
  return 0;
}
```

Ovo je, bar za sada, ispravno C++ rješenje. Međutim, postoji velika mogućnost da se u novijim verzijama jezika C++ ni ovo rješenje neće smatrati ispravnim. Naime, standardna biblioteka funkcija prilagođena jeziku C, kojoj između ostalog pripada i zaglavlje "`stdio.h`", ne uklapa se dobro u neke moderne koncepte jezika C++. Stoga će većina kompajlera na gornji program dati upozorenje da je upotreba zaglavlja "`stdio.h`" *pokuđena* odnosno *izrazito nepreporučljiva* (engl. *deprecated*) u jeziku C++ i da vjerovatno u budućnosti neće biti podržana (pokuđene osobine jezika su one osobine za koje je ustanovljeno da se ne uklapaju dobro u filozofiju jezika i za koje nema garancije da će ih budući standardi jezika i dalje podržavati, tako da ih treba prestati koristiti). Umjesto toga, u jeziku C++ napravljene su nove verzije zaglavlja standardne biblioteke za one elemente jezika C++ koji su naslijeđeni iz jezika C, a koje su napravljene tako da se bolje uklapaju u koncepte jezika C++. Stoga, ukoliko je "`pqr.h`" naziv nekog zaglavlja standardne biblioteke jezika C, u C++-u umjesto njega treba koristiti zaglavlje imena "`cpqr`", odnosno bez sufiksa "`.h`" i s prefiksom "`c`" (npr. "`cstdio`" umjesto "`stdio.h`", "`cmath`" umjesto "`math.h`", "`cctype`" umjesto "`ctype.h`", itd.). Stoga, potpuno ispravna verzija gornjeg programa u C++-u glasi ovako:

```
#include <cstdio>

int main() {
  std::printf("Vozdra raja!\n");      /* Možda će raditi i bez prefiksa "std::" */
  return 0;
}
```

Ovdje vidimo još jednu novinu: specifikator "`std::`" ispred imena funkcije "`printf`" (uskoro ćemo vidjeti da postoji mogućnost da će ovaj program raditi i bez ovog specifikatora, ali *možda i neće*, tako da se na to ne smijemo osloniti). Ovo je jedna od novina zbog koje su uopće i uvedena nova zaglavlja – tzv. *imenici* (engl. *namespaces*). Naime, pojavom velikog broja nestandardnih biblioteka različitih proizvođača postalo je nemoguće spriječiti konflikte u imenima koje mogu nastati kada dvije različite biblioteke upotrijebe isto ime za dva različita objekta ili dvije različite funkcije. Na primjer, biblioteka za rad s bazama podataka može imati funkciju nazvanu "`update`" (za ažuriranje podataka u bazi), dok neki autor biblioteke za rad s grafikom može odabrati isto ime "`update`" za ažuriranje grafičkog prikaza na ekranu. Tada mogu nastati problemi ukoliko isti program koristi *obje biblioteke* – poziv funkcije "`update`" postaje nejasan, jer se ne zna na koju se funkciju "`update`" misli. Zbog toga je odlučeno da se imena (identifikatori) svih objekata i funkcija mogu po potrebi razvrstavati u *imenike* (kao što se datoteke mogu razvrstavati po folderima). Dva različita objekta ili funkcije mogu imati isto ime, pod uvjetom da su definirani u različitim imenicima. Identifikatoru "*ime*" koji se nalazi u imeniku "*imenik*" pristupa se pomoću konstrukcije "*imenik*`::`*ime*". Dalje, standard propisuje da se svi identifikatori koji spadaju u standardnu biblioteku jezika C++ moraju nalaziti u imeniku "`std`". Odatle potiče specifikacija "`std::printf`". Sličnu specifikaciju bismo morali dodati ispred svih drugih objekata i funkcija iz standardne biblioteke (npr. "`std::sqrt`" za funkciju "`sqrt`" iz zaglavlja "`cmath`", koja računa kvadratni korijen). Postoji i jedan "bezimeni" imenik, nazvan *globalni imenik*. Svi identifikatori za koje nije specificirano da će pripadati nekom određenom imeniku, pripadaju globalnom imeniku. Identifikatorima u globalnom imeniku može se pristupati *bez ikakvog prefiksa* (osim ukoliko ne dođe do konflikta, o kojem ćemo govoriti uskoro), ili koristeći prefiks "`::`" (tj. navođenjem konstrukcije "`::` *ime*").

Sad ćemo objasniti zbog čega je rečeno da postoji mogućnost da će prethodni program raditi i bez prefiksa "`std::`". Naime, kada su imenici uvedeni, da bi se olakšalo prenošenje biblioteka koje postoje u jeziku C u C++, standard je *dozvolio* (ali nije naredio) da svi identifikatori iz biblioteka koje postoje u jeziku C mogu i dalje ostati u globalnom imeniku, ali također moraju biti prisutni i u imeniku "`std`" (pod uvjetom da se koristi novo ime zaglavlja, s prefiksom "`c`" i bez nastavka "`.h`"). Ovo je zamišljeno kao

olakšica *autorima kompajlera*, a ne programerima. Stoga se ne smijete osloniti na to da će oni zaista biti prisutni u globalnom imeniku (mada u većini kompajlera jesu), tako da se prefiks "`std::`" mora navoditi ukoliko želimo da program bez problema radi na *svim kompajlerima* za C++.

Stalno navođenje imena imenika može biti naporno. Stoga je moguće pomoću ključne riječi "**`using`**" navesti da će se podrazumijevati da se neki identifikator uzima iz navedenog imenika, ukoliko se ime imenika izostavi. Sljedeći primjer pokazuje takav pristup:

```
#include <cstdio>
using std::printf;              /* Podrazumijevaj "std::" za "printf" */
int main() {
  printf("Vozdra raja!\n");
  return 0;
}
```

Ovim smo naveli da se podrazumijeva da identifikator "`printf`" uzimamo iz imenika "`std`", ako se drugačije ne kaže eksplicitnim navođenjem imenika ispred identifikatora. Moguće je također zadati da se neki imenik (u našem primjeru "`std`") podrazumijeva ispred *bilo kojeg identifikatora* koji nije definiran u globalnom imeniku, a ispred kojeg nije eksplicitno naveden neki drugi imenik, na način kao u sljedećem primjeru:

```
#include <cstdio>
using namespace std;            /* Podrazumijevaj "std::" za sve identifikatore */
int main() {
  printf("Vozdra raja!\n");
  return 0;
}
```

Mada je u kratkim programima ovakvo rješenje vjerovatno najjednostavnije, ono se ipak *ne preporučuje*, jer na taj način u program "uvozimo" cijeli imenik, čime narušavamo osnovni razlog zbog kojeg su imenici uopće uvedeni. Imenik "`std`" je zaista *veoma bogat* i "uvoženjem" cijelog ovog imenika postaje sasvim moguće da će se neko ime koje postoji u ovom imeniku poklopiti s imenom neke promjenljive, objekta ili funkcije koju programer definira u svom programu (pogotovo ako se kao identifikatori koriste engleska imena) ili imenom iz neke od nestandardnih biblioteka iz trećih izvora koje je programer koristio u programu, što može uzrokovati konflikte koji se teško otkrivaju. Ovo je ilustrirano u sljedećem primjeru. U prikazanom programu se koristi (standardna) biblioteka nazvana "`algorithm`" koja, između ostalog, definira funkciju nazvanu "`max`" (u imeniku "`std`"). Ta funkcija nakon deklaracije "**`using namespace`** `std`" postaje dostupna bez navođenja prefiksa "`std::`". Međutim, nakon toga se definira globalna promjenljiva nazvana također "`max`" (u globalnom imeniku). Problem nastaje kada se kasnije toj globalnoj promjenljivoj pokuša pristupiti, jer je nejasno da li se identifikator "`max`" odnosi na nju, ili na funkciju "`max`" koja je također direktno dostupna:

```
#include <algorithm>        /* Biblioteka algorithm sadrži funkciju "max" */
#include <cstdio>
using namespace std;        /* Ovim smo "uvezli" i funkciju "max" */
int max = 2;                /* Globalna promjenljiva se isto zove "max" */
int main() {
  printf("%d", max);        /* Nejasno je na šta se ovo "max" odnosi! */
  return 0;
}
```

Ukoliko načinimo ovakav konflikt, jedino rješenje je da eksplicitno naglasimo na šta mislimo, tj. da pišemo "`::max`" ukoliko mislimo na identifikator "`max`" iz globalnog imenika (tj. na globalnu promjenljivu koju smo definirali), odnosno "`std::max`" ukoliko mislimo na identifikator "`max`" iz imenika "`std`" (tj. na funkciju "`max`" definiranu u biblioteci "`algorithm`"). Interesantno je da do opisanog problema ne bi došlo da je "`max`" bila *lokalna promjenljiva*, definirana recimo unutar funkcije "`main`". Naime, prema pravilima jezika C i C++, lokalni identifikatori uvijek imaju prioritet u odnosu na globalne, odnosno ukoliko su na nekom mjestu u programu istovremeno vidljivi i neki lokalni i neki globalni identifikator, podrazumijeva se da se pristupa lokalnom identifikatoru (osim ako dodamo prefiks "`::`", kojim tada označavamo da želimo pristupiti globalnom identifikatoru).

Da bismo izbjegli opisane probleme i eventualne konflikte, u nastavku ovih materijala ćemo *uvijek eksplicitno navoditi* prefiks "`std::`" ispred svih identifikatora koji pripadaju standardnoj biblioteci. Ovo ima dodatnu prednost što će se čitaoci na taj način podsvjesno pamtiti koji identifikatori pripadaju standardnoj biblioteci. One koje nervira stalno navođenje "`std::`" prefiksa mogu, na sopstveni rizik, koristiti naredbu "**`using`**", ali mi je u nastavku nećemo koristiti. Čak i oni koji žele koristiti naredbu "**`using`**", savjetuje im se da eksplicitno navode za koje identifikatore ne žele navoditi prefiks, umjesto da "uvoze" cijeli imenik "`std`" konstrukcijom "**`using namespace`** `std`".

Mada su prethodni primjeri dotjerani tako da budu u potpunosti po standardu jezika C++, oni još uvijek nisu "u duhu" jezika C++, zbog korištenja biblioteke "`cstdio`" i njene funkcije "`printf`", koje su isuviše "niskog nivoa" s aspekta jezika C++, koji je konceptualno jezik višeg nivoa nego C (zapravo, C++ je "hibridni jezik", u kojem se može programirati na "niskom nivou", na "visokom nivou", u proceduralnom, neobjektnom stilu, kao i u objektno-orijentiranom stilu pa čak i u nekim apstraktnim stilovima kao što su generički i funkcionalni stil programiranja, za razliku od jezika kao što je Java, koji forsira isključivo objektno-orijentirani stil, i kad treba, i kad ne treba). Slijedi primjer koji radi isto što i svi prethodni primjeri, samo što je pisan u duhu jezika C++:

```
#include <iostream>

int main() {
  std::cout << "Vozdra raja!\n";
  return 0;
}
```

U ovom primjeru je umjesto funkcije "`printf`" upotrijebljen tzv. *objekat izlaznog toka podataka* (engl. *output stream*) "`cout`", koji je povezan sa standardnim izlaznim uređajem (obično ekranom). Ovaj objekat definiran je u biblioteci "`iostream`" i poput svih objekata iz standardnih biblioteka, također se nalazi u imeniku "`std`" (stoga, ukoliko smo prethodno koristili naredbu "**`using`** `std::cout`" ili čak "**`using namespace`** `std`" koja "uvozi" čitav imenik "`std`", mogli bismo pisati samo "`cout`" umjesto "`std::cout`"; u suprotnom, ako bismo napisali samo "`cout`" umjesto "`std::cout`", kompajler bi nam javio da ne poznaje objekat "`cout`", što je vrlo česta greška kod početnika). Znak "`<<`" predstavlja *operator umetanja* (engl. *insertion*) u izlazni tok, koji pojednostavljeno možemo čitati kao "šalji na". Stoga konstrukcija

```
std::cout << "Vozdra raja!\n";
```

"umeće" niz znakova (string) `"Vozdra raja\n"` (podsjetimo se da je "`\n`" znak za prelaz u novi red) u izlazni tok, odnosno "šalje ga" na standardni izlazni uređaj (ekran).

Prvi operand operatora "`<<`" je obično neki objekat izlaznog toka (postoje i drugi objekti izlaznog toka osim "`cout`", npr. objekti izlaznih tokova vezani s datotekama), dok drugi operand može biti proizvoljni ispisivi izraz. Na primjer, sljedeće konstrukcije su posve legalne (uz pretpostavku da je uključena biblioteka "`cmath`", u kojoj je deklarirana funkcija "`sqrt`" za računanje kvadratnog korijena):

```
std::cout << 32 - (17 + 14 * 7) / 3;
std::cout << 2.56 - 1.8 * std::sqrt(3.12);
```

Primijetimo da objekat izlaznog toka "`cout`" sam "vodi računa" o tipu podataka koji se "umeću" u tok, tako da nije potrebno eksplicitno specificirati tip podatka koji se ispisuje, kao što bismo morali prilikom korištenja funkcije "`printf`", gdje bismo morali pisati nešto poput:

```
std::printf("%d", 32 - (17 + 14 * 7) / 3);
std::printf("%f", 2.56 - 1.8 * std::sqrt(3.12));
```

Važno je napomenuti da konstrukcija poput "`std::cout << 2 + 3`" u jeziku C++ *također predstavlja izraz*, kao što je izraz i sama konstrukcija "`2 + 3`". Šta je *rezultat* ovog izraza? U C++-u rezultat operatora umetanja primjenjen na neki objekat izlaznog toka kao rezultat daje ponovo isti objekat izlaznog toka (odnosno "`cout`" u našem primjeru), što omogućava *nadovezivanje* operatora "`<<`", kao u sljedećem primjeru:

```
std::cout << "2 + 3 = " << 2 + 3 << "\n";
```

Naime, ovaj izraz se interpretira kao

```
((std::cout << "2 + 3 = ") << 2 + 3) << "\n";
```

Drugim riječima, u izlazni tok se prvo umeće niz znakova `"2 + 3 = "`. Kao rezultat tog umetanja dobijamo ponovo objekat "`cout`" kojem se šalje vrijednost izraza "`2 + 3`" (odnosno 5). Rezultat tog umetanja je ponovo objekat "`cout`", kojem se šalje znak za novi red. Krajnji rezultat je ponovo objekat "`cout`", ali tu činjenicu možemo ignorirati, s obzirom da jezik C++, slično jeziku C, dozvoljava da se krajnji rezultat ma kakvog izraza ignorira (recimo, i funkcija "`printf`" daje kao svoj rezultat broj ispisanih znakova, ali se taj rezultat gotovo uvijek ignorira). U duhu jezika C, prethodna naredba napisala bi se ovako:

```
std::printf("2 + 3 = %d\n", 2 + 3);
```

U zaglavlju "`iostream`" je definiran i objekat za prelazak u novi red "`endl`". Logički gledano, on ima sličnu ulogu kao i string `"\n"`, mada u izvedbenom smislu postoje značajne razlike (u efektivnom smislu, razlika je što upotreba objekta "`endl`" uvijek dovodi do pražnjenja spremnika izlaznog toka, što može biti značajno kod tokova vezanih za datoteke). Stoga smo mogli pisati i:

```
std::cout << "2 + 3 = " << 2 + 3 << std::endl;
```

Na ovom mjestu nije loše napomenuti da standard C++14 predviđa da standardna biblioteka jezika C++ obavezno mora sadržavati biblioteke sa sljedećim zaglavljima (standard C++17 uključuje još i biblioteke sa zaglavljima "`any`", "`optional`", "`variant`", "`memory_resource`", "`charconv`", "`execution`" i "`filesystem`", dok C++20 uključuje i biblioteke sa zaglavljima "`concepts`", "`corutine`", "`compare`", "`version`", "`source_location`", "`format`", "`span`", "`ranges`", "`bit`", "`numbers`", "`syncstream`", "`stop_token`", "`semaphore`", "`latch`" i "`barrier`"):

```
algorithm           array               atomic              bitset
cassert             ccomplex            cctype              cerrno
cfenv               cfloat              chrono              cinttypes
ciso646             climits             clocale             cmath
codecvt             complex             condition_variable  csetjmp
csignal             cstdalign           cstdbool            cstdarg
cstddef             cstdint             cstdio              cstdlib
cstring             ctgmath             ctime               cuchar
cwchar              cwctype             deque               exception
forward_list        fstream             functional          future
initializer_list    iomanip             ios                 iosfwd
iostream            istream             iterator            limits
list                locale              map                 memory
mutex               new                 numeric             ostream
queue               random              ratio               regex
scoped_allocator    set                 shared_mutex        sstream
stack               stdexcept           streambuf           string
strstream           system_error        thread              tuple
type_traits         typeindex           typeinfo            unordered_map
unordered_set       utility             valarray            vector
```

Pored navedenih standardnih zaglavlja iz standardne biblioteke, mnogi kompajleri za C++ dolaze s čitavim skupom *nestandardnih biblioteka*, koje ne predstavljaju propisani standard jezika C++. Tako, na primjer, biblioteka za rad s grafikom sigurno ne može biti unutar standarda jezika C++, s obzirom da C++ uopće ne podrazumijeva da računar na kojem se program izvršava mora imati čak i ekran, a kamoli da mora biti u stanju da vrši grafički prikaz. Također, biblioteka sa zaglavljem "`windows.h`" koja služi za pisanje Windows aplikacija ne može biti dio standarda C++ jezika, jer C++ ne predviđa da se programi moraju nužno izvršavati na Windows operativnom sistemu. Zaglavlja nestandardnih biblioteka gotovo uvijek imaju i dalje nastavak "`.h`" ili "`.hpp`" na imenu, da bi se razlikovala od standardnih biblioteka.

Slično objektu "`cout`", biblioteka "`iostream`" definira i *objekat ulaznog toka podataka* (engl. *input stream*) "`cin`", koji je povezan sa standardnim uređajem za unos (tipično tastaturom). Ovaj objekat se obično koristi zajedno s *operatorom izdvajanja* (engl. *extraction*) iz ulaznog toka "`>>`", koji pojednostavljeno možemo čitati kao "šalji u". Njegov smisao je suprotan u odnosu na smisao operatora umetanja "`<<`" koji se koristi uz objekat izlaznog toka "`cout`". Slijedi primjer programa koji zahtijeva unos dva cijela broja s tastature, a koji zatim ispisuje njihov zbir:

```
#include <iostream>

int main() {
  int broj_1, broj_2;
  std::cout << "Unesite prvi broj: ";
  std::cin >> broj_1;
  std::cout << "Unesite drugi broj: ";
  std::cin >> broj_2;
  std::cout << "Zbir brojeva " << broj_1 << " i " << broj_2 << " glasi "
    << broj_1 + broj_2 << std::endl;
  return 0;
}
```

Isti program, napisan u duhu C-a, mogao bi izgledati recimo ovako:

```
#include <cstdio>

int main() {
  int broj_1, broj_2;
  std::printf("Unesite prvi broj: ");
  std::scanf("%d", &broj_1);
  std::printf("Unesite drugi broj: ");
  std::scanf("%d", &broj_2);
  std::printf("Zbir brojeva %d i %d glasi %d\n", broj_1, broj_2, broj_1 + broj_2);
  return 0;
}
```

Primijetimo da nas objekat ulaznog toka također oslobađa potrebe da vodimo računa o tipu podataka koje unosimo i da eksplicitno prenosimo *adresu odredišne promjenljive* pomoću operatora "`&`", što moramo raditi pri upotrebi funkcije "`scanf`". Za razliku od operatora umetanja, desni operand kod operatora izdvajanja ne može biti proizvoljan izraz, već samo izraz koji predstavlja promjenljivu ili neki objekat iza kojeg postoji rezervirani prostor u memoriji, kao što je npr. element niza, dereferencirani pokazivač, itd. (takvi izrazi nazivaju se *l-vrijednosti*, engl. *l-values*).

Promjenljive se u jeziku C++ deklariraju na isti način kao u jeziku C i za njihova imenovanja vrijede ista pravila kao u jeziku C, uz iznimku da C++ posjeduje više ključnih riječi od jezika C, tako da postoji veći broj riječi koje ne smiju biti imena promjenljivih (tako, npr. "`friend`" i "`this`" ne mogu biti imena promjenljivih u C++-u, a mogu u C-u, jer su u C++-u "`friend`" i "`this`" ključne riječi). Standard C++14 propisuje sljedeće ključne riječi (standard C++17 ne dodaje nikakve nove ključne riječi u odnosu na standard C++14, dok standard C++20 dodaje još i ključne riječi "`concept`", "`requires`", "`char8_t`", "`consteval`", "`constinit`", "`co_await`", "`co_return`" i "`co_yield`"):

```
alignas         alignof         and             and_eq
asm             auto            bitand          bitor
bool            break           case            catch
char            char16_t        char32_t        class
compl           const           constexpr       const_cast
continue        decltype        default         delete
do              double          dynamic_cast    else
enum            explicit        export          extern
false           float           for             friend
goto            if              inline          int
long            mutable         namespace       new
noexcept        not             not_eq          nullptr
operator        or              or_eq           private
protected       public          register        reinterpret_cast
return          short           signed          sizeof
static          static_assert   static_cast     struct
switch          template        this            thread_local
throw           true            try             typedef
typeid          typename        union           unsigned
using           virtual         void            volatile
wchar_t         while           xor             xor_eq
```

Radi lakšeg uočavanja, rezervirane riječi se u programima obično prikazuju **podebljano**, što je učinjeno i u dosadašnjim primjerima i što će biti ubuduće primjenjivano u svim primjerima koji slijede.

Slično kao kod operatora umetanja, rezultat operatora izdvajanja primijenjen nad objektom ulaznog toka daje kao rezultat ponovo sam objekat ulaznog toka, što omogućava ulančavanje i ovog operatora. Recimo, konstrukcija poput

```
std::cin >> a >> b >> c;
```

ima isto dejstvo kao da smo napisali

```
std::cin >> a;
std::cin >> b;
std::cin >> c;
```

Ovo je iskorišteno u sljedećem primjeru, u kojem se s tastature učitavaju tri cijela broja, a zatim ispisuju na ekran, svaki u novom redu:

```
#include <iostream>
int main() {
  int a, b, c;
  std::cin >> a >> b >> c;
  std::cout << a << std::endl << b << std::endl << c << std::endl;
  return 0;
}
```

Na ovom mjestu je bitno ukazati na jednu čestu početničku grešku. Mada nekome može izgledati logična konstrukcija poput

```
std::cin >> a, b, c;                                /* OVO NE RADI DOBRO!!! */
```

ona *neće dati očekivani rezultat* (iz ulaznog toka će biti učitana samo vrijednost promjenljive "`a`"). Što je najgore, kompajler *neće prijaviti nikakvu grešku* s obzirom da je ova konstrukcija *sintaksno ispravna* u skladu s općim sintaksnim pravilima jezika C i C++. Ovo je neželjeni propratni efekat načina na koji se u jezicima C i C++ interpretira znak zareza u općem slučaju. S obzirom da su greške zbog *pogrešne upotrebe zareza* u jezicima C i C++ dosta česte (pogotovo u C++-u), nije na odmet malo detaljnije razmotriti o čemu je zapravo riječ. Naime, ponekad se u jezicima C i C++ javlja potreba da se na nekom mjestu gdje sintaksa jezika očekuje *tačno jedan izraz* upotrijebi *više izraza*. Zbog toga je u jezike C i C++ uvedena konvencija da se skupina od *više izraza koji su međusobno razdvojeni zarezima* logički tretira kao *jedan jedini izraz* (tj. može se koristiti u kontekstima u kojima se očekuje tačno jedan izraz), ali se samo rezultat *onog posljednjeg izraza u nizu* posmatra kao rezultat tog složenog izraza (bez obzira što će se izvršiti sve njegove komponente i to redom slijeva nadesno). Ova mogućnost ima raznolike primjene u jezicima C i C++ (ali nažalost i mnoge propratne neželjene efekte). Jedan jednostavan primjer primjene je sljedeći. Neka je potrebno unositi s tastature cijele brojeve i ispisivati na ekran njihove kvadrate, sve dok se ne unese negativan broj, nakon čega se dalji unos prekida. Postoji mnogo načina da se ovo izvede, a ubjedljivo najkraći je sljedeći, u kojem se na vješt način koristi upravo zarez:

```
int x;
while(std::cin >> x, x >= 0) std::cout << x * x << std::endl;
```

Kako ovaj primjer radi? Jasno je da se unutar zagrada kod "`while`" petlje treba navesti neki izraz, koji predstavlja neki uvjet (tj. koji može biti tačan ili netačan). Međutim, u ovom primjeru, unutar zagrada u "`while`" petlji navedena su *dva izraza* od kojih je prvi "`std::cin >> x`" a drugi "`x >= 0`". S druge strane, kako su ova dva izraza međusobno povezana zarezom, oni se tretiraju kao *jedan složeni izraz*. U ovom slučaju, zarez nema ulogu "razdvojnika" (separatora) nego "poveznika" koji dva ili više izraza povezuje u jedan složeni izraz (zarez upotrijebljen na ovaj način naziva se *zarez-operator* ili engl. *comma operator*). Dakle, nailaskom na "`while`" petlju, izvršavaju se oba podizraza složenog izraza koji se nalazi u zagradi. Prvo će se izvršiti podizraz "`std::cin >> x`" koji će izvršiti unos cijelog broja s tastature u promjenljivu "`x`", a nakon toga će se izvršiti podizraz "`x >= 0`" koji testira da li je unesena vrijednost veća ili jednaka od nule. Međutim, kao rezultat tog složenog izraza uzima se samo rezultat posljednjeg izraza (tj. rezultat obavljenog testiranja), tako da će se na osnovu tog testiranja odlučiti da li ćemo uopće ući u tijelo petlje ili ne (tijelo petlje se ovdje sastoji samo od naredbe koja ispisuje kvadrat unesenog broja). Nakon eventualno obavljenog ispisa, vraćamo se ponovo na "uvjet" petlje, u kojem se ponovo vrše dvije akcije: unos novog broja i njegovo testiranje na nenegativnost. Ovakve konstrukcije na prvi pogled nisu posve očigledne, ali se prilično često koriste, kako u C-u, tako i u C++-u.

Pogledajmo sada kako se u kontekstu ovoga što smo upravo objasnili interpretira izraz

```
std::cin >> a, b, c;                                /* OVO NE RADI DOBRO!!! */
```

Osnovni problem kod ovako formiranog izraza je što se on zapravo interpretira kao *objedinjenje* tri izraza "`std::cin >> a`", "`b`" i "`c`" u jednu cjelinu. Mada će se sva tri izraza izvršiti, samo prvi od njih zaista obavlja unos s tastature (drugi i treći izraz, odnosno izrazi "`b`" i "`c`" neće proizvesti nikakvu akciju). Posljedica će biti da će samo vrijednost promjenljive "`a`" biti unesena s tastature, dok će promjenljive "`b`" i "`c`" ostati kakve su bile i ranije. Još jednom treba napomenuti da činjenica da zarez u nekim slučajevima služi kao *separator* (tj. kao "razdvajač" više stavki u nekom popisu, kao recimo prilikom deklaracije promjenljivih), dok u nekim slučajevima služi kao *poveznik*, te da treba *tačno znati* kad je u pitanju jedna, a kad druga uloga, dovodi do čestih zabuna u C i C++ programima (šteta je što za funkciju povezivanja u ovim jezicima nije upotrijebljen neki drugi znak, a ne baš zarez – zabune bi tada bile mnogo rjeđe).

Razmotrimo sada malo detaljnije šta se tačno dešava prilikom upotrebe ulaznog toka. Svi znakovi koje korisnik unese sve do pritiska na taster ENTER čuvaju se u *spremniku* (*baferu*) *ulaznog toka*. Međutim, operator izdvajanja vrši izdvajanje iz ulaznog toka samo *do prvog razmaka* ili *do prvog znaka koji ne odgovara tipu podataka koji se izdvaja* (npr. do prvog znaka koji nije cifra u slučaju kada izdvajamo cjelobrojni podatak). Preostali znakovi *i dalje su pohranjeni* u spremniku ulaznog toka. Sljedeća upotreba operatora izdvajanja će nastaviti izdvajanje iz niza znakova zapamćenog u spremniku. Tek kada se *ulazni tok isprazni*, odnosno kada se *istroše svi znakovi* pohranjeni u spremniku, biće zatražen novi unos s ulaznog uređaja. Sljedeća slika prikazuje nekoliko mogućih scenarija prilikom izvršavanja programa prikazanog u dnu stranice 6. (kurzivom su prikazani podaci koje unosi korisnik nakon što se program pokrene):

```
10 20 30 40(ENTER)
10
20
30
```

```
10 20(ENTER)
30(ENTER)
10
20
30
```

```
10(ENTER)
20(ENTER)
30 40(ENTER)
10
20
30
```

Ovakvo ponašanje je u nekim slučajevima povoljno, ali u nekim nije. Nekada želimo da smo uvijek sigurni da će operator izdvajanja pročitati "svježe unesene" podatke, a ne neke podatke koji su od ranije preostali u spremniku ulaznog toka. Zbog toga je moguće pozivom funkcije "`ignore`" nad objektom ulaznog toka "`cin`" isprazniti ulazni tok (u jeziku C++ postoje funkcije koje se ne pozivaju samostalno, nego uvijek nad nekim objektom, koriteći sintaksu poput "*objekat*`.`*funkcija*`(`*argumenti*`)`", o čemu ćemo kasnije detaljno pričati). Ova funkcija ima dva argumenta, od kojih je prvi cjelobrojnog tipa, a drugi znakovnog tipa (tipa "`char`"). Ona uklanja znakove iz ulaznog toka, pri čemu se uklanjanje obustavlja ili kada se ukloni onoliko znakova koliko je zadano prvim argumentom ili dok se ne ukloni znak zadan drugim argumentom. Na primjer, naredba

```
std::cin.ignore(50, '.');
```

uklanja znakove iz ulaznog toka dok se ne ukloni 50 znakova ili dok se ne ukloni znak "`.`". Ova naredba se najčešće koristi da *kompletno isprazni* ulazni tok. Za tu svrhu, treba zadati naredbu poput

```
std::cin.ignore(10000, '\n');
```

Ova naredba će uklanjati znakove iz ulaznog toka ili dok se ne ukloni 10000 znakova ili dok se ne ukloni oznaka kraja reda "`\n`" (nakon čega je zapravo ulazni tok prazan). Naravno, kao prvi argument smo umjesto 10000 mogli staviti neki drugi veliki broj (naš je cilj zapravo *samo* da uklonimo sve znakove dok ne uklonimo oznaku kraja reda, ali moramo nešto zadati i kao prvi argument). Opisana tehnika je iskorištena u sljedećoj sekvenci naredbi:

```
int broj_1, broj_2;
std::cout << "Unesite prvi broj: ";
std::cin >> broj_1;
std::cin.ignore(10000, '\n');                       /* Brisanje spremnika */
std::cout << "Unesite drugi broj: ";
std::cin >> broj_2;
```

Izvršenjem ovih naredbi ćemo biti sigurni da će se na zahtjev "Unesi drugi broj:" zaista tražiti unos broja s tastature, čak i ukoliko je korisnik prilikom zahtjeva "Unesi prvi broj:" unio odmah dva broja.

U slučaju unosa pogrešnih podataka (npr. ukoliko se očekuje broj, a već prvi uneseni znak nije cifra), ulazni tok će dospjeti u tzv. *neispravno stanje*. Da je tok u neispravnom stanju, možemo provjeriti primjenom operatora negacije "`!`" nad objektom toka. Rezultat je tačan ako i samo ako je tok u neispravnom stanju, što možemo iskoristiti kao uvjet unutar naredbe "**`if`**", "**`while`**" ili "**`for`**". Ovo ilustrira sljedeći fragment:

```
int broj;
std::cout << "Unesite neki broj: ";
std::cin >> broj;
if(!std::cin) std::cout << "Niste unijeli broj!\n";
else std::cout << "Zaista ste unijeli broj!\n";
```

Kao uvjet unutar naredbi "**`if`**", "**`while`**" i "**`for`**", može se iskoristiti i sam objekat toka, koji se tada interpretira kao tačan ako i samo ako je u ispravnom stanju. Tako se prethodni isječak mogao napisati i ovako:

```
int broj;
std::cout << "Unesite neki broj: ";
std::cin >> broj;
if(std::cin) std::cout << "Zaista ste unijeli broj!\n";
else std::cout << "Niste unijeli broj!\n";
```

Kada ulazni tok jednom dospije u neispravno stanje, on u takvom stanju ostaje i nadalje, i svaki sljedeći pokušaj izdvajanja iz ulaznog toka *biće ignoriran*. Tok možemo ponovo vratiti u ispravno stanje pozivom funkcije "`clear`" bez parametara nad objektom ulaznog toka. U narednom primjeru ćemo iskoristiti ovu funkciju i "**`while`**" petlju sa ciljem ponavljanja unosa sve dok unos ne bude ispravan:

```
int broj;
std::cout << "Unesite broj: ";
std::cin >> broj;
while(!std::cin) {
  std::cout << "Nemojte se šaliti, unesite broj!\n";
  std::cin.clear();                                    /* Oporavak toka */
  std::cin.ignore(10000, '\n');                        /* Brisanje spremnika */
  std::cin >> broj;
}
std::cout << "U redu, unijeli ste broj " << broj << std::endl;
```

Rad ovog isječka je prilično jasan. Ukoliko je nakon zahtijevanog unosa broja zaista unesen broj, ulazni tok će biti u ispravnom stanju, uvjet "`!std::cin`" neće biti tačan i tijelo "**`while`**" petlje neće se uopće ni izvršiti. Međutim, ukoliko tok dospije u neispravno stanje, unutar tijela petlje ispisujemo poruku upozorenja, vraćamo tok u ispravno stanje, uklanjamo iz ulaznog toka znakove koji su doveli do problema (pozivom funkcije "`ignore`" nad objektom ulaznog toka) i zahtijevamo novi unos. Nakon toga, uvjet petlje se ponovo provjerava i postupak se ponavlja sve dok unos ne bude ispravan (tj. sve dok uvjet "`!std::cin`" ne postane netačan).

U prethodnom primjeru, naredba za unos broja s tastature ponovljena je unutar petlje. Može li se ovo dupliranje izbjeći? Mada na prvi pogled izgleda da je ovo dupliranje nemoguće izbjeći (unos je potreban *prije* testiranja uvjeta petlje, dakle praktično *izvan* petlje, a potrebno ga je ponoviti u slučaju neispravnog unosa, odnosno *unutar* petlje), odgovor je ipak potvrdan, uz upotrebu jednog prilično "prljavog" trika, koji vrijedi objasniti, s obzirom da se često susreće u C++ programima. Ideja je da se sam unos s tastature ostvari kao *propratni efekat uvjeta petlje* (slično kao u primjeru u kojem smo demonstrirali zarez-operator). Naime, konstrukcija "`std::cin >> broj`" sama po sebi predstavlja izraz (s propratnim efektom unošenja vrijednosti u promjenljivu "`broj`"), koji pored činjenice da očitava vrijednost promjenljive "`broj`" iz ulaznog toka, također kao rezultat vraća sam objekat ulaznog toka "`cin`", na koji je dalje moguće primijeniti operator "`!`" sa ciljem testiranja ispravnosti toka. Stoga je savršeno ispravno formirati izraz poput "`!(std::cin >> broj)`" koji će očitati vrijednost promjenljive "`broj`" iz ulaznog toka, a zatim testirati stanje toka. Rezultat ovog izraza biće tačan ili netačan u zavisnosti od stanja toka, odnosno on predstavlja sasvim ispravan uvjet, koji se može iskoristiti za kontrolu "**`while`**" petlje! Kada uzmemo sve ovo u obzir, nije teško shvatiti kako radi sljedeći programski isječak:

```
int broj;
std::cout << "Unesite broj: ";
while(!(std::cin >> broj)) {
  std::cout << "Nemojte se šaliti, unesite broj!\n";
  std::cin.clear();                                    /* Oporavak toka */
  std::cin.ignore(10000, '\n');                        /* Brisanje spremnika */
}
std::cout << "U redu, unijeli ste broj " << broj << std::endl;
```

Znatno je čistije sljedeće rješenje, u kojem je prvo upotrijebljena konstrukcija "`for(;;)`" koja predstavlja petlju bez prirodnog izlaza, a koja se nasilno prekida pomoću naredbe "`break`" u slučaju ispravnog unosa (bez obzira na činjenicu da su mnogi teoretičari programiranja prilično kritički nastrojeni prema rješenjima kod kojih se vrši "iskakanje" iz unutrašnjosti petlje, što u ovom primeru radimo upravo pomoću naredbe "`break`"):

```
int broj;
std::cout << "Unesite broj: ";
for(;;) {
  std::cin >> broj;
  if(std::cin) break;
  std::cout << "Nemojte se šaliti, unesite broj!\n";
  std::cin.clear();                                    /* Oporavak toka */
  std::cin.ignore(10000, '\n');                        /* Brisanje spremnika */
}
std::cout << "U redu, unijeli ste broj " << broj << std::endl;
```

Već smo rekli da je korištenje objekata ulaznog i izlaznog toka "`cin`" i "`cout`" mnogo jednostavnije i fleksibilnije (kao što ćemo uskoro vidjeti) od korištenja funkcija iz biblioteke "`cstdio`", kao što su "`printf`" i "`scanf`". To ipak ne znači da biblioteku "`cstdio`" treba potpuno zaboraviti. Dva su slučaja u kojima treba koristiti funkcije iz ove biblioteke. Prvo, objekti poput "`cin`" i "`cout`" su, zbog svoje velike fleksibilnosti, veoma masivni i njihova upotreba osjetno produžuje dužinu generiranog izvršnog koda. Stoga, već i samo uključenje biblioteke "`iostream`" u program, koja deklarira i definira ove objekte, tipično *produžava dužinu generisanog izvršnog koda za nekoliko desetina do stotina kilobajta* (zavisno od kompajlera). U slučaju da želimo *generiranje kratkog izvršnog koda*, treba zaobići ovu biblioteku i koristiti funkcije iz biblioteke "`cstdio`". Drugo, funkcije iz biblioteke "`cstdio`" su *brže* od ekvivalentnih operacija s tokovima, tako da je u slučaju kada iz ulaznog toka treba pročitati više desetina hiljada podataka (ili kada treba ispisati više desetina hiljada podataka na izlazni tok) za kratko vrijeme, preporučuje se ne koristiti objekte tokova, nego funkcije iz biblioteke "`cstdio`". U svim ostalim slučajevima, upotreba biblioteke "`cstdio`" se u jeziku C++ *ne preporučuje*. Ipak, za naprednije studente nije na odmet znati da postoji veoma jednostavan način kako se rad s objektima iz biblioteke "`iostream`" često može osjetno ubrzati. Naime, u većini implementacija biblioteke "`iostream`", pri radu s objektima iz ove biblioteke mnogo se vremena gubi da se ostvari neophodna sinhronizacija koja omogućava da se u istom programu može miješati stari stil rada s ulazno-izlaznim uređajima koji se oslanja na funkcije iz biblioteke "`cstdio`", i novi stil koji se oslanja na objekte iz biblioteke "`iostream`". Ovu sinhronizaciju moguće je isključiti navođenjem naredbe

```
std::ios::sync_with_stdio(false);
```

Nakon što se izvrši ova naredba, rad s objektima toka iz biblioteke "`iostream`" biće tipično znatno brži, međutim tada se neće uporedo smjeti koristiti i funkcije iz biblioteke "`cstdio`" (što svakako nije ni preporučljivo), tačnije miješanje ova dva stila rada s ulazno-izlaznim uređajima imaće nepredvidljivo dejstvo. U slučaju potrebe, sinhronizacija se može ponovo uspostaviti izvršavanjem naredbe slične gore navedenoj, samo uz zamjenu riječi "`false`" sa "`true`".