Uvod u programiranje

- predavanja -

listopad 2020.

Tipovi podataka u programskom jeziku C

- 3. dio -

Osnovni tipovi podataka

Raspon realnih brojeva - standardna preciznost

- Najmanji normalizirani pozitivni broj koji se može prikazati je:

  $1.000\ 0000\ 0000\ 0000\ 0000\ 0000_2 \cdot 2^{-126} \approx 1.17 \cdot 10^{-38}$

- Najmanji denormalizirani pozitivni broj koji se može prikazati je:

  $0.000\ 0000\ 0000\ 0000\ 0000\ 0001_2 \cdot 2^{-126}$

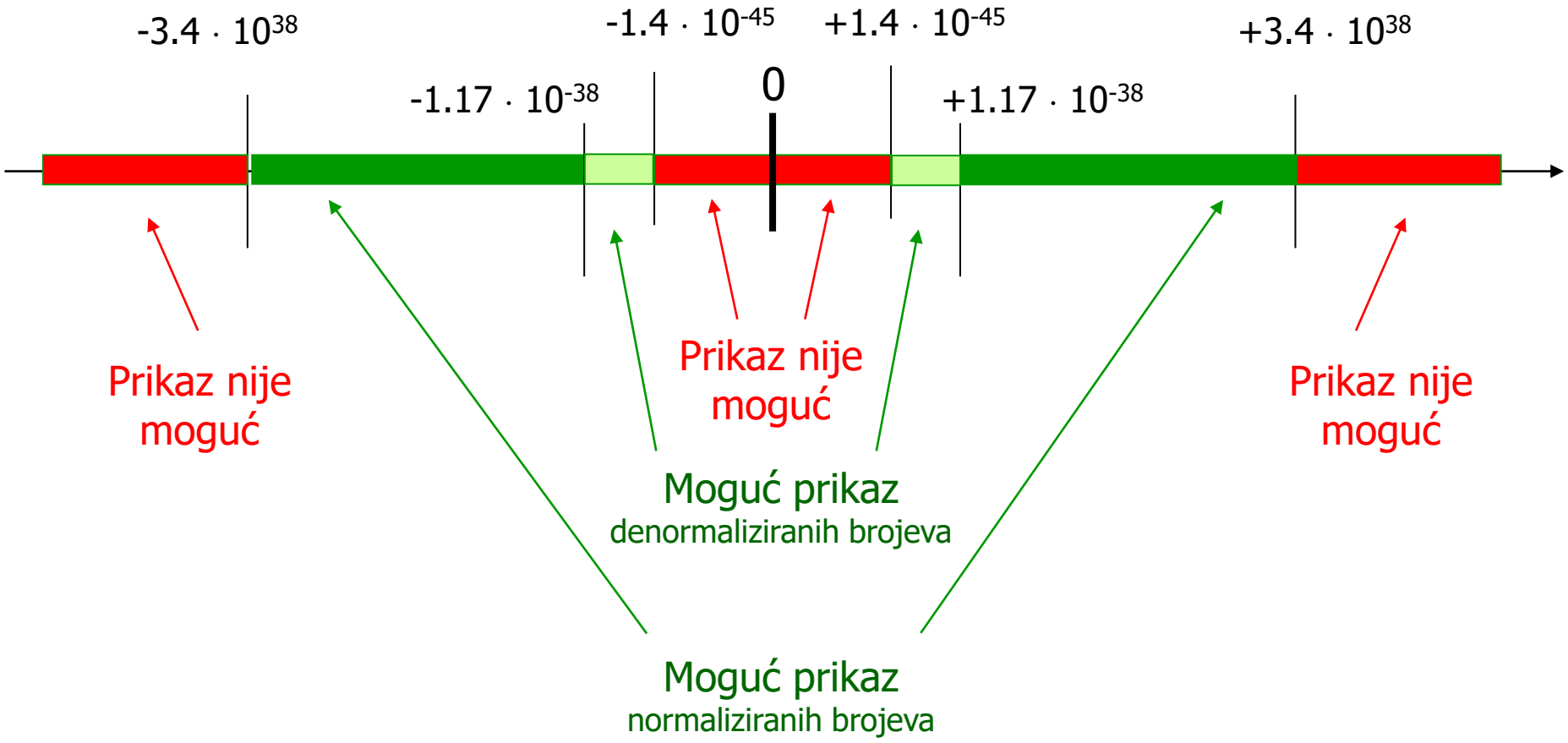
  $= 1 \cdot 2^{-149} \approx 1.4 \cdot 10^{-45}$

- Najveći broj koji se može prikazati je:

  $1.111\ 1111\ 1111\ 1111\ 1111\ 1111_2 \cdot 2^{127}$

  $\approx 1 \cdot 2^{128} \approx 3.4 \cdot 10^{38}$

Raspon realnih brojeva - standardna preciznost



Prikaz cijelih brojeva u računalu (podsjetnik)

- U registru računala s konačnim brojem bitova moguće je prikazati konačan broj različitih brojeva
- Koliko je različitih **cijelih** brojeva moguće prikazati u registru od *n* bitova?
  - 2^n različitih cijelih brojeva

- Skup cijelih brojeva $\mathbb{Z}$ je beskonačan - nije moguće prikazati sve brojeve iz skupa $\mathbb{Z}$, ali
  - za točan prikaz **svih** cijelih brojeva iz intervala $[0, 2^n-1]$ ili iz intervala $[-(2^{n-1}), 2^{n-1} - 1]$ dovoljan je registar od *n* bitova

Prikaz realnih brojeva u računalu

- Koliko bitova bi trebao imati registar u kojem bi se mogli točno prikazati **svi** realni brojevi iz intervala [-1.0, 1.0] ?
  - u intervalu ima beskonačno mnogo brojeva, tj. $|[-1.0, 1.0]| \equiv |R|$
  - stoga, potrebno je beskonačno mnogo bitova
- Samo konačan podskup realnih brojeva iz nekog intervala [-a, a] moguće je **točno** (bez pogreške) prikazati u registru s konačnim brojem bitova. Ostali realni brojevi iz tog intervala mogu se pohraniti samo kao njihove približne vrijednosti

Prikaz realnih brojeva u računalu

- Zašto se svi decimalni razlomci, iako imaju konačni broj dekadskih znamenaka iza decimalne točke, ne mogu prikazati konačnim brojem bitova
  - jer ekvivalentni binarni razlomak može imati beskonačni broj binarnih znamenki
    - $0.3_{10} = 0.01001100110011001100 \ldots _2$
  - jer binarni broj može imati previše znamenaka u odnosu na broj bitova raspoloživih za pohranu. Npr. za standardnu preciznost:
    - $4194304.125 = 1.00000000000000000000001_2 \cdot 2^{22}$
    - broj ima previše znamenaka mantise da bi se mogao bez pogreške prikazati u IEEE 754 formatu standardne preciznosti
    - isti broj se može bez pogreške prikazati u IEEE 754 formatu dvostruke preciznosti (objašnjeno kasnije)

Koliko se različitih realnih brojeva može prikazati

- Za IEEE 754 - standardnu preciznost
  - za svaki $K \in [0, 254]$
    - moguće su 2^{23} različite mantise
    - moguća su dva predznaka
    - ukupno $255 \cdot 2^{23} \cdot 2 = 4\,278\,190\,080$ različitih realnih brojeva
  - uz K = 255, moguće je prikazati $+\infty$, $-\infty$ i *NaN*
  - bez pogreške je moguće prikazati približno $4.3 \cdot 10^9$ različitih realnih brojeva iz intervala $[-3.4 \cdot 10^{38}, -1.4 \cdot 10^{-45}] \cup [1.4 \cdot 10^{-45}, 3.4 \cdot 10^{38}]$ i broj nula
    - za ostale realne brojeve (njih beskonačno mnogo) iz navedenih intervala moguće je prikazati samo približne vrijednosti (uz veću ili manju pogrešku)
    - realni brojevi izvan navedenih intervala se uopće ne mogu prikazati

Preciznost pohrane realnih brojeva

- Preciznost je svojstvo koje ovisi o količini informacije korištene za prikaz broja. Veća preciznost znači:
  - moguće je prikazati više različitih brojeva
  - brojevi su na brojevnom pravcu međusobno "bliži" (veća rezolucija)
  - smanjuje se najveća moguća pogreška pri prikazu broja

Pogreške pri prikazu realnih brojeva

- x - realni broj kojeg treba pohraniti u registar
- x* - približna vrijednost broja x koja je zaista pohranjena
  - Apsolutna pogreška α

    $$\alpha = |x^* - x|$$
  - Relativna pogreška ρ

    $$\rho = \alpha / |x|$$

- Primjer:
  - ako je u registru umjesto potrebne vrijednosti `x = 100.57` pohranjena vrijednost `x* = 100.625`

    $$\alpha = |100.625 - 100.57| = 0.055$$

    $$\rho = 0.055 / |100.57| = 0.000546$$

Pogreške pri prikazu realnih brojeva

- Najveća moguća *relativna pogreška* ovisi o broju bitova mantise
  - Za IEEE 754 standardne preciznosti:

$$\rho \leq 2^{-24} \approx 6 \cdot 10^{-8}$$

- Najveća moguća *apsolutna pogreška* ovisi o broju bitova mantise i konkretnom broju x koji se prikazuje
  - Za IEEE 754 standardne preciznosti:

$$\alpha \leq 2^{-24} \cdot |x| \approx 6 \cdot 10^{-8} \cdot |x|$$

Primjer

- Najveća apsolutna pogreška koja se uz standardnu preciznost može očekivati pri pohrani realnog broja 332.3452:

$$\alpha \leq 6 \cdot 10^{-8} \cdot |332.3452| \approx 2 \cdot 10^{-5}$$

```
float f1 = 332.3452f;
printf("%19.15f", f1);
```

  - očekuje se da će biti pohranjen broj $332.3452 \pm 2 \cdot 10^{-5}$

```
332.345214843750000
```

  - Zaista, apsolutna pogreška je $1.484375 \cdot 10^{-5}$, što je manje od $2 \cdot 10^{-5}$

Primjer

- Najveća apsolutna pogreška koja se uz standardnu preciznost može očekivati pri pohrani realnog broja `0.7`:

$$\alpha \leq 6 \cdot 10^{-8} \cdot |0.7| \approx 4.2 \cdot 10^{-8}$$

```
float f2 = 0.7f;
printf("%19.15f", f2);
```

  - očekuje se da će biti pohranjen broj $0.7 \pm 4.2 \cdot 10^{-8}$

```
0.699999988079071
```

  - Zaista, apsolutna pogreška je $1.1920929 \cdot 10^{-8}$, što je manje od $4.2 \cdot 10^{-8}$

Realni tipovi podataka

Tipovi podataka double i long double

Tip podatka double

- primjer definicije varijable tipa double

```
double temperatura;
```

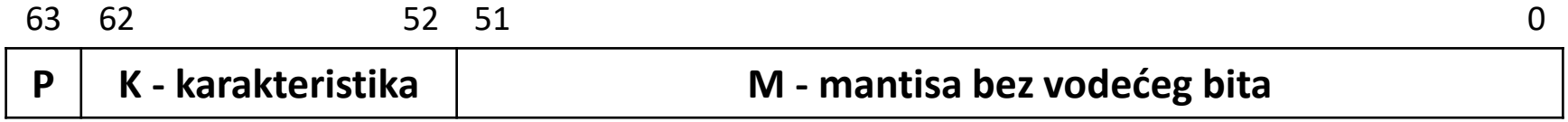
- primjeri konstanti tipa double
  - jednako kao konstante tipa float, ali bez znaka f ili F na kraju

```
2.                      = 2.0
-2.34                   = -2.34
-1.34e5                 = -1.34 · 10^5
9.1093E-31              = 9.1093 · 10^-31
```

- gcc i arhitektura x86_64
  - koristi se IEEE 754 dvostruke preciznosti (8 bajtova)
  - principi pohrane jednaki su kao kod IEEE 754 standardne preciznosti. Razlika je samo u povećanom broju bitova koji se koriste za karakteristiku i mantisu

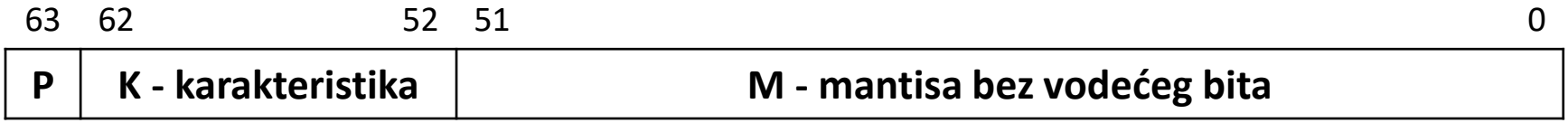
Realni brojevi dvostruke preciznosti

- IEEE 754 - *double precision*
  - najčešće se koristi za prikaz vrijednosti tipa double
  - binarni razlomak, u normaliziranom obliku, pohranjuje se u ukupno 64 bita (8 bajtova) u sljedećem obliku:



  - 1 bit za pohranu predznaka broja
    - sam sadržaj bita određuje predznak broja
  - 11 bitova za pohranu karakteristike
    - pozitivni ili negativni binarni eksponent (BE) preslikan u uvijek pozitivnu karakteristiku (K)
  - 52 bita za pohranu mantise *bez vodećeg bita*
    - pohrana prvog bita (koji je uvijek jedinica) je nepotrebna

Realni brojevi dvostruke preciznosti



- predznak P: na mjestu najznačajnijeg bita upisuje se
  - 1 ako se radi o negativnom broju
  - 0 ako se radi o pozitivnom broju
- karakteristika K:
  - raspon binarnog eksponenta BE $\in$ [-1022, 1023]
    - radi izbjegavanja dvojnog komplementa za prikaz negativnih eksponenata, umjesto BE pohranjuje se karakteristika K
    - raspon karakteristike: K $\in$ [0, 2047], K = BE + 1023 $\Rightarrow$ BE $\in$ [-1022, 1023]
    - K = 0 i K = 2047 se koriste za posebne slučajeve
- mantisa M:
  - ne pohranjuje se vodeći bit mantise - jedinica ispred binarne točke

Realni brojevi dvostruke preciznosti

- Posebni slučajevi
  - kada je K=0 i svi bitovi mantise su postavljeni na 0, radi se o prikazu realnog broja 0 (tada se ne smatra da je vodeći bit implicitno 1)
  - kada je K = 0 i postoje bitovi mantise koji nisu 0, radi se o "denormaliziranom" broju
  - kada je K = 2047 i svi bitovi mantise su postavljeni na 0, radi se o prikazu $+\infty$ ili $-\infty$, ovisno o bitu za predznak
  - Ako je K=2047 i postoje bitovi mantise koji nisu 0, radi se o nedefiniranoj vrijednosti ili vrijednosti koju nije moguće prikazati (*NaN - Not a Number*), tj. ne radi se o legalnom prikazu broja

Raspon realnih brojeva - dvostruka preciznost

- Na povećanje raspona utječe povećanje broja bitova koji se koriste za karakteristiku
  - Najmanji normalizirani pozitivni broj koji se može prikazati je:

    $1.000...000_2 \cdot 2^{-1022} \approx 2.2 \cdot 10^{-308}$
  - Najmanji denormalizirani pozitivni broj koji se može prikazati je:

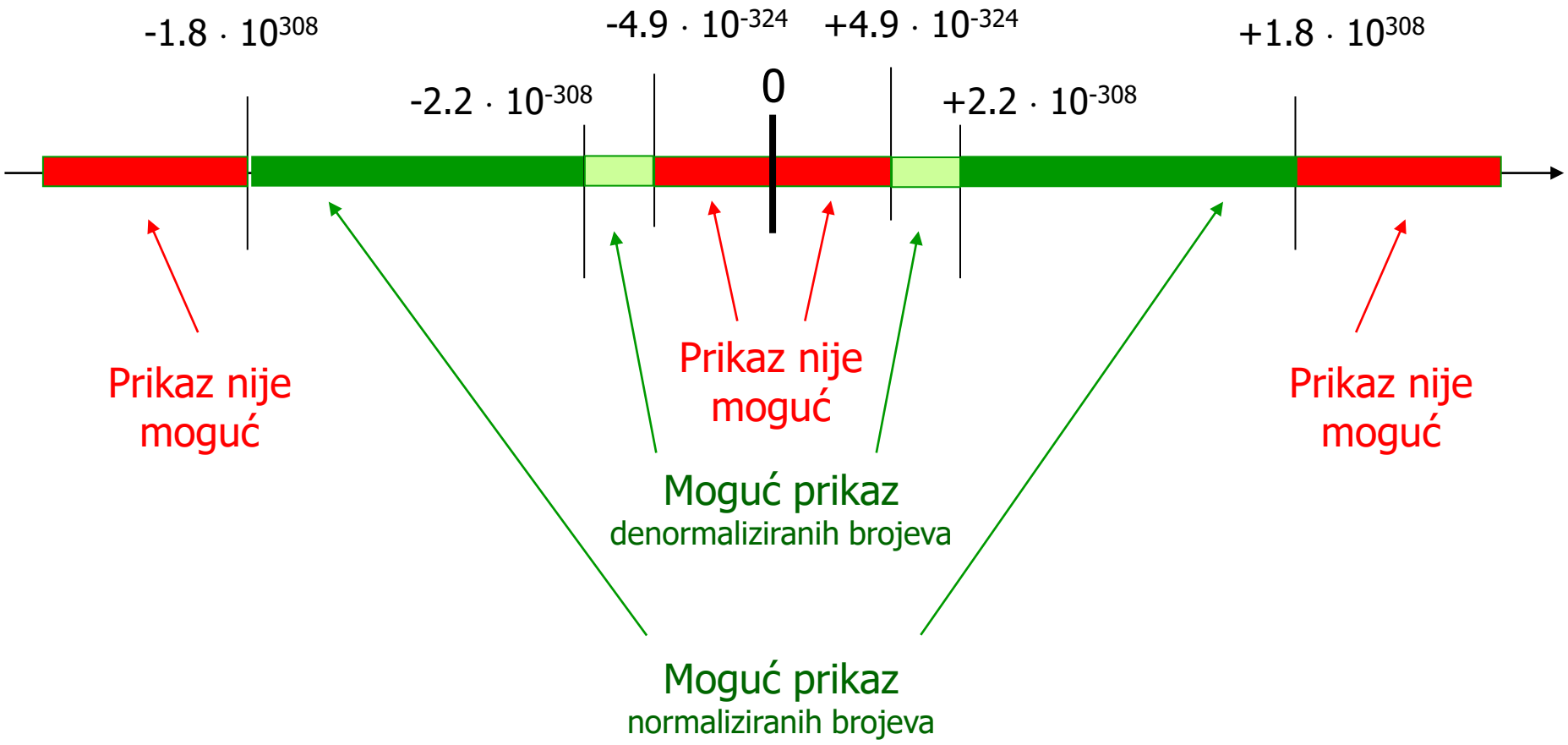
    $0.000...0001_2 \cdot 2^{-1022}$

    $= 1 \cdot 2^{-1074} \approx 4.9 \cdot 10^{-324}$
  - Najveći broj koji se može prikazati je:

    $1.111...1111_2 \cdot 2^{1023}$

    $\approx 1 \cdot 2^{1024} \approx 1.8 \cdot 10^{308}$

Raspon realnih brojeva - dvostruka preciznost



Pogreške za IEEE 754 dvostruke preciznosti

- Povećana preciznost, odnosno smanjenje najveće moguće relativne pogreške, rezultat je korištenja većeg broja bitova mantise
- Najveća moguća *relativna pogreška*
  - Za IEEE 754 dvostruke preciznosti:

$$\rho \leq 2^{-53} \approx 1.1 \cdot 10^{-16}$$

- Najveća moguća *apsolutna pogreška*
  - Za IEEE 754 dvostruke preciznosti:

$$\alpha \leq 2^{-53} \cdot |x| \approx 1.1 \cdot 10^{-16} \cdot |x|$$

Primjer

- Najveća apsolutna pogreška koja se uz dvostruku preciznost može očekivati pri pohrani realnog broja `0.7`:

$$\alpha \leq 1.1 \cdot 10^{-16} \cdot |0.7| \approx 7.7 \cdot 10^{-17}$$

```
double f = 0.7;
printf("%19.17f", f);
```

  - očekuje se da će biti pohranjen broj $0.7 \pm 7.7 \cdot 10^{-17}$

```
0.69999999999999996
```

  - Zaista, apsolutna pogreška je $4.0 \cdot 10^{-17}$, što je manje od $7.7 \cdot 10^{-17}$

Realni brojevi povećane preciznosti

- IEEE 754 - *extended precision*
  - broj i raspored bitova nisu propisani standardom već ovise o implementaciji
  - gcc povećanu preciznost implementira pomoću 80 bitova, pri čemu se za pohranu koristi 12 bajtova
    - 15 bitova karakteristike, 63 bita mantise
    - K = BE + 16383
    - koristi se za pohranu tipa podataka long double
    - raspon $\approx \pm [3.65 \cdot 10^{-4951}, 1.19 \cdot 10^{4932}]$
    - najveća relativna pogreška $\approx 2^{-64} \approx 5.4 \cdot 10^{-20}$

Tip podatka long double

- primjer definicije varijable tipa long double

```
long double temperatura;
```

- primjeri konstanti tipa long double
  - jednako kao konstante tipa double, ali sa znakom l ili L na kraju

```
2.L                     = 2.0
-2.34l                  = -2.34
-1.34e5L                = -1.34 · 10^5
9.1093E-31l             = 9.1093 · 10^-31
```

- gcc i arhitektura x86_64
  - koristi se IEEE 754 povećane preciznosti (80 bitova)

Konverzijske specifikacije za scanf i printf

Tip	spec.
float	%f
double	%lf
long double	%Lf

- Primjer

```
float x;
double y;
long double z;
scanf("%f %lf %Lf", &x, &y, &z);
printf("%5.2f %.3lf %Lf", x, y, z);
```

```
1.333 -15 6.1e8↵
 1.33 -15.000 610000000.000000
```

Napomena: %Lf ne radi za gcc prevodilac na operacijskom sustavu Windows

Numeričke pogreške

- Neki dekadski brojevi se ne mogu prikazati pomoću konačnog broja binarnih znamenaka
  - Primjer:

```
float f = 0.3f;
printf("%18.16f", f);
```

```
0.3000000119209290
```

```
0011 1110 1001 1001 1001 1001 1001 1001 1001 1001 ...
0011 1110 1001 1001 1001 1001 1001 1010 (zaokruženo na bližu vrijednost)
```

- Neki realni brojevi imaju konačan broj, ali ipak "previše" binarnih znamenaka
  - Primjer:

```
float f = 4194304.125f;
printf("%14.6f", f);
```

```
4194304.000000
```

```
0100 1010 1000 0000 0000 0000 0000 0000 01
0100 1010 1000 0000 0000 0000 0000 0000 (zaokruženo na bližu vrijednost)
```

Numeričke pogreške

- Računanje s brojevima bitno različitog reda veličine može dovesti do numeričke pogreške
  - Primjer

```
float f = 6000000.0f; float malif = 0.25f;
f = f + malif;
f = f + malif;
f = f + malif;
f = f + malif;
printf("%f ", f);
```

```
6000000.000000
```

  - bolje:

```
float malaSuma = 0.f;
malaSuma = malaSuma + malif;
malaSuma = malaSuma + malif;
malaSuma = malaSuma + malif;
malaSuma = malaSuma + malif
f = f + malaSuma;
printf("%f ", f);
```

```
6000001.000000
```

Numeričke pogreške

- Potreban je oprez kod uspoređivanja realnih vrijednosti
  - jesu li dva broja "jednaka"
    - možda treba uzeti u obzir neku toleranciju
  - kada je kontrolna varijabla petlje dosegla granicu za prekid petlje?

Primjer

- Učitati pozitivan cijeli broj **n** (nije potrebno provjeravati je li učitan ispravan broj). Zatim ispisati realne brojeve od 0.5 do **n** s korakom od 0.1. Za ispis realnih brojeva koristiti formatsku specifikaciju %12.9f.

```
Upisite n > 2↵
 0.500000000↵
 0.600000000↵
 0.700000000↵
 0.800000000↵
 0.900000000↵
 1.000000000↵
 1.100000000↵
 1.200000000↵
 1.300000000↵
 1.400000000↵
 1.500000000↵
 1.600000000↵
 1.700000000↵
 1.800000000↵
 1.900000000↵
 2.000000000↵
```

Primjer

```
#include <stdio.h>

int main(void) {
   int n;
   float x;
   printf("Upisite n > ");
   scanf("%d", &n);

   for (x = 0.5f; x <= n; x = x + 0.1f) {
      printf("%12.9f\n", x);
   }
}
```

```
Upisite n > 2↵
 0.500000000↵
 0.600000024↵
 0.700000048↵
 0.800000072↵
 0.900000095↵
 1.000000119↵
 1.100000143↵
 1.200000167↵
 1.300000191↵
 1.400000215↵
 1.500000238↵
 1.600000262↵
 1.700000286↵
 1.800000310↵
 1.900000334↵
```

- zašto se ne ispisuju točne vrijednosti?
  - čak niti za 1.0 ili 1.5?
- zašto se petlja zaustavila prerano?

Primjer

- Poboljšanje:
  - dopustiti određenu toleranciju kod uspoređivanja brojeva/uvjeta za prekid petlje

```
#include <stdio.h>
#define TOLER 0.00001f

int main(void) {
   int n;
   float x;
   printf("Upisite n > ");
   scanf("%d", &n);

   for (x = 0.5f; x <= n + TOLER; x = x + 0.1f) {
      printf("%12.9f\n", x);
   }
}
```

```
Upisite n > 2↵
 0.500000000↵
 0.600000024↵
 0.700000048↵
 0.800000072↵
 0.900000095↵
 1.000000119↵
 1.100000143↵
 1.200000167↵
 1.300000191↵
 1.400000215↵
 1.500000238↵
 1.600000262↵
 1.700000286↵
 1.800000310↵
 1.900000334↵
 2.000000238↵
```

Primjer

- Poboljšanje:
  - gdje je moguće, koristiti cijele brojeve
    - u operacijama s cijelim brojevima nema numeričkih pogrešaka

```
#include <stdio.h>

int main(void) {
   int n, brojac;
   float x;
   printf("Upisite n > ");
   scanf("%d", &n);

   for (brojac = 5;
        brojac <= n * 10;
        brojac = brojac + 1) {
      x = brojac / 10.f;
      printf("%12.9f\n", x);
   }
}
```

```
Upisite n > 2↵
 0.500000000
 0.600000024
 0.699999988
 0.800000012
 0.899999976
 1.000000000
 1.100000024
 1.200000048
 1.299999952
 1.399999976
 1.500000000
 1.600000024
 1.700000048
 1.799999952
 1.899999976
 2.000000000
```

Primjer

- Objasniti razlike u rezultatima sljedećih operacija:

```
double x;

x = 0.1f + 0.1f;
printf("%20.18f\n", x);

x = 0.1f + 0.1;
printf("%20.18f\n", x);

x = 0.1 + 0.1;
printf("%20.18f\n", x);
```

```
0.200000002980232240
0.200000001490116120
0.200000000000000010
```

Definicija tipa podataka

typedef

Definicija tipa

- kreiranje zamjenskog imena (sinonim) za postojeći tip podataka
  - unapređenje razumljivosti (dokumentiranosti) programskog koda
  - unapređenje prenosivosti (portabilnosti) programskog koda
  - pojednostavljivanje naredbi za definiciju varijabli
- opći oblik naredbe

  `typedef nazivPostojecegTipa noviNazivTipa;`
  - naredba se mora nalaziti ispred (ne nužno neposredno) naredbi koje će koristiti novi naziv tipa
- Primjer:

```
int main(void) {
   typedef int cijeliBroj_t;
   typedef float realniBroj_t;
   cijeliBroj_t m, n;
   ...
   realniBroj_t x, y;
```

Ovo je samo ilustracija.
Ne definirati nove tipove samo radi prijevoda!

Unapređenje razumljivosti (dokumentacija)

- postojećem tipu podataka dodati zamjensko ime
  - dodatno opisuje domenu vrijednosti varijabli

```
typedef unsigned int maticniBroj_t;
typedef unsigned char ocjenaBroj_t;
typedef char ocjenaSlovo_t;
...
maticniBroj_t mbrStud1 = 123456789;
ocjenaBroj_t ocjeneGrupe[10] = {1, 1, 5, 4, 2, 1, 1, 4, 3, 5};
ocjenaSlovo_t najmanjaSlovnaOcjena = 'F';
```

pretpostavlja se da mbr. nema više od 9 znamenaka i da se za pohranu tipa int koristi 4 bajta

konvencija (nije pravilo): dodati _t na kraj imena

razumljivije je od

```
unsigned int mbrStud1 = 123456789;
unsigned char ocjeneGrupe[10] = {1, 1, 5, 4, 2, 1, 1, 4, 3, 5};
char najmanjaSlovnaOcjena = 'F';
```

Unapređenje prenosivosti

- ako prethodni program treba prenijeti na platformu na kojoj je tip podatka int pohranjen u samo dva bajta [0, 65 535], dok je long int pohranjen u 4 bajta [0, 4 294 967 295]
  - umjesto unsigned int za sve podatke o matičnim brojevima trebat će koristiti unsigned long int
  - ako se u programu koristilo zamjensko ime tipa maticniBroj_t, dovoljno je (na samo jednom mjestu!) promijeniti definiciju tipa i prevesti program na novoj platformi
    - inače, trebalo bi promijeniti tip u definiciji svih varijabli u kojima se pohranjuje matični broj

```
...
~~typedef unsigned int maticniBroj_t;~~
typedef unsigned long int maticniBroj_t;
...
```

Prenosivost - ugrađene definicije tipova

- neka zamjenska imena za tipove unaprijed su definirana
  - npr. operator sizeof (koji će detaljnije biti objašnjen kasnije) vraća veličinu objekta (npr. varijable) izraženu u bajtovima
  - radi se o cijelom broju koji na različitim platformama može imati različite raspone i biti definiran na temelju različitih osnovnih tipova
    - npr. unsigned int ili unsigned long ili unsigned long long
  - u stdlib.h je definiran tip size_t koji se u većini slučajeva može na siguran način koristiti kao da se radi o tipu unsigned int

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
   int n;
   size_t velicina;
   velicina = sizeof(n);
   printf("%u", velicina);
   ...
```

4

Prenosivost - ugrađene definicije tipova

- broj bajtova koji se koristi za pohranu cjelobrojnih tipova podataka u C-u nije propisan jezikom
  - rasponi se razlikuju u različitim prevodiocima i arhitekturama
  - ako je potrebno fiksirati preciznost i raspon, neovisno o prevodiocu i arhitekturi

```
#include <stdio.h>
#include <stdint.h>
int main(void) {
   int m;          veličina (dakle i raspon) ovisi o prevodiocu
   int8_t n1;      točno 1 bajt, signed
   int16_t n2;     točno 2 bajta, signed
   int32_t n4;     točno 4 bajta, signed
   int64_t n8;     točno 8 bajtova, signed
   uint8_t un1;    točno 1 bajt, unsigned
   uint16_t un2;   točno 2 bajta, unsigned
   ...
```

Pojednostavljivanje naredbi za definiciju

- definicija tipa često se koristi umjesto deklaracije strukture

pomoću deklaracije strukture:

```
struct datum_s {
   int dan;
   int mj;
   int god;
};
struct interval_s {
   struct datum_s dat_od;
   struct datum_s dat_do;
};

struct interval_s zim_rok =
  {{11, 2, 2019}, {22, 2, 2019}};

struct interval_s ljetni_rok =
  {{17, 6, 2019}, {3, 7, 2019}};
```

pomoću definicije tipa:

```
typedef struct {
   int dan;
   int mj;
   int god;
} datum_t;
typedef struct {
   datum_t dat_od;
   datum_t dat_do;
} interval_t;

interval_t zim_rok =
  {{11, 2, 2019}, {22, 2, 2019}};

interval_t ljetni_rok =
  {{17, 6, 2019}, {3, 7, 2019}};
```

Izrazi s različitim tipovima podataka

Implicitna pretvorba tipova podataka

Izrazi s različitim tipovima podataka

- Što je rezultat operacije `i + f`
  - `2. + 2.99` $\Rightarrow$ `4.99` ili `2 + 2` $\Rightarrow$ `4`
- Što je rezultat operacije `i >= f`
  - `2. >= 2.99` $\Rightarrow$ `0` ili `2 >= 2` $\Rightarrow$ `1`

```
int i;
float f;
i = 2;
f = 2.99f;
```

- Kada su operandi različitog tipa, prije obavljanja operacije obavlja se implicitna (automatska) pretvorba (konverzija) "manje važnog" tipa operanda u "važniji" od tipova operanada koji sudjeluju u operaciji.
- U prikazanim primjerima, prije nego se obavi operacija, vrijednost koja se nalazi u varijabli `i` (int je "manje važan") pretvara se u vrijednost `2.f` (float je "važniji" tip)
  - pri tome tip i sadržaj varijable `i` ostaje nepromijenjen.

Implicitna pretvorba tipova podataka

- Implicitna pretvorba tipova podataka u aritmetičkim i relacijskim izrazima obavlja se prema jednom od sljedećih 6 pravila. Treba iskoristiti prvo po redu pravilo koje se može primijeniti na konkretan slučaj!

  1. Ako je jedan od operanada tipa **long double**, preostali operand se pretvara u tip **long double**
  2. Ako je jedan od operanada tipa **double**, preostali operand se pretvara u tip **double**
  3. Ako je jedan od operanada tipa **float**, preostali operand se pretvara u tip **float**
  4. Ako je jedan od operanada tipa **long long**, preostali operand se pretvara u tip **long long**
  5. Ako je jedan od operanada tipa **long**, preostali operand se pretvara u tip **long**
  6. Operandi tipa **_Bool**, **short** i **char** pretvaraju se u tip **int**

- Kada se u izrazima pojavljuju unsigned tipovi, pravila pretvorbe su složenija. Zato se ovdje neće razmatrati.

Primjer

```
_Bool b;        char c;         short s;
int i;          long l;         long long ll;
float f;        double d;       long double ld;
```

Izraz	Pretvorba tipa prije obavljanja operacije	Prema pravilu
`ld * i`	sadržaj **i** → long double	1.
`c % i`	sadržaj **c** → int	6.
`s / f`	sadržaj **s** → float	3.
`l % i`	sadržaj **i** → long	5.
`d + c`	sadržaj **c** → double	2.
`s - c`	sadržaj **s** → int, sadržaj **c** → int	6.
`b == ll`	sadržaj **b** → long long	4.

Pretvorba tipova podataka kod pridruživanja

- Kod operacije pridruživanja, podatak s desne strane operatora pridruživanja pretvara se u tip podatka lijeve strane izraza
  - Primjer:

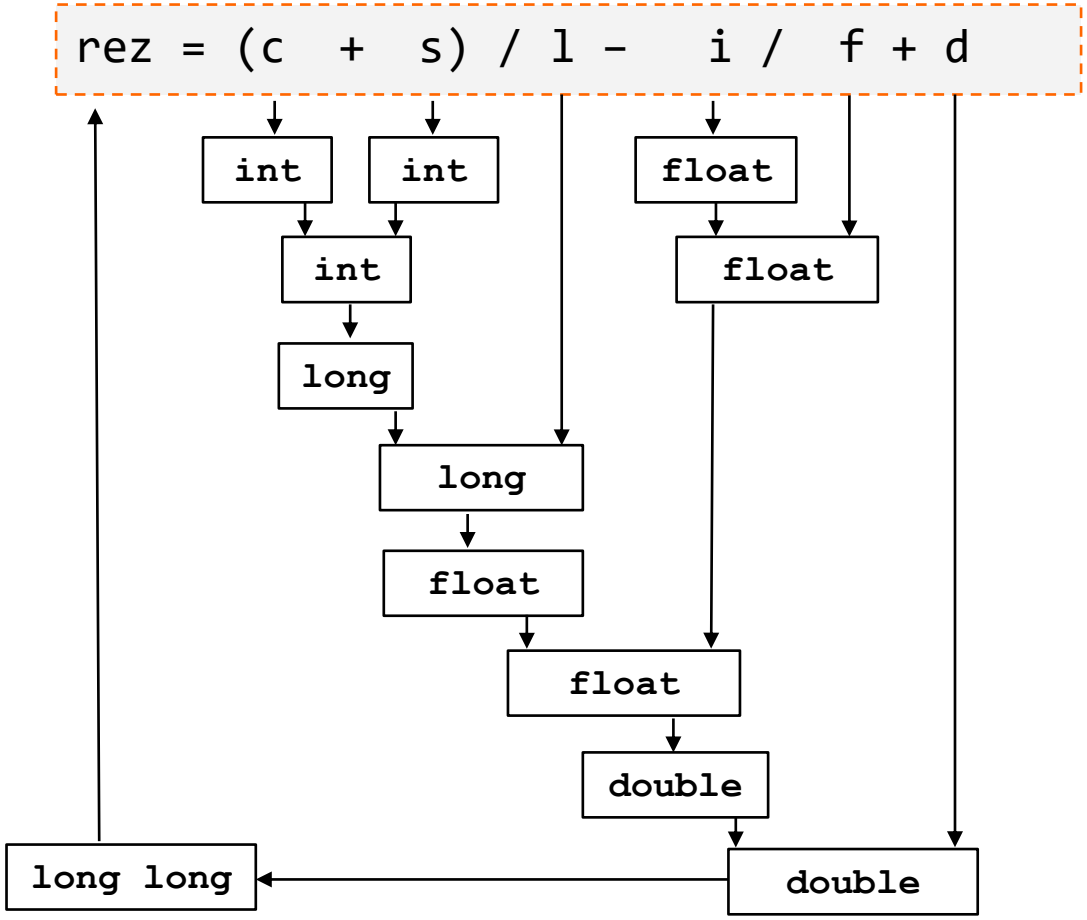
```
int i;
char c = '2';
long double ld;

i = 1.75;
ld = i + c;
```

  - nakon inicijalizacije varijabli
    - vrijednost konstante tipa double (1.75) pretvara se u vrijednost tipa int (1) i pridružuje varijabli i
    - vrijednost varijable c tipa char (50 u jednom bajtu) pretvara se u int (50 u četiri bajta), izračunava se rezultat tipa int (51) , pretvara se u long double (51.L) i pohranjuje u varijablu ld

Primjer

```
char c;
short int s;
int i;
long l;
float f;
double d;
long long rez;
```



Eksplicitna (zadana) pretvorba tipa podataka

- Opći oblik zadane (eksplicitne) pretvorbe (eng. cast operator):

  `(tip_podatka) izraz`

- Primjer: radi realnog dijeljenja dviju cjelobrojnih varijabli, korištena je nespretna pomoćna operacija množenja realnom konstantom

```
int i, j;
float x;
...
x = 1.f * i / j;
```

  - operator *cast* omogućava puno bolje rješenje:

```
x = (float) i / j;
```

  - analizirajte zašto sljedeće nije ispravno:

```
x = (float) (i / j);
```

Eksplicitna (zadana) pretvorba tipa podataka

- Operator *cast* ne treba primjenjivati bez potrebe
  - Umjesto korištenja operatora za eksplicitnu pretvorbu tipa nad konstantom, koristiti konstantu odgovarajućeg tipa
  - Primjer: ako se u realnu varijablu x želi pridružiti vrijednost ⅓

```
float x;

x = 1 / 3;              neispravno! U varijablu x će se upisati vrijednost 0.0
x = (float) 1 / 3;      nepotrebno! Nema potrebe za operatorom cast
x = 1.f / 3;            ispravno
x = 1 / 3.f;            ispravno
x = 1.f / 3.f;          ispravno
```

Pretvorba tipova podataka - mogući gubici

- Potrebno je obratiti pozornost da se pri pretvorbi jednog u drugi tip podatka može "izgubiti" manji ili veći dio informacije

```
int i;
float x;
i = 2.99999f;
x = 2147483638;
printf("%d\n%f", i, x);
```

```
2
2147483648.000000
```

  - "izgubljene" su decimale 0.99999 kod pretvorbe vrijednosti
  - float tip ima premalo bitova mantise za točan prikaz vrijednosti. Najbliži broj koji se može prikazati je za 10 veći od 2147483638

Primjer

```
int i = 2147000000;
float x;
x = i;
printf("%f\n", x);
x = x + 1.f;
printf("%f\n", x);
```

```
2147000064.000000
2147000064.000000
```

- pri operaciji pridruživanja x = i, cjelobrojna vrijednost 2147000000 pretvorila se u tip podatka float. Najbliža realna vrijednost koja se može prikazati u standardnoj preciznosti je 2147000064.0
- tijekom zbrajanja vrijednosti 2147000064.0 i 1.0, došlo je do numeričke pogreške zbog zbrajanja vrlo velike i vrlo male vrijednosti, rezultat zbrajanja je 2147000064.0 te se ta vrijednost pridružila varijabli x
- obje pogreške posljedica su ograničenog broja bitova mantise

Primjer

```
double d1 = 123456789.;
double d2 = 1.e40;
float f1 = d1;
float f2 = d2;
printf("%lf\n%f\n"
       "%lf\n%f",
        d1, f1,
        d2, f2);
```

```
123456789.000000
123456792.000000
10000000000000000303786028427003666890752.000000
inf
```

- nedovoljno bitova mantise u f1 za točnu pohranu broja 123456789
- nedovoljno bitova mantise u d2 za točnu pohranu broja $1 \cdot 10^{40}$
- nedovoljni raspon (karakteristika) u f2 za pohranu broja $1 \cdot 10^{40}$

Primjer

- Koje su vrijednosti i tipovi rezultata evaluacije sljedećih izraza:

1. `3 / 2 * 2`
2. `3 / 2 * 2.`
3. `3 / 2. * 2`
4. `3 / (float)2 * 2`
5. `(double)3 / 2`
6. `(double)(3 / 2)`
7. `2+0.5   * 4`
8. `(2 + 0.5) * 4`
9. `(int)(0.5 + 2) * 4`
10. `(float) ((int)1.5 + 2) / 4`
11. `(int)1.6 + (int)1.6`
12. `(int)(1.6 + 1.6)`