Uvod u programiranje

- predavanja -

listopad 2019.

5. Tipovi podataka u programskom jeziku C

Osnovni tipovi podataka

Osnovni tipovi podataka

- Tip podatka varijable određuje karakteristike podatka koji u tu varijablu može biti pohranjen i operacije koje se nad podatkom mogu obaviti
  - U programskom jeziku C na raspolaganju su sljedeći osnovni tipovi podataka
    - cjelobrojni tipovi (*integer types*)
      - char
      - int
      - _Bool
    - brojevi s pomičnim zarezom (*floating point types*)
      - float
      - double
  - opcionalnim kvalifikatorima (short/long, signed/unsigned) opisuju se dodatne karakteristike nekih od navedenih tipova

Potrebno (i očekivano) predznanje

- dekadski, binarni, oktalni, heksadekadski brojevni sustavi
  - pretvorbe cijelih brojeva između bilo kojih od navedenih brojevnih sustava: $27_{10} = 11011_2 = 33_8 = 1B_{16}$
- način pohrane pozitivnih i negativnih cijelih brojeva u računalu
- zbrajanje pozitivnih i negativnih brojeva u binarnom brojevnom sustavu
- raspon brojeva koji se mogu pohraniti u registru od n bitova
  - ako se pohranjuju samo pozitivni brojevi

    $[0, 2^n - 1]$
  - ako se pohranjuju i pozitivni i negativni brojevi

    $[-(2^{n-1}), 2^{n-1} - 1]$

- Zašto je za programera važno poznavati način pohrane podataka?

Primjer

- Napisati program koji s tipkovnice učitava cijeli broj n, a zatim na zaslon ispisuje rezultat operacije n + 5

```
#include <stdio.h>

int main(void) {
   int n, m;
   scanf("%d", &n);
   m = n + 5;
   printf("%d", m);
   return 0;
}
```

za n = 2 000 000 000

```
2000000000↵
2000000005
```

✓

za n = -2 000 000 000

```
-2000000000↵
-1999999995
```

✓

za n = 2 147 483 647

```
·2147483647↵
-2147483644
```

?

Primjer

- Napisati program koji s tipkovnice učitava realni broj x, a zatim na zaslon ispisuje rezultat operacije x+0.125, u ukupnoj širini od 15 znakova, s pet znamenki iza decimalne točke

```
#include <stdio.h>

int main(void) {
   float x, y;
   scanf("%f", &x);
   y = x + 0.125f;
   printf("%15.5f", y);
   return 0;
}
```

za x = 2 097 151.0

```
··2097151.0↵
··2097151.12500
```

✓

za x = -2 097 151.0

```
·-2097151.0↵
·-2097150.87500
```

✓

za x = 2 097 152.0

```
··2097152.0↵
··2097152.00000
```

?

za x = 1 000 000.3

```
··1000000.3↵
··1000000.43750
```

?

Osnovni tipovi podataka

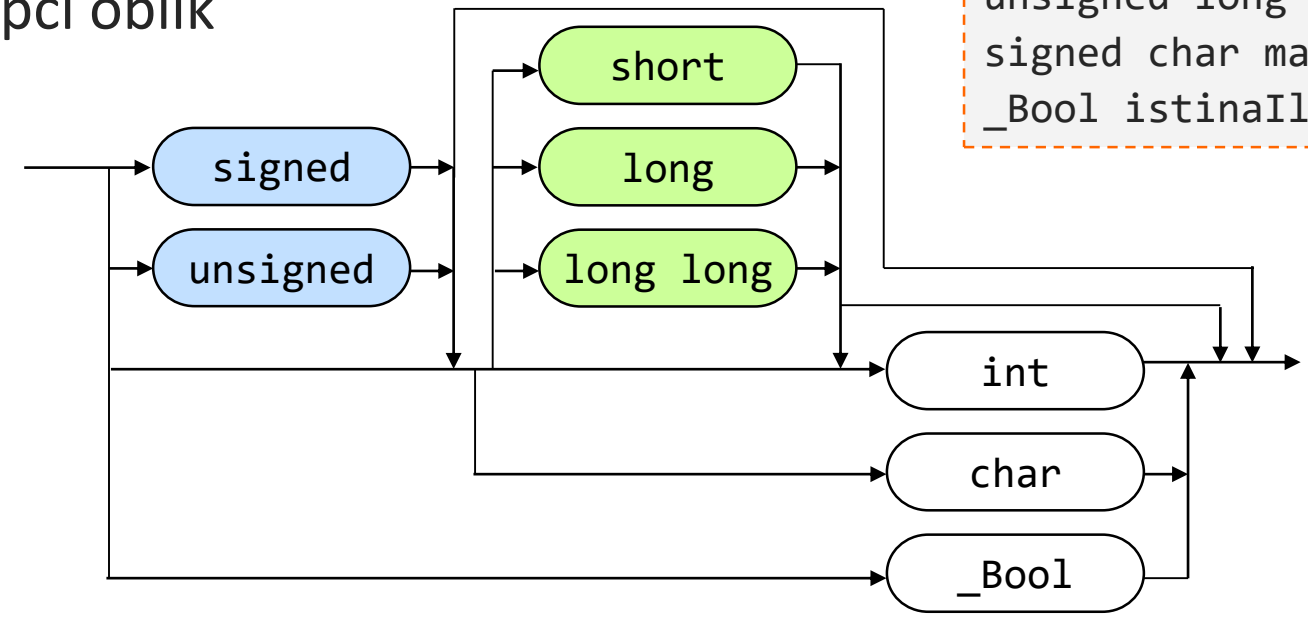
Cjelobrojni tipovi podataka

Cjelobrojni tipovi podataka

Tip podatka int

Cjelobrojni tipovi

- *integer types*
- opći oblik



Primjer definicije varijabli

```
unsigned long int velikiBroj;
signed char maliBroj;
_Bool istinaIliLaz;
```

- Prefiksi (kvalifikatori):
  - signed/unsigned: određuju mogu li se u varijablu tipa int ili char pohraniti samo pozitivni ili i negativni brojevi
  - short/long/long long: smanjuju/povećavaju raspon brojeva koji se mogu pohraniti u varijablu tipa int

Tip podatka int (*integer*) - varijante

- kvalifikator *signed* i ključna riječ *int* se podrazumijevaju (*default*), stoga se mogu ispustiti

Puni naziv tipa	Sinonimi	Preporučeni oblik u programima
signed int	int, signed	**int**
unsigned int	unsigned	**unsigned int**
signed short int	signed short, short int, short	**short**
unsigned short int	unsigned short	**unsigned short**
signed long int	signed long, long int, long	**long**
unsigned long int	unsigned long	**unsigned long**
signed long long int	signed long long, long long int, long long	**long long**
unsigned long long int	unsigned long long	**unsigned long long**

Kvalifikatori short, long i long long

- Tip podatka int, s obzirom na dopušteni raspon brojeva koji se mogu pohraniti (određen brojem binarnih znamenki), može se dodatno kvalificirati kao short, long ili long long
- Raspon brojeva za pojedine tipove nije propisan jezikom, ali vrijede pravila:
  - short: minimalno 2 bajta i ne smije imati raspon veći od int
  - int: minimalno 2 bajta i ne smije imati raspon veći od long
  - long: minimalno 4 bajta i ne smije imati raspon veći od long long
  - long long: minimalno 8 bajtova
- Stvarni rasponi brojeva ovisi o implementaciji prevodioca i arhitekturi računala

Primjer

- gcc i arhitektura x86_64

Tip	Bajtova	Raspon brojeva koje je moguće pohraniti
signed short int	2	-32 768, 32767
signed int	4	-2 147 483 648, 2 147 483 647
signed long int	4	-2 147 483 648, 2 147 483 647
signed long long int	8	-9 223 372 036 854 775 808, 9 223 372 036 854 775 807

Kvalifikatori signed i unsigned

- Vrijednost (binarne znamenke) pohranjena u varijablu tipa signed smatra se pozitivnom ili negativnom, ovisno o vrijednosti najznačajnijeg bita
- Vrijednost pohranjena u varijablu tipa unsigned, bez obzira na vrijednost najznačajnijeg bita, uvijek se smatra pozitivnom
  - time se udvostručuje najveća pozitivna vrijednost koja može biti pohranjena
- Primjer: ako varijabla tipa short koristi 2 bajta
  - raspon brojeva u signed varijabli: [-32 768, 32 767]
  - raspon brojeva u unsigned varijabli: [0, 65 535]

Primjer

- Rezultat operacije u kojoj se koristi signed ili unsigned int

```
unsigned int ua = 2147483647;
ua = ua + 1;
if (ua > 0)
   printf("veci od nule\n");
else
   printf("manji od nule");
```

01111111111111111111111111111111

10000000000000000000000000000000

promatra se isključivo kao pozitivni broj

ispis: veci od nule

```
signed int a = 2147483647;
a = a + 1;
if (a > 0)
   printf("veci od nule\n");
else
   printf("manji od nule\n");
```

01111111111111111111111111111111

10000000000000000000000000000000

promatra se kao pozitivni ili negativni broj

ispis: manji od nule

Primjer

- prevodilac gcc i arhitektura x86_64

Tip	Bajtova	Raspon brojeva koje je moguće pohraniti
unsigned short int	2	0, 65535
unsigned int	4	0, 4 294 967 295
unsigned long int	4	0, 4 294 967 295
unsigned long long int	8	0, 18 446 744 073 709 551 615

Kvalifikatori signed i unsigned

- savjet: ne miješati signed i unsigned tipove
  - usložnjavaju se konverzijska pravila
  - rezultat operacija može biti ovisan o implementaciji prevodioca (rezultat: neprenosivi programi)
- najčešće nije potrebno koristiti unsigned tip podatka. Na ovom predmetu će se koristiti u rijetkim prilikama, npr.
  - kada se ciljano želi povećati mogućnost pohrane (približno dvostruko) većih pozitivnih brojeva, na štetu mogućnosti pohrane negativnih brojeva
  - kada se obavljaju operacije nad bitovima, naročito u slučaju operacije posmaka bitova
    - operacije nad bitovima će biti objašnjene kasnije

Cjelobrojne konstante

- konstante se u memoriji računala pohranjuju na isti način kao i varijable
  - broj bajtova koji se koristi za pohranu i interpretacija konstante kao isključivo pozitivnog ili i negativnog broja ovisi o tipu konstante

Tip	Primjeri konstanti
signed short int	ne postoji konstanta tipa signed short, koristiti konstantu tipa int
signed int	2000000000, -2000000000
signed long int	2000000000**L**, -2000000000**L**
signed long long int	-5000000000**LL**, 5000000000**LL**
unsigned short int	ne postoji konstanta tipa unsigned short
unsigned int	2000000000**U**
unsigned long int	2000000000**UL**
unsigned long long int	5000000000**ULL**

  - umjesto sufiksa U, L ili LL može se koristiti u, l ili ll (mala slova U i L)

Cjelobrojne konstante

- cjelobrojne konstante se najčešće pišu u dekadskom brojevnom sustavu, ali u programu ih je moguće zapisati i u oktalnom ili heksadekadskom sustavu
  - oktalna konstanta započinje prefiksom 0 i sadrži znamenke 0-7
  - heksadekadska konstanta započinje prefiksom 0x ili 0X i sadrži znamenke 0-9 i A-F ili a-f

```
printf("%d", 17);          17
printf("%d", 017);         15
printf("%d", 019);         prevodilac dojavljuje pogrešku!
printf("%d", 0x17);        23
printf("%d", 0X1a);        26
printf("%d", 0x7FFFFFFF);  2147483647
printf("%d", 0xFFFFFFFF);  -1
```

Cjelobrojne konstante

- sufiksi U, L i LL mogu se dodati na konstante prikazane u bilo kojem brojevnom sustavu

```
017LL          signed long long konstanta, jednako 15LL
0x17L          signed long konstanta, jednako 23L
0xFFFFFFFFu    unsigned int konstanta, jednako 4294967295U
0xFFFFFFFF     signed int konstanta, jednako -1
```

Konverzijske specifikacije za printf i scanf

Tip	signed	unsigned
short int	%hd	%hu
int	%d	%u
long int	%ld	%lu
long long int	%lld	%llu

- **u** se može zamijeniti s **x**, **X**, **o** radi čitanja ili ispisa u heksadekadskom ili oktalnom obliku

```
printf("%d", 0xFFFFFFFF);              -1
printf("%u", 0xFFFFFFFF);              4294967295
printf("%lld", 0xFFFFFFFFFFFFFFFF);    -1
printf("%llu", 0xFFFFFFFFFFFFFFFF);    18446744073709551615
printf("%o", 0xFFFFFFFF);              37777777777
printf("%x", 0xFFFFFFFF);              ffffffff
printf("%X", 0xFFFFFFFF);              FFFFFFFF
printf("%llo", 0xFFFFFFFFFFFFFFFF);    1777777777777777777777
```

Primjer

```
#include <stdio.h>

int main(void) {
   unsigned int ubroj = 2000000000U;
   ubroj = ubroj * 2 / 2;
   printf("ubroj=%u\n", ubroj);

   signed int sbroj = 2000000000;
   sbroj = sbroj * 2 / 2;
   printf("sbroj=%d\n", sbroj);
   return 0;
}
```

```
ubroj=2000000000
```

```
sbroj=-147483648
```

- Objašnjenje:
  - ubroj * 2 = 4000000000, 4000000000 / 2 = 2000000000
  - sbroj * 2 = -294967296, -294967296 / 2 = -147483648

Konverzijske specifikacije - logičke pogreške

- Važno je koristiti odgovarajuće konverzijske specifikacije jer prevodilac neće prepoznati njihovu pogrešnu upotrebu
  - nastaju teško prepoznatljive logičke pogreške

```
printf("%lld", 10000000000LL);          10000000000
printf("%d", 10000000000LL);            1410065408

int a = 98304;
scanf("%hd", &a);                       za ulaz 10
printf("%d", a);                        65546
```

Cjelobrojni tipovi podataka

Tip podatka char

Tip podatka char

- cjelobrojni tip podatka koji se koristi za pohranu malih brojeva
- izgovor za tip podatka *char*: tʃaː ili (rjeđe) 'kær
  - u engleskom govornom području koriste se oba oblika
  - kolokvijalno ćemo koristiti naziv *karakter* ili *znak*
- standardom se od tipa podatka char zahtijeva mogućnost pohrane **cijelih brojeva** u prostoru za pohranu od najmanje jednog bajta
  - obično to upravo jest jedan bajt (npr. za gcc i arhitekturu x86_64)
- dopušteni rasponi
  - signed: [-128, 127]
  - unsigned: [0, 255]

Upotreba tipa podatka char za prikaz znakova

- Znakovi se ne mogu pohraniti u računalu!
  - moguće je pohraniti binarne brojeve koji predstavljaju unaprijed dogovorene kodove znakova, npr.

Znak	binarno	dekadski
A	01000001	65
B	01000010	66

  - u upotrebi su razne kodne stranice
    - ASCII: sadrži 128 različitih znakova i upravljačkih znakova kodiranih vrijednostima 0-127
    - ISO-8859: kodovi 0-127 kao u 7-bitnom kodu, dok se kodovi 128-255 koriste za razne dodatne znakove, ovisno o kodnoj stranici, npr.
      - ISO 8859-1: dodatni znakovi zapadnoeuropskih jezika
      - ISO 8859-2: dodatni znakovi istočnoeuropskih jezika
        - npr. š = 185_{10}, Č = 200_{10}
    - Na ovom predmetu koristit će se kodna stranica ASCII

ASCII - American Standard Code for Information Interchange

ISO - International Organization for Standardization

Unicode

- Noviji industrijski standard za kodiranje znakova
  - Koristi više bajtova za kodiranje znakova i omogućuje predstavljanje gotovo svih pisama korištenjem jedinstvenog skupa znakova
  - The Unicode Consortium: www.unicode.org

  - Primjer: kod za znak LA iz pisma Tagalog (Filipini) jest $170E_{16}$

170E

ASCII tablica kodova znakova (1)

Dec. broj	C konst.	Znak
0	'\0'	Nul znak (NULL)
1		početak zaglavlja (SOH)
2		početak teksta (STX)
3		kraj teksta (ETX)
4		kraj prijenosa (EOT)
5		kraj upita (ENQ)
6		Potvrda (ACK)
7	'\a'	Alarm (BEL)
8	'\b'	Backspace (BS)
9	'\t'	vodoravni tabulator (HT)
10	'\n'	sljedeći red/novi red (LF)
11	'\v'	okomiti tabulator (VT)
12	'\f'	nova stranica (FF)
13	'\r'	skok na početak reda (CR)
14		pomak van (SO)
15		pomak unutra (SI)

Dec. broj	Znak
16	znak prekida veze (DLE)
17	provjera uređaja 1 (DC1)
18	provjera uređaja 2 (DC2)
19	provjera uređaja 3 (DC3)
20	provjera uređaja 4 (DC4)
21	negativna potvrda (NAK)
22	sinkrono mirovanje (SYN)
23	kraj prijenosnog bloka (ETB)
24	otkaži (CAN)
25	kraj medija (EM)
26	Zamjena (SUB)
27	Escape (ESC)
28	razdjelnik datoteka (FS)
29	razdjelnik grupe (GS)
30	razdjelnik zapisa (RS)
31	razdjelnik jedinice (US)

ASCII tablica kodova znakova (2)

Dec. broj	Znak
32	razmak
33	!
34	"
35	#
36	$
37	%
38	&
39	'
40	(
41	)
42	*
43	+
44	,
45	-
46	.
47	/

Dec. broj	Znak
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
58	:
59	;
60	<
61	=
62	>
63	?
64	@

Dec. broj	Znak
65	A
66	B
67	C
68	D
69	E
70	F
71	G
72	H
73	I
74	J
75	K
76	L
77	M
78	N
79	O

Dec. broj	Znak
80	P
81	Q
82	R
83	S
84	T
85	U
86	V
87	w
88	X
89	Y
90	Z
91	[
92	\
93	]
94	^
95	_

ASCII tablica kodova znakova (3)

Dec. broj	Znak
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o

Dec. broj	Znak
112	p
113	q
114	r
115	s
116	t
117	u
118	v
119	w
120	x
121	y
122	z
123	{
124	|
125	}
126	~
127	DEL

Znakovi za upravljanje ulazno-izlaznim jedinicama računala (kontrolni znakovi, *non-printable*) nalaze se na pozicijama 0-31 i na poziciji 127 (DEL)

Znakovi koji se mogu tiskati (*printable*) nalaze se na pozicijama 32-126

Konverzijske specifikacije za printf i scanf

- u slučaju kada se podatak tipa char prikazuje kao broj

Tip	signed	unsigned
char	%hhd	%hhu

- **d** ili **u** se mogu zamijeniti s **x**, **X**, **o** radi ispisa u heksadekadskom ili oktalnom obliku

```
signed char a = -10;
unsigned char b = 255;
printf("%hhd %hhu", a, b);          -10 255
printf("%hho %hhx", a, b);          366 ff
```

- u većini primjena dopušteno je kod ispisa koristiti format %d jer se odgovarajuće konverzije char ⇔ int obavljaju automatski

```
char a = 65;
printf("%d", a);                    65
```

Konverzijske specifikacije za printf i scanf

- kada se podatak tipa char prikazuje ili čita kao znak, koristi se konverzijska specifikacija %c

```
char a, b, c, d;

a = 65;
printf("%hhd %c", a, a);              65 A

b = a + 3;
printf("%hhd %c", b, b);              68 D

scanf("%c", &c);                      za ulaz 7
printf("%hhd %c", c, c);              55 7

scanf("%c", &d);                      za ulaz E
printf("%hhd %c", d, d);              69 E
```

Konstante

- konstanta tipa char ne postoji. Koriste se konstante tipa int
  - prikladno je koristiti poseban oblik pisanja konstante tipa int
    - znak (koji se može ispisati) iz ASCII tablice napisan pod jednostrukim navodnicima, npr.

      `'B'`
    - ovako napisana konstanta tipa int zauzima 4 bajta i ima vrijednost 00000042_{16}, odnosno 66_{10}
    - sljedeća dva programska odsječka će dati isti rezultat:

```
char znak;
znak = 66;
```

```
char znak;
znak = 'B';
```

    - u oba slučaja sadržaj konstante veličine 4 bajta bit će upisan u sadržaj varijable znak veličine jednog bajta (pri čemu se odbacuju tri bajta sa značajnijim bitovima, koji ionako sadrži samo nule)

Primjeri cjelobrojnih konstanti

C program	Hex.	Dek.	U ASCII tablici odgovara kodu znaka:
`'A'`	`0x61`	65	veliko slovo A
`'0'`	`0x30`	48	znamenka nula

- znakove koji u sintaksi imaju posebno značenje ili se ne mogu jednostavno prikazati jednim znakom, prikazuju se pomoću tzv. "*escape sequence*"
  - niz od dva ili više znakova koji započinju znakom \ koji nagovještava "specijalno" značenje znakova koji slijede

C program	Hex.	Dek.	U ASCII tablici odgovara kodu znaka:
`'\a'`	`0x07`	7	alarm (*bell, beep*), aktivira zvuk na terminalu
`'\t'`	`0x09`	9	vodoravni tabulator
`'\n'`	`0x0A`	10	skok u novi red
`'\0'`	`0x00`	0	nul-znak, oznaka kraja niza
`'\\'`	`0x5C`	92	obrnuta kosa crta (*backslash*)
`'\''`	`0x27`	39	jednostruki navodnik
`'\"'`	`0x22`	34	dvostruki navodnik

Pojednostavljenje načina izražavanja

```
char c1;
c1 = 'E';
```

- Radi pojednostavljenja, koristit će se kolokvijalni izrazi:
  - *'E' je znakovna konstanta*
    - iako je poznato da se radi o cjelobrojnoj (int) konstanti vrijednosti 69_{10}
  - *varijabla c1 je znakovnog tipa*
    - iako je poznato da je cjelobrojnog tipa (ali očito će se koristiti za pohranu ASCII koda znaka)
  - *u varijablu c1 upisan je znak E*
    - iako je poznato da je u varijablu upisan ASCII kod znaka E, odnosno cijeli broj 69_{10}

Primjer

```
#include <stdio.h>

int main(void) {
   char x = 'A', y = x + 32, z = '\n';
   printf("%hhd %c\n", x, x);
   printf("%hhd %c\n", y, y);
   printf("%hhd %c\n", z, z);
   printf("%d %c\n", '0' + 2, '0' + 2);
   printf("%d %c\n", '0' + '2', '0' + '2');

   return 0;
}
```

```
65 A↵
97 a↵
10 ↵
↵
50 2↵
98 b↵
```

Primjer

```
#include <stdio.h>

int main(void) {
   char c = 'D';
   printf ("%c %d\n", c, c);
   printf ("%c\n", c + 32);
   printf ("%d\n", 'C' - 'A');

   return 0;
}
```

```
D 68↵
d↵
2↵
```

Znamenke 0 - 9 u ASCII tablici

- Treba obratiti pažnju na to da ASCII kodovi znamenki 0-9 ne odgovaraju numeričkim vrijednostima znamenki

```
char a = '1';
```

u varijabli je pohranjena numerička vrijednost 49

- ako se na temelju koda znamenke želi dobiti njezina numerička vrijednost, potrebno je od te vrijednosti oduzeti 48, jer vrijednost 48 predstavlja kod znaka '0'

```
char c;
scanf("%c", &c);                          za ulaz  7
printf("%c %hhd %d", c, c, c - 48);       7 55 7
```

ili

```
printf("%c %hhd %d", c, c, c - '0');      7 55 7
```

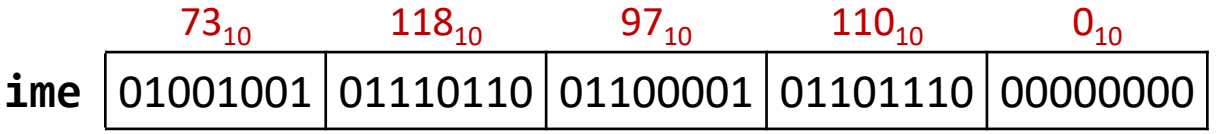
Digresija: nizovi znakova

Niz znakova - *string*

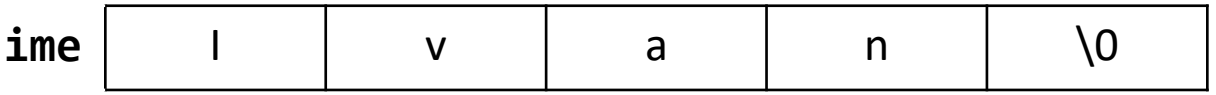
- U programskom jeziku C ne postoji osnovni tip podatka koji podržava rad s nizovima znakova
- za pohranu niza znakova koristi se jednodimenzijsko polje čiji su članovi tipa char, pri čemu se kraj niza obavezno označava članom polja koji sadrži nul-znak '\0' (kod iz ASCII tablice numeričke vrijednosti 0)

```
char ime[5];

ime[0] = 'I';
ime[1] = 'v';
ime[2] = 'a';
ime[3] = 'n';
ime[4] = '\0';



printf("%s", ime);
```

Ivan



- radi pojednostavljenja, na slikama će se umjesto kodova znakova prikazivati znakovi

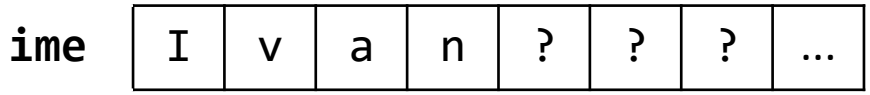


- kao konverzijsku specifikaciju koristiti %s
- kao vrijednost koju treba ispisati koristiti naziv polja u kojem je niz znakova pohranjen

Čemu služi terminator niza '\0' ?

- Pomoću znaka '\0' može se zaključiti gdje se nalazi kraj niza znakova. Npr.
  - funkcija printf prema konverzijskoj specifikaciji %s ispisuje znakove iz zadanog niza znakova, znak po znak, sve dok ne dođe do člana polja u kojem se nalazi vrijednost '\0'
  - ako kraj niza nije ispravno označen znakom '\0', funkcija će nastaviti ispisivati znakove čiji ASCII kodovi odgovaraju vrijednostima koji se nalaze u memoriji iza kraja niza

```
char ime[4];

ime[0] = 'I';
ime[1] = 'v';
ime[2] = 'a';
ime[3] = 'n';

printf("%s", ime);
```



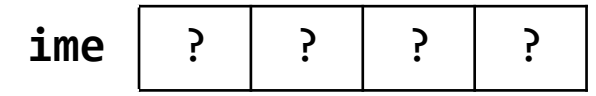
- ispis će se nastaviti sve dok se negdje u memoriji ne naiđe na bajt u kojem je upisana vrijednost nula

```
Ivan*)%&/!)=()Z)(B#DW=)$/")#*'@!"/...
```

Definicija niza znakova uz inicijalizaciju

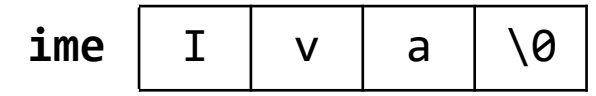
- Jednako kao kod polja ostalih tipova podataka:
  - bez inicijalizacije, članovi polja sadrže nedefinirane vrijednosti

```
char ime[4];
```



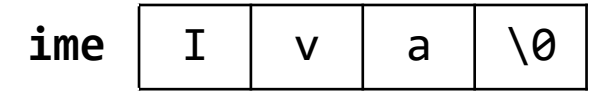
  - početne vrijednosti se mogu redom navesti u tzv. inicijalizatoru

```
char ime[4] = {'I', 'v', 'a', '\0'};
```



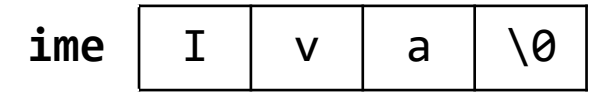
  - ili

```
char ime[] = {'I', 'v', 'a', '\0'};
```



  - ili

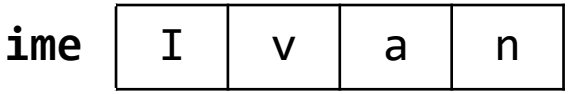
```
char ime[4] = {'I', 'v', 'a'};
```



Definicija niza znakova uz inicijalizaciju

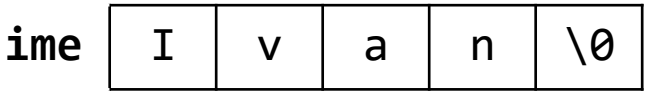
- Obratiti pažnju
  - ovakav način inicijalizacije jednodimenzijskog polja nije ispravan ako se njegov sadržaj namjerava koristiti kao niz znakova (string)

```
char ime[4] = {'I', 'v', 'a', 'n'};
```



  - ispravno:

```
char ime[4+1] = {'I', 'v', 'a', 'n'};
```



Jednostavniji način inicijalizacije niza znakova

- Umjesto:

```
char ime[4] = {'I', 'v', 'a', '\0'};
```

  - može se (i bolje je) koristiti drugačiji oblik inicijalizatora

```
char ime[4] = "Iva";
```

znak **\0** će biti dodan, programer **mora** osigurati prostor za barem jedan znak više!

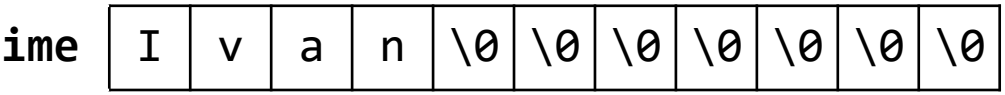
  - ili

```
char ime[] = "Iva";
```

potrebnu veličinu polja odredit će prevodilac, znak **\0** će biti dodan

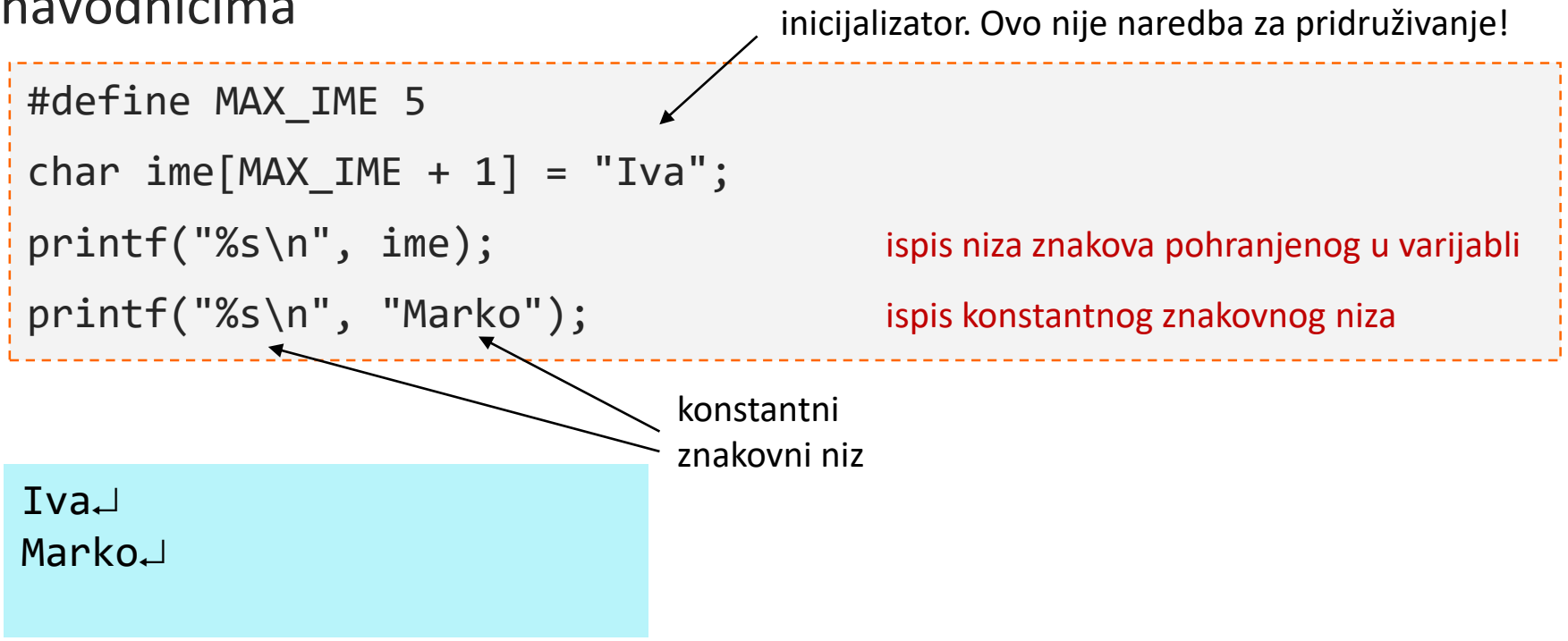
  - uobičajeno se nizovi znakova definiraju uz pomoć simboličkih konstanti. Pri tome, dobra je praksa koristiti notaciju "+ 1". Time se naglašava da nije zaboravljen prostor za znak kojim se terminira niz

```
#define MAX_IME 10
char ime[MAX_IME + 1] = "Ivan";
```



Konstantni znakovni niz

- U programu se konstantni znakovni niz označava dvostrukim navodnicima



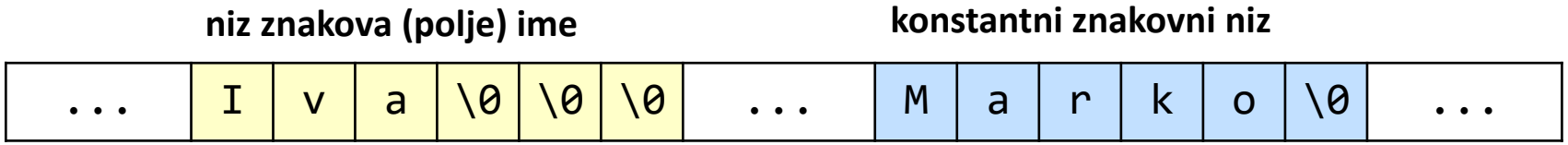

```
char ime[10 + 1];
ime = "Iva";
```

Nije dopušteno! Polje je ***non-modifiable*** *lvalue*!

Konstantni znakovni niz

```
#define MAX_IME 5
char ime[MAX_IME + 1] = "Iva";
printf("%s\n", ime);
printf("%s\n", "Marko");
```

- konstantni znakovni niz "Marko" u memoriji je pohranjen slično nizu znakova ime



- ali postoji bitna razlika: sadržaj niza znakova može se mijenjati

```
ime[1] = 'd';
printf("%s\n", ime);
```

```
Ida↵
```

Konstantni znakovni niz

- Konstantni znakovni niz se u programima ne smije lomiti kroz više redaka

```
printf("%s", "Fakultet
 u Unskoj 3");
```

prevodilac dojavljuje pogrešku

- ako je niz predugačak da bi se mogao pregledno napisati u jednom retku, treba koristiti jednostavno nadovezivanje

```
printf("%s", "Fakultet"
             " u Unskoj 3");
```

- konstantni znakovni niz napisan u dva ili više dijelova, u memoriji će biti pohranjen na isti način kao da je napisan u jednom dijelu

```
printf("%s", "Fakultet u Unskoj 3");
```

...	F	a	k	u	l	t	e	t		u		U	n	s	k	o	j		3	\0	...

Konverzijska specifikacija %s za printf

- Za ispis niza znakova (također i konstantnog znakovnog niza) koristi se konverzijska specifikacija %s

```
printf("Ime: %s\nPrezime: %s\nAdresa:\n%s\n",
       "Ana",
       "Novak",
       "Ilica 1\n10000 Zagreb\\Centar");
```

```
Ime: Ana↵
Prezime: Novak↵
Adresa:↵
Ilica 1↵
10000 Zagreb\Centar↵
```

Konverzijska specifikacija %s za scanf

- Konverzijska specifikacija %s omogućuje čitanje niza znakova do pojave prve praznine, oznake novog retka ili tabulatora

```
char ime[10 + 1], prezime[10 + 1];
scanf("%s %s", ime, prezime);
printf("%s, %s", prezime, ime);
```

```
Anamarija Horvat Novak↵
Horvat, Anamarija
```

- Čitanje konverzijskom specifikacijom %s na kraj niza ugrađuje '\0'
- Čitanje niza znakova pomoću funkcije scanf i konverzijske specifikacije %s je potencijalno opasno
  - što će se dogoditi ako korisnik upiše predugo ime ili prezime (u odnosu na duljinu niza znakova navedenu u definiciji)

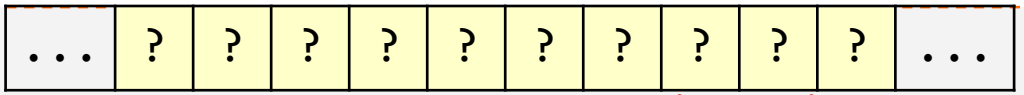
```
Anamarija Horvat-Novak↵
```

Učitavanje niza znakova na siguran način

- Ako je potrebno onemogućiti unos predugog niza znakova ili je potrebno omogućiti čitanje niza znakova koji sadrži praznine ili tabulatore, tada za čitanje niza znakova s tipkovnice umjesto funkcije scanf treba koristiti funkciju fgets
  - funkcija fgets će detaljnije biti objašnjena kasnije. Ovdje je naveden skraćeni opis, koji će biti dovoljan za njeno ispravno korištenje
  - funkcija fgets prima tri argumenta
    - ime polja (varijabla tipa niza znakova) `niz` u koje s tipkovnice treba učitati niz znakova
    - najveći dopušteni broj znakova n koje funkcija fgets smije upisati u to polje (za vrijeme dok učitava znakove s tipkovnice)
    - *tok podataka* iz kojeg se učitavaju znakovi. Za sada uvijek napisati `stdin` (skraćenica dolazi od *standard input* - vrlo pojednostavljeno: tipkovnica). Detaljnije objašnjenje slijedi u poglavlju o datotekama

Učitavanje niza znakova

```
char tekst[10];
fgets(tekst, 10, stdin);
```



- u zadani niz učitava znakove s tipkovnice sve dok ne pročita oznaku novog retka ili učita n-1 znakova (u ovom primjeru 9). Iza zadnjeg učitanog znaka u niz dodaje znak '\0'
  - zašto se ovoj funkciji mora zadati najveći dopušteni broj znakova?

↵	... \n \0 ? ? ? ? ? ? ? ? ...
Pas↵	... P a s \n \0 ? ? ? ? ? ...
Pet pasa↵	... P e t ␣ p a s a \n \0 ...
Dva macka↵	... D v a ␣ m a c k a \0 ...
Jedanaest pasa↵	... J e d a n a e s t \0 ...

Primjer

- Programski zadatak
  - s tipkovnice učitati niz znakova iz jednog retka. Niz znakova, uključujući oznaku novog retka (ako bude učitana), ne smije biti dulji od 10 znakova.
  - ako učitani niz sadrži znak za prelaz u novi redak, izbaciti ga iz niza. Sva mala slova u nizu pretvoriti u velika. Ispisati novi sadržaj niza i odmah iza kraja niza uskličnik.

```
Upisite niz znakova > Kvaka 22↵
KVAKA 22!
```

```
Upisite niz znakova > Put u svemir↵
PUT U SVEM!
```

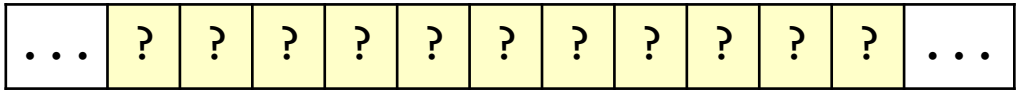
```
Upisite niz znakova > ↵
!
```

Rješenje

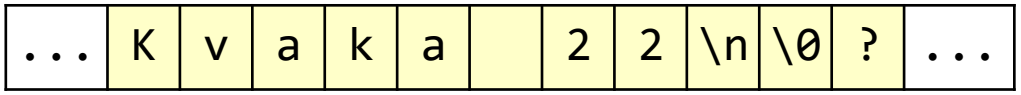
```
#include <stdio.h>
#define MAX_NIZ 10

int main(void) {
   char niz[MAX_NIZ + 1];
   int i = 0;
   printf("Upisite niz znakova > ");
   fgets(niz, MAX_NIZ + 1, stdin);
   while (niz[i] != '\0') {
      if (niz[i] >= 'a' && niz[i] <= 'z') {
         niz[i] = niz[i] - ('a' - 'A');
      } else if (niz[i] == '\n') {
         niz[i] = '\0';
      }
      i = i + 1;
   }
   printf("%s!", niz);
   return 0;
}
```

Objašnjenje

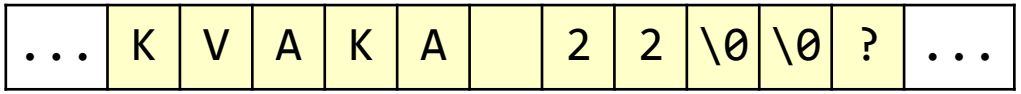
```
char niz[10];
```



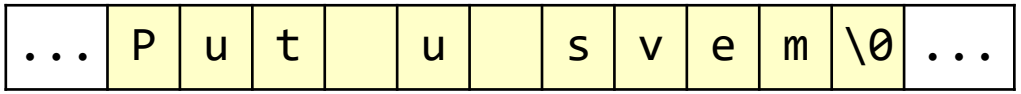

```
fgets(tekst, 10, stdin);
```



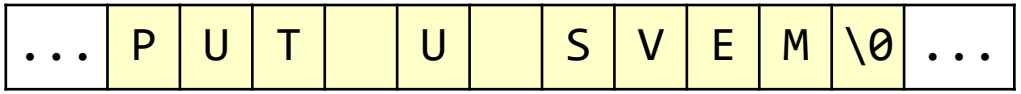
nakon petlje while



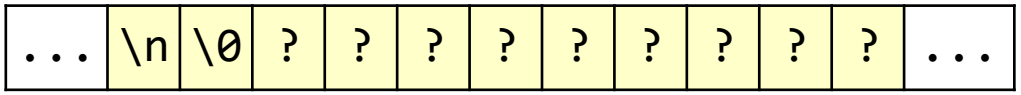

```
fgets(tekst, 10, stdin);
```



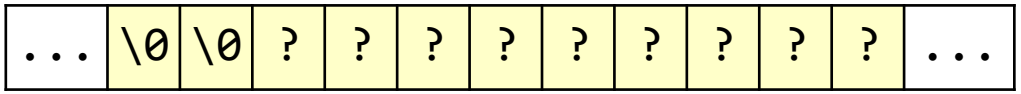
nakon petlje while




```
fgets(tekst, 10, stdin);
```



nakon petlje while



Cjelobrojni tipovi podataka

Tip podatka _Bool

Tip podatka _Bool

- cjelobrojni tip podatka koji se koristi za pohranu logičke vrijednosti *istina* ili *laž*
  - interno se vrijednost u varijablu tipa _Bool pohranjuje kao cijeli broj
    - logička vrijednost *istina* predstavljena je cjelobrojnom vrijednošću 1, a logička vrijednost *laž* cjelobrojnom vrijednošću 0
    - npr. za gcc i arhitekturu x86_64, koristi se prostor veličine 1 bajt
  - po čemu se tip _Bool razlikuje od ostalih cjelobrojnih tipova?
    - ako se varijabli tipa _Bool pridruži bilo koja numerička vrijednost različita od nule, varijabla će poprimiti vrijednost 1

```
_Bool padaKisa;
padaKisa = 37;
printf("%hhd", padaKisa);
```

```
1
```

Logičke vrijednosti u programskom jeziku C

- svaki podatak, bilo kojeg tipa, u programskom jeziku C može se (ne znači da je poželjno) koristiti kao logička vrijednost. To znači da svaka vrijednost koja je numerički
  - jednaka nuli odgovara logičkoj vrijednosti *laž*
  - različita od nule odgovara logičkoj vrijednosti *istina*
- sljedeći primjer nije smjernica kako treba pisati programe (upravo suprotno) nego ilustracija prethodno navedenog

ispravno, ali nepotrebno nejasno

```
float a, b;
scanf("%f %f", &a, &b);
if (a + b) {
   printf("Zbroj nije nula");
}
```

ispravno i jasno

```
float a, b;
scanf("%f %f", &a, &b);
if (a + b != 0.f) {
   printf("Zbroj nije nula");
}
```

Rezultat logičkih izraza

- u programskom jeziku C rezultat logičkih izraza jest uvijek cjelobrojna vrijednost 0 (ako je rezultat *laž*) ili cjelobrojna vrijednost 1 (ako je rezultat *istina*). Rezultat po tipu odgovara tipu podatka int

```
float x = 5.0f;
printf("%d\n", x > 10.f || x < 0.f);
printf("%d", x != 20.f);
```

```
0
1
```

- to što je rezultat logičkog izraza cijeli broj 0 ili 1, nije opravdanje za pisanje ovakvog koda:

VRLO LOŠE!

```
if (x != 20.0 == 1)
```

Rezultat logičkih izraza

- rezultat logičkog izraza može se pohraniti u varijablu tipa _Bool, a potom koristiti u daljnjim logičkim izrazima
- Sljedeći primjer je ilustracija, a ne smjernica kako treba pisati

```
float x;
_Bool x_je_veci_od_10, x_je_manji_od_20;

scanf("%f", &x);
x_je_veci_od_10 = x > 10.f;
x_je_manji_od_20 = x < 20.f;

if (x_je_veci_od_10 && x_je_manji_od_20)
   printf("x je iz intervala <10, 20>");
```

VRLO LOŠE!

```
if (x_je_veci_od_10 == 1 && x_je_manji_od_20 == 1)

if ((x_je_veci_od_10 && x_je_manji_od_20) == 1)
```

Primjer

- unapređenje primjera s prim brojevima

```
...
int i, n, djeljiv = 0;
_Bool djeljiv = 0;                          // hipoteza: nije djeljiv
printf("Upisite prirodni broj > ");
scanf("%d", &n);
i = 2;
while (i <= n - 1 && djeljiv == 0 ! djeljiv) {
   if (n % i == 0) {
      djeljiv = 1;                          // hipoteza je bila pogresna
   }
   i = i + 1;
}
if (djeljiv == 1 djeljiv || n == 1)    // jer broj 1 je specijalan slucaj
   printf("%d nije prim broj\n", n);
else
   ...
```

Zamjenski *naziv* za tip *_Bool*, konstante *false* i *true*

- u `<stdbool.h>` se nalaze:
  - *makro* definicija *bool*
    - omogućuje da se umjesto naziva tipa _Bool koristi zamjenski naziv bool
    - naziv *bool* prikladniji je od *_Bool* jer je više u duhu jezika C: ostali ugrađeni tipovi podataka, npr. tipovi int, float, itd. nemaju u svom nazivu znak _ i ne sadrže velika slova
  - *makro* definicije *true* i *false*
    - omogućuju korištenje simboličkih konstanti *true* i *false*, umjesto cjelobrojnih konstanti 0 i 1
    - povećava se jasnoća programa

Makro definicije će biti detaljno objašnjene u kasnijim predavanjima. Za sada je dovoljno znati da se nakon uključivanja datoteke <stdbool.h>, u programu mogu koristiti nazivi *bool*, *true* i *false* kao zamjena za *_Bool*, *1* i *0*.

Primjer

- unapređenje primjera s prim brojevima

```
#include <stdbool.h>
...
int i, n;
_Bool djeljiv = 0;
bool djeljiv = false;                    // hipoteza: nije djeljiv

printf("Upisite prirodni broj > ");
scanf("%d", &n);
i = 2;
while (i <= n - 1 && djeljiv == 0 ! djeljiv) {
   if (n % i == 0) {
      djeljiv = 1 true;                  // hipoteza je bila pogresna
   }
...
```

Realni tipovi podataka

Tip podatka float

Tip podatka float

- Primjer definicije varijable

```
float temperatura;
```

- Primjeri konstanti tipa float

```
2f                  = 2.0
-2.34F              = -2.34
-1.34e5F            = -1.34 · 10^5
9.1093E-31f         = 9.1093 · 10^-31
```

- Na koji način su u računalu pohranjene vrijednosti tipa float?

Binarni razlomci

- racionalni broj **q** je *decimalni* razlomak ako se može zapisati u obliku

$$\mathbf{q} = \pm (c_n \cdot 10^n + c_{n-1} \cdot 10^{n-1} + \ldots c_1 \cdot 10^1 + c_0 \cdot 10^0 + r_1 \cdot 10^{-1} + r_2 \cdot 10^{-2} + \ldots + r_m \cdot 10^{-m})$$

$$c_i \in \{0, 1, 2, \ldots, 9\}, i = 0, \ldots, n$$

$$r_j \in \{0, 1, 2, \ldots, 9\}, j = 1, \ldots, m$$

$$13.75_{10} = 1 \cdot 10^1 + 3 \cdot 10^0 + 7 \cdot 10^{-1} + 5 \cdot 10^{-2}$$

- slično, *binarni* razlomak **q** može se zapisati u obliku

$$\mathbf{q} = \pm (c_n \cdot 2^n + c_{n-1} \cdot 2^{n-1} + \ldots c_1 \cdot 2^1 + c_0 \cdot 2^0 + r_1 \cdot 2^{-1} + r_2 \cdot 2^{-2} + \ldots + r_m \cdot 2^{-m})$$

$$c_i \in \{0, 1\}, i = 0, \ldots, n$$

$$r_j \in \{0, 1\}, j = 1, \ldots, m$$

$$1101.11_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2}$$

Pretvaranje decimalnog u binarni razlomak

- cijeli dio (dio ispred decimalne točke) pretvara se u cijeli dio (dio ispred binarne točke) binarnog razlomka uzastopnim dijeljenjem brojem 2
- razlomljeni dio (dio iza decimalne točke) pretvara se u razlomljeni dio binarnog razlomka uzastopnim množenjem brojem 2. Prekida se kad se za razlomljeni dio dobije točno nula
  - $9.625_{10} = ?_2$

    $9_{10} = 1001_2$

    $0.625 \cdot 2 = \mathbf{1}.250$

    $0.250 \cdot 2 = \mathbf{0}.5$

    $0.5 \cdot 2 = \mathbf{1}.0$ ← kraj

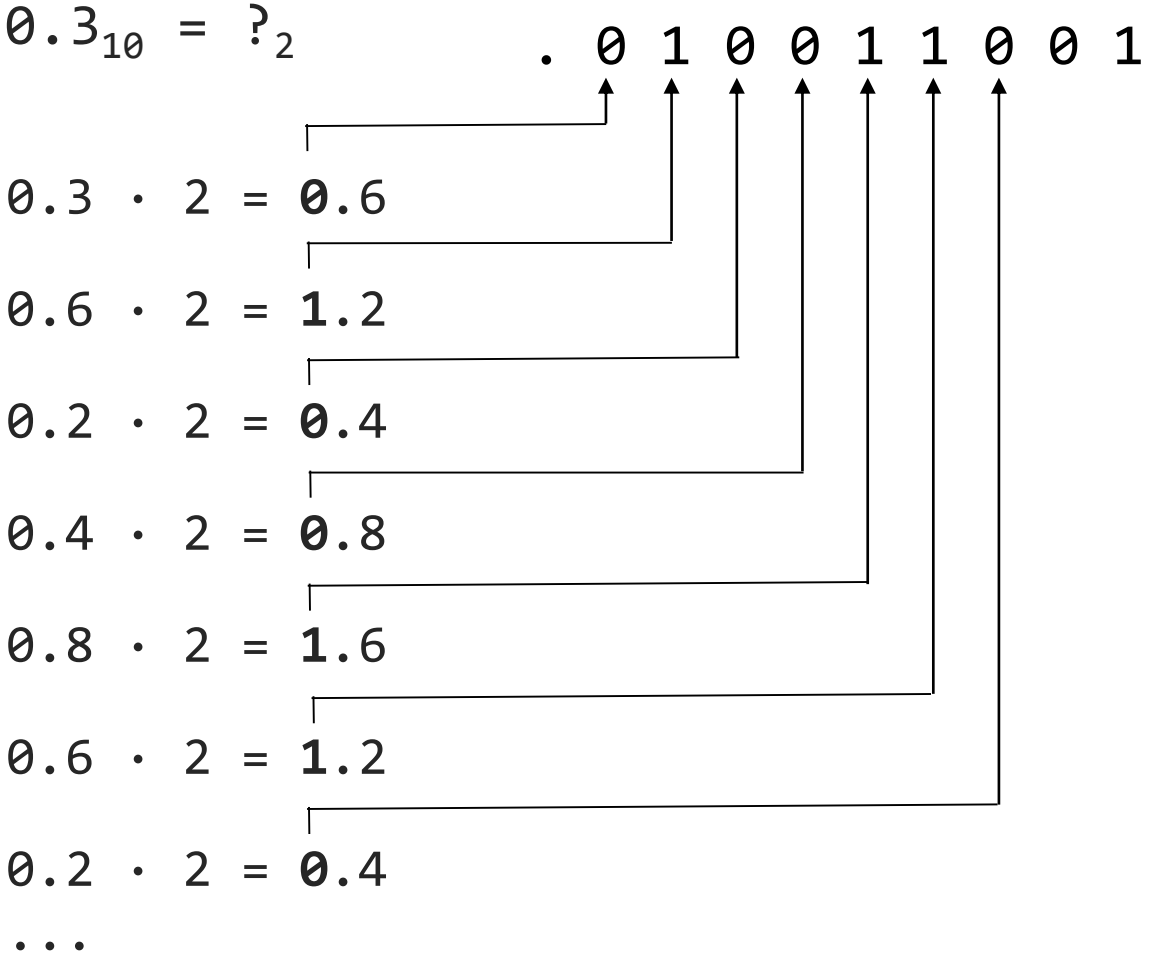
    . 1 0 1
  - $9.625_{10} = 1001.101_2$

Pretvaranje decimalnog u binarni razlomak

- ako nazivnik korespondentnog racionalnog broja sadrži faktor koji nije potencija broja 2, tada se decimalni razlomak neće moći prikazati kao binarni broj s konačnim brojem znamenaka. Npr.
  - konačnim brojem binarnih znamenaka mogu se prikazati brojevi
    - 1/2, 1/4, 1/8, 1/16, ..., 3/2, 3/4, 3/8, ..., 5/2, 5/4, 5/8, ...
  - konačnim brojem binarnih znamenaka ne mogu se prikazati brojevi
    - 1/3, 1/5, 1/6, 1/7, 1/9, 1/10, ..., 2/3, 2/5, 2/7, ...
- također očito: ako realni broj nije decimalni razlomak (ne može se prikazati s konačnim brojem znamenaka iza decimalne točke) tada se neće moći prikazati niti kao binarni broj s konačnim brojem znamenaka

Pretvaranje decimalnog u binarni razlomak

- $3/10 = 0.3_{10} = ?_2$



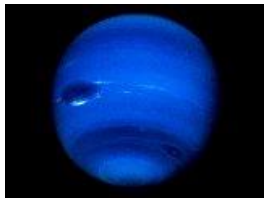
dok se ne dosegne zadovoljavajuća ili moguća preciznost

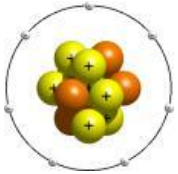
Množenje binarnog broja s potencijama broja 2

- binarni broj se množi s potencijama baze 2 tako da se binarna točka pomakne odgovarajući broj mjesta desno ili lijevo, ovisno o tome je li predznak potencije pozitivan ili negativan
- Primjer:
  - $111.000 \cdot 2^{2} = 11100.0$
  - $0001.101 \cdot 2^{-3} = 0.001101$

Prikaz vrlo velikih i vrlo malih brojeva

- Kako inženjeri i znanstvenici prikazuju vrlo velike i vrlo male brojeve?
  - Kolika je prosječna udaljenost Neptuna i Sunca?
    - 450393**0000000** m
  - Koliko iznosi masa elektrona?
    - 0.**000000000000000000000000000000**910938188 kg
  - previše znamenaka (nula) troši se samo za prikaz magnitude broja





- zato se koristi znanstvena notacija: decimalni broj s jednom znamenkom ispred decimalne točke (zareza), pomnožen odgovarajućom potencijom broja 10 (jer je baza brojanja = 10)
  - $4.50393 \cdot 10^{12}$
  - $9.10938188 \cdot 10^{-31}$

mantisa

eksponent

Prikaz vrlo velikih i vrlo malih binarnih brojeva

- Slično, binarni razlomak se može prikazati kao binarni broj s jednom binarnom znamenkom ispred binarne točke, pomnožen odgovarajućom potencijom broja 2 (jer je baza brojanja = 2)
- Za broj u takvom obliku kaže se da je normaliziran. Npr.

$$101.11 = 1.0111 \cdot 2^{2}$$

$$0.0000000000010011 = 1.0011 \cdot 2^{-14}$$

binarna mantisa

binarni eksponent

- Normalizacija broja omogućava prikaz vrlo velikih i vrlo malih binarnih brojeva, uvijek u istoj formi, bez korištenja velikog broja nula

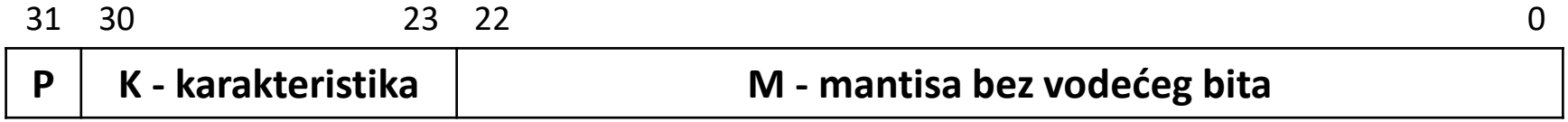
Prikaz realnih brojeva u računalu

- kako se podatak tipa float pohranjuje u registru računala?
- realni brojevi se u računalu pohranjuju u normaliziranom binarnom obliku
  - naravno, samo realni brojevi koji se mogu prikazati kao binarni razlomci s prihvatljivim brojem binarnih znamenaka
- Kako točno normalizirani binarni broj pohraniti u registar računala?
  - Standardom IEEE 754 propisan je način pohrane realnih brojeva
    - IEEE 754 standardna preciznost - *single precision*
    - IEEE 754 dvostruka preciznost - *double precision*

IEEE - Institute of Electrical and Electronics Engineers

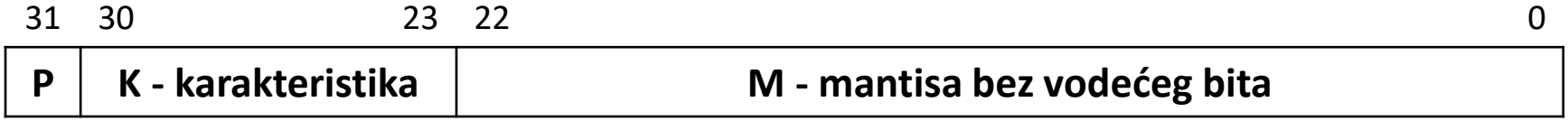
Realni brojevi standardne preciznosti

- IEEE 754 - *single precision*
  - najčešće se koristi za prikaz vrijednosti tipa float
  - binarni razlomak, u normaliziranom obliku, pohranjuje se u ukupno 32 bita (4 bajta) u sljedećem obliku:



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

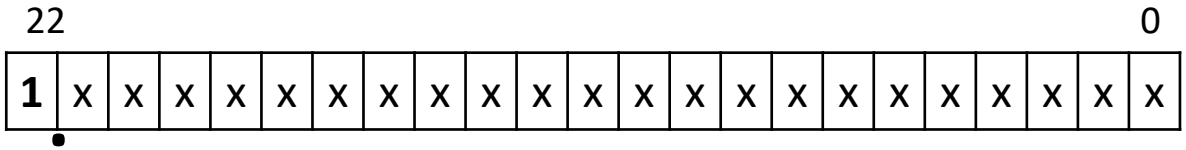
Realni brojevi standardne preciznosti



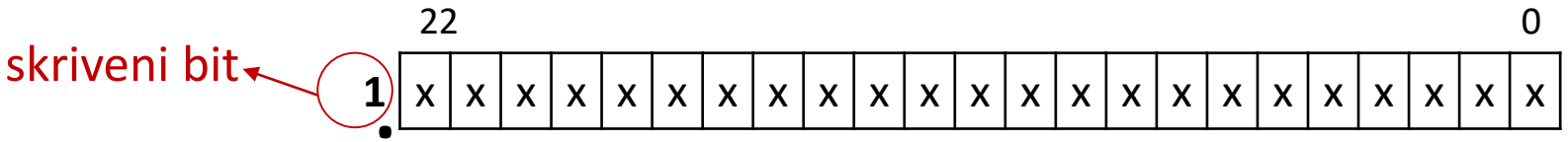
- predznak P: na mjestu najznačajnijeg bita upisuje se
  - 1 ako se radi o negativnom broju
  - 0 ako se radi o pozitivnom broju
- karakteristika K:
  - raspon binarnog eksponenta BE $\in$ [-126, 127]
    - radi izbjegavanja dvojnog komplementa za prikaz negativnih eksponenata, umjesto BE pohranjuje se karakteristika K
    - raspon karakteristike: K $\in$ [0, 255], K = BE + 127 $\Rightarrow$ BE $\in$ [-126, 127]
    - K = 0 i K = 255 se koriste za posebne slučajeve (objašnjeno kasnije)
- mantisa M:
  - ne pohranjuje se vodeći bit mantise - jedinica ispred binarne točke

Prvi bit mantise je implicitno određen

- ako se u 23 bita mora pohraniti *cijela* mantisa
  - iza binarne točke mogu se pohraniti najviše **22** binarne znamenke



  - uočiti: jedina znamenka koja se u normaliziranom broju može pojaviti ispred binarne točke je 1 (osim za prikaz broja nula). Jedan bit se troši nepotrebno jer je njegova vrijednost implicitno poznata
- stoga, vodeći bit (*leading bit*, *hidden bit*) se neće pohranjivati



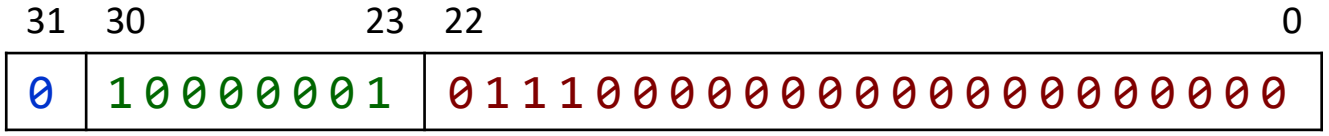
  - iza binarne točke mogu se pohraniti najviše **23** binarne znamenke
  - povećava se preciznost prikaza broja

Primjer

- Odrediti sadržaj registra u kojeg je pohranjen broj 5.75 prema standardu IEEE 754, standardnoj preciznosti. Sadržaj registra izraziti u binarnom i heksadekadskom brojevnom sustavu
  1. Odrediti bit za predznak: broj je pozitivan, stoga je P = 0
  2. Na temelju decimalnog razlomka izračunati binarni razlomak i normalizirati broj

     $5.75_{10} = 101.11_2 = 1.0111_2 \cdot 2^2$
  3. Izračunati karakteristiku i izraziti je u binarnom obliku, u 8 bitova

     $K = 2_{10} + 127_{10} = 129_{10} = 1000\ 0001_2$
  4. U registar prepisati bit za predznak, karakteristiku i mantisu <u>bez vodećeg bita</u>



$40\ B8\ 00\ 00_{16}$

Primjer

- 2.0

  $P = 0$

  $10_2 \cdot 2^0 = 1.0_2 \cdot 2^1$

  $BE = 1,\ K = 1 + 127 = 128_{10} = 10000000_2$

  $M = 1.000\ 0000\ \ldots\ 0000_2$

  $0100\ 0000\ 0000\ 0000\ \ldots\ 0000_2 = 4000\ 0000_{16}$

- - 2.0

  $P = 1$

  sve ostalo je jednako kao za 2.0

  $1100\ 0000\ 0000\ 0000\ \ldots\ 0000_2 = C000\ 0000_{16}$

Primjer

- 4.0

  $P = 0$

  $100_2 \cdot 2^0 = 1.0_2 \cdot 2^2$

  $BE = 2,\ K = 2 + 127 = 129_{10} = 10000001_2$

  $M = 1.000\ 0000\ \ldots\ 0000_2$

  $0100\ 0000\ 1000\ 0000\ \ldots\ 0000_2 = 4080\ 0000_{16}$

- 6.0

  $P = 0$

  $110_2 \cdot 2^0 = 1.10_2 \cdot 2^2$

  $BE = 2,\ K = 2 + 127 = 129_{10} = 10000001_2$

  $M = 1.100\ 0000\ \ldots\ 0000_2$

  $0100\ 0000\ 1100\ 0000\ \ldots\ 0000_2 = 40C0\ 0000_{16}$

Primjer

- 1.0

  P = 0

  $1_2 \cdot 2^0 = 1.0_2 \cdot 2^0$

  BE = 0, K = 0 + 127 = 127_{10} = 01111111_2

  M = $1.000\ 0000\ ...\ 0000_2$

  $0011\ 1111\ 1000\ 0000\ ...\ 0000_2 = 3F80\ 0000_{16}$

- 0.75

  P = 0

  $0.11_2 \cdot 2^0 = 1.1_2 \cdot 2^{-1}$

  BE = -1, K = -1 + 127 = 126_{10} = 01111110_2

  M = $1.100\ 0000\ ...\ 0000_2$

  $0011\ 1111\ 0100\ 0000\ ...\ 0000_2 = 3F40\ 0000_{16}$

Kalkulator za vježbanje

- Internet stranica na kojoj se nalazi dobar kalkulator za uvježbavanje zadataka s prikazivanjem realnih brojeva

  http://babbage.cs.qc.cuny.edu/IEEE-754/

Posebni slučajevi: prikaz realnog broja 0

- vodeća znamenka (bit) normaliziranog broja u binarnom obliku je uvijek 1, osim u slučaju realnog broja 0. Zato se u IEEE 754 standardu koristi posebno pravilo:
  - 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)
  - s obzirom da se pri tome bit za predznak može postaviti na 0 ili 1, moguće je prikazati "pozitivnu" i "negativnu" nulu, tj. +0 i -0
    - $00\ 00\ 00\ 00\ 00_{16} \rightarrow +0$
    - $80\ 00\ 00\ 00\ 00_{16} \rightarrow -0$
  - u relacijskim izrazima se te dvije vrijednosti smatraju jednakim

```
float x = 0.f, y = -0.f;
printf("x=%f, y=%f\n", x, y);
if (x == y) printf("x je jednak y");
```

```
x=0.000000, y=-0.000000↵
x je jednak y
```

Posebni slučajevi: denormalizirani broj

- kada je K = 0 i postoje bitovi mantise koji nisu 0, radi se o "denormaliziranom" broju
  - vodeći bit implicitno ima vrijednost 0
  - vrijednost binarnog eksponenta (BE) fiksirana je na -126 (ne koristi se izraz BE = K - 127)
  - omogućuje prikaz još manjih brojeva, ali uz smanjenu preciznost
    - stoga, izbjegavati – koristiti precizniji tip

0000 0000 0110 0000 0000 0000 0000 0000

$\rightarrow 0.11 \cdot 2^{-126}$

0000 0000 0000 0000 0000 0000 0000 0110

$\rightarrow 0.000\ 0000\ 0000\ 0000\ 0000\ 0110 \cdot 2^{-126}$

$\rightarrow 0.11 \cdot 2^{-146}$

Posebni slučajevi: prikaz +∞ i −∞

- kada je K = 255 i svi bitovi mantise su postavljeni na 0, radi se o prikazu +∞ ili −∞, ovisno o bitu za predznak

  `0111 1111 1000 0000 0000 0000 0000 0000` → +∞

  `1111 1111 1000 0000 0000 0000 0000 0000` → -∞

  - vrijednosti se mogu dobiti npr.
    - dijeljenjem s nulom u realnoj domeni
    - prekoračenjem najvećeg broja koji se može prikazati

```
float x = 1.f / 0.f;
float y = -1.f / 0.f;
float z = 3.e38f * 2.f;

printf("%f %f %f", x, y, z);
```

```
inf -inf inf
```

Posebni slučajevi: prikaz "ne-broja" (NaN)

- Ako je K=255 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
  - bit za predznak nema značenje
  - *NaN* je posljedica obavljanja operacije čiji je rezultat nedefiniran ili se prilikom obavljanja operacije dogodila pogreška, npr.

```
float x = 1.f / 0.f + -1.f / 0.f;
float y = 0.f / 0.f;
float z = -1.f / 0.f * 0.f;
float w = x * 0.f;

printf("%f %f %f %f\n", x, y, z, w);
```

```
nan nan nan nan
```

x → 1111 1111 1100 0000 0000 0000 0000 0000

Raspon realnih brojeva - standardna preciznost

- Najmanji normalizirani pozitivni broj koji se može prikazati je:

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

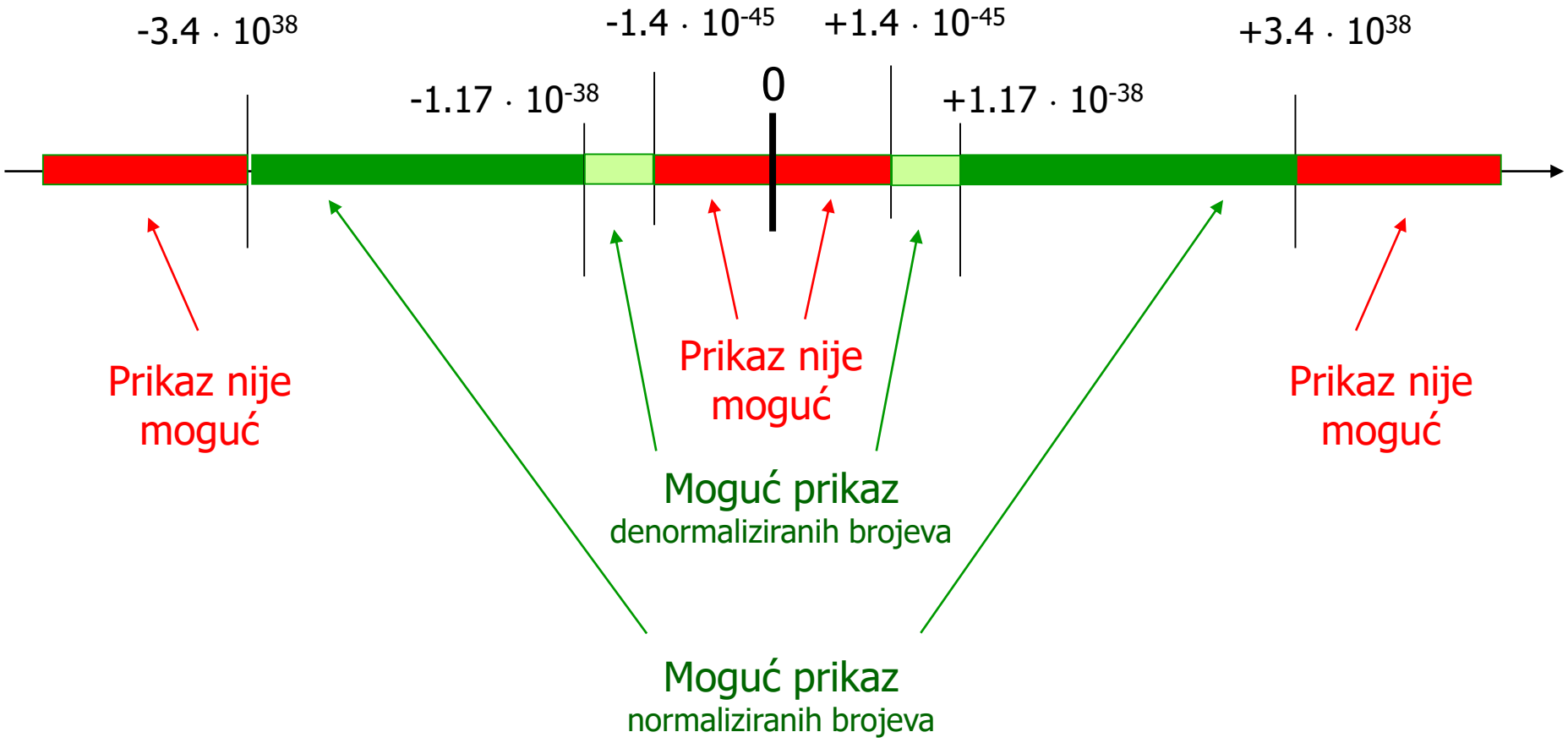
- Najmanji denormalizirani pozitivni broj koji se može prikazati je:

$$0.000\ 0000\ 0000\ 0000\ 0000\ 0001_2 \cdot 2^{-126}$$
$$= 1 \cdot 2^{-149} \approx 1.4 \cdot 10^{-45}$$

- 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;
```

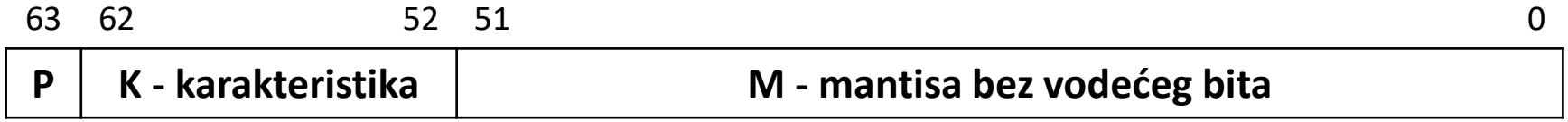
- primjeri konstanti tipa double
  - jednako kao konstante tipa float, ali bez znaka f ili F na kraju

```
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 koji se koriste za karakteristiku i mantisu

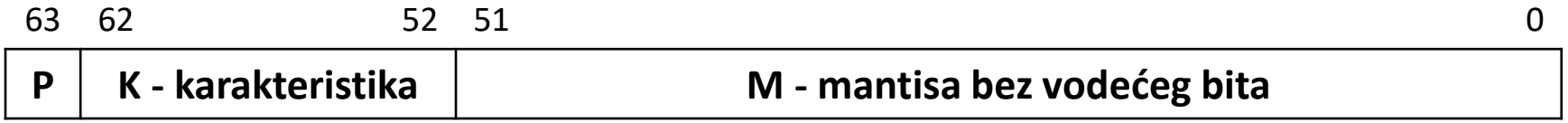
Realni brojevi dvostruke preciznosti

- 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:

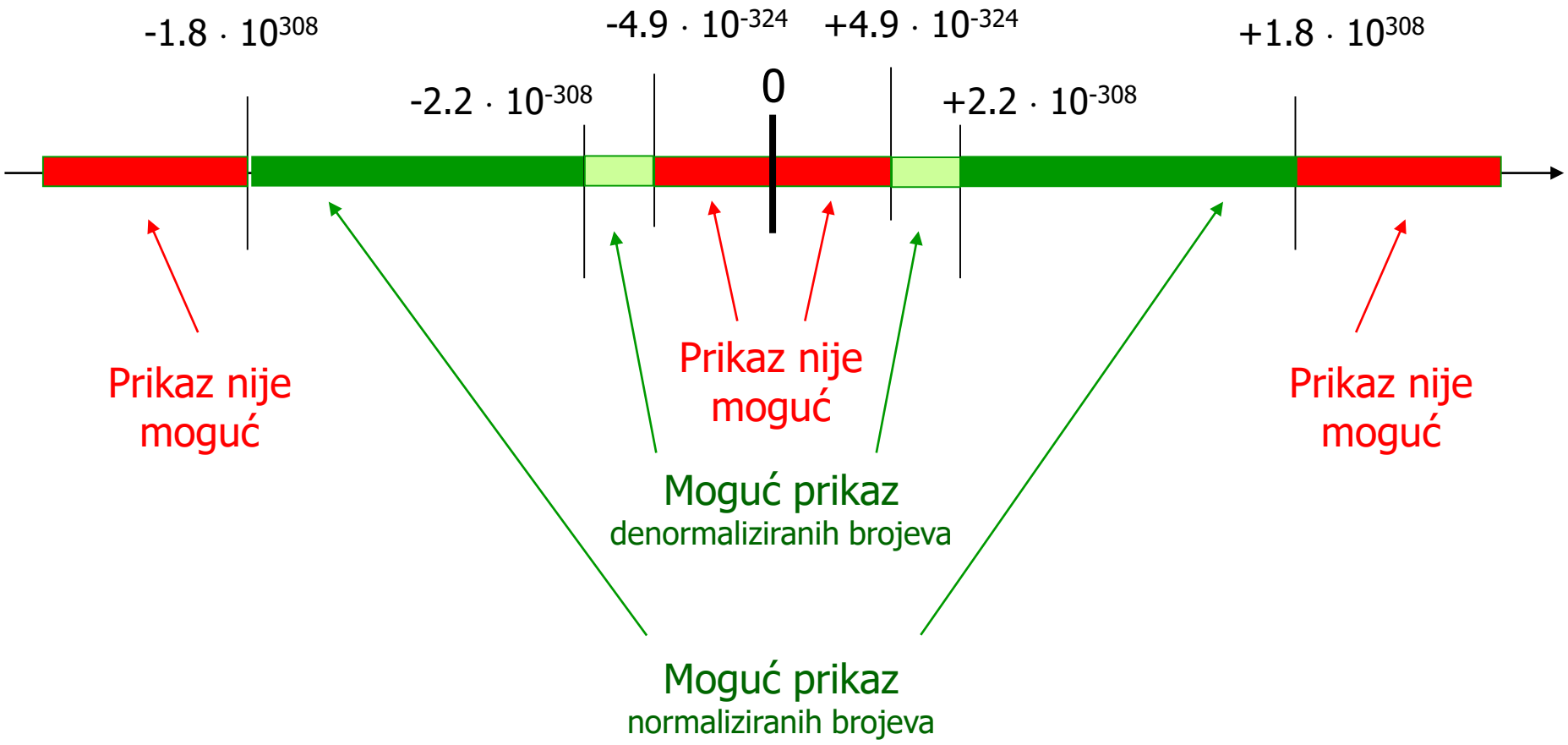
    $0.000...0001_2 \cdot 2^{-1022}$

    $= 1 \cdot 2^{-1074} \approx 4.9 \cdot 10^{-324}$
  - 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

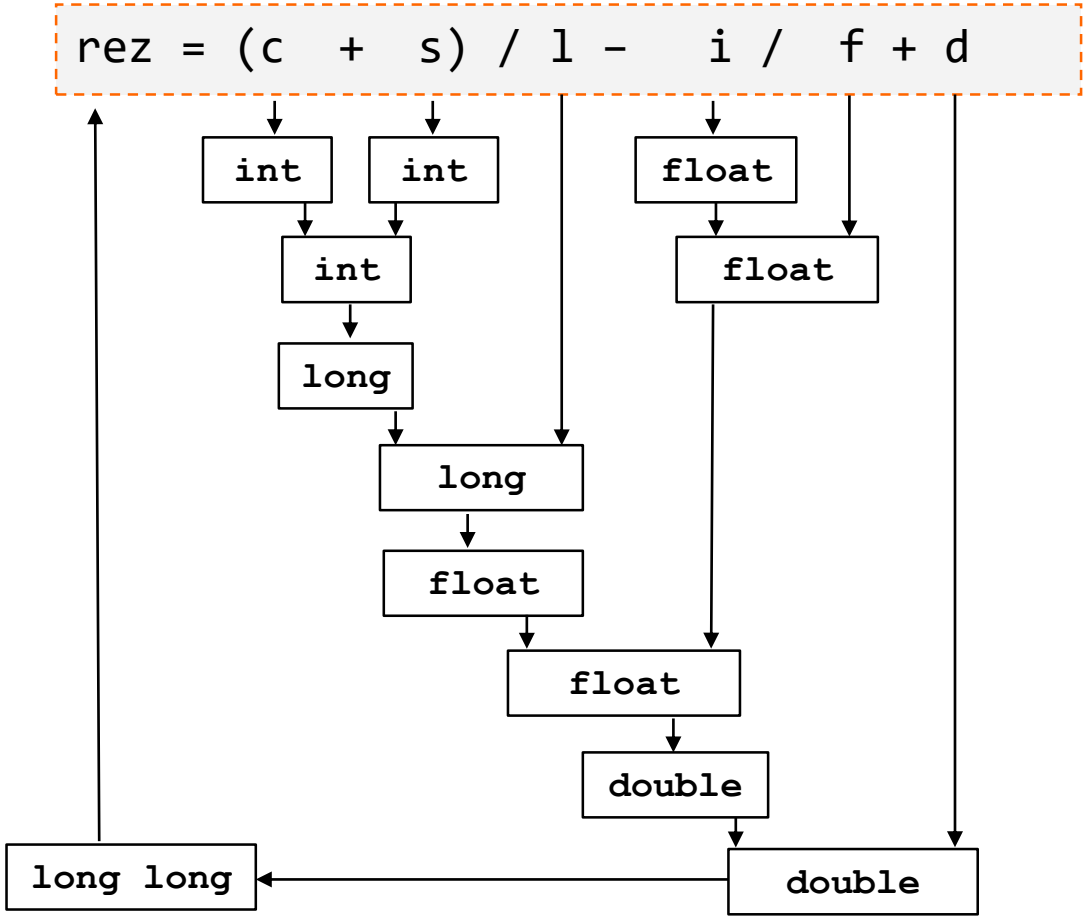
- Kod operacije pridruživanja, podatak s desne strane operatora pridruživanja pretvara se u tip podatka lijeve strane izraza
  - Primjer:

```
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)`