Programiranje 1 — predavanja

Marko Horvat

(prema slajdovima prof. Singera)

akademska godina 2023/24.

Uvodne informacije

Predavanja: Marko Horvat, Matej Mihelčić

Vježbe: Karmen Grizelj, Ivan Novak, Bruno Predojević

Demonstratori (privremeni popis): Karlo Ćoso, Nera Majtanić, Vice Perica, Ivan Premuš, Denis Ratković Rajhvajn, Lovro Razum, Petar Sruk, Hrvoje Žlepalo

Moji podaci:

- Marko Horvat
  - `mhorvat@math.hr` (službeni mejl)
  - `artakserkso@gmail.com` (privatni mejl)
- Ured: A306
- Konzultacije: srijedom, 10-12 (uz prethodnu najavu)

Vaš mail (na student-u) je oblika `mhorvat@student.math.hr`
Alias: `marko.horvat@student.math.hr`
Nastojte koristiti tu adresu za službenu komunikaciju.

Uvodne informacije

Kolegij se drži na PDM (1. godina) i PDMN (2. godina)

Obavezni računarski kolegiji na PDM (po semestrima):
$\boxed{\text{Prog1}} \rightarrow$ Prog2 $\rightarrow$ SPA $\rightarrow$ RP1

Cilj kolegija:

- oblikovanje, implementacija i analiza osnovnih algoritama
- usvajanje algoritamskog načina razmišljanja

Sadržaj kolegija:

- Uvod u algoritme, građa i principi rada računala
- Matematičke osnove računarstva (baze, logika, formalni jezici)
- Prikaz osnovnih podataka u računalu i operacije s njima
- **Programski jezik C**

Uvodne informacije

/Proći kroz Pravila ocjenjivanja/

Dostupni računalni praktikumi (kad u njima nema nastave):
Pr1 (podrum), Pr2 (prizemlje), Pr3,Pr4 (1. kat)
(dežuraju demonstratori, pitajte ih za pomoć ako treba)

Svi imate AAI identitet:

- Kratka verzija username-a (npr. mhorvat): login i webmail
- Duga verzija (npr. mhorvat.math@pmf.hr): sve ostalo
- Odmah promijenite početnu lozinku.
- U slučaju problema, javite se u RC (Računski centar, podrum).

Uvodne informacije

Literatura:

- Ovi (i drugi, npr. Singer, Nogo) slajdovi
- Skripta za vježbe (V. Šego)
- Kernighan, Ritchie: The C Programming Language
- Besplatan nacrt ANSI C standarda (1990. godina)
- cppreference.com

Programska podrška:

- Windows: Code::Blocks s gcc kompajlerom, Visual Studio
- Unix/Linux: cc, gcc

Najbitnija stvar koju ću reći danas: kodirajte što više.

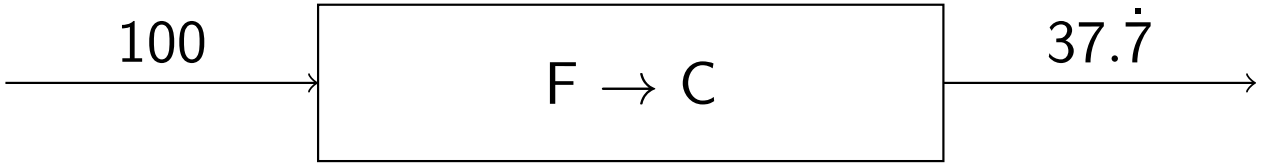
Uvod u algoritme

Algoritam je **KONAČAN** opis postupka za rješavanje nekog problema. Za sve ulaze na koje se dani problem odnosi (one koji su iz *domene* danog problema), taj postupak mora završiti u konačnom broju koraka i dati ispravan izlaz. Za ulaze van domene problema, postupak ne mora stati (čak ni imati smisla).

Uvod u algoritme

Algoritam je **KONAČAN** opis postupka za rješavanje nekog problema. Za sve ulaze na koje se dani problem odnosi (one koji su iz *domene* danog problema), taj postupak mora završiti u konačnom broju koraka i dati ispravan izlaz. Za ulaze van domene problema, postupak ne mora stati (čak ni imati smisla).

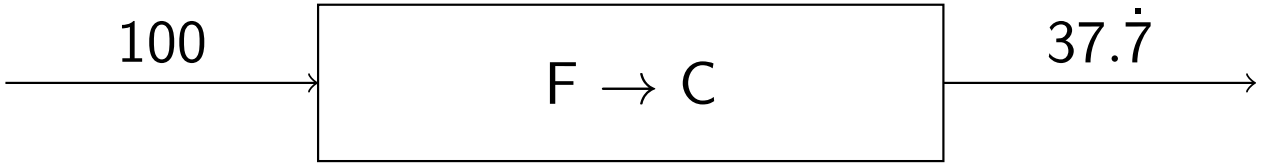
Primjer: pretvaranje F u C.



Uvod u algoritme

Algoritam je **KONAČAN** opis postupka za rješavanje nekog problema. Za sve ulaze na koje se dani problem odnosi (one koji su iz *domene* danog problema), taj postupak mora završiti u konačnom broju koraka i dati ispravan izlaz. Za ulaze van domene problema, postupak ne mora stati (čak ni imati smisla).

Primjer: pretvaranje F u C.



Gornja „definicija" algoritma je krajnje neprecizna (npr. što je korak?). Što bismo dobili davanjem precizne definicije kojom algoritam postaje matematički objekt?

Ako želite znati (puno) više o toj temi, položite (ili barem odslušajte) *Interpretaciju programa* i *Izračunljivost*.

Uvod u algoritme

Recimo da me zanima je li neki prirodan broj *prost* (točno dva prirodna broja ga dijele). Mogu li ga probati podijeliti svakim prirodnim brojem?

Uvod u algoritme

Recimo da me zanima je li neki prirodan broj *prost* (točno dva prirodna broja ga dijele). Mogu li ga probati podijeliti svakim prirodnim brojem? Općenito, to neće biti konačan postupak. Mogu li nekako to izbjeći?

Uvod u algoritme

Recimo da me zanima je li neki prirodan broj *prost* (točno dva prirodna broja ga dijele). Mogu li ga probati podijeliti svakim prirodnim brojem? Općenito, to neće biti konačan postupak. Mogu li nekako to izbjeći?

Da! Kod prirodnih brojeva djelitelj ne može biti veći od djeljenika, pa je dovoljno ulazni broj probati podijeliti manjim brojevima.

Uvod u algoritme

Recimo da me zanima je li neki prirodan broj *prost* (točno dva prirodna broja ga dijele). Mogu li ga probati podijeliti svakim prirodnim brojem? Općenito, to neće biti konačan postupak. Mogu li nekako to izbjeći?

Da! Kod prirodnih brojeva djelitelj ne može biti veći od djeljenika, pa je dovoljno ulazni broj probati podijeliti manjim brojevima.

Iako je ovaj postupak konačan, koraka u danom algoritmu može biti *jako* puno. Npr. zanima me je li $10^6 + 1$ prost. Ne želim ručno dijeliti, pa pišem program koji odgovara navedenom algoritmu:

1. instrukcija
2. instrukcija

   ...

n. instrukcija

Uvod u algoritme

Recimo da me zanima je li neki prirodan broj *prost* (točno dva prirodna broja ga dijele). Mogu li ga probati podijeliti svakim prirodnim brojem? Općenito, to neće biti konačan postupak. Mogu li nekako to izbjeći?

Da! Kod prirodnih brojeva djelitelj ne može biti veći od djeljenika, pa je dovoljno ulazni broj probati podijeliti manjim brojevima.

Iako je ovaj postupak konačan, koraka u danom algoritmu može biti *jako* puno. Npr. zanima me je li $10^6 + 1$ prost. Ne želim ručno dijeliti, pa pišem program koji odgovara navedenom algoritmu:

1. instrukcija
2. instrukcija

   ...

   n. instrukcija

Naravno, neću napisati 10^6 instrukcija. Instrukcije mogu biti *petlje* koje služe za ponavljanje drugih instrukcija (određen broj puta ili dok vrijedi neki uvjet). DZ: Napišite „pseudokod".

Uvod u algoritme

Petlje su samo jedan od primjera *(lokalne) kontrole toka*. Još jedan primjer je *grananje* („ako... onda... inače"). Ima i drugih instrukcija za kontrolu toka. O tome kasnije.

Osnovna svojstva algoritma:

- Broj (možda 0) i tip ulaznih podataka
- Broj (najčešće 1) i tip izlaznih podataka
- Nedvosmislen je
- U konačno koraka daje točan izlaz (za dozvoljene ulaze)
- Dovoljno je efikasan za danu svrhu
  - Za neke probleme se zna da nema efikasnog algoritma.
  - Za neke probleme se čini da nema efikasnog algoritma.
  - Za neke probleme se zna da nema algoritma *uopće*!

Program treba provjeriti je li ulaz dozvoljen: $x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

Uvod u algoritme

Što je instrukcija ovisi o vrsti i moći izvršitelja. Pogledajmo primjer dodavanja elementa `x` na prvo mjesta polja `polje`.

Python:

```
polje.insert(0, x)
```

C:

```
for(i = n-1; i >= 0; --i)
        polje[i+1] = polje[i];
polje[0] = x;
```

Ako želite vidjeti assemblerske instrukcije, kompajlirajte *program.c* sa *gcc -S program.c* i otvorite *program.s* u omiljenom text editoru.

Kratak uvod u građu računala

Sasvim pojednostavljeno, osnovni dijelovi računala su:

- Ulazna jedinica (pretvara podatke izvana u binarni oblik)
- Procesor
  - Aritmetičko-logička jedinica
  - Upravljačka jedinica
- Memorija računala
- Izlazna jedinica (pretvara binarne zapise u razumljive podatke)

Procesor ima *registre*, vlastitu memoriju u koju dohvaća podatke (instrukcije i operande) iz memorije računala, obavlja instrukcije na operandima i sprema rezultat nazad u memoriju računala.

Samo izračunavanje rezultata obavlja *aritmetičko-logička jedinica* procesora, a sve ostalo (fetch, decode, execute) *upravljačka jedinica* procesora.

Kratak uvod u građu računala

Memoriju računala čine bistabili (eng. flip-flop), sklopovi koji *pamte* jedan bit (0 ili 1). Preciznije, nalaze se u jednom od dva stabilna stanja dok se ne uloži energija kako bi se prebacili u ono drugo.

Danas se memorija izrađuje od sićušnih tranzistora koji rade kao elektronički prekidači.

Složeniji (*logički*) sklopovi omogućavaju implementaciju logičkih operacija na bitovima. Te operacije su analogne aritmetičkim:

- negacija bita b (NOT) $\Longleftrightarrow$

Kratak uvod u građu računala

Memoriju računala čine bistabili (eng. flip-flop), sklopovi koji *pamte* jedan bit (0 ili 1). Preciznije, nalaze se u jednom od dva stabilna stanja dok se ne uloži energija kako bi se prebacili u ono drugo.

Danas se memorija izrađuje od sićušnih tranzistora koji rade kao elektronički prekidači.

Složeniji (*logički*) sklopovi omogućavaju implementaciju logičkih operacija na bitovima. Te operacije su analogne aritmetičkim:

- negacija bita b (NOT) $\iff 1 - b$
- konjunkcija (AND) $\iff$

Kratak uvod u građu računala

Memoriju računala čine bistabili (eng. flip-flop), sklopovi koji *pamte* jedan bit (0 ili 1). Preciznije, nalaze se u jednom od dva stabilna stanja dok se ne uloži energija kako bi se prebacili u ono drugo.

Danas se memorija izrađuje od sićušnih tranzistora koji rade kao elektronički prekidači.

Složeniji (*logički*) sklopovi omogućavaju implementaciju logičkih operacija na bitovima. Te operacije su analogne aritmetičkim:

- negacija bita b (NOT) $\iff$ $1 - b$
- konjunkcija (AND) $\iff$ min (množenje)
- disjunkcija (OR) $\iff$

Kratak uvod u građu računala

Memoriju računala čine bistabili (eng. flip-flop), sklopovi koji *pamte* jedan bit (0 ili 1). Preciznije, nalaze se u jednom od dva stabilna stanja dok se ne uloži energija kako bi se prebacili u ono drugo.

Danas se memorija izrađuje od sićušnih tranzistora koji rade kao elektronički prekidači.

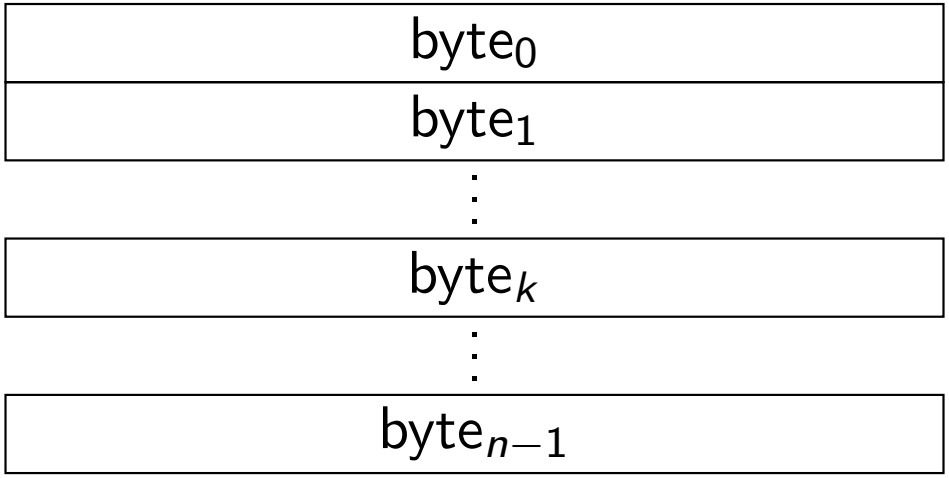
Složeniji (*logički*) sklopovi omogućavaju implementaciju logičkih operacija na bitovima. Te operacije su analogne aritmetičkim:

- negacija bita b (NOT) $\iff$ $1 - b$
- konjunkcija (AND) $\iff$ min (množenje)
- disjunkcija (OR) $\iff$ max („modificirano" zbrajanje)

Kad bitove organiziramo u veće cjeline, pomoću logičkih sklopova možemo implementirati zbrajanje, množenje, itd. Upravo te operacije obavlja aritmetičko-logička jedinica procesora.

Kratak uvod u građu računala

Memoriju možemo zamišljati kao konačan niz *riječi* (najmanje cjeline koje je moguće adresirati). Pretpostavljat ćemo da je svaka riječ *byte* od točno osam bita:

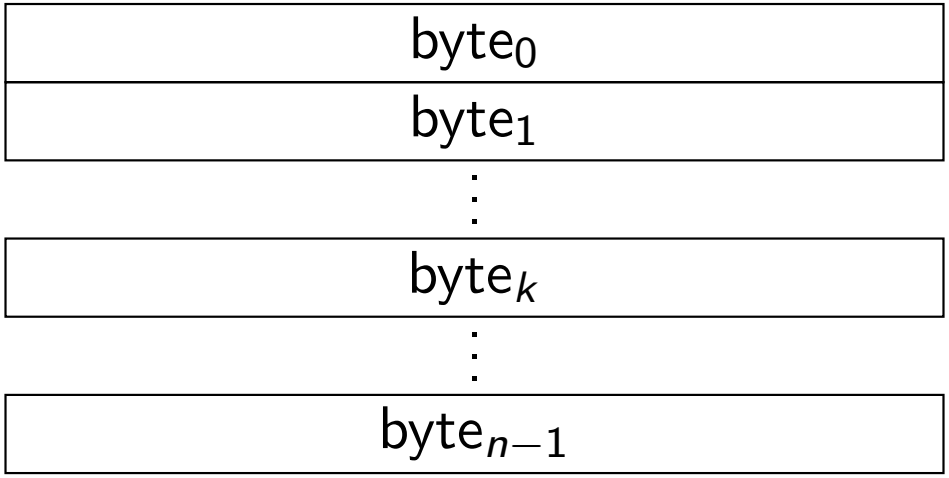


Primjerice, byte_k se nalazi na adresi k, a vrijednost bi mu mogla biti 00001010. Važno: i adrese i instrukcije su tako spremljene!

Stariji procesori mogu izravno operirati sa 32 bita, dakle 4 bajta. Tako je moguće adresirati

Kratak uvod u građu računala

Memoriju možemo zamišljati kao konačan niz *riječi* (najmanje cjeline koje je moguće adresirati). Pretpostavljat ćemo da je svaka riječ *byte* od točno osam bita:



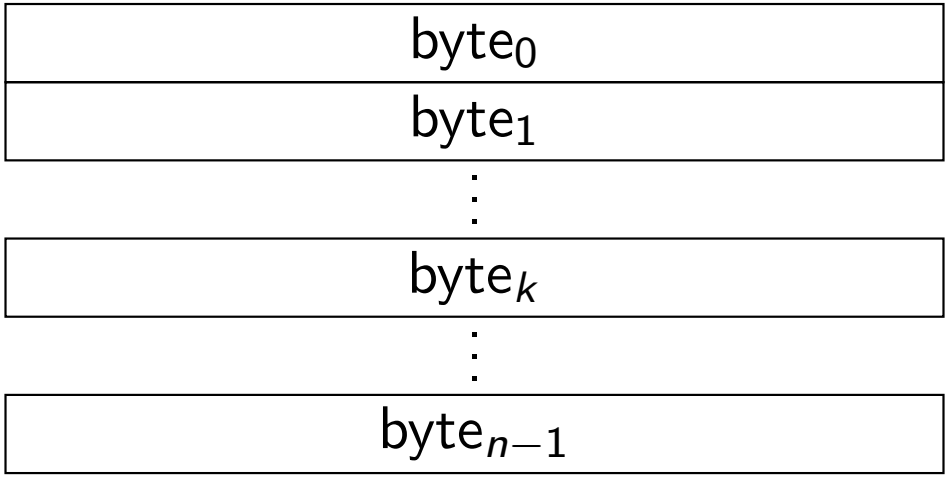
Primjerice, byte_k se nalazi na adresi k, a vrijednost bi mu mogla biti 00001010. Važno: i adrese i instrukcije su tako spremljene!

Stariji procesori mogu izravno operirati sa 32 bita, dakle 4 bajta. Tako je moguće adresirati 4 GB (ipak GiB, gibibajta :/) memorije.

Na novijim računalima (poput mog laptopa) adrese imaju 64 bita, odnosno moguće je adresirati

Kratak uvod u građu računala

Memoriju možemo zamišljati kao konačan niz *riječi* (najmanje cjeline koje je moguće adresirati). Pretpostavljat ćemo da je svaka riječ *byte* od točno osam bita:



Primjerice, byte_k se nalazi na adresi k, a vrijednost bi mu mogla biti 00001010. Važno: i adrese i instrukcije su tako spremljene!

Stariji procesori mogu izravno operirati sa 32 bita, dakle 4 bajta. Tako je moguće adresirati 4 GB (ipak GiB, gibibajta :/) memorije.

Na novijim računalima (poput mog laptopa) adrese imaju 64 bita, odnosno moguće je adresirati 16 EB (zapravo EiB, eksbibajta ://).

Primjer: Program za zbrajanje dva broja

0100	B6	napuni registar A sadržajem mem. lokacije 0201
0101	02	
0102	01	
0103	9B	pribroji sadržaju registra A sadržaj mem. lokacije 01FF
0104	01	
0105	FF	
0106	B7	napuni mem. lokaciju 0202 sadržajem registra A
0107	02	
0108	02	
⋮	⋮	
01FF	1A	drugi operand
⋮	⋮	
0201	23	prvi operand
0202	3D	rezultat

Tipovi podataka u računalu

Greške u programu idealno uočava programer, npr. pri prevođenju programa iz C-a u strojni kod, a ne korisnik pri izvođenju.

U tome pomažu *tipovi podataka* (ako operandi nisu onih tipova koje operacija očekuje, prevoditelj javi grešku). Naime, jedan ili više bajtova u memoriji mogu se interpretirati na različite načine:

- Cijeli brojevi s predznakom (`int`)
- Brojevi s pomičnim zarezom (`float`, `double`)
- Nenegativni cijeli brojevi bez predznaka (`unsigned`)
- Druge verzije numeričkih tipova (`short`, `long`, itd.)
- Osnovni (ASCII) znakovi (`char`)

Značajno širi spektar od `char`: UTF-8 (Unicode[1] implementacija).

Nema logičkog tipa; 0 je laž, a ostali `int`ovi istina. Adrese liče na `unsigned`, ali možda nisu te duljine na platformi za koju se program prevodi! Može se definirati još tipova (tek na Prog2).

[1] `https://www.cprogramming.com/tutorial/unicode.html`

Tipovi podataka u računalu

U bajtu bitove indeksiramo zdesna nalijevo od nultog („najnižeg”):

0	0	0	0	1	0	1	0
7	6	5	4	3	2	1	0

Indeks bita odgovara potenciji od 2 u binarnom zapisu.

Na IA-32, slično se zapisuje podatak od više bajtova: na nižim memorijskim lokacijama pišu niži bajtovi (little endian).

- Vidjeli smo *big endian*: 0101 | 02 / 0102 | 01 je bio zapis od 0201.
- Ovdje imamo *little endian*: ovo gore bi bio zapis od 0102.

Dogovor u C-u: adresa cjeline je ona na kojoj cjelina počinje.

Recimo, adresa `int`-a koji se proteže od 0100 do 0103 je 0100.

Tipovi podataka u računalu

Tipovi podataka daju značenje sadržaju memorije. O tom značenju ovise i vrste operacija koje možemo izvršavati na podacima danog tipa (npr. prikaz, aritmetika, itd.)

Jednostavan tip podatka: podaci su izravno podržani arhitekturom računala, tj. postoje instrukcije za njih. Takve operacije su uglavnom brze.

Jedan takav tip podatka su znakovi (`char`). Kodiraju se cijelim brojevima, a operacije na njima se svode na elementarne operacije s cijelim brojevima.

```
#include <stdio.h>
int main(void) {
        char c = '1';
        printf("%c\n", c); /* 1 */
        printf("%d\n", c); /* 49 */
        return 0;
}
```

Tipovi podataka u računalu

Primjer za logički „tip”:

```
#include <stdio.h>

int main(void) {
        int i = 10, j = 20;

        printf("%d\n", i < j); /* 1 */
        printf("%d\n", i >= j); /* 0 */
        printf("%d\n", i == j); /* 0 */

        return 0;
}
```

Tipovi podataka u računalu

Uobičajene *beskonačne* skupove brojeva ne možemo prikazati u računalu. No, možemo prikazati korisne podskupove.

Kod svakog tipa možemo u određenoj mjeri varirati broj bitova koji se koriste za prikaz i time mijenjati broj prikazivih brojeva.

Prijelaz na konačne skupove bitno mijenja realizaciju aritmetike — ne nasljeđuje se projekcijom s originalnog skupa! (Detalji uskoro.)

Cijeli brojevi bez predznaka: ima 2^n brojeva prikazivih u n bitova.

$$\mathbb{Z}_{2^n} = \{0, 1, 2, \ldots, 2^n - 2, 2^n - 1\}$$

Najveći prikazivi broj $(2^n - 1)$ za česte vrijednosti od n:

n	$2^n - 1$
8	255
16	65 535
32	4 294 967 295
64	18 446 744 073 709 551 615

Tipovi podataka u računalu

Dan je zapis nenegativnog broja B u bazi 2:

$$B = (b_k b_{k-1} \ldots b_1 b_0)_2 = b_k \cdot 2^k + \ldots + b_1 \cdot 2 + b_0$$

Tada su sljedeće tvrdnje ekvivalentne:

- $B \in \{0, 1, \ldots, 2^n - 1\}$.
- $k \leq n - 1$ („B je prikaziv u najviše n bitova").

Binarni zapis dopunimo nulama slijeva akko $k < n - 1$. Dakle, prikaz broja B kao cijelog broja bez predznaka je:

$$\text{bit}_i = \begin{cases} b_i, & i = 0, \ldots, k, \\ 0, & i = k+1, \ldots, n-1. \end{cases}$$

Tipovi podataka u računalu

Dan je zapis nenegativnog broja B u bazi 2:

$$B = (b_k b_{k-1} \ldots b_1 b_0)_2 = b_k \cdot 2^k + \ldots + b_1 \cdot 2 + b_0$$

Tada su sljedeće tvrdnje ekvivalentne:

- $B \in \{0, 1, \ldots, 2^n - 1\}$.
- $k \leq n - 1$ („B je prikaziv u najviše n bitova").

Binarni zapis dopunimo nulama slijeva akko $k < n - 1$. Dakle,

prikaz broja B kao cijelog broja bez predznaka je:

$$\text{bit}_i = \begin{cases} b_i, & i = 0, \ldots, k, \\ 0, & i = k+1, \ldots, n-1. \end{cases}$$

Primjer za $n = 8$ i $B = 123$: Zaista je $123 \leq 255 = 2^8 - 1$, pa je 123 prikaziv u 8 bita (čak u 7, ali hoćemo *prošireni binarni zapis*):

$$123 = \mathbf{0} \cdot 2^7 + \mathbf{1} \cdot 2^6 + \mathbf{1} \cdot 2^5 + \mathbf{1} \cdot 2^4 + \mathbf{1} \cdot 2^3 + \mathbf{0} \cdot 2^2 + \mathbf{1} \cdot 2^1 + \mathbf{1} \cdot 2^0.$$

0	**1**	**1**	**1**	**1**	**0**	**1**	**1**
7	6	5	4	3	2	1	0

Kraće: [0 1 1 1 1 0 1 1].

Tipovi podataka u računalu

Aritmetika cijelih brojeva bez predznaka s n bitova je aritmetika modulo 2^n. Drugim riječima, za prikazive operande A i B je rezultat od $A \, op \, B$ zapravo $(A \, op \, B) \bmod 2^n$. S tehničke strane, ovo je jednostavno i brzo (uzimamo samo najnižih n bitova rezultata), a s matematičke se radi o dobro poznatoj algebarskoj strukturi.

Pripadna algebarska struktura: $\mathbb{Z}_{2^n}$, *prsten ostataka modulo* 2^n.

Uobičajena matematička notacija: $(\mathbb{Z}_{2^n}, \oplus_{2^n}, \odot_{2^n})$.

Važno: ako je međurezultat (prije $\bmod 2^n$) neprikaziv, ne javlja se nužno greška — postavlja se bit prijenosa (eng. carry bit) na 1 u kontrolnom registru. Sam program na to treba paziti i smisleno postupiti. Primjerice, ako se radi o ,,adresi", program joj ne smije pristupiti; u protivnom bi operacijski sustav morao javiti grešku (poput *memory protection violation*) i srušiti program.

Dijeljenje cijelih brojeva bez predznaka

Obično dijelimo u poljima, poput $\mathbb{Q}$ i $\mathbb{R}$; rezultat dijeljenja je uvijek element danog polja. No, $\mathbb{Z}_{2^n}$, osim aritmetički nezanimljivog $\mathbb{Z}_2$, nije polje. Zato tamo dijelimo s ostatkom:

Teorem (o dijeljenju s ostatkom). Za sve $a \in \mathbb{Z}$ i $b \in \mathbb{N}$, postoje jedinstveni $q, r \in \mathbb{Z}$ takvi da je $a = q \cdot b + r$ i $0 \leq r < b$.

Dokaz. Na kolegiju *Elementarna matematika 1*. $\square$

Uvedimo sljedeće oznake: $a \operatorname{div} b := q$, $a \operatorname{mod} b := r$.

Veza s dijeljenjem (posljedica od $r \in \mathbb{Z}_b = \{0, \ldots, b-1\}$):

$$a \operatorname{div} b = \left\lfloor \frac{a}{b} \right\rfloor$$

U računalu, dijeljenje s ostatkom cijelih brojeva <u>bez predznaka</u> (unutar $\mathbb{N}_0 \times \mathbb{N}$) je restrikcija gore opisanih operacija div i mod.

Primjer: cijeli brojevi bez predznaka

```c
#include <stdio.h>

int main(void) {
        unsigned short i = 65535;

        printf("%d\n", i / 10);

        i = i + 3;
        printf("%d\n", i);

        return 0;
}
```

`USHRT_MAX` $= 65535$ u zaglavlju `limits.h`. Ovdje je $n = 16$.

Primjer: cijeli brojevi bez predznaka

```
#include <stdio.h>

int main(void) {
        unsigned short i = 65535;

        printf("%d\n", i / 10);   /* 6553 */

        i = i + 3;
        printf("%d\n", i);

        return 0;
}
```

USHRT_MAX $= 65535$ u zaglavlju `limits.h`. Ovdje je $n = 16$.

Primjer: cijeli brojevi bez predznaka

```
#include <stdio.h>

int main(void) {
        unsigned short i = 65535;

        printf("%d\n", i / 10);    /* 6553 */

        i = i + 3;
        printf("%d\n", i);         /* 2, ne 65538 */

        return 0;
}
```

USHRT_MAX $= 65535$ u zaglavlju `limits.h`. Ovdje je $n = 16$.

Primjer: cijeli brojevi bez predznaka

```
#include <stdio.h>

int main(void) {
        unsigned short i = 2, j = 4;

        i = i - j;
        printf("%d\n", i);

        return 0;
}
```

Koji je ispis ako premjestimo `i-j` direktno u `printf` (bez `i=i-j`)?

Primjer: cijeli brojevi bez predznaka

```c
#include <stdio.h>

int main(void) {
        unsigned short i = 2, j = 4;

        i = i - j;
        printf("%d\n", i);   /* 65534, a ne -2 */

        return 0;
}
```

Koji je ispis ako premjestimo `i-j` direktno u `printf` (bez `i=i-j`)?

Cijeli brojevi s predznakom

Ako želimo u n bitova spremati i negativne brojeve, složit ćemo se da je najbolje prikazati podjednako mnogo negativnih i pozitivnih brojeva. Možemo li tada sačuvati modularnu aritmetiku $\bmod 2^n$?

Cijeli brojevi s predznakom

Ako želimo u n bitova spremati i negativne brojeve, složit ćemo se da je najbolje prikazati podjednako mnogo negativnih i pozitivnih brojeva. Možemo li tada sačuvati modularnu aritmetiku $\bmod 2^n$?

Da! Recimo, za $n = 3$:

-4	1	0	0
-3	1	0	1
-2	1	1	0
-1	1	1	1
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1

Cijeli brojevi s predznakom

Ako želimo u n bitova spremati i negativne brojeve, složit ćemo se da je najbolje prikazati podjednako mnogo negativnih i pozitivnih brojeva. Možemo li tada sačuvati modularnu aritmetiku $\bmod 2^n$?

Da! Recimo, za $n = 3$:

-4	1	0	0
-3	1	0	1
-2	1	1	0
-1	1	1	1
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1

Primijetite:

- Sustav ostataka je sad $\mathbb{Z}_{2^n}^{-} = \{-2^{n-1}, \ldots, 2^{n-1} - 1\}$, pa je prsten $(\mathbb{Z}_{2^n}^{-}, \oplus_{2^n}, \odot_{2^n})$. Do na domenu, operacije su iste!

Cijeli brojevi s predznakom

Ako želimo u n bitova spremati i negativne brojeve, složit ćemo se da je najbolje prikazati podjednako mnogo negativnih i pozitivnih brojeva. Možemo li tada sačuvati modularnu aritmetiku $\bmod 2^n$?

Da! Recimo, za $n = 3$:

-4	1	0	0
-3	1	0	1
-2	1	1	0
-1	1	1	1
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1

Primijetite:

- Sustav ostataka je sad $\mathbb{Z}_{2^n}^{-} = \{-2^{n-1}, \ldots, 2^{n-1} - 1\}$, pa je prsten $(\mathbb{Z}_{2^n}^{-}, \oplus_{2^n}, \odot_{2^n})$. Do na domenu, operacije su iste!
- Od pozitivnog B lako dolazimo do $-B$:

Cijeli brojevi s predznakom

Ako želimo u n bitova spremati i negativne brojeve, složit ćemo se da je najbolje prikazati podjednako mnogo negativnih i pozitivnih brojeva. Možemo li tada sačuvati modularnu aritmetiku $\bmod 2^n$?

Da! Recimo, za $n = 3$:

-4	1	0	0
-3	1	0	1
-2	1	1	0
-1	1	1	1
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1

Primijetite:

- Sustav ostataka je sad $\mathbb{Z}_{2^n}^{-} = \{-2^{n-1}, \ldots, 2^{n-1} - 1\}$, pa je prsten $(\mathbb{Z}_{2^n}^{-}, \oplus_{2^n}, \odot_{2^n})$. Do na domenu, operacije su iste!
- Od pozitivnog B lako dolazimo do $-B$: to je $2^n - B$. Može li brže?

Cijeli brojevi s predznakom

Ako želimo u n bitova spremati i negativne brojeve, složit ćemo se da je najbolje prikazati podjednako mnogo negativnih i pozitivnih brojeva. Možemo li tada sačuvati modularnu aritmetiku $\bmod 2^n$?

Da! Recimo, za $n = 3$:

-4	1	0	0
-3	1	0	1
-2	1	1	0
-1	1	1	1
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1

Primijetite:

- Sustav ostataka je sad $\mathbb{Z}_{2^n}^{-} = \{-2^{n-1}, \ldots, 2^{n-1} - 1\}$, pa je prsten $(\mathbb{Z}_{2^n}^{-}, \oplus_{2^n}, \odot_{2^n})$. Do na domenu, operacije su iste!
- Od pozitivnog B lako dolazimo do $-B$: to je $2^n - B$. Može li brže? Da, komplementiramo $(0 \leftrightarrow 1)$ i dodamo 1 $(\bmod 2^n)$!

Cijeli brojevi s predznakom

Postupak opisan na kraju prošlog slajda (komplementiraj, dodaj 1 modulo 2^n) naziva se *dvojni komplement*. Opravdajmo ga ovdje.

Za B cijeli broj s predznakom označimo sa $\overline{B}$ broj dobiven zamjenom $0 \leftrightarrow 1$ u zapisu od B. Tada vrijedi:

$$B + \overline{B} =$$

Cijeli brojevi s predznakom

Postupak opisan na kraju prošlog slajda (komplementiraj, dodaj 1 modulo 2^n) naziva se *dvojni komplement*. Opravdajmo ga ovdje.

Za B cijeli broj s predznakom označimo sa $\overline{B}$ broj dobiven zamjenom $0 \leftrightarrow 1$ u zapisu od B. Tada vrijedi:

$$B + \overline{B} = [\ 1\ 1\ 1\ \ldots 1\ 1\ ] = 2^n - 1, \text{ tj.}$$

$$B + \overline{B} + 1 = 2^n.$$

Ako lijevu i desnu stranu pogledamo modulo 2^n, dobivamo

Cijeli brojevi s predznakom

Postupak opisan na kraju prošlog slajda (komplementiraj, dodaj 1 modulo 2^n) naziva se *dvojni komplement*. Opravdajmo ga ovdje.

Za B cijeli broj s predznakom označimo sa $\overline{B}$ broj dobiven zamjenom $0 \leftrightarrow 1$ u zapisu od B. Tada vrijedi:

$$B + \overline{B} = [\ 1\ 1\ 1 \ldots 1\ 1\ ] = 2^n - 1, \text{ tj.}$$

$$B + \overline{B} + 1 = 2^n.$$

Ako lijevu i desnu stranu pogledamo modulo 2^n, dobivamo

$$B \oplus_{2^n} (\overline{B} \oplus_{2^n} 1) = 0.$$

Dakle, suprotni element od $B \in \mathbb{Z}_{2^n}^-$ je zaista $\overline{B} \oplus_{2^n} 1$.

Cijeli brojevi s predznakom

Primjer: Zbrojimo najveći prikazivi broj i 1:

$$\begin{aligned}(2^{n-1}-1) \oplus_{2^n} 1 &= ((2^{n-1}-1)+1) \bmod 2^n \\ &= 2^{n-1} \bmod 2^n = -2^{n-1} \in \mathbb{Z}_{2^n}^-.\end{aligned}$$

$$[\ 0\ 1\ 1 \ldots 1\ 1\ ] + [\ 0\ 0\ 0 \ldots 0\ 1\ ] = [\ 1\ 0\ 0 \ldots 0\ 0\ ].$$

Primjer: Probajmo zbrojiti -1 i 1:

$$[\ 1\ 1\ 1 \ldots 1\ 1\ ] + [\ 0\ 0\ 0 \ldots 0\ 1\ ] = 1\ [\ 0\ 0\ 0 \ldots 0\ 0\ ].$$

Ovdje postoji prijenos, ali se modulo 2^n on efektivno ignorira.

Činjenica da prsteni $(\mathbb{Z}_{2^n}, \oplus_{2^n}, \odot_{2^n})$ i $(\mathbb{Z}_{2^n}^-, \oplus_{2^n}, \odot_{2^n})$ sadrže iste operacije (do na ovdje ionako nebitnu interpretaciju rezultata) omogućuje računalu da te operacije izvodi na <u>isti način</u>, odnosno pomoću <u>istih sklopova</u>.

Cijeli brojevi s predznakom

Primjer za $n = 8$ i $B = -120$: B je prikaziv jer vrijedi

Cijeli brojevi s predznakom

Primjer za $n = 8$ i $B = -120$: B je prikaziv jer vrijedi

$$-2^7 = -128 \leq -120 \leq 127 = 2^7 - 1.$$

Nađimo najprije prikaz, a zatim dvojni komplement od 120:

Cijeli brojevi s predznakom

Primjer za $n = 8$ i $B = -120$: B je prikaziv jer vrijedi

$$-2^7 = -128 \leq -120 \leq 127 = 2^7 - 1.$$

Nađimo najprije prikaz, a zatim dvojni komplement od 120:

$$\begin{aligned} 120 &= 64 + 32 + 16 + 8 \\ &= \mathbf{0} \cdot 2^7 + \mathbf{1} \cdot 2^6 + \mathbf{1} \cdot 2^5 + \mathbf{1} \cdot 2^4 \\ &+ \mathbf{1} \cdot 2^3 + \mathbf{0} \cdot 2^2 + \mathbf{0} \cdot 2^1 + \mathbf{0} \cdot 2^0. \end{aligned}$$

Cijeli brojevi s predznakom

Primjer za $n = 8$ i $B = -120$: B je prikaziv jer vrijedi

$$-2^7 = -128 \leq -120 \leq 127 = 2^7 - 1.$$

Nađimo najprije prikaz, a zatim dvojni komplement od 120:

$$\begin{aligned} 120 &= 64 + 32 + 16 + 8 \\ &= \mathbf{0} \cdot 2^7 + \mathbf{1} \cdot 2^6 + \mathbf{1} \cdot 2^5 + \mathbf{1} \cdot 2^4 \\ &+ \mathbf{1} \cdot 2^3 + \mathbf{0} \cdot 2^2 + \mathbf{0} \cdot 2^1 + \mathbf{0} \cdot 2^0. \end{aligned}$$

$$[\ 0\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ ] \mapsto [\ 1\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ ] \mapsto [\ 1\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ ] = -120.$$

Cijeli brojevi s predznakom

Primjer za $n = 8$ i $B = -120$: B je prikaziv jer vrijedi

$$-2^7 = -128 \leq -120 \leq 127 = 2^7 - 1.$$

Nađimo najprije prikaz, a zatim dvojni komplement od 120:

$$\begin{aligned} 120 &= 64 + 32 + 16 + 8 \\ &= \mathbf{0} \cdot 2^7 + \mathbf{1} \cdot 2^6 + \mathbf{1} \cdot 2^5 + \mathbf{1} \cdot 2^4 \\ &\quad + \mathbf{1} \cdot 2^3 + \mathbf{0} \cdot 2^2 + \mathbf{0} \cdot 2^1 + \mathbf{0} \cdot 2^0. \end{aligned}$$

$$[\ 0\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ ] \mapsto [\ 1\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ ] \mapsto [\ 1\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ ] = -120.$$

Najmanji i najveći prikazivi brojevi za česte vrijednosti od n:

n	-2^{n-1}	$2^{n-1} - 1$
8	-128	127
16	$-32\,768$	$32\,767$
32	$-2\,147\,483\,648$	$2\,147\,483\,647$

Danas se sve više koristi $n = 64$:

$$2^{63} - 1 = 9\,223\,372\,036\,854\,775\,807.$$

Dijeljenje cijelih brojeva s predznakom

Znamo kako računalo dijeli cijele brojeve bez predznaka. No, što će se dogoditi ako je barem jedan operand negativan? Probajmo:

a	b	a/b	a%b
5	3	1	2
−5	3	−1	−2
5	−3	−1	2
−5	−3	1	−2

U gornjoj tablici, / i % su redom implementacije operacija div i mod, proširenih sa skupa $\mathbb{Z} \times \mathbb{N}$ na skup $\mathbb{Z} \times (\mathbb{Z} \setminus \{0\})$.

Što kaže standard? C90: rezultat ovisi o implementaciji. C99 propisuje rezultat: kvocijent se zaokružuje prema nuli, ostatak ima predznak djeljenika (često je tako i kod drugih jezika). Preciznije:

$$q = sign\left(\frac{a}{b}\right) \cdot \left\lfloor \left|\frac{a}{b}\right| \right\rfloor, \qquad r = sign(a) \cdot (|a| \bmod |b|)$$

Uočimo: $r \in \mathbb{Z}_b$ ako je $a \geq 0$, odnosno $r \in -\mathbb{Z}_b$ ako je $a < 0$.

Primjeri: aritmetika cijelih brojeva u C-u

```c
#include <stdio.h>

int main(void) {
        short i = 32766;

        i += 1;
        printf("%d\n", i);
        i += 1;
        printf("%d\n", i);

        return 0;
}
```

`SHRT_MAX` $= 32767$ u zaglavlju `limits.h`. Za tip `short` je $n = 16$.

Primjeri: aritmetika cijelih brojeva u C-u

```c
#include <stdio.h>

int main(void) {
        short i = 32766;

        i += 1;
        printf("%d\n", i); /* 32767 */
        i += 1;
        printf("%d\n", i);

        return 0;
}
```

`SHRT_MAX` $= 32767$ u zaglavlju `limits.h`. Za tip `short` je $n = 16$.

Primjeri: aritmetika cijelih brojeva u C-u

```c
#include <stdio.h>

int main(void) {
        short i = 32766;

        i += 1;
        printf("%d\n", i); /* 32767 */
        i += 1;
        printf("%d\n", i); /* -32768, a ne 32768 */

        return 0;
}
```

`SHRT_MAX` $= 32767$ u zaglavlju `limits.h`. Za tip `short` je $n = 16$.

Primjeri: aritmetika cijelih brojeva u C-u

```c
#include <stdio.h>

int main(void) {
        int broj;
        broj = 5;
        printf("broj = %d\n", broj);
        broj = 2000000000;
        printf("broj = %d\n", broj);
        broj = 4000000000;
        printf("broj = %d\n", broj);
        broj = 8000000000;
        printf("broj = %d\n", broj);
        return 0;
}
```

Upozorava li i vaš kompajler *samo* na zadnje pridruživanje? Zašto?

Primjeri: aritmetika cijelih brojeva u C-u

```c
#include <stdio.h>

int main(void) {
        int broj;
        broj = 5;
        printf("broj = %d\n", broj); /* 5 */
        broj = 2000000000;
        printf("broj = %d\n", broj);
        broj = 4000000000;
        printf("broj = %d\n", broj);
        broj = 8000000000;
        printf("broj = %d\n", broj);
        return 0;
}
```

Upozorava li i vaš kompajler *samo* na zadnje pridruživanje? Zašto?

Primjeri: aritmetika cijelih brojeva u C-u

```c
#include <stdio.h>

int main(void) {
        int broj;
        broj = 5;
        printf("broj = %d\n", broj); /* 5 */
        broj = 2000000000;
        printf("broj = %d\n", broj); /* 2000000000 */
        broj = 4000000000;
        printf("broj = %d\n", broj);
        broj = 8000000000;
        printf("broj = %d\n", broj);
        return 0;
}
```

Upozorava li i vaš kompajler *samo* na zadnje pridruživanje? Zašto?

Primjeri: aritmetika cijelih brojeva u C-u

```c
#include <stdio.h>

int main(void) {
        int broj;
        broj = 5;
        printf("broj = %d\n", broj); /* 5 */
        broj = 2000000000;
        printf("broj = %d\n", broj); /* 2000000000 */
        broj = 4000000000;
        printf("broj = %d\n", broj); /* -294967296 */
        broj = 8000000000;
        printf("broj = %d\n", broj);
        return 0;
}
```

Upozorava li i vaš kompajler *samo* na zadnje pridruživanje? Zašto?

Primjeri: aritmetika cijelih brojeva u C-u

```c
#include <stdio.h>

int main(void) {
        int broj;
        broj = 5;
        printf("broj = %d\n", broj); /* 5 */
        broj = 2000000000;
        printf("broj = %d\n", broj); /* 2000000000 */
        broj = 4000000000;
        printf("broj = %d\n", broj); /* -294967296 */
        broj = 8000000000;
        printf("broj = %d\n", broj); /* -589934592 */
        return 0;
}
```

Upozorava li i vaš kompajler *samo* na zadnje pridruživanje? Zašto?

Primjeri: aritmetika cijelih brojeva u C-u

```
#include <stdio.h>

int main(void) {
        int broj;

        scanf("%d", &broj);
        printf(" ucitani broj = %d\n", broj);

        return 0;
}
```

Hoće li se i ovaj put kompajler žaliti pri pridruživanju? Zašto?

Primjeri: aritmetika cijelih brojeva u C-u

```c
#include <stdio.h>

int main(void) {
        int i, f50 = 1;                /* n = 32 */

        for (i = 2; i <= 50; ++i)
                f50 *= i;

        printf(" f50 = %d\n", f50);

        return 0;
}
```

Primjeri: aritmetika cijelih brojeva u C-u

```c
#include <stdio.h>

int main(void) {
        int i, f50 = 1;                 /* n = 32 */

        for (i = 2; i <= 50; ++i)
                f50 *= i;

        printf(" f50 = %d\n", f50);  /* f50 = 0 */

        return 0;
}
```

Primjeri: aritmetika cijelih brojeva u C-u

```
#include <stdio.h>

int main(void) {
        int i, f50 = 1;                 /* n = 32 */

        for (i = 2; i <= 50; ++i)
                f50 *= i;

        printf(" f50 = %d\n", f50);  /* f50 = 0 */

        return 0;
}
```

Zašto se ispiše nula?

Primjeri: aritmetika cijelih brojeva u C-u

```c
#include <stdio.h>

int main(void) {
        int i, f50 = 1;                 /* n = 32 */

        for (i = 2; i <= 50; ++i)
                f50 *= i;

        printf(" f50 = %d\n", f50);  /* f50 = 0 */

        return 0;
}
```

Zašto se ispiše nula? Uputa: nađite najveći k takav da 2^k dijeli 50!.

Primjeri: aritmetika cijelih brojeva u C-u

```c
#include <stdio.h>

int main(void) {
        int i, f50 = 1;                 /* n = 32 */

        for (i = 2; i <= 50; ++i)
                f50 *= i;

        printf(" f50 = %d\n", f50);  /* f50 = 0 */

        return 0;
}
```

Zašto se ispiše nula? Uputa: nađite najveći k takav da 2^k dijeli $50!$.

Zadatak: Za proizvoljan k, nađite najmanji n takav da 2^k dijeli $n!$.

Tablica $n!$ u modularnoj aritmetici i u $\mathbb{Z}$

n!	16 bitova	32 bita	aritmetika na $\mathbb{Z}$
7!	5040	5040	5040
8!	-25216	40320	40320
9!	-30336	362880	362880
10!	24320	3628800	3628800
11!	5376	39916800	39916800
12!	-1024	479001600	479001600
13!	-13312	1932053504	6227020800
14!	10240	1278945280	87178291200
15!	22528	2004310016	1307674368000
16!	-32768	2004189184	20922789888000
17!	-32768	-288522240	355687428096000
18!	0	-898433024	6402373705728000
19!	0	109641728	121645100408832000
20!	0	-2102132736	2432902008176640000

Uvod u prikaz realnih brojeva

Naravno da želimo računati i s brojevima koji nisu cijeli. Kako ćemo reprezentirati realne[2] brojeve?

Htjeli bismo pohranjivati i jako velike i jako male brojeve. U tome će nam pomoći *znanstvena notacija*.

Primjerice, za brojeve

$$678000000000000000000000.0 \qquad 0.00000000000000000002078$$

koristimo *znanstvenu notaciju* ovako:

$$6.78 \cdot 10^{22} \qquad 2.078 \cdot 10^{-19}$$

Za početak, ugrubo: -1 na predznak, puta *mantisa* („značajne znamenke"), puta baza na eksponent.

$$\text{broj} = (-1)^{\text{predznak}} \cdot \text{mantisa} \cdot \text{baza}^{\text{eksponent}}$$

[2] Većinom približno. Zapravo ćemo pamtiti i računati samo s racionalnim brojevima kojima mantisa nije preduga, a ostale ćemo njima aproksimirati.

Uvod u prikaz realnih brojeva

Za broj različit od nule zapisan kao binarni niz s točno jednom decimalnom točkom, *mantisa (u normaliziranom obliku)* počinje prvom jedinicom u tom zapisu te nastavlja točkom i ostalim značajnim znamenkama (onima iza kojih nisu sve znamenke nula). U prvom od sljedeća dva primjera, mantisa je 1.01011:

$$1010.11 = 1.01011 \cdot 2^3$$

$$0.0001011011 = 1.011011 \cdot 2^{-4}$$

Primijetite da se, kod broja različitog od nule, vodeća jedinica mantise u normaliziranom obliku ne mora pamtiti (ta jedinica se zove *skriveni bit*, eng. *hidden bit*; u nastavku ga ne smatramo dijelom mantise). To je zgodno jer dodatni bit koristimo za pamćenje dodatne znamenke.

U prvom primjeru, ako imamo 23 bita za zapis mantise i dopunjujemo nulama zdesna[3], zapis bi bio ,,01011" i 18 nula.

[3] Malo kasnije: tako će izgledati mantisa podatka tipa `float`.

Uvod u prikaz realnih brojeva

Mala komplikacija: eksponent se pohranjuje u pomaknutom obliku (takav oblik naziva se *karakteristika*).

Zašto? Eksponent je cijeli broj s predznakom kako bismo mogli reprezentirati sićušne i ogromne vrijednosti. No, lakše[4] je usporediti -3 i 3 u $\mathbb{Z}_8$ (pomaknute za $+4$ na 1 i 7) nego u $\mathbb{Z}_8^-$:

$$\mathbb{Z}_8^- \left\{ \begin{array}{rccc} -4 & 1 & 0 & 0 \\ -3 & 1 & 0 & 1 \\ -2 & 1 & 1 & 0 \\ -1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 2 & 0 & 1 & 0 \\ 3 & 0 & 1 & 1 \end{array} \right. \qquad \mathbb{Z}_8 \left\{ \begin{array}{rccc} -4+4 & 0 & 0 & 0 \\ -3+4 & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ -2+4 & 0 & 1 & 0 \\ -1+4 & 0 & 1 & 1 \\ 0+4 & 1 & 0 & 0 \\ 1+4 & 1 & 0 & 1 \\ 2+4 & 1 & 1 & 0 \\ 3+4 & \mathbf{1} & \mathbf{1} & \mathbf{1} \end{array} \right.$$

[4] Inače se u hardver ugrađuje prespecifična dodatna logika (u odnosu na leksikografsku usporedbu), pa je hardver nepotrebno veći, sporiji i skuplji.

Uvod u prikaz realnih brojeva

Format IEEE binary32 predviđa 8 bita za karakteristiku, dakle ona je iz skupa $\{0, \ldots, 255\}$. Dvije rubne su posebne (detalji malo kasnije), pa će „obične" karakteristike biti iz skupa $\{1, \ldots, 254\}$.

Mogli bismo pomisliti da su eksponenti prikazivi u IEEE binary32 iz skupa $E = \{-128, \ldots, 127\}$. To bi se svakako slagalo s našom idejom zapisa cijelog broja s predznakom u 8 bita. Ipak, nije tako!

Želimo[5] moći invertirati $2^{\min(E)}$. Za to nam treba

[5] Zašto baš njega? Jer najveći prikazivi broj u oba slučaja možemo invertirati denormaliziranim brojevima s malom mantisom; u binary32 idu čak do 2^{-149}.

Uvod u prikaz realnih brojeva

Format IEEE binary32 predviđa 8 bita za karakteristiku, dakle ona je iz skupa $\{0, \ldots, 255\}$. Dvije rubne su posebne (detalji malo kasnije), pa će „obične" karakteristike biti iz skupa $\{1, \ldots, 254\}$.

Mogli bismo pomisliti da su eksponenti prikazivi u IEEE binary32 iz skupa $E = \{-128, \ldots, 127\}$. To bi se svakako slagalo s našom idejom zapisa cijelog broja s predznakom u 8 bita. Ipak, nije tako!

Želimo[5] moći invertirati $2^{\min(E)}$. Za to nam treba $2^{-\min(E)}$, odnosno mora biti

[5] Zašto baš njega? Jer najveći prikazivi broj u oba slučaja možemo invertirati denormaliziranim brojevima s malom mantisom; u binary32 idu čak do 2^{-149}.

Uvod u prikaz realnih brojeva

Format IEEE binary32 predviđa 8 bita za karakteristiku, dakle ona je iz skupa $\{0, \ldots, 255\}$. Dvije rubne su posebne (detalji malo kasnije), pa će „obične" karakteristike biti iz skupa $\{1, \ldots, 254\}$.

Mogli bismo pomisliti da su eksponenti prikazivi u IEEE binary32 iz skupa $E = \{-128, \ldots, 127\}$. To bi se svakako slagalo s našom idejom zapisa cijelog broja s predznakom u 8 bita. Ipak, nije tako!

Želimo[5] moći invertirati $2^{\min(E)}$. Za to nam treba $2^{-\min(E)}$, odnosno mora biti $-\min(E) \leq \max(E)$. Mogli bismo umjesto E uzeti $E + 1$, samo što smo se dogovorili čuvati rubove za posebne slučajeve. Prema tome, skup eksponenata za IEEE binary32 jest

[5] Zašto baš njega? Jer najveći prikazivi broj u oba slučaja možemo invertirati denormaliziranim brojevima s malom mantisom; u binary32 idu čak do 2^{-149}.

Uvod u prikaz realnih brojeva

Format IEEE binary32 predviđa 8 bita za karakteristiku, dakle ona je iz skupa $\{0, \ldots, 255\}$. Dvije rubne su posebne (detalji malo kasnije), pa će ,,obične" karakteristike biti iz skupa $\{1, \ldots, 254\}$.

Mogli bismo pomisliti da su eksponenti prikazivi u IEEE binary32 iz skupa $E = \{-128, \ldots, 127\}$. To bi se svakako slagalo s našom idejom zapisa cijelog broja s predznakom u 8 bita. Ipak, nije tako!

Želimo[5] moći invertirati $2^{\min(E)}$. Za to nam treba $2^{-\min(E)}$, odnosno mora biti $-\min(E) \leq \max(E)$. Mogli bismo umjesto E uzeti $E + 1$, samo što smo se dogovorili čuvati rubove za posebne slučajeve. Prema tome, skup eksponenata za IEEE binary32 jest

$$\{-126, \ldots, 127\},$$

pa je pomak do skupa ,,običnih" karakteristika

[5] Zašto baš njega? Jer najveći prikazivi broj u oba slučaja možemo invertirati denormaliziranim brojevima s malom mantisom; u binary32 idu čak do 2^{-149}.

Uvod u prikaz realnih brojeva

Format IEEE binary32 predviđa 8 bita za karakteristiku, dakle ona je iz skupa $\{0, \ldots, 255\}$. Dvije rubne su posebne (detalji malo kasnije), pa će ,,obične" karakteristike biti iz skupa $\{1, \ldots, 254\}$.

Mogli bismo pomisliti da su eksponenti prikazivi u IEEE binary32 iz skupa $E = \{-128, \ldots, 127\}$. To bi se svakako slagalo s našom idejom zapisa cijelog broja s predznakom u 8 bita. Ipak, nije tako!

Želimo[5] moći invertirati $2^{\min(E)}$. Za to nam treba $2^{-\min(E)}$, odnosno mora biti $-\min(E) \leq \max(E)$. Mogli bismo umjesto E uzeti $E + 1$, samo što smo se dogovorili čuvati rubove za posebne slučajeve. Prema tome, skup eksponenata za IEEE binary32 jest

$$\{-126, \ldots, 127\},$$

pa je pomak do skupa ,,običnih" karakteristika **127**. Općenito, za karakteristiku *širine* w (ovdje je $w = 8$), pomak je

[5] Zašto baš njega? Jer najveći prikazivi broj u oba slučaja možemo invertirati denormaliziranim brojevima s malom mantisom; u binary32 idu čak do 2^{-149}.

Uvod u prikaz realnih brojeva

Format IEEE binary32 predviđa 8 bita za karakteristiku, dakle ona je iz skupa $\{0, \ldots, 255\}$. Dvije rubne su posebne (detalji malo kasnije), pa će ,,obične" karakteristike biti iz skupa $\{1, \ldots, 254\}$.

Mogli bismo pomisliti da su eksponenti prikazivi u IEEE binary32 iz skupa $E = \{-128, \ldots, 127\}$. To bi se svakako slagalo s našom idejom zapisa cijelog broja s predznakom u 8 bita. Ipak, nije tako!

Želimo[5] moći invertirati $2^{\min(E)}$. Za to nam treba $2^{-\min(E)}$, odnosno mora biti $-\min(E) \leq \max(E)$. Mogli bismo umjesto E uzeti $E + 1$, samo što smo se dogovorili čuvati rubove za posebne slučajeve. Prema tome, skup eksponenata za IEEE binary32 jest

$$\{-126, \ldots, 127\},$$

pa je pomak do skupa ,,običnih" karakteristika **127**. Općenito, za karakteristiku *širine* w (ovdje je $w = 8$), pomak je $2^{w-1} - 1$.

[5] Zašto baš njega? Jer najveći prikazivi broj u oba slučaja možemo invertirati denormaliziranim brojevima s malom mantisom; u binary32 idu čak do 2^{-149}.

Uvod u prikaz realnih brojeva

Primjer s kolokvija 2019/20.:

Koji prikaz[6] *u tipu float (IEEE binary32) ima realan broj* -259.5?

Prvo zapišimo -259.5 u znanstvenoj notaciji:

$$-259.5 =$$

[6] Kako biste provjerili da je broj prikaziv (nije potrebno na kolokviju)?

Uvod u prikaz realnih brojeva

Primjer s kolokvija 2019/20.:

Koji prikaz[6] *u tipu float (IEEE binary32) ima realan broj* -259.5?

Prvo zapišimo -259.5 u znanstvenoj notaciji:

$$\begin{aligned} -259.5 &= (-1)^1 \cdot 100000011.1 \\ &= \end{aligned}$$

[6] Kako biste provjerili da je broj prikaziv (nije potrebno na kolokviju)?

Uvod u prikaz realnih brojeva

Primjer s kolokvija 2019/20.:

Koji prikaz[6] *u tipu float (IEEE binary32) ima realan broj* -259.5?

Prvo zapišimo -259.5 u znanstvenoj notaciji:

$$\begin{aligned} -259.5 &= (-1)^1 \cdot 100000011.1 \\ &= (-1)^1 \cdot 1.000000111 \cdot 2^8 \end{aligned}$$

[6] Kako biste provjerili da je broj prikaziv (nije potrebno na kolokviju)?

Uvod u prikaz realnih brojeva

Primjer s kolokvija 2019/20.:

Koji prikaz[6] *u tipu float (IEEE binary32) ima realan broj* -259.5?

Prvo zapišimo -259.5 u znanstvenoj notaciji:

$$\begin{aligned} -259.5 &= (-1)^1 \cdot 100000011.1 \\ &= (-1)^1 \cdot 1.000000111 \cdot 2^8 \end{aligned}$$

Dakle, predznak je

[6] Kako biste provjerili da je broj prikaziv (nije potrebno na kolokviju)?

Uvod u prikaz realnih brojeva

Primjer s kolokvija 2019/20.:

Koji prikaz[6] *u tipu float (IEEE binary32) ima realan broj* -259.5?

Prvo zapišimo -259.5 u znanstvenoj notaciji:

$$\begin{aligned} -259.5 &= (-1)^1 \cdot 100000011.1 \\ &= (-1)^1 \cdot 1.000000111 \cdot 2^8 \end{aligned}$$

Dakle, predznak je 1, mantisa je

[6] Kako biste provjerili da je broj prikaziv (nije potrebno na kolokviju)?

Uvod u prikaz realnih brojeva

Primjer s kolokvija 2019/20.:

Koji prikaz[6] *u tipu float (IEEE binary32) ima realan broj* -259.5?

Prvo zapišimo -259.5 u znanstvenoj notaciji:

$$\begin{aligned} -259.5 &= (-1)^1 \cdot 100000011.1 \\ &= (-1)^1 \cdot 1.000000111 \cdot 2^8 \end{aligned}$$

Dakle, predznak je 1, mantisa je 000000111 i 14 nula, karakteristika je

[6] Kako biste provjerili da je broj prikaziv (nije potrebno na kolokviju)?

Uvod u prikaz realnih brojeva

Primjer s kolokvija 2019/20.:

Koji prikaz[6] *u tipu float (IEEE binary32) ima realan broj* -259.5?

Prvo zapišimo -259.5 u znanstvenoj notaciji:

$$\begin{aligned} -259.5 &= (-1)^1 \cdot 100000011.1 \\ &= (-1)^1 \cdot 1.000000111 \cdot 2^8 \end{aligned}$$

Dakle, predznak je 1, mantisa je 000000111 i 14 nula, karakteristika je binarni zapis od 8+127=135, odnosno

[6] Kako biste provjerili da je broj prikaziv (nije potrebno na kolokviju)?

Uvod u prikaz realnih brojeva

Primjer s kolokvija 2019/20.:

Koji prikaz[6] *u tipu float (IEEE binary32) ima realan broj* -259.5?

Prvo zapišimo -259.5 u znanstvenoj notaciji:

$$\begin{aligned} -259.5 &= (-1)^1 \cdot 100000011.1 \\ &= (-1)^1 \cdot 1.000000111 \cdot 2^8 \end{aligned}$$

Dakle, predznak je 1, mantisa je 000000111 i 14 nula, karakteristika je binarni zapis od 8+127=135, odnosno 10000111. Zapisujemo redom predznak, karakteristiku i mantisu u 32 bita.

Primjer:

Koji prikaz u tipu float (IEEE binary32) ima realan broj 519?

[6] Kako biste provjerili da je broj prikaziv (nije potrebno na kolokviju)?

Uvod u prikaz realnih brojeva

Primjer s kolokvija 2019/20.:

Koji prikaz[6] *u tipu float (IEEE binary32) ima realan broj* -259.5?

Prvo zapišimo -259.5 u znanstvenoj notaciji:

$$\begin{aligned} -259.5 &= (-1)^1 \cdot 100000011.1 \\ &= (-1)^1 \cdot 1.000000111 \cdot 2^8 \end{aligned}$$

Dakle, predznak je 1, mantisa je 000000111 i 14 nula, karakteristika je binarni zapis od 8+127=135, odnosno 10000111. Zapisujemo redom predznak, karakteristiku i mantisu u 32 bita.

Primjer:

Koji prikaz u tipu float (IEEE binary32) ima realan broj 519?

To je zapravo dvostruko veći broj (samo s drugim predznakom). Stoga je predznak sad 0 te karakteristika za 1 veća (10001000).

[6] Kako biste provjerili da je broj prikaziv (nije potrebno na kolokviju)?

Uvod u prikaz realnih brojeva

Ukratko prolazimo kroz posebne slučajeve prikaza za primjer $n = 8$.

Nula ima dva prikaza: karakteristika i mantisa su same nule, a predznak 0 („pozitivna nula") ili 1 („negativna nula"). Te dvije vrijednosti su pri usporedbi jednake.

Ako je karakteristika nula, a mantisa nije, mantisa počinje s „0." (*denormalizirani brojevi*) i eksponent je najmanji mogući, tj. -126.

Ako je karakteristika 255, a mantisa 0, onda o predznaku ovisi je li zapisana pozitivna ili negativna beskonačnost (`Inf` ili `-Inf`). To je rezultat[7] spremanja prevelikog broja (eng. *overflow*) ili dijeljenja ne-nul broja nulom.

Ako je karakteristika 255, a mantisa nije 0, onda je zapisan `NaN` („Not a Number"). Dobije se npr. pri dijeljenju nule nulom, korjenovanju negativnog broja, oduzimanju `Inf` od `Inf`, itd.

[7] Program primi i SIGFPE signal. Defaultno se program tad ruši, ali to se može mijenjati: `https://www.gnu.org/software/libc/manual/html_node/Signal-Handling.html`.

Uvod u prikaz realnih brojeva

IEEE tip binary32 $\approx$ `float` u C-u:

$$v = \begin{cases} (-1)^s \cdot 2^{k-127} \cdot 1.m, & \text{ako } 0 < k < 255, \\ (-1)^s \cdot 2^{-126} \cdot 0.m, & \text{ako } k = 0 \text{ i } m \neq 0, \\ (-1)^s \cdot 0, & \text{ako } k = 0 \text{ i } m = 0, \\ (-1)^s \cdot \texttt{Inf}, & \text{ako } k = 255 \text{ i } m = 0, \\ \texttt{NaN}, & \text{ako } k = 255 \text{ i } m \neq 0. \end{cases}$$

IEEE tip binary64 $\approx$ `double` u C-u:

$$v = \begin{cases} (-1)^s \cdot 2^{k-1023} \cdot 1.m, & \text{ako } 0 < k < 2047, \\ (-1)^s \cdot 2^{-1022} \cdot 0.m, & \text{ako } k = 0 \text{ i } m \neq 0, \\ (-1)^s \cdot 0, & \text{ako } k = 0 \text{ i } m = 0, \\ (-1)^s \cdot \texttt{Inf}, & \text{ako } k = 2047 \text{ i } m = 0, \\ \texttt{NaN}, & \text{ako } k = 2047 \text{ i } m \neq 0. \end{cases}$$

<u>DZ:</u> Koji su najmanji i najveći prikazivi pozitivni brojevi?

Uvod u prikaz realnih brojeva

Neke realne brojeve ne možemo egzaktno spremiti jer imaju predugačku mantisu.

Primjer: Realni broj (u binarnom zapisu)

$$B = 1.0001\ 0000\ 1000\ 0011\ 1001\ 0111$$

ima 24 znamenke mantise i ne može se egzaktno spremiti kao realni broj jednostruke točnosti, odnosno podatak tipa `float`.

Tada se pronalaze B_- i B_+, dva najbliža prikaziva susjeda broju B (primijetite da tako možemo pogriješiti najviše za 2^{-24}):

$$B_- < B < B_+$$

U gornjem primjeru

$$B_- = 1.0001\ 0000\ 1000\ 0011\ 1001\ 011$$
$$B_+ = 1.0001\ 0000\ 1000\ 0011\ 1001\ 100$$

Defaultni rezultat (može se mijenjati): onaj koji je strogo bliži ako postoji; inače onaj sa zadnjim bitom jednakim 0 (ovdje je to B_+).

Aritmetika realnih brojeva u računalu

Primjer: Na prethodnom slajdu smo vidjeli da broj

$$B = 1.0001\ 0000\ 1000\ 0011\ 1001\ 0111$$

nije podatak tipa `float` jer ima predugu mantisu. Vidimo da se B može dobiti kao zbroj sljedeća dva broja:

$$B_1 = 1.0001\ 0000\ 1000\ 0011\ 1001\ 011$$
$$B_2 = 1.0 \cdot 2^{-24}$$

Drugim riječima, zbrojimo li B_1 i B_2 u računalu, nećemo dobiti B, već B_+ (aproksimacija broja B koja jest prikaziva u računalu):

$$B_+ = 1.0001\ 0000\ 1000\ 0011\ 1001\ 100$$

Standard IEEE 754-2008 za realnu aritmetiku računala propisuje da osnovne aritmetičke operacije ne smiju pogriješiti više pri zaokruživanju nego je moguće pogriješiti pri prikazu broja. Slično vrijedi za drugi korijen, ali ne npr. za sin oko 0 ili ln oko 1.

Aritmetika realnih brojeva u računalu

Zbog zaokruživanja u realnoj aritmetici računala, ne vrijede asocijativnost i distributivnost (komutativnost vrijedi!).

Primjer: n-tu parcijalnu sumu harmonijskog reda

$$1 + \frac{1}{2} + \frac{1}{3} + \dots$$

možemo računati u bilo kojem od sljedeća dva redoslijeda:

$$S_{n,1} = \left(\dots \left(\left(1 + \frac{1}{2} \right) + \frac{1}{3} \right) + \dots + \frac{1}{n-1} \right) + \frac{1}{n}$$

$$S_{n,2} = 1 + \left(\frac{1}{2} + \left(\frac{1}{3} + \dots + \left(\frac{1}{n-1} + \frac{1}{n} \right) \dots \right) \right)$$

Za $n = 1\,000\,000$, dobit ćemo sljedeće rezultate u računalu:

`float` S_1	14.3573579788208007812
`float` S_2	14.3926515579223632812
`double` S_1	14.3927267228647810526
`double` S_2	14.3927267228657544962

Aritmetika realnih brojeva u računalu

Kao što vidimo, stalno akumuliramo greške u aritmetici računala (zato je dobro koristiti što veću preciznost, recimo tip `double` umjesto tipa `float`). Najveće greške nastaju oduzimanjem podjednako velikih brojeva („kraćenje"), tj. dobivanjem malih brojeva aditivnim operacijama.

Puno više o ovoj temi ćete čuti na kolegiju *Numerička matematika*, a u međuvremenu pogledajte materijale prof. Singera (slajdove i dodatak ovom predavanju). Mi ćemo sad napraviti još tri primjera.

Primjer: katastrofalno kraćenje

Pretpostavimo da u računalu radimo realnu aritmetiku u bazi 10 te mantisa može imati do 4 dekadske znamenke, a eksponent 2.

Tada sljedeći brojevi nisu prikazivi:

$$x = 8.8866 = 8.8866 \cdot 10^0 \text{ (spremamo } \texttt{float}\ 8.887 \cdot 10^0\text{)}$$
$$y = 8.8844 = 8.8844 \cdot 10^0 \text{ (spremamo } \texttt{float}\ 8.884 \cdot 10^0\text{)}$$

Ako oduzmemo `float`-ove (izjednačimo eksponente uz eventualni pomak točke, oduzmemo mantise te normaliziramo), naići ćemo na sljedeći problem:

$$8.887 \cdot 10^0 - 8.884 \cdot 10^0 = 0.003 \cdot 10^0 = 3.??? \cdot 10^{-3}$$

„Popravimo" poput računala kako bi oduzimanje bilo egzaktno: $3.000 \cdot 10^{-3}$. No, počeli smo od aproksimacija i zato fulali već u prvoj (najbitnijoj!) znamenci: $x - y = 2.2 \cdot 10^{-3}$.

Da su polazni operandi bili egzaktni, opet bismo zaokruživali pri oduzimanju, ali bi rezultat bio najbliži zapisiv (*benigno kraćenje*).

Primjer: kvadratna jednadžba

Popričali smo o rješenjima kvadratne jednadžbe $x^2 + px + q = 0$:

$$x_{1,2} = \frac{-p \pm \sqrt{p^2 - 4q}}{2}$$

Bolje ih je računati ,,po formuli'':

$$x_{1,2} = -\frac{p}{2} \pm \sqrt{\left(\frac{p}{2}\right)^2 - q}$$

uz to da na početku izračunamo i zapamtimo $\frac{p}{2}$ ili $-\frac{p}{2}$ (manja mogućnost overflowa zbog p^2 sad kad smo proveli tzv. *skaliranje*; također, imamo jedno množenje manje).

Dvije operacije koje su još uvijek problematične u gornjoj formuli su dva oduzimanja — ono ispod korijena ne možemo jednostavno popraviti, ali pokušat ćemo razmotriti ono izvan korijena.

Primjer: kvadratna jednadžba

Primjer: Rješavamo $x^2 - 56x + 1 = 0$. U bazi 10 s duljinom mantise 5 dobijemo

$$x_1 = 0.0180, \quad x_2 = 55.982,$$

a točna rješenja su:

$$x_1 = 0.0178628\ldots, \quad x_2 = 55.982137\ldots;$$

veći rezultat je točan, ali manji nije!

Kraćenje koje je uzrokovalo relativnu grešku od $7.7 \cdot 10^{-3}$: oduzimanje na mjestu $\pm$.

Možemo ga izračunati iz

$$x_1 \cdot x_2 = q.$$

Naime, x_2 je točan, pa je formula za x_1 bez „opasnog" kraćenja:

$$x_1 = \frac{q}{x_2}.$$

Primjer: promašaj rakete Patriot

U prvom Zaljevskom ratu, 25. veljače 1991. godine, američke rakete Patriot nisu uspjele oboriti iračku Scud raketu iznad Dhahrana u Saudijskog Arabiji. 28 mrtvih, 100+ ranjenih.

Primjer: promašaj rakete Patriot

U prvom Zaljevskom ratu, 25. veljače 1991. godine, američke rakete Patriot nisu uspjele oboriti iračku Scud raketu iznad Dhahrana u Saudijskog Arabiji. 28 mrtvih, 100+ ranjenih.

Sustav je računao vrijeme brojeći desetinke sekundi od uključivanja.

$$(0.1)_{10} = (0.0\overline{0011})_2 = 0.\underbrace{00011001100110011001100}_{\text{23 bita}}\ldots$$

Kolika je greška (korištena je nenormalizirana mantisa od 23 bita)?

Primjer: promašaj rakete Patriot

U prvom Zaljevskom ratu, 25. veljače 1991. godine, američke rakete Patriot nisu uspjele oboriti iračku Scud raketu iznad Dhahrana u Saudijskog Arabiji. 28 mrtvih, 100+ ranjenih.

Sustav je računao vrijeme brojeći desetinke sekundi od uključivanja.

$$(0.1)_{10} = (0.0\overline{0011})_2 = 0.\underbrace{00011001100110011001100}_{\text{23 bita}}\ldots$$

Kolika je greška (korištena je nenormalizirana mantisa od 23 bita)?

$$2^{-24} + 2^{-25} + 2^{-28} + 2^{-29} + 2^{-32} + \ldots =$$

Primjer: promašaj rakete Patriot

U prvom Zaljevskom ratu, 25. veljače 1991. godine, američke rakete Patriot nisu uspjele oboriti iračku Scud raketu iznad Dhahrana u Saudijskog Arabiji. 28 mrtvih, 100+ ranjenih.

Sustav je računao vrijeme brojeći desetinke sekundi od uključivanja.

$$(0.1)_{10} = (0.0\overline{0011})_2 = 0.\underbrace{00011001100110011001100}_{\text{23 bita}}\ldots$$

Kolika je greška (korištena je nenormalizirana mantisa od 23 bita)?

$$2^{-24} + 2^{-25} + 2^{-28} + 2^{-29} + 2^{-32} + \ldots = \frac{2^{-21}}{5} \approx 9.54 \cdot 10^{-8}.$$

Sitnica, zar ne?

Primjer: promašaj rakete Patriot

U prvom Zaljevskom ratu, 25. veljače 1991. godine, američke rakete Patriot nisu uspjele oboriti iračku Scud raketu iznad Dhahrana u Saudijskog Arabiji. 28 mrtvih, 100+ ranjenih.

Sustav je računao vrijeme brojeći desetinke sekundi od uključivanja.

$$(0.1)_{10} = (0.0\overline{0011})_2 = 0.\underbrace{00011001100110011001100}_{\text{23 bita}}\ldots$$

Kolika je greška (korištena je nenormalizirana mantisa od 23 bita)?

$$2^{-24} + 2^{-25} + 2^{-28} + 2^{-29} + 2^{-32} + \ldots = \frac{2^{-21}}{5} \approx 9.54 \cdot 10^{-8}.$$

Sitnica, zar ne? Ne. Stotinjak sati računalo nije resetirano:

$$100 \cdot 3600 \cdot 10 \cdot \frac{2^{-21}}{5} \approx 0.343s$$

Scud je brzine Mach 5, dakle oko 1.6 km/s, pa je tražena više od pola kilometra daleko od stvarnog položaja. Inače, greška je uočena 2 tjedna ranije, nakon 8 sati rada jednog drugog sustava, pa se mogla spriječiti običnim resetom. *Patch* je stigao dan prekasno.

Uvod u programske jezike

Gruba podjela programskih jezika:

- Strojni jezici: izvršni program, instrukcije u *binarnom kodu*
- Assembleri: izvorni program (tekst koji treba prevesti), prijevod u binarni kod je trivijalan (*mnemonički kod* može sadržavati imena instrukcija, registara i podataka)
- Viši programski jezici: izvorni program, udaljava se od arhitekture računala kako bi se približio namjeni

Primjeri: C, C++, C#, Java, Python, Rust, itd., itd., itd.

Nedostaci strojnih jezika/assemblera:

- Nedostatak portabilnosti (prenosivosti)
- Programi dugi i nepregledni
  - Programiranje podložnije greškama koje se teško otkrivaju

Prednosti strojnih jezika/assemblera:

- Puna kontrola nad hardverom (paralelno računanje, cache)
- Maksimalna brzina/efikasnost

Uvod u programske jezike

Prednosti viših jezika nad strojnim jezicima/assemblerima:

- Neovisnost o arhitekturi računala (portabilnost)
- Prilagođeniji namjeni
- Složenije naredbe bliže našem razmišljanju
- Brže i jednostavnije programiranje

Program se iz izvornog koda (eng. *source code*) prevodi u izvršni kod (eng. *executable code*).

Prevođenje obavlja prevoditelj (eng. *compiler*). Čin prevođenja može biti vrlo složen (više na *Interpretaciji programa*). Dobiven izvršni program se izvršava na ciljnoj platformi.

C je viši programski jezik opće namjene.

- Autor: Dennis Ritchie
- Vrijeme: '70. godine prošlog stoljeća
- Svrha: Pisanje jezgre operacijskog sustava Unix
- Motivacija: Portabilnost

Uvod u C

Inicijalna svrha jezika C je jako utjecala na njegov izgled.

C, u odnosu na druge više jezike, je poprilično blizu arhitekturi računala, odnosno assemblerima („high level assembler").

S *low-level* strane, C nudi:

- Direktnu manipulaciju znakovima, brojevima i adresama...
- ...Pomoću aritmetičkih, logičkih, relacijskih (usporedbe) i bitwise operacija

S *high-level* strane, C ima:

- Grupiranje naredbi (*blokove*)
- Naredbe za kontrolu toka (petlje, grananja, itd.)
- Složene tipove podataka (polja, strukture, datoteke, itd.)
- Modularnost (program je podijeljen u više datoteka)
- Manje cjeline/potprograme (funkcije)
  - Vraćaju vrijednosti osnovnih i (nekih) složenih tipova (strukture)
  - Mogu se rekurzivno pozivati

Uvod u C

C ima standardnu programsku biblioteku koja sadrži:

- Funkcije za interakciju s operacijskim sustavom
  - (Formatirano) čitanje i pisanje
  - Alokaciju memorije
  - Operacije sa znakovima i stringovima
- Standardna zaglavlja (*header files*)
  - Uniformirani pristup deklaraciji funkcija i tipova podataka

Jezik je opisan u Kernighan-Ritchie (*The C Programming Language*). I jezik i operativni sustav Unix su se brzo proširili '70-ih i '80-ih godina prošlog stoljeća. ANSI je standardizirao C 1989. godine postroženjem osnovnih pravila jezika (gramatika) — više grešaka se otkloni pri prevođenju.

Drugo izdanje Kernighan-Ritchieja opisuje taj standard, ANSI C. No, 1990. godine ga je usvojio i ISO (tzv. ISO C). Skraćeno taj ANSI/ISO standard zovemo **C90** (njega mi koristimo).

ISO je 1999. prihvatio novi C standard (manje dopune u odnosu na C90), zove se C99 (tek tu propisana podrška za IEEE 754-1985). Kasnije: C11 i C17/C18, iduće se očekuje C23.

Uvod u C

Postupak programiranja u programskom jeziku C na raznim operacijskim sustavima je vrlo sličan:

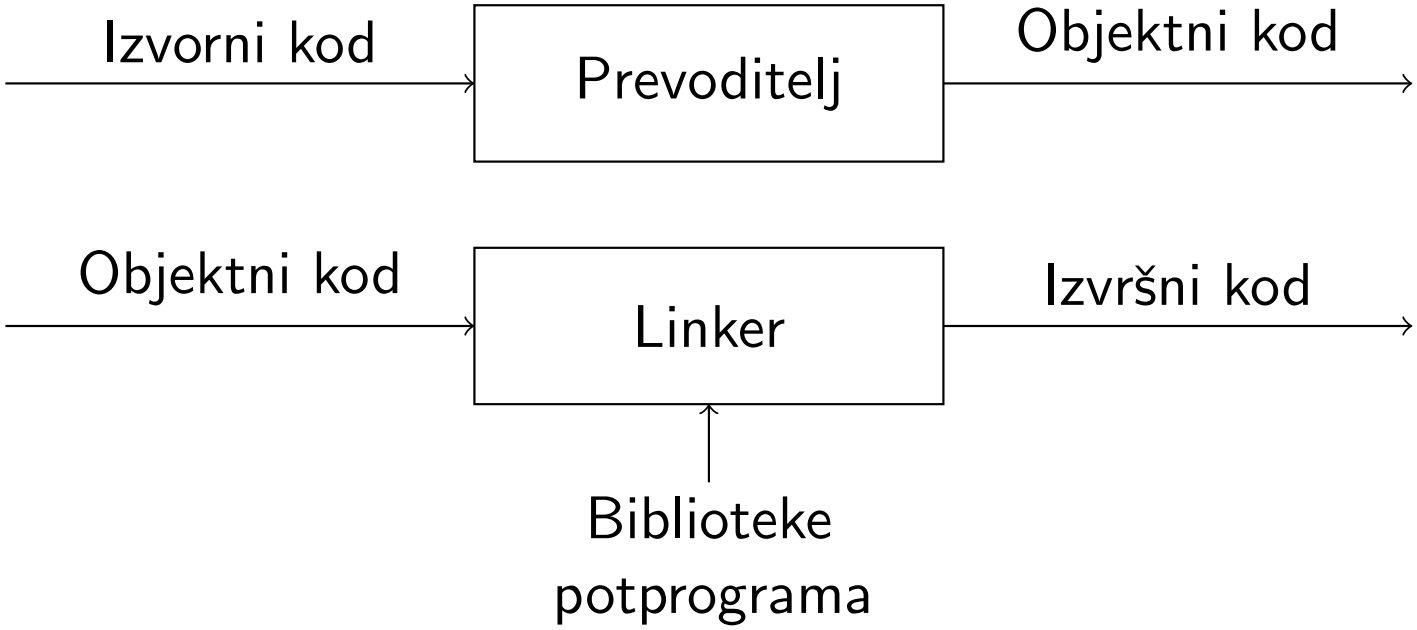
- Pomoću omiljenog programa za unos teksta napišemo *izvorni kod* (tekst) programa i spremimo ga u jednu ili više datoteka.
  - Datoteke će imati ekstenziju .c (ili .h za zaglavlja).
  - Primjer za jednu datoteku (više datoteka kasnije): prvi.c
- Pozovemo program koji prevede izvorni kod u izvršni kod.
  - Taj program se zove *C prevoditelj* ili *kompajler* (eng. *compiler*).
  - Prevoditelj proizvede .out datoteku na Unixu/Linuxu, .exe na Windowsima.
- Pozovemo datoteku koju je prevoditelj proizveo kako bi se izvršio naš program.

Naravno, stvari *nikad* ne idu tako glatko:

- Može biti compile-time grešaka (nalazi ih prevoditelj).
- Može biti run-time grešaka (idealno, nalazi ih programer).
- Krpanje greške može dovesti do novih grešaka obaju tipova.
- Nakon dovoljno iteracija, program bi trebao raditi.

Uvod u C

Shematski prikaz s jednim međukorakom:



Program za unos teksta: Code::Blocks (to je samo dio *integrirane razvojne okoline*, eng. *Integrated Development Environment*, IDE)

Prevoditelj: cc ili gcc

Linker: ld (defaultno ga poziva kompajler)

Biblioteke potprograma: uključe se sa `-l` (npr. `-lm` za `math.h`)

Za pomoć: `man` (npr. `man scanf`) ili guglajte.

Uvod u C

Program (preciznije, izvorni kod programa) u C-u je tekst. O tome koji su znakovi dopušteni, kako se oni udružuju u veće sintaksne cjeline poput imena i naredbi te koja je semantika (značenje) tih cjelina ćemo sasvim detaljno govoriti kasnije.

Zasad ugrubo opisujemo globalnu strukturu C programa i prolazimo kroz jednostavne primjere. C program se sastoji od *imenovanih blokova* (*funkcija*).

Blok počinje znakom { i završava znakom }, a sadrži:

- Deklaracije/definicije
- Naredbe
- Neimenovane blokove

Definicija/deklaracija završava znakom `;`.

Svaki C program mora imati funkciju `main`. Izvršavanje programa počinje njenim izvršavanjem. [*Demo na računalu*]

Uvod u C

Programski jezik C koristi sljedeće znakove:

- Velika i mala slova engleske abecede
- Dekadske znamenke
- Specijalne znakove:

```
+  -  *  /  =  %  &  #
!  ?  ^  "  '  ~  \  |
<  >  (  )  [  ]  {  }
:  ;  .  ,  _
```

  - Specijalni znakovi uključuju bjeline, ali ih je teško vidjeti otisnute. To su razmak (*blank*), horizontalni i vertikalni tabulator te znakovi za prijelaz u novi red, na novu stranicu i vraćanje na početak reda.

Blank i znak za novi red su *separatori* (riječi ili drugih jezičnih cjelina). Višak separatora izvan stringova se ignorira.

Uvod u C

Komentari se pišu između para znakova `/*` i para znakova `*/`, višelinijski su i preskaču se pri prevođenju.

```
/*
  Ovo je komentar.
  Moze se protezati
  kroz vise linija.
*/
```

Komentari se očito ne mogu ugnježđivati (zašto?).

C99 dopušta jednolinijske komentare poput onih u C++-u (od para znakova `//` do kraja linije).

```
// I ovo je komentar u kasnijim standardima jezika C.
```

Uvod u C

Identifikatori su imena koja pridružujemo elementima programa, npr. varijablama, poljima i funkcijama. Pravila njihove strukture:

- Sastoje se od underscorea (_) te alfanumeričkih znakova (slova i znamenki), ali ne počinju znamenkom.
- Velika i mala slova se razlikuju.
- Dozvoljena duljina identifikatora varira, ali prevoditelj nije dužan razlikovati identifikatore koji su isti na prvih 6–63 znakova (ovisno o standardu i namjeni identifikatora).

Ključne riječi već imaju značenje, pa ne mogu biti identifikatori:

`auto`	`break`	`case`	`char`
`const`	`continue`	`default`	`do`
`double`	`else`	`enum`	`extern`
`float`	`for`	`goto`	`if`
`int`	`long`	`register`	`return`
`short`	`signed`	`sizeof`	`static`
`struct`	`switch`	`typedef`	`union`
`unsigned`	`void`	`volatile`	`while`

Uvod u C

Jako detaljno smo govorili o osnovnim tipovima u C-u, pa ovdje samo navodimo neke dodatne sitnice.

Operator[8] `sizeof` daje broj bajtova rezerviranih za prikaz vrijednosti odgovarajućeg tipa. Vrijedi:

$$\texttt{sizeof(short)} \leq \texttt{sizeof(int)} \leq \texttt{sizeof(long)} \text{ te}$$

C90 propisuje sljedeće:

$$\texttt{sizeof(int)} \geq 2, \quad \texttt{sizeof(long)} \geq 4$$

Postoje i neki ugrađeni tipovi o kojima nismo govorili (jer se rijetko koriste), recimo `signed char` (raspon: -128 do 127).

[8] Iako se poziva kao svaka druga funkcija, nećemo reći da je funkcija: naime, argument mu je ime tipa.

Konstante i varijable

Konstante i varijable su osnovni operandi u jeziku *C*.

Svi oni imaju ime, vrijednost i tip. Ime varijable je identifikator, a konstante njena vrijednost.

Cjelobrojne konstante mogu biti zapisane u bazi 10, 8 ili 16:

$$254 = 0376 = \texttt{0xFe}$$

$$\texttt{-}425 = \texttt{-}0651 = \texttt{-}0\texttt{X}1a9$$

$$0 = 00 = \texttt{0x}0$$

Primijetite da zapis u bazi 10 ne smije imati višak vodećih nula! Također, konstante 17 i 17. nisu istog tipa (prva je tipa `int`, a druga tipa `double`). Jednake su pri usporedbi, ali nisu iste pri dijeljenju!

U zapisu u bazi 16, slova `x` te `a`,`b`,`c`,`d`,`e`,`f` smiju biti i mala i velika (u bilo kojoj kombinaciji).

Konstante i varijable

Prethodne cjelobrojne konstante su sve bile tipa `int` (oznaka konverzije pri formatiranom unos/ispisu: `%d`). Konstante ostalih cjelobrojnih tipova oblikujemo dodavanjem slova na kraj (*sufiks*).

	`L`		`long`	(`%ld`)
Ako je sufiks	`U`	, konstanta je tipa	`unsigned`	(`%u`) .
	`UL`		`unsigned long`	(`%lu`)

Slova `U` i `L` mogu biti velika i mala (u bilo kojoj kombinaciji, odnosno poretku). Naravno, `unsigned` konstante ne smiju imati negativan predznak.

Primjer:

```
500000U
123456789L
123456789ul
123456789LU
01234561
0X50000U
```

Konstante i varijable

Znakovne konstante (tipa `char`) se zapisuju kao jedan znak u jednostrukim navodnicima `'`. Primjerice, `'A'`, `'x'`, `'5'`, `'?'`, `' '`.

Sve vrijednosti tipa `char`, pa tako i znakovne konstante, prikazuju se kao cijeli brojevi bez predznaka (npr. u ASCII kodu).

Stoga ih možemo zapisati i preko koda (`\???` oct, `\x??` hex):

$$\texttt{'x'} = 120 = \texttt{'\textbackslash 170'} = \texttt{'\textbackslash x78'}$$

Posebni znakovi (napisani bez jednostrukih navodnika):
`\b`, `\f`, `\n`, `\r`, `\t`, `\v`, `\0`, `\?`, `\"`, `\'`, `\\`, `\a`

Zadnji znak je izvorno slan kako bi zazvonio malim elektromehaničkim zvoncem na teleprinterima i time obavijestio rukovatelja da stiže poruka.

Konstante i varijable

Niz znakova koji završava *nul-znakom* \0 zove se *string*.

Zapis stringovne konstante počinje i završava dvostrukim navodnicima (nul-znak se ne navodi!):

`"Zagreb"`, `"01/07/2021"`, `"Linija 1\nLinija 2\nLinija 3"`

Primijetite, `'a'` nije isto što i `"a"`.

Dva načina za napisati string predug za jedan red koda:

- Završiti sve osim zadnje linije sa \
- Navesti više stringovnih konstanti jednu za drugom

```
char s1[] = "Vrlo dugacak \
niz znakova";

char s2[] = "Vrlo dugacak "
            "niz znakova";
```

Konstante i varijable

Realna konstanta je broj zapisan u dekadskom sustavu koji:

- Sadrži decimalnu točku i/ili
- eksponent baze 10 (tada točka nije potrebna)

Eksponent je cijeli broj kojem prethodi slovo e (veliko ili malo).

Primjer: `0.`, `1.`, `-0.2`, `5000.0`

Primjer: `300000.`, `3e5`, `3E+5`, `3.0e+5`, `.3e6`, `30E4`

Sve te konstante su tipa `double`, za druge tipove treba sufiks.

Ako je sufiks `F` / `L`, realna konstanta je tipa `float` / `long double`.

Slova `F` i `L` mogu biti velika i mala.

`3.f`, `0.337f`, `-3.e4f`, `3e-3f`, `1.345E-8F`, `-0.2e3l`, `5000.0L`

Oznake konverzije za `scanf`: `%g` za float, `%lg` za double, `%Lg` za long double.

Oznake konverzije za `printf`: `%g` za double (float se uvijek pretvara u double), `%Lg` za long double.

Konstante i varijable

Simboličke konstante su imena koja pretprocesor doslovno zamjenjuje zadanim tekstom (osim u stringovnim konstantama). Definiraju se najčešće na početku programa i olakšavaju čitljivost.

```
#define ime tekst
```

Primjer:

```
#define PI 3.141593
#define TRUE 1
#define FALSE 0
```

Ovo prvo je jako loša ideja. **Uvijek** koristite π u punoj točnosti realnog tipa s kojim računate:

```
#include <math.h>
const double pi = 4.0*atan(1.0);
```

(Može i `#define PI (4.0*atan(1.0))`, ali onda moraju ići navedene vanjske zagrade! Zašto? A efikasnost?)

Konstante i varijable

Varijable su simbolička imena za lokacije u memoriji na koje možemo pohraniti neke vrijednosti. Imaju adresu i sadržaj.

Osnovni tipovi varijabli:

- Numerički
- Znakovni
- Pokazivači (sadrže adrese lokacija u memoriji)

Ime varijable je identifikator (duljina može biti ograničena). Dobro je izbjegavati imena koja počinju znakom _ kako ne bi došlo do kolizije sa sistemskim ili internim imenima!

Primjer: `x`, `y2`, `rez_mjerenja`, `Program_V03`, `__SYSTEM`

Primjer: ~~`"x"`~~, ~~`ad-c`~~, ~~`extern`~~, ~~`3x`~~

Konstante i varijable

Deklaracija varijable određuje ime i tip varijable. Ima oblik:

```
tip ime;
tip ime1, ime2, ime3;
```

Varijable se mogu i *inicijalizirati* pri deklaraciji:

```
tip ime = izraz;
```

Primjer:

```
int a, b;
unsigned c;
char d = '\t';
short e;
short f;
double g=-3.5, h=g+1;
```

Konstante i varijable

Varijabla ima adresu i vrijednost. Kad ju deklariramo, operativni sustav na određenoj adresi u memoriji rezervira (*alocira*) prostor za vrijednost varijable. Vrijednost se definira tek kad varijablu inicijaliziramo (pri deklaraciji ili kasnije).

```
int a = 7, b;
```

Varijabla `a` je deklarirana i inicijalizirana. Njezina adresa je `&a`, a vrijednost joj je 7. Varijabla `b` je deklarirana, ali ne i inicijalizirana (mogli bismo reći da `&b` ima vrijednost, ali `b` nema). Kvalifikator `const` zabranjuje mijenjanje vrijednosti već pri prevođenju.

```
unsigned c;
char d = '\t';
short e;
short f;
double g=-3.5, h=g+1;
const char i = '\n';
```

Polja

Polje duljine `n` je konačan niz varijabli istog tipa, indeksiranih cjelobrojnim indeksom od `0` do `n-1`. Deklaracija je oblika:

```
tip ime[dimenzija];
```

Elementi polja su `a[0]`, ..., `a[n-1]`, svaki od njih tipa `tip`.

Primjer:

```
float vektor[10];
```

Mogu se inicijalizirati pri deklaraciji:

```
char a1[] = {'I', 'n', 'i', 't'};
```

Ponekad slične inicijalizacije možemo pisati jednostavnije:

```
char a2[] = "Init";
```

Imaju li `a1` i `a2` jednake elemente?

Pokazivači

Pokazivači su varijable čije vrijednosti su adrese (varijabli zadanog tipa):

```
tip *ime;
```

Zvjezdica označava da varijabla nije tipa `tip`, već tipa `tip *` (pokazivač na `tip`). Čitamo unazad („`ime` je pokazivač na `tip`").

Pažljivo! Ovo su različite deklaracije:

```
double *pu, *u;
double* pu, u;
```

Primjer:

```
float u = 7.5f, *pu = &u, v = *pu;
```

Znak `&` je unarni *operator adrese*. Do vrijednosti spremljene na adresi dolazimo unarnim *operatorom dereferenciranja* `*`.

Enumeracije

Enumeracijama odjednom uvodimo nekoliko simboličkih konstanti:

```
enum { FALSE, TRUE };
```

Efekt će biti isti kao:

```
#define FALSE 0
#define TRUE 1
```

Ako navedemo ime nakon `enum`, definirat ćemo *pobrojani tip*:

```
enum bool { FALSE, TRUE };
```

Sada možemo deklarirati varijable tipa `bool`:

```
enum bool x, y = FALSE;
```

Možemo i mijenjati defaultne vrijednosti:

```
enum kleene3 { TRUE = 't', FALSE = 'f', UNDEF = 'u' };
enum kleene3v2 { UNDEF = -1, FALSE, TRUE };
```

Ključna riječ typedef

Tipove možemo (pre)imenovati pomoću ključne riječi typedef.

Opći oblik typedef deklaracije je:

```
typedef stari_tip novi_tip;
```

Primjer:

```
typedef float real;
real a, b;
enum bool { FALSE, TRUE };
typedef enum bool logical;
logical x, y, flag;
typedef enum { FALSE, TRUE } boolean;
boolean z;
```

Operandi i operatori

Velik dio izvršavanja programa svodi se na računanje vrijednosti raznih izraza u programu. Tipična mjesta pojave izraza su:

- Desna strana naredbe pridruživanja
- Argument funkcije
- Uvjet u grananjima i petljama
- Granice u petljama

Ugrubo opisujemo pravila pisanja izraza (inače najkompliciraniji dio gramatike jezika C).

Svaki izraz ima tip i vrijednost, a formira se od operanada i operatora.

Operand je vrijednost nekog tipa. S jezične strane, može biti:

- Konstanta
- Varijabla
- Vrijednost funkcije
- Podizraz, itd.

Operandi i operatori

Operatori djeluju na operande određenih tipova i daju vrijednost nekog tipa kao rezultat. Zasad se ne bavimo operatorima vezanim za strukture i pokazivače.

Osnovni operator pridruživanja je =. Prioritet mu je niži od većine ostalih operatora (, je izuzetak). Zato je čest oblik naredbe pridruživanja upravo sljedeći:

```
varijabla = izraz;
```

Primjer:

```
x = 3.17;
y = x + 5.342;
a = a + 1;
c = 'm';
```

Operandi i operatori

Aritmetički operatori: `+ - * / %`

Napomena: aritmetički operatori + i - mogu biti i unarni i binarni.

Operandi mogu biti:

- Cjelobrojnog tipa
- Realnog tipa
- Znakovnog tipa

Kojeg tipa je rezultat ako operandi nisu istog tipa? Dolazi do automatske konverzije ili pretvaranja (*implicit conversion*) tipova.

Primjerice, ako su oba operanda cjelobrojna, / daje cjelobrojni rezultat. Ako je barem jedan operand realnog tipa, dijeljenje je uobičajeno dijeljenje realnih brojeva (`3.0/2` je `1.5`).

Operator % (modulo) djeluje samo na cjelobrojnim operandima na način koji smo detaljno opisali ranije.

Operandi i operatori

Vratimo se na konverzije tipova. One su automatske:

- U aritmetičkim izrazima, kad neka operacija djeluje na operande različitog tipa.
- U operaciji pridruživanja, ako tip lijeve strane nije isti kao tip desne strane.
- Pri prijenosu argumenata u funkciju, ako su odgovarajući parametar funkcije i pozivni argument različitog tipa.
- Pri prijenosu vrijednosti iz funkcije na mjesto poziva, ako je povratna vrijednost različitog tipa od deklariranog.

Operandi i operatori

Kod aritmetičkih operatora, operand ,,užeg" tipa se u pravilu promovira u ,,širi tip", pa se izvrši operacija i rezultat je šireg tipa. No, ako je širi tip cjelobrojni **sa** predznakom, a uži **bez** i uži ne stane u nenegativni dio šireg tipa, idu u još širi tip bez predznaka (ali: `long`=`int` ⇒ `long` i `unsigned` idu u `unsigned` ~~`long`~~).

Redom po širini (od najšireg) tipovi idu ovako: `long double`, `double`, `float`, `unsigned long long`, `long long`, `unsigned long`, `long`, `unsigned`, `int`, `char`/`short`. Pripazite: `-1L` > `1UL`.

Pretvaranje se radi onim redom kojim se računaju operacije u izrazu. **Nemojte miješati redoslijed izračunavanja i prioritet.**

Za izraze `A`, `B`, `C`, vrijednost od `A+B*C` se može izračunati ovako:

- Izračuna se `A`, pa `B`, pa `C`, pa `B*C`, pa `A+B*C`.
- Izračuna se `A`, pa `C`, pa `B`, pa `B*C`, pa `A+B*C`.
- Izračuna se `B`, pa `C`, pa `B*C`, pa `A`, pa `A+B*C`, itd.

No, pri izračunavanju `A || B` se nužno najprije izračuna `A`. Zašto?

Operandi i operatori

Kod pridruživanja se desna strana konvertira u tip operanda na lijevoj strani. Time se širi tip može konvertirati u uži i izgubiti informacija (npr. realni tip se pretvori u cjelobrojni).

Primjer:

```
double x = 2.9;
float y = 3.0f;
int z;
z = x + y;
```

Ovdje se dogode sljedeće konverzije:

- Prije zbrajanja, `y` se pretvori u `double`.
- Zatim se pri pridruživanju rezultat zbrajanja pretvori u `int` (zaokruži se prema nuli, odnosno odbaci se decimalni dio — to dvoje **nije** isto kao zaokruživanje na najbliži cijeli broj!).

Operandi i operatori

Pri prijenosu argumenata u funkciju, konverzije ovise o tome ima li funkcija prototip:

- Ako nema, onda se svaki argument tipa `char` i `short` konvertira u `int`, a `float` u `double`.
- Ako ima, onda se svaki argument pri pozivu pretvara u tip deklariran u prototipu.

Primjer:

```
void f(float);
...
f(2.4);
```

Funkciji šaljemo konstantu 2.4 (koja je tipa

Operandi i operatori

Pri prijenosu argumenata u funkciju, konverzije ovise o tome ima li funkcija prototip:

- Ako nema, onda se svaki argument tipa `char` i `short` konvertira u `int`, a `float` u `double`.
- Ako ima, onda se svaki argument pri pozivu pretvara u tip deklariran u prototipu.

Primjer:

```
void f(float);
...
f(2.4);
```

Funkciji šaljemo konstantu 2.4 (koja je tipa `double`), a prototip funkcije deklarira parametar tipa

Operandi i operatori

Pri prijenosu argumenata u funkciju, konverzije ovise o tome ima li funkcija prototip:

- Ako nema, onda se svaki argument tipa `char` i `short` konvertira u `int`, a `float` u `double`.
- Ako ima, onda se svaki argument pri pozivu pretvara u tip deklariran u prototipu.

Primjer:

```
void f(float);
...
f(2.4);
```

Funkciji šaljemo konstantu 2.4 (koja je tipa `double`), a prototip funkcije deklarira parametar tipa `float`, dakle

Operandi i operatori

Pri prijenosu argumenata u funkciju, konverzije ovise o tome ima li funkcija prototip:

- Ako nema, onda se svaki argument tipa `char` i `short` konvertira u `int`, a `float` u `double`.
- Ako ima, onda se svaki argument pri pozivu pretvara u tip deklariran u prototipu.

Primjer:

```
void f(float);
...
f(2.4);
```

Funkciji šaljemo konstantu 2.4 (koja je tipa `double`), a prototip funkcije deklarira parametar tipa `float`, dakle dolazi do konverzije iz tipa `double` u tip `float`. Funkcija zapravo radi s `2.4f`. Općenito, pri ovakvim konverzijama može doći do gubitka točnosti.

Operandi i operatori

Možemo i eksplicitno pretvoriti vrijednost u željeni tip:

```
(tip) izraz
```

Gornji operator nazivamo *cast operator* ili *operator eksplicitne konverzije tipa* (unarni je, po jedan za svaki tip). Prvo se računa vrijednost izraza, a onda se radi konverzija. Važno: cast operator ima visoki prioritet (pa će `izraz` često trebati biti u zagradama).

Primjer:

```
printf("%g", (double)3/2);
printf("%g", (double)(3/2));
```

Operandi i operatori

Možemo i eksplicitno pretvoriti vrijednost u željeni tip:

```
(tip) izraz
```

Gornji operator nazivamo *cast operator* ili *operator eksplicitne konverzije tipa* (unarni je, po jedan za svaki tip). Prvo se računa vrijednost izraza, a onda se radi konverzija. Važno: cast operator ima visoki prioritet (pa će `izraz` često trebati biti u zagradama).

Primjer:

```
printf("%g", (double)3/2);
printf("%g", (double)(3/2));
```

Donja eksplicitna konverzija ne radi ono što želimo: prekasno stiže u pomoć. No, u jednom smislu bilo bi još gore da je nema — `printf("%g", 3/2)` ima nedefinirano ponašanje (UB)!

Operandi i operatori

Primjer:

```
double x;
int i, j;
...
j = (int)(i + x) % 2;
```

Ova eksplicitna konverzija je

Operandi i operatori

Primjer:

```
double x;
int i, j;
...
j = (int)(i + x) % 2;
```

Ova eksplicitna konverzija je nužna. Bez nje bismo dobili grešku pri prevođenju: operator % prima cjelobrojne operande, a vrijednost od `i + x` je tipa `double`.

Prototip funkcije `sqrt` (drugi korijen, zaglavlje `<math.h>`) je:

```
double sqrt(double);
```

Ako je `n` varijabla tipa `int`, možemo pisati (je li cast nužan?):

```
x = sqrt((double) n);
```

Operandi i operatori

Operator inkrementiranja ++ i operator dekrementiranja -- su unarni operatori koji mijenjaju vrijednost numeričke varijable (dakle može biti i pokazivač). Susreli smo ih kod `for` petlji.

Operator ++ poveća vrijednost[9] varijable za 1, a -- ju smanji za 1.

```
++x; /* ekvivalentno sa: x = x + 1; */
```

Osim *prefiksne*, ovi operatori imaju i *postfiksnu* verziju. Razlika je kod složenih izraza: `++x` vraća **novu**, a `x++` **staru** vrijednost od `x`.

```
int x = 3, y = ++x, z = x++;
printf("%d %d %d", x, y, z);
```

Oprez: nemojte varijablu na koju djeluje ++ ili -- ponavljati u istom izrazu. Izrazi `a[i++] = i++` te `i = i++` nisu samo nejasni, već i nemaju propisanu semantiku (opet imamo UB)!

[9] Kod pokazivača je nešto drugačije: vrijednost se poveća ili smanji za veličinu tipa na koji pokazivač pokazuje (tzv. *aritmetika pokazivača*).

Operandi i operatori

Operator inkrementiranja ++ i operator dekrementiranja -- su unarni operatori koji mijenjaju vrijednost numeričke varijable (dakle može biti i pokazivač). Susreli smo ih kod `for` petlji.

Operator ++ poveća vrijednost[9] varijable za 1, a -- ju smanji za 1.

```
++x; /* ekvivalentno sa: x = x + 1; */
```

Osim *prefiksne*, ovi operatori imaju i *postfiksnu* verziju. Razlika je kod složenih izraza: `++x` vraća **novu**, a `x++` **staru** vrijednost od `x`.

```
int x = 3, y = ++x, z = x++;
printf("%d %d %d", x, y, z); /* ispis: 5 4 4 */
```

Oprez: nemojte varijablu na koju djeluje ++ ili -- ponavljati u istom izrazu. Izrazi `a[i++] = i++` te `i = i++` nisu samo nejasni, već i nemaju propisanu semantiku (opet imamo UB)!

[9] Kod pokazivača je nešto drugačije: vrijednost se poveća ili smanji za veličinu tipa na koji pokazivač pokazuje (tzv. *aritmetika pokazivača*).

Operandi i operatori

Operator `sizeof` je unarni operator čiji je operand izraz[10] (u tom slučaju vraća veličinu njegovog tipa u bajtovima; sam izraz se ne izračuna) ili ime tipa (vraća veličinu tipa čije je to ime). Ako je operand ime tipa, mora biti zatvoren u zagrade.

```
int i;
printf("%u %u", sizeof i, sizeof(int));
```

C90 propisuje da je `sizeof(char)` jednak 1.

```
char tekst[] = "Program";
printf("Broj znakova u varijabli tekst = %zu\n",
        sizeof(tekst));
```

Operator `sizeof` vraća cjelobrojnu vrijednost bez predznaka koja ovisi o implementaciji (zato `%zu` umjesto `%u`). Pripadni tip `size_t` (tip za veličine/duljine objekata) definiran je u `<stddef.h>`.

[10] Preciznije, `unary_expression`: ugrubo, sadrži samo unarne operatore.

Operandi i operatori

Operatore s kojima smo se dosad upoznali možemo ubaciti u korisnu *tablicu prioriteta* (puno operatora ćemo još dopisati):

Kategorija	Operatori	Asociranost
unarni	`++ -- + - * & (tip) sizeof`	D→L
ar. multiplikativni	`* / %`	L→D
ar. aditivni	`+ -`	L→D
pridruživanje	`=`	D→L

Primjerice, dodat ćemo šest relacijskih operatora (u dvije razine prioriteta): <, <=, >, >=, ==, !=. Prva četiri su većeg prioriteta od posljednja dva (ali nižeg od prioriteta aritmetičkih i unarnih operatora). Svi ti operatori su lijevo asocirani (L→D). Zadnja dva relacijska operatora nazivamo *operatorima jednakosti*.

```
int a = 1, b = 20, limit = 100;
rezultat = (a + b) >= limit;
```

Operandi i operatori

Kategorija	Operatori	Asociranost
unarni	`++ -- + - * & (tip) sizeof`	D→L
ar. multiplikativni	`* / %`	L→D
ar. aditivni	`+ -`	L→D
relacijski	`< <= > >=`	L→D
rel. jednakost	`== !=`	L→D
pridruživanje	`=`	D→L

Relacijskim operatorima formiraju se *logički izrazi*. Njihova vrijednost je istina (1) ili laž (0). C90 nema poseban logički tip, pa je vrijednost logičkih izraza tipa `int` (i u C11, ali tamo postoji i poseban tip `_Bool`).

Primjer: Za `i=1, j=2, k=4` imamo

Izraz	Istinitost	Vrijednost
`i < j`	istinito	1
`(i + j) >= k`	neistinito	0
`i == 2`	neistinito	0
`k != i`	istinito	1

Operandi i operatori

Prioritet relacijskih operatora niži je od prioriteta aritmetičkih:

```
i >= 'A' - 'a' + 1
```

je izraz ekvivalentan sa

```
i >= ('A' - 'a' + 1)
```

Ponekad je lakše čitati izraz s više zagrada, poput `(i + j) >= k`.

Složeniji logički izrazi tvore se pomoću *logičkih operatora*: `!` (unarna logička negacija), `&&` (logičko „i") te `||` (logičko „ili"). Cjelobrojni operandi različiti od nule se interpretiraju kao istina, a nula kao laž. Vrijednost složenog logičkog izraza je 0 ili 1.

Unarni operator `!` je u tablici prioriteta gdje i ostali unarni operatori (asociranost je, naravno, ista za cijelu razinu prioriteta).

Binarni logički operatori imaju niži prioritet od relacijskih i aritmetičkih operatora. Asocirani su uobičajeno, dakle L→D. Operator `&&` je višeg prioriteta od `||`.

Operandi i operatori

Primjer:

Ako je `i > 1` i `c == 't'` istinito, a `j < 6` lažno, onda je:

Izraz	Istinitost	Vrijednost
`i > 1 || j < 6`	istinito	1
`i > 1 && j < 6`	neistinito	0
`!(i > 1)`	neistinito	0
`i > 1 && (j < 6 || c != 't')`	neistinito	0

Primjer:

```
int a = 0, b = 10, c = 100, d = 200;
printf("%d%d%d", !(c < d), (a - b) && 1, d || b && a);
```

Primijetimo dvije stvari: budući da je `if (!istinito)` isto što i `if (istinito == 0)`, prvi oblik radije koristimo za logičke testove, a drugi za numeričke (npr. test parnosti).

Zatim, koja je vrijednost (sintaksno ispravnog) izraza 3>2>1?

Operandi i operatori

Kategorija	Operatori	Asociranost
unarni	`! ++ -- + - * & (tip) sizeof`	D→L
ar. multiplikativni	`* / %`	L→D
ar. aditivni	`+ -`	L→D
relacijski	`< <= > >=`	L→D
rel. jednakost	`== !=`	L→D
logičko „i”	`&&`	L→D
logičko „ili”	`||`	L→D
pridruživanje	`=`	D→L

Skraćeno izračunavanje logičkih izraza nam može pomoći u pisanju vrlo elegantnih programa.

Primjer:

```
char x[128];
/* inicijalizacija niza x */
for (i = 0; i < 128 && x[i] != 'a'; ++i) { ... }
/* sto ako zamijenimo poredak uvjeta? */
```

Operandi i operatori

Operatori nad bitovima (*bitwise operators*) mogu se primijeniti na cjelobrojne podatke, a djeluju na bitove u prikazu:

Operator	Značenje
~	jedinični komplement (negacija bit po bit)
<<	lijevi pomak bitova
>>	desni pomak bitova
&	logičko ,,i" bit po bit
^	ekskluzivno logičko ,,ili" bit po bit
|	logičko ,,ili" bit po bit

Prvi operator je unarni, ostali su binarni. Nemojte zamijeniti logički operator (npr. ||) s njegovom bit po bit verzijom (npr. |).

Primjer (za $n = 16$; ostali operatori DZ):

```
      a  =  0x0003   /* = 0000 0000 0000 0011 */
      b  =  0x0009   /* = 0000 0000 0000 1001 */
  a & b  =  0x0001   /* = 0000 0000 0000 0001 */
```

Operandi i operatori

Operatori pomaka (*shift*) << i >> pomiču binarni zapis broja u lijevo ili u desno (ideja: množenje/dijeljenje potencijom od 2).

Operandi moraju biti cjelobrojnog tipa (uz promociju kratkih tipova) te nenegativni (inače rezultat ovisi o implementaciji). Štoviše, drugi operand ne smije biti negativan ili premašiti broj bitova u prvom operandu — inače rezultat nije definiran!

Nad prvim operandom se vrši operacija i rezultat ima njegov tip (nakon eventualne početne promocije), a drugi operand je broj bitova pomaka (i njegov tip se prvo promovira, ako treba).

Prioritet operatora << i >> je isti, ispod aritmetičkih aditivnih, iznad relacijskih. Asociranost je L→D.

Primjer (za $n = 16$):

```
      a   =   0x60ac   /* = 0110 0000 1010 1100 */
  a << 6  =   0x2b00   /* = 0010 1011 0000 0000 */
-------------------------------------------------
      a   =   0x60ac   /* = 0110 0000 1010 1100 */
  a >> 6  =   0x0182   /* = 0000 0001 1000 0010 */
```

Operandi i operatori

Bitovni logički operatori najčešće služe tzv. *maskiranju* (prekrivanju ili filtriranju) pojedinih bitova u operandu:

- Operator `&` postavlja na 0 bitove koji su 0 u masci,
- Operator `|` postavlja na 1 bitove koji su 1 u masci,
- Operator `^` nalazi mjesta gdje operandi imaju različite bitove,
- Operator `~` služi promjeni $0 \leftrightarrow 1$.

Ostali bitovi se ne mijenjaju, tj. "propuštaju se kroz filter".

Primjer (postavljanje 10. najmanje značajnog bita, a_9, na nulu):

```
unsigned mask = ~(1 << 9);
a = a & mask;
```

Primjer (b postanu same jedinice osim $b_5 := a_5, \ldots, b_0 := a_0$):

```
b = a | ~0x3f;
```

Zašto ne `0xffffffc0` umjesto `~0x3f`?

Operandi i operatori

Kategorija	Operatori	Asoc.
unarni	! ~ ++ -- + - * & (tip) sizeof	D→L
ar. multiplikativni	* / %	L→D
ar. aditivni	+ -	L→D
op. pomaka	<< >>	L→D
relacijski	< <= > >=	L→D
rel. jednakost	== !=	L→D
bitovni ,,i"	&	L→D
bitovni eks. ,,ili"	^	L→D
bitovni ,,ili"	|	L→D
logičko ,,i"	&&	L→D
logičko ,,ili"	||	L→D
pridruživanje	=	D→L

Operandi i operatori

Vratimo se još malo operatoru pridruživanja. Njegov prioritet je nizak da bi naredba pridruživanja oblika `varijabla = izraz;` prvo izračunala izraz na desnoj strani, a onda tu vrijednost pridružila varijabli na lijevoj strani operatora.

No, pridruživanje `varijabla = izraz` je ujedno izraz. Vrijednost tog izraza je vrijednost varijable na lijevoj strani (*nakon* što se izvrši pridruživanje).

Primjer:

```
while ((a = x[i]) != 0) {
        ...
        ++i;
}
```

Operandi i operatori

Čest je slučaj da *feature* dovede do (teško uočljivog) *buga*.

Primjer:

```
if (varijabla = izraz) ...;
```

umjesto

```
if (varijabla == izraz) ...;
```

U prvoj `if` naredbi:

Operandi i operatori

Čest je slučaj da *feature* dovede do (teško uočljivog) *buga*.

Primjer:

```
if (varijabla = izraz) ...;
```

umjesto

```
if (varijabla == izraz) ...;
```

U prvoj `if` naredbi: prvo se vrijednost izraza pridruži varijabli, a zatim se izvršava tijelo `if` naredbe ako je varijabla različita od nule.

U drugoj `if` naredbi:

Operandi i operatori

Čest je slučaj da *feature* dovede do (teško uočljivog) *buga*.

Primjer:

```
if (varijabla = izraz) ...;
```

umjesto

```
if (varijabla == izraz) ...;
```

U prvoj `if` naredbi: prvo se vrijednost izraza pridruži varijabli, a zatim se izvršava tijelo `if` naredbe ako je varijabla različita od nule.

U drugoj `if` naredbi: vrijednost varijable se ne mijenja, već se samo uspoređuje s vrijednošću izraza. Tijelo `if` naredbe se izvršava ako su te dvije vrijednosti jednake.
Asociranost operatora pridruživanja je zdesna nalijevo, D→L, pa ih ima smisla ulančati (uočite da se `izraz` samo jednom računa):

```
a = b = c = izraz;
```

Operandi i operatori

Složeni operatori pridruživanja su:

- `+= -= *= /= %=` (binarni aritmetički, pa =)
- `<<= >>= &= ^= |=` (binarni bitovni, pa =)

Općenito, izraz oblika `izraz1 op= izraz2` gdje je op jedna od operacija `+ - * / % << >> & ^ |` je ekvivalentan sa `izraz1 = izraz1 op (izraz2)`. Ovi složeni operatori pridruživanja su na istoj razini prioriteta s osnovnim operatorom pridruživanja =, pa su asocirani zdesna nalijevo, D→L.

Uvjetni izraz je izraz oblika `izraz1 ? izraz2 : izraz3`. Operator `? :` je *ternarni operator* (ima tri operanda). Prvo se izračuna `izraz1`; ako je istinit, vrijednost uvjetnog izraza je vrijednost od `izraz2` (`izraz3` se ne računa). Ako je lažan, uzima se vrijednost od `izraz3` (`izraz2` se ne računa). Prioritet ovog operatora je nizak, odmah iznad = (asociranost je isto D→L).

```
min_ab = (a < b) ? a : b;
```

Operandi i operatori

Operator zarez `,` separira dva izraza; izračuna se prvi izraz, izračuna se drugi izraz te se vrati vrijednost drugog izraza.

Primjer:

```
i = (i = 3, i + 4); /* na kraju i ima vrijednost   */
```

Operator zarez se uglavnom koristi u `for` naredbi (kasnije).
Prioritet operatora `,` je niži od =, asociranost L→D.

Upotpunimo tablicu prioriteta *primarnim operatorima*:

- `()` (poziv funkcije),
- `[]` (pristup elementima polja),
- `.` (pristup članovima strukture),
- `->` (pristup članovima strukture preko pokazivača).

Primarni operatori su grupa s najvišim prioritetom, a asociranost im je uobičajena, L → D.

Operandi i operatori

Operator zarez `,` separira dva izraza; izračuna se prvi izraz, izračuna se drugi izraz te se vrati vrijednost drugog izraza.

Primjer:

```
i = (i = 3, i + 4); /* na kraju i ima vrijednost 7 */
```

Operator zarez se uglavnom koristi u `for` naredbi (kasnije).
Prioritet operatora `,` je niži od `=`, asociranost L→D.

Upotpunimo tablicu prioriteta *primarnim operatorima*:

- `()` (poziv funkcije),
- `[]` (pristup elementima polja),
- `.` (pristup članovima strukture),
- `->` (pristup članovima strukture preko pokazivača).

Primarni operatori su grupa s najvišim prioritetom, a asociranost im je uobičajena, L → D.

Operandi i operatori (potpuna tablica prioriteta)

Kategorija	Operatori	Asoc.
primarni	`() [] -> .`	L→D
unarni	`! ~ ++ -- + - * & (tip) sizeof`	D→L
ar. multiplikativni	`* / %`	L→D
ar. aditivni	`+ -`	L→D
op. pomaka	`<< >>`	L→D
relacijski	`< <= > >=`	L→D
rel. jednakost	`== !=`	L→D
bitovni ,,i"	`&`	L→D
bitovni eks. ,,ili"	`^`	L→D
bitovni ,,ili"	`|`	L→D
logičko ,,i"	`&&`	L→D
logičko ,,ili"	`||`	L→D
uvjetni	`? :`	D→L
pridruživanje	`= += -= *= /= %= &= ^= |= <<= >>=`	D→L
operator zarez	`,`	L→D

Izrazi i naredbe

Već smo puno toga rekli o izrazima i naredbama, ali sad ćemo ih malo detaljnije obraditi. Rad programa opisan je nizom naredbi. Prema formalnoj gramatici jezika C, postoji 6 vrsti naredbi.

Jednostavne naredbe počinju izrazom i završavaju znakom ;.

```
x = 3;
++n;
printf(...);
x+y; 17; ; /* besmisleno, ali dozvoljeno */
```

Složene naredbe redom sadrže otvorenu vitičastu zagradu, bilo koji broj deklaracija i naredbi, pa zatvorenu vitičastu zagradu.

```
{x = 3;
++n;
printf(...);}
```

Složena naredba je sintaksno ekvivalentna jednoj naredbi, tj. može se pojaviti gdje i jednostavna naredba.

Izrazi i naredbe

Najjednostavniji oblik `if` naredbe je sljedeći:

```
if (uvjet) naredba
```

gdje je uvjet aritmetički, odnosno logički izraz.

Redoslijed izvršavanja:

1. Računa se vrijednost izraza `uvjet`.
2. Ako ta vrijednost nije nula (tj. istina je), izvrši se `naredba`.
3. Inače (jest nula, odnosno laž je), `naredba` se ne izvrši i program nastavlja izvršavanje prvom sljedećom naredbom.

```
if (i > j) {
        temp = i;
        i = j;
        j = temp;
}
```

Možemo li jednostavnije napisati zamjenu?

Izrazi i naredbe

Naredba `if` može imati i pripadnu `else` klauzulu:

```
if (uvjet)
        naredba1
else
        naredba2
```

Semantika (značenje): izvrši točno jednu naredbu — ako je `uvjet` istina, onda naredbu `naredba1`, inače `naredba2`.

Postoji jedna poteškoća u sintaksnoj analizi (*parsiranju*). Kad ugnijezdimo `if` naredbe, kojoj od njih pripada `else`?

```
if (n > 0)
        if (a > b)
                z = a;
        else
                z = b;
```

```
if (n > 0)    /* lose */
        if (a > b)
                z = a;
else
                z = b;
```

C:

Izrazi i naredbe

Naredba `if` može imati i pripadnu `else` klauzulu:

```
if (uvjet)
        naredba1
else
        naredba2
```

Semantika (značenje): izvrši točno jednu naredbu — ako je `uvjet` istina, onda naredbu `naredba1`, inače `naredba2`.

Postoji jedna poteškoća u sintaksnoj analizi (*parsiranju*). Kad ugnijezdimo `if` naredbe, kojoj od njih pripada `else`?

```
if (n > 0)
        if (a > b)
                z = a;
        else
                z = b;
```

```
if (n > 0)    /* lose */
    if (a > b)
        z = a;
else
        z = b;
```

C: bližoj! Intendirano značenje desno:

Izrazi i naredbe

Naredba `if` može imati i pripadnu `else` klauzulu:

```
if (uvjet)
        naredba1
else
        naredba2
```

Semantika (značenje): izvrši točno jednu naredbu — ako je `uvjet` istina, onda naredbu `naredba1`, inače `naredba2`.

Postoji jedna poteškoća u sintaksnoj analizi (*parsiranju*). Kad ugnijezdimo `if` naredbe, kojoj od njih pripada `else`?

```
if (n > 0)
        if (a > b)
                z = a;
        else
                z = b;
```

```
if (n > 0) { /* okej */
    if (a > b)
        z = a; }
else
        z = b;
```

C: bližoj! Intendirano značenje desno: dodatkom vitica.

Izrazi i naredbe

U `<stdlib.h>` deklarirana je funkcija `exit` (ugrubo) ovako:

```
void exit(int status)
```

Poziv te funkcije u naredbi `exit(status);` zaustavlja izvršavanje programa i vrijednost `status` predaje operacijskom sustavu (isto je kao kod `return` u funkciji `main`, ali `exit` možemo pozvati iz bilo koje funkcije).

U `<stdlib.h>` definirane su i dvije standardne vrijednosti za `status`: `EXIT_SUCCESS` i `EXIT_FAILURE` (0 i `EXIT_SUCCESS` znače uspješan, a `EXIT_FAILURE` neuspješan završetak programa).

```
if (x == 0) {
printf("Greska: djelitelj jednak nuli!\n");
exit(EXIT_FAILURE); /* treba #include <stdlib.h> */
} else y /= x;
```

Izrazi i naredbe

Sljedeće dvije naredbe su ekvivalentne:

```
max = a >= b ? a : b;
```

```
if (a >= b)
        max = a;
else
        max = b;
```

Obje postavljaju `max` na maksimum vrijednosti varijabli `a` i `b`.

Može li se `if` s prošlog slajda zapisati pomoću uvjetnog operatora?

Napomena: `if...else` naredba se može ugnježđivati:

```
if (uvjet1) naredba1
else if (uvjet2) naredba2
else naredba3
```

Izrazi i naredbe

Naredba switch slična je nizu ugniježdenih if-else naredbi:

```
switch (izraz) {
case konstanta_1: naredbe_1 /* moze vise naredbi! */
case konstanta_2: naredbe_2
...
case konstanta_n: naredbe_n
default:          naredbe
}
```

Ideja: izračunaj vrijednost od `izraz`; ako je jednaka nekoj od konstanti, izvrši naredbe pripadnog slučaja, inače slučaja `default`.

Detaljnije, `izraz` mora imati cjelobrojnu vrijednost (`char`, `int` ili `enum`). Nakon svake ključne riječi `case` ide cjelobrojna konstanta ili konstantni izraz[11] (mora biti različit/a od ostalih), pa dvotočka.

[11] Ovi izrazi se računaju prilikom prevođenja, pa čak `const` varijable nisu dopuštene; no zato prevoditelj zna javiti grešku ako vrijednosti nisu jedinstvene!

Izrazi i naredbe

Najčešća upotreba naredbe `switch` na kraju svakog slučaja koji nije posljednji ima `break`. U protivnom, počeo bi se izvršavati idući slučaj.

```
unsigned int i;
...
switch (i) {
case 0:
case 1:
case 2:  printf("i < 3\n");
         break;
case 3:  printf("i = 3\n");
         break;
default: printf("i > 3\n");
}
```

Ovdje još samo treba spomenuti da `default` ne mora postojati.

Izrazi i naredbe

DZ: Isprogramirajte infiks i postfiks kalkulator koji prima dva `double`-a i osnovnu računsku operaciju (`char`) i ispiše rezultat. Probajte ga napraviti sa i bez `switch`.

Kad ćete pisati kalkulator, možda ćete imati ovakav unos:

```
printf("operand1=");
scanf("%lf", &operand1);
printf("operacija=");
scanf(" %c", &operacija);
printf("operand2=");
scanf("%lf", &operand2);
```

Je li razmak prije %c bitan? Zašto ga nema prije drugog `%lf`?
DZ: Isprobajte program sa i bez razmaka u oba slučaja.

Izrazi i naredbe

`while` petlja ima oblik:

```
while (izraz) naredba
```

Sve dok je `izraz` istinit (različit od 0), naredba se ponavlja.

```
i = 0;
while (i < 10) {
        printf("%d\n", i);
        ++i;
}
```

`while` petlja najčešće se koristi kad se broj ponavljanja ne zna unaprijed, već je pod kontrolom uvjeta `izraz`.

DZ: Napišite program koji učitava realne brojeve dok se ne unese 0 te ispisuje njihovu srednju vrijednost. Na što ovdje treba paziti?

Izrazi i naredbe

for petlja ima oblik:

```
for (izraz_1; izraz_2; izraz_3) naredba
```

i *gotovo* je ekvivalentna s

```
izraz_1;          /* obicno: inicijalizacija brojaca */
while (izraz_2) {  /* provjera vrijednosti brojaca */
        naredba
        izraz_3;   /* inkrementiranje brojaca */
}
```

Naime, bitna razlika je ponašanje `continue` naredbe unutar petlje (kasnije). Srednji izraz `izraz_2` interpretira se kao logički izraz, a ostala dva izraza se pretvaraju u (jednostavne) naredbe.

```
for (brojac = 1; brojac < 5; ++brojac) ...
for (brojac = 1; brojac < 5; brojac += 2) ...
for (;;);
```

Izrazi i naredbe

for petlja ima oblik:

```
for (izraz_1; izraz_2; izraz_3) naredba
```

i *gotovo* je ekvivalentna s

```
izraz_1;          /* obicno: inicijalizacija brojaca */
while (izraz_2) {  /* provjera vrijednosti brojaca */
        naredba
        izraz_3;   /* inkrementiranje brojaca */
}
```

Naime, bitna razlika je ponašanje `continue` naredbe unutar petlje (kasnije). Srednji izraz `izraz_2` interpretira se kao logički izraz, a ostala dva izraza se pretvaraju u (jednostavne) naredbe.

```
for (brojac = 1; brojac < 5; ++brojac) ...
for (brojac = 1; brojac < 5; brojac += 2) ...
for (;;); /* besk. petlja, default za izraz_2: 1 */
```

Izrazi i naredbe

Što se ispiše u sljedećem primjeru?

```
for (brojac = 1; brojac < 5; ++brojac)
        printf("brojac = %d\n", brojac);
```

Izrazi i naredbe

Što se ispiše u sljedećem primjeru?

```
for (brojac = 1; brojac < 5; ++brojac)
        printf("brojac = %d\n", brojac);
```

Ispiše se 1,2,3,4. A ovdje?

```
for (brojac = 1; brojac < 5; ++brojac);
        printf("brojac = %d\n", brojac);
```

Izrazi i naredbe

Što se ispiše u sljedećem primjeru?

```
for (brojac = 1; brojac < 5; ++brojac)
        printf("brojac = %d\n", brojac);
```

Ispiše se 1,2,3,4. A ovdje?

```
for (brojac = 1; brojac < 5; ++brojac);
        printf("brojac = %d\n", brojac);
```

Ispiše se samo 5.

Razlog:

Izrazi i naredbe

Što se ispiše u sljedećem primjeru?

```
for (brojac = 1; brojac < 5; ++brojac)
        printf("brojac = %d\n", brojac);
```

Ispiše se 1,2,3,4. A ovdje?

```
for (brojac = 1; brojac < 5; ++brojac);
        printf("brojac = %d\n", brojac);
```

Ispiše se samo 5.

Razlog: `printf` je u prvom primjer unutar petlje (prema indentaciji, čini se da je postignuta intencija), a u drugom izvan petlje (pa je tijelo petlje prazno, izgleda suprotno intenciji).

Izrazi i naredbe

do-while petlja ima oblik:

```
do
        naredba
while (izraz);
```

`naredba` se izvršava sve dok je `izraz` istinit, tj. vrijednost mu je različita od nule. Za razliku od `while` petlje, gdje se vrijednost izraza računa i provjerava na početku petlje, u `do-while` se to događa nakon prolaza kroz petlju (petlja se izvrši barem jednom).

```
int x;
do {
        printf("x=");
        scanf("%d", &x);
} while (x <= 0);
```

Jesmo li mogli ovdje upotrijebiti `while` petlju?

Izrazi i naredbe

Naredba break služi za:

- izlazak iz switch naredbe
- i prekid izvršavanja petlje bilo koje vrste.

Pri nailasku na naredbu `break`, kontrola toka se prenosi na prvu naredbu iza (najbliže okolne!) `switch` naredbe ili petlje unutar koje se taj `break` nalazi.

```
int i;
while (1) {
        scanf("%d", &i);
        if (i < 0) break;
        ... /* obrada učitanog broja */
}
```

Ovdje imamo beskonačnu petlju koja služi za učitavanje i obradu nenegativnih brojeva. Čim se učita negativan broj, izvršavanje se nastavlja prvom naredbom iza ove `while` petlje.

Izrazi i naredbe

Naredba `continue` može se koristiti unutar svih vrsta petlji za preskakanje svih kasnijih naredbi u petlji. Program tada nastavlja sa sljedećim prolazom kroz petlju.

Preciznije, sljedeća naredba koja se izvršava je:

- test uvjeta u `while` i `do-while`,
- `izraz_3;` (često je to pomak brojača) u `for`.

Navedeno je najavljena razlika između `for` i pripadne `while` petlje.

Uočite da naredba `continue` nema smisla u `switch` naredbi (za razliku od `break`).

```
int i;
while (1) {
        scanf("%d", &i);
        if (i < 0) continue;
        ... /* obrada učitanog broja */
}
```

Što u ovom slučaju obrada mora sadržavati?

Izrazi i naredbe

Naredba `goto` prekida sekvencijalno izvršavanje programa i nastavlja izvršavanje s naredbom koja ima pripadnu oznaku (tzv. *bezuvjetni skok*).

Pravilo pisanja goto naredbe je:

```
goto label;
```

gdje je `label` identifikator koji služi za označavanje naredbe kojom se nastavlja program. Sintaksa označavanja je:

```
label: naredba
```

Oznaka na koju se vrši skok mora biti unutar iste funkcije kao i `goto` naredba, tj. pomoću `goto` se ne može „izaći iz funkcije".

Izrazi i naredbe

Naredba goto se može koristiti za reakcije na greške:

```
double x, s = 0.0;
while (1) {
        scanf("%lg", &x);
        if (x < 0.0) goto error;
        if (x == 0.0) break;
        s += sqrt(x);          /* zbraja korijene */
}
...        /* normalni zavrsetak posla */
error:
        /* reakcija na gresku */
        printf("Greska: negativan broj!\n");
        exit(EXIT_FAILURE);
```

Izrazi i naredbe

Još jedna upotreba je za izlazak iz više ugniježđenih petlji:

```
while (...)
        for (...)
                for (...) {
                        scanf("%d", &x);
                        if (x < 0) goto van;
                        ... /* obrada broja */
                }
van: ...
```

DZ: Za vježbu[12], pomoću `goto` simulirajte rad naredbi `break` i `continue` u sva tri tipa petlje (i isto to za `break` u `switch`-u).

[12] ... koja nije nužno najmetodičnija na predmetu koji vas podučava programiranju u višem programskom jeziku. Više na idućem slajdu.

Izrazi i naredbe

Napomena. Program koji ima puno naredbi `goto` bitno je teže pročitati i razumjeti od programa koji ne koristi `goto`. Zato upotrebu te naredbe treba izbjegavati.

Ugrubo, naredba `goto` služi samo za reakciju na greške te izlaz iz više ugniježđenih petlji odjednom.

Može biti korisna i za preskakanje ogromnih dijelova programa *prema naprijed* (jer indentacija tad ne pomaže povećanju čitljivosti).

No, nije dobro koristiti `goto` za skok *prema natrag*. Time izlazite iz paradigme višeg programskog jezika i simulirate petlju kao što biste činili u assembleru! Prevoditelj čini puno korisnih stvari kako vi ne biste morali i to je svakako jedna od njih.

Osnovni algoritmi na cijelim brojevima

Cilj ove točke: konstrukcija, implementacija i analiza jednostavnih (osnovnih) algoritama sastavljenih od jedne petlje i nekoliko uvjetnih naredbi, na jednostavnim podacima (cijelim brojevima)

Kasnije ćemo iste ili slične algoritme koristiti na složenijim podacima (nizovi, vezane liste i sl.)

Ulazi će nam biti tipa `unsigned` ili `int`, ovisno o potrebama. Algoritme ćemo realizirati u cjelobrojnoj aritmetici (realnu aritmetiku izbjegavamo zbog mogućih grešaka pri zaokruživanju).

Oprez: neovisno o tipu podataka koji koristimo, skup prikazivih brojeva u računalu je konačan, a aritmetika cijelih brojeva je modularna. To je bitno pri izboru algoritma: naime, on treba raditi za što veći skup ulaznih podataka, idealno za svaki prikaziv ulazni podatak.

Osnovni algoritmi na cijelim brojevima

Primjer. Program treba učitati cijeli broj n (tipa int) i naći broj dekadskih znamenki tog broja.

Najlakše je

Osnovni algoritmi na cijelim brojevima

Primjer. Program treba učitati cijeli broj n (tipa int) i naći broj dekadskih znamenki tog broja.

Najlakše je brisati znamenke zdesna (i usput ih brojati). Naredba za brisanje znamenke bit će

Osnovni algoritmi na cijelim brojevima

Primjer. Program treba učitati cijeli broj `n` (tipa int) i naći broj dekadskih znamenki tog broja.

Najlakše je brisati znamenke zdesna (i usput ih brojati). Naredba za brisanje znamenke bit će `n /= 10;`. Stajemo ako je

Osnovni algoritmi na cijelim brojevima

Primjer. Program treba učitati cijeli broj n (tipa int) i naći broj dekadskih znamenki tog broja.

Najlakše je brisati znamenke zdesna (i usput ih brojati). Naredba za brisanje znamenke bit će `n /= 10;`. Stajemo ako je `n==0`.

Npr. za $\mathtt{n} = 123$ ćemo triput podijeliti s 10 prije nego dođemo do 0; pri brisanju ćemo inkrementirati brojač:

- `n /= 10` daje $\mathtt{n} = 12$, a broj obrisanih znamenki je 1.
- `n /= 10` daje $\mathtt{n} = 1$, a broj obrisanih znamenki je 2.
- `n /= 10` daje $\mathtt{n} = 0$, a broj obrisanih znamenki je 3.

Naravno, brojač je na početku 0 jer tad još nijedna znamenka nije obrisana. Čak i za $\mathtt{n} = 1$ dobit ćemo ispravan odgovor.
A za $\mathtt{n} = 0$? A za negativne brojeve?

Osnovni algoritmi na cijelim brojevima

```
#include <stdio.h>

/* Broj dekadskih znamenki cijelog broja. */

int main(void)
{
        int n, broj_znam = 0;
        printf("n=");
        scanf("%d", &n);

        while (n != 0) {
                ++broj_znam;
                n /= 10;
        }
        printf(" Broj znamenki = %d\n", broj_znam);
        return 0;
}
```

Osnovni algoritmi na cijelim brojevima

Primijetimo sljedeće:

- Vrijednost od `n` je na kraju izvršavanja uvijek

Osnovni algoritmi na cijelim brojevima

Primijetimo sljedeće:

- Vrijednost od `n` je na kraju izvršavanja uvijek 0. Drugim riječima, algoritam uništava ulazni podatak (*destruktivan* je). Ako to nećemo, moramo napraviti ,,backup" vrijednosti varijable `n` (u npr. varijablu `n_orig`). Ili možemo napraviti i uništavati kopiju od `n` (npr. `n_temp`).
- Složenost ovog algoritma je

Osnovni algoritmi na cijelim brojevima

Primijetimo sljedeće:

- Vrijednost od `n` je na kraju izvršavanja uvijek 0. Drugim riječima, algoritam uništava ulazni podatak (*destruktivan* je). Ako to nećemo, moramo napraviti ,,backup" vrijednosti varijable `n` (u npr. varijablu `n_orig`). Ili možemo napraviti i uništavati kopiju od `n` (npr. `n_temp`).
- Složenost ovog algoritma je linearna u veličini (broju znamenki) ulaza. Drugim riječima, $\Theta(\mathtt{n})$.

U nastavku prelazimo na nenegativne brojeve kako ne bismo morali razmišljati o predznaku. Oznaka konverzije za `unsigned` je `%u`. Konstante tog tipa se pišu s nastavkom (sufiksom) `u`, npr. `0u`.

U programima koji slijede namjerno je ispušten `u` zbog čitljivosti (bitni dijelovi svih njih rade i za podatke tipa `int`).

Osnovni algoritmi na cijelim brojevima

```
int main(void)
{
        unsigned int b = 10, n, broj_znam = 0;
        printf("n=");
        scanf("%u", &n);
        printf("\n Broj %u", n);
        while (n > 0) {
                ++broj_znam;
                n /= b;
        }
        printf(" ima %u znamenki u bazi %u\n",
                broj_znam, b);
        return 0;
}
```

Osnovni algoritmi na cijelim brojevima

Ako je $n \in \mathbb{N}$ i ako je

$$n = a_k \cdot b^k + a_{k-1} \cdot b^{k-1} + \cdots + a_1 \cdot b + a_0$$

normalizirani prikaz tog broja u bazi b, tj. vrijedi $a_0, \ldots, a_k \in \{0, 1, \ldots, b-1\}$ i $a_k > 0$, onda broj znamenki, odnosno $k+1$, možemo izračunati i direktno preko logaritma i funkcije ,,najveće cijelo" (zapravo cast u `int` ili `unsigned`):

$$k + 1 = \lfloor \log_b n \rfloor + 1.$$

Međutim, to zahtijeva realnu aritmetiku, a ona ima greške zaokruživanja (problematično je ako se log zaokruži na dolje).

U zaglavlju `<math.h>` postoje dvije funkcije za logaritam: `log` (ln) i `log10` ($\log_{10}$). Logaritam u bazi b računamo vrlo jednostavno:

$$\log_b n = \frac{\ln n}{\ln b} = \frac{\log_{10} n}{\log_{10} b}.$$

Osnovni algoritmi na cijelim brojevima

Greške zaokruživanja ovise o konkretnoj C biblioteci koja stiže uz prevoditelj. Primjerice, `gcc` radi sljedeće (za format `%.15lf`):

- `log` ne radi ispravno za $n = 10^6$ (6 umjesto 7 znamenki!):

  `log(1000000)/log(10)=5.999999999999999`

- ne radi ni za $n = 3^5$ i bazu 3 (5 umjesto 6 znamenki!):

  `log(243)/log(3)=4.999999999999999`

Umjesto *top-down*, destruktivnog brisanja znamenki straga, možemo ići *bottom-up*: potenciraj bazu.

Treba paziti s uvjetom zaustavljanja: nije dobro *sve dok je broj n veći ili jednak od potencije baze*! Zašto?

Osnovni algoritmi na cijelim brojevima

Greške zaokruživanja ovise o konkretnoj C biblioteci koja stiže uz prevoditelj. Primjerice, `gcc` radi sljedeće (za format `%.15lf`):

- `log` ne radi ispravno za $n = 10^6$ (6 umjesto 7 znamenki!):

  `log(1000000)/log(10)=5.999999999999999`

- ne radi ni za $n = 3^5$ i bazu 3 (5 umjesto 6 znamenki!):

  `log(243)/log(3)=4.999999999999999`

Umjesto *top-down*, destruktivnog brisanja znamenki straga, možemo ići *bottom-up*: potenciraj bazu.

Treba paziti s uvjetom zaustavljanja: nije dobro *sve dok je broj n veći ili jednak od potencije baze*! Zašto? Ako je n blizu najvećeg prikazivog broja, prva veća potencija baze bi mogla biti neprikaziva!

Bolje: dok je kvocijent od n i potencije baze veći ili jednak od baze.

<u>DZ:</u> Napišite program koji odgovara ovom algoritmu. Implementirajte i ispis znamenki *sprijeda*.

Osnovni algoritmi na cijelim brojevima

Ako na neki način želimo obraditi sve znamenke zapisa broja u nekoj bazi, a nije bitno kojim redoslijedom ih obrađujemo, algoritam je sasvim sličan onom za brojanje znamenki.

Primjer. Želimo napisati program koji učitava nenegativni cijeli broj n te ispisuje zbroj znamenki od n u bazi 10. Traženi rezultat je $a_0 + a_1 + \cdots + a_k$. Algoritamski,

$$\texttt{rezultat} = \texttt{rezultat} + a_i,\ i = 0, 1, ..., k.$$

```
suma = 0;
while (n > 0) {
        suma += n % b;
        n /= b;
}
printf(" Suma znamenki u bazi %u je %u\n", b, suma);
```

Suma se inicijalizira na neutralni element za zbrajanje. To je i najprirodniji rezultat sume (svih elemenata) praznog skupa.
DZ: Množenje/min/max znamenki. Inicijalizacija?

Osnovni algoritmi na cijelim brojevima

Najjednostavniji primjeri provjere nekog svojstva u danom skupu odgovaraju standardnim kvantifikatorima u matematici:

- Postoji ($\exists$) li objekt sa zadanim svojstvom?
- Ima li svaki ($\forall$) objekt zadano svojstvo?

Rezultat je odgovor na postavljeno pitanje, tj. rezultat je logičkog (aritmetičkog) tipa.

Primjer. Program treba učitati nenegativni cijeli broj n i odrediti postoji li znamenka tog broja u bazi 10 koja je jednaka 5. Traženi rezultat je $a_0 = 5 \vee a_1 = 5 \vee \cdots \vee a_k = 5$. Algoritamski:

$$\texttt{rezultat} = \texttt{rezultat} \ \texttt{||} \ (a_i \texttt{ == } 5),\ i = 0, 1, \ldots, k.$$

Inicijalizacija za egzistenciju/disjunkciju?

Osnovni algoritmi na cijelim brojevima

Najjednostavniji primjeri provjere nekog svojstva u danom skupu odgovaraju standardnim kvantifikatorima u matematici:

- Postoji ($\exists$) li objekt sa zadanim svojstvom?
- Ima li svaki ($\forall$) objekt zadano svojstvo?

Rezultat je odgovor na postavljeno pitanje, tj. rezultat je logičkog (aritmetičkog) tipa.

Primjer. Program treba učitati nenegativni cijeli broj n i odrediti postoji li znamenka tog broja u bazi 10 koja je jednaka 5. Traženi rezultat je $a_0 = 5 \vee a_1 = 5 \vee \cdots \vee a_k = 5$. Algoritamski:

$$\texttt{rezultat} = \texttt{rezultat} \ \texttt{||} \ (a_i \texttt{ == } 5),\ i = 0, 1, \ldots, k.$$

Inicijalizacija za egzistenciju/disjunkciju? Laž.

Osnovni algoritmi na cijelim brojevima

```
odgovor = 0;
while (n > 0) {
        znam = n % b;
        if (znam == trazena) {
                odgovor = 1;
                break;
        }
        n /= b;
}
```

DZ: Univerzalna svojstva. Inicijalizacija?

Primjer. Program treba učitati nenegativni cijeli broj n i odrediti je li n palindrom u bazi 10. Na primjer, 14741 jest, a 14743 nije.

Osnovni algoritmi na cijelim brojevima

Prvi način: provjerom odgovarajućih znamenki (prva = zadnja, druga = predzadnja, itd.)

```
for (pot = 1; n / pot >= b; pot *= b);
palindrom = 1;
while (n >= b) {
        znam1 = n / pot;
        znam2 = n % b;
        if (znam1 != znam2) {
                palindrom = 0;
                break;
        }
        n %= pot;
        n /= b;
        pot = pot /b/b;
}
```

Osnovni algoritmi na cijelim brojevima

Drugi način: konstrukcijom broja s obratnim poretkom znamenki.

```
#include <stdio.h>
int main(void)
{
        unsigned b = 10, n, m1, m2, palindrom;
        printf("n=");
        scanf("%u", &n);
        m1 = n;
        m2 = 0;
        while (n > 0) {
                m2 = m2 * b + n % b;
                n /= b;
        }
        palindrom = m1 == m2 ? 1 : 0;
        printf(" Palindrom = %u\n", palindrom);
        return 0;
}
```

Osnovni algoritmi na cijelim brojevima

Radi li ovaj program korektno za svaki prikazivi ulaz?

Hint:

Osnovni algoritmi na cijelim brojevima

Radi li ovaj program korektno za svaki prikazivi ulaz?

Hint: Je li obratni broj uvijek prikaziv?

Možemo li zato dobiti pogrešan odgovor u bazi $b = 10$?

Osnovni algoritmi na cijelim brojevima

Radi li ovaj program korektno za svaki prikazivi ulaz?

Hint: Je li obratni broj uvijek prikaziv?

Možemo li zato dobiti pogrešan odgovor u bazi b = 10? Ne!
DZ: Dokažite to!

Zadatak. Neka je baza $b = 2^{30} + 1 = 1073741825$, a broj je $n = b + 5 = 1073741830$. Je li odgovor („da") točan?

Zadatak. Koliko ima baza b (u tipu `unsigned` u 32 bita) kod kojih ovaj algoritam daje pogrešan odgovor za barem jedan n?

Izazov prof. Singera. Program „s podosta matematike prije toga": ukupan broj nađenih baza = 715827889, vrijeme = 0.23 s !

Osnovni algoritmi na cijelim brojevima

Jedan od prvih poznatih algoritama je Euklidov algoritam za nalaženje najveće zajedničke mjere $M(a, b)$ cijelih brojeva a i b, uz pretpostavku da je $b \neq 0$.

Algoritam se bazira na teoremu o dijeljenju s ostatkom:

$$|a| = |b| \cdot q + r,$$

za neke $q, r \in \mathbb{Z}$ gdje je $0 \leq r < |b|$.

Ključni koraci: Ako $d \mid |a|$ i $d \mid |b|$, onda $d \mid r$, pa je $M(a, b) = M(|a|, |b|) = M(|b|, r)$ (smanjujemo argumente). Ako je $r = 0$, onda je $|a| = |b| \cdot q$, pa je $M(a, b) = |b|$.

<u>DZ:</u> Probajte za npr. a = 48, b = 36 ili a = 21, b = 13. Isto tako, probajte izvršiti algoritam s negativnim brojevima!

Napisat ćemo ključni dio programa na dva načina (drugi je malo kraći). <u>DZ:</u> Može li bez %?

Osnovni algoritmi na cijelim brojevima

```
a = abs(a); b = abs(b); /* ide #include <stdlib.h> */
while (1) {
        ostatak = a % b;
        if (ostatak == 0) { mjera = b; break; }
        a = b;
        b = ostatak; }
```

```
a = abs(a); b = abs(b); /* ide #include <stdlib.h> */
while (b) {
        ostatak = a % b;
        a = b;
        b = ostatak; }
mjera = a;
```

Može i rekurzivno (gradivo Prog2), u „jednom” retku:

Osnovni algoritmi na cijelim brojevima

```
a = abs(a); b = abs(b); /* ide #include <stdlib.h> */
while (1) {
        ostatak = a % b;
        if (ostatak == 0) { mjera = b; break; }
        a = b;
        b = ostatak; }
```

```
a = abs(a); b = abs(b); /* ide #include <stdlib.h> */
while (b) {
        ostatak = a % b;
        a = b;
        b = ostatak; }
mjera = a;
```

Može i rekurzivno (gradivo Prog2), u „jednom” retku:

```
int gcd(int a, int b) {
 return b ? gcd(abs(b), abs(a) % abs(b)) : abs(a); }
```

Osnovni algoritmi na cijelim brojevima

Primjer. Program treba učitati prirodni broj n i odrediti je li n potencija broja d, tj. je li n oblika d^k za neki $k \in \mathbb{N}_0$.

Za zadani $d \geq 2$, svaki $n \in \mathbb{N}$ možemo jednoznačno prikazati kao

$$n = d^k \cdot m, \quad d \nmid m,\ k \in \mathbb{N}_0.$$

DZ: Dokažite to! Hint: kao faktorizacija, ali d nije nužno prost.

Algoritam (ideja): Dijelimo n sa d dokle ide (k puta). Tada će od n ostati m. Dakle, n je potencija od d akko na kraju vrijedi $m = 1$.

```
while (n % d == 0)
        n /= d;
odgovor = n == 1;
```

DZ: Napišite program koji prikazuje broj tipa `int` kao niz bitova.

```
Prikaz broja -3:
1111 1111 1111 1111 1111 1111 1111 1101
```

Funkcije

Funkcija je programska cjelina koja

- prima neke ulazne podatke,
- izvršava određeni niz naredbi,
- i vraća rezultat svog izvršavanja na mjesto poziva.

Slično kao u matematici: domena, pravilo pridruživanja, kodomena.

Definicija funkcije ima oblik:

```
tip ime_funkcije(tip_1 par_1, ..., tip_n par_n)
{
        tijelo_funkcije
}
```

`tip` je tip podatka povratne vrijednosti funkcije (kodomena funkcije). `ime_funkcije` je identifikator. Zatim, unutar okruglih zagrada nalazi se deklaracija *parametara* funkcije (ako ih ima). Prvi parametar `arg_1` je lokalna varijabla tipa `tip_1`, drugi parametar `arg_2` je lokalna varijabla tipa `tip_2`, itd.

Funkcije

```
tip ime_funkcije(tip_1 par_1, ..., tip_n par_n)
{
        tijelo_funkcije
}
```

Tipovi parametara opisuju domenu funkcije (koja je njihov Kartezijev produkt). Deklaracije pojedinih parametara međusobno se odvajaju zarezom (to nije operator `,`).

Dio definicije funkcije koji smo zasad opisali katkad se još zove i *zaglavlje funkcije*. Okrugle zagrade moraju se napisati (čak i kad nema[13] parametara) jer signaliziraju prevoditelju da je riječ o funkciji.

Iza zaglavlja dolazi tijelo funkcije. Piše se unutar vitičastih zagrada i ima strukturu bloka, odnosno složene naredbe.

[13] Tada treba napisati `void` u zagrade; u protivnom će prevoditelj misliti da je definirana funkcija s varijabilnim brojem argumenata.

Funkcije

Svaki blok ili složena naredba u programu sastoji se od deklaracija (objekata poput varijabli ili tipova) i naredbi koje se izvršavaju ulaskom u blok. Isto vrijedi i za tijelo funkcije.

Za bilo koji blok, pa tako i za tijelo funkcije, vrijede sljedeća pravila o redoslijedu deklaracija i naredbi (detaljnije na Prog2):

- C90: sve deklaracije moraju prethoditi prvoj naredbi.
- C99: deklaracija svakog objekta mora prethoditi prvom korištenju tog objekta.

Funkciju pozivamo s konkretnom vrijednosti, odnosno *argumentom* za `x` u kojem želimo izračunati vrijednost funkcije. Na primjer, u pozivu sin(2.35), argument je 2.35.

Pri pozivu funkcije, njeni parametri postaju lokalne varijable (postoje sve dok funkcija ne završi izvršavanje). Razlika izmedu tih varijabli i ostalih varijabli deklariranih u tijelu funkcije: parametri se inicijaliziraju prilikom poziva funkcije iz argumenata s kojima je funkcija pozvana.

Funkcije

Bitno je reći kojih sve tipova može biti povratna vrijednost funkcije.

Funkcija može vratiti aritmetički tip, strukturu, uniju, ili pokazivač, ali ne može vratiti drugu funkciju ili polje. Međutim, može vratiti pokazivač na funkciju ili na polje (prvi element polja).

Ako `tip` nije naveden (što je dozvoljeno), pretpostavlja se da funkcija vraća podatak tipa int. Nemojte to koristiti — prevoditelj to radi samo zbog kompatibilnosti s prastarim C programima (pisanim u „prvotnom" C-u prema Kernighan-Ritchie, prije ANSI/ISO standarda).

Naredba `return` služi za vraćanje rezultata izvršavanja funkcije. Opći oblik te naredbe je `return izraz;`. Ako je tip vrijednosti izraza u naredbi `return` različit od tipa podatka koji funkcija vraća, vrijednost izraza bit će pretvorena u `tip`. Naredba return, ujedno, završava izvršavanje funkcije.

Funkcije

Izvršavanje programa nastavlja se tamo gdje je funkcija bila pozvana, a povratna se vrijednost (ako je ima) ,,uvrštava" na mjesto poziva funkcije.

```
double y, phi=2.35, r=6.81;
y = r * sin(phi);
```

Trigonometrijske funkcije postoje u standardnoj biblioteci. Pripadna datoteka zaglavlja je `<math.h>`.

Ako funkcija vraća neku vrijednost, povratna vrijednost se ne mora iskoristiti. Primjerice, standardne funkcije `scanf` i `printf` također vraćaju neku vrijednost (više na sljedećem predavanju). Uobičajeni pozivi tih funkcija odbacuju povratnu vrijednost.

```
scanf("%d", &n);
printf("n = %d\n", n);
```

Funkcije

Ako nam povratne vrijednosti trebaju, možemo napisati

```
procitano = scanf("%d", &n);
napisano = printf("n = %d\n", n);
```

Primjer. Sljedeća funkcija pretvara mala slova engleske abecede u velika. Ostale znakove ne mijenja. Parametar je samo jedan (c) i tipa je `char`. Vraćena vrijednost je tipa char. Ime funkcije je `malo_u_veliko` (ekvivalent u `<ctype.h>`: `toupper`).

```
char malo_u_veliko(char c)
{
 char znak;
 znak = ('a' <= c && c <= 'z') ? ('A' + c - 'a') : c;
 return znak;
}
```

```
char malo_u_veliko(char c)
{ return 'a' <= c && c <= 'z' ? 'A' + c - 'a' : c; }
```

Funkcije

Funkcija se poziva navođenjem imena funkcije i popisa argumenata u zagradama.

```
veliko = malo_u_veliko(slovo);
```

Ovdje je varijabla `slovo` jedini argument u pozivu funkcije. Trenutna vrijednost te varijable se prenosi u funkciju, kao početna vrijednost parametra `c`.

```
int main(void)
{
        char malo, veliko;
        printf("Unesite malo slovo: ");
        scanf("%c", &malo);
        veliko = malo_u_veliko(malo);
        printf("Veliko slovo = %c\n", veliko);
        return 0;
}
```

Funkcije

Pri pozivu funkcije, argument je izraz čija se vrijednost računa i prenosi u funkciju, odnosno pridružuje se parametru. Redoslijed izračunavanja argumenata nije definiran standardom i ovisi o implementaciji.

```
double x=1.5, y=0.1415926;
rezultat = sin(2 * x + y);
veliko = malo_u_veliko('a' + 3);
veliko = malo_u_veliko(veliko + 3);
```

Funkcija može imati više od jedne `return` naredbe (primjerice u raznim `if` granama).

```
char malo_u_veliko(char c) {
        if ('a' <= c && c <= 'z')
                return ('A' + c - 'a');
        else
                return c;
}
```

Funkcije

Ako funkcija ne vraća nikakvu vrijednost, onda se za tip povratne vrijednosti koristi ključna riječ `void` (eng. prazan).

```
void ispisi_max(int x, int y)
{
        int max;
        max = (x >= y) ? x : y;
        printf(" Maksimalna vrijednost = %d\n", max);
}
```

Naravno, onda nema smisla pisati:

```
m = ispisi_max(x, y);
```

Prevoditelj bi trebao javiti grešku (gcc: *error: void value not ignored as it ought to be*). Ako se to ne dogodi, nabavite novog prevoditelja (vrijednost od `m` će biti nepredvidiva pri izvođenju).

Funkcije

Funkcija koja ne prima nikakve argumente definira se ovako:

```
tip ime_funkcije(void)
{
        tijelo_funkcije
}
```

Ključna riječ `void` (unutar zagrada) označava da funkcija nema parametara, npr. `main` (no može se omogućiti da `main` prima argumente). Zatim, funkcija `getchar` za čitanje jednog znaka (sa standardnog ulaza) nema parametara (v. sljedeće predavanje). Poziv takve funkcije ima praznu listu argumenata u zagradama.

```
varijabla = ime_funkcije();
```

Zagrade () su obavezne, jer informiraju prevoditelja da je identifikator `ime_funkcije` ime funkcije.

Funkcije

Pitanje na koje sad želimo odgovoriti: kako se funkcije ,,spajaju" u cijeli program te kojim se redom pišu?

Svaka bi funkcija, prije prvog poziva u programu, trebala biti deklarirana — navođenjem tzv. *prototipa*. Deklaracija informira prevoditelja o:

- imenu funkcije,
- broju i tipu argumenata te
- tipu povratne vrijednosti funkcije.

Mogućnost da se funkcija ne deklarira prije poziva ostavljena je samo zbog kompatibilnosti s prastarim C programima (u jeziku C++ čak više ne postoji). Stoga nemojte koristiti tu mogućnost!

Ako je funkcija definirana u istoj datoteci u kojoj se poziva (tako će biti do Prog2), i to prije prvog poziva te funkcije, onda njena definicija služi i kao deklaracija. Inače ju treba posebno deklarirati.

Funkcije

```c
#include <stdio.h>

void ispisi_max(double x, double y)
{
        double max;
        max = x >= y ? x : y;
        printf("Maksimalna vrijednost = %g\n", max);
}

int main(void)
{
        double x, y;
        printf("Unesite dva realna broja: ");
        scanf("%lg %lg", &x, &y);
        ispisi_max(x, y);
        return 0;
}
```

Funkcije

Ako definiciju funkcije smjestimo nakon poziva funkcije, onda tu funkciju moramo deklarirati prije prvog poziva. Deklaracija ili prototip funkcije ima oblik:

```
tip ime_funkcije(tip_1 par_1,..., tip_n par_n);
```

Dakle, deklaracija sadrži samo zaglavlje funkcije, bez bloka u kojem je tijelo funkcije. Imena parametara mogu biti izostavljena, jer se njihovi tipovi vide i bez toga:

```
tip ime_funkcije(tip_1, ..., tip_n);
```

Neke deklaracije mogu se spojiti slično kao kod varijabli.

```
int n, f(double), g(int, double);
```

U ovoj deklaraciji, `n` je varijabla tipa `int`, a `f` i `g` su funkcije koje vraćaju vrijednost tipa `int`. Obično se deklaracija piše na početku datoteke ili u funkciji u kojoj je poziv.

Funkcije

```
#include <stdio.h>

int main(void) {
        double x, y;
        void ispisi_max(double, double);
        printf(" Unesite dva realna broja: ");
        scanf("%lg %lg", &x, &y);
        ispisi_max(x, y);
        return 0;
}

void ispisi_max(double x, double y) {
        double max;
        max = (x >= y) ? x : y;
        printf(" Maksimalna vrijednost = %g\n", max);
}
```

Funkcije

```
#include <stdio.h>

void ispisi_max(double, double);

int main(void) {
        double x, y;
        printf(" Unesite dva realna broja: ");
        scanf("%lg %lg", &x, &y);
        ispisi_max(x, y);
        return 0;
}

void ispisi_max(double x, double y) {
        double max;
        max = (x >= y) ? x : y;
        printf(" Maksimalna vrijednost = %g\n", max);
}
```

Funkcije

Općenito u programskim jezicima, postoje dva načina prijenosa argumenata u funkciju prilikom njenog poziva:

- prijenos po vrijednosti (eng. *call by value*) te
- prijenos po adresi (eng. *call by reference*).

Kod prijenosa po vrijednosti, funkcija prima kopije vrijednosti argumenata, što znači da funkcija ne može izmijeniti argumente s kojima je pozvana. Kod prijenosa po adresi, funkcija prima adrese argumenata, pa može izmijeniti argumente (sadržaje na tim adresama); argumenti tada moraju biti varijable.

U C-u postoji samo prijenos argumenata po vrijednosti. Ako funkcijom želimo promijeniti vrijednost nekog podatka (tzv. *varijabilni argument*), pripadni argument treba biti pokazivač na taj podatak, tj. njegova adresa! Tada se adresa prenosi po vrijednosti — kopira se u funkciju, a čitati/mijenjati sadržaj na toj adresi možemo koristeći *operator dereferenciranja* $*$. Naravno, promjena kopije adrese ne mijenja adresu podatka.

Funkcije

```
#include <stdio.h>

void f(int x)
{
        x += 1;
        printf("Unutar funkcije: x = %d\n", x);
        return;
}

int main(void)
{
        int x = 5;
        printf("Prije poziva: x=%d\n", x);
        f(x);
        printf("Nakon poziva: x=%d\n", x);
        return 0;
}
```

Funkcije

```
#include <stdio.h>

void f(int *px)
{
        *px += 1;
        printf("Unutar funkcije: x = %d\n", *px);
        return;
}

int main(void)
{
        int x = 5;
        printf("Prije poziva: x=%d\n", x);
        f(&x);
        printf("Nakon poziva: x=%d\n", x);
        return 0;
}
```

Funkcije

Kod prijenosa po vrijednosti, poziv funkcije `f` u glavnom programu može glasiti ovako:

```
f(x + 2);
```

tj. argument smije biti izraz (štoviše, u C-u je to uvijek slučaj).

Za razliku od toga, kod „prijenosa po adresi", poziv funkcije `f` u glavnom programu ne može biti:

```
f(&(x + 2));
```

jer izraz nema adresu! Ali, smisleno je napisati `f(&x + 2)` — ovdje se koristi *aritmetika pokazivača*!

Funkcije

```
#include <stdio.h>
void f(int x, int y) {
        x += y;
        y += x;
        printf("Unutar fje: x=%d, y=%d\n", x, y);
}

int main(void) {
        int x = 2, y = 3;
        printf("Prije poziva: x=%d, y=%d\n", x, y);
        f(y, x + y);
        printf("Nakon poziva: x=%d, y=%d\n", x, y);
        return 0;
}
```

Funkcije

```
#include <stdio.h>
void f(int *px, int *py) {
        *px += *py;
        *py += *px;
        printf("Unutar fje: x=%d, y=%d\n", *px, *py);
}

int main(void) {
        int x = 2, y = 3, z;
        z = x + y;
        printf("Prije poziva: x=%d, y=%d\n", x, y);
        f(&y, &z);
        printf("Nakon poziva: x=%d, y=%d\n", x, y);
        return 0;
}
```

Funkcije

Iako smo se dogovorili nikad ne koristiti funkcije bez deklaracije, pogledajmo što će se dogoditi ako pozovete funkciju koja nije prethodno deklarirana. Prevoditelj će javiti upozorenje i:

- Prevoditelj pretpostavlja da funkcija vraća podatak tipa `int` i ne pravi nikakve pretpostavke o broju i tipu parametara.
- Na svaki argument cjelobrojnog tipa primjenjuje se integralna promocija (pretvaranje argumenata tipa `short` i `char` u `int`), a svaki argument tipa `float` pretvara se u `double`.
- Broj i tip (pretvorenih) argumenata mora se podudarati s brojem i tipom parametara da bi poziv bio korektan.

```
int main(void) {
        printf("%d\n", f(2));                /*UF!*/
        printf("%d\n", g(2.0f));
        return 0;
}
int f(double x) { return (int) x*x; }
double g(double x) { return (int) x*x; }   /*uf!*/
```

Funkcije

Vratimo se nekim programima otprije i pogledajmo kako možemo prebaciti bitne dijelove tih programa u funkcije.

```
unsigned broj_znamenki(unsigned n, unsigned b)
{
        unsigned broj_znam = 0;
        while (n > 0) {
                ++broj_znam;
                n /= b;
        }
        return broj_znam;
}
```

Funkcije

```
int odgovor(unsigned n, unsigned b, unsigned trazena)
{
        while (n > 0) {
                if (n % b == trazena)
                        return 1;
                n /= b;
        }
        return 0;
}
```

Funkcije

```
int euklid(int a, int b) {
        int ostatak;
        a = abs(a); b = abs(b);
        while (1) {
                ostatak = a % b;
                if (ostatak == 0)
                        return b;
                a = b;
                b = ostatak;
        }
}
```

Sjetimo se da može i kraće (rekurzijom):

```
int gcd(int a, int b) {
 return b ? gcd(abs(b), abs(a) % abs(b)) : abs(a); }
```

Funkcije

```
int odgovor(unsigned n, unsigned d, unsigned *pk) {
        unsigned k = 0;
        while (n % d == 0) {
                ++k;
                n /= d;
        }
        *pk = k;
        return n == 1 && k > 0;
}
```

Funkcije

Već smo vidjeli da C dozvoljava *rekurzivne funkcije*, tj. funkcije koje (unutar vlastite definicije) pozivaju same sebe. U pravilu, rekurzivni algoritmi su kraći, ali izvođenje može[14] trajati značajno dulje. Svaki rekurzivni algoritam mora imati „nerekurzivni dio" (u kojem funkcija ne poziva opet samu sebe) koji omogućava prekidanje rekurzije. Najčešće je to nekakav bazni uvjet.

```
long int fakt(int n) {
        if (n <= 1)
                return 1L;
        else
                return n * fakt(n - 1);
}
```

Iako je lako shvatiti ovu rekurziju, možemo napisati nerekurzivni kod koji ne poziva funkciju `fakt` čak n puta da bi izračunao $n!$.

[14] Jedan način rješavanja tog problema naziva se *dinamičko programiranje*. O tome više na Prog2 i SPA.

Funkcije

Faktorijele možemo izračunati u jednoj petlji. Varijanta sa silaznom petljom po n:

```
long int fakt(int n) {
        long int f = 1L;
        for (; n > 1; --n) f *= n;
        return f;
}
```

Varijanta s uzlaznom petljom do n (trebamo pomoćnu varijablu za faktor u petlji):

```
long int fakt(int n) {
        long int f = 1L;
        int i;
        for (i = 2; i <= n; ++i) f *= i;
        return f;
}
```

Funkcije

Kako bismo mogli rekurzivno ispisati znamenke od n u bazi b?

Funkcije

Kako bismo mogli rekurzivno ispisati znamenke od n u bazi b?

```
#include <stdio.h>
void ispis_u_bazi(unsigned n, unsigned b) {
        if (n <= 0) return;
        ispis_u_bazi(n / b, b);
        printf("%u", n % b);
}
int main(void) {
        unsigned b, n;
        printf("Upisi n i b: ");
        scanf("%u%u", &n, &b);
        printf("Prikaz od %u u bazi %u: ", n, b);
        ispis_u_bazi(n, b);
        printf("\n");
        return 0;
}
```

Funkcije

Zadatak. Ako je $n = 0$, onda naša funkcija ne piše ništa (nema znamenki). Modificirajte funkciju (ne glavni program!) tako da napiše znamenku 0 za $n = 0$. Pripazite da se ne doda vodeća nula kad n nije jednak nuli.

Zadatak. Napravite proširenje na veće baze, tako da znamenke mogu biti i slova (dakle do 36), ili tako da se znamenke u bazi b pišu kao dekadski brojevi (tada ih odvajamo, primjerice, prazninom).

Zanimljiviji primjeri rekurzivnih algoritama i funkcija su:

- quicksort i mergesort algoritmi za sortiranje,
- Hanojski tornjevi,
- particije broja u pribrojnike,
- algoritmi bazirani na obilasku binarnog/uređenog stabla,
- sintaksna analiza programa (pars(ir)anje), itd.

Prva tri primjera obradit ćete na Prog2, četvrti na SPA, a sintaksnu analizu tek na Interpretaciji programa.

Funkcije

Proučimo detaljnije funkcije za ulaz/izlaz (input/output). U standardnoj I/O biblioteci (zaglavlje `stdio.h`) postoje sljedeće:

- `getchar`, `putchar`: za ulaz/izlaz znakova,
- `gets`, `puts`: za ulaz/izlaz stringova,
- `scanf`, `printf`: za formatirani ulaz/izlaz

Funkcije za ulaz rade na standardnom ulazu („datoteka" `stdin`), a za izlaz na standardnom izlazu („datoteka" `stdout`). Zato datoteka nije parametar u tim funkcijama.
Pogledajmo najprije funkcije `getchar` i `putchar`. Njihove deklaracije su sljedećeg oblika:

```
int getchar(void);
int putchar(int c);
```

Funkcija `getchar` čita jedan znak sa standardnog ulaza i vraća ga (zašto je tip `int`?). Učitani znak je prvi znak u ulaznom spremniku (koji je FIFO queue) ako je taj spremnik neprazan, inače `getchar` čeka na ulaz. Primjer poziva: `c_var = getchar();`

Funkcije

Funkcije poput `fgetc` koje učitavaju znakove iz proizvoljne datoteke vrate `EOF` (simbolička konstanta definirana u `stdio.h`, najčešće -1) kad dođu do kraja ulaznih podataka. Slično, kad se ulazni podaci čitaju sa standardnog ulaza, kraj se može utipkati sa Ctrl+D (Unix, Linux) ili Ctrl+Z (Windows).

Funkcija `putchar` ispisuje jedan znak (argument s kojim je funkcija pozvana) na standardni izlaz. Vraća ispisani znak ili `EOF` (ako ispis znaka nije uspio). Primjer poziva: `putchar(c);`

```
#include <stdio.h>
#include <ctype.h>
int main(void) {
        int c;
        while ((c = getchar()) != EOF)
                putchar(toupper(c));
        return 0;
}
```

Funkcije

```
#include <stdio.h>

void f(void) {
        int znak;
        if ((znak = getchar()) != '\n')
                f();
        putchar(znak);
}

int main(void) {
        printf("Unesite niz znakova: ");
        f();
        return 0;
}
```

Funkcije

String je niz znakova koji završava *nul-znakom* '\0'.

```
char niz_znakova[5] = {'h', 'e', 'l', 'l', 'o'};
char string[6] = {'h', 'e', 'l', 'l', 'o', '\0'};
char string[] = "hello";
```

Zapamtite: ime polja je (konstantan!) pokazivač na prvi element polja. Npr. `string` je isto što i `&string[0]`. Pogledajmo funkcije za čitanje i pisanje znakovnih nizova i stringova:

```
char *gets(char *s);
int puts(const char *s);
```

Funkcija `gets` čita znakove sa standardnog ulaza sve do '\n' ili kraja podataka. Nul-znak se učita odmah nakon zadnjeg učitanog znaka. Znak za novi red se ne učita, ali se ukloni iz ulaznog spremnika. Vraća pokazivač na učitani string ili `NULL` (ako se došlo do kraja ulaznih podataka ili se javila greška prilikom čitanja). Simbolička konstanta `NULL` definirana je u `<stdio.h>`.

Funkcije

Funkcija `puts` uzima kao argument niz znakova i ispisuje ga na standardni izlaz. Ispisuje znakove do nul-znaka, a umjesto njega ispisuje znak `'\n'` za kraj reda. Funkcija vraća cijeli broj (tipa `int`). Ta vrijednost je nenegativan broj ako je ispis uspio, a `EOF` ako nije (greška kod ispisa).

```
#include <stdio.h>

int main(void) {
        char red[128];
        while (gets(red) != NULL)
                puts(red);
        return 0;
}
```

Što ako ulazna linija ima više od 128 znakova? Doći će do *buffer overflowa*! Zato koristite `fgets` i `gets_s` (`gets` uopće nije u C11!).

Funkcije

Značajno bolja verzija ove funkcije koja jest u C11 je:

```
char *gets_s(char *s, rsize_t n);
```

Pri pozivu mora vrijediti `s` $\neq$ `NULL` i $0 <$ `n` $\leq$ `RSIZE_MAX`. Drugi parametar je maksimalni broj znakova u polju `s` koji će biti spremljen, uključivo i nul-znak. Dakle, funkcija učitava najviše `n`-1 znak (znakovi za novi red se preskaču) s ulaza u string `s`. Ako je linija predugačka, preskače višak znakova (do kraja linije ili kraja datoteke) i to se smatra greškom. U slučaju greške, postavlja se `s[0]` na `'\0'` i vraća `NULL`.

```
#include <stdio.h>
int main(void) {
        char red[128];
        while (gets_s(red, sizeof(red)) != NULL)
                puts(red);
        return 0;
}
```

Funkcije

Funkcija `scanf` služi za formatirano učitavanje podataka sa standardnog ulaza (`stdin`). Opći oblik poziva funkcije je

```
scanf(kontrolni_string, arg_1, arg_2, ..., arg_n)
```

Podaci koje scanf stvarno čita su znakovi, koji dolaze sa standardnog ulaza (obično tipkovnica). Ako se unosi više podataka, oni (u principu) moraju biti odvojeni bjelinom (npr. razmak, novi red). Kod čitanja numeričkih podataka, vodeće bjeline se preskaču, a podaci na ulazu moraju imati isti oblik kao i numeričke konstante pripadnog tipa u C-u (bez sufiksa). Primjer:

```
int x, y, z;
scanf("%i %i %i", &x, &y, &z);
```

(`%i` dozvoljava zapis u bazi 8, 10 i 16). Ako se učita ulazna linija

```
13  015    0Xd
```

u sve tri varijable `x`, `y`, `z` nalazi se ista vrijednost.

Funkcije

Može i bez prefiksa, naime

```
int x, y, z;
scanf("%d %o %x", &x, &y, &z);
```

će imati isti efekt pri učitavanju kao na prošlom slajdu za ulaz:

```
13  15    d
```

Znakovima konverzije `d`, `i`, `o`, `u`, `x` treba dodati `h` („half") ako je argument pokazivač na `short`, a `l` („long") ako je na `long`.

```
int x;
short y;
long z;
scanf("%d %hd %ld", &x, &y, &z);
```

Funkcije

Prisjetimo se: znakovi konverzije `e`, `f` i `g` služe za učitavanje vrijednosti u varijablu tipa `float`. Ako se učitava vrijednost u varijablu tipa `double`, mora se koristiti prefiks `l` (`le`, `lf` ili `lg`).

```
float x;
double y;
scanf("%f %lg", &x, &y);
```

Prefiks L koristi se za učitavanje realne vrijednosti u varijablu tipa `long double` (ako postoji).

Funkcija `scanf` dijeli ulazni niz znakova u tzv. polja znakova za konverzije u zadane tipove. Sljedeće polje počinje prvim nepročitanim znakom, nakon eventualnog preskakanja vodećih bjelina (kod numeričkih konverzija i čitanja stringova). Nakon toga se, ovisno o znaku za konverziju, čita (i po potrebi konvertira) najdulji dozvoljeni niz znakova koji odgovara obliku ulaza za taj znak konverzije (npr. za string stane prije iduće bjeline).

Funkcije

Uz svaki znak konverzije može se zadati i maksimalna širina ulaznog polja koje će se učitati, npr. `%3d` ili `%9s`. Ako podatak ima više dozvoljenih znakova od zadanog, pripadno polje znakova za konverziju skraćuje se na zadani broj (višak znakova ostaje za naknadno učitavanje).

Razmak u kontrolnom stringu znači da se na tom mjestu u ulazu preskaču sve bjeline. Primjerice, kontrolni stringovi `"%f%d"` i `"%f %d"` imat će isti efekt, no razmak prije `%c` i `%[` promijenit će značenje tih znakova za konverziju.
U kontrolnom stringu, osim razmaka i oznaka konverzije, mogu se pojaviti i drugi znakovi. Njima moraju odgovarati identični znakovi na ulazu (*sparivanje*). Primjer:

```
scanf("%f,%d", &x, &i);
```

Ulazni podaci ne smiju imati bjelinu između prvog broja i zareza:

```
1.456, 8
```

Funkcije

Oznakom konverzije %s nije moguće učitati niz znakova koji sadrži bjeline jer one služe kao oznaka za kraj polja. Za učitavanje nizova znakova koji uključuju i bjeline, koristimo %[...]. Funkcija scanf će, u pripadni argument, učitati najdulji niz znakova s ulaza koji se sastoji samo od znakova navedenih unutar uglatih zagrada.

Učitavanje završava ispred prvog znaka na ulazu koji nije naveden u uglatim zagradama. Na kraj učitanog niza dodaje se nul-znak. Vodeće bjeline se ne preskaču.

```
char linija[128];
...
scanf(" %[ ABCDEFGHIJKLMNOPQRSTUVWXYZ]", linija);
```

Prvi razmak je bitan, primjerice, ako je preostala neka bjelina u ulaznom nizu (recimo pri prijelazu u novi red); inače bi rezultat bio prazan string. Mogu se koristiti rasponi i negacije:

```
scanf(" %[ A-Z] %[^\n]", linija1, linija2);
```

Funkcije

I ovdje bi trebalo navesti maksimalnu širinu polja (jedan znak treba ostaviti za nul-znak):

```
char str_1[16], str_2[33], str_3[80];
...
scanf("%15s", str_1);
scanf("%32[A-Z]", str_2);
scanf("%79[^\n]", str_3);
```

Znaku konverzije %c možemo zadati maksimalnu širinu polja. Tada se čita uzastopni blok od točno toliko znakova kolika je zadana maksimalna širina polja, a pripadni argument je pokazivač na niz koji mora imati mjesta za barem toliko elemenata (nema kontrole). Za razliku od %s i %[, ne dodaje se nul-znak („nije string").

Naravno, %c ne preskače vodeće bjeline. Mogu se preskočiti sa " %c", a sa " %c %c" se učitaju prva dva znaka koji nisu bjeline (bjeline sve do drugog od njih se preskaču).

Funkcije

```
char kratica[4], drzava[3];
scanf("%4c", kratica);
scanf("%3c", &drzava[0]);
```

Moguće je preskakati polja (npr. stupac u tablici) pomoću *:

```
scanf(" %s %*d %d", ime, &n);
```

Što će se ovdje ispisati za ulaz 17.19x17:

```
double x; char c; int i;
scanf("%lg%c%d", &x, &c, &i);
printf("x = %g, c = '%c', i = %d\n", x, c, i);
```

Što će se ovdje ispisati za ulaz 56789 0123 56a72:

```
int i; float x; char s[50];
scanf("%2d%f%*d %[0-9]", &i, &x, s);
printf("i = %d, x = %f, s = %s\n", i, x, s);
```

Funkcije

Funkcija `scanf` vraća (nenegativan) broj uspješno učitanih podataka, ili `EOF` ako je do greške ili kraja datoteke došlo prije početka čitanja prvog podatka.

```
int n;
while (scanf("%d",&n) == 1 && n >= 0) {
        /* Radi nesto s brojem n. */
}
```

`while` petlja se prekida ako čitanje broja nije uspjelo, ili ako je učitan negativan broj (skraćeno izračunavanje od `&&`).

Funkcije

Još ćemo dopuniti neke detalje o funkciji `printf` koje nismo eksplicitno obradili i onda napraviti neke primjere.

Funkcija `printf` vraća broj ispisanih znakova (nenegativan) ili `EOF` ako je došlo do greške. Koristi prvi argument („format string") za određivanje **broja** argumenata koji slijede i njihovih **tipova**. Ako je argumenata premalo ili su pogrešnog tipa, ponašanje je nedefinirano.

Sitnica: ako želimo ispisati %, stavit ćemo %% u format string. Što će ispisati sljedeći primjer?

```
int n = 13;
printf("%%10d\n", n);
```

Zbog viška argumenata, neki prevoditelji mogu javiti upozorenje.

Ako ispišemo znak sa `%d`, dobit ćemo njegovu ASCII vrijednost:

```
char c = '1';
printf("c(char) = %c, c(int) = %d\n", c, c);
```

Funkcije

Koristimo li znakove konverzije za ispis u bazi 8 ili 16, ne ispisuje se vodeća nula za bazu 8, odnosno `0x` ili `0X` za bazu 16. No, možemo koristiti *alternativnu formu ispisa* pomoću *zastavice* (*flag*) # iza %:

```
int i = 64;
printf("i(dec) = %d = %i\n", i, i);
printf("i(oct) = %o = %#o\n", i, i);
printf("i(hex) = %x = %#x\n", i, i);
```

ispisuje

```
i(dec) = 64 = 64
i(oct) = 100 = 0100
i(hex) = 40 = 0x40
```

Za `i=-3` bi se ispisalo `-3`, `37777777775` i `fffffffd` (zašto?).

Funkcije

Promjena duljine osnovnog tipa zadaje se modifikatorom tipa u odgovarajućoj oznaci konverzije, koji se piše ispred znaka konverzije (tj. kao prefiks).

Za cjelobrojne tipove, modifikatori tipa su:

- `h`: argument treba ispisati kao `short` ili `unsigned short` (argument od `printf` je već pretvoren u `int` ili `unsigned`!)
- `l`: argument je tipa `long` ili `unsigned long` (nema automatskog pretvaranja tipova, pa treba napisati modifikator tipa osim ako je slučajno `int` isto što i `long int`)

Postoji i `ll` za ispis vrijednosti tipa `long long` (koristan je samo ako platforma podržava taj tip, tj. ako jest dulji od `int` i `long`).

Moguće je zadati minimalnu širinu ispisa tako da se ispred znaka konverzije i prefiksa stavi odgovarajući broj (npr. `%3d` ili `%9s`. Ako podatak treba manje znakova, bit će slijeva dopunjen razmacima (upoznat ćemo zastavicu koja to mijenja). Ako treba više znakova, bit će ispisan sa svim potrebnim znakovima.

Funkcije

Sljedeći kod:

```
int n = 54321;
printf("%10d,%5o,%5x\n", n, n, n);
```

ispisuje

```
     54321,152061, d431
```

Oktalni zapis ima svih 6 potrebnih znakova, a heksadekadski je dopunjen razmakom. Sljedeći kod:

```
double x = 1.2;
printf("%1g\n%3g\n%5g\n", x, x, x);
```

ispisuje tri retka s 1.2, samo što treći ima i dva vodeća razmaka.

Funkcije

Pored minimalne širine, moguće je zadati i preciznost ispisa. Primjerice, `%7.3e` znači ispis u e formatu s najmanje 7 znakova, pri čemu je točno (za `e` i `f` formate; za `s` je najviše, a za ostale formate najmanje toliko) 3 znamenke iza decimalne točke. Ako preciznost nije zadana, jednaka je 6.

```
#include <stdio.h>
#include <math.h>

int main(void) {
        double pi = 4.0 * atan(1.0);
        printf("%5f, %10.5f, %5.10f, %5.4f\n",
                pi, pi, pi, pi);
        return 0;
}
```

ispisuje

```
3.141593,    3.14159, 3.1415926536, 3.1416
```

Funkcije

Širina i preciznost ispisa se mogu zadaci dinamički, tj. tijekom izvođenja programa. Iznos širine ili preciznosti u formatu treba zamijeniti znakom *, a na pripadnom mjestu u listi argumenata koje odgovara tom znaku * mora biti cjelobrojni izraz (obično varijabla). Trenutna vrijednost tog izraza određuje širinu i preciznost. Vrijednost tog argumenta se ne ispisuje.

```
#include <stdio.h>
#include <math.h>
int main(void) {
        double pi = 4.0 * atan(1.0); int i = 10;
        printf("%*f, %*.*f, %5.*f\n",
                11, pi, 16, 14, pi, i, pi);
        return 0;
}
```

ispisuje

```
   3.141593, 3.14159265358979, 3.1415926536
```

Funkcije

Znak konverzije %s služi za ispis stringova. Ispisuje sve znakove u stringu, dok ne dode do nul-znaka (njega ne ispisuje).

```
char naslov[] = "Programski jezik C";
printf("%s", naslov);
```

ispisuje

```
Programski jezik C
```

I ovdje možemo zadati ,,maksimalnu širinu" (ovdje najmanji broj znakova u ispisu) i/ili ,,preciznost" (ovdje najveći broj znakova u ispisu):

```
char naslov[] = "Programski jezik C";
printf("%.16s\n", naslov);
```

ispisuje

```
Programski jezik
```

Funkcije

Znak konverzije %s može se koristiti i za ispis polja znakova koje nije string:

```
char kratica[4] = { 'c', 'e', 'r', 'n' };
printf("%.4s, %.2s\n", kratica, &kratica[1]);
```

ispisuje

```
cern, er
```

Zastavice služe za modificiranje standardnog ponašanja znakova konverzije. Pišu se odmah iza znaka %, smije ih biti i više, i mogu biti napisane u bilo kojem međusobnom poretku:

- -: konvertirani argument ispisuje se lijevo unutar polja za ispis
- +: broj se ispisuje s predznakom
- ␣: ako prvi znak (nakon pretvorbe) nije predznak, dodat će se razmak na početak.
- 0: kod numeričkih konverzija, polje za ispis se dopuni (do širine polja) vodećim nulama umjesto razmacima.

Funkcije

označava alternativnu formu ispisa za pojedine znakove konverzije:

- `%#o`: prva znamenka bit će nula
- `%#x` ili `%#X`: dodat će znakove `0x` odnosno `0X` na početak rezultata različitog od nule (ako je nula, neće učiniti ništa)
- `%#e`, `%#E`, `%#f`, `%#g` i `%#G`: ispisani broj će biti ispisan s decimalnom točkom. Dodatno, za `g` i `G`, nule na kraju decimalnog broja (koje bi se mogle brisati) će se ispisati.

Nizovi/polja podataka

Polje je konačan niz varijabli istog tipa, sa zajedničkim imenom, numeriranih nenegativnim cjelobrojnim indeksima (indeksiranje počinje od nule). Uočite sličnost s vektorom u matematici.

```
double x[3]; /* polje x s 3 clana tipa double */
x[0] = 0.2;
x[1] = 0.7;
x[2] = 5.5;
x[3] = 4.4;  /* -> greska, x[3] nije definiran! */
```

Jednodimenzionalno polje definira se na sljedeći način:

```
mem_klasa tip ime[izraz];
```

gdje je `mem_klasa` memorijska klasa cijelog polja (v. Prog2), `tip` tip podatka svakog elementa polja, `ime` zajednički dio imena svih elemenata, ali i **adresa prvog elementa polja**, tj. &ime[0], `izraz` konstantan, cjelobrojan, pozitivan izraz koji zadaje broj elemenata, tj. duljinu polja (često pozitivna cjelobrojna/simbolička konstanta).

Nizovi/polja podataka

Elementi jednodimenzionalnog polja su:

```
ime[0], . . . , ime[izraz - 1].
```

Svaki element `ime[i]` je varijabla tipa tip. Deklaracija memorijske klase nije obavezna. Polje deklarirano bez memorijske klase: unutar funkcije je automatska varijabla (rezervacija memorije na „stacku", ulaskom u funkciju), a izvan svih funkcija je statička varijabla. Unutar neke funkcije, polje se može učiniti statičkim pomoću identifikatora memorijske klase `static` (v. Prog2).

Elementi polja se smještaju u uzastopne, redom indeksirane memorijske lokacije (zbog efikasnosti pristupa). Stoga je polje u memoriji jednoznačno definirano adresom prvog elementa (imenom, za koje smo već rekli da je sinomim za tu adresu, tj. konstantni pokazivač koji sadrži tu adresu), tipom svakog elementa (jer to određuje duljinu elementa) te brojem elemenata.

Nizovi/polja podataka

Adresa elementa `v[i]` ovisi samo o početnoj adresi polja, tipu i indeksu (jer se tada može udaljiti od početka za `sizeof(tip)*i`; C ovo podjednostavljuje programeru, potrebno je samo napisati `ime+i` ili `&ime[i]` umjesto `ime+sizeof(tip)*i`; ako želimo pristupiti toj adresi, pišemo `*(ime+i)` ili `ime[i]`). Ovakvo „računanje s adresama" naziva se *aritmetika pokazivača*.

Primijetite da broj elemenata u polju (iz definicije polja) nije potreban za aritmetiku pokazivača. On služi samo pri inicijalnoj rezervaciji memorije za polje, i nigdje se posebno ne pamti. Zato u C-u nema kontrole granica za indeks elementa polja (programer mora o tome voditi računa).

Polja se mogu inicijalizirati element po element, ili navođenjem popisa vrijednosti elemenata unutar vitičastih zagrada. Pojedine vrijednosti odvajaju se zarezom (koji nije operator zarez).

```
double v[3] = {1.17, 2.43, 6.11};
```

Nizovi/polja podataka

Ako je broj inicijalizacijskih vrijednosti veći od duljine polja javlja se greška. Ako je manji od duljine polja, preostale vrijednosti se inicijaliziraju na nulu.

Prilikom inicijalizacije, duljina polja ne mora biti zadana. Tad se duljina polja računa automatski, iz broja inicijalizacijskih vrijednosti.

```
double v[] = {1.17, 2.43, 6.11};
char c[] = "tri";
```

Posljednji način pridruživanja dozvoljen je samo u definiciji varijable (kao inicijalizacija). Nije dozvoljeno pisati:

```
c = "tri"; /* Pogresno! Koristiti strcpy! */
```

jer lijeva strana pridruživanja ne smije biti polje (ime polja je konstantni pokazivač — adresa prvog elementa).

Nizovi/polja podataka

```c
#include <stdio.h>

int main(void) {
        double v[] = {2.0, 3.11, 4.05, -1.07};
        int n = sizeof(v) / sizeof(double), i;
        double a_sredina = 0.0;

        for(i = 0; i < n; ++i)
                a_sredina += v[i];
        a_sredina /= n;
        printf(" Sredina je %20.12f\n", a_sredina);
        return 0;
}
```

Nizovi/polja podataka

Funkciju `f`, koja prima polje `v` s elementima tipa `tip` kao argument, možemo deklarirati na dva načina:

```
f(tip v[], ...)   ili   f(tip *v, ...)
```

U prvom načinu ne treba navesti duljinu polja. Drugi način direktno kaže da je ime polja `v` pokazivač na objekt tipa `tip` i podrazumijeva se da je to adresa prvog elementa polja.

Ako ne želimo da funkcija mijenja elemente polja unutar funkcije, onda dodajemo ključnu riječ `const` na početku deklaracije argumenta (npr. prvi string u `scanf`, `printf`):

```
f(const tip v[], ...)   ili   f(const tip *v, ...)
```

```
double srednja_vrijednost(int n, const double v[]) {
        int i;
        double suma = 0.0;
        for (i = 0; i < n; ++i) suma += v[i];
                return suma / n; }
```

Nizovi/polja podataka

```
int a[10], b[10];
...
a = a + 1; /* Greska, a je konst. pokazivac. */
b = a;     /* Greska! */
```

```
int a[10], *pa;
...
pa = a;       /* ekviv. s pa = &a[0]; */
pa = pa + 2; /* Nije greska - &a[2] */
pa++;         /* &a[3] */
```

```
int a[10], *pa;
...
pa = &a[0];
*(pa + 3) = 20; /* ekviv. s a[3] = 20; */
*(a + 1) = 10;  /* ekviv. s a[1] = 10; */
```

Nizovi/polja podataka

Pogledajmo primjer koji ističe važnost prioriteta (i asociranosti).

```
int a[4] = {0, 10, 20, 30};
int *ptr, x;
ptr = a;
```

naredba	x	ptr	a[0]	a[1]	a[2]	a[3]
`x = *ptr;`						

Nizovi/polja podataka

Pogledajmo primjer koji ističe važnost prioriteta (i asociranosti).

```
int a[4] = {0, 10, 20, 30};
int *ptr, x;
ptr = a;
```

naredba	x	ptr	a[0]	a[1]	a[2]	a[3]
x = *ptr;	0	0098F830	0	10	20	30
x = *ptr++;						

Nizovi/polja podataka

Pogledajmo primjer koji ističe važnost prioriteta (i asociranosti).

```
int a[4] = {0, 10, 20, 30};
int *ptr, x;
ptr = a;
```

naredba	x	ptr	a[0]	a[1]	a[2]	a[3]
x = *ptr;	0	0098F830	0	10	20	30
x = *ptr++;	0	0098F834	0	10	20	30
x = (*ptr)++;						

Nizovi/polja podataka

Pogledajmo primjer koji ističe važnost prioriteta (i asociranosti).

```
int a[4] = {0, 10, 20, 30};
int *ptr, x;
ptr = a;
```

naredba	x	ptr	a[0]	a[1]	a[2]	a[3]
x = *ptr;	0	0098F830	0	10	20	30
x = *ptr++;	0	0098F834	0	10	20	30
x = (*ptr)++;	10	0098F834	0	11	20	30
x = *++ptr;						

Nizovi/polja podataka

Pogledajmo primjer koji ističe važnost prioriteta (i asociranosti).

```
int a[4] = {0, 10, 20, 30};
int *ptr, x;
ptr = a;
```

naredba	x	ptr	a[0]	a[1]	a[2]	a[3]
x = *ptr;	0	0098F830	0	10	20	30
x = *ptr++;	0	0098F834	0	10	20	30
x = (*ptr)++;	10	0098F834	0	11	20	30
x = *++ptr;	20	0098F838	0	11	20	30
x = ++(*ptr);						

Nizovi/polja podataka

Pogledajmo primjer koji ističe važnost prioriteta (i asociranosti).

```
int a[4] = {0, 10, 20, 30};
int *ptr, x;
ptr = a;
```

naredba	x	ptr	a[0]	a[1]	a[2]	a[3]
x = *ptr;	0	0098F830	0	10	20	30
x = *ptr++;	0	0098F834	0	10	20	30
x = (*ptr)++;	10	0098F834	0	11	20	30
x = *++ptr;	20	0098F838	0	11	20	30
x = ++(*ptr);	21	0098F838	0	11	21	30

Osnovne operacije s nizovima

Zadan je cijeli broj `n` i (ako je pozitivan) polje `x` realnih brojeva (recimo `double`) duljine `n`. Treba naći zbroj svih članova niza (kako bismo tretirali slučaj $n \leq 0$?).

```
...
zbroj = 0.0;
for (i = 0; i < n; ++i)
        zbroj += x[i];
...
printf("Zbroj clanova niza = %f.\n", zbroj);
```

Možemo algoritam izlučiti u funkciju:

```
double zbroj_clanova(int n, double x[]) {
        int i;
        double zbroj = 0.0;
        for (i = 0; i < n; ++i)
                zbroj += x[i];
        return zbroj;
}
```

Osnovne operacije s nizovima

Našu funkciju iz funkcije `main` možemo pozvati ovako:

```
int main(void) {
        int n = 5;
        double v[] = {1.2, 2.6, 1.8, 4.4, 0.8};

        printf("Zbroj svih clanova niza = %f\n",
                zbroj_clanova(n, v) );

        printf("Zbroj srednja tri clana niza = %f\n",
                zbroj_clanova(3, &v[1]) );

        return 0;
}
```

Osnovne operacije s nizovima

Zadan je pozitivan cijeli broj `n` i polje `x` realnih brojeva (recimo `double`) duljine `n`. Treba naći minimalni član niza.

```
min = x[0];
poz = 0;

for (i = 1; i < n; ++i)
        if (x[i] < min) {
                min = x[i];
                poz = i;
        }
...
printf("Najmanji clan niza: x[%d] = %f\n",
        poz, min);
```

Je li bilo bitno zahtijevati $n > 0$? Koja je složenost ovog algoritma?

Osnovne operacije s nizovima

Opet možemo sav bitan posao prebaciti u funkciju:

```
double min_clan(int n, double x[])
{
        int i;
        double min = x[0];

        for (i = 1; i < n; ++i)
                if (x[i] < min)
                        min = x[i];
        return min;
}
```

Zadatak: Doradite gornju funkciju tako da vrati i poziciju/indeks (nekog) minimalnog elementa u polju kao varijabilni argument.

Pretraživanje nizova

Problem pretraživanja: provjerite nalazi li se zadani element `elt` među članovima zadanog niza `x`. Drugim riječima, provjerite postoji li indeks `i` takav da je `elt` jednak `x[i]`.

Za početak želimo samo odgovor na ovo pitanje, tj. rezultat je logička vrijednost, istina (1) ili laž (0).

Ako niz nije sortiran, možemo ga pretraživati bilo kojim redom, element po element (*sekvencijalno pretraživanje*). Stanemo kad smo ili našli traženi element, ili iscrpili niz. U najgorem slučaju (elementa nema ili je baš zadnji koji obradimo), treba nam jedan prolaz kroz sve elemente polja (složenost?). U najboljem, obradit ćemo samo jedan element. DZ: Napišite donji kod bez `nasli`.

```
nasli = 0;
i = 0;
while (!nasli && i < n) {
        if (x[i] == elt) nasli = 1;
        else ++i;
}
```

Pretraživanje nizova

Pripadna funkcija bi mogla izgledati ovako:

```
int seq_search(int x[], int n, int elt)
{
        int i;

        for (i = 0; i < n; ++i)
                if (x[i] == elt)
                        return 1;

        return 0;
}
```

Pretraživanje nizova

Složenost pretraživanja mjerimo brojem usporedbi ,,jednak" odnosno ,,različit" (jer nema uređaja). Te usporedbe vršimo unutar tipa za članove niza. Operacije na indeksima ne brojimo, njih ima podjednako mnogo kao i usporedbi. U najgorem slučaju, moramo provjeriti sve članove niza, tj. broj usporedbi je najviše n.

Ova mjera složenosti je dobra procjena za trajanje izvršavanja (oznaka: $T(n)$) algoritma sekvencijalnog pretraživanja. Vidjeli smo da trajanje u najgorem slučaju linearno ovisi o n, pa pišemo $T(n) \in O(n)$. Općenito, zapis

$$T(n) \in O(f(n))$$

(ponekad ćete vidjeti $=$ umjesto $\in$) za funkcije $T, f : \mathbb{N} \to \mathbb{R}$ znači:

$$(\exists c \in \mathbb{R})(\exists n_0 \in \mathbb{N})(\forall n \in \mathbb{N})(n \geq n_0 \Rightarrow T(n) \leq c \cdot f(n)).$$

Intuitivno: od nekog mjesta nadalje, T ne raste brže nego $c \cdot f$, tj. od tog mjesta graf od T ne prelazi iznad grafa od $c \cdot f$.

Pretraživanje nizova

Ako je niz uzlazno ili silazno sortiran, potraga se može drastično ubrzati. Recimo da zamislite neki prirodan broj između 1 i 1000. Što mislite, koliko pokušaja mi treba za pogoditi ga ako mi samo kažete ,,(moj broj je) manji" ili ,,(moj broj je) veći"?

Pretraživanje nizova

Ako je niz uzlazno ili silazno sortiran, potraga se može drastično ubrzati. Recimo da zamislite neki prirodan broj između 1 i 1000. Što mislite, koliko pokušaja mi treba za pogoditi ga ako mi samo kažete ,,(moj broj je) manji" ili ,,(moj broj je) veći"?

Vratimo se na apstraktni model. Za elemente niza vrijedi

$$x_0 \leq x_1 \leq \ldots \leq x_i \leq \ldots \leq x_{n-2} \leq x_{n-1},$$

pri čemu je x_i odabrani objekt koji ćemo usporediti sa zadanim elementom `elt`.

Kako ne znamo koji su elementi u nizu, niz je dobro prepoloviti. Onda je podjednako vjerojatno da se `elt` nalazi u prvom ili drugom dijelu (jer su prvi i drugi dio podjednake veličine). U tom slučaju, bez obzira gdje se element nalazi, potragu smo smanjili na podniz s polovičnim brojem elemenata.

Pretraživanje nizova

Precizno, neka je $l = 0$ indeks prvog, a $d = n - 1$ indeks zadnjeg elementa u nizu. Srednji element ima indeks

$$i = \left\lfloor \frac{l+d}{2} \right\rfloor \quad \text{ili} \quad i = \left\lceil \frac{l+d}{2} \right\rceil .$$

Budući da cjelobrojnim dijeljenjem u C-u dobijemo prvi izbor, najjednostavnije nam je njega uzeti za ,,sredinu".

Elemente niza x svrstali smo u 3 skupine:

- elemente s indeksima od $l = 0$ do $i - 1$,
- element s indeksom i,
- elemente s indeksima od $i + 1$ do $d = n - 1$.

Pretraživanje nizova

Postavljamo tri pitanja:

- `elt` $< x_i$? Odgovor "da" znači da nastavljamo tražiti u podnizu s indeksima od l do $d = i - 1$ (ispred x_i),
- `elt` $> x_i$? Odgovor ,,da" znači da nastavljamo tražiti u podnizu s indeksima od $\mathrm{l} = \mathrm{i} + 1$ do d (iza x_i),
- `elt` $= x_i$? Odgovor ,,da" znači da smo pronašli `elt`.

Točno jedno od toga je istinito (treba najviše 2 pitanja)! Potraga traje sve dok nismo našli traženi element i vrijedi $l \leq d$. U protivnom ($l > d$), element nije u nizu.

```
nasli = 0; l = 0; d = n - 1;
while (!nasli && l <= d) {
        i = (l + d) / 2;
        if (elt < x[i]) d = i - 1;
        else if (elt > x[i]) l = i + 1;
        else nasli = 1;}
```

Zadatak: Izbacite uvjet `!nasli` i iskoristite `break` gdje treba.

Pretraživanje nizova

Evo i funkcije koja vraća traženi odgovor:

```
int binary_search(int x[], int n, int elt) {
        int l = 0, d = n - 1, i;
        while (l <= d) {
                i = (l + d) / 2;
                if (elt < x[i])
                        d = i - 1;
                else if (elt > x[i])
                        l = i + 1;
                else
                        return 1;
        }
        return 0;
}
```

Pretraživanje nizova

Koliko traje najdulja potraga (ako elementa nema ili je baš zadnji koji pogledamo)?

- Nakon 1. podjele, preostaje najviše $\frac{n}{2}$ elemenata.
- Nakon 2. podjele, preostaje najviše $\frac{n}{4}$ elemenata.
- Nakon k. podjele, preostaje najviše $\frac{n}{2^k}$ elemenata.

Zadnji prolaz je k. ako je duljina baš u tom prolazu pala strogo ispod 1, tj. u prošlom koraku je duljina niza još bila barem 1:

$$\frac{n}{2^k} < 1 \leq \frac{n}{2^{k-1}},$$

$$2^{k-1} \leq n < 2^k.$$

Složenost opet mjerimo brojem usporedbi, ali sada koristimo uređaj ,,manji ili jednak" definiran među elementima niza (tim uređajem je niz sortiran). Operacije na indeksima, opet, ne brojimo. U najgorem slučaju, za broj raspolavljanja k vrijedi

Pretraživanje nizova

Koliko traje najdulja potraga (ako elementa nema ili je baš zadnji koji pogledamo)?

- Nakon 1. podjele, preostaje najviše $\frac{n}{2}$ elemenata.
- Nakon 2. podjele, preostaje najviše $\frac{n}{4}$ elemenata.
- Nakon k. podjele, preostaje najviše $\frac{n}{2^k}$ elemenata.

Zadnji prolaz je k. ako je duljina baš u tom prolazu pala strogo ispod 1, tj. u prošlom koraku je duljina niza još bila barem 1:

$$\frac{n}{2^k} < 1 \leq \frac{n}{2^{k-1}},$$

$$2^{k-1} \leq n < 2^k.$$

Složenost opet mjerimo brojem usporedbi, ali sada koristimo uređaj ,,manji ili jednak" definiran među elementima niza (tim uređajem je niz sortiran). Operacije na indeksima, opet, ne brojimo. U najgorem slučaju, za broj raspolavljanja k vrijedi

$$k = 1 + \lfloor \log_2 n \rfloor.$$

Sva ona imaju najviše 2 usporedbe, pa usporedbi ima najviše $2k$.

Pretraživanje nizova

Prema tome, složenost u najgorem slučaju (*worst-case complexity*) je $O(\log n)$. Možemo pisati i

$$T(n) \in O(\log n).$$

Prisjetite se našeg primjera s pogađanjem broja od 1 do 1000. Zaista je trebalo (uz sortiran niz i binarno pretraživanje) najviše 10 pokušaja (isto tako bi bilo skroz do 1024; no, do 1025 bi općenito bio potreban dodatan pokušaj).

Bitan zaključak: sortiramo kako bismo brže pretraživali.

Zadatak: Može se napraviti i varijanta sa samo jednom usporedbom u svakom raspolavljanju i još jednom usporedbom na kraju. Probajte sami, pa usporedite sa slajdovima prof. Singera.

Sortiranje nizova

Zadan je niz od n elemenata

$$x_0, x_1, \ldots, x_{n-1}$$

koje možemo uspoređivati relacijom uređaja $\leq$ ili $\geq$. Treba preurediti zadani niz tako da članovi budu uzlazno (silazno) poredani.

Uzlazni poredak ($\nearrow$): $x_{i_0} \leq x_{i_1} \leq \ldots \leq x_{i_{n-1}}$
Silazni poredak ($\searrow$): $x_{i_0} \geq x_{i_1} \geq \ldots \geq x_{i_{n-1}}$

Ideja: koristimo usporedbe i zamjene elemenata u nizu. Npr.

- Nađi (neki) minimalni element niza (ovo smo radili).
- Dovedi ga na početak (zamjenom s prvim elementom).
- Ponovi postupak na nizu $x_1, \ldots, x_{n-1}$ (kad stanemo?).

Opisani algoritam se zove *selection sort* („odabiremo" minimalni element).

Sortiranje nizova

```
void selection_sort(int x[], int n) {
        int i, j, ind_min, x_min, temp;
        for (i = 0; i < n - 1; ++i) {
                ind_min = i;
                x_min = x[i];
                for (j = i + 1; j < n; ++j) {
                        if (x[j] < x_min) {
                                ind_min = j;
                                x_min = x[j];
                        }
                }
                if (i != ind_min) {
                        x[ind_min] = x[i];
                        x[i]=x_min;
                }
        }
}
```

Sortiranje nizova

```
int main(void) {
        int x[] = {42, 12, 55, 94, 18, 44, 67};
        int i, n = 7;

        selection_sort(x, n);

        printf("Sortirano polje x:\n");
        for (i = 0; i < n; ++i)
                printf(" x[%d] = %d\n", i, x[i]);

        return 0;
}
```

Zadatak: Promijenite kod funkcije `selection_sort` tako da se pamti samo indeks elementa na kojem se ekstrem dostiže. (Općenito govoreći, skraćivanje koda ne mora ubrzati program!)

Zadatak: Sortirajte niz dovođenjem maksimalnog elementa na kraj.

Sortiranje nizova

Kako ćemo uspoređivati koliko je brzo sortiranje niza raznim algoritmima (u ovisnosti o duljini niza)? Mjeriti vrijeme nije sasvim precizno, posebno za kratke nizove, pa uspoređujemo broj operacija koje algoritmi obavljaju.

Primijetite da kod sortiranja imamo dvije bitno različite operacije na elementima (koje ne moraju jednako trajati): uspoređivanje te zamjena (preciznije, tri pridruživanja).

Za selection sort, broj usporedbi u svakom koraku jednak je trenutnoj duljini niza minus 1 (svaki element osim prvog uspoređuje se sa zasad minimalnim). Ukupno, broj usporedbi je

$$(n-1)+(n-2)+\cdots+2+1=\frac{(n-1)\cdot n}{2}\approx\frac{n^2}{2}.$$

Dakle, broj usporedbi kvadratno ovisi o n. Uočite da je broj zamjena najviše $n-1$, tj. linearan u n. Ukupno, složenost ovog algoritma je $O(n^2)$ (je li ista za sortiranje dovođenjem maksimalnog elementa na kraj?). No, postoje i značajno efikasniji algoritmi.

Sortiranje nizova

Kako bismo provjerili je li zadani niz već sortiran (npr. uzlazno)? Mora biti monotono rastući, tj. za svaki par indeksa $j, k \in \{0, \ldots, n-1\}$ mora vrijediti $j < k \Rightarrow x_j \leq x_k$. Primijetite da nije potrebno provjeriti sve parove indeksa — tih usporedbi bi bilo čak $n(n-1)/2$. Dovoljno je provjeriti samo sve susjedne parove indeksa, tj. možemo uzeti $k = j + 1$. Onda mora vrijediti $x_j \leq x_{j+1}$, za sve $j \in \{0, \ldots, n-2\}$. Drugim riječima,

$$(x_0 \leq x_1) \wedge (x_1 \leq x_2) \wedge \cdots \wedge (x_{n-2} \leq x_{n-1}).$$

```
int sortiran(int x[], int n) {
        int j;
        for (j = 0; j < n - 1; ++j)
                if (x[j] > x[j + 1])
                        return 0;
        return 1;
}
```

Sortiranje nizova

Ako niz nije sortiran, onda postoji indeks $j \in \{0, \ldots, n-2\}$ takav da je $x_j > x_{j+1}$, Ako želimo sortirati niz, onda je prirodna ideja: ispraviti poredak takvih susjeda. Hoće li jedan prolaz biti dovoljan?

Ne samo da neće biti dovoljan pri toj ideji, nego pri svakoj! Nema algoritma linearne složenosti (u najgorem slučaju) koji sortira općeniti niz usporedbama i zamjenama!

Za sortiranje zamjenama susjeda (*bubble sort*), ideja je sljedeća:

- Prolazimo kroz niz slijeva na desno.
- Ako dva susjedna člana niza x_j i x_{j+1} nisu u dobrom poretku, zamijenimo ih.
- Kad stignemo do kraja niza, ponovimo postupak (kad stanemo?).

Primijetite da je maksimalni element u nizu došao do kraja već u jednom prolazu. Zato sigurno možemo stati nakon $n-1$ prolaza (s tim da svaki put prolazimo jedan manje element). Možemo li ponekad stati i ranije?

Sortiranje nizova

U bubble sortu, uspoređujemo $n-1$ parova susjeda, u drugom $n-2$, i tako redom, do 1 u zadnjem koraku. Dakle, ukupan broj usporedbi je, kao i kod izbora ekstrema, $(n-1) \cdot n/2$.

S druge strane, broj zamjena može drastično varirati: od 0 ako je niz već sortiran, do $(n-1) \cdot n/2$ (svaka usporedba rezultira zamjenom) ako je niz ,,naopako sortiran". Dakle, složenost (u najgorem slučaju) je ista kao i za selection sort.

Možemo li napraviti još neku optimizaciju? Da! Zapamtimo li indeks zadnje zamjene u trenutnom prolazu, možemo zaključiti da su elementi nakon tog mjesta ispravno sortirani (i idući prolaz raditi samo do tog mjesta). Primijetite da nam za ispravno sortiran niz treba samo jedan prolaz, ali za obrnuto sortiran niz nema uštede.

Postoje još neke varijante (npr. *shaker sort* gdje mijenjamo smjer pri svakom prolazu) sa svojim optimizacijama, ali ne možemo zaobići činjenicu da uvijek ima previše zamjena (selection sort je tu bolji). Iako je takve algoritme najbolje ne koristiti (o boljima kasnije), pogledajmo kod za poboljšanu verziju bubble sorta.

Sortiranje nizova

```
void bubble_sort(int x[], int n) {
        int i = n - 1, j, ind_zamj, temp;
        do {
                ind_zamj = -1;
                for (j = 0; j < i; ++j)
                        if (x[j] > x[j + 1]) {
                                temp = x[j];
                                x[j] = x[j + 1];
                                x[j + 1] = temp;
                                ind_zamj = j;
                        }
                i = ind_zamj;
        } while (i > 0);
}
```

Sortiranje nizova

Zadan je uzlazno sortirani niz `x` s `n` elemenata. U taj niz treba ubaciti (dodati, umetnuti) zadani element `elt`, ali tako da novi niz `x` sa `n+1` elemenata i dalje bude uzlazno sortiran. Želimo to napraviti *in place*, tj. operacije obaviti unutar niza `x` (pretpostavljamo da u nizu ima mjesta za `n+1` elemenata) — ako niz zamijenimo vezanom listom, problem postaje lakši, v. Prog2.

Opisano *sortirano ubacivanje* elementa `elt` u niz `x` ima dva bitna koraka. Prvo treba naći pravo mjesto (označimo indeks sa `j`), primjerice sekvencijalno od početka ili kraja, ili binarnim pretraživanjem. Zatim treba pomaknuti elemente `x[j]`, ..., `x[n-1]` za jedno mjesto udesno kako bismo oslobodili `x[j]` za `elt`.

Uočite da je pomake (koji su zapravo kopiranje) potrebno raditi od kraja niza. Stoga ćemo početi tražiti od kraja i usput raditi potrebne pomake. Očito, krenemo li od prvog elementa niza (sortiran jednočlan niz), možemo ostatak niza sortirano ubaciti element po element (taj algoritam se zove *insertion sort*).

Sortiranje nizova

```
void insertion_sort(int x[], int n) {
        int i, j, elt;

        for (i = 1; i < n; ++i)
             if (x[i - 1] > x[i]) {
                  elt = x[i];
                  j = i;
                  do {
                       x[j] = x[j - 1];
                       --j;
                  }
                  while (j >= 1 && x[j - 1] > elt);
                  x[j] = elt;
             }
}
```

Sortiranje nizova

U jednom koraku algoritma, broj usporedbi može varirati od jedne (`x[i]` je na pravom mjestu) do i (element koji ubacujemo bit će minimalni u sređenom dijelu). Dakle, u najgorem slučaju (obratno sortirani niz), ukupan broj usporedbi je, kao i kod ranijih algoritama, $\frac{(n-1)\cdot n}{2}$. Može biti i bitno manji, samo $n-1$ za već sortirani niz.

Broj pomaka u ovom algoritmu je skoro jednak broju usporedbi. No, pomaci su brži od zamjena (svaka ima 3 pridruživanja). Zato je insertion sort nešto brži od klasičnih (sporih) algoritama premda je u istoj klasi složenosti (radi se o multiplikativnoj konstanti).

Zadatak: Skratite kod izbacivanjem provjere uvjeta `x[i-1] > x[i]` (hint: dozvolite kopiranje elementa niza u `elt` i natrag čak i kad nije potrebno).

Zadatak: Optimizirajte kod dovođenjem minimalnog elementa niza na početak (tada ne treba provjeravati `j>=1`) i binarnim pretraživanjem umjesto sekvencijalnog.

Sortiranje nizova

Može se značajno povećati efikasnost insertion sorta ako ne mičemo veće elemente samo za jedno mjesto, nego sortiramo više podnizova s većim razmacima. Dobit ćemo *Shell sort* (Donald L. Shell, 1959.). Originalno (Shell) su razmaci bili $n/2$ (podnizovi s oko dva elementa), $n/4$, ..., 2, 1 (cijeli niz). Analiza složenosti je vrlo zahtjevna (ovisno o izboru razmaka, algoritam može biti bitno brži od kvadratnog).

Može se pokazati da za bilo koji algoritam sortiranja, koji koristi usporedbe parova članova (binarne relacije $<$, $>$), usporedbi mora biti barem $cn \log n$, gdje je c neka pozitivna konstanta, tj. broj usporedbi je reda veličine barem $n \log n$. To je bitno brže nego n^2 usporedbi (za velike n). Koji algoritmi rade tako "brzo"?

Inače, postoje i brži algoritmi sortiranja (ne koriste usporedbe), ali rade samo za specijalne vrste podataka! Recimo, *radix sort* za nenegativne brojeve ima linearnu složenost.

Sortiranje nizova

Spomenimo neke praktične algoritme za sortiranje koje ćete naučiti uskoro:

- *quicksort*, C. A. R. (Tony) Hoare, 1962. godine. Prosječna složenost mu je reda veličine $n \log n$. U najgorem slučaju, složenost je reda veličine n^2 . Koristi se zbog dobre prosječne brzine i dio je standardne C biblioteke. Radimo ga na početku Prog2.
- *heapsort*, John W. J. Williams, 1964. godine. Prosječna i najgora složenost su reda veličine $n \log n$, ali je u praksi nešto sporiji od quicksorta. Detaljniji opis na SPA, jer koristi strukturu hrpe (heap).
- *mergesort*, John von Neumann, 1945. godine. Najgora složenost je, takoder, reda veličine $n \log n$, tj. dostiže donju ogradu. Lakše ga je napraviti na vezanoj listi, nego na polju. Na polju, treba pomoćno polje, i zato ima dodatna kopiranja izmedu ta dva polja. Radimo ga kod vezanih listi na Prog2.

Sortiranje nizova

Vrijeme u sekundama za sortiranje polja duljine 10^5 (indeksi predstavljaju različite varijante algoritama, prevoditelj je Intelov):

Algoritam	Slučajno	Uzlazno	Silazno
`min_1`	1.763	1.766	1.765
`min_2`	1.758	1.755	1.861
`max_1`	1.422	1.425	1.423
`max_2`	1.424	1.425	1.428
`bubble_1`	12.064	2.154	5.367
`bubble_2`	11.952	0.000	5.361
`bubble_3`	12.427	0.000	5.320
`ins_1`	1.212	0.000	2.431
`ins_1a`	1.191	0.000	2.383
`ins_3`	0.822	0.000	1.627

Kako je moguće da je kod bubble sorta trajanje u prosječnom slučaju veće od trajanja u najgorem slučaju?

Sortiranje nizova

Moderni procesori predviđaju grananja (*branch prediction*). Da bi se ubrzalo izvršavanje instrukcija, procesor skuplja statistiku za rezultate usporedbi u `if` naredbi — izvršava li se zamjena, ili se preskače. Na temelju tih rezultata, procesor predviđa ishod grananja i priprema instrukcije koje će (najvjerojatnije) izvršiti.

Ako uspješno predvidi ishod, instrukcije su već spremne kad dodu na red za izvršavanje. U protivnom, priprema ne samo da nije bila korisna, nego se na nju potrošilo dodatno vrijeme; sad treba dohvatiti one prave instrukcije za izvršavanje.

Bubble sort: U najgorem slučaju, uvijek se radi zamjena i predviđanje je korektno. U općem slučaju, često je pogrešno!

Sortiranje nizova

Vrijeme u sekundama za sortiranje polja duljine 10^6 (10^5):

Algoritam	Slučajno	Uzlazno	Silazno
`qs_p2_1`	0.101	2.528	2.295
`qs_spamod`	0.076	1.442	1.441
`qs_std`	0.174	0.005	0.005
`heap_spaa`	0.111	0.061	0.060
`heap_b_4a`	0.102	0.042	0.048
`merge_spa`	0.099	0.024	0.024
`merge_2`	0.098	0.021	0.022
`shell_spa`	0.130	0.014	0.021
`shell_3`	0.116	0.009	0.016
`radix_256b`	0.015	0.026	0.025