PROGRAMIRANJE - UVOD

⇒ algoritam - pravilo koje opisuje rješavanje nekog problema
⇒ program - opis algoritma u nekom programskom jeziku
⇒ programiranje je pisanje programa
↳ algoritam je jednoznačan, a programiranja nije

```
#include <stdio.h>
```
↳ uputa pretprocesoru: uključuje standardne operacije

```
int main(void) {.....
.....
.....
return 0; }
```
↳ tijelo programa

```
int m, n, rez;
```
↳ definicija varijabli (prostor u memoriji računala, poznate veličine, kojem je dodijeljeno ime i čiji se sadržaj može mijenjati)

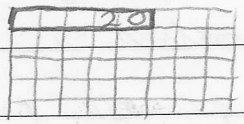
```
/* učitaj dva cijela broja */
```
↳ komentar koji ne utječe na program

```
scanf("%d %d", &m, &n)
```
↳ funkcija za učitavanje vrijednosti s tipkovnice
↳ U PRIMJERU: %d (cjelobrojne vrijednosti)
%f (realne vrijednosti)

printf("%d %d \n", m, n);

↳ funkcija za ispisivanje na zaslon

↳ npr. "%5d" rezervira 5 mjesta na ekranu:



↳ npr. "%5.2f" ⇒ rezervira 5 mjesta i zaokružuje na dvije decimale

"%.2f" ⇒ samo zaokružuje na dvije decimale

⇒ realno dijeljenje: float x;

x = 3./4

ZAD: Napiši pseudokod za upisivanje 2 pravca i provjeri sjeku li se

⇒ učitaj koeficijente

⇒ a_1, a_2, b_1, b_2 (float)

⇒ ako je ($a_1 == a_2$)

pravci ||

inače

pravci se sijeku

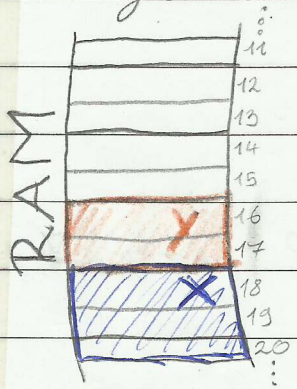
⇒ return 0;

⇒ C program se sastoji od blokova, deklaracija, varijabli i funkcija

#define PI 3.14159 ⇒ definicija simboličke konstante

↳ kad se program compile-ira, prepisuje se vrijednost na svako mjesto u programu gdje treba

⇒ o varijablama:

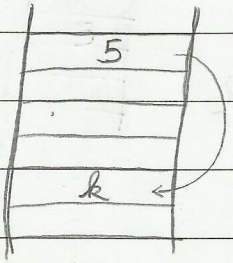


⇒ računalo sprema varijable na RAM u stack (stol)

⇒ u C-u, ne možemo definirat varijablu u petlji: for (int i=0; i<5; i++)

⇒ izraz ⇒ niz znakova koj daje rezultat

⇒ o pridruživanju:




```
int k;
k = 5;
```

↳ uzme konstantu 5 iz mjesta u memoriji i stavi ju na mjesto konstante „k"

⇒ za sve brojeve osim nule, grana „if ( )" reprezentira vrijednost 5 kao TRUE

TIPOVI PODATAKA

npr. float x, y; X = 1 000 000.3

Y = X + 0.125 ⇒ Y = 1 000 000.437500 ? GREŠKA ?!

PRIKAZ CIJELIH BROJEVA (INT):

0	1	0	0	1	1

BIT ⇒ „binary digit"

↳ pretvorba dekadskog u binarni:

174_{10}

$174 - 128 = 46$

$46 - 32 = 14$

$14 - 8 = 6$

$6 - 4 = 2$

$2 - 2 = 0$

128 64 32 16 8 4 2 1

$174_{10} = 10101110_2$

↳ koliko bitova treba za neki dekadski broj:

37541 → [| | | | |]

d = 5 ?

$10^d - 1 = 2^b - 1$

$10^d = 2^b \mid \log$

$d = b \cdot \log 2 \Rightarrow b = \frac{d}{\log 2}$

(binarne znamenke, dekadske znamenke)

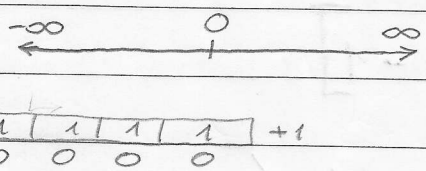
(slajd 11)

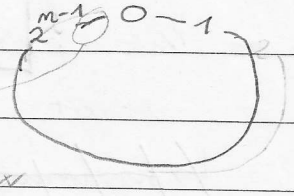
$\boxed{b = 3.33\,d}$

⇒ stoga, problem računala je ograničena pohrana:

npr. u 8-bitni registar, najveći broj koji možemo zapisati je $2^8 - 1 = 255$

⇒ u matematici imamo pravac, u računalu krug:





⇒ kako zapisemo negativne brojeve? (slajd 12)

↳ ako rezerviramo jedan bit za predznak, javljaju se greške kod zbrajanja

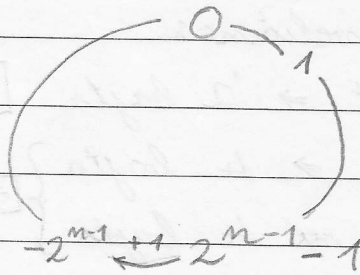
↳ zato se odlučima DVOJNIM KOMPLEMENTOM:

↳ komplement od 1 je: 111, u 3-bitnom registru

↳ oduzimanje pomoću zbrajanja:

$$\begin{array}{r} 001 \quad (1) \\ +\,111 \quad (-1) \\ \hline 1000 \end{array}$$

↳ brojevni krug za komplemente:



ZAD: Napiši broj -11 kao dvojni komplement u registru od 5 bitova!

11 ⇒

16	8	4	2	1
0	1	0	1	1

10100 (+1)

10101 → to je broj -11

↳ invertiraj bitove
dodaj jedan

POSTUPAK

⇒ u tehnici dvojnog komplementa pokrivamo brojeve:

$$\left[-2^{n-1}, 2^{n-1}-1\right]$$

~~↳ napomena: to je broj jedinog bita rezerviranog za prefiksa~~

↳ napomena: broj -2^{n-1} komplementiran daje sebe

⇒ PREFIKSI (KVALIFIKATORI):

a) **short** int - smanjuje raspon vrijednost (broj bitova)
- manje memorije

b) **long** int - povećava raspon vrijednosti (broj bitova)
- više memorije

1) **signed** - pridružuvanje i negativnih vrijednosti i pozitivnih vrijednosti

2) **unsigned** - pridruživanje samo pozitivnih vrijednosti

↳ obično su veličine:

short ⇒ 2 bajta [-32768, 32767] ili [0, 65535]

int ⇒ 4 bajta } [-2milijarde, 2milijarde] ili [0, 4milijarde]

long ⇒ 4 bajta }

⇒ zadavanje u heksadekadskom sustavu:

int a = 0xF01ACU;

0x → PAZI!

F01AC → broj

U → unsigned

⇒ zadavanje u oktalnom sustavu;

int a = 0173;

0 → PAZI!

⇒ u `printf("...",)` naredbi, „%u" znači ispis u unsigned modu (nije greška ako pišemo „%d")

⇒ ako koristimo neku veliku konstantu, koristi long:

```
long int = 1087624L;
```

⇒ ispis unsigned oktalnih brojeva: %O

⇒ ispis unsigned heksadekadskih brojeva: %X

ZNAKOVI U „C"-u

⇒ ASCII kod ⇒ izumljen za teleprintere (zato ima „čudne" komande)

⇒ zadržan do danas

⇒ 7-bita, osmi je za detekciju pogrešaka

↳ 128 kombinacija:

- 26 velikih slova
- 26 malih slova
- 10 znamenaka

⇒ zbog potrebe za drugim znakovima kod nas se koristio YUSCII ({ ⇒ š ; \ ⇒ Đ ; ...)

⇒ danas, zbog problema potrebe za znakove drugih jezika, koristi se UNICODE (UTF-16)

⇒ windows: \r\n ⇒ carriage return, line feed (novi red)

unix: \n ⇒ line feed (novi red)

↳ zato su inkompatibilne datoteke iz jednog u drugi OS

⇒ ASCII:

↳ „null" znak: '0' ⇒ '\0'

↳ „line feed": 10 ⇒ '\n'

↳ razmak: 32 ⇒ ' '

↳ nula - devet: 48-57 ⇒ '0-9'

↳ slovo 'A': 65 ⇒ 'A'

↳ slovo 'a': 97 ⇒ 'a'

↳ znak '@': 64 ⇒ '@'

CHAR - ZNAKOVNI TIP

⇒ char je zapravo broj [-128, 128]

↳ unsigned char [0, 128]

⇒ upis u char:

```
char c = 'A';
```

↳ pogledaj u ASCII tablici i upiši u char broj povezan s A

```
char c = 65;
char c = 0x41;
char c = '\x41';
char c = '\101';
```

↳ upis „problematičnih" znakova u char koristi se znak 'backslash' (\): npr. char c = '\\'

↳ pohranjuje broj od \ (backslash)

npr. char c = '\''

↳ pohranjuje broj od ' (navodnik)

PRIKAZ REALNIH BROJEVA

Pr: Pretvori u binarni: a) 15.75

$15 = 1111_2$; $.75 \cdot 2 = 1.5$

$0.75 = 0.11_2$ $.5 \cdot 2 = 1.0$ → 11

$15.75 = 1111.11_2$

b) 13.3

$13 = 1101_2$; $0.3 \cdot 2 = 0.6$

$0.6 \cdot 2 = 1.2$ .0100**1**

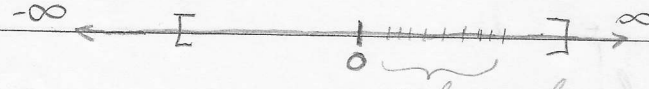
$0.2 \cdot 2 = 0.4$ ↳ decimalni dio

$0.4 \cdot 2 = 0.8$ u binarnom nije

$0.8 \cdot 2 = 1.6$ konačan

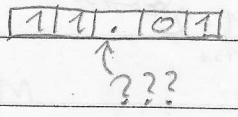
⋮

⇒ Kako onda pohraniti realne brojeve?



↳ ne mogu pohraniti sve!

↳ kako napisati točku u registru?



↳ kako pohraniti mali i velike brojeve? (slajd 67)

- ODGOVOR: znanstveni zapis: mantisa

a) dekadski: $3.13457 \cdot 10^6$

b) binarni:

$$\pm 1.M \cdot 2^{BE}$$

BE → binarni eksponent

"NORMALNA (znanstvena) NOTACIJA"

ukupno 4 bajta

PREDZNAK	K BE	MANTISA
1 bit	8 bita	23 bita

K = BE + 127

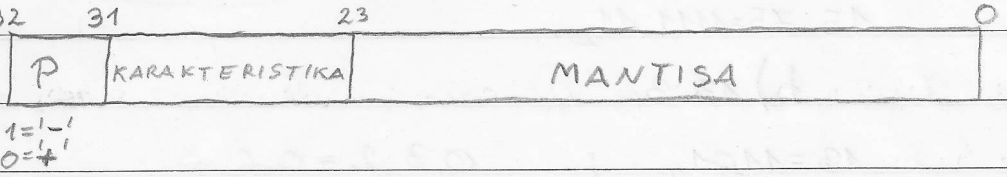
BE ∈ [-126, 127]

K ∈ [0, 255]

pohranjuje eksponent kao neutvrđen broj samo zbog brže usporedbe dva broja u procesoru

⇒ IEEE 754 je glavni standard za pohranu realnih brojeva

⇒ „K" je karakteristika broja (eksponent + 127) tako da imamo samo pozitivne brojeve od 1 do 254 (0, i 255 su sačuvani za posebne slučajeve



↳ NAPOMENA: u mantisi nema „skrivene jedinice"

⇒ dobro za programiranje:

↳ kad upisuješ „float" podatak, na kraj dodaj „f" ili „F" kako bi dodatno naznačio manji tip jer C po default-u radi s nekom vrstom „double" tipa.

↳ tako program radi jednu konverziju manje

PR: `float a;`
`a = 3.2f;`

PR: Kako bi računalo pohranilo 5.75?

$5.75 = 101.11_2 = 1.\underbrace{0111}_{\text{MANTISA}} \cdot 2^{\textcircled{2}\to BE}$ $\quad K = BE + 127 = 129 = 10000001$

P K M

|0| |1|0|0|0|0|0|0|1| |01110000000000000000000|

↳ to je ne čitljiva, pa za bolji prikaz koristimo heksadekatski zapis: $40B80000_{16}$

ZAD: Kako bi pohranilo neko računalo broj:

a) 2 $\quad 2 = 10_2 = 1.0 \cdot 2^1$ $\quad K = 128 = 10000000_2$

|0| |10000000| |00000000000000000000000|

40000000_{16}

b) -2 $[1]=|10000000|00000000000000000000000|$

$C0000000_{16}$

c) 5 $5=101_2=1.01\cdot 2^2$ $K=129=10000001$

$|0|10000001|01000000000000000000000|$

$40A00000_{16}$

d) 0.75 $0.75=0.11_2=1.1\cdot 2^{-1}$ $K=126=0111\ 1110$

$|0|0111\ 1110|1000\ 0000\ 0000\ 0000\ 0000\ 000|$

$3F400000_{16}$

⇒ za što se koriste $K=0$ i $K=255$? (ISPIT ∇)

	$M=0$	$M\neq 0$
$K=0$	koristi se za prikaz nule	zapisivanje još manjih brojeva od 2^{-126}
$K=255$	koristi se za prikaz $\pm\infty$	NaN

↳ radi se o denormaliziranom broju $\pm 0.M\cdot 2^{-126}$

PR: $2^{-130}=0.0001\cdot 2^{-126}$

|0|0000 0000|0001 0000 0000...000|

„not a number"

↳ raspon realnih brojeva: (slajd 77)

⇒ najveći broj koji možemo upisati u registar s „k" mjesta za karakteristiku:

$$2^{k-1}-1$$

→ ne pamti ∇ Uvod !

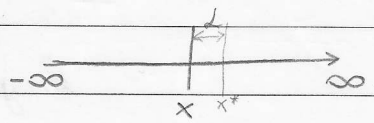
⇒ ukupan broj brojeva koji možemo zapisati je $255\cdot 2^{23}\cdot 2$

↳ sve ostale zaokružujemo

⇒ kolika je dozvoljena pogreška?

$X\Rightarrow$ željeni broj $X^*\Rightarrow$ približna vrijednost koju mogu upisati

↳ apsolutna pogreška: $\Delta=X^*-X$



↳ relativna pogreška: $\varepsilon=\frac{\Delta}{X}=\frac{X^*-X}{X}$

⇒ greška pri 4 bajta (relativna) ovisi o broju bitova mantise jer se dogadja na zadnjem → 24. bitu mantise:

$$|\varepsilon_{max}| = 2^{-24} \approx 6 \cdot 10^{-8} \quad \text{(relativna greška)}$$

$$|\Delta_{max}| = X \cdot 6 \cdot 10^{-8} \quad \text{(apsolutna greška)}$$

(slajd 82)

↳ 9 preciznih znamenaka

⇒ za „float" je stoga postavljen da ispisuje 6 decimala

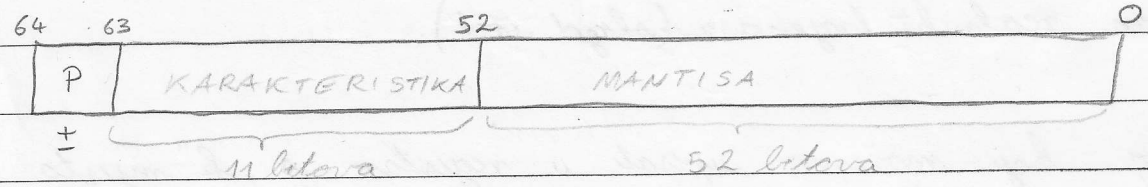
↳ float je dovoljno precizan na 7 znamenaka

(DOBRO ZA PRAKSU)

⇒ standard zaokružuje na „bliži" broj i tako smanjuje pogrešku

⇒ REALNI BROJ DVOSTRUKE PRECIZNOSTI:

↳ koristi se 8 okteta (bajta)



$$K = BE + 1023 \quad ; \quad BE \in [-1022, 1023]$$

$$|\varepsilon_{max}| = 2^{-53} \approx 1.1 \cdot 10^{-16}$$

$$|\Delta_{max}| = X \cdot 1.1 \cdot 10^{-16}$$

(slajd 94)

→ 17 preciznih znamenaka

⇒ C ne propisuje preciznost tipova „float", „double", „long double", ali je praksa: float → 32 bita

double → 64 bita

long double → 64, 80, 96, 128 bita

↳ varijabilno, nema standarda

⇒ zapis realnih konstanti za različite tipove: (slajd 98)

a) FLOAT: npr. 2.f ; 2.34F ; -1.34e5f

b) DOUBLE: npr. 5. ; 2.34 ; 9e-8 ; 8.34e+25

c) LONG DOUBLE: npr. 1.L ; 2.34L ; -2.5e-37L

⇒ za učitavanje kod „scanf" naredbe:

%f ⇒ float

%lf ⇒ double

↳ „long double" ne možemo učitat !?

⇒ za ispis kod „printf" naredbe:

↳ sve možemo ispisati sa %f, al ispis tipa „double" možemo i pomoću %lf

⇒ korisna naredba: int x, y;

y = sizeof(x); // y = 4

↳ broj bajtova od „x"

ZAMJENA ZA „LOGIČKI TIP PODATAKA"

⇒ poprima dvije vrijednosti: TRUE il FALSE

⇒ u „C"-u nema „boolean" varijable koja označava podatak logičkog tipa

↳ u „C"-u, svi brojevi različiti od nule, tretiraju se kao istina, a nula kao laž

PR: !4 = 0

!0 = 1

PR: if(!X) { ... } — isto kao: if(X==0) { ... }

⇒ C uključuje 3 logička operatora:

NE ⟶ !

I ⟶ &&

ILI ⟶ ||

PRETVORBA TIPOVA PODATAKA

PR: int i;
float f;
i = 2;
f = 2.99f;

↳ što je rezultat „i" + „f": 4 il 4.99

- kako zbrojit:

0000 0100 + |0|K|M| = ?

⇓ (upcasting - proširenje na „float"

→ proširenje

|0|K|M| + |0|K|M|

float float

⇒ računalo - manji tipove pretvara u veće, sve do onog najvećeg u izrazu

⇒ „short" i „char" se uvijek pretvaraju u „int" kada se s njima radi operacije (ako se ide na još „više" razinu, onda short i char također idu na „višu" razinu, npr. double)

PR: char a = 2;
char b = 3;
char c = a + b;

c = a + b

↓int ↓int

„int a" + „int b" = int c

↓char

„char c"

⇒ eksplicitna (zadana) pretvorba tipa:

(tip_podatka) varijabla

PR: (double) x

OSTALI OPERATORI

⇒ unarni operatori:

sizeof() ⇒ zauzeće memorije

(float), (int), (double) ⇒ pretvorba tipa

! ⇒ logičko ne

\+ ⇒ unarni plus (radi vrlo malo), vraća isti broj

↳ npr. char a = 'A';

int c = +a;

↘ ovaj znak vel: „želim da se ovaj „char" pretvori u „int", iako bi se i bez toga pretvorio" - koristi se za naglašavanje namjere up il down-castinga

\- ⇒ unarni minus, prebacuje broj u suprotni

++ ⇒ uvećanje broja za 1

-- ⇒ umanjenje broja za 1

↳ napomena: može se pisati i „++m" i „m++", ali to nije potpuno isto:

↳ npr. j = ++i; ⇒ i = i + 1;
j = i;

↳ npr. j = i++; ⇒ j = i;
i = i + 1;

↳ no, računalo ne garantira da prvo „i" ulaćuje u „j", pa onda računa - redoslijed naredbi ovis o compileru

↳ velik problem: i = 2i;
i = i++; → ne znamo je li i=2 il i=3

↳ izbjegavat mijenjat istu varijablu više puta unutar istog izraza;

↳ npr.: k = --i * i++;

⇒ binarni operatori:

>> ⇒ pomicanje bitova u desno

<< ⇒ pomicanje bitova u lijevo

& ⇒ binarno „I"

| ⇒ binarno „ILI"

^ ⇒ binarni "EX-OR"

~ ⇒ binarni "NOT" (invertor bitova)

↳ jedinični komplement: ~x = -x - 1

PR: Koliko je 10 & 7 ?

```
  0000 0000 0000 0000 0000 0000 0000 1010
& 0000 0000 0000 0000 0000 0000 0000 0111
-------------------------------------------
  0000 0000 0000 0000 0000 0000 0000 0010
```

10 & 7 = 2

⇒ binari operatori se koriste za pretvaranje charova u znamenke:

'7' ⇒ 0011 0111

& 0000 1111 → ova maska

0000 0111

↳ također, koristi se za maskiranje IP adresa, te npr. za provjeru je li ta adresa iz mojeg „subnetwork-a", to jest, mojeg poduzeća / tvrtke / škole ... ⇒ ako nije, sistem je hakiran (uz pretpostavku da nitko iz vanjske mreže ne smije pristupiti našoj) //

⇒ pomak bitova ⇒ pomak „n" bitova u desno je n-puta dijeljenje sa 2

⇒ pomak „n" bitova u lijevo je n-puta množenje sa 2

Pr: 0000 0010 << 2 = 0000 1000

2 8

⇒ ternarni operator: on je ustvari „if-else"

x = A ? B : C;

↳ if A

x = B;

else

x = C;

⇒ dobro za programiranje:

if(a % b)

↳ ako „a" nije djeljivo sa „b"

if(!a)

↳ ako je „a" jednak nul

⇒ SKRAĆENO PRIDRUŽIVANJE:

i = i + 3; ⇒ i += 3;

a = a & b ⇒ a &= b

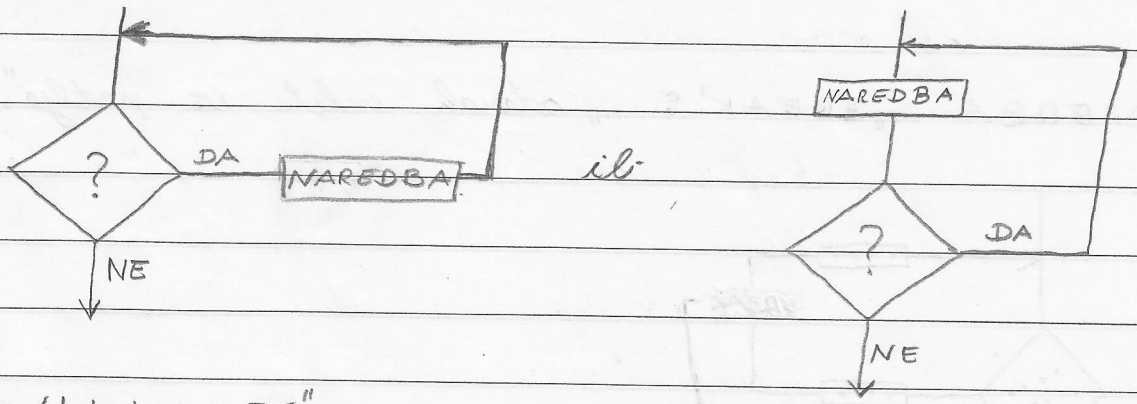
x = x / (a+b) ⇒ x /= a+b

⋮

KONTROLNE NAREDBE

⇒ „zakompliciraju" tok programa
↳ npr. „if" → grananje (dvostrano)

⇒ PROGRAMSKE PETLJE:



„WHILE" i „FOR"

„DO-WHILE"

⇒ „DO-WHILE":

```
do
{niz naredbi;}
while (uvjet);
```

⇒ „FOR": „while" s točnim brojem ponavljanja
↳ broj „i" inicijaliziraj na početku programa, a u petlji postavi na 0
↳ for (inicijalizacija; uvjet; korak)
{ ....
.... }
↳ inicijalizacija se izvršava samo jednom
↳ napomena: u jedan od izraza u zagradi „for" petlje ~~mije~~ obavezan

⇒ beskonačna petlja: koristi se „izrecama"

↳ NAREDBE:

break

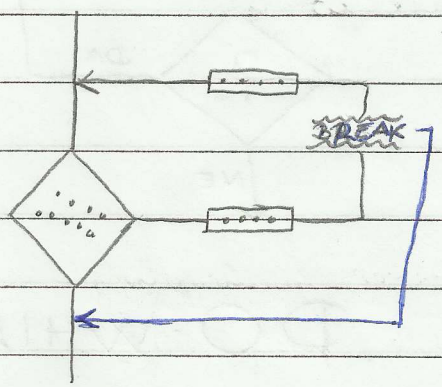
return

exit

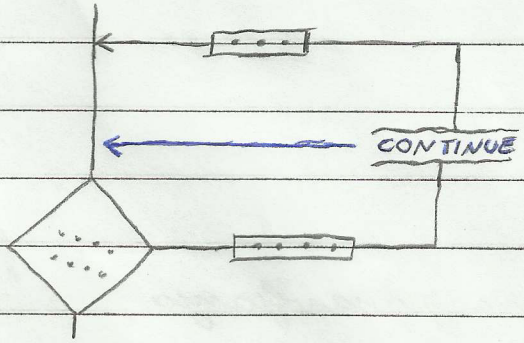
goto

↳ „CTR" + „C" - prekida beskonačnu petlju

⇒ NAREDBA „BREAK": „odmah izlet iz petlje"



⇒ NAREDBA „CONTINUE": „odmah se vrati na ulaz petlje"

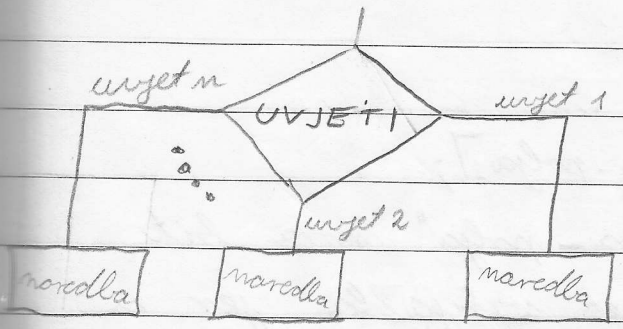


⇒ NAREDBA „GOTO": „odmah otiđi do zadanog"

```
goto oznaka_naredbe;
      ⋮
oznaka_naredbe:
nastavak naredbi...
```

↳ danas, u praksi nema razloga za korištenje „goto" naredbe (eventualno, ako se zapetljamo u više petlji i iz unutarnjih želimo „izletiti" više petlji van - tada umjesto više „break" naredbi možemo staviti „goto")

⇒ NAREDBA „SWITCH": (slajd 76)



⇒ SINTAKSA:

switch (cjelobrojni-izraz)
{case const

↳ napomena: ako se jedan uvjet zadovolji, C će proći kroz sve „case"-ove poslije ovoga i izvršiti njihove naredbe (to se naziva „fall through")
- da to izbjegnemo, svaki „case" mora imati „break"

POLJA

⇒ polje je inspirirano matematičkim pojmom niza
⇒ elemente se indeksiraju .

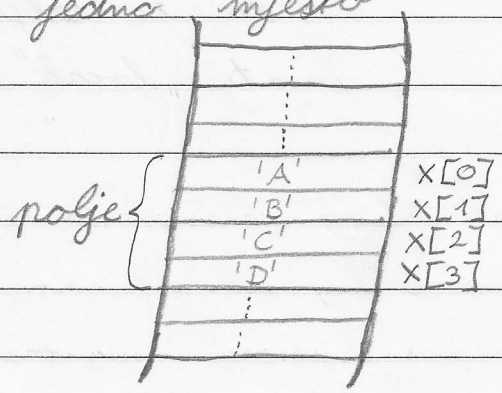
a[5] | a_0 | a_1 | a_2 | a_3 | a_4 |

⇒ kako definiramo polje?
↳ ovako:

tip ime[veličina_polja];

↳ napomena: „veličina_polja" mora biti konstanta, a ne varijabla jer u trenutku „compile"-iranja računalo mora znati koliko mjesta treba rezervirati za varijable („int a[4*2+8]" je dobro, „int a[X]" nije)

↳ napomena: polje se u RAM uvijek sprema na jedno mjesto



↳ napomena: C ne provjerava da li s prestupa polje ili ne, te ne baca grešku „index out of area boundries" već će pokupiti neko smeće iz memorije il prebrisati neku drugu varijablu - stoga, oprez!

⇒ inicijalizacija vrijednosti članova polja:

int x[4] = {1, 2}

↳ prva dva su postavljena na vrijednosti:

x[0] = 1

x[1] = 2

↳ druga dva su postavljena na nulu:

x[2] = 0

x[3] = 0

↳ ako ne definiramo ni jedan član polja, vrijednosti članova su neko „smeće" iz memorije

⇒ STRING ⇒ polje kao niz znakova / chorova

PR: char ime[] = {'I', 'v', 'a', 'n', '\0'};

↳ čvora polje od 5 znakova (čita se sve do null-znaka)

↳ ako se ne iskoriste sva mjesta u memoriji (svi članovi polja), ona se postavljaju na null-znak

PR: char ime[5] = "Ivan";

↳ programer osigurava mjesto za null-znak, pa je sigurnije pisati: char ime[] = "Ivan"; jer računalo automatski osigurava mjesto za null-znak

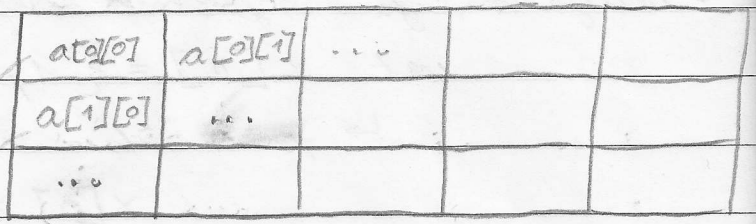
⇒ za učitavanje niza znakova koristimo naredbu: **gets**(ime);

↳ čita sve do „\n" (ENTER)

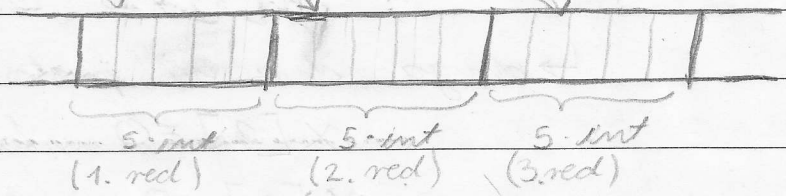
↳ automatski stavlja „\0" na kraj niza znakova

⇒ dvodimenzijska polja:

int a[3][5];



↳ pohranjivanje u memoriji:



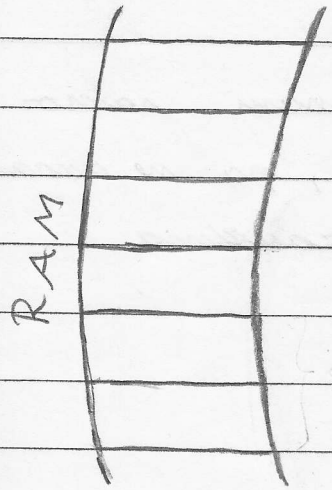
⇒ kod višedimenzijskih polja nije dopušteno ne-specificirati dimenzije

⇒ ispitivanje veličine polja: int a[6][2]

sizeof(a);

↳ veličina mog polja: 6 · 2 byte

POKAZIVAČI



⇒ 32-bitni sustavi koriste 32 bita za adresu, što je oko 4 milijarde adresa
↳ zato je za 32-bitni operacijski sustav maksimalno 4 GB RAM memorije
↳ primjer 32-bitne adrese:
0x1348AC25

⇒ kad najavimo varijablu, ona se „smješta" i sprema u RAM te dobiva svoju adresu / adrese
⇒ RAM se puni odozdo prema gore
↳ s koje strane se varijabla „smješta", ovisi o procesoru

⇒ POINTER ⇒ pokazuje na objekt u RAM-u

⇒ „%p" je adresa neke varijable
↳ npr. short a, b;
printf("%p %p", &a, &b);

napomena: u memoriji nije sve savršeno, to računala rade sprema na parne adrese i adrese djeljive s 4 (ima neki „hardverske-ske" razlog ⇒ zbog brzine)

⇒ kako spremit adresu?

↳ ako pišem ovako:

„pointer p = &a", znam samo mjesto, a ne veličinu, pa ne znam koliko mjesta varijabla zauzima

↳ zato se piše ovako:

$$\boxed{int \quad *p = \&a}$$

↳ il ovako: int* p = &a

↳ il ovako: int * p = &a

↳ tim načinom znam koliko bajtova (registara) varijabla zauzima, te njezinu adresu

⇒ svi pokazivači su iste veličine, ovisno o operacijskom sustavu:

..., 16-bita, 32-bita, 64-bita, ...

⇒ ako je pokazivač nerecijaliziran, on pokazuje na neko „smeće" u memoriji

↳ ako program nema pristup tom „smeću", onda se program ruši

⇒ pokazivač nekog tipa ne može pokazivat na varijablu nekog drugog tipa

⇒ pokazivač može pročitati što se nalazi na adresi na koju pokazivač pokazuje:

↳ to radimo pomoću DEREFERENTNOG OPERATORA: *

↳ npr. int a=5;

int *p=&a;

printf("%d", a); ⇒ ispiše se 5

printf("%d", *p); ⇒ isto se ispiše 5

⇒ najčešći način korištenja pokazivača je da pokazuje na neki element polja - tzv. „bookmark" polja

↳ puno bolje nego gibanje preko indeksa

⇒ ARITMETIKA S POKAZIVAČIMA:

↳ zbrajanje i oduzimanje ima smisla, a dijeljenje i množenje baš i ne

↳ kada napišemo ovo:

```
int *p;
int a;
p=&a;
p=p+2;
```

↓ pomiče se za dvije veličine tipa podatka (u ovom slučaju: 2 · sizof("int") = 8 bajta)

⇒ skraćena usmjerenje pokazivača na prvi član polja:

```
int x[4];
int *p;
p=x;
```

→ isto kao da piše: p=&x[0];

⇒ varijabla polja bez jedne dimenzije je ustvari adresa prvog elementa polja

Pr: `int x[8][7][3][4];`

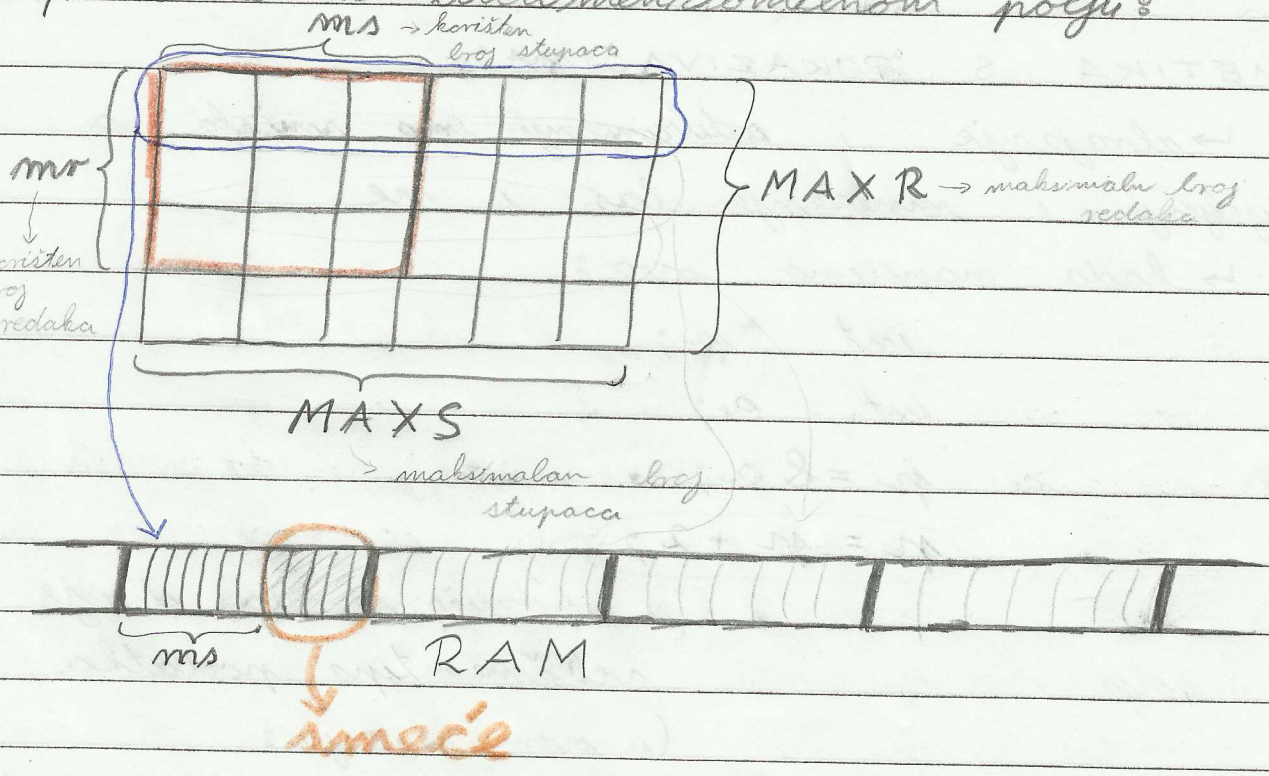
`int *p;`

`p = x[0][0][0];`

→ pokazuje na prvi element polja

⇒ odurzimanjem dva pokazivača istog tipa i istog polja saznajemo koliko je elemenata polja između ta dva pokazivača

⇒ pokazivači u dvodimenzionalnom polju:



a[i][j] ⇒ notacija koja olakšava programeru

↳ ram je jednodimenzionalan te ako preko pointera želimo doći do a[i][j] treba pisati: *a[0][0] + i * MAXS + j //

FUNKCIJE

⇒ koriste se kada vidimo da neku naredbu ponavljamo više puta

PR: Računanje binomnog koeficijenta:

↳ umjesto računanja faktorijela za svaku varijablu pomoću „for" petlje, možemo načinit funkciju:

```
double fakt(int n)
{ int i;
  double umnozak = 1.;
  for(i = 2; i <= n; ++i)
      umnozak = umnozak * i;
  return umnozak;
}
```

double: tip rezultata funkcije

int n: formalni argumenti

↳ „n" je varijabla koja uzima vrijednost od varijable ili konstante koju je koristio navedu u funkciju (uzima vrijednost stvarnih argumenata)

↳ ovaj „return" ne vraća operacijskom sustavu, već programu (drugoj funkciji)

⇒ funkcije se ne mogu definirati u drugoj funkciji, stoga ih navodimo prije int main(void)

{ ....

.... }

⇒ SVOJSTVA FUNKCIJA:

↳ stvarni argument mogu bit izrazi

↳ ako nema tipa funkcije, računalo pretpostavi da je „int"

↳ funkcija u pozivu može sadržat funkciju u sebi

↳ funkcija može bit dio izraza

⇒ funkcija koja nema argumenata:

PR: double vrati PI (void)

{ return 3.1415926; }

↳ poziv funkcije: vrati PI ( );

primjet: da u pozivu moramo staviti zagrade, ali ne navodimo argumente

⇒ FUNKCIJA TIPA „void":

↳ funkcija koja ne vraća rezultat

↳ na kraju funkcije može stajat „return" bez argumenta il jednostavno ništa

↳ preporučljivo je staviti return

⇒ kod strukturnog programiranja, preporučljivo je konstit jedan „return" po funkciji

⇒ ako tip podatka u „return" naredbi ne odgovara tipu funkcije, radi se „implicitni casting"

↳ PAZI NA GREŠKE! (casting mora bit moguć)

⇒ stvarni argument i formalni argumenti mogu dijelit ime, ali su on necvisn

⇒ FUNKCIJE KOJE VRAĆAJU VIŠE VRIJEDNOSTI: (ISPIT ☆)

↳ C nema „call by reference", već samo „call by value"

↳ no, možemo simulirat „call by reference" slanjem pokazivača funkciji koj sadrži adresu varijable koju želimo mijenjati

PR:
```
void sumaprod (int x, int y, int *psuma, int *pprod)
{ *psuma = x + y;
  *pprod = x * y;
  return; }

int main (void)
{ int x = 3, y = 4;
  int suma, prod;
  sumaprod (x, y, &suma, &prod);
  printf ("....", x, y, suma, prod);
  return 0; }
```

↳ isto tako, ako funkcija mora mijenjat varijable, ona mora primati pointere

ORGANIZACIJA SLOŽENIH PROGRAMA

⇒ funkcije se najavljuju prije glavnog programa kako bi compiler prije pozivanja funkcije znao koj tip treba očekivat

↳ ako baš želimo funkciju najavit poslje glavog programa, trebamo mu dat POTPIS il PROTOTIP funkcije:

„tip-funkcije" „ime funkcije" („ulazni parametri")

PR: `double fakt (int n);`

↳ il bez imena varijable:

`double fakt (int);`

⇒ prije no što se funkcija pozva prvi put, compiler je već trebao „naučit" (pročitat) il kompletnu funkciju il prototip funkcije (il oboje)

⇒ PRIMJENA: funkcije smještene u više datoteka

↳ kako će compiler provjerit tipove funkcija i argumenata?

↳ napravit ću funkcije u više datoteka, a glavni program u jednoj

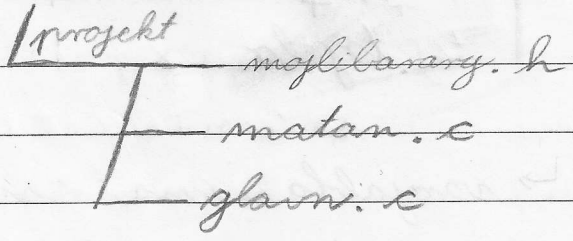
↳ zatim, onoliko koliko datoteka funkcija imam, toliko ".h" datoteka („header file") napravim sa prototipima funkcija

↳ zatim ih pomoću „#include"

najavom u funkcije

PR: #include <stdio.h>
#include "majlibarary.h"
↳ navodnici znače da je majlibarary.h u istoj mapi kao i ovaj .c program

⇒ header datoteke se ne compile-iraju
⇒ hijerarhija projekta:



⇒ kako compile-irati projekt s više datoteka?

1. način:

1) cc -ansi -Wall -pedantic-errors -o prog.exe majlibarary.c glavni.c
↳ jednom naredbom dovršeno prevođenje i povezivanje (linking)

2. način:

1) cc -ansi -Wall -pedantic-errors -c glavni.c
↳ izvorni kod prevoden u objektni kod (glavni.o)
2) cc -ansi -Wall -pedantic-errors -c matan.c
↳ izvorni kod prevoden u objektni kod (matan.o)
3) cc glavni.o matan.o -o prog.exe
↳ povezuje „glavni.o" i „matan.o" te kreira izvršni kod (prog.exe)

kod ovog načina ne greške dolazimo odmah, a ne sve odjednom

compile, don't link

⇒ DEFINICIJA I DEKLARACIJA:

↳ deklaracija je objava, bez detalja, bez načina izvođenja, ...

↳ definicija objašnjava i proces rješavanja nekog problema

	FUNKCIJA	VARIJABLA
DEKLARACIJA	prototip	
DEFINICIJA	prototip + tijelo	

↳ varijabla ima svoj SCOPE (opseg) - on govori o tome gdje je varijabla u programu vidljiva (bez pointera)

↳ varijabla ima i svoj DURATION (trajanje) - on govori o tome koliko varijabla traje, a nakon isteka tog trajanja, varijabla nestaje

PR: int i, j;
{ int i;
i = j; }

↳ vanjski „i" je nevidljiv u unutarnjem bloku

↳ unutarnji „i" traje za vrijeme trajanja bloka

SMJEŠTAJNI RAZRED	SCOPE	DURATION
auto (lokalne varijable)	unutar bloka (osim ako su preklopljene)	za vrijeme trajanja bloka
register (lokalne varijable)	smješta varijablu na memoriji procesora radi brzine izvođenja naredbi koje jako puno koriste tu varijablu (danas se to više ne koristi jer compiler sam zna procijeniti)	
static (globalna varijabla)	unutar opsega gdje je definirana	za vrijeme cijelog programa
extern (globalna varijabla)	gdje god je deklarirana (ili definirana)	za vrijeme cijelog programa

⇒ prava najava varijable je:

„smještajni_razred" „tip_varijable" „ime_varijable"

↳ ako ne pišemo smještajni razred, podrazumijeva se „auto"

⇒ static - te varijable se smještaju na poseban dio RAMa

- ne „umiru" kada završimo s blokom („zombie" varijabla)
- te varijable se automatski inicijaliziraju na nulu
- vidljivost:

a) definirana u bloku - od mjesta najave do kraja bloka

b) definirana izvan bloka - od mjesta najave do kraja file-a (modula / datoteke)

⇒ extern - „prava" globalna varijabla

- može se dijeliti među datotekama (nije baš preporučljivo u strukturnom programiranju)
- ona je jedina varijabla koja se može deklarirat
- ona se definira izvan blokova (i to bez riječi „extern")
- ona se automatski inicijalizira na nulu

Pr:

FILE 1

```
#define <stdio.h>
int x;
extern int a=0;
int main (void)
{ ....
  ....
  return 0; }
```

} definicija (izvan funkcije)

↓

ili bez „extern" i bez inicijalizacije ili sa „extern" i inicijalizacijom ali može i bez „extern" i s inicijalizacijom

FILE 2

```
int funkcija (....)
{
  extern int x;
  extern int a;
}
```

} deklaracija

↓

„nadi mi tu eksternu varijablu u drugim fileovima"

- ako eksternu varijablu koristimo unutar istog file-a u kojem je definirana, ne treba ga deklarirati

POINTERI I FUNKCIJE

$*(x+i) \equiv x[i]$

↳ to je prilično lijepo

↳ tako se krećemo po elementima polja pomoću pointera

⇒ što ako pomoću funkcije želim modificirat polje?

↳ ne mogu polje prenijeti u funkciju, pa znam da trebam pomoću pointera

⇒ SLANJE POLJA U FUNKCIJU:

1.) šaljem **pokazivač** na prvi član polja

2.) šaljem **duljinu** polja

↳ napomena: smijem koristit notaciju sa uglatim zagradama [ ]

↳ to je jer, iako radim s pokazivačima, lakše razmišljam s elementima polja umjesto pokazivača

PR:
```
void f(int *p, int n)
{ int i;
  for(i=0; i<n; i++)
      printf("%d ", p[i]);
  return; }
```

ili ovako

```
void f(int p[], int n)
{ ....
  ....
  return; }
```

⇒ notacija „int p[ ]" u najavi funkcije govori ovo: „radim s poljem - pretvaram se da sa kopiram funkciju u polje, iako nisam već sam samo poslao pointer"

↳ stoga: „int p[ ]" je isto pointer kao „int *p" u slučaju polja

↳ to se

⇒ kada napišemo: „int a = 5;" računalo radi ovako:

1) smjesti konstantu „5" u mjesto u memoriji za konstante

2) prekopira konstantu „5" na neko mjesto u „stog" memoriji

3) nazove to mjesto „a"

↳ takav rad možemo provjeriti pomoću pointera i vidjet ćemo da se pointeri dramatično razlikuju

↳ NAPOMENA: dio memorije za konstante ne smijemo mijenjat (možemo ga mijenjati pomoću pointera i neke funkcije koja mijenja nešto na što pointer pokazuje - tada će se program srušiti (čest uzrok BLUE SCREEN OF DEATH-a))

⇒ DVODIMENZIJSKA I VIŠEDIMENZIJSKA POLJA KAO ARGUMENTI FUNKCIJE:

↳ kod jednodimenzijskog polja, u majoni argumenata pišemo „int p[ ]" kao pokazivač na prvi član polja (vrijed i ovdje !)

↳ kod dvodimenzijskog polja je greška pisati ovako u majoni argumenata funkcije:

„int p[ ][ ]" ⇒ GREŠKA!

„int p[ ]" ⇒ DOBRO!

↳ u funkciji, možemo koristiti:

a) polje notaciju:

↳ p[i * maxst + j]

b) pointer notaciju:

*(m + i * maxst + j)

MACRO S PARAMETRIMA

⇒ koristi se u kompleksnim programima gdje se npr. ponavlja puno puta isto:

```
int main (void)
{ :
  a = vec (i, j);
  b = vec (2*m, 3+n);
  :
  :
}

int vec (int i, int j)
{
  return i>j ? i : j;
}
```

↳ recimo da ovaj program pomoću while petlje vrtimo 10^6 puta

↳ vidjet ćemo da je program spor jer za svaki put kada pozovemo funkciju, pauziramo glavni program

↳ stoga, umjesto funkcije možemo koristiti macro s parametrima:

```
#define VEC1 (a, b) a>b ? a : b
int main (void)
{ :
  a = VEC1(i, j);
  b = VEC1(2*m, n);
  c = 3 * VEC1(2, 5);
```

→ primjet grešku, očekivano c = 15,

↳ dešava se ovo: PREPISIVANJE:

PRIORITET ← c = 3 * 2 > 5 ? 2 : 5;

↳ kako se mogu desavati logičke greške zbog toga pišemo sve u zagradi (čak i oko parametara jer on mogu biti izraz):

```
#define VECI(a, b) ((a)>(b)?(a):(b))
```

⇒ vidimo da macro s parametrima nije ništa pametno, već samo prepisivanje

↳ zbog toga stavljamo sve u zagradu

⇒ na ispitu se pojavljuju krivo radani macro-i, pa treba pogoditi što će se desiti nakon izvršavanja programa

⇒ NAPOMENE:

a) između macro-a i parametara NE SMIJE biti razmak

PR: `#define MACI (a) ((-a))`

↳ na sva mjesta gdje je „MAC", prepisat će se „(a) ((-a))"

b) macro mora biti napisan u jednom redu, inače se javlja error, a ako baš želimo u novi red, najavi to sa znakom \ (backslash)

PR: `#define PI \`

REDEFINIRANJE TIPOVA

⇒ to se radi sljedećom notacijom:

```
typedef int cijeli;
cijeli a, b;
```

↳ ovo se koristi kada u početku nismo sigurni koji tip trebamo koristiti, pa ga poslije puno lakše to izmijenit

STRUKTURE

⇒ to je složen tip podatka čiji se element razlikuju po tipu

↳ ideja je da se nešto iz stvarnog svijeta prenosi u model u računalu

Pr:
```
struct osoba
{ char jmbg [13+1];
  char prezime [40+1];
  char ime [40+1];
  int visina;
  float tezina;
};
struct tocka t1, t2;
```

DEKLARACIJA "nacrt strukture"

↳ il ovako:
```
struct osoba
{ char jmbg [13+1];
  :
  float tezina;
} ; t1, t2, t3;
```

⇒ kako referenciramo pojedinu varijablu unutar strukture:
```
t1.tezina = 82;
t2.visina = 168;
```

⇒ programerima je dojadilo pisati:

```
struct osoba o1;
struct osoba o2;
struct osoba o3;
struct ...
struct ...
   ⋮
```

↳ stoga, počelo se pisati:

```
typedef struct o
   { char ime[13+1];
      ⋮
     float tezina;
   }; osoba;
```

↳ sada, kada nam treba poslovje samo napravimo:

```
osoba o1;
osoba o2;
osoba o3;
```

⇒ napomena: kod deklaracije strukture nije niti potrebno napisati ime strukture jer sama deklariramo kako ta struktura izgleda, pa možemo pisati:

```
typedef struct
   { char ime[13+1];
      ⋮
     float tezina;
   }; osoba;
```

UGRAĐENE FUNKCIJE

⇒ funkcija za apsolutnu vrijednost:

↳ prvo je bila u „stdlib.h" datoteci i izgledala je ovako:

```
int  abs(int x);
long labs(long x);
```

↳ kasnije, ta funkcija se stavila u „math.h" paket i izgleda ovako:

```
double fabs(double x);
```

↳ koristi se najčešće

↳ „long" i „long int" su danas ista stvar, pa nema neke zabune između „abs" i „labs"

⇒ „math.h" ima neke standardne matematičke funkcije (vidi podsjetnik i C reference card)

↳ par na tapet koje funkcije vraćaju

⇒ „stdlib.h":

a) funkcija „exit(x);"

↳ prekida program i vraća sustavu neki broj koji daje neku povratnu informaciju

↳ ako se exit nađe u bilo kojoj funkciji, ona i dalje ruši program i daje neki broj operacijskom sustavu

b) funkcija „int rand (void);"

↳ generira pozitivan broj iz intervala [0, RAND_MAX]

↳ nekad je RAND_MAX bio 32767 (danas je to 2147483647)

↳ ako bi u istom programu pokrenuli rand() u for petlji više puta, on bi svaki put dao iste brojeve

↳ zato, imamo funkciju:

void srand (unsigned int seed);

↳ parametar „seed" je broj koji pomaže „rand"-u da bude što nasumičniji

↳ najčešće se za „seed" koristi funkcija time(NULL) iz paketa „time.h"

↳ NAPOMENA: funkciji „srand" prethodi prije „rand" funkcije - iako se „rand" izvršava više puta, „srand" je dovoljno napisati jednom kako bi „uremirio" algoritam funkcije „rand" za generiranje nasumičnog broja

↳ NAPOMENA: time(NULL) vraća broj milisekundi od 1.1.1970.

↳ NAPOMENA: ako želimo, npr. nasumični broj iz intervala [a, b], koristimo formulu:

```
X = rand();
```

$$nasumični_{ab} = X \,\%\, (b - a + 1) + a$$

↳ 2. način za ograničavanje intervala „rand()" funkcije na [a, b]

```
X = rand();
```

$$nasumični_{ab} = \frac{(x-a)}{(b-a+1)} \cdot (d-c+1) + c$$

↳ preslikavanje broja iz intervala [a, b] u interval [c, d]

⇒ razlika između konstantnog znakovnog niza i niza znakova:

```
char *ime = "Ana";
```

↳ pokazivač na konstantu što memoriju

↳ ako preko pokazivača pokušamo mijenjati konstantu niz „Ana", program će se srušiti

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

↳ niz znakova

↳ normalno se mijenja

↳ konstantu niz se ne može mijenjati, a dobro ga je koristiti jer je on na posebnom mjestu u memoriji koji radi brže

UČITAVANJE I ISPIS PODATAKA

⇒ kernel → dio operacijskog sustava koji upravlja s hardwareom (pomoću drivera)

⇒ između „stdin" (tipkovnica) tijeka i operacijskog sustava (kernela) postoji međuspremnik – to se naziva BUFFER

↳ program uzima znakove iz tog međuspremnika (BUFFER-a), a ne direktno s tipkovnice

PR:
```
int main (void)
{ char d1, d2;
printf("Upiši prvi znak:");
scanf("%c", &d1);
printf("Upiši drugi znak:");
scanf("%c", &d1);
return 0; }
```

⇒ FUNKCIJA „GETCHAR"

↳ iz knjižnjice <stdio.h>

↳ prototip: int getchor(void);

↳ učitava jedan znak

↳ ako pročita znak: ^Z il 0x1A, to, znači EOF (end of file) što je najčešće broj „-1"

⇒ FUNKCIJA „PUT CHAR"

↳ iz knjižnice <stdio.h>

↳ prototip: int putchor(int ch);

↳ ispisuje jedan znak (vraća vrijednost uspješno upisanog znaka il vraća EOF ako ispis nije uspio)

⇒ napomena: pomoću „getchor" možemo upisat niz znakova pomoću petlje (npr, xchile(znak!=↵))

↳ također, pomoću „putchor" i petlje možemo čitati niz znakova iz buffera

(slajd 45)

⇒ čim pritisnem na neku tipku, vrijednost se pohranjuje u buffer - to se radi sve dok ne pritisnem na tipku ENTER

⇒ sa funkcijama „getchor" i „putchor" imamo apsolutnu kontrolu nad upisom i ispisom - možemo odredit koj znak predstavlja prekid upisa (dobre su za procesiranje na razini znaka)

⇒ FUNKCIJA „GETS"

↳ iz knjižnice <stdio.h>

↳ obično compileri javljaju warning jer korisnik mora biti disciplinovan - korisnik ne zna s kolikom veličinom polja radi

↳ „gets" čita sve znakove do prvog ENTERA ili EOF-a - na kraju se ENTER zamjenjuje s \0

↳ prototip: char *gets(char *s);

⇒ FUNKCIJA „PUTS"

↳ iz knjižnice <stdio.h>

↳ ispisuje sve znakove do '\0' ili vraća EOF ako ispis nije uspio

⇒ funkcije „gets" i „puts" su jednostavnije za rad s njima, ali ne pružaju sve funkcionalnosti kao „getchar" i „putchar"

⇒ FUNKCIJA „SCANF"

↳ kompleksnija funkcija od „getchar" i „gets"

↳ ona čita iz BUFFER-a u skladu s zadanim formatom i to razumijet kako bi znakove pohranio kao podatke

scanf (" _____ " , ...)

formatska specifikacija — niz pokazivača

↳ formatska specifikacija - govori funkciji kakve podatke očekujemo na ulazu

↳ prototip : int scanf(.....)

↳ primjetit da može vratit neki broj - broj uspješno interpretiranih varijabli

↳ vraća EOF ako ups nije uspješan

↳ KONVERZIJSKA (FORMATSKA) SPECIFIKACIJA:

% [širina] [modifikator] tip

TIP

d
o
x
u
} cijeli brojevi

e, f, g } realni brojevi

c } jedan znak

s } niz znakova

MODIFIKATOR

h } uz cjelobrojni tip - konverzija u „short int"

l } konverzija u long int (cjelobrojni) il konverzija u long double (realni)

ŠIRINA - maksimalni broj znakova iz „stdin"

⇒ kako računalo interpretira konverzijske specifikacije?

↳ NAPOMENA: sve osim %c

① preskoči sve praznine

② čitaj i stani kad:

a) ne razumiješ znak

ili

b) naletiš na prazninu

ili

c) je popunjena max. širina

↳ %c ne radi na tom principu - on će „progutati" bilo koji znak

⇒ NAPOMENA: ako kod formatske specifikacije stavimo prazninu:

scanf(" ␣ %d...) il scanf(" ␣ %c...)

↳ to znači da se preskoče sve praznine

↳ ovo vrijedi i za %c

↳ štoviše, ima smisla samo za %c jer brojevi ignoriraju preskoče praznine

⇒ NAPOMENA: scanf("%s",...) ista sama do prve praznine

↳ stoga, imamo trik:

scanf("%[^\n]", ...)

↳ sa znači: čitaj sve do znaka novog linea ('\n')

↳ dobro za programiranje:

- umjesto gets(niz) koristi konstrukciju %15[^\n] jer tako korisnik ne može pristupiti beskonačnom RAM-u, već samo 15 znakova

- napomena: ne dodaje '\0' na kraj niza (dodaj ga sam)

⇒ FUNKCIJA „PRINTF"

↳ konverzijska specifikacija:

% [znak] [širina] [preciznost] tip

znak: npr., poravnanje lijevo ili desno (vidi podjetnik)

širina: minimalna širina koja se rezervira i ispisuje na ekranu

preciznost: broj decimala

tip: tip varijable (slično kao kod scanf funkcije, samo što ima različite ispise za realan broj (e, f, g) te može ispisati pointer (p))

DATOTEKE

⇒ datoteka je skup bitova koj imaju smisao
↳ taj skup je negdje pohranjen i ima svoje ime

⇒ računalo, osim sa standardnim tokovima (stdin - tipkovnica, stdout - zaslon) komunicira i s datotekama

⇒ datoteke se spremaju u stalnu memoriju
↳ slijedni pristup podatcima (magnetska traka)
↳ s direktnim pristupom podacima (disketa, magnetski disk, SSD)

⇒ FILE SYSTEM ⇒ dio operacijskog sustava koj omogućuje korisniku da radi s sustavom mapa i datoteka, umjesto da direktno pristupa adresama sektora na hard-disku
↳ neki file sustavi: NTFS, FAT, EXT6, ...

⇒ RECORD (zapis) ⇒ skup susjednih podataka u datoteci koj imaju zasebno značenje

⇒ pomoću notacije FILE* [ime_toka] inicijaliziramo tok
↳ napomena: znak * ovdje ne predstavlja pokazivač na RAM

⇒ naredbom `tok Pod = fopen ("tekst.txt", "r");`
otvaramo tok do datoteke „tekst.txt" za čitanje
↳ "r" predstavlja read (čitanje)

⇒ dobro je provjerit je l tok otvoren:

```
if ( tok Pod == NULL)
        printf ("Tok je zatvoren.");
else
        printf ("Tok je otvoren.");
```

⇒ na kraju programa, zatvori tok!

```
fclose ( tok Pod);
```

⇒ konstrukt za `fopen ("...", "konstrukt");`

"r" ⇒ omogućava čitanje, vraća NULL ako datoteka ne postoj

"w" ⇒ omogućava pisanje, stvara datoteku ako datoteka ne postoj (napomena: ako je u datoteci bilo sadržaja, on se briše)

"a" ⇒ (append) omogućava pisanje na kraj datoteke, ako datoteka ne postoj, stvara datoteku

↳ '+' označava da je datoteka i ulazna i izlazna, jedina razlika je u ponašanju (ovisno o slovu: npr. "w+", "a+")

↳ za čitanje binarne datoteke, na kraj dodaj b (npr. "w+b")

⇒ zbog povijesnih razloga, ~~windows~~ i unix se razlikuju u notaciji za navođenje do datoteke:

a) UNIX:

```
fopen("c:/folder1/folder2/file.txt", "r");
```

b) WINDOWS:

```
fopen("c:\\folder1\\folder2\\file.txt", "r");
```

↳ to je zbog toga što C koristi backslash za oznaku specijalnog znaka, pa kako bi dobili jedan backslash, treba ga staviti dva

↳ NAPOMENA: preporučljivo je i na windowsima koristiti unix notaciju jer „pobrzi" kod compilera, a i kod je tada kompatibilan s unix operacijskim sustavom

⇒ DVIJE VRSTE DATOTEKA:

a) binarne

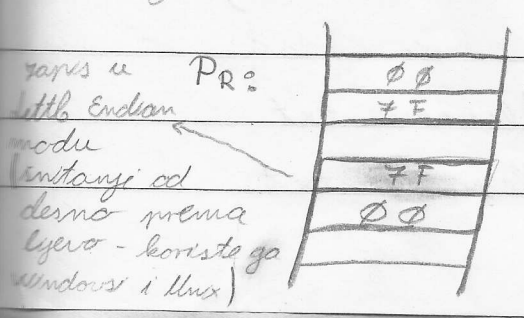
↳ prekopirani sadržaj RAM-a

↳ napomena: sve datoteke su, na neki način, binarne jer se sastoje od jedinica i nula

b) tekstualne

↳ čisti ASCII tekst

↳ može se urediti sa bilo kojem tekst editora



zapis u Big Endian modu (čitanje od lijevo prema desno)

short s = 127;

127 = 0000 0000 0111 1111

0 0 7 F

⇒ napomena: nizovi i polja se zapisuju „normalnim" (Big Endian) redoslijedom u RAM

⇒ napomena: binarna datoteka može imati bilo kakvu ekstenziju, ali je konvencija da ima „.bin" nastavak

↳ binarnu datoteku je najbolje otvoriti **hex-editorom**, jer ako otvorimo notepadom, on bi pokušao interpretirati taj niz znakova i dobili bi smeće

⇒ NAPOMENA: Windowsi koriste \r\n za novi red, a unix \n

↳ „C" compiler prepoznaje na kojem operacijskom sustavu radimo pa prilagodava zapis → stoga, tekstualne datoteke na windowsima imaju više bajtova nego iste u unixu - ovisno o broju „novih redova"

↳ kada na konstrukt za **fopen** funkciju dodamo 'b', onda isključujemo to ponašanje „C" compilera kako bi dobili točan binarni zapis iz RAM-a (bez '\r' znaka)

⇒ ako želimo neki file iz unixa prebaciti u Windowse i nemamo „novih redova", ond možemo iskoristiti program:

unix2DOS filename.txt

↳ obrnuto: DOS2unix filename.txt

⇒ FUNKCIJE ZA RAD S .TXT DATOTEKAMA:

① fgetc(tok) → čita jedan znak iz toka

② fscanf(.....) → scanf s nekim drugim parametrima za tok (slajd 17)

③ fgets(tok, n, niz) → razlikuje se od funkcije gets() po ovome:

a) „n" je maksimalni broj znakova koji će se učitati (nema kod gets())

b) ova funkcija ne mijenja '\n' sa '\0' već ga dodaje poslije znaka '\n'

④ fputc(....) → ispisuje jedan znak iz toka

⑤ fputs(....) → ispisuje niz znakova iz toka

⑥ fprintf(....) → printf za neki drugi tok

⇒ FUNKCIJE ZA RAD S BINARNIM DATOTEKAMA:

↳ najprimitivnije funkcije

① fwrite(void *ptr, size, n, tok)

pokazivač koji može pokazivat na bilo koji tip jer nije bitno za funkciju

↳ funkcija vraća broj objekata koji su uspješno upisani

② fread(void *ptr, size, n, tok)

↳ čita objekte iz toka

↳ vraća broj uspješno pročitanih objekata

↳ ne provjerava je li tip u RAM-u onaj tip koji želimo čitati (kao većina ovih funkcija)

⇒ NAPOMENA:

↳ u radu s binarnim datotekama najčešće se koriste strukture

↳ to je zbog toga što struktura uvijek zauzima isti broj bajtova u RAM-u, te ne moramo paziti koliko bajt tip i kada se bajt tip pojavljuje u binarnoj datoteci

⇒ FILE *nešto ⇒ to nije pointer, već tak, al u neku ruku je pointer zato što pamti gdje smo stali u datoteci

↳ stoga, čitanje i pisanje se vrši od trenutne pozicije u datoteci

⇒ FUNKCIJA:

- int fseek (FILE *stream, long offset, int whence)

↳ funkcija iz paketa <stdio.h>

↳ offset ⇒ pozicioniranje na taj znak od „whence"

↳ whence:

SEEK_END - stavlja pokazivač na kraj datoteke

SEEK_CURRENT - trenutna pozicija

SEEK_SET - stavlja pokazivač na početak datoteke

↳ vraća 0 ako je pomicanje uspjelo

- long ftell (FILE *stream)
  ↳ vraća trenutnu poziciju u datoteci izraženu u broju bajtova

⇒ napomena: fseek (...) najčešće koristimo s binarnim datotekama jer kod njih puno lakše znamo što se nalazi gdje nego u .txt datoteci

⇒ TOK PODATAKA „stderr":
  ↳ on je (standardni) tok koj se otvara paralelno s standardnim izlazom
  ↳ njegova zadaća je ispisati greške na ekran bez obzira da li smo preusmjerili izlaz programa u neku datoteku

⇒ kako doznat što je program vratio operacijskom sustavu preko naredbe **return** ili **exit**?

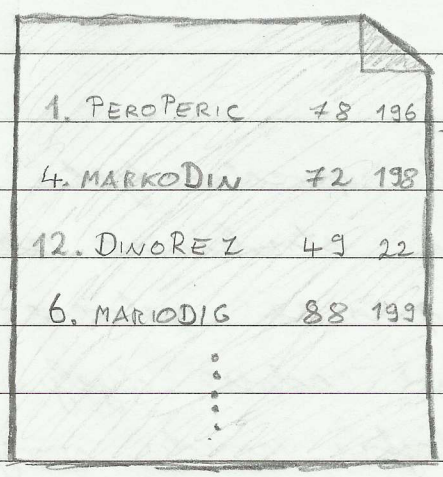
⇒ NAPOMENA: teško je unaprijed predvidjeti koliko bajtova neka struktura zauzima jer računala vol parne adrese, a za decimalne brojeve adrese djeljive s 4 pa stoviše parne bajtove između ponekad ubacuju (PADDING)

SLIJEDNE I DIREKTNE DATOTEKE

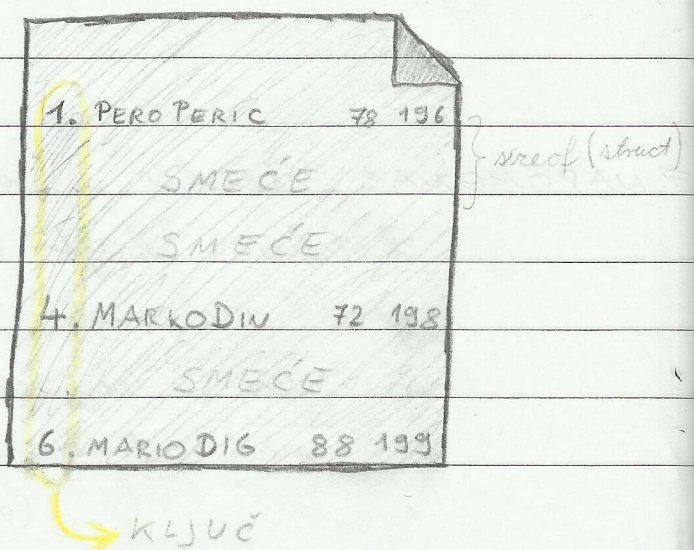
⇒ slijedne datoteke - podac su slijedno zapisani, pa pri čitanju moramo slijedno čitati sve dok ne „naletimo" na podatak koj nas zanima
↳ sporo s velikim datotekama

⇒ direktne datoteke - podac su „pametno" zapisani te točno znamo gdje je koj podatak zapisan, pa možemo koristit fseek (...) funkciju za kretanje do podatka koj nas zanima
↳ brzo čitanje, al ako svi redni brojevi (ključevi) nisu popunjeni, datoteka je bespotrebno velika (smeće izmeću varijabli)

SLIJEDNA DATOTEKA



DIREKTNA DATOTEKA



⇒ napomena: Windows dodaje neke dodatne znakove u tekstualne datoteke (npr. '\r'), pa je direktna datoteka gotovo isključivo binarna, jer moramo uvijek točno znati koliko je bajtova između dva podatka

⇒ napomena: kod direktnih datoteka, „smeće" između podataka su najčešće nule