Programarea Calculatoarelor
Cursul 2

Tipuri de date
Expresii
Instrucțiuni de decizie

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Obiective

- Cum stochează calculatorul datele necesare
  - un număr întreg? o literă? un număr real? un text?
- Cum putem să facem calcule folosind formule?
- Cum scriem cod care se ramifică?

Sisteme de numerație

- Calculatorul folosește circuite cu 2 stări pentru a stoca datele - se utilizează baza 2 (binară)
- Oamenii folosesc baza 10 (zecimală)
- Un bit este o cifră binară (binary digit)
- Un byte (sau un octet) este format din 8 bits

Unitate	Denumire	Număr bytes [10^x]	Număr bytes [2^x]
1 KB	kilobyte	$1024 \sim 10^3$	2^{10}
1 MB	megabyte	$1024^2 \sim 10^6$	2^{20}
1 GB	gigabyte	$1024^3 \sim 10^9$	2^{30}
1 TB	terabyte	$1024^4 \sim 10^{12}$	2^{40}
1 PB	petabyte	$1024^5 \sim 10^{15}$	2^{50}

Conversie baza 10 - baza 2

- Se face prin împărțire repetată cu 2
  - se scrie câtul dedesubt și restul în dreapta
  - cifrele în binar sunt resturile în ordine inversă
- Exemple, $157_{(10)} = 10011101_{(2)}$, $64_{(10)} = 2^6 = 1000000_{(2)}$

157	1
78	0
39	1
19	1
9	1
4	0
2	0
1	1
0	

64	0
32	0
16	0
8	0
4	0
2	0
1	1
0	

Conversie baza 2 - baza 10

- Cifrele în binar au pondere egală cu puteri crescătoare ale lui 2, începând cu 2^0, de la cel mai nesemnificativ bit
- Exemple,

$$10011101_{(2)} = 1*2^7+0*2^6+0*2^5+1*2^4+1*2^3+1*2^2+0*2^1+1*2^0 = 157_{(10)}$$

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

$$1000000_{(2)} = 1*2^6 = 64_{(10)}$$

putere	6	5	4	3	2	1	0
cifrele	1	0	0	0	0	0	0

Reprezentare numere întregi pozitive

- Numerele pozitive sunt reprezentate folosind reprezentarea lor binară
- Biții sunt grupați în bytes
  - numărul de biți divizibil cu 8 - se completează cu 0-uri
  - se păstrează în memorie prima data byte-ul cel mai puțin semnificativ (little-endian)
- Cu n biți putem reprezenta 2^n valori posibile
  - cel mai mic număr: 0 are reprezentarea 00...0 (n biți de 0)
  - cel mai mare număr: 2^n-1 are reprezentarea 11...1 (n biți de 1)
- Operațiile de aritmetice (adunare, scădere, înmulțire) în baza 2 se efectuează analog ca în baza 10
  - se adaugă transport, respectiv se împrumută când trecem peste 2 nu peste 10

Reprezentare numere întregi pozitive - exemplu

314 stocat ca short int = 16 biți = 2 bytes

$314_{(10)} = 100111010_{(2)}$ = `0000 0001` `0011 1010`

byte 1 byte 0

În memorie se stochează în ordine byte 0, byte 1

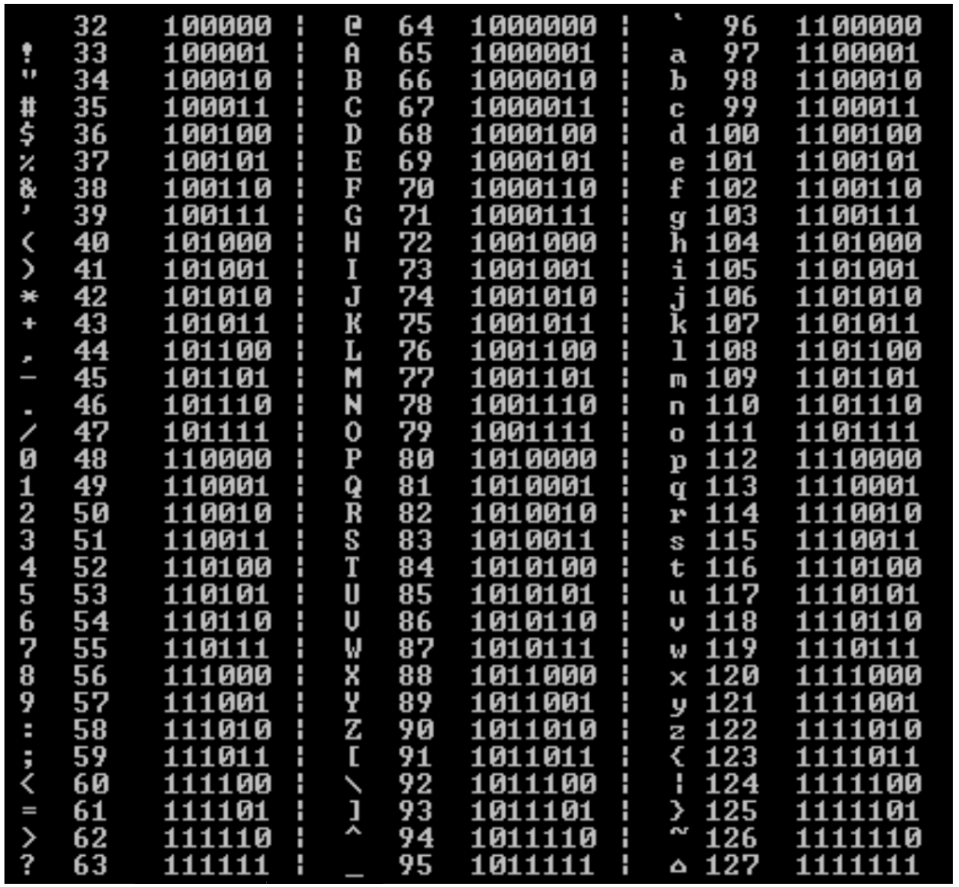
`0011 1010` `0000 0001`

Baza octală și cea hexazecimală

- Metodele de conversie prezentate se pot aplica și pentru conversii în alte baze
- Pentru baze de forma 2^k există metode mai ușoare de conversie din și în binar
- Deoarece reprezentarea binară este lungă este de multe ori convenabil să schimbăm în baza octală (8) sau cea hexazecimală (16)
  - câte 3 cifre binare corespund la o cifră octală
  - câte 4 cifre binare corespund la o cifră hexazecimală
  - un byte = 2 cifre hexazecimale
- Exemplu, $1001\ 1101_{(2)} = 9D_{(16)} = 10\ 011\ 101_{(2)} = 235_{(8)}$

Reprezentare caractere

- Fiecare caracter este asociat cu un număr întreg = Cod ASCII
- Ordinea caracterelor se bazează pe acest cod
- Se pot efectua operații ca pe numerele întregi
  - ‘a’-’A’ = 32
- 0-31 caractere speciale:
  - backspace 8
  - line nouă 10
  - escape 27



	32	100000	@	64	1000000	`	96	1100000
!	33	100001	A	65	1000001	a	97	1100001
"	34	100010	B	66	1000010	b	98	1100010
#	35	100011	C	67	1000011	c	99	1100011
$	36	100100	D	68	1000100	d	100	1100100
%	37	100101	E	69	1000101	e	101	1100101
&	38	100110	F	70	1000110	f	102	1100110
'	39	100111	G	71	1000111	g	103	1100111
(	40	101000	H	72	1001000	h	104	1101000
)	41	101001	I	73	1001001	i	105	1101001
*	42	101010	J	74	1001010	j	106	1101010
+	43	101011	K	75	1001011	k	107	1101011
,	44	101100	L	76	1001100	l	108	1101100
-	45	101101	M	77	1001101	m	109	1101101
.	46	101110	N	78	1001110	n	110	1101110
/	47	101111	O	79	1001111	o	111	1101111
0	48	110000	P	80	1010000	p	112	1110000
1	49	110001	Q	81	1010001	q	113	1110001
2	50	110010	R	82	1010010	r	114	1110010
3	51	110011	S	83	1010011	s	115	1110011
4	52	110100	T	84	1010100	t	116	1110100
5	53	110101	U	85	1010101	u	117	1110101
6	54	110110	V	86	1010110	v	118	1110110
7	55	110111	W	87	1010111	w	119	1110111
8	56	111000	X	88	1011000	x	120	1111000
9	57	111001	Y	89	1011001	y	121	1111001
:	58	111010	Z	90	1011010	z	122	1111010
;	59	111011	[	91	1011011	{	123	1111011
<	60	111100	\	92	1011100	¦	124	1111100
=	61	111101	]	93	1011101	}	125	1111101
>	62	111110	^	94	1011110	~	126	1111110
?	63	111111	_	95	1011111	⌂	127	1111111

www.asciitable.com

Numere întregi negative

- Pentru a reprezenta atât numerele negative cât și cele pozitive un bit trebuie rezervat pentru semn
- Reprezentare cu bit de semn și magnitudine
  - primul bit arată semnul, restul biților formează numărul
  - NU se folosește
  - 0 are două reprezentări 00...0 și 10...0
  - se modifică regulile de adunare
- Reprezentare C2
  - cel mai semnificativ bit are pondere negativă egală cu -2^{n-1}
  - se utilizează
  - 0 are o singură reprezentare
  - regulile de adunare rămân la fel
    - x + (-x) în această reprezentare rezultă în 0 (cu overflow)

Numere întregi negative - Complement față de 2

- Reprezentare C2
  - Dacă se folosesc n biți
    - Numere pozitive de la
    - 000...0 = 0 până la 011...1 = 2^{n-1}-1
    - Numere negative de la
    - 100...0 = -2^{n-1} până la 111...1 = -1
- Conversie rapidă număr negativ în reprezentare C2
  - Se convertește valoarea în modul în binar
  - Se adaugă zerouri în față până la n biți
  - Se schimbă starea biților, 1 în 0 și 0 în 1 (complement față de 1)
  - Se adună 1 la rezultatul în binar
  - Exemplu, pe 4 biți

număr	reprezentare C2
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111

$$-6 = \sim 0110_{(2)} + 1_{(2)} = 1001_{(2)} + 1_{(2)} = 1010_{(2)}$$

Numere întregi negative - Complement față de 2

- Verificare pentru x = 6 pe 4 biți

```
 6 =   0110
-6 =   1010
-------------
 0 =  10000
```

- se iau în considerare doar ultimii 4 biți

- Pentru *x* general pe *n* biți
  - $x + -x = x + (\sim x + 1) = (x + \sim x) + 1 = 2^n - 1 + 1 = 2^n = 1000...0$

    1 urmat de *n* biți de 0
  - se iau în considerare doar ultimii *n* biți atunci devine 0

Reprezentare numere întregi negative - exemplu

-108,828 ca un int = 32 biți = 4 bytes

Convertim valoarea în modul în binar

$108,828_{(10)}$= 1 1010 1001 0001 1100 $_{(2)}$ = $1A91C_{(16)}$

Completăm cu 0-uri până la 32 biți

`0000 0000 0000 0001 1010 1001 0001 1100`

Efectuăm complement față de 1

`1111 1111 1111 1110 0101 0110 1110 0011`

Adunăm 1 în binar

`1111 1111 1111 1110 0101 0110 1110 0100`

Reprezentarea în memorie (se începe de la byte-ul 0)

`1110 0100 0101 0110 1111 1110 1111 1111`

Conversie baza 10 - baza 2 - numere subunitare

- Se face prin înmulțire repetată cu 2
  - se separă partea fracționară de cea întreagă
  - partea fracționară se copiază dedesubt
  - se repetă până când partea fracționară devine 0, sau până la numărul de cifre disponibile
  - cifrele după virgulă sunt formate din părțile întregi obținute pe parcurs, în ordinea originală
- Exemplu, $0.3125_{(10)} = 0.0101_{(2)}$
  - 0.3125 x 2 = **0** + 0.625
  - 0.625 x 2 = **1** + 0.25
  - 0.25 x 2 = **0** + 0.5
  - 0.5 x 2 = **1** + 0

Conversie baza 2 - baza 10 - numere subunitare

- Cifrele în binar din dreapta virgulei zecimale au pondere egală cu puteri descrescătoare ale lui 2, începând cu 2^{-1}
- Exemple,

$$0.0101_{(2)} = 0*2^{-1}+1*2^{-2}+0*2^{-3}+1*2^{-4} = 1/4 + 1/16 = 0.3125_{(10)}$$

putere	-1	-2	-3	-4
cifrele	0	1	0	1

$$0.11111_{(2)} = 1*2^{-1}+1*2^{-2}+1*2^{-3}+1*2^{-4}+1*2^{-5} = 1-2^{-5} = 0.96875_{(10)}$$

putere	-1	-2	-3	-4	-5
cifrele	1	1	1	1	1

Consecințe

- Numerele zecimale generale se convertesc prin aplicarea primei metode pe partea întreagă și celei de a doua metode pe partea fracționară
- Nu toate numerele zecimale din baza 10 au reprezentare finită în baza 2
  - $3/10 = 0.3_{(10)} = 0.0(1001)_{(2)}$ - reprezentare periodică infinită
  - doar numerele care se pot scrie ca *a/b*, unde *b* este 2^k, au reprezentare finită
- Numerele reale în general (radical(2), pi, e) au un număr infinit de cifre
  - se aproximează cu cele mai importante cifre
- Nu se poate verifica egalitatea între două numere flotante
  - se verifică dacă diferența dintre ele este mică

Numere cu virgulă flotantă

- Reprezentarea cu virgulă flotantă a unui număr x implică aducerea lui la o formă standard:

$$x = \pm a.bcde\ldots\ baza^{putere}$$

- Intuitiv, când lucrăm cu un număr ne interesează doar semnul, cifrele cele mai semnificative, și cât de mare este
  - Exemplu, distanțe dintre 2 orașe 120 km = 1.2×10^5 m
  - Exemplu, sarcina unui electron = $-1.60217662 \times 10^{-19}$ Coulomb
- În procesul transformării mutăm virgula zecimală la poziția corectă prin înmulțire cu *baza* la o putere

Conversie număr zecimal în virgulă flotantă

- Tipurile pentru numere reale (float, double, long double) folosesc reprezentarea cu virgulă flotantă în baza 2
- Este rezervat
  - 1 bit pentru semn,
  - *a* biți pentru exponent (putere) și
  - *b* biți pentru mantisă (cifrele cele mai importante)
- Pentru conversie numărul trebuie adus la forma

$$x = \pm 2^{exponent-deplasament} 1.abcde\ldots$$

1. Conversie din baza 10 în baza 2
2. Mutare virgulă prin înmulțire cu 2^k
   a. k poate fi negativ, pozitiv sau 0
3. Se calculează exponentul prin adăugarea deplasamentului la *k* și se convertește în binar

Reprezentare număr zecimal în virgulă flotantă - exemplu

- 132.57 stocat ca float = 1 bit semn, 8 biți exponent, 23 biți mantisă, deplasament $01111111_{(2)} = 127_{(10)}$

$132.57_{(10)} = 10000100.10010001111011100\ldots_{(2)}$

$\sim 10000100.10010001111011100_{(2)} =$

k este 7

$= 1.00001001001000111101100_{(2)} \times 2^{7} =$

$= 2^{134-127} \times 1.00001001001000111101100_{(2)}$

$= 2^{10000110_{(2)}-127} \times 1.00001001001000111101100_{(2)}$

bit semn 0 pozitiv, 1 negativ; biți exponent; biți mantisă - se ignoră cifra de 1 din stânga

- Cei 4 bytes ai reprezentării (în memorie octeții se stochează invers):
  - 0100 0011 0000 0100 1001 0001 1110 1100

Tipuri de date - întregi

tip	octeți*	minim	maxim
unsigned char	1	0	255 = 2^8-1
unsigned short int	2	0	65,535 = 2^{16}-1
unsigned int	4	0	4,294,967,295 = 2^{32}-1 ~ 4 x 10^9
unsigned long long int	8	0	18,446,744,073,709,551,615 = 2^{64}-1 ~ 10^{19}

tip	octeți*	minim	maxim
char	1	-128 = -2^7	127 = 2^7-1
short int	2	-32,768 = -2^{15}	32,767 = 2^{15}-1
int	4	-2,147,483,648 = -2^{31}	2,147,483,647 = 2^{31}-1 ~ 2 x 10^9
long long int	8	-9,223,372,036,854,775,808 = -2^{63}	9,223,372,036,854,775,807 = 2^{63}-1 ~ 9 x 10^{18}

* dimensiunea în octeți se dă pentru Windows 64 biți, poate fi diferită pe alt sistem de operare
* long long int este introdus în standardul C99

Tipuri de date - reale

tip	octeți*	exponent	deplasament	mantisă	minim normal	maxim normal	precizie (nr. cifre)
float	4	8	$127 = 2^7-1$	23	1.17×10^{-38}	3.40×10^{38}	6
double	8	11	$1023 = 2^{10}-1$	52	2.22×10^{-308}	1.79×10^{308}	15
long double	16	15	$16383 = 2^{14}-1$	64	3.7×10^{-4932}	1.2×10^{4932}	18

- minim normal: exponentul = 1, toți biții din mantisă egali cu 0
  - $\sim 2^{-\text{deplasament} + 1}$
- maxim normal: exponentul = maxim – 1, toți biții din mantisă egali cu 1
  - $\sim 2^{\text{deplasament} + 1}$
- precizie = câte cifre este capabil să păstreze în mod corect
  - $10^{\text{precizie}} \sim 2^{\text{biți mantisă}}$ deci precizie = (biți mantisă) $\log_{10}2$
- Dacă exponentul conține doar biți de 1 – rezervat pentru infinit și not-a-number
- Dacă exponentul conține doar biți de 0 – numere subnormale = mantisa începe cu bit 0

* dimensiunea în octeți se dă pentru Windows 64 biți, poate fi diferită pe alt sistem de operare
* long double este introdus în standardul C99

https://en.wikipedia.org/wiki/IEEE_754 https://float.exposed/

Depășirea limitei - Overflow

- Tipurile au un domeniu finit
- Dacă se depășește limita maximă se întâmplă overflow
- Pentru tipuri întregi fără semn pe *n* biți
  - se păstrează doar ultimii *n* biți din răspuns, se aruncă biții mai mari
- Pentru tipuri întregi cu semn
  - la fel ca la cele fără semn dar răspunsul poate fi și negativ conform reprezentării în complement față de 2 (C2)
- Pentru tipuri reale se obține infinit
- Se întâmplă analog ca la depășirea limitei inferioare (underflow)

Depășirea limitei - Overflow - exemple

```
//intregi fara semn
unsigned int a = 4294967295;
a = a + 7;
printf("%u\n", a);
```

- a conține valoarea maximă pentru unsigned int
- a+1 ar fi 2^{32} care are biții 31, 30, ..., 0 egali cu 0
- a+7 este 2^{32} + 6 care are biții 31, 30, ..., 0 egali cu ...0110
- 6 fiindcă se aruncă bitul 32

```
//intregi cu semn
int b = 2147483646;
b = b + 10;
printf("%d\n", b);
```

- b conține valoarea maximă pentru int -1
- b = 0111 1111 1111 1111 1111 1111 1111 1110 $_{(2)}$
- b = 1000 0000 0000 0000 0000 0000 0000 1000 $_{(2)}$
- -2147483640 = -2^{31} + 8 fiindcă se interpretează în C2

```
//reali
float c = 1e20f;
c = c*c;
printf("%f\n", c);
```

- se inițializează c cu notație științifică
- c = 10^{20}
- c = 10^{40} - nu se poate reprezenta ca float - overflow
- inf

Expresii - introducere

- O **expresie** este
  - o variabilă sau o constantă x sau 14
  - un operator unar și o expresie -x sau +14
  - un operator binar și două expresii x+y sau x*14
  - un operator ternar și trei expresii x ? 0 : 1
- Practic, este o formulă care se evaluează la o valoare în momentul execuției programului
- În general formulele matematice se transcriu aproape identic ca expresii în C
- Exemplu, x + 3*y - 5*(z+4)

Clasificare operatori

- Aritmetici
  - + - / * %
  - rezultatul operației este un număr
  - % este restul împărțirii = modulo
  - nu există ridicare la putere
- Relaționali
  - > >= < <= == !=
  - rezultatul operației este adevărat 1 sau fals 0
  - == verifică egalitate, = este pentru atribuire, != diferit
- Logici
  - ! && ||
  - ! - not, negare logică, && - și logic, || - sau logic
  - rezultatul operației este adevărat 1 sau fals 0
- Incrementare, decrementare (post și pre)
  - ++ --

Evaluarea expresiilor

- Dacă expresia este o constantă sau o variabilă valoarea ei este egală cu valoarea constantei/variabilei și se păstrează tipul
- Dacă expresia este formată din mai mulți operanzi de același tip atunci rezultatul va fi tot de acest tip
- Dacă expresia este formată din mai mulți operanzi de tipuri diferite atunci tipurile cu domeniu mai mic sunt automat (implicit) convertite în tipuri cu domeniu mai mare și rezultatul va fi de acest tip = **conversie implicită**
  - ordinea tipurilor de la domeniu mic la domeniu mare:

char -> unsigned char -> short -> unsigned short -> int -> unsigned int -> long long -> unsigned long long -> float -> double -> long double

Evaluarea expresiilor

- În cazul în care toți operanzii sunt întregi se efectuează promovarea întregilor
- Tipurile de date cu rang mai mic decât int, cum sunt tipurile de date char și short, sunt promovate la int
- În cazul variantei fără semn (unsigned) acestea sunt promovate la unsigned int

```
char x = 120;
char y = 110;
int z = x + y;
printf("%d\n", z); // 230
```

Evaluarea expresiilor

- Putem schimba tipul unei expresii în mod forțat prin **conversie explicită** folosind operatorul de cast
- Se precedă expresia cu (tip), unde tip este orice tip de date
- Exemplu, `float x = 1 / (float) 2;`
  - fără conversie explicită x avea valoarea 0
- Exemplu, `float x = (float) 5;`
  - în acest caz și dacă omitem operatorul de cast conversia se întâmpla automat prin conversie implicită
- Exemplu, `int x = (float) 1;`
  - se convertește 1 în float în mod explicit, apoi se convertește în int în mod implicit

Operatori aritmetici

```
//operatori unari
-a
+-6
```

- dacă a este 1, -a este -1; dacă a este -2, -a este 2; 0 neafectat
- este tot -6, operatorul unar + este doar cosmetic

```
//operatori binari
int x = 6-7;
int y = 1/2;
float z = 1/2;
float u = 1.f/2;
float v = 1/2.f;
float w = 1/2.0;
float q = (float)1/2;
```

- -1, două constante de tip int, rezultatul tot de tip int
- 0, se calculează câtul împărțirii dacă ambii operanzi sunt int
- 0.f, rezultatul împărțirii este tot int, care apoi este convertit la float
- 0.5f, operandul 2 de tip int este convertit implicit la float
- 0.5f, operandul 1 de tip int este convertit implicit la float
- 0.5f, rezultatul 0.5 de tip double este convertit implicit la float
- 0.5f, operandul 1 de tip int este convertit explicit la float

```
int a = -1/2;
int b = -7/4.0;
int c = 5%3;
int d = -5%3;
```

- 0, se calculează câtul împărțirii
- -1, rezultatul -1.75 este trunchiat, se aruncă partea fracționară
- 2, returnează restul împărțirii - ambii operanzi trebuie să fie întregi
- -2, pentru numere negative -a%b = -(a%b), deși -5 mod 3 = 1

Operatori aritmetici - împărțire la 0

```
//operanzi intregi
int a = 5;
int b = a/0;
```

- nu este eroare de compilare
- eroare la rularea programului, nu se poate calcula pentru întregi

```
//operanzi reali
float x = 5.0f;
float y = x/0;
float z = -x/0.f;
```

- se poate face împărțire cu 0 pe tipuri reale
- inf, valoare specială rezervată pentru numere peste limita maximă
- -inf, valoare specială pentru numere sub limita minimă

```
//not a number
```

- valoare specială pentru o operație al cărui rezultat nu se poate evalua
- nan, caz nedeterminat

```
float t = y+z;
float w = y*z;
float s = y/z;
```

- -inf, se poate calcula
- nan, caz nedeterminat

Operatori aritmetici - incrementare și asignare

- Limbajul C definește operatori unari pentru creșterea și descreșterea cu 1 a unei variabile
  - x++ post-incrementare, crește valoarea lui x cu 1 dar expresia este evaluată la valoarea originală lui x
  - ++x pre-incrementare, crește valoarea lui x cu 1 și expresia este evaluată la valoarea nouă lui x
  - x--, --x post- și pre-decrementare, analog
- O expresie cu operatorul de asignare (atribuire) se evaluează la valoarea asignată (asociativ de la dreapta la stânga)
  - putem scrie x = y = z = 1 echivalent cu x = (y = (z = 1))
- Asignare compusă
  - este de forma *operator aritmetic* =
  - de exemplu, x += 1 este echivalent cu x = x + 1

Operatori aritmetici - prioritate și asociativitate

- Într-o expresie compusă operațiile se evaluează după prioritatea lor
- Dacă operațiile au aceeași prioritate atunci se evaluează conform asociativității
- Pentru a schimba ordinea operațiilor se folosesc paranteze ()

Operatori aritmetici - prioritate și asociativitate - tabel

Prioritate	Nume	Operator	Asociativitate
1	post-increment post-decrement	.++ .--	
2	pre-increment pre-decrement plus unar minus unar	++. --. +. -.	
3	multiplicative	* / %	
4	aditive	+ -	
5	asignare	= *= /= %= += -=	

Operatori relaționali și logici

- În C nu există tip pentru valoare booleană
- Orice valoare nenulă este echivalentă cu adevărat
- 0 de orice tip este echivalent cu fals
- Operatorii relaționali și cei logici produc rezultat 0 sau 1
- Operatorii || și && implementează **scurt-circuitare**
  - e1 || e2 || 1 || e3 … - expresiile e3 și cele care urmează nu sunt evaluate fiindcă rezultatul este sigur 1
  - e1 && e2 && 0 && e3 … - expresiile e3 și cele care urmează nu sunt evaluate fiindcă rezultatul este sigur 0

Expresii - greșeli des întâlnite

```
float x = 1/2;
```

- x va fi 0, împărțire de numere întregi
- cea mai frecventă greșeală la expresii

```
int a = 2;
float y = 1 / 2.0 * a;
```

- y va fi 1, operatorul / are aceeași prioritate ca *
- se efectuează împărțirea apoi înmulțirea cu a

```
int a = 5, b;
int c = (b = a + 2) - (a = 1);
```

- să evităm expresiile care au efecte secundare, adică modifică valorile altor variabile
- expresia are valoare nedeterminată fiindcă depinde de ordinea în care se evaluează subexpresiile

Instrucţiunea alternativă if

- Ramifică fluxul de control pe una din maxim două alternative în funcţie de valoarea de adevăr a expresiei evaluate

```
if ( expresie )
   instructiune_1;
else
   instructiune_2;
```

- Expresia trebuie obligatoriu inclusă între paranteze rotunde
- Dacă `expresie` este adevărată (nenulă) atunci se execută `instructiune_1` altfel se execută `instructiune_2`
- Ramura else poate să lipsească

Instrucţiunea alternativă if - exemplu simplu

```c
#include <stdio.h>

int main() {
    int x;
    scanf("%d", &x);
    if (x%2)
        printf("numar impar");
    else
        printf("numar par");
    return 0;
}
```

- determină dacă numărul x citit este par sau impar
- x % 2 este 1, deci adevărat, doar dacă x este impar

Instrucţiunea alternativă if - exemplu complex

```c
#include <stdio.h>

int main() {
    float x;
    int r = scanf("%f", &x);
    if (r == 0)
        printf("format incorect");
    else if (x < 0 || x > 10)
        printf("in afara intervalului");
    else{
        x = x / 2;
        printf("%f\n", x);
    }
    return 0;
}
```

- citește x și îl înjumătățește doar dacă aparține intervalului [0, 10]
- salvăm valoarea returnată de scanf
- dacă r este 0, nu am citit
- expresie logică
- dacă avem mai multe instrucțiuni trebuie să formăm un bloc cu acolade

Instrucţiunea alternativă if - erori frecvente

```
int a = 2;
if (a = 10)
    printf("a este 10");
```

- operatorul de atribuire = este confundat cu operatorul de comparație ==
- expresia a = 10 schimbă a în 10 și este evaluată ca 10 (adevărat)
- mesajul va fi afișat întotdeauna

```
int a = 2;
if (a == 10);
    printf("a este 10");
```

- după expresia din if nu e corect să punem ;
- printf-ul nu este influențat de if și mesajul va fi afișat întotdeauna

```
int a = 2;
if (a == 10)
    printf("a este 10");
```

- varianta corectă

Instrucţiunea alternativă switch

- Ramifică fluxul de control pe una sau mai multe alternative în funcţie de valoarea numerică a unei expresii întregi

```
switch(expr){
   case c1: instr1;
   case c2: instr2;
   ...
   case cn: instrn;
   default: instr;
}
```

- Se evaluează expresia și se execută instrucțiunile începând de la prima etichetă cu care se potrivește rezultatul

Instrucţiunea alternativă switch

- expresia poate fi doar de tip întreg (caracter inclus)
- ramura `default` este pentru valori care nu se potrivesc cu nici un caz de mai sus - este opțională
  - recomandat sa fie ultima etichetă
- dacă dorim să se execute doar instrucțiunile de pe o singură ramură se adaugă instrucțiunea `break`

Instrucţiunea alternativă switch - exemplu

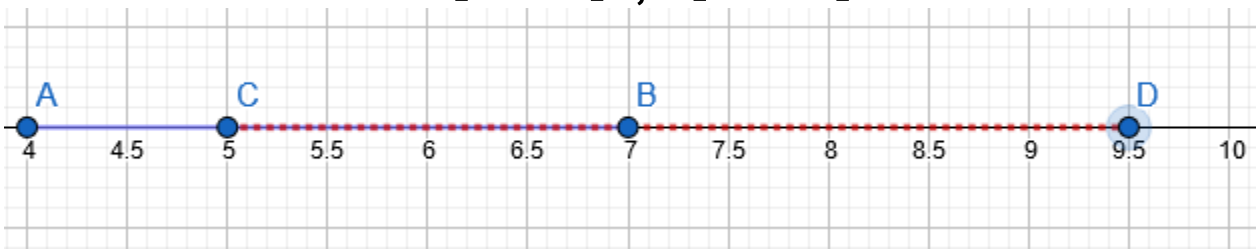
```c
#include <stdio.h>

int main() {
  char c;
  scanf("%c", &c);
  switch(c){
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u': puts("vocala"); break;
    default: puts("consoana");
  }
  return 0;
}
```

- citește caracterul c (o literă mică) și determină dacă este consoană sau vocală
- switch cu expresie de tip caracter
- etichete pentru cazurile de vocală
- se execută instrucțiunile după ramura cu care se potrivește, până la break
- pentru toate celelalte cazuri se intră pe ramura default

Discuție problemă - Intersecția a două intervale

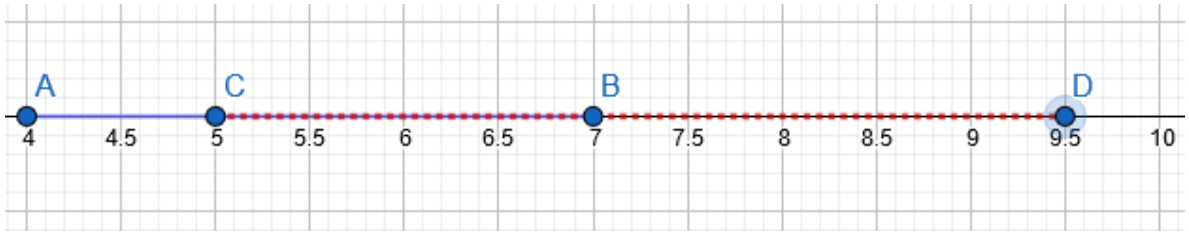
- Să se determine lungimea intersecției a două intervale de pe axa reală
- Notăm intervalele cu [A, B] și [C, D]



- Trebuie să determinăm lungimea intervalului care este în comun
  - răspunsul există întotdeauna
  - poate fi 0 dacă intervalele nu se intersectează
  - sunt multe cazuri diferite

Discuție problemă - Intersecția a două intervale

- Capătul din stânga al intersecției trebuie să fie punctul A sau punctul C
  - se ia cel care are coordonată maximă
- Capătul din dreapta al intersecției trebuie să fie punctul B sau punctul D
  - se ia cel care are coordonată minimă
- Dacă punctul ales pentru capătul din stânga este la stânga celuilalt punct atunci intersecția este diferența coordonatelor
  - altfel este 0
- Exprimat succint:



lungime = max(0, min(B, D) - max(A, C))

Discuție problemă - Intersecția a două intervale

```c
#include <stdio.h>

int main() {
    float A, B, C, D;
    scanf("%f%f", &A, &B);
    scanf("%f%f", &C, &D);
    float L = A;
    if (C > L)
        L = C;
    float R = B;
    if (D < R)
        R = D;
    if (R > L)
        printf("%f\n", R-L);
    else
        printf("0");
    return 0;
}
```

- calculăm în L maximul dintre A și C
- calculăm în R minimul dintre B și D
- se intersectează
- atunci lungimea este diferența
- altfel lungimea este 0