3. Tipuri şi variabile.

3.1. Tipuri.

Un *tip de date* este precizat prin:
- *o mulțime finită de* ***valori*** corespunzătoare tipului (constantele tipului)
- *o mulțime de* ***operatori*** prin care se prelucrează valorile tipului
- *o multime de* ***restricții*** de utilizare a operatorilor.

De exemplu tipul întreg (`int`) este definit prin:
- *mulțimea valorilor* reprezentând numere întregi (între **`-32768`** şi **`32767`**)
- *mulțimea operatorilor* **`: +, -, *, /, %`**
- *mulțimea restricțiilor*: pentru operatorul / împărțitorul nu poate fi 0, etc.

Tipurile pot fi *tipuri fundamentale* şi *tipuri derivate*.
Tipurile fundamentale (*predefinite* sau *de bază* sunt:
- Tipul boolean (**`bool`**)
- Tipul caracter (**`char`**)
- Tipuri întregi (**`int, short, long`**)
- Tipuri reale (**`float, double, long double`**)
- Tipul vid (**`void`**)
- Tipurile enumerate (**`enum`**)

Tipurile boolean, caracter, întreg, real şi tipurile enumerate sunt *tipuri aritmetice*, deoarece valorile lor pot fi interpretate ca numere.
Tipurile boolean, caracter, întreg şi enumerările sunt *tipuri întregi*.
Tipurile derivate sunt construite pornind de la tipurile fundamentale. Tipurile derivate sunt:
- tablourile
- funcțiile
- pointerii
- referințele
- structurile (sau înregistrările)
- uniunile (înregistrările cu variante)

În cele ce urmează, vom înțelege prin *obiect*, o zonă de memorie.
Declararea unui obiect specifică numai *proprietățile obiectului, fără a aloca memorie* pentru acesta.
Definirea unui obiect specifică *proprietățile obiectului şi alocă memorie* pentru obiect.
Declararea obiectelor ne permite referirea la obiecte care vor fi definite ulterior în acelaşi fişier sau în alte fişiere care conțin părți ale programului.

3.2. Tipuri fundamentale.

Calculatoarele pot lucra în mod direct cu caractere, întregi şi reali. Acestea sunt *tipuri fundamentale* (*predefinite*).

3.2.1. Caracterele (tipul `char`)

Valorile asociate tipului caracter (**`char`**) sunt elemente din mulțimea caracterelor – setul de caractere ASCII. Drept consecință, caracterele pot fi tratate ca întregi, şi invers.
Afişarea sau citirea unei variabile de tip **`char`** la terminal se face cu descriptorul de format **`%c`**.
Fiecărei constante caracter i se asociază o valoare întreagă, valoarea caracterului în setul de caractere ASCII. Pentru reprezentarea de caractere în alt set de caractere (de exemplu Unicode) se foloseşte tipul **`wchar_t.`**
Un literal caracter se reprezintă prin caracterul respectiv inclus între apostrofi (dacă este tipăribil).
Caracterele netipăribile se reprezintă prin mai multe caractere speciale numite *secvențe escape*. Acestea sunt:
`\n` sfârşit de linie (LF)

\t tabulare orizontală (HT)
\v tabulare verticală (VT)
\b revenire la caracterul precedent (BS)
\r revenire la început de linie (CR)
\f avans la pagină nouă (FF)
\a alarmă (BEL)
**** caracterul ****
\? caracterul **?**
\' caracterul **'**
\" caracterul **"**
\ooo caracterul cu codul octal **ooo**
\xhh caracterul cu codul hexazecimal **hh**

3.2.2. Întregii (tipul `int`).

Întregii pot avea 3 dimensiuni: **`int, short int`** (sau **`short`**) şi **`long int`** (sau **`long`**).
Constantele intregi pot fi date in bazele:

```
10:    257,  -65,  +4928
 8:    0125,  0177
16:    0x1ac, 0XBF3
```

Afişarea unei variabile de tip **`int`** în baza 10 la terminal se face cu descriptorul de format **`%d`** sau **`%i`**, în baza 8 cu **`%o`**, iar în baza 16 **`cu %x`**.

Calificatorii `long, short` şi `unsigned`

Calificatorul **`long`** situat înaintea tipului **`int`** extinde domeniul tipului întreg de la **$(-2^{15},2^{15}-1)$** la **$(-2^{31}, 2^{31}-1)$.**

Constantele intregi lungi se scriu cu sufixul **`L: 125436L`**.

Afişarea sau citirea unei variabile de tip **`long int`** la terminal se face cu descriptorul de format **`%ld`** sau **`%li`** în baza 10, **`%lo`** în baza 8 şi **`%lx`** în baza 16.

Calificatorul **`short`** situat înaintea tipului **`int`** restrânge domeniul întregilor.

Afişarea sau citirea unei variabile de tip **`short int`** la terminal se face cu descriptorul de format **`%hd`** sau **`%hi`** în baza 10, **`%ho`** în baza 8 şi **`%hx`** în baza 16.

Calificatorul **`unsigned`** înainte de **`int`** deplasează domeniul întregilor **$(-2^{15},2^{15}-1)$** la **$(0,2^{16}-1)$.**

Afişarea sau citirea unei variabile de tip **`unsigned int`** la terminal se face cu descriptorul de format **`%ud`** sau **`%ui`** în baza 10, **`%uo`** în baza 8 şi **`%ux`** în baza 16.

Constantele fără semn se specifică cu sufixul **`U`** sau **`u`**.

Există 6 tipuri întregi, formate folosind calificatorii:

```
{ [ signed | unsigned ] } { [ short | long ] } int
```

Avem următoarele echivalențe:

```
int = signed int
short = short int = signed short int
long = long int
unsigned = unsigned int
unsigned short = unsigned short int
unsigned long = unsigned long int
```

Literali întregi fi scrişi în:
- *zecimal* - un şir de cifre zecimale, dintre care prima nu este 0.
- *octal* - un şir de cifre octale care începe cu cifra 0
- *hexazecimal* - un şir de cifre hexa care începe prin 0x.

Un numar negativ este precedat de semnul -. Există şi semnul + unar.
Exemple: **`125 01473 0x2AFC +645 -8359`**

Literalii întregi se pot termina prin **u** sau **U** (fără semn) sau **l** sau **L** (de tip lung).
Dacă constanta nu are sufix, atunci ea va aparține primului tip din succesiunea: **int, long int**, **unsigned long int** care permite reprezentarea valorii.

3.2.3. Realii (tipurile float şi double).

Partea întreagă sau cea fracționară din constanta reală poate lipsi:

întreg.fractie sau ***întreg.*** sau ***.fractie***

Exemple: **2.25, 1., -.5, +234.5**

Constantele reale pot fi exprimate cu *mantisă* şi *exponent* (notația ştiințifică):

$$mantisaEexponent = mantisa\ 10^{exponent}$$

Exemple: **1.5e-3, 0.5E6**.

Tipul **float** asigură o precizie de 7 cifre zecimale semnificative şi exponentul maxim 38. Reprezentarea se face pe 4 octeți.

Afişarea unei variabile de tip **float** la terminal se face cu descriptorii de format **%f** sau **%e**.

Tipul **double** este foarte asemănător tipului **float**, cu deosebirea că se asigură o precizie de 16 cifre zecimale semnificative şi exponentul maxim 306. Reprezentarea se face pe 8 octeți.

Afişarea sau citirea unei variabile de tip **double** la terminal se face cu descriptorul de format **%lf**, iar a unei variabile de tip **long double** cu **%Lf**. Reprezentarea internă pentru **long double** se face pe 10 octeți.

Numerele reale conțin punct zecimal şi/sau exponent, având forma:

[<parte întreaga>][.<parte fracționara>][E<exponent>]

Partea întreagă si partea fracționară pot lipsi, dar nu simultan. Punctul zecimal şi exponentul sunt opționale, dar nu simultan.

Constanta poate avea un sufix:

- **f** sau **F** precizează o constantă de tip **float**
- **l** sau **L** precizează o constantă de tip **long double**.

Exemple: **.25 -7.628 15E-3**

3.2.4. Definiri de tip cu typedef.

Un tip de date poate avea şi un alt nume, prin folosirea declarației **typedef**. De exemplu:

```
typedef int  intreg
```

Acelaşi efect se obține cu directiva **define**:

```
#define  intreg  int
```

3.2.5. Tipuri enumerate.

Folosirea tipului enumerare poate creşte claritatea programului, întrucât numele sunt mai semnificative decât valorile care se ascund în spatele lor.

Astfel este mai naturală folosirea valorilor **ROSU, ALB, NEGRU, VERDE** pentru a desemna nişte culori, decât a valorilor **0, 1, 2, 3**.

Constantele simbolice pot fi introduse cu macrodefinitii **#define**.

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

Un *tip enumerat* foloseşte în locul valorilor tipului **0,1,2**,… nume simbolice:

```
enum CULORI {ROSU, VERDE, GALBEN, ALBASTRU, NEGRU};
enum boolean {FALSE, TRUE};
```

Este posibil sa forțăm pentru numele simbolice alte valori întregi decât **0,1,2,**…

```
enum ZILE {LUNI=1, MARTI, MIERC, JOI, VIN, SAMB, DUM};
```

```
enum escapes {BELL='\a',BACKSPACE='\b',TAB='\t',NEWLINE='\n',
              VTAB='\v',RETURN='r'};
```

Folosind **typedef** putem defini tipuri enumerative:

```
typedef enum {ROSU, GALBEN, ALBASTRU} culori;
typedef enum { lu, ma, mi, jo, vi, si, du } zile;
```

3.2.6. Tipul vid (void).

Tipul **void** precizează o mulțime vidă de valori. Acesta se foloseşte pentru a specifica tipul unei funcții care nu întoarce nici un rezultat, sau a unui pointer generic.

3.3. Tipuri derivate.

Pe baza tipurilor fundamentale se pot construi tipuri derivate ca:

- tablouri de obiecte de un anumit tip
- functii care întorc obiecte de un anumit tip
- pointeri la obiecte de un anumit tip
- structuri care conțin obiecte de tipuri diferite
- uniuni care conțin obiecte de tipuri diferite

3.4. Declararea variabilelor.

O variabilă constă din două componente: obiectul şi numele obiectului. Numele pot fi identificatori sau expresii.

Definirea sau declararea unei variabile are forma:

clasă-memorie tip declaratori;

Clasa de memorie sau tipul pot fi omise. Declaratorii sunt identificatori.

Fiecare declarator poate fi urmat de un *inițializator*, care specifică valoarea inițială asociată identificatorului declarator.

Asupra claselor de memorie vom reveni mai târziu, aşa că în exemplele curente le vom omite:

```
int i, j=0.
char c;
float x=1.5, y;
enum CULORI s1, s2=ROSU;
enum BOOLEAN p=TRUE;
culori c1, c2;
zile z, d;
```

3.5. Echivalența tipurilor.

Două tipuri se consideră echivalente în următoarele situații:

- echivalență structurală a tipurilor
- echivalența numelui tipului

Două obiecte sunt de tipuri structural echivalente, dacă au acelaşi tip de componente.

Două obiecte sunt de tipuri echivalente după nume, dacă au fost definite folosind acelaşi nume de tip.

Exemple:

```
typedef int integer;
int x, y;   /* x şi y sunt echivalente dupa numele tipului*/
integer z;/* x şi z sunt de tipuri structural echivalente */
```

4. Operatori şi expresii.

Un *operator* este un simbol care arată ce operaţii se execută asupra unor operanzi (termeni).

Un *operand* este o constantă, o variabilă, un nume de funcţie sau o subexpresie a cărei valoare este prelucrată direct de operator sau suportă în prealabil o conversie de tip.

Operatorii, după numărul de operanzi asupra cărora se aplică pot fi: *unari*, *binari* şi *ternari*.

În C există 45 de operatori diferiţi dispuşi pe 15 *niveluri de prioritate*.

În funcţie de tipul operanzilor asupra cărora se aplică, operatorii pot fi: aritmetici, relaţionali, binari, logici, etc.

Operatorii sunt împărţiţi în *clase de precedenţă* (sau de *prioritate*). În fiecare clasă de precedenţă este stabilită o *regulă de asociativitate*, care indică ordinea de aplicare a operatorilor din clasa respectivă: de la stânga la dreapta sau de la dreapta la stânga.

O expresie este o combinaţie de operanzi, separaţi între ei prin operatori; prin *evaluarea* unei expresii se obţine o *valoare rezultat*.Tipul valorii rezultat depinde de tipul operanzilor şi a operatorilor folosiţi.

Evaluarea unei expresii poate avea *efecte laterale*, manifestate prin modificarea valorii unor variabile.

4.1. Conversii de tip.

Valorile pot fi convertite de la un tip la altul. Conversia poate fi implicită sau realizată în mod explicit de către programator.

4.1.1. Conversii implicite de tip.

Conversiile implicite au loc atunci când este necesar ca operatorii şi argumentele funcţiilor să corespundă cu valorile aşteptate pentru acestea.

Acestea pot fi sintetizate prin tabelul:

Tip	Tip la care se converteşte implicit
`char`	**`int, short int, long int`**
`int`	**`char`** (cu trunchiere) **`short int`** (cu trunchiere) **`long int`** (cu extensia semnului)
`short int`	ca şi **`int`**
`long int`	ca şi **`int`**
`float`	**`double`** **`int, short int, long int`**
`double`	**`float`** **`int, short int, long int`**

4.1.2. Conversii aritmetice.

Când un operator binar se aplică între doi operanzi de tip diferit, are loc o *conversie implicită* a tipului unuia dintre ei, şi anume, operandul de tip "mai restrâns" este convertit la tipul "mai larg" al celuilalt operand. Astfel în expresia **`f + i`**, operandul **`int`** este convertit în **`float`**.

Operatorii aritmetici convertesc automat operanzii la un anumit tip, dacă operanzii sunt de tip diferit. Se aplică următoarele reguli:

- operanzii **`char`** şi **`short int`** se convertesc în **`int`**; operanzii **`float`** se convertesc în **`double`**.
- dacă unul din operanzi este **`double`** restul operanzilor se convertesc în **`double`** iar rezultatul este tot **`double`**.

- dacă unul din operanzi este **long** restul operanzilor se convertesc în **long** , iar rezultatul este tot **long**.
- dacă unul din operanzi este **unsigned** restul operanzilor se convertesc în **unsigned** , iar rezultatul este tot **unsigned**.
- dacă nu se aplică ultimele 3 reguli, atunci operanzii vor fi de tip **int** şi rezultatul de asemeni de tip **int.**

```
double      ←float
  ↑
long
  ↑
unsigned
  ↑
 int        ←char, short
```

Astfel `n = c - '0'` în care **c** reprezintă un caracter cifră calculează valoarea întreagă a acestui caracter.

Conversii implicite se produc şi în cazul operației de atribuire, în sensul că valoarea din partea dreaptă este convertită la tipul variabilei acceptoare din stânga.
Astfel pentru declarațiile:

```
int i;
float f;
double d;
char c;
```

sunt permise atribuirile:

```
i=f;  /* cu trunchierea partii fractionare */
f=i;
d=f;
f=d;
c=i;
i=c;
```

4.1.3. Conversiile de tip explicite (cast).

Conversiile explicite de tip (numite şi cast) pot fi forțate în orice expresie folosind un operator unar (*cast*) într-o construcție de forma:

```
(tip) expresie
```

în care expresia este convertită la tipul numit.

Operatorul cast are aceeaşi precedență cu a unui operator unar.

Astfel funcția **sqrt()** din biblioteca **<math.h>** cere un argument **double**, deci va fi apelată cu un cast: **sqrt((double) n).**

Apelurile cu argumente de alt tip vor fi convertite în mod automat la tipul **double:** **x=sqrt(2)** va converti constanta **2** în **2.0**.

4.2. Operatorii aritmetici.

Operatorii aritmetici binari sunt: **+, -, *, /** şi **%** (modul = restul impărțirii întregi).

Prioritatea operatorilor aritmetici este:

+, -	unari
***, /, %**	binari
+, -	binari

Regula de asociativitate este de la stânga la dreapta (la priorități egale operatorii sunt evaluați de la stânga la dreapta).

Operatori multiplicativi

operator	descriere	tip operanzi	tip rezultat	precedență
`*`	înmulțire	aritmetic	`int,unsigned,long, double`	3
`/`	împărțire	aritmetic	`int,unsigned,long, double`	3
`%`	restul împărțirii întregi	întreg	`int,unsigned,long`	3

Operatori aditivi

operator	descriere	tip operanzi	tip rezultat	precedență
`+`	adunare	aritmetici pointer şi întreg	`int,unsigned,long,double` `pointer`	4

`-`	scădere	aritmetici
		pointer şi întreg
		doi pointeri
			`int,unsigned,long,double`
			`pointer`
			`int`
				4

4.3. Operatorii de atribuire.

Operația de atribuire modifică valoarea asociată unei variabile (partea stângă) la valoarea unei expresii (partea dreaptă). Valoarea transmisă din partea dreaptă este convertită implicit la tipul părții stângi.

Atribuiri de forma: **`a = a op b`** se scriu mai compact **`a op= b`** în care **`op=`** poartă numele de *operator de atribuire*, `op` putând fi un operator aritmetic **`(+, -,*,/,%)`** sau binar **`(>>, <<, &, ^, |)`.**

O *atribuire multiplă* are forma:

`v1=v2=...=vn=expresie`

şi este asociativă la dreapta.

O operație de atribuire terminată prin punct-virgulă (terminatorul de instrucțiune) se transformă într-o *instrucțiune de atribuire*.

4.4. Operatorii relaționali.

Operatorii relaționali sunt: **`>, >=, <, <=`** , care au toți aceeaşi prioritate (precedență).

Cu prioritate mai mică sunt: **`==, !=.`**

Operatorii relaționali au prioritate mai mică decât operatorii aritmetici. Putem deci scrie **`a < b -1`** în loc de **`a < (b -1)`**

Exemple: **`car >= 'a' && car <= 'z'`**

Operatori relaționali

operator	descriere	tip operanzi	tip rezultat	precedență
`<`	mai mic	aritmetic sau pointer	`int`	6
`>`	mai mare	aritmetic sau pointer	`int`	6
`<=`	mai mic sau egal	aritmetic sau pointer	`int`	6
`>=`	mai mare sau egal	aritmetic sau pointer	`int`	6
`==`	egal	aritmetic sau pointer	`int`	7
`!=`	neegal	aritmetic sau pointer	`int`	7

4.5. Operatorii booleeni.

Există următorii operatori logici:

`!` - NEGATIE (operator unar)

`&&` - ŞI logic (operatori binari)

`||` - SAU logic

Exemple:

```
i<n-1 && (c=getchar()) != '\n' && c != EOF
```

nu necesită paranteze suplimentare deoarece operatorii logici sunt mai puțin prioritari decât cei relaționali.

```
bisect= an % 4 == 0 && an % 100 != 0 || an % 400 == 0;
estecifra= c >= '0' && c <= '9'
```

Condiția `x == 0` este echivalentă cu `!x`

`x != 0 && y != 0 && z != 0` este echivalentă cu `x && y && z`

Operatori booleeni

operator	descriere	Tip operanzi	Tip rezultat	precedență	asociativitate
`!`	negație	aritmetic sau pointer	`int`	2	DS
`&&`	ȘI logic	aritmetic sau pointer	`int`	11	SD
`||`	SAU logic	aritmetic sau pointer	`int`	12	SD

4.6. Operatorii binari (la nivel de biți).

În C există 6 operații de manipulare a biților aplicate asupra unor operanzi întregi (`char, short, int, long`) cu sau fără semn:

`&` - ȘI
`|` - SAU inclusiv
`^` - SAU exclusiv
`<<` - deplasare stânga
`>>` - deplasare dreapta
`~` - complement fată de 1 (inversare)

Operatorul ȘI se foloseste pentru *operația de mascare* a unor biți.

```
n=n & 0177;  /* pune pe 0 bitii din pozitia 8 in sus */
```

Operatorul SAU pune pe 1 biții specificați printr-o mască:

```
x=x | MASCA; /* pune pe 1 bitii care sunt 1 in MASCA */
```

Deplasarea dreapta a unui întreg cu semn este aritmetică, iar a unui întreg fără semn este logică.

De exemplu:

```
x |= 1 << 7; /* pune pe 1 bitul 0 din octetul x */
x &= ~(1 << 7); /* pune pe 0 bitul 0 din octetul x */
```

Operatori pe biți

operator	descriere	Tip operand	tip rezultat	precedență
`<<`	deplasare stânga	Întreg	ca operandul stâng	5
`>>`	deplasare dreapta	Întreg	ca operandul stâng	5
`&`	ȘI pe biți	Întreg	`int,long,unsigned`	8
`^`	SAU exclusiv pe biți	Întreg	`int,long,unsigned`	9
`|`	SAU inclusiv pe biți	Întreg	`int,long,unsigned`	10

4.7. Operatorul condițional.

Decizia

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

poate fi reprezentată prin expresia condițională:

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

În general **ex1 ? ex2 : ex3** determină evaluarea **ex1**; dacă aceasta nu este 0 atunci valoarea expresiei condiționale devine **ex2,** altfel **ex3**.

4.8. Operatorul secvență.

Este reprezentat prin **,** și se foloseste în situațiile în care sintaxa impune prezența unei singure expresii, dar prelucrarea presupune prezența și evaluarea mai multor expresii. Exemplu:

```
a < b ? (t=a, a=b, b=t) : a
```

4.9. Operatori unari

a) Operatorul `sizeof`.

Aplicat asupra unei variabile furnizează numărul de octeți necesari stocării variabilei respective. Poate fi aplicat și asupra unui tip sau asupra tipului unei expresii:

```
sizeof variabila
sizeof tip
sizeof expresie
```

`sizeof` este un operator cu efect la compilare.

operator	descriere	tip operand	tip rezultat	precedență
`sizeof`	necesar de memorie	variabilă sau tip	**`unsigned`**	2

b) Operatorii de incrementare /decrementare.

operator	descriere	tip operand	tip rezultat	precedență
++	preincrementare	aritmetic sau pointer	**`int,long,double, unsigned,pointer`**	2
++	postincrementare	aritmetic sau pointer	la fel	2
--	predecrementare	aritmetic sau pointer	la fel	2
--	postdecrementare	aritmetic sau pointer	la fel	2

c) Operatori de adresare indirectă / determinare adresă

`& entitate` - obține adresa unei entități,

`* pointer` - pentru adresare indirectă - adică memorează adresa unei entități printr-o valoare a unui pointer

operator	descriere	tip operand	tip rezultat	precedență
`*`	indirectare	pointer la **`T`**	**`T`**	2
`&`	adresare	**`T`**	pointer la **`T`**	2
`~`	negație	aritmetic	**`int,long,double`**	2
`!`	negație logică	aritmetic sau pointer	`int`	2

d) Operatori de acces :

- la elementele unui tablou
- la câmpurile unei structuri sau unei uniuni
- indirect prin intermediul unui pointer la câmpurile unei structuri sau unei uniuni

Operatorii de acces sunt:

- **`[]`** *indexare* folosit în expresii de forma **`tablou[indice]`**
- **`.`** *selecție directă* - pentru adresarea unui câmp dintr-o structură sau uniune sub forma: **`struct.selector`**
- → *selecție indirectă* - pentru accesul la un câmp dintr-o structură sau uniune, a cărei adresă este memorată într-un pointer

  `pointer -> selector` este echivalent cu **`(* pointer) . selector`**

Toți aceşti operatori de acces, împreună cu operatorul de apel de funcție `()` au cea mai ridicată prioritate, şi anume 1.

Operatori de acces

operator	descriere	exemple	precedență
`()`	apel de funcție	`sqrt(x),printf("salut\n")`	1
`[]`	indexare tablou	`x[i], a[i][j]`	1
`.`	selector structură	`student.nastere.an`	1
`->`	selector indirect structură	`pstud->nume`	1

Ordinea evaluarii operanzilor.

precedență	operatori	simbol	asociativitate
1	apel funcție / selecție	`() [] . ->`	SD
2	unari	`* & - ! ~ ++ -- sizeof`	DS
3	multiplicativi	`* / %`	SD
4	aditivi	`+ -`	SD
5	deplasări	`<< >>`	SD
6	relaționali	`< > <= >=`	SD
7	egalitate / neegalitate	`== !=`	SD
8	ŞI pe biți	`&`	SD
9	SAU exclusiv pe biți	`^`	SD
10	SAU inclusiv pe biți	`|`	SD
11	ŞI logic	`&&`	SD
12	SAU logic	`||`	SD
13	condițional	`?:`	DS
14	atribuire	`= op=`	DS
15	virgula	`,`	SD