Universitatea "Politehnica" Timisoara
Departamentul Calculatoare
Digital Signal Processing Labs Timisoara

Programarea Calculatoarelor

Valentin STÂNGACIU

Ver. 1.52/2025

Secțiunea I

Introducere

Introducere disciplină

Programarea Calculatoarelor (PC)

Introducere

Disciplina Programarea Calculatoarelor

- PC – disciplină fundamentală a întregului program de studii
- Precondiție pentru majoritatea disciplinelor ce vor urma:
  - Tehnici de programare (an I, sem 2)
  - Structuri de date si algoritmi (an II, sem 1)
  - Proiectarea si analiza algoritmilor (an II, sem 2)
  - Sisteme de operare (an II, sem 2)
  - Sisteme de operare 2 (an III, sem 1)
- Obiectivele disciplinei:
  - Dobândirea noțiunilor de bază din programare în limbajul C
  - Proiectarea şi implementarea unor programe C de complexitate mică şi medie
  - Obținerea unor deprinderi de testare şi depanare a programelor
- Organizarea disciplinei:
  - 42 ore de curs: 3 ore curs / săptămână
  - 28 ore de laborator: 2 ore laborator / săptămână
- Suport desfasurare si evaluare disciplină disciplină:
  - **Campus Virtual –** materiale curs, materiale laborator, discutii, anunturi, note, prezente, medii

Introducere

Disciplina Programarea Calculatoarelor

- Frequently Asked Questions (FAQs)
  - Am facut programare în liceu? Pot să fiu scutit integral sau parţial de această disciplină ?
  - NU. Ai acum șansa de a te perfecționa. În plus, disciplina este predată la nivel de inginerie și adaugă multe noțiuni ce nu au fost predate în liceu. De asemenea, avem probleme adaptate pentru nivele diferite de cunoștințe.
  - Nu am făcut deloc programare in liceu. Va fi o problema ?
  - Nu. Vei avea șansa să înveți foarte bine programare chiar dacă nu ai făcut deloc în liceu.
  - Pot să mă mut dintr-o semigrupă în alta ?
  - Doar in cazul acestei discipline nu ! Se poate doar la secretariat și mutarea este permanentă pe tot parcursului anului universitar și implicit la toate disciplinele.
  - Dacă am lipsit la un laborator îl pot recupera?
  - Da. Se poate recupera, în limita locurilor disponibile la laborator, în aceeași săptămână sau în săptămâna 14.
  - Căte laboratoare pot absenta?
  - Conform regulamentului UPT, se poate absenta maxim 25% din numărul total de laboratoare. În cazul disciplinei PC se poate absenta la maxim 3 laboratoare. Totuși, chiar și în acest caz, laboratoarele absentate trebuie obligatoriu să fie recuperate. Aceasta este o condiție necesară promovării activității pe parcus

Introducere

Disciplina Programarea Calculatoarelor

- Evaluarea disciplinei:
  - Evaluare finală prin examen practic pe calculator: examen teoretic grila, examen rezolvare probleme
  - Evaluare pe parcurs la laborator
- Nota finala disciplină:

$$M_{PC} = ROUND\,(0.5 \cdot E + 0.5 \cdot AP) = ROUND\left(\frac{E + AP}{2}\right), k_1 = 0.5,\ k_2 = 0.5$$

$$AP = L + P$$

E – notă examen, AP – notă activitate pe parcurs, L – nota laborator, P – punct prezență curs

$$P = \begin{cases} +1, PrezentaCurs \in [90, 100]\% \\ 0, PrezentaCurs \in [60, 90)\ \% \\ -1, PrezentaCurs \in [30, 60)\ \% \\ -2, PrezentaCurs \in [0, 30)\ \% \end{cases}$$

- Punct prezență curs – se acordă doar daca $L \geq 5$
- Nu se acordă mărire de notă la notele de la laborator și nici la media finală de la laborator. Nu se dau teste suplimentare la laborator pentru promovare, mărire sau corecție de notă
- Notele la laborator, activitate pe parcurs si examen nu se rotunjesc, se calculează cu 2 zecimale
- Media finală se rotunjește conform regulamentului UPT

Introducere

Programarea Calculatoarelor – Echipa didactică

- **Titular curs**: sl. dr. ing. Valentin STÂNGACIU
  - Contact: B417, valentin.stangaciu@cs.upt.ro
  - Domenii de interes: sisteme embedded, sisteme de operare, sisteme timp-real, protocoale de comunicații, rețele de senzori
- **Echipa de laborator**:
  - as. drd. ing. Petra CSEREOKA
    - Contact: B417, petra.csereoka@cs.upt.ro
    - Domenii de interes: Modeling, formal verification and testing, Embedded systems, Intelligent robotic environments, Operating systems, Artificial intelligence, Machine learning, Neural network
  - ing. Vlad Plăvăț
    - Contact: vlad.plavat@cs.upt.ro
    - Domenii de interes: embedded systems

Introducere

Programarea Calculatoarelor – Sfaturi preliminare

- **Ce avem nevoie pentru a învăța programare ?**
  - **Intelegere**
    - noțiunile trebuie înțelese
    - programare NU se poata învăța mecanic
  - **Respectarea regulilor** – un limbaj de programare se bazeaza pe un set de reguli bine definite
  - **Gândire** – rezolvarea unei probleme presupune o gândire matematică și o abordare inginerească. NU se învată șabloane !
  - nu căutați soluții rezolvate pe Internet – mai mult încurcă decât ajută – nu ajută la dezvoltarea unei gândiri analitice
  - **Multă muncă și exercițiu susținut și continuu**
    - **Programare** se poate învăța **doar** prin **studiu individual**
    - Recomandare: scrieți cod **minim 2 ore pe zi**, minim **10-12 ore / săptămână** – toate ca **studiu individual**

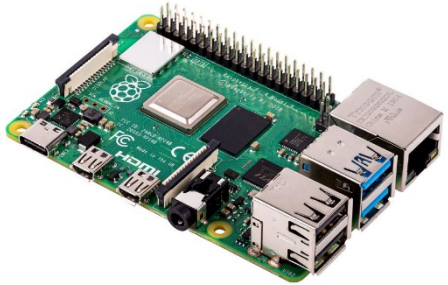
Introducere

Programarea Calculatoare

Introducere

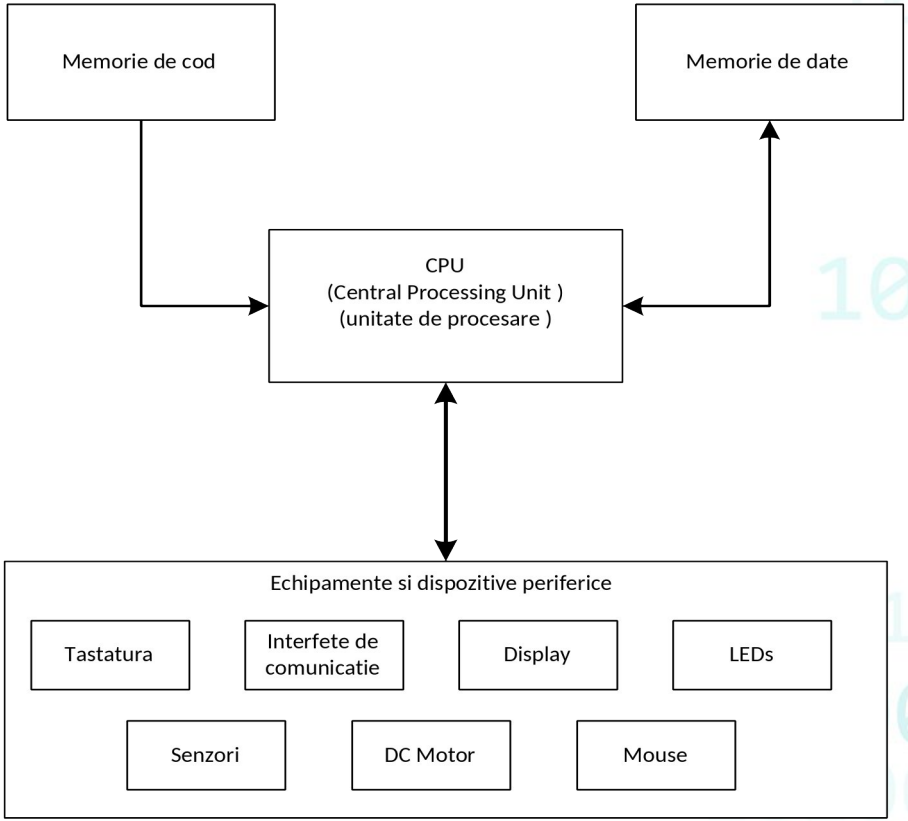
Programe și sisteme de calcul

- PROGRAM
  - Secvență de instucțiuni executată de un sistem de calcul pentru îndeplinirea unei funcționalități
- SISTEM DE CALCUL – dispozitiv fizic (hardware) ce execută un **program** cu scopul de a îndeplini o anumita funcție
  - Calculator personal (PC) : Desktop / Laptop
  - Telefon mobil, tableta, Smart TV
  - Sisteme incorporate: microcontrollere (ATMEGA16)
  - Sisteme cu microcontroller: Raspberry PI, BeagleBone Black, etc



Introducere

Structura generala a unui sistem de calcul



Introducere

Cod masina. Limbaje de asamblare vs limbaje de programare

- Structura generală a unui program: **date de intrare -> prelucrare date -> date de iesire**
- SISTEM DE CALCUL: execută secvențe de instrucțiuni în **cod mașină ->** program codificat în secvențe binare înterpretate ca și instrucțiuni de către sistemul de calcul
- Scrierea de cod mașină (programarea în cod mașină)
  - extrem de dificil
  - ineficient
  - aproape imposibil (astazi)
- Soluții: limbaj de asamblare, limbaje de programare (C, C++, Java, etc…)
- Limbaj de asamblare:
  - limbaj de programare de nivel redus ce permite unui programator să folosească o serie de simboluri pentru a implementa programe, ușor interpretabile de om. Simbolurile desemneaza practic instrucțiunile mașină ale sistemului de calcul
  - Greu de urmărit, menținut, modificat
  - Necesită cunoștinte extrem de avansate despre arhitectura hardware a sistemului de calcul

Introducere

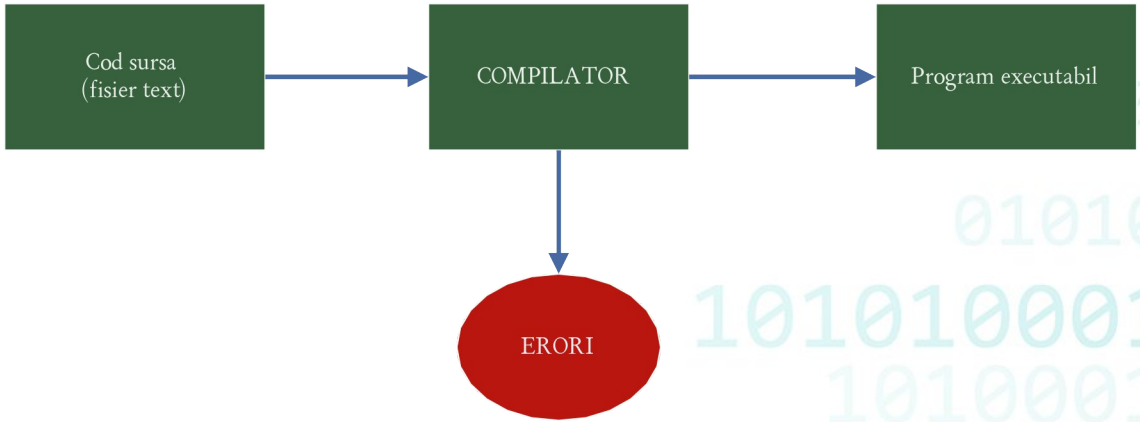
Cod masina. Limbaje de asamblare vs limbaje de programare

- Limbaj de asamblare (cont.):
  - codul efectiv este total depdendent de arhitectura mașinii țintă (target)
  - Necesită un **asamblor**: program specializat ce translatează codul în limbaj de asamblare scris de programator în cod mașină executabil ce poate fi rulat de către sistemul de calcul țintă (target)
  - Asamblorul produce un cod mașină “ad literam” după codul în limbaj de asamblare scris de programator

Introducere

Cod masina. Limbaje de asamblare vs limbaje de programare

- Limbaj de programare
  - reprezintă un limbaj formal prin care programatorul modelează o problema spre a fi rezolvată de un sistem de calcul
  - este format dintr-un set de reguli bine definite, bine determinate, o sintaxă, o gramatică și o semantică
    - Sintaxa - determină forma și structura *propozițiilor* din limbaj
    - Semnatica - determină semnificația *propozițiilor* din limbaj



Introducere

Cod masina. Limbaje de asamblare vs limbaje de programare

- Limbaj de programare (cont.)
  - Codul nu este deloc dependent de arhitectura mașinii țintă
  - Codul poate fi compilat pentru aproape orice tip de mașină țintă fără a fi rescris
  - Necesită un **compilator**: program specializat ce translatează codul sursă scris de programator într-un anumit limbaj de programare, in cod mașină executabil.
  - Compilatorul verifică toate elementele de limbaj (sintaxă, semnatică) înainte de a genera codul mașină
  - Compilatorul nu traduce codul “ad-literam” in cod masină
  - Compilatorul generează o secvență de una sau mai multe instrucțiuni mașină pentru o instrucțiune a limbajului de programare

Introducere

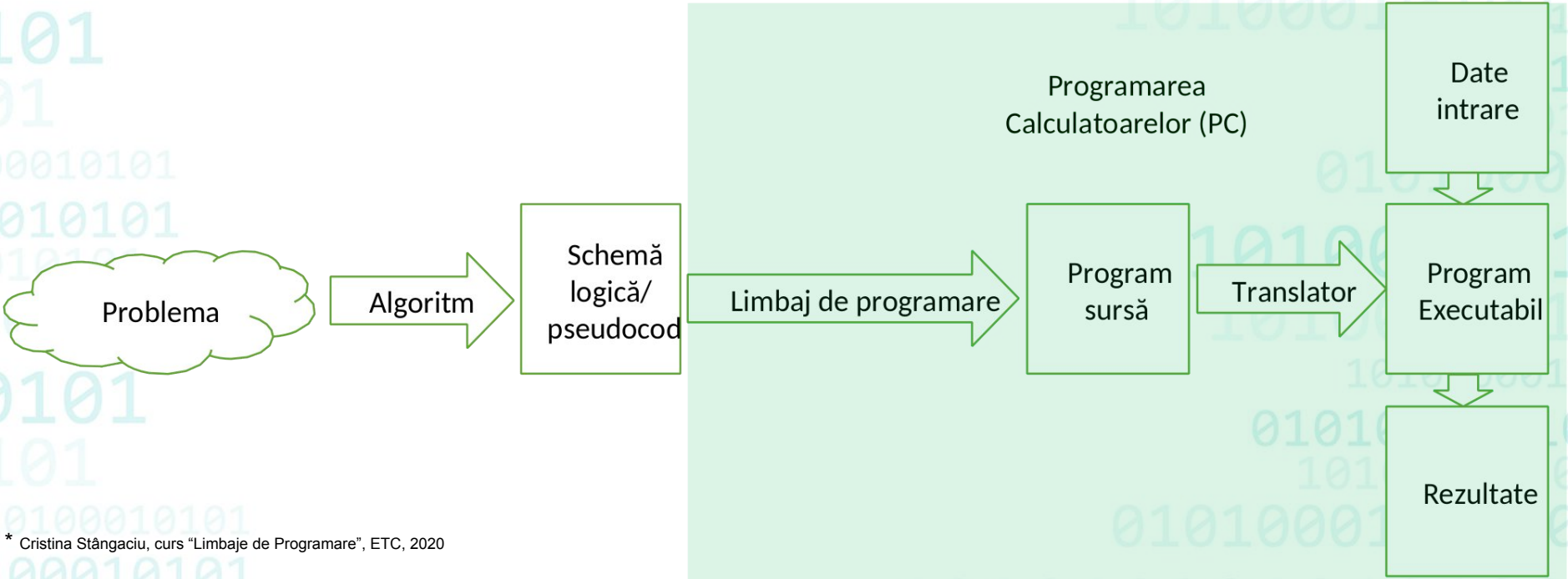
Etapele rezolvării unei probleme

1. Analizarea si abstractizarea problemei
2. Împărțirea problemei în sub-probleme, în task-uri, module, funcționalități
3. Stabilirea unor metode de rezolvare a sub-problemelor
4. Elaborarea algoritmilor de rezolvare a sub-problemelor
5. Implementarea agloritmului cu ajutorul schemelor logice sau pseudocod
6. **Implementarea algoritmului printr-un limbaj de programare**

* Adaptare după – Cristina Stângaciu, curs “Limbaje de Programare”, ETC, 2020

Introducere

Etapele rezolvării unei probleme

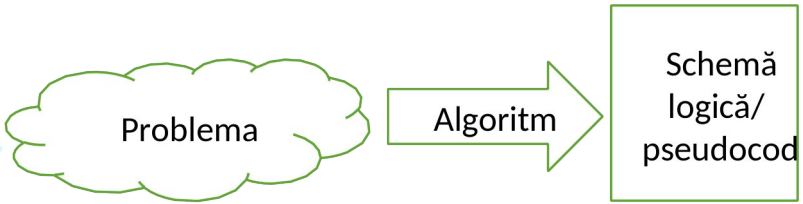


* Cristina Stângaciu, curs “Limbaje de Programare”, ETC, 2020

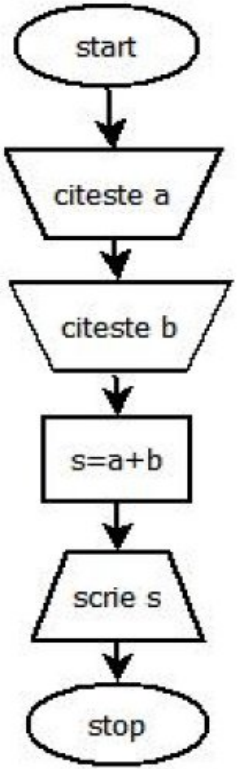
Introducere

Etapele rezolvării unei probleme

- Pas 5 – implementare algoritm folosind scheme logice



- Scheme logice:
  - descrierea unui algoritm, in mod grafic, folosind anumite notații grafice (blocuri) cu tranziții între acestea
  - Exemplu: implementarea unui algoritm de realizare a sumei a două numere

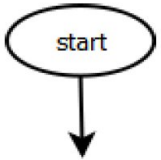


- Adaptare după: Cristina Stângaciu, curs “Limbaje de Programare”, ETC, 2020
- Adaptare după: Ciprian Chirilă, curs “Utilizarea și programarea calculatoarelor”, MPT

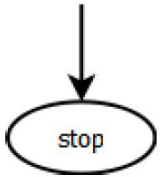
Introducere

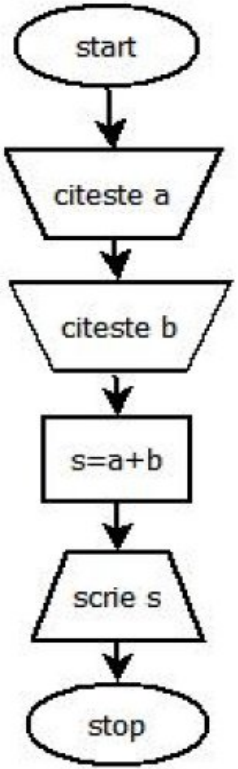
Etapele rezolvării unei probleme

- Blocuri din scheme logice
  - Bloc de start: semnifică începutul algoritmului



  - Bloc de stop: semnifică sfârșitul algoritmului



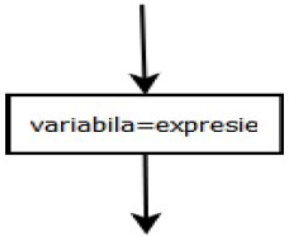


- Adaptare după: Cristina Stângaciu, curs “Limbaje de Programare”, ETC, 2020
- Adaptare după: Ciprian Chirilă, curs “Utilizarea și programarea calculatoarelor”, MPT

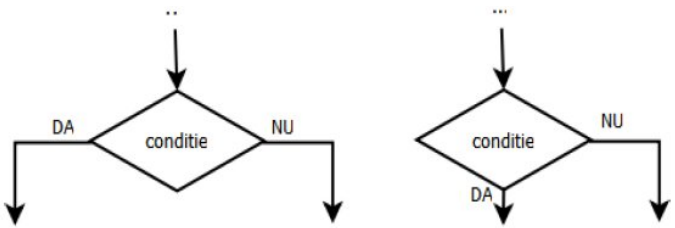
Introducere

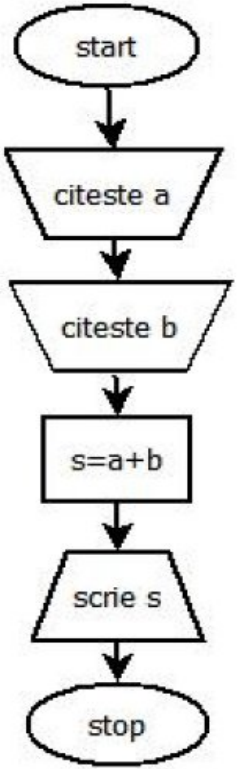
Etapele rezolvării unei probleme

- Blocuri din scheme logice
  - Bloc de atribuire: semnifică operațiuni de atribuire de valori



  - Bloc de decizie: semnifică operațiuni de luare de decizie



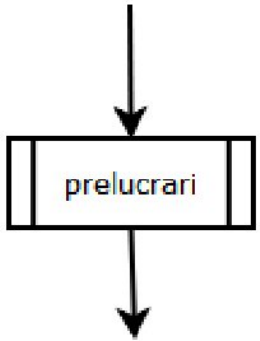


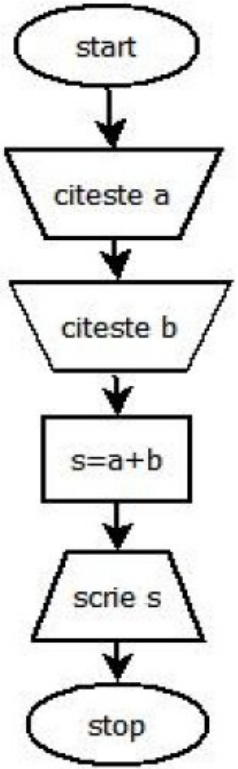
- Adaptare după: Cristina Stângaciu, curs “Limbaje de Programare”, ETC, 2020
- Adaptare după: Ciprian Chirilă, curs “Utilizarea și programarea calculatoarelor”, MPT

Introducere

Etapele rezolvării unei probleme

- Blocuri din scheme logice
  - Bloc de secvență: conține mai multe blocuri – o sub-schema logica



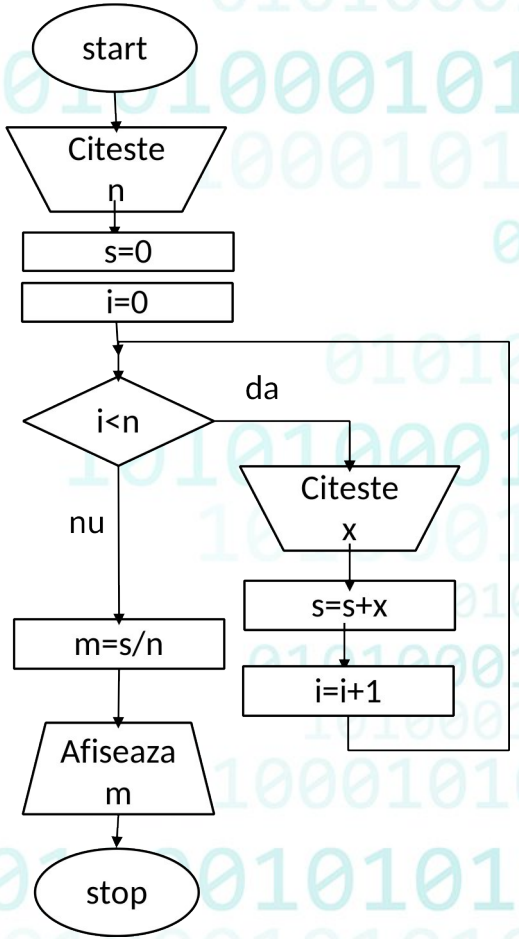


- Adaptare după: Cristina Stângaciu, curs “Limbaje de Programare”, ETC, 2020
- Adaptare după: Ciprian Chirilă, curs “Utilizarea și programarea calculatoarelor”, MPT

Introducere

Etapele rezolvării unei probleme

- Blocuri din scheme logice
  - exemplu – media a *n* numere



Introducere

Sistemul de operare Linux. Generalități de utilizare

Introducere

Elemente de utilizare Linux

- Mediul de lucru la curs si laborator PC: **GNU Debian** sau Ubuntu (Canonical)
- Nu se va lucra la PC în Windows (si nici in Linux) folosind IDE: Visual Studio, Codeblocks, …
- Se permite utilizarea de Windows 10 Windows Subsystem for Linux (WSL) cu Ubuntu sau Debian
- Mediu de lucru: orice distribuție de Linux – **GNU Debian**, Ubuntu
  - Linie de comandă
  - Edior de texte: Emacs, Gedit, mcedit, nano
  - Compilare în linie de comanda: gcc
- Campus Virtual:
  - Tutorial video de instalare VirtualBox, Ubuntu – contine si un exemplu simplu de scriere a unui program C

Introducere

Elemente de utilizare Linux

- Linux
  - Format din 2 componente: Kernel (nucleu), biblioteci software
    - Kernel – implementat inițial de Linus Torvalds (release 1991) – in 2021 are peste 27.8 milioane linii de cod
    - Biblioteci software – GNU Project – Richard Stallman (RMS) – Free Software Movement
  - Kernel + GNU Project = GNU/Linux
  - Distribuții Linux
    - Free: Debian (Debian project), Ubuntu (Canonical Ltd.), Fedora (Project Fedora – Red Hat)
    - Comerciale: Red Hat Enterprise Linux, Suse Linux Enterprise Server
  - codul este disponibil: licență GNU General Public License (GPL)
  - 90% cloud și servere, 74% smartphone (2020)
  - Cel mai “apropiat” sistem de operare pentru dezvoltatori software (și puțin embedded)
  - Mai puțin suport pentru dezvoltare embedded – dar în creștere
  - Caracterizat prin simplitate și omogenitate

Introducere

Elemente de utilizare Linux

- Sistemul de fișiere
  - Structură arborescentă cu intrări bine definite
  - Rădăcina sistemului (/) – denumit și root (rădăcină)
  - Toate căile de fișiere și directoare se pot scrie absolut relativ la /
  - Suport pentru fișiere, directoare (foldere – Windows), legături simbolice (≈ shortcut – Windows)
  - Diferită fundamental și structural față de sistemul de fișiere din Windows dar prezintă și asemănări
  - Identic pentru toate variațiile (distribuțiile) de Linux și Unix
  - Directoarele, driverele, echipamentele periferice – tipuri diferite și specializate de fișiere

```
/
|-- bin
|-- boot
|-- dev
|-- etc
|-- home
| |-- student
| |-- student1
|-- lib
|-- lib32
|-- lib64
|-- media
|-- mnt
|-- opt
|-- proc
|-- root
|-- run
|-- sbin
|-- srv
|-- sys
|-- tmp
|-- usr
| |-- bin
| |-- games
| |-- include
| |-- lib
| |-- lib32
| |-- local
| |-- sbin
| |-- share
| `-- src
|-- var
|-- initrd.img -> boot/initrd.img-4.9.0-8-amd64
|-- initrd.img.old -> boot/initrd.img-4.9.0-7-amd64
|-- vmlinuz -> boot/vmlinuz-4.9.0-8-amd64
`-- vmlinuz.old -> boot/vmlinuz-4.9.0-7-amd64
```

Introducere

Elemente de utilizare Linux

- Semnificația principalelor directoare
- **/bin** – “binaries” - binarele (fișierele executabile) ale principalelor programe utilitare fundamentale: ls, cp, ln, less, echo, touch,….
- **/boot** – fișiere necesare pornirii sistemului (kernel, imagine inițială, etc)
- **/dev** – “devices” – fișiere speciale ce reprezintă echipamentele hardware ale sistemului (/dev/mem – întreaga memorie a sistemului, /dev/cdrom – unitatea optică a sistemului)
- **/etc** – dedicat stocării fișierelor de configurare ale programelor instalate în sistem
- **/lib, /lib32, /lib64** – “libraries” – conține bibliotecile software din system (similar cu C:\Windows\System32)
- **/usr** – “user filesystem” – conține programele utilizator instalate în sistem sub formă arborescentă similară cu / (root) - /usr/bin, /usr/lib, …..
- **/tmp** – “temporar” - folosit pentru stocarea de fișiere temporare (similar cu C:\Windows\Temp)
- **/home** – conține directoarele personale ale fiecărui utilizator din sistem
  - Similar cu C:\Users – începând cu Windows 7
  - Fiecare utilizator are aici un director personal în care are drepturi depline de creare, ștergere, execuție fișiere și directoare

```
/
|-- bin
|-- boot
|-- dev
|-- etc
|-- home
| |-- student
| |-- student1
|-- lib
|-- lib32
|-- lib64
|-- media
|-- mnt
|-- opt
|-- proc
|-- root
|-- run
|-- sbin
|-- srv
|-- sys
|-- tmp
|-- usr
| |-- bin
| |-- games
| |-- include
| |-- lib
| |-- lib32
| |-- local
| |-- sbin
| |-- share
| `-- src
|-- var
|-- initrd.img -> boot/initrd.img-4.9.0-8-amd64
|-- initrd.img.old -> boot/initrd.img-4.9.0-7-amd64
|-- vmlinuz -> boot/vmlinuz-4.9.0-8-amd64
`-- vmlinuz.old -> boot/vmlinuz-4.9.0-7-amd64
```

Introducere

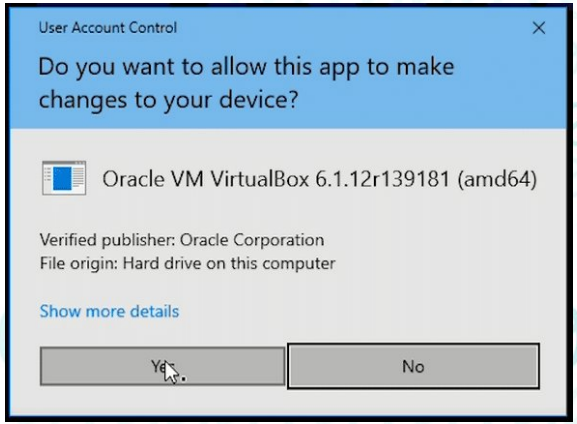
Elemente de utilizare Linux

- Utilizatorii din sistemele Linux
  - Utilizatori obișnuiți
    - Au director personal de home în /home/<nume_utilizator>
    - Au drepturi depline de a crea, sterge, modifica fișiere în directorul home
    - Au drepturi (sau nu) de a executa programre din directorul personal
    - Au drepturi (sau nu) de a executa aplicații/programe puse la dispoziție de sistem
    - NU au drepturi de a instala/dezinstala aplicații din sistem
    - NU au drepturi de a modifica parametrii și configurările sistemului
    - Protejați obligatoriu de o parolă
  - Utilizatorul ***root***
    - Singurul utilizator cu drept de adminstrare în sistemele Linux, UNIX
    - Are drept deplin asupra sistemului
    - Are drept de configurare/instalare/dezinstalare programe/aplicații din sistem
    - Are drept de a modifica drepturile altor utilizatori
    - Are director personal în /root (nu în /home ca și ceilalți utilizatori).
    - Similar cu utilizatorul “Administrator” din Windows

Introducere

Elemente de utilizare Linux

- Utilizatorii din sistemele Linux
  - Utilizatorul se poate schimba oricând în timpul rulării sistemului prin logout ☾ login
  - Se poate schimba și în utilizatorul root în orice moment (prin comanda ***su***)
  - Unii utilizatori pot avea dreptul de a cere drept de *root (de administrare)* pentru execuția unor programe și comenzi folosind comanda ***sudo***. (fără a se schimba în userul *root*)
  - Comanda sudo permite unui utilizator să execute o comanda sau program cu drept de root
    - Similar cu user account control (UAC) din Windows



Introducere

Elemente de utilizare Linux

- Prompt
  - Sintaxă: user@hostname:path$
    - user – reprezintă numele utilizatorului autentificat
    - hostname – reprezintă numele calculatorului
    - path – reprezintă calea curentă
  - Calea
    - Absolută – se specifică întreaga cale începând cu referința /
      ex: /home/student1/directorul_meu
    - relativă – se specifică calea relativ la calea curentă
      ex: calea curentă /home , calea relativă student1/directorul_meu
    - Semnul ~ (tilda) – reprezintă o scurtătură către directorul personal (home) al utilizatorului autentificat

    Ex: pentru student1 – ~ Ⓟ /home/student1
    pentru root - ~ Ⓟ /root
    pentru student - ~ Ⓟ /home/student
  - Consola/terminalul implicit se deschide în directorul home al utilizatorului autentificat

```
/
|-- bin
|-- boot
|-- dev
|-- etc
|-- home
|  |-- student
|  |-- student1
|-- lib
|-- lib32
|-- lib64
|-- media
|-- mnt
|-- opt
|-- proc
|-- root
|-- run
|-- sbin
|-- srv
|-- sys
|-- tmp
|-- usr
|  |-- bin
|  |-- games
|  |-- include
|  |-- lib
|  |-- lib32
|  |-- local
|  |-- sbin
|  |-- share
|  `-- src
|-- var
|-- initrd.img -> boot/initrd.img-4.9.0-8-amd64
|-- initrd.img.old -> boot/initrd.img-4.9.0-7-amd64
|-- vmlinuz -> boot/vmlinuz-4.9.0-8-amd64
`-- vmlinuz.old -> boot/vmlinuz-4.9.0-7-amd64
```

Introducere

Elemente de utilizare Linux

- Comenzi de bază
  - Schimbarea directorului: cd (change directory)
    - Sintaxa:

```
cd [cale_director]
```

```
valy@staff:~$
valy@staff:~$ cd public_html
valy@staff:~/public_html$ cd

valy@staff:~$ cd public_html
valy@staff:~/public_html$ cd ..

valy@staff:~$ cd so/pipe/
valy@staff:~/so/pipe$ cd ..
valy@staff:~/so$ cd ..
valy@staff:~$
valy@staff:~$ cd /home/valy/public_html
valy@staff:~/public_html$
```

Introducere

Elemente de utilizare Linux

- Comenzi de bază
  - Listare conținutului unui director: ls (list directory contents)
    - Sintaxa:

```
ls [argumente] [cale_director]
```

```
valy@staff:~$ ls
apache2passwd  dis   mac_process  removeoffender.sh  wake_server
backups        ipxe  pi           scripts
comm_test      key   public_html  teaching
valy@staff:~$ ls dis
distribute  hosts.xml  id_dsa  id_dsa.pub
valy@staff:~$ ls -l
total 52
-rw-r--r--  1 valy valy      43 Jul 28  2018 apache2passwd
drwxr-xr-x  2 valy valy    4096 Jul 20 14:40 backups
drwxr-xr-x  2 valy valy    4096 Oct 17  2018 comm_test
drwxr-xr-x  2 valy staffcs 4096 Aug 23  2016 dis
drwxr-xr-x  5 valy staffcs 4096 Jan  8  2018 ipxe
drwxr-xr-x  2 valy staffcs 4096 Oct 18  2016 key
drwxr-xr-x  2 valy staffcs 4096 Dec  2  2019 mac_process
drwxr-xr-x  4 valy staffcs 4096 May 11  2018 pi
drwxr-xr-x  8 valy staffcs 4096 Mar  5 08:46 public_html
-rwxr-xr-x  1 valy valy     583 Nov 18  2019 removeoffender.sh
drwxr-xr-x  6 valy staffcs 4096 Jul 20 14:39 scripts
drwxr-xr-x 13 valy valy    4096 Jul 20 14:42 teaching
drwxr-xr-x  2 valy valy    4096 Apr 17  2019 wake_server
valy@staff:~$
```

Introducere

Elemente de utilizare Linux

- Comenzi de bază
  - Crearea unui director: mkdir (make directory)
    - Sintaxa:

```
mkdir <nume_director]
```

```
valy@staff:~$ ls
apache2passwd  dis   mac_process  removeoffender.sh  wake_server
backups        ipxe  pi           scripts
comm_test      key   public_html  teaching
valy@staff:~$ mkdir directorul_meu
valy@staff:~$ ls
apache2passwd  directorul_meu  key          public_html        teaching
backups        dis             mac_process  removeoffender.sh
wake_server
comm_test      ipxe            pi           scripts
valy@staff:~$ cd directorul_meu
valy@staff:~/directorul_meu$ cd ..
valy@staff:~$
```

Introducere

Elemente de utilizare Linux

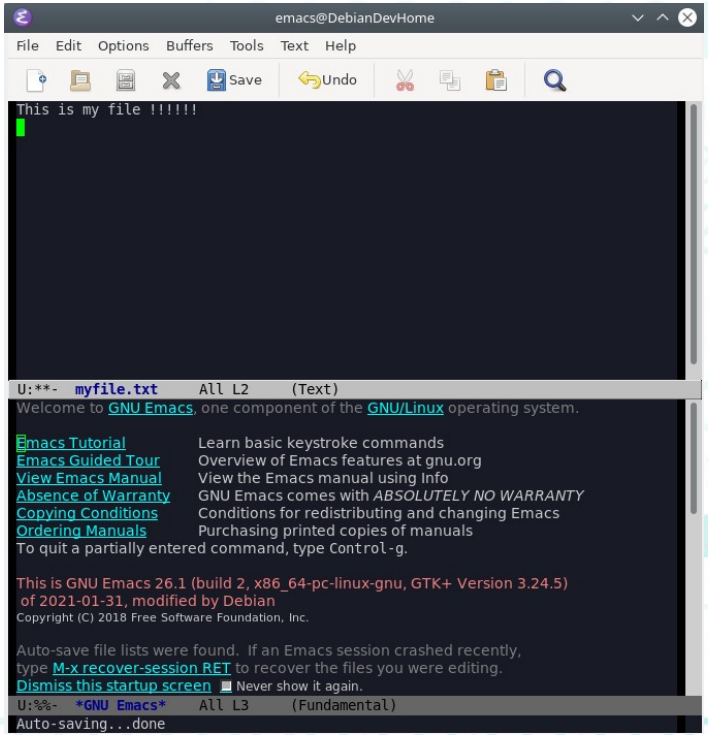
- Pagini de manual
  - În Linux și Unix există pagini de manual pentru majoritatea comenzilor și funcțiilor C standard
  - Se folosește comanda man
    - Sintaxa: `man [optiuni] [seciune] comanda`
  - Secțiuni pagini de manual:
    1. Secțiunea 1 - descrie comenzile standard (programe executabile și comenzi shell script)
    2. Secțiunea 2 - apeluri sistem UNIX apelabile în limbajul C
    3. Secțiunea 3 - funcțiile de bibliotecă C
    4. Secțiunea 4 - informații despre fișierele speciale (în principal cele din /dev)
    5. Secțiunea 5 - informații despre convențiile și formatele anumitor fișiere specifice sistemului
    6. Secțiunea 6 - manuale de la jocurile din Linux
    7. Secțiunea 7 - informații despre diverse teme ce nu pot fi incluse în alte secțiuni (spre exemplu man 7 signal)
    8. Secțiunea 8 - comenzi de administrare a sistemului (de obicei doar pentru userul root)
    9. Secțiunea 9 - rutine kernel
  - Daca nu se specifică secțiunea, se caută comanda în ordin crescător al secțiunilor și se returnează prima apariție
  - Din pagina de manual se iese cu tasta **q**

```
man printf          man 2 read          man strcpy          man scanf
man ls              man fread           man mkdir           man man
```

Introducere

Elemente de utilizare Linux

- Editarea si creare unui fișier text folosind un editor de texte (Emacs)
  - Sintaxa: `editor_text nume_fisier &`
  - Exemplu folosind emacs: `emacs file.txt &`
  - Semnul & - ampersant – necesar pentru a “trimite” comanda de dinainte lui în background
    - Lipsa operatorului & ar “bloca” terminalul până la închiderea comenzii/programului lansat
  - Keyboard shortcuts:
    - C-x C-s = CTRL+X CTRL+S – save (sau din meniu File)
    - C-x C-c = CTRL+X CTRL+c – quit (sau din meniu File)
    - C-x C-1 = CTRL+X CTRL+1 – scoate fereastra de jos (….)



Introducere

Elemente de utilizare Linux

- Comenzi de bază
  - Afisarea unui fișier text în terminal: comanda cat
    - Sintaxa:

```
cat <cale_fisier>
```

```
valy@staff:~$ cat file.txt
This is my file !!!!!!
valy@staff:~$
```

- Extensii de fișiere: nume_fisier.**ext**
- Sistem de operare Linux – NU folosește extensii
  - Extensiile de fișiere sunt folosite în Linux doar de utilizator și anumite programe pentru a interpreta mai ușor anumite fișiere ☾ pentru asocierea programelor cu anumite tipuri de fișiere prin extensii;
  - Ex: dacă fișierul este cu extensia .c editorul de texte poate activa *syntax highligh* (aplicarea unei scheme de culori în funcție de elementele de sintaxă ale limbajului)

Mai multe despre Linux:

Sisteme de operare

an 2, Sem 2

disciplină obligatorie

Introducere

Limbajul C

Introducere

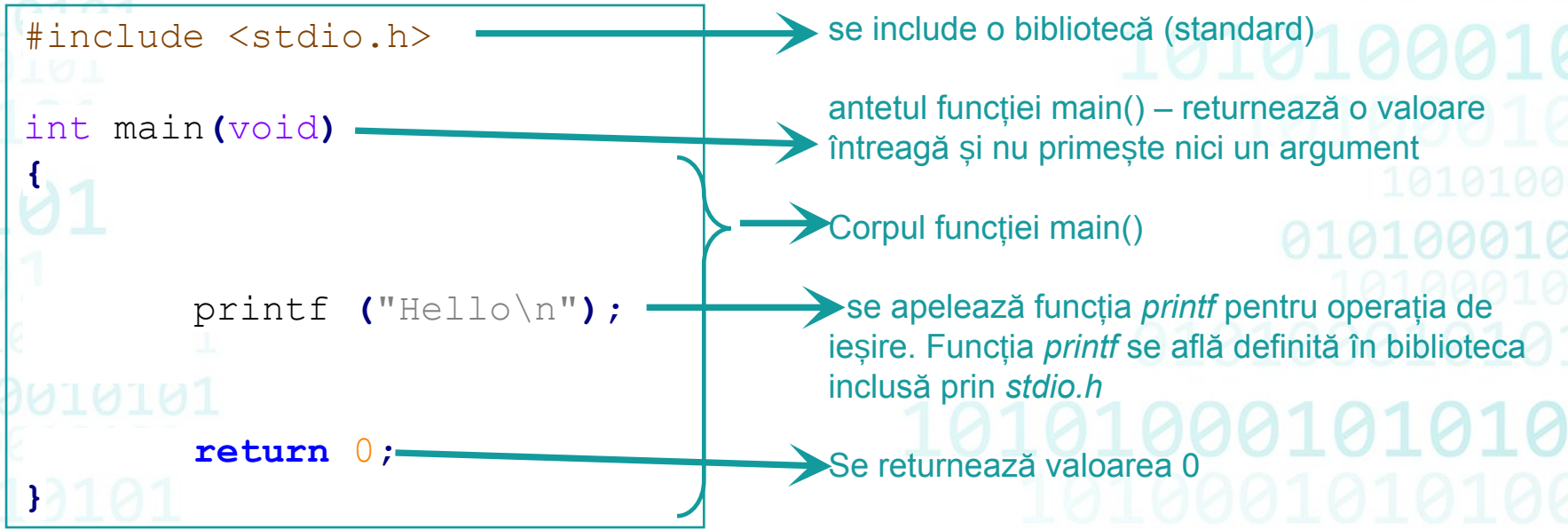
Limbajul C

- Limbaj de programare de nivel înalt, procedural, de generația a treia
- Proiectat și implementat de Dennis Ritchie în 1973 la AT&T Bell Labs
- Realizat pentru a ajuta la scrierea anumitor părți din kernel-ul de UNIX dezvoltat de Ken Thompson în 1971-1973
- Kernel-ul de UNIX scris inițial în limbaj de asamblare ajunge să fie rescris în totalitate în C
- Cartea de referință:
  - Brian W. Kernighan şi Dennis Ritchie, "The C Programming Language" 1978, Prentice Hall
- Reprezintă unul dintre cele mai populare limbaje de programare
- Limbaj de bază în învățarea altor limbaje de programare
- Oferă un grad extrem de ridicat de portabilitate la nivel de cod
- Dezavantaj: limbajul nu face verificări exhaustive dpdv al tipurilor de date, la conversii de date Ⓟ multe erori de execuție uneori greu de rezolvat

Introducere

Limbajul C

- Un program C
  - Format din funcții și variabile ☾ cod executabil poate exista doar în cadru funcțiilor
  - Conține obligatoriu funcția main() – execuția începe cu această funcție
    - Functia main() ☾ Entry point



Introducere

Limbajul C

- Scrierea și compilarea unui program C într-un mediu de dezvoltare sub Linux
  - Deschiderea și editarea unui fisier text de cod C

```
valy@staff:~$ emacs program.c &
```

  - Scrierea programului în editorul de texte
  - Salvarea fișierului
  - Compilarea programului

```
valy@staff:~$ gcc –Wall –o prog_executabil program.c
```

  - Rezolvarea eventualelor erori de compilare. In urma unei compilări cu succes se va crea fișierul binar și executabil *prog_executabil* (*gcc* nu tipărește nici un mesaj dacă compilarea este cu succes)
  - Se execută programul

```
valy@staff:~$ ./prog_executabil
Hello
valy@staff:~$
```

Introducere

Limbajul C

- Analiza comenzii de invocare a compilatorului

```
valy@staff:~$ gcc -Wall -o prog_executabil program.c
```

  - Propunere de sintaxă generală

```
valy@staff:~$ gcc -Wall -o <nume_program_executabil> <fisier1.c> <fisier2.c> …
```

- gcc - compilatorul de C - GNU C Compiler - program existent pe disc
- -Wall - parametru al gcc prin care i se spune compilatorului să afișeze toate mesajele de tip Warning (atenționări). Mesajele de tip warning nu duc la necompilarea programului dar sunt extrem de importante. NU se recomandă ignorarea lor. Lipsa acestui argument poate duce la a nu se afișa toate warning-urile și astfel duce le ignorarea acestora
- -o <nume_program_executabil> - în cazul unei compilări cu succes se precizează care să fie numele programului binar executabil. Lipsa acestei opțiuni duce la crearea unui program executabil nu mele **a.out** . În cazul unor erori, programul executabil NU este creat.
- <fisier1.c> <fisier2.c> - se specifică fișiere .c ce vor fi compilate spre a produce un program executabil nou

```
valy@staff:~$ ./prog_executabil
Hello
valy@staff:~$
```

Introducere

Limbajul C

- Analiza comenzii de invocare de rulare a programului executabil

```
valy@staff:~$ ./prog_executabil
Hello
valy@staff:~$
```

- În Linux, pentru orice program executabil este necesar să i se specifice calea
  - În exemplul de mai sus – **cale relativă** la directorul curent
  - Exemplu similar dar cu cale absolută

```
valy@staff:~$ /home/valy/prog_executabil
Hello
valy@staff:~$
```

  - În cazul nespecificării căii nici absolute și nici relative sistemul de operare va genera eroare
    - Sistemul de operare “nu știe” de unde să execute comanda / programul

```
valy@staff:~$ prog_executabil
prog_executabil: command not found
valy@staff:~$
```

  - Se preferă execuția conform primului exemplu, folosind cale relativă

Introducere

Limbajul C

- Un alt program C, mai “complicat”

```c
#include <stdio.h>

int m;

int main(void)
{
        // this is a one line comment
        /*
                This is a multiple line comment
        */
        int a = 0;
        int b;

        a = 4; // this is another one line comment
        b = 10;


        m = (a + b) / 2;
        printf ("Mean of %d and %d is %d\n", a, b,
m);
        return 0;
}
```

declaratie de variabilă globală

Comentariu in C pe o linie

Comentariu in C pe mai multe linii

Declaratii de variabile locale a unei funcții cu sau fără inițializare

Asignare de valori unor variabile

Asignare de valoare unei variabile cu rezultatul unei expresii

Secțiunea II

Baze de numerație

Baze de numerație

Definiții

- Bază de numerație – număr de unități necesare pentru a se obține o unitate de ordin imediat superior

$$d_{n-1}d_{n-2}d_{n-3}...d_2d_1d_0 = d_{n-1}\cdot b^{n-1} + d_{n-2}\cdot b^{n-2} + d_{n-3}\cdot b^{n-3} + ... + d_2\cdot b^2 + d_1\cdot b^1 + d_0\cdot b^0$$

  - d – digit (cifra) , b – baza de numerație
- Baza de numerație 10 – fiecare cifră a numărului are o pondere de 10 unități (b = 10)
  - Exemplu: $39741_{(10)} = 3\cdot10^4 + 9\cdot10^3 + 7\cdot10^2 + 4\cdot10^1 + 1\cdot10^0$

Baze de numerație

Baza de numerație 10

- Baza de numerație 10 – fiecare cifră a numărului are o pondere de 10 unități (b = 10)
  - Exemplu: $39741_{(10)} = 3 \cdot 10^4 + 9 \cdot 10^3 + 7 \cdot 10^2 + 4 \cdot 10^1 + 1 \cdot 10^0$
- In limbajul C, numerele în baza 10 se reprezintă direct, in limbaj natural arab (as is)

Baze de numerație

Baza de numerație 2

- Baza de numerație 2 – fiecare cifră a numărului are o pondere de 2 unități (b = 2)
  - Cifrele (digits): 0 , 1
  - Exemplu 1: Exprimare număr în baza 2 și transformarea lui în baza 10

$$1100010110_{(2)} = 1\cdot2^9 + 1\cdot2^8 + 0\cdot2^7 + 0\cdot2^6 + 0\cdot2^5 + 1\cdot2^4 + 0\cdot2^3 + 1\cdot2^2 + 1\cdot2^1 + 0\cdot2^0$$

$$1100010110_{(2)} = 512 + 256 + 16 + 4 + 2 = 790_{(10)}$$

  - Transformarea unui număr din baza 10 în baza 2 – prin împărțire repetată până când câtul devine 0. Rezultatul se obține prin preluarea valorilor resturilor în ordinea inversă obținerii acestora
  - Exemplu 2: Transformarea unui număr (29) din baza 10 în baza 2

$$29_{(10)} = 11101_{(2)}$$

Operație	Cât	Rest	
29:2	14	1	d0
14:2	7	0	d1
7:2	3	1	d2
3:2	1	1	d3
1:2	0	1	d4

Baze de numerație

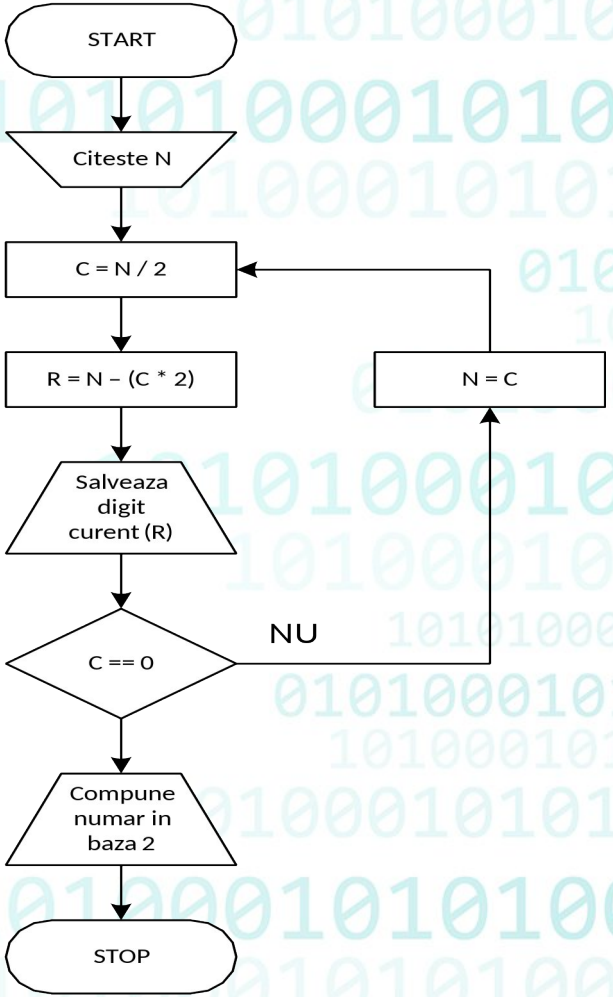
Baza de numerație 2

- Algoritm transformare număr din baza 10 în baza 2

- Limbajul C ofera suport pentru lucrul direct cu numere în baza 2. Se reprezintă folosind prefixul 0b în fața numărului. Exemplu

```
int n = 0b00111001;
```

- ATENȚIE: Nu toate compilatoarele suporta această facilitate



Baze de numerație

Baza de numerație 2

- Baza de numerație 2
  - extrem de importantă în sisteme de calcul – circuitele electronice logice funcționează pe principiul 0 / 1 (0V / 5V)
  - Operațiile în baza 2 – algebra Booleana: SI, SAU, NU
    - matematicianul englez George Boole, “The Laws of Thought” (1853)
  - În calculatoare – o cifră (digit) în baza 2 - bit ; 8 biți - 1 byte (octet)
  - byte – unitatea elementară de stocare a informației ce poate fi adresată de un sistem de calcul

Exemplu: exprimarea numărului 29 din baza 10 în baza 2 pe dimensiunea de 1 byte

$$29_{(10)} = 00011101_{(2)}$$

- MSB – Most Significant Bit (cel mai semnificativ bit)
- LSB – Least Significant Bit (cel mai puțin semnificativ bit)

Baze de numerație

Puteri ale lui 2

N	2^N	N	2^N
0	1	**9**	512
1	2	**10**	1.024
2	4	**11**	2.048
3	8	**12**	4.096
4	16	**13**	8.192
5	32	**14**	16.384
6	64	**15**	32.768
7	128	**16**	65.536
8	256	**17**	131.072

Baze de numerație

Reprezentarea numerelor cu semn în binar complement de 2

- se folosește pentru a reprezenta binar numere cu semn
- se rezervă cel mai semnificativ bit al unui număr pentru reprezentarea semnului
  - daca MSB = 0 - numărul se consideră a fi pozitiv
  - daca MSB = 1 - numărul se considera a fi negativ
- numar pozitiv pe n biți, 1 bit rezervat pentru semn:

$$0d_{n-2}d_{n-3}...d_2d_1d_0 = d_{n-2}\cdot 2^{n-2} + d_{n-3}\cdot 2^{n-3} + ... + d_2\cdot 2^2 + d_1\cdot 2^1 + d_0\cdot 2^0$$

- număr negativ pe n biți, 1 bit rezervat pentru semn

$$1d_{n-2}d_{n-3}...d_2d_1d_0 = -2^{n-1} + d_{n-2}\cdot 2^{n-2} + d_{n-3}\cdot b^{n-3} + ... + d_2\cdot b^2 + d_1\cdot b^1 + d_0\cdot b^0$$

Baze de numerație

Baza de numerație 16

- Baza de numerație 16 – fiecare cifră a numărului are o pondere de 16 unități (b = 16)
  - Cifrele (digits):

Hexazecimal	Zecimal	Binar	Hexazecimal	Zecimal	Binar
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	A, a	10	1010
3	3	0011	B, b	11	1011
4	4	0100	C, c	12	1100
5	5	0101	D, d	13	1101
6	6	0110	E, e	14	1110
7	7	0111	F, f	15	1111

Baze de numerație

Baza de numerație 16

- Se folosește pentru a facilita reprezentarea numerelor
- Baza 16 este strans legată de baza 2
- Transformarea în zecimal se face folosind acelasi algoritm
  - Exemplu. Transformarea numărului A8E9 din baza 16 in baza 10

$$A8E9_{(16)} = 10 \cdot 16^3 + 8 \cdot 16^2 + 14 \cdot 16^1 + 9 \cdot 16^0 = 43241_{(10)}$$

- Transformarea în binar se face folosind tabelul anterior cu reprezentarea cifrelor bazei 16

$$A8E9_{(16)} = 1010\ 1000\ 1110\ 1001_{(2)}$$

- Limbajul C ofera suport pentru lucrul direct cu numere în baza 16. Se reprezintă folosind prefixul 0x în fața numărului. Exemplu

```
int n = 0xA8E9;
```

Mai multe despre baza 2, baza 16, reprezentarea numerelor:
Arhitectura Calculatoarelor
an 2, Sem 1
disciplină obligatorie
Logica digitală
An 1, Sem 2
disciplină obligatorie

Secțiunea III

Tipuri de date. Variabile. Expresii. Operatori aritmetici

Tipuri de date

Tipuri de date

Tipuri de date

- Tipuri de date
  - Fundamentale
    - **char** – tipul caracter
    - **int** – tipuri întregi
    - **float**, **double** – tipuri reale (în virgulă flotantă)
    - **enum** – tipuri enumerate
    - **void** – tip vid
  - Derivate
    - Tablouri (array)
    - pointer
    - **struct** – structuri
    - **union** - uniuni

Tipuri de date

Tipuri de date

- Modificatori de tip
  - Modificator: cuvând cheie ce are menirea de schimba proprietățile tipurilor de date fundamentale, în sensul de a modifica intervalul de valori și memoria necesară stocării unei variabile de acel tip
  - Modificatori de semn
    - ***signed*** (implicit când nu este specificat)
    - **unsigned**
  - Modificatori de dimensiune
    - **short**
    - **long**
    - **long long**

Tipuri de date

Tipuri de date standard întregi

Modificator semn	Modifcator dimensiune	Tip standard	Dimensiune (bytes)	Dimensiune minima (bytes)	Interval de valori
signed		char	**1**	1	$-2^7 \dots 2^7-1$ −128 ...127
unsigned			**1**	1	$0 \dots 2^8-1$ 0 ...255
signed	short	int	**2**	2	$-2^{15} \dots 2^{15}-1$ −32.768...32.767
unsigned			**2**	2	$0 \dots 2^{16}-1$ 0 ...65535
signed			**4** sau 2	2	$-2^{31} \dots 2^{31}-1$ −2.147.483.648...2.147.483.647
unsigned			**4** sau 2	2	$0 \dots 2^{32}-1$ 0 ...4.294.967.295
signed	long		**8** sau 4	4	$-2^{63} \dots 2^{63}-1$
unsigned			**8** sau 4	4	$0 \dots 2^{64}-1$
		float	Tipuri de date pentru numere în virgulă flotantă		
		double			

Tipuri de date

Tipuri de date standard

- Avantaje
  - portabile la nivel de cod
  - nu necesită biblioteci externe – sunt implicite în limbaj (standard)
  - disponibile pe orice compilator ce respectă standard-ul (în general)
- Dezavantaje
  - dimensiunea lor depinde de arhitectura procesorului si de versiunea de compilator
  - din cauza diferentei de dimensiune pot aparea probleme de portabilitate
  - în general crează mari probleme de portabilitate
- **Soluția**: tipuri de date din biblioteca standard **stdint.h**
  - dimensiunea este fixă și nu depinde de arhitectură
  - nu depind de compilator fiind o bibliotecă externă
  - dezavantaj: este o bibliotecă externă ce trebuie inclusă în fișierele de cod

Tipuri de date

Tipuri de date din biblioteca standard stdint.h

- Convenție generală

  `[u]intXX_t`

  – [u] – caracterul u – opțional – unsigned – specifică tip întreg fără semn – cu semn dacă lipsește

  – XX – dimensiunea în biți a tipului de date

Tip de date	Semn	Dimensiune (bytes)	Interval de valori
`int8_t`	cu semn	1	$-2^7 \dots 2^7-1$ −128 ...127
`uint8_t`	fără semn	1	$0 \dots 2^8-1$ 0 ...255
`int16_t`	cu semn	2	$-2^{15} \dots 2^{15}-1$ −32.768...32.767
`uint16_t`	fără semn	2	$0 \dots 2^{16}-1$ 0 ...65535
`int32_t`	cu semn	4	$-2^{31} \dots 2^{31}-1$ −2.147.483.648...2.147.483.647
`uint32_t`	fără semn	4	$0 \dots 2^{32}-1$ 0 ...4.294.967.295
`int64_t`	cu semn	8	$-2^{63} \dots 2^{63}-1$
`uint64_t`	fără semn	8	$0 \dots 2^{64}-1$

Tipuri de date

Dimensiunea unui tip

- Limbajul C oferă un operator pentru aflarea dimensiunii unui tip: sizeof. Sintaxă:

```
sizeof(tip);
sizeof(variabilă);
```

- Operatorul sizeof returnează dimensiunea în bytes a tipului sau a variabilei dată ca argument
- Exemplu:

```
int n = 0;
n = sizeof(n);              // 4
n = sizeof(int);            // 4
n = sizeof(uint8_t);        // 1
n = sizeof(uint16_t);       // 2
n = sizeof(uint64_t);       // 8
```

Constante

Constante

Definiții

- Valori fixe ce se pot atribui variabilelor sau pot fi implicate în calculele din expresii
- exemplu constantă: *67455*
- o constantă întreagă este considerată implicit o constantă de tip **int**
- se poate specifica către compilator tipul unei constante prin sufixe:
  - constantă de tip long: adăugarea sufixului L sau l . Exemplu: **345L** sau 345l
  - constantă de tip unsigned: adăugarea sufixului U sau u . Exemplu: **27U** sau 27u
  - constantă de tip long unsigned: adăugarea sufixului UL sau ul . Exemplu **34UL** sau 34ul
- constantă în virgulă flotantă: 56.23
  - implicit se consideră de tip double
  - constantă de tip float: adăugarea sufixului F sau f . Exemplu: **17.24F** sau 17.24f
  - constantă de tip long double: adăugarea sufixului L sau l . Exemplu: **167.12L** sau 167.12l

Constante

Definiții

- Contantă în forma hexazecimal: se adaugă prefixul **0x** sau 0X. Se poate adăuga și sufix.
  - Exemplu: 0x0A34 , 0x12345678UL
- Constantă în format binar: se adaugă prefixul **0b** sau 0B
  - Exemplu: 0b0011 , 0b00111110
- o constantă caracter este reprezentată ca un întreg și se scrie între ghilimele simple: ‘a’ ‘b’ ‘c’

Variabile

Variabile

Definiții

- Variabila este un nume, dat unei zone de memorie, ce stochează date de o anumită dimensiune dată, de dimensiunea tipului de date a variabilei
- datele din zona de memorie referită de variabilă sunt interpretate în funcție de tipul variabilei
- Declarație de variabilă

```
tip identificator;
```

  - *tip* – tipul de date al variabilei (standard sau definit de utilizator)
  - *identificator* – numele variabilei – caractere alflanumerice și caracterul special ‘ _ ‘ (underscore), nu poate să înceapă cu cifră (digit), nu poate conține spații și cuvinte cheie, poate să înceapă cu underscore, trebuie să fie unic (la nivel de context de vizibilitate), nu poate fi vreun cuvânt cheie
- Declarație multiplă de variabile de același tip

```
tip identificator1, identificator2, identificator3;
```

- Atentie: Limbajul C este case-sensitive: face diferența între litere mari și litere mici

Variabile

Definiții

- Declarația de variabilă
  - trebuie făcută înainte de folosirea variabilei în orice mod în program (o variabilă trebuie declarată înainte de utilizare)
  - specifică numele și tipul de date
  - se termină cu punct-și-virgulă **;**
  - nu pot exista declarate 2 variabile cu același nume în același context de vizibilitate
- Definiția unei variabile:
  - presupune o declarație completă a variabilei
  - implicit, dacă nu se specifică altfel, o declarație de variabilă implică și definiția ei
  - implică și alocarea de memorie de dimensiunea tipului variabilei
  - o variabilă “există” doar dacă este și definită (i se alocă și memorie)
- Exemple declarații variabile:

```
int a,b;
int x;
uint32_t N;
float my_float, VariabilaMeaInVirgulaFlotanta;
char ch_01;
```

Variabile

Statements. Blocuri de cod

- ***statement*** – (trad. afirmație, propoziție, **instrucțiune**) – reprezintă o linie de cod cum ar fi
  - apel de funcție
  - asignare de varibilă cu o valoare, expresie, alta variabilă
  - este urmată de semnul punct-și-virgulă **;** (statement terminator)
- ***code block*** – (trad. bloc de cod, bloc de instrucțiuni) – reprezintă o secvență de cod grupată între paranteze acolade { … } (braces) și formează un *compound statement*
  - sunt practic echivalente cu un *statement*
  - folosit ca să se grupeze cod
  - definește practic și un domeniu de vizibilitate al declarațiilor
  - nu se pune semnul punct-și-virgulă **;** practic terminatorul blocului este paranteza acoladă ce îl închide
  - se pot declară blocuri încuibate (nested), dar nu se obișnuiește decât în cazuri bine definite și nu ca și sub-bloc direct fără existența unei instrucțiuni

Variabile

Domenii de vizibilitate ale unei variabile

- o variabilă este vizibilă doar în interiorul blocului de cod în care a fost declarată și doar după ce a fost declarată
- o variabilă declarată în cadrul unui bloc este vizibilă și în sub-blocurile declarate în cadrul blocului inițial (părinte)
- o variabilă globală este vizibiă în tot fișierul de cod

```
int a;
int main(void)
{
            // x nu este vizibila aici
            // a este vizibila aici
            int x;
            {
                        int b;
                        // x este vizibila aici
            }
            // b nu este vizibila aici
            // x este vizibila aici
            // a este vizibila aici
}
```

Variabile

Domenii de vizibilitate ale unei variabile

- Consecințe
  - parametrii locali ai unei funcții sunt vizibili doar în interiorul funcției
  - variabilele locale ale unei funcții sunt vizibile doar în interiorul funcției
- Durata de viață (de existență în memorie) a unei variabile
  - variabilă locală a unei funcții – doar pe durata apelului și execuției funcției
  - parametrul local a unei funcții – doar pe durata apelului și execuției funcției
  - variabilă globală – pe toată durata execuției programului

Variabile

Domenii de vizibilitate ale unei variabile

- Modificatorul **const**: sintaxă: **`const tip identificator;`**
  - **`const`** – cuvânt cheie rezervat al limbajului C
  - pus în fața declarației unei variabile determina ca variabila respectivă să fie constantă (read-only), nu se poate modifica
  - o variabilă declarată cu const poate fi doar intițializată și utilizată doar în dreapta operatorului =
  - o variabilă declarată cu const nu i se poate atribui o valoarea – de obicei duce la eroare de compilator

```c
int main(void)
{
  const int x = 3;
  x = 2;
  x++;
  return 0;
}
```

```
valy@staff:~/teaching$ gcc -Wall -o p var_const.c
var_const.c: In function 'main':
var_const.c:4:5: error: assignment of read-only variable 'x'
   x = 2;
     ^
var_const.c:5:4: error: increment of read-only variable 'x'
   x++;
    ^~
```

Variabile

Atribuiri de valori unei variabile

- Atribuire de valoare unei variabile
  - operațiunea prin care se modifică conținutul zonei de memorie referențiat de variabilă
  - unei variabile i se poate atribui:
    - o constantă
    - valoarea unei alte variabile,
    - rezultatul unei expresii
  - sintaxă:

```
nume_variabila = expresie;
```

  - este de tip statement și se termină cu punct-și-virgulă ;
  - are ca efect evaluarea expresie din partea dreapă a operatorului “=”
  - semnul “=“ – reprezintă operatorul de atribuire
- exemple atribuire:

```
a = 5;
a = 3 + 1;
b = a * 5 + a;
c = a + b;
```

Variabile

Inițializarea unei variabile

- Inițializarea unei variabile reprezintă operațiunea prin care în momentul alocării variabilei de către compilator, conținutul zonei de memorie devine egal cu valoarea de inițializare
- o variabilă se poate inițializa doar cu o constantă
- ATENTIE:
  initializarea unei variabile NU este același lucru cu atribuirea de valoare
  - Atribuirea unei valori unei variabile generează cod executabil pe când inițializarea nu generează cod ci doar se scrie memoria înainte de execuția programului
- inițializarea se poate face o singură dată, odată cu declararea variabilei
- exemple de inițializări de variabile

```
int a = 3;
int x = 1 + 1;
int m,n = 0; // variabila m nu este initializata, doar n
int a1 = 0, a2 = 4;
```

Variabile

Inițializarea unei variabile

- O variabilă neinițializată are o valoare **nedefinită**
  - o valoare nedefinită
    - nu se cunoaște valoarea în nici un moment,
    - nu se poate presupune valoarea în nici un moment
    - poate avea orice valoare necunoscută și impredictibilă

- O variabila neinițializată - RISC MAJOR

Expresii. Operatori

Operatori

Definiții

- Operator – reprezentat printr-un simbol și arată ce operații se execută asupra unor operanzi (termeni)
- Operandul poate fi: o constantă, o variabilă, o funcție, o expresie
- În funcție de numărul operanzilor asupra cărora se aplică, operatorii pot fi:
  - unari – se aplică unui singur operator
  - binari – se aplică asupra a doi operatori
  - ternari – se aplică asupra a trei operatori
- În funcție de tipul operanzilor asupra cărora se aplică, operatorii pot fi:
  - aritmetici
  - relaționali
  - logici
  - binari
- Operatorii sunt împărțiți în clase de precedență (sau prioritate): definește ordinea în care se aplică operatorii într-o expresie cu mai mulți operatori
- Operatorii se aplică și în funcție de regulile de asociativitate: indică ordinea de aplicare a operatorilor atunci când într-o expresie există mai multi operatori din aceeași clasă de precedență
  - asociativitate stânga
  - asociativitate dreapta

Operatori

Listă operatori. Clasificare operatori

Clasă de precedență	Operator	Descriere	Tip operator	Asociativitate
1	[ ]	Indexare	binar	stânga->dreapta
	. şi ->	Selecție membru (prin structură, respectiv pointer)	binar	
	++ şi --	Postincrementare/postdecrementare	unar	
2	!	Negare logică	unar	dreapta->stânga
	~	Complement față de 1 pe biți Negare pe biți	unar	
	++ şi --	Preincrementare/predecrementare	unar	
	+ şi -	+ şi - unari	unar	
	*	Dereferențiere	unar	
	&	Operator adresă	unar	
	(tip)	Conversie de tip - typecast	unar	
	sizeof()	Mărimea în octeți	unar	

Operatori

Listă operatori. Clasificare operatori

Clasă de precedență	Operator	Descriere	Tip operator	Asociativitate
3	*	Înmulțire	binar	stânga->dreapta
	/	Împărțire	binar	
	%	Restul împărțirii	binar	
4	+ şi -	Adunare/scădere	binar	stânga->dreapta
5	<< si >>	Deplasare stânga/ dreapta a biților	binar	stânga->dreapta
6	<	Mai mic	binar	stânga->dreapta
	<=	Mai mic sau egal	binar	
	>	Mai mare	binar	
	>=	Mai mare sau egal	binar	
7	==	Egal	binar	stânga->dreapta
	!=	Diferit	binar	

Operatori

Listă operatori. Clasificare operatori

Clasă de precedență	Operator	Descriere	Tip operator	Asociativitate
8	&	ŞI pe biți	binar	stânga->dreapta
9	^	SAU-EXCLUSIV pe biți	binar	stânga->dreapta
10	|	SAU pe biți	binar	stânga->dreapta
11	&&	ŞI logic	binar	stânga->dreapta
12	||	SAU logic	binar	stânga->dreapta
13	?:	Operator condițional	ternar	dreapta->stânga

Operatori

Listă operatori. Clasificare operatori

Clasă de precedență	Operator	Descriere	Tip operator	Asociativitate
14	=	Atribuire	binar	dreapta->stânga
	+= şi -=	Atribuire cu adunare/scădere	binar	
	*= şi /=	Atribuire cu multiplicare/împărțire	binar	
	%=	Atribuire cu modulo	binar	
	&= si |=	Atribuire cu ŞI/SAU	binar	
	^=	Atribuire cu SAU-EXCLUSIV pe biți	binar	
	<<= şi >>=	Atribuire cu deplasare de biți	binar	
15	,	Operator secvență	binar	stânga->dreapta

- Adaptare după: Cristina Stângaciu, curs “Limbaje de Programare”, ETC, 2020

Operatori

Exemple precedență și asocitivitate

- Exemplu de aplicare a precedenței

$$x = 10 + 4 \cdot 7 = 10 + 28 = 38$$

- Exemplu de aplicare a asociativității (stânga->dreapta)

$$x = 10/5 \cdot 4 = 2 \cdot 4 = 8$$

- OBSERVAȚIE: NU se recomandă utilizarea unor expresii bazate doar pe regulile de precedență și asociativitate.
- Se recomandă utilizarea parantezelor simple (…) pentru gruparea operațiilor.
  - OFERĂ UN GRAD MAI MARE DE LIZIBILITATE DE COD

Operatori

Operatori aritmetici

- Operatori aritmetici în limbajul C
  - adunarea a două numere **+**
  - scăderea a două numere **–**
  - înmulțirea a două numere ***
  - împărțirea a două numere **/**
  - restul împărțirii a două numere întregi **%**
  - incrementarea unui număr (adunarea numărului cu valoarea 1) **++**
  - decrementarea unui număr (scăderea cu valoarea 1 a numărului) **–**
- Operatori relaționali
  - mai mic, mai mic sau egal, mai mare, mai mare sau egal: <, <=, >, >=
  - egal == folosit pentru testarea egalității a doi operanzi
  - diferit !=
- Operatorul de atribuire: **=** este folosit pentru a atribui valori
- ATENTIE: a nu se confunda operatorul de atribuire cu cel relațional de egalitate

Expresii

Definiții

- Expresie
  - este o combinație de operanzi separați prin operatori
  - operatorii sunt indicați explicit (nu ca în matematică)
  - parantezele simple (…) sunt folosite pentru a grupa părți de expresii (sub-expresii)
  - expresiile sunt evaluare în funcție de regulile de ordine de precedență și asociativitate – cea mai mare precedență o au expresiile din paranteze (…)
  - evaluarea unei expresii conduce la un rezultat (sau eroare – excepție)
- Exemplu expresie

```
x = 1 - 1 && 3 + 4 * 9

x = (1 - 1) && (3 + 4 * 9)
```

Expresii

Echivalențe

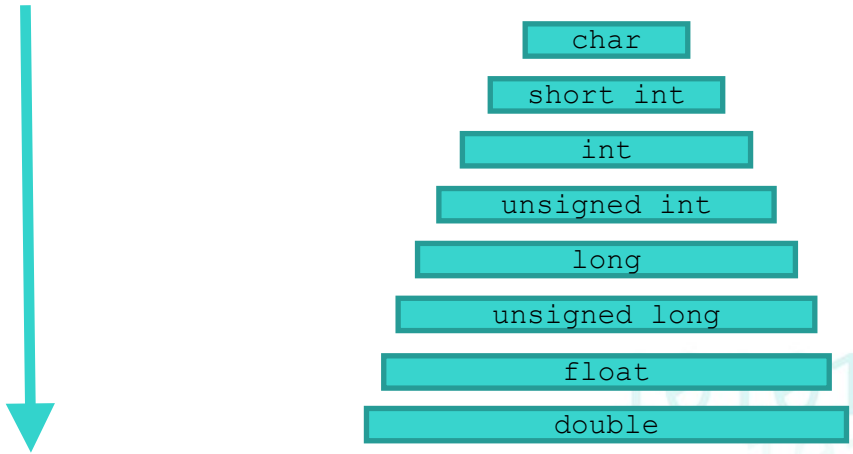
expresie	expresie echivalentă
`x++;`	`x = x + 1;`
`x--;`	`x = x - 1;`
`x+=5;`	`x = x + 5;`
`x-=2;`	`x = x - 2;`
`x/=10;`	`x = x / 10;`
`x*=23;`	`x = x * 23;`
`x%=9;`	`x = x % 9;`

Conversii de tipuri

Conversii de tipuri

Definiții. Exemple. Explicații

- Conversie implicită
  - Conversia de tip implicită apare atunci când un operator acționează unor variabile de tipuri diferite și se face conversia la un tip de date comun
  - de obicei se convertește un operand de la un tip cu domeniu de valori mai mic la un tip cu domeniu de valori mai mare



- Adaptare după: Cristina Stângaciu, curs “Limbaje de Programare”, ETC, 2020

Conversii de tipuri

Definiții. Exemple. Explicații

- Conversie implicită
  - Există situații când nu se poate converti la un tip date cu un domeniu mai cuprinzător ci la un tip de date cu domeniu mai restrâns - apare pierdere de informație - compilatorul poate sau nu să genereze *warning* de atenționare
  - Exemplu:

```
uint8_t n;
uint16_t m = 256;
n = m;
```

- variabila m este stocată pe 16 biți iar n pe 8 biți - pierdere de informație

m = 0000 0001 0000 0000

n

  - dacă semnele celor 2 operanzi sunt diferite și unul dintre tipuri cuprinde tot domeniul de valori al celulalt tip - conversia se face la tipul mai cuprinzător
  - dacă semnele celor 2 operanzi sunt diferite și ambele tipuri au aceeași dimensiune (același domeniu de valori) - conversia se face la tipul fără semn

- Adaptare după: Marius Minea, curs “Programarea calculatoarelor 2”, AC, 2005

Conversii de tipuri

Definiții. Exemple. Explicații

- Conversie explicită
  - Limbajul C oferă o modalitate prin care programatorul poate converti explicit la un anumit tip. Operațiunea se numește *type-cast*. Sintaxă:

```
(tip) expresie
```

  - Exemplu:

```
float f = 2.3;
int n = 0;
n = (int)f + 1;
```

Conversii de tipuri

Consecințe

- Operația de împărțire

```
float f = 0;
int n = 5;
f = n / 2; // f = 2.000000
f = (float)n / 2; // f = 2.500000
```

  - în linia 3 operația de împărțire se face ca fiind operație între întregi (fără virgulă flotantă) iar rezultatul este 2 care atribuindu-se lui f (float) devine 2.00000 în virgulă flotantă. acest lucru se întâmplă chiar dacă f este în virgulă flotantă deoarece întâi se evaluează expresia din dreapta și apoi cea din stânga (atribuirea)
  - în linia 4 se face o conversie explicită la float pentru operandul n iar astfel operația de împărțire se va face în virgulă flotantă (primul operand este în virgulă flotantă). Astfel, rezultatul este tot în virgulă flotantă iar în urma atribuirii rămâne neschimbat

Secțiunea IV

Controlul fluxului de execuție

Controlul fluxului

Elemente de matematică booleană

Elemente de algebră booleană

Definiții

- Operatori logici

Clasă de precedență	Operator	Descriere	Tip operator	Asociativitate
2	!	Negare logică	unar	dreapta->stânga
11	&&	ȘI logic	binar	stânga->dreata
12	||	SAU logic	binar	stânga->dreata

- tabele de adevăr

x	! x
true	false
false	true

x	y	x && y
true	true	true
true	false	false
false	true	false
false	false	false

x	y	x || y
true	true	true
true	false	true
false	true	true
false	false	false

- Adaptare după: Cristina Stângaciu, curs “Limbaje de Programare”, ETC, 2020

Elemente de algebră booleană

Implementare în C

- Valoarea **true** în C : (expresie) != 0
- Valoarea **false** în C : (expresie) == 0

- Exemplu 1:

$$x = e1 \;\&\&\; e2 \;\&\&\; e3$$

  - dacă e1 e falsă => **x** e fals - evaluarea e2 && e3 nu mai este necesară - evaluarea se oprește

- Exemplu 2:

$$x = e1||e2||e3$$

  - dacă e1 e true => **x** e true - evaluarea e2 || e3 nu mai este necesară - evaluarea se oprește

Controlul fluxului

Instrucțiuni condiționale. Instrucțiunea if

Instrucțiuni condiționale

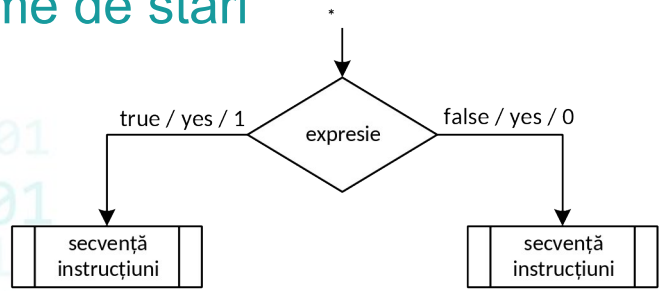
Instrucțiunea if

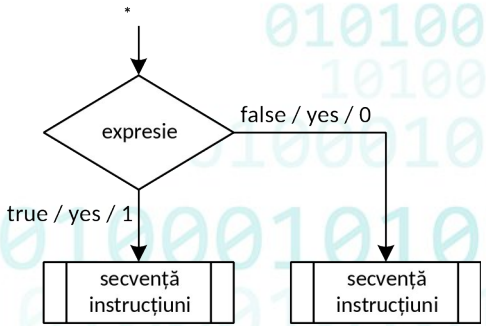
- Instrucțiune folosită pentru luarea de decizii
- Sintaxa

```
if (expresie)
        instrucțiune;
else
        instrucțiune;
```

```
if (expresie)
{
        secvență_instrucțiuni;
}
else
{
        secvență_instrucțiuni;
}
```

- Cuvinte cheie (cuvinte rezervate limbajului) – keywords: **if, else**
- Diagrame de stări





Instrucțiuni condiționale

Instrucțiunea if

- echivalență

```
if (expresie)
```

```
if (expresie != 0)
```

- partea de cod cu **else** este opțională
- cazuri de ambiguitate
  - de care instrucțiune **if** aparține **else** ?

```
if (a > 14)
          if (n == 0)
          x++;
else
          x--;
```

```
if (a > 14)
{
          if (n == 0)
          {
                    x++;
          }
          else
          {
                    x--;
          }
}
```

  - ambiguitate: din indentare proastă, din lipsă paranteze, din lipsă acolade
- recomandare: folosirea parantezelor în expresii și a acoladelor în cod

Instrucțiuni condiționale

Instrucțiunea if

- Exemplu simplu utilizare **if-else**:

```
#include <stdio.h>

int main(void)
{
        int n = 0;
        printf("Introduceti un numar: ");
        scanf("%d", &n); // citire intreg de la tastatura
        if (n >= 0)
        {
                printf ("\nNumarul este pozitiv\n");
        }
        else
        {
                printf ("\nNumarul este negativ\n");
        }
        return 0;
}
```

Instrucțiuni condiționale

Instrucțiunea if

- Implementare modulo (valoare absolută): $abs(x) = \begin{cases} -x, & x < 0 \\ x, & x \geq 0 \end{cases}$

```
#include <stdio.h>

int main(void)
{
        int n = 0;
        printf("Introduceti un numar:"); // citire intreg de la tastatura
        scanf("%d", &n); // citire intreg de la tastatura
        int abs = 0;
        if (n >= 0)
        {
                abs = n;
        }
        else
        {
                abs = -n;
        }

        printf ("valoare absoluta %d\n", abs);
        return 0;
}
```

Instrucțiuni condiționale

Expresia condițională

- reprezintă o varintă de implementare față de instrucțiunea if

Clasă de precedență	Operator	Descriere	Tip operator	Asociativitate
13	?:	Operator condițional	ternar	dreapta ☾ stânga

- Sintaxă
- Echivalență cu if

```
rez = (expresie) ? valoare_true : valoare_false;
```

```
if (expresie)
{
        rez = valoare_true;
}
else
{
        rez = valoare_false;
}
```

Instrucțiuni condiționale

Expresia condițională

- Implementare modulo (valoare absolută): $abs(x) = \begin{cases} -x, & x < 0 \\ x, & x \geq 0 \end{cases}$

```c
#include <stdio.h>

int main(void)
{
        int n = 0;
        printf("Introduceti un numar:"); // citire intreg de la tastatura
        scanf("%d", &n); // citire intreg de la tastatura
        int abs = 0;

        abs = n >= 0 ? n : -n;


        printf ("valoare absoluta %d\n", abs);
        return 0;
}
```

- mai lizibil și fără a ne baza pe precedență și asociativitate

```c
abs = (n >= 0) ? n : -n;
```

Instrucțiuni condiționale

Exerciții propuse

- Realizati schema logică pentru implmentarea funcției abs din codul precedent.
- Realizati o implementare a funcției signum: $sgn(x) = \begin{cases} -1, & x < 0 \\ 0, & x = 0 \\ +1, & x > 0 \end{cases}$. Realizați o implementare atât în limbajul C cât și ca și schemă logică
-

Controlul fluxului

Instrucțiuni condiționale. Instrucțiunea switch-case

Instrucțiuni condiționale

Instrucțiunea switch-case

- Instrucțiune folosită pentru luarea de decizii
- Sintaxa

```
switch (expresie)
{
            case valoare1:
                        secvență_instrucțiuni_1;
                        break;
            case valoare2:
                        secvență_instrucțiuni_2;
                        break;
            default:
                        secvență_instrucțiuni_default;
}
```

```
switch (expresie)
{
            case valoare1:
            {
                        secvența_instrucțiuni_1;
                        break;
            }
            case valoare2:
            {
                        secvența_instrucțiuni_2;
                        break;
            }
            default:
            {
                        secvența_instrucțiuni_default;
            }
}
```

- Keywords: **switch, case, break, default**

Instrucțiuni condiționale

Instrucțiunea switch-case

- Funcționare instrucțiune switch-case
  - fiecare case conține o posibilă valoare a expresiei
  - la început se evaluează expresia din switch
  - dacă valoarea expresiei este egală cu valoarea unei directive **case,** execuția codului sare la acea directivă **case**
  - execuția codului continuă până la instrucțiunea **break**.
  - la instrucțiunea **break** execuția codului sare la prima instrucțiune de după corpul instrucțiunii **switch**.
  - dacă instrucțiunea **break** lipsește, execuția codului continuă chiar dacă trece și peste alte directive (instrucțiuni) case
  - dacă valoarea expresiei nu este egală cu nici o valoare a unei directive case, execuția codului sare la directiva **default** dacă aceasta există
  - dacă **default** nu există, atunci execuția codului sare la prima instrucțiune după corpul instrucțiunii **switch**.

Instrucțiuni condiționale

Instrucțiunea switch-case. Exemplu simplu de utlizare

```c
#include <stdio.h>
int main(void)
{
	int n = 0;
	printf ("Introduceti un numar:");
	scanf ("%d", &n);
	switch (n)
	{
		case 0:
		{
			printf ("numarul este 0\n");
			break;
		}
		case 1:
		{
			printf ("numarul este 1\n");
			break;
		}
		case 2:
		{
			printf ("numarul este 2\n");
			break;
		}
		default:
		{
			printf ("Ati ales nu alt numar\n");
			break;
		}
	}
	return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
	int n = 0;
	printf ("Introduceti un numar:");
	scanf ("%d", &n);
	switch (n)
	{
		case 2:
		case 4:
		case 6:
		case 8:
		{
			printf ("numarul este par\n");
			break;
		}
		case 1:
		case 3:
		case 5:
		case 7:
		case 9:
		{
			printf ("numarul este impar\n");
			break;
		}
		default:
		{
			printf ("Nu stiu ce fel de numar este\n");
			break;
		}
	}
	return 0;
}
```

Controlul fluxului

Instrucțiuni de ciclare. Instrucțiunea while

Instrucțiuni de ciclare

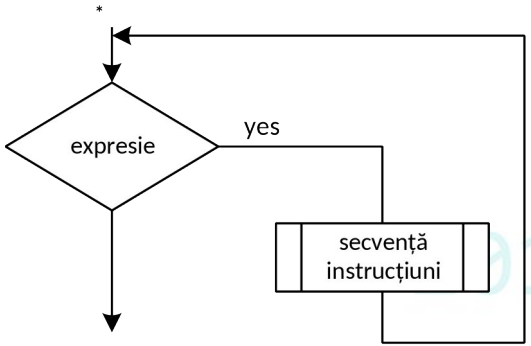
Instrucțiunea while

- Instrucțiuni de ciclare: permit realizarea de coduri ciclice, de coduri repetitive, în funcție de evaluarea unor condiții
- Instrucțiunea **while** – sintaxă

```
while (expresie)
        instrucțiune;
```

```
while (expresie)
{
        secvență_instrucțiuni;
}
```

- Schemă de funcționare

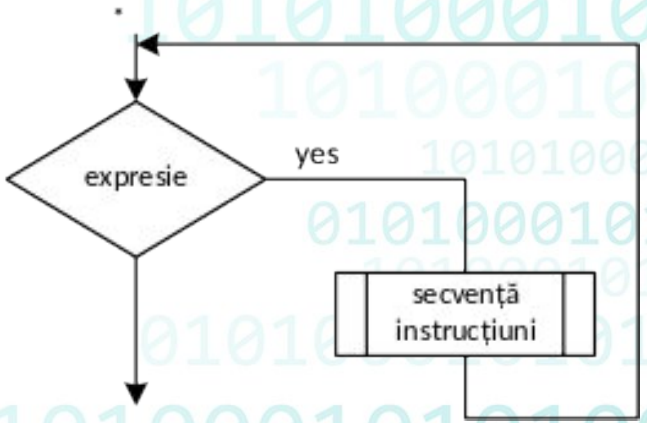


Instrucțiuni de ciclare

Instrucțiunea while

- Funcționare instrucțiune while
  - se evaluează expresia
  - dacă expresia este adevărată, se execută instrucțiunea / secvența de instrucțiuni
  - după execuția secvenței de instrucțiuni se reevaluează condiția și se reia execuția secvenței de instrucțiuni dacă condiția este adevărată
  - dacă condiția este falsă instrucțiunea while se termină și nu se mai execută secvența de instrucțiuni din corpul acesteia. Execuția sare la instrucțiunea imediat următoare corpului instrucțiunii while
- **ciclu cu test inițial** – condiția se verifică înaintea secvenței de instrucțiuni din corpul instrucțiunii de ciclare (while)
- **ciclu infinit** – atunci când expresia condițională nu se modifică
  - în corpul instrucțiunii de ciclare (while)
  - *din exterior*

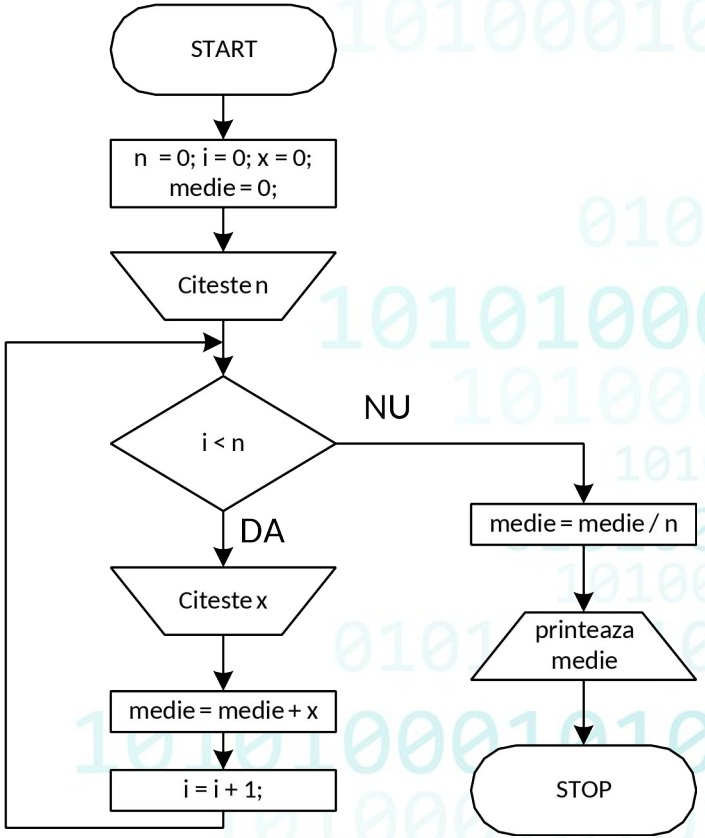
```
while (expresie)
{
            secvență_instrucțiuni;
}
```



Instrucțiuni de ciclare

Instrucțiunea while. Exemplu simplu de utlizare

```c
#include <stdio.h>

int main(void)
{
    int n = 0;
    int i = 0;
    int x = 0;
    float medie = 0;
    printf ("Dati valoarea lui n:");
    scanf("%d", &n);
    i = 0;
    while (i < n)
    {
        printf ("Dati numarul:");
        scanf("%d", &x);
        medie = medie + x;
        i++;
    }
    medie = medie / n;
    printf ("Media este %lf\n", medie);
    return 0;
}
```



Controlul fluxului

Instrucțiuni de ciclare. Instrucțiunea do-while

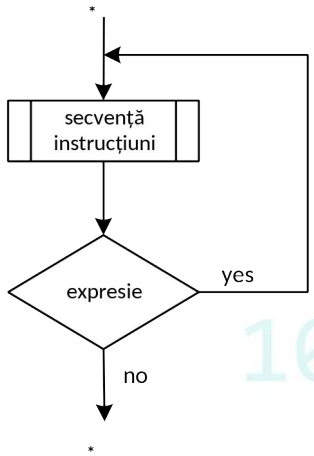
Instrucțiuni de ciclare

Instrucțiunea do-while

- Instrucțiunea **do-while** – sintaxă

```
do
{
            secvență_instrucțiuni;
}
while (expresie);
```

- Schemă funcționare

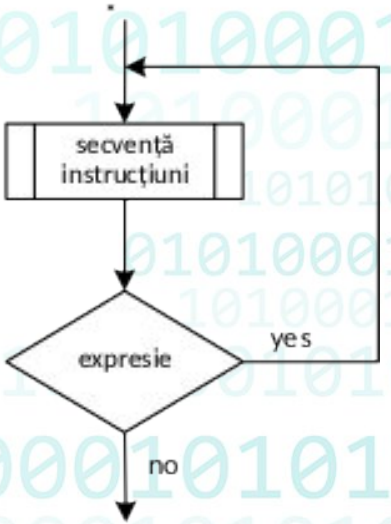


Instrucțiuni de ciclare

Instrucțiunea while

- Funcționare instrucțiune do-while
  - se execută secvența de instrucțiuni din corpul instrucțiunii do-while
  - se evaluează expresia
  - dacă expresia este adevărată, se execută din nou secvența de instrucțiuni din corpul instrucțiunii do-while
  - dacă expresia este falsă instrucțiunea do-while se termina și nu se mai reia execuția secvenței de instrucțiuni din corpul acesteia. Execuția sare la instrucțiunea imediat următoare corpului instrucțiunii do-while
- **ciclu cu test final** – condiția se verifică după execuția secvenței de instrucțiuni din corpul instrucțiunii de ciclare (do-while)
  - **Consecință**: indiferend de valoarea expresie secvența de instrucțiuni din corpul instrucțiunii do-while se execută !
- **ciclu infinit** – atunci când expresia condițională nu se modifică
  - în corpul instrucțiunii de ciclare (while)
  - *din exterior*

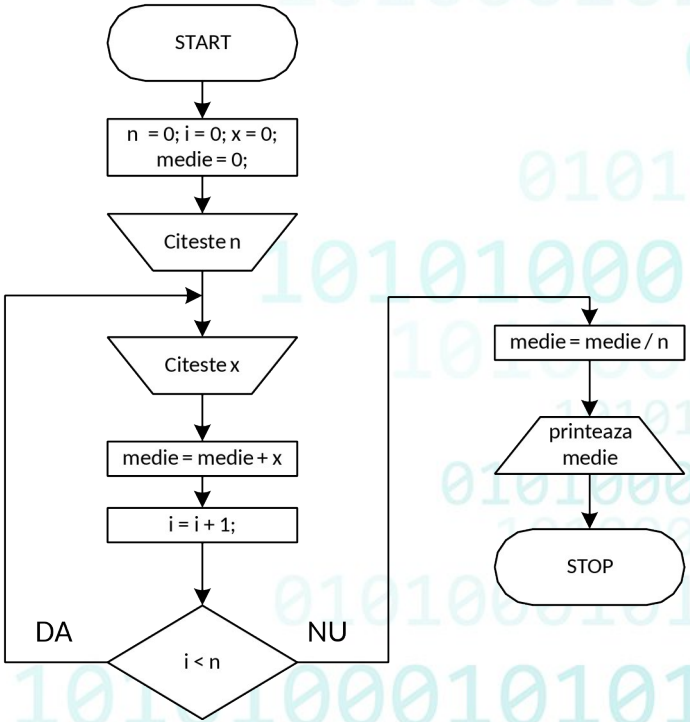
```
do
{
            secvență_instrucțiuni;
}
while (expresie);
```



Instrucțiuni de ciclare

Instrucțiunea do-while. Exemplu simplu de utlizare

```c
#include <stdio.h>

int main(void)
{
    int n = 0;
    int i = 0;
    int x = 0;
    float medie = 0;
    printf ("Dati valoarea lui n:");
    scanf("%d", &n);
    i = 0;
    do
    {
        printf ("Dati numarul:");
        scanf("%d", &x);
        medie = medie + x;
        i++;
    }
    while (i < n);
    medie = medie / n;
    printf ("Media este %lf\n", medie);
    return 0;
}
```



Controlul fluxului

Instrucțiuni de ciclare. Instrucțiunea for

Instrucțiuni de ciclare

Instrucțiunea for

- Instrucțiunea **for** – sintaxă

```
for (expr1; expr2; expr3)
        instrucțiune;
```

```
for (expr1; expr2; expr3)
{
        secvență_instrucțiuni;
}
```

- unde
  - $expr_1$ – reprezintă instrucțiunea ce se va executa înainte de începerea ciclului, o singură dată (de obice se folosește o instrucțiune de atribuire), poate fi chiar si o declarație de variabilă cu inițializare (sau fără)
  - $expr_2$ – reprezintă o instrucțiune condițională ce este testată ca și condiție de intrare în iterație (condiție de continuare a ciclului)
  - $expr_3$ – reprezintă o instrucțiune ce se va executa la sfarșitul fiecărei iterații a ciclului, imediat după instrucțiunea / secvența de instrucțiuni din corpul instrucțiunii for

Instrucțiuni de ciclare

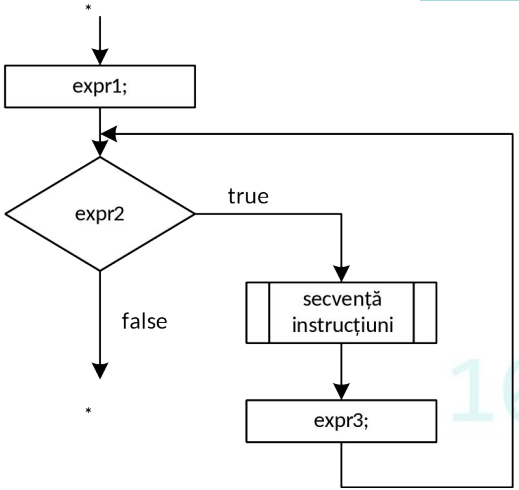
Instrucțiunea for

- Instrucțiunea **for** – echivalență cu instrucțiunea **while**

```
for (expr1; expr2; expr3)
{
        secvență_instrucțiuni;
}
```

```
expr1;
while (expr2)
{
        secventa_instructiuni;
        expr3;
}
```

- Schemă funcționare

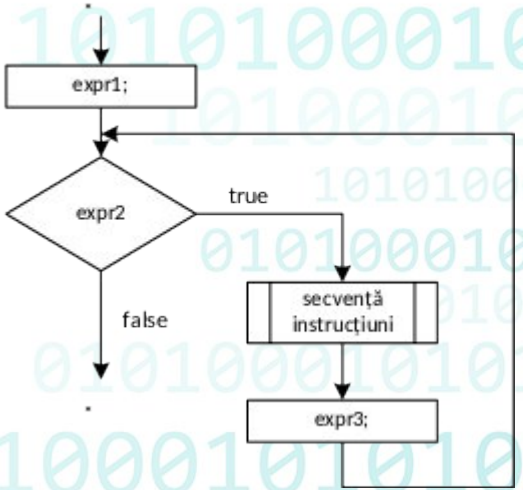


Instrucțiuni de ciclare

Instrucțiunea for

- Funcționare instrucțiune for
  - se execută (o singură dată la început) instrucțiunea reprezentată de expr1 (de obicei sunt instrucțiuni de atribuire, declarație-inițializare)
  - se evaluează expresia expr2
  - dacă expresia expr2 este adevărată, se execută secvența de instrucțiuni din corpul instrucțiunii for
  - la sfârșitul execuției secvenței de instrucțiuni din corpul instrucțiunii for, deci la sfârșitul fiecărei iterații, se execută instrucțiunea reprezentată de expr3
  - se reia ciclul prin reevaluarea expresiei expr2
  - dacă expresia expr2 este falsă instrucțiunea for se încheie și nu se mai execută nici o instrucțiune din corpul instrucțiunii for, nu se execută nici expr3 (practic nu se mai reia o nouă iterație)
  - se continuă execuția codului cu prima instrucțiune de după corpul instrucțiunii for
- **ciclu cu test inițial**

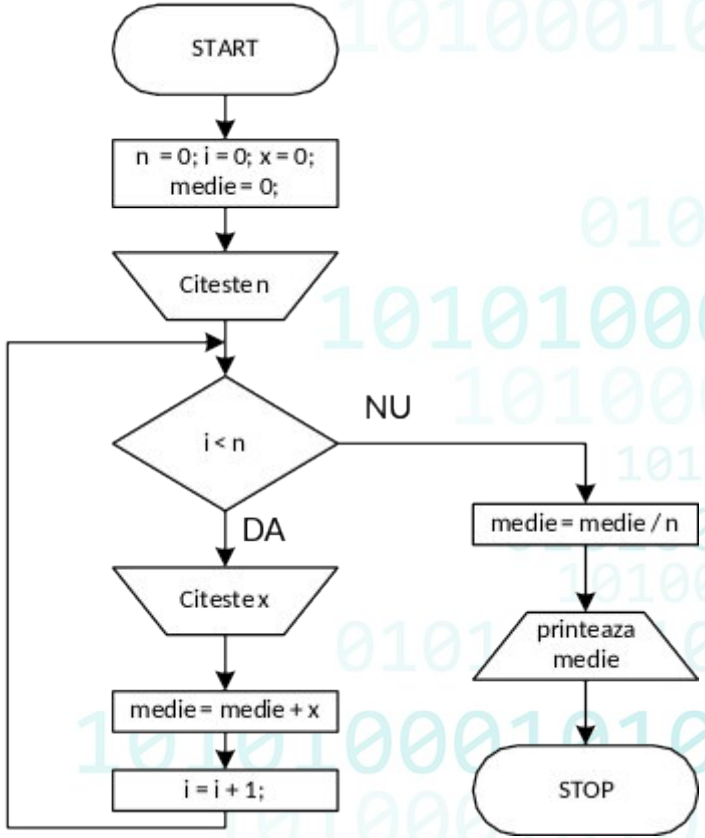
```
for (expr1; expr2; expr3)
{
            secvență_instrucțiuni;
}
```



Instrucțiuni de ciclare

Instrucțiunea for. Exemplu simplu de utlizare

```c
#include <stdio.h>

int main(void)
{
    int n = 0;
    int i = 0;
    int x = 0;
    float medie = 0;
    printf ("Dati valoarea lui n:");
    scanf("%d", &n);
    for (i = 0; i < n; i++)
    {
        printf ("Dati numarul:");
        scanf("%d", &x);
        medie = medie + x;
    }
    medie = medie / n;
    printf ("Media este %lf\n", medie);
    return 0;
}
```



Controlul fluxului

Instrucțiuni de ciclare. Instrucțiunile continue si break

Instrucțiuni de ciclare

Instrucțiunea continue

- Instrucțiunea **continue** are sens doar într-un ciclu (while, do-while, for) și are ca scop oprirea execuției iterației curente și reluarea unei noi iterații
- codul ce urmează după instrucțiunea continue nu se va executa niciodată
- se poate folosi în orice combinație cu alte instrucțiuni
- exemplu: tipărirea numerelor pare din intervalul 0 – 10

```
#include <stdio.h>
int main(void)
{
    int i = 0;
    for (i = 0; i < 10; i++)
    {
        if ((i % 2) == 0)
        {
            continue;
        }
        printf ("%d\n", i);
    }
    return 0;
}
```

Instrucțiuni de ciclare

Instrucțiunea break

- Instrucțiunea **break** are sens în două situații
  - utilizată împreuna cu instrucțiunile switch-case – are rolul de a ieși din corpul instrucțiunii switch
  - utilizată într-un ciclu (while, do-while, for) are scopul de a termina de tot ciclul fără a se relua vreo iterație. În momentul în care instrucțiunea break este întâlnită ciclul se oprește, înstrucțiunile ce urmează după break din corpul ciclului nu se mai execută iar următoarea instrucțiune executată este prima instrucțiune de după corpul instrucțiunii ce implementează ciclul
- exemplu: ieșirea dintr-un ciclu infinit

```c
#include <stdio.h>

int main(void)
{
    int n = 10;
    int i = 0;
    while (1)
    {
        printf ("%d\n", i);
        i++;
        if (i == n)
        {
            break;
        }
        printf ("Sfarsit iteratie\n");
    }
    return 0;
}
```

Controlul fluxului

Instrucțiuni de ciclare. Observații și consecințe

Instrucțiuni de ciclare

Observații și consecințe

- **ciclu cu test inițial** – condiția se verifică înaintea secvenței de instrucțiuni din corpul instrucțiunii de ciclare (while)
- **ciclu cu test final** – condiția se verifică după execuția secvenței de instrucțiuni din corpul instrucțiunii de ciclare (do-while)
  - **Consecință**: indiferend de valoarea expresie secvența de instrucțiuni din corpul instrucțiunii do-while se execută !
- **ciclu infinit** – atunci când expresia condițională nu se modifică
  - în corpul instrucțiunii de ciclare (while)
  - *din exterior*
- implementare ciclu infinit

```
for (;;)
{
    secvență_instrucțiuni;
}
```

```
while (1)
{
    secvență_instrucțiuni;
}
```

```
do
{
    secvență_instrucțiuni;
}
while (1);
```

Instrucțiuni de ciclare

Observații și consecințe

- **Variabilă declarată în interiorul unui ciclu**

```c
#include <stdio.h>

int main(void)
{
    int i = 0;
    for (i = 0; i < 10; i++)
    {
        int n = 0;
        n++;
        printf ("%d\n", n);
    }
}
```

```c
#include <stdio.h>

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

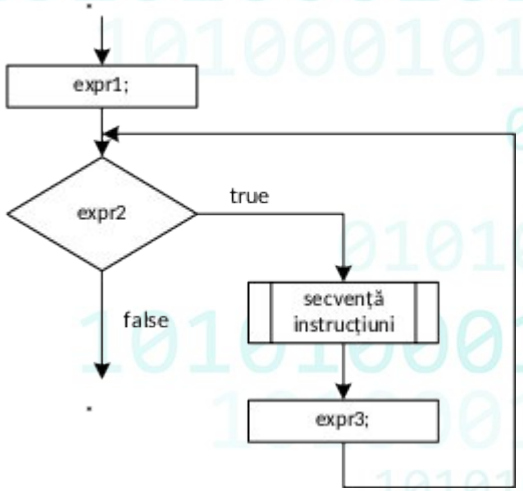
- variabila este vizibilă doar în corpul instrucțiunii de ciclare (de fapt o consecință a faptului ca o variabilă este vizibilă doar în blocul de instrucțiuni în care este declarată)
- variabila se declară, se definește și implicit i se alocă memorie doar o dată și nu la fiecare iterație
- dacă se specifică inițializare atunci variabila este inițializată la fiecare iterație a ciclului
- durata de viață a variabilei este pe toată durata execuției ciclului

Instrucțiuni de ciclare

Observații și consecințe

- **Variabilă declarată în interiorul unui ciclu for în cadrul $expr_1$**

```c
#include <stdio.h>

int main(void)
{
    for (int i = 0; i < 10; i++)
    {
        printf ("%d\n", i);
    }
}
```



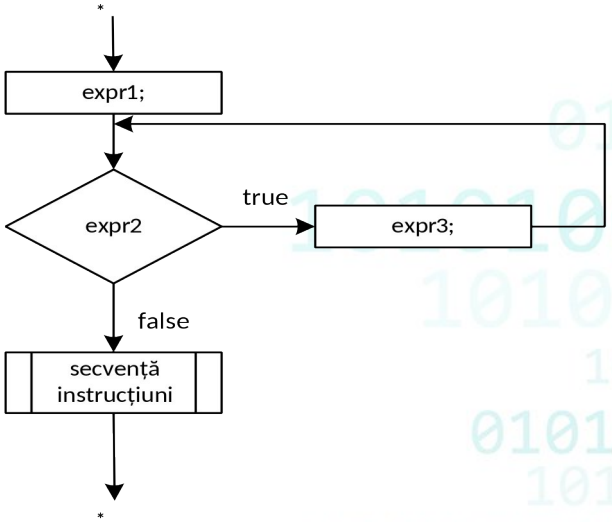
- variabila este vizibilă doar în corpul instrucțiunii for, ca și cum ar fi fost declarată în corpul acesteia (o consecință a faptului ca o variabilă este vizibilă doar în blocul de instrucțiuni în care este declarată)
- variabila se declară, se definește și implicit i se alocă memorie doar o dată și nu la fiecare iterație
- dacă se specifică inițializare atunci variabila este inițializată doar o singură dată la început
- dacă nu se specifică inițializare atunci variabila rămâne neinițializată și prezintă un risc major
- durata de viață a variabilei este pe toată durata execuției ciclului
- variabila nu este vizibilă inainte și după corpul instrucțiunii for (consecință a primului punct)
- este posibil să nu fie suportată de orice compilator. Nu este suportată această facilitate in ANSI C

Instrucțiuni de ciclare

Observații și consecințe

- **Efectul adăugării simbolului punct-și-virgulă imediat după linia ce declară ciclul**

```c
#include <stdio.h>

int main(void)
{
    int i = 0;
    for (i = 0; i < 10; i++);
    {
        printf ("%d\n", i);
    }
}
```



- linia cu instrucțiunea for se va transforma într-o instrucțiune simplă de sine stătătoare și se va separa de blocul de cod ce urmează
- astfel blocul de cod ce urmează liniei cu for se va executa doar o singură dată în afara ciclului for

Secțiunea V

Funcții
Zonele de memorie ale unui program

Funcții

Definiții. Exemple.

Funcții

Definiții

- **Funcția** – este o grupare de instrucțiuni si variabile menite să implementeze o anumită sarcină
  - este absolut necesară pentru a împărți un program în mai multe funcționalități de bază elementare
  - rezolvă problema duplicării de cod
  - orice program C are minim o funcție, funcția main()
  - numele funcției reprezintă practic o adresă în memorie de unde începe codul (corpul) acesteia
- Declararea unei funcții. Sintaxa:

```
tip-rezultat nume-funcție(declarații de parametri)
{
     declaratii variabile;
     lista_instructiuni
     return valoare;
}
```

```
tip-rezultat nume-funcție(tip param1, tip param2)
{
     declaratii variabile;
     lista_instructiuni
     return valoare;
}
```

- *tip-rezultat* – semnifică tipul de date al valorii returnate – poate fi și tipul vid – **void** – în acest caz poate să lipsească instrucțiunea return de la sfârșitul funcției
- declarații de parametri – se declară similar variabilelor, într-o listă de declarații separați prin virgulă (,). Se pot declara oricâți parametri de orice tip de date disponibil. Poate fi și o listă nulă (goala – ( ) sau desemnată nulă prin cuvântul **void**)
- orice funcție se termină cu instrucțiunea return urmată de o valoare (sau referențiată printr-o variabilă) de același tip de date ca și cel specificat în tip-rezultat
- instrucțiunea de return este implicită și poate lipsi când tip-rezultat este void
- dacă tip-rezultat nu este void și instrucțiunea return lipsește compilatorul va genera warning
- Keywords: **void**, **return**

Funcții

Definiții

- **Exemplu declarație și definire funcție:**

```
int adunare(int a, int b)
{
    int rezultat;
    rezultat = a + b;
    return rezultat;
}
```

antetul funcției: conține tipuri valorii returnate, numele funcției, lista de parametri

corpul funcției

- numele funcției – aceleași reguli ca și în cazul numelor de variabile – poate conține caractere alfanumerice și caracter '_' , nu poate începe cu o cifră
- corpul unei funcții este format dintr-un bloc de cod între paranteze acolade (braces)
- varibilele din interiorul funcției se numesc variabile locale
- ca să fie **definită** complet o funcție trebuie sa aibă specificat atât antetul cât și corpul funcției
- **declarația** unei funcții – conține doar antetul, fără corp și urmat de semnul punct-și-virgulă
- Apelul unei funcții. Sintaxă:

```
nume_functie(lista_argumente);
nume_functie();
```

Funcții

Exemplu de apel a unei funcții definite de utilizator

- valoarea returnată se poate folosi în expresii sau se poate ignora
- argumentele ce se dau parametrilor funcțiilor sunt expresii și acestea vor fi evaluate înainte de apelul funcției
- funcția în sine este de fapt un caz particular de expresie și poate fi folosită, prin valoarea returnată, ca și operand

```
int adunare(int a, int b)
{
    int rezultat;
    rezultat = a + b;
    return rezultat;
}

int main(void)
{
    int n1 = 10;
    int n2 = 3;
    int r = 0;
    r = adunare(n1, n2);
    r = adunare(7, 9);
    r = adunare(7, n2);
    if (adunare(n1, n2) > 20)
    {
        // o secventa de cod
    }
    r = adunare( adunare(n1, 2), 10);
    adunare(1,2);
    return 0;
}
```

Funcții

Alte exemple de funcții

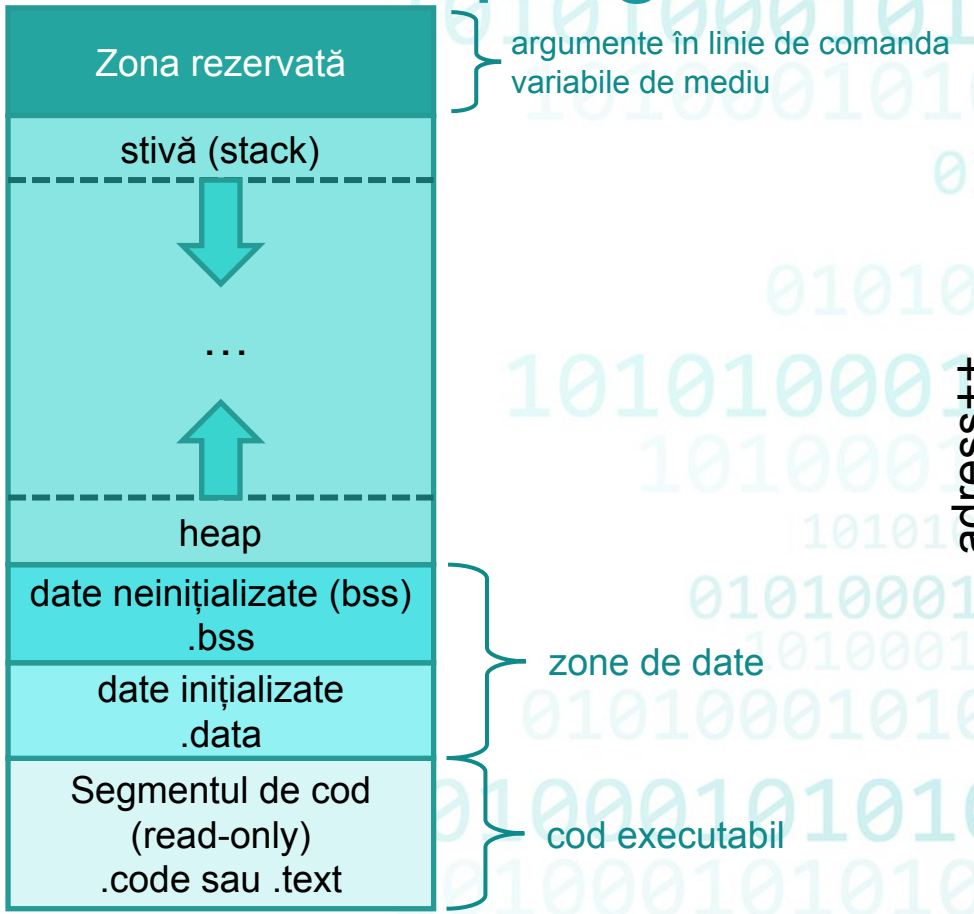
- o funcție cu tipul de return void nu va returna o valoare, deci, nu se poate folosi în expresii (linia 3 din main)
- funcțiile functie2 și functie3 nu au parametri, deci se poate folosi cuvântul void pentru a specifica o lista nulă de parametri

```
#include <stdio.h>
void functie1(int a, int b)
{
	printf ("abc");
}

int functie2(void)
{
	return 1;
}

void functie3(void)
{
	printf ("abc");
}

int main(void)
{
	int a = 0;
	functie1(); // incorect, nu va compila, lipsesc argumentele
	a = functie1(1,2); // incorect, tip void in expresie
	a = functie2();
	functie3();
	return 0;
}
```

Zone de memorie ale unui program

Zone de memorie ale unui program C

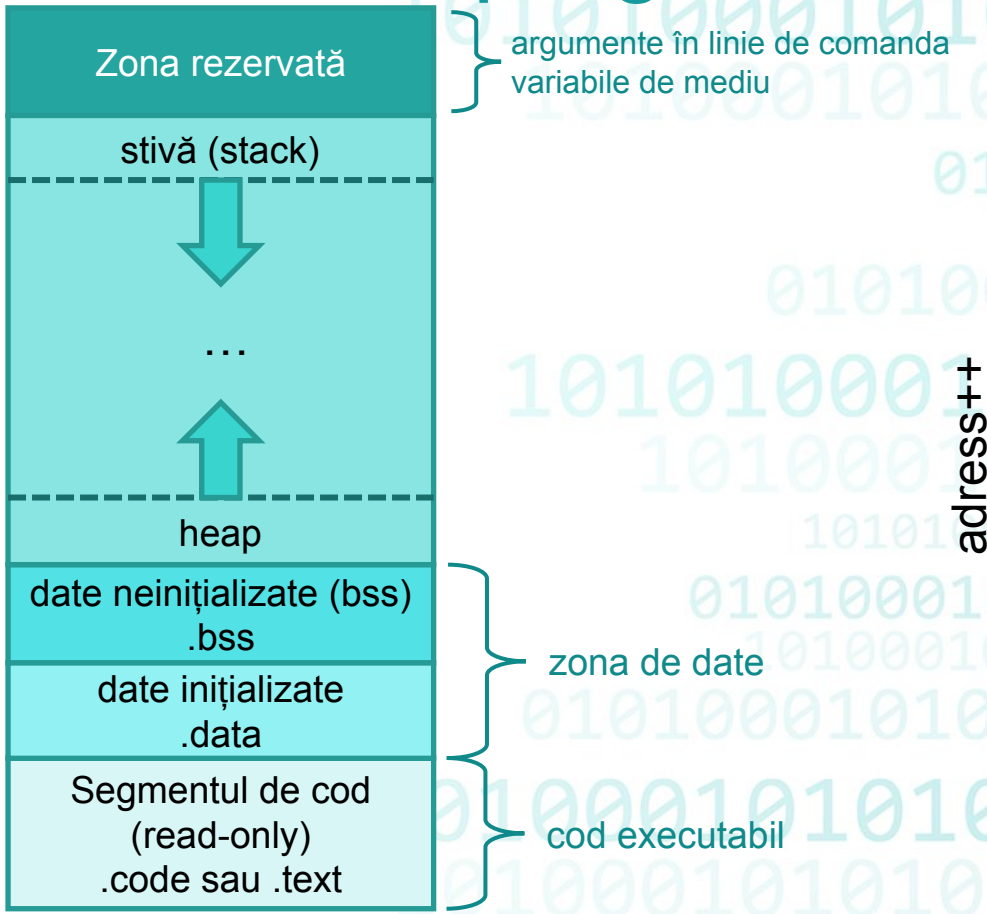
- segmentul de cod (.text)
  - conține codul executabil al programului precum și constantele
  - are dimensiune fixă, egală cu dimensiunea codului executabil
  - este read-only (protejat la scriere) din punct de vedere al programului (programul nu își poate singur rescrie codul)
- **segmentul de date (.data și .bss)**
- segmentul .data
  - conține date statice, declarate global sau cu modificatorul static în funcție, ce sunt inițializate de programator în cod
  - este în politică read-write dar structura ei nu poate fi modificată în timpul execuției programului (la runtime)
- segmentul .bss (block started by symbol)
  - conține date statice, declarate global sau cu modificatorul static în funcție, ce NU sunt inițializate de programator în cod
  - această zonă este inițializată de sistemul de operare cu 0 chiar înainte de lansarea în execuție a programului ☺ dar NU E O REGULĂ
  - este în politică read-write dar structura ei nu poate fi modificată la runtime



- Schema adaptare după: Cristina Stângaciu, curs “Limbaje de Programare”, ETC, 2020

Zone de memorie ale unui program C

- segmentul de heap
  - conține blocuri de memorie alocate dinamic
  - este gestionată de funcțiile standard de bibliotecă ce se ocupă cu alocarea dinamică
  - este limitată doar de capacitățile hardware (memorie) ale sistemului pe care rulează programul (target)
- segmentul de stivă
  - zonă de memorie specială adresată în politică LIFO (Last In First Out)
  - este folosită pentru apelurile de funcții și conține: variabilele locale ale unei funcții, parametri unei funcții, valoarea de return a unei funcții, adresa de return a funcției
  - este limitată de sistemul de operare și este în general destul de mică

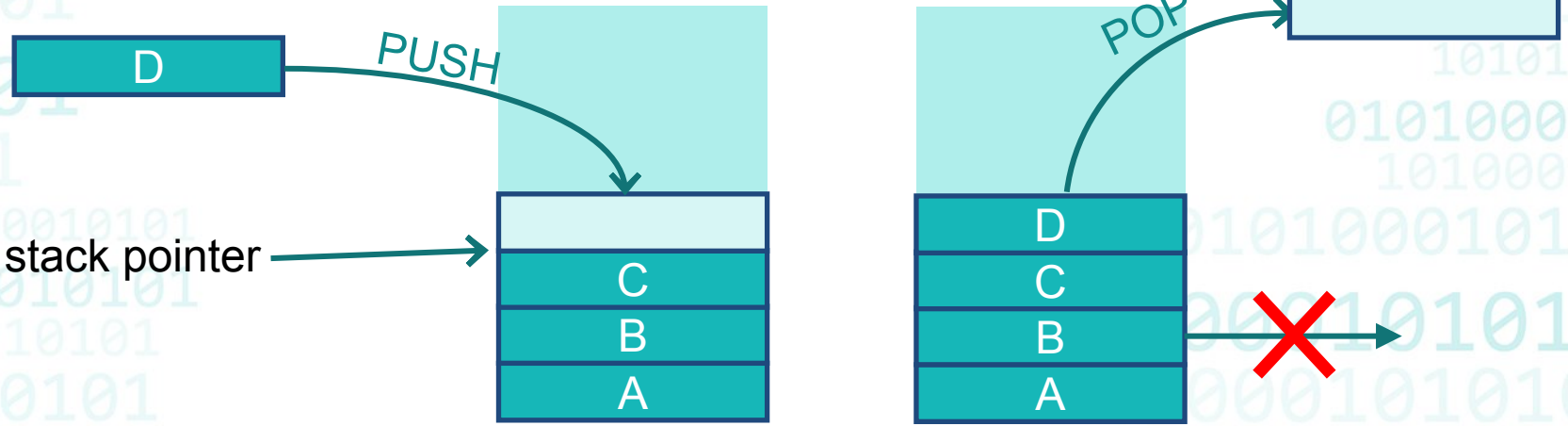


- Schema adaptare după: Cristina Stângaciu, curs “Limbaje de Programare”, ETC, 2020

Zone de memorie ale unui program C

Stiva (stack)

- Stiva
  - o structură de date adresabilă în politică LIFO (Last In First Out)
  - se poate adresa folosind 2 operații:
    - PUSH – prin care se adaugă deasupra stivei un nou element
    - POP – prin care se scoate de deasupra stivei un element
  - nu se poate modifica/elimina un element din “mijlocul” stivei



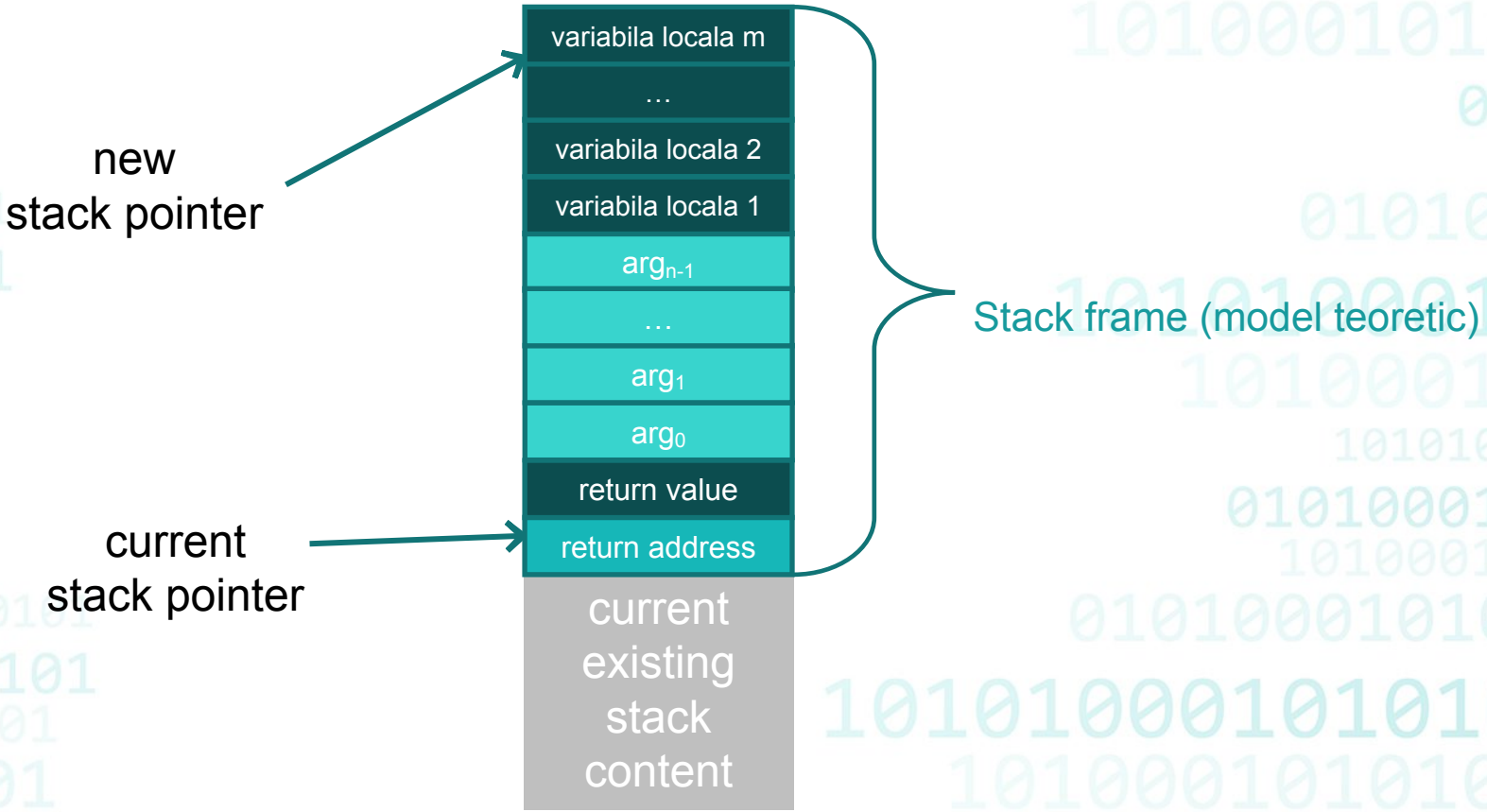
Zone de memorie ale unui program C

Ce se întâmplă și cum se face un apel de funcție ?

- Apelurile de funcții se fac folosind ca zonă de memorie exclusiv stiva – inexistența stivei pe un sistem duce la imposibilitatea realizării apelurilor de funcție
- La începutul apelului unei funcții se adaugă pe stivă un “stack frame”
- La sfârșitul apelului funcției se scoate de pe stivă stack frame-ul adăugat la începutul apelului
- După apel stiva rămâne la aceeași structură și valori ca și înainte de apel
- Stack frame conține
  - adresa de return a funcției – adresa instrucțiunii din cod imediat următoare după apelul de funcției. Este necesară pentru a se putea întoarce execuția după terminarea apelului funcției
  - valorile argumentelor, valorile parametrilor funcției
  - variabilele locale ale unei funcții
- La un apel de funcție **se copiază** valorile argumentelor de la apel în locațiile din stivă ce reprezintă parametri locali ai funcției – funcția “vede” doar niște valori și NU are acces la variabilele inițiale (dacă sunt… căci pot fi doar valori constante)
  - În C parametri unei funcții se transmit doar prin valoare

Zone de memorie ale unui program C

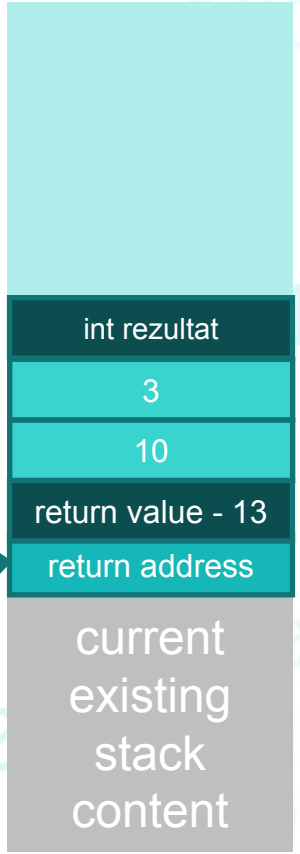
Ce se întâmplă și cum se face un apel de funcție ?



Zone de memorie ale unui program C

Apel de funcție. Exemplu grafic pe stivă

```c
int adunare(int a, int b)
{
    int rezultat = 0;
    rezultat = a + b;
    return rezultat;
}

int main(void)
{
    int n1 = 10;
    int n2 = 3;
    int r = 0;
    r = adunare(n1, n2);
    printf("r = %d\n", r);
    return 0;
}
```



Zone de memorie ale unui program C

Apel de funcție. Consecințe

```c
#include <stdio.h>

void MyFunction(int a)
{
  printf ("Valoarea lui a in functie inainte de modificare: %d\n", a);
  a = 17;
  printf ("Valoarea lui a in functie dupa modificare %d\n", a);
  printf ("MyFunction done\n");
}

int main(void)
{
  int x = 4;
  printf ("Valoarea lui x in main inainte de apel functie %d\n", x);
  MyFunction(x);
  printf ("Valoarea lui x in main dupa apel functie %d\n", x);
  printf ("Sfarsit program\n");
  return 0;
}
```

```
valy@staff:~/teaching$ ./p
Valoarea lui x in main inainte de apel functie 4
Valoarea lui a in functie inainte de modificare: 4
Valoarea lui a in functie dupa modificare 17
MyFunction done
Valoarea lui x in main dupa apel functie 4
Sfarsit program
valy@staff:~/teaching$
```

Zone de memorie ale unui program C

!!!! Apel de funcție. Consecințe !!!!

- La un apel de funcție **se copiază** valorile argumentelor de la apel în locațiile din stivă ce reprezintă parametri locali ai funcției – funcția “vede” doar niște valori și NU are acces la variabilele inițiale
(dacă sunt… căci pot fi doar valori constante)

În C parametri unei funcții se transmit doar prin valoare

Zone de memorie ale unui program C

Modificatorul static

- Modificatorul **static** pus în fața declarației unei variabile locale ale unei funcții, are rolul de a muta variabila locală de pe stivă în zona de date
  - va fi mutată în .data dacă este inițializată
  - va fi mutată în .bss dacă nu este inițializată
- consecință
  - se modifică durata de viață a variabilei – de la o durată de viață egală cu durata apelului de funcție la o durată de viață egală cu execuția întregului program
  - practic variabila rămâne și își păstrează valoarea de la un apel la altul al aceleași funcții
  - variabila se comporta ca și o variabilă globală dar rămâne vizibilă doar la nivelul funcției (blocului funcției)

```c
#include <stdio.h>

void functie(void)
{
  int n = 0;
  n++;
  printf ("n = %d\n", n);
}

int main(void)
{
  functie();
  functie();
  functie();
  return 0;
}
```

```
valy@staff:~/teaching$ ./p
n = 1
n = 1
n = 1
```

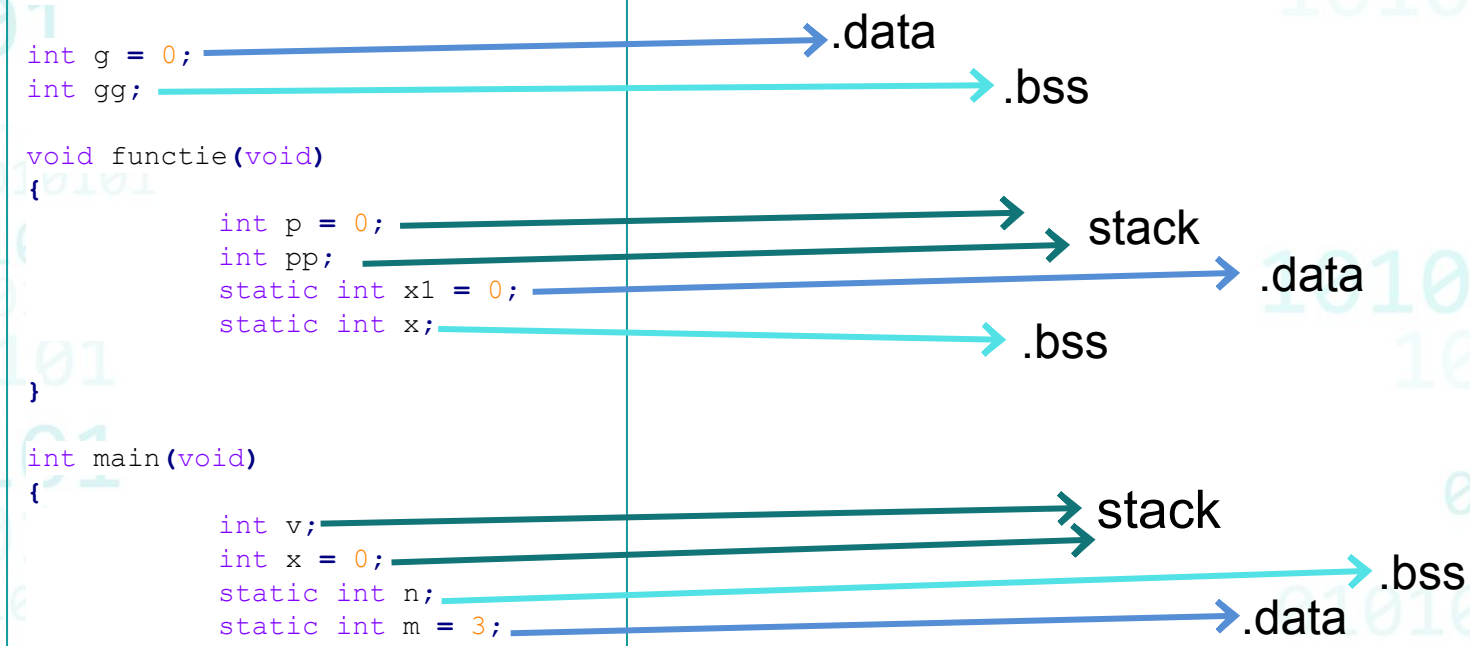
```c
#include <stdio.h>

void functie(void)
{
  static int n = 0;
  n++;
  printf ("n = %d\n", n);
}

int main(void)
{
  functie();
  functie();
  functie();
  return 0;
}
```

```
valy@staff:~/teaching$ ./p
n = 1
n = 2
n = 3
```

Zone de memorie ale unui program C

Exemplu grafic

```c
#include <stdio.h>

int g = 0;
int gg;

void functie(void)
{
        int p = 0;
        int pp;
        static int x1 = 0;
        static int x;

}

int main(void)
{
        int v;
        int x = 0;
        static int n;
        static int m = 3;
        functie();
        functie();
        functie();
        return 0;
}
```



Zone de memorie ale unui program C

Noțiuni practice zone de memorie

- În sistemele Linux dimensiunea stivei se poate afla astfel:

```
valy@staff:~/teaching$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 31865
max locked memory       (kbytes, -l) 65536
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 31865
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

- In sistemele Linux, dimensiunea segmentelor de memorie unui program:

```
valy@staff:~/teaching$ size p
   text    data     bss     dec     hex filename
   1543     584       8    2135     857 p
```

Secțiunea VI

Tipul de date caracter
Operații de intrare-iesire

Tipul de date caracter

Tipul de date caracter

Definiții și interpretare

Modificator semn	Tip standard	Dimensiune (bytes)	Dimensiune minima (bytes)	Interval de valori
signed	char	**1**	1	$-2^7 \dots 2^7-1$ $-128 \dots 127$
unsigned		**1**	1	$0 \dots 2^8-1$ $0 \dots 255$

- Tipul de date **char** – este de fapt un întreg pe dimensiune de 1 byte
- Convenție:
  - se consideră tipul de date **unsigned char** – cu intervalul e valori 0…255
  - se asignează câte un caracter fiecărei valori din intervalul 0..255 - **tabela ASCII**
- ASCII – American Standard Code for Information Interchange
- tabela ASCII – o tabelă standardizată ce conține o asignare de valori între 0…255 pentru caracterele de bază pe dimensiune de 1 byte
  - https://www.asciitable.com/
  - în pagina de manual de Linux: man ascii

Tipul de date caracter

Tabela ASCII standard

Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH	(start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX	(start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX	(end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT	(end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ	(enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK	(acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL	(bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS	(backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB	(horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT	(vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR	(carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO	(shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI	(shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE	(data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1	(device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2	(device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3	(device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4	(device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK	(negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB	(end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN	(cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM	(end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB	(substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC	(escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS	(file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS	(group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS	(record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US	(unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

https://www.asciitable.com/

Tipul de date caracter

Tabela ASCII extinsă

128	Ç	144	É	160	á	176	▒	192	└	208	╨	224	α	240	≡
129	ü	145	æ	161	í	177	▒	193	┴	209	╤	225	ß	241	±
130	é	146	Æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥
131	â	147	ô	163	ú	179	│	195	├	211	╙	227	π	243	≤
132	ä	148	ö	164	ñ	180	┤	196	─	212	╘	228	Σ	244	⌠
133	à	149	ò	165	Ñ	181	╡	197	┼	213	╒	229	σ	245	⌡
134	å	150	û	166	ª	182	╢	198	╞	214	╓	230	µ	246	÷
135	ç	151	ù	167	º	183	╖	199	╟	215	╫	231	τ	247	≈
136	ê	152	ÿ	168	¿	184	╕	200	╚	216	╪	232	Φ	248	°
137	ë	153	Ö	169	⌐	185	╣	201	╔	217	┘	233	Θ	249	∙
138	è	154	Ü	170	¬	186	║	202	╩	218	┌	234	Ω	250	·
139	ï	155	¢	171	½	187	╗	203	╦	219	█	235	δ	251	√
140	î	156	£	172	¼	188	╝	204	╠	220	▄	236	∞	252	ⁿ
141	ì	157	¥	173	¡	189	╜	205	═	221	▌	237	φ	253	²
142	Ä	158	₧	174	«	190	╛	206	╬	222	▐	238	ε	254	■
143	Å	159	ƒ	175	»	191	┐	207	╧	223	▀	239	∩	255	

Source: www.LookupTables.com

Tipul de date caracter

Tabela ASCII

- Tabela ASCII standard (de bază)
  - conține caractere mapate pe intervalul 0…127
  - intervalul 0…31 (0x00 – 0x1F) – caractere neprintabile ce au o semnificație specială în anumite situații și domenii
  - intervalul 32…47 (0x20 – 0x2F) – caractere printabile speciale, semne de punctuație, semne aritmetice… etc
  - intervalul 48…57 (0x30 – 0x39) – caractere printabile ce reprezintă cifrele (digits) în baza 10
  - intervalul 58…64 (0x3A – 0x40) – caractere printabile ce reprezintă semne speciale, punctuație, relaționale, @
  - intervalul 65…90 (0x41 – 0x5A) – caractere printabile ce reprezintă litere mari (majuscule) [A-Z]
  - intervalul 91…96 (0x5B – 0x60) - caractere printabile ce reprezintă semne speciale, punctuație, relaționale, matematice
  - intervalul 97…122 (0x61 – 0x7A) – caractere printabile ce reprezintă litere mici [a-z]
  - intervalul 123…126 (0x7B – 0x7E) – caractere printabile ce reprezintă semne speciale, punctuație, relaționale, etc
  - numărul 127 (0x7F) – caracterul neprintabil DEL
- Tabela ASCII extinsă
  - conține caractere speciale printabile ce sunt folosite relativ rar
  - de obicei sunt folosite în a desena interfețe grafice cu utilizatorul în mod text

Tipul de date caracter

Asignare de caracter. Constanta de tip caracter

- Constanta de tip caracter este reprezentată de caracterul în sine scris între apostroafe simple
- Sintaxă exemplu: `'a'`
- Exemplu asignare constante caracter:

```
char ch1;
char ch2;
char ch3;
int n1;

ch1 = 'a';
ch2 = 0x61;
ch3 = 97;

n1 = 'a';
n1 = 0x61;
n1 = 97;
```

- cele 3 variabile de tip char (ch1, ch2, ch3) au practic aceeași valoare și reprezintă același lucru, adica, caracterul ‘a’.
  Totul ține de interpretare
- variabila n1 are și ea aceeași valoare ca și ch1,ch2,ch3 adică caracterul ‘a’ – are doar un domeniu de valori mai mare dar tot poate fi folosită și interpretată ca și caracter

Tipul de date caracter

Constanta de tip șir de caractere (string)

- Constanta de tip șir de caractere este reprezentată de șirul de caractere în sine scris între ghilimele
- Sintaxă exemplu:

```
”Acesta este un sir de caractere”
```

- ATENTIE: a nu se confunda o constantă de tip șir de caractere ce conține un singur caracter cu o constantă de tip caracter
  - **“a”** nu este același lucru cu **‘a’**

Tipul de date caracter

Funcții standard de clasificare a caracterelor din biblioteca ctype.h

```
int isalnum(int c)   (isalpha(c) || isdigit(c))
int isalpha(int c)   ('A' <= c && c <= 'Z' || 'a' <= c && c <= 'z')
int isblank(int c)   (c == ' ' || c == '\t')
int iscntrl(int c)   // caracter de control, valoare: 0 - 31
int isdigit(int c)   ('0' <= c && c <= '9')
int isgraph(int c)   // caracter tiparibil, exceptand spatiu
int islower(int c)   ('a' <= c && c <= 'z')
int isprint(int c)   // caracter tiparibil, inclusiv spatiu
int ispunct(int c)   // tiparibil şi nu alnum şi nu spatiu
int isspace(int c)   (c == ' ' || c == '\t' || c == '\n' ||
/* "spatii albe" */ c == '\v' || c == '\f' || c == '\r')
int isupper(int c)   ('A' <= c && c <= 'Z')
int isxdigit(int c) // cifr˘a hexazecimal˘a: 0-9, A-F, a-f
int tolower(int c)   // A - Z -> a - z, restul neschimbat
int toupper(int c)   // a - z -> A - Z, restul neschimbat
```

- Funcțiile se găsesc declarate în header-ul ctype.h
- Se poate obține pagina de manual în linux folosind ca și argument pentru max oricare din numele acestor funcții există o singură pagină de manual pentru toate. Exemplu

```
man isblank
```

- Adaptare după: Marius Minea, curs “Programarea calculatoarelor 2”, AC, 2005

Tipul de date caracter

Notații

- Câteva notații
  - notație ‘\n’ – reprezintă o notație pentru **newline (rand nou)** – este în funcție de implementare ori o succesiune a caracterelor din tabela ASCII: caracterul 10 = 0x0A (LF – Line Feed) și caracterul 13 = 0x0D (CR – Carridge Return) sau doar a caracterului 10 = 0x0A (LF – Line Feed)
  - notația ‘\t’ – reprezintă caracterul **tab** – obținut prin apăsarea tastei tab – implementat prin caracterul 9 (TAB) din ASCII
  - notația ‘\0’ – reprezintă caracterul **NUL** (null) – implementat prin caracterul 0 din ASCII
  - notația ‘\’’ – reprezintă caracterul apostrof
  - notația ‘\\’ – reprezintă caracterul \ backslash

Operații elementare de intrare-ieșire

Operații elementare de I/O

Descriptori (fișiere) standard de intrare ieșire

- Fiecare program are acces implicit la 3 “fișiere” pentru operațiuni de intare ieșire (scriere/citire de la tastatura/ecran)
  - **stdin** – **standard input** – intrarea standard a programului – toate funcțiile standard de intrare vor citi implicit de la intrarea standard (se poate spune aproximativ că vor citi, generic, de la tastatură)
  - **stdout** – **standard output** – ieșirea standard a programului – toate funcțiile standard de ieșire vor scrie implicit la ieșirea standard (se poate spune aproximativ ca vor scrie pe ecran, ca vor tipări)
  - **stderr** – **standard error** – ieșirea standard de eroare a programului – similar cu stdout dar se folosește pentru a “tipări” erori
- putem afirma temporar
  - stdout = ecran, terminal
  - stdin = tastatura
- fișierele stdin, stdout, stderr sunt deschise implicit la pornirea programului. O data închise acestea nu mai pot fi deschise pe parcursul execuției programului
- funcția putchar() – funcție elementară de scriere la stdout – “de printare pe ecran”
- funcția getchar() – funcție elementară de citire de la stdin – “de citire de la tastatură”

Operații elementare de I/O

Funcțiile standard putchar si getchar

- Funcția putchar()
  - man page: *man putchar*
  - declarație/antet/synopsis: `int putchar(int c);`
  - are rolul de a scrie caracterul dat ca și argument (prin parametrul c) la ieșirea standard stdout (echiv. cu a “a printa” / “a tipări” un caracter)
  - returnează caracterul scris la stdout (după ce a fost scris) convertit la un int sau EOF (End Of File) în caz de eroare
- Funcția getchar()
  - man page: *man getchar*
  - antet: `int getchar(void);`
  - are rolul de a citi un caracter de la stdin (echiv cu a citi de la tastatură)
  - returneaza caracterul citit de la stdin convertit la un int sau EOF (End Of File) în caz de eroare
  - funcția este cu blocare, programul se blochează până la citirea unui caracter sau până la citirea lui EOF
- Caracterul special EOF (End Of File)
  - semnifică sfârșitul fișierului stdin sau stdout. După apariția lui EOF (valoare -1) nu se mai poate scrie la stdout, respectiv nu se mai poate citi de la stdin, fișierul (stdin sau stdout) fiind închis
- obținere de la tastatură
  - Linux: combinație de taste CTRL+D
  - Windows: combinație de teste CTRL+Z

Operații elementare de I/O

Cateva precizări de I/O

- Funcțiile de scriere la stdout (ex. putchar, printf, … etc) nu vor scrie efectiv la stdout exact în momentul apelului. Fișierul stdout este precedat de o memorie tampon (buffer)
- Datele scrise la stdout prin intermediul funcțiilor de intrare ieșire se vor scrie în următoarele situații
  - se întâlnește caracterul '\n'
  - se umple buffer-ul tampon (dimensiunea de obicei 4096 bytes)
  - se întâlnește end of file (EOF)
- Fișierul stderr nu este precedat de vreun buffer tampon

Operații elementare de I/O

Exemple cod cu putchar() și getchar()

- Program care citește de la stdin (tastatură) si printează totul de stdout (terminal)

```c
#include <stdio.h>

int main(void)
{
	int c = 0;
	while ((c = getchar()) != EOF)
	{
		putchar(c);
	}
	return 0;
}
```

- Testarea programului

```
valy@staff:~/teaching$ gcc -Wall -o p in_out_1.c
valy@staff:~/teaching$ ./p
test test test
test test test
abc
abc
ana are mere
ana are mere
valy@staff:~/teaching$
```

compilare program

lansare program în execuție

linie scrisă de utilizator urmată de '\n' – tasta enter

linie scrisă de program (prin putchar())

linie scrisă de program (prin putchar())
după utilizatorul a trimis EOF prin CTRL+D

Operații elementare de I/O

Exemple cod cu putchar() și getchar()

- O altă metodă de a testa programul anterior
  - se poate redirecta un fișier ca fiind stdin pentru programul în execuție. În acest caz, un fișier existent pe disc poate să ia locul lui stdin pentru un program. Când fișierul se termină programul detectează prin EOF

```
valy@staff:~/teaching$ ./p < testfile.txt
o linie de test
inca o linie de text !
Imi place cursul de PC ! :))
Ultima linie din fisier

valy@staff:~/teaching$
```

  - programul “nu știe” de existența fișierului testfile.txt și nici nu știe faptul ca fizic datele vin de acolo
  - programul în continuare citește de la stdin (programul “crede” că citește în continuare de la tastatură)
  - prin semnul “<“ se redirectează fișierul dat prin calea din dreapta la stdin al programului din stânga

Operații elementare de I/O

Exemple cod cu putchar() și getchar()

- Program care citește de la stdin și transformă literele mici în literele mari (upper case to lower case)

```c
#include <stdio.h>
#include <ctype.h>

int main(void)
{
	int c = 0;
	while ((c = getchar()) != EOF)
	{
		putchar( toupper(c) );
	}
	return 0;
}
```

```
valy@staff:~/teaching$ gcc -Wall -o p in_out_2.c
valy@staff:~/teaching$ ./p < testfile.txt
O LINIE DE TEST
INCA O LINIE DE TEXT !
IMI PLACE CURSUL DE PC ! :))
ULTIMA LINIE DIN FISIER

valy@staff:~/teaching$
```

- Program care numără câte litere sunt date la intrare

```c
#include <stdio.h>
#include <ctype.h>

int main(void)
{
  int c = 0;
  int count = 0;
	while ((c = getchar()) != EOF)
	{
		if (isalpha(c))
		{
			count++;
		}
	}
  printf ("numar total de litere: %d\n", count);
  return 0;
}
```

```
valy@staff:~/teaching$ gcc -Wall -o p in_out_3.c
valy@staff:~/teaching$ ./p < testfile.txt
numar total de litere: 66
valy@staff:~/teaching$
```

Operații elementare de I/O

Tema. Execitii

- Scrieți un program, folosind getchar() și putchar() prin care se numără cuvintele de la stdint. Se consideră un cuvânt ca fiind o secvență de litere mici și/sau mari care despărțite prin unul sau mai multe spații, tab-uri, linii noi și EOF. Testarea se va face atât clasic prin scriere la stdin cât și prin redirectare cu un fișier text realizat inainte ca și caz de test.
- Scrieți un program, folosind getchar() și putchar(), care, primind la stdin un fișier ce conține un cod C, va printa la stdout același fișier dar fără comentarii. Practic, programul va elimina comentariile din fișierul C primit prin redirectare de stdin și va printa rezultatul la stdout. Se consideră comentarii în C:

```
// comentariu pe o linie
/* comentariu pe o linie 2 */
/* comentariu pe mai
multe
linii */
```

De asemenea, se consideră ca comentariile nu încep doar pe rând nou:

```
int x;// comentariu pe o linie
apel_de_functie();/* comentariu pe o linie 2 */
/* comentariu pe mai
multe
linii */
```

Funcții standard de intrare-ieșire

Funcții standard de intrare-ieșire

Funcțiile printf și scanf

- Funcția printf – funcție standard de printare la stdout a unor șiruri de caractere complexe
  - pagină de manual: **`man 3 printf`**
  - antet: **`int printf(const char *format, ...);`**
  - explicație preliminară a parametrului *const char *format* - constantă de tip șir de caractere ce conține șirul de caractere ce va fi printat la ieșirea standard. Șirul de caractere poate conține niște tipare (directive de formatare) bine definite ce specifică tipul/tipurile
    de date al argumentelor ce urmează după parametrul *format*.
  - argumentul **...** – semnifică faptul ca urmează un număr variabil de argumente – acestea sunt date în funcție de directivele de formatare ce apar în constanta de tip șir de caractere reprezentată de *format*.
  - tiparele (pattern) de format, directivele de formatare sunt niște sintagme ce încep cu caracterul % - caracterul are o semnificație aparte pentru funcțiile printf și scanf și este interpretat ca fiind o directivă de formatare
  - returnează numărul de caractere printate
- Funcția scanf – funcție standard de citire de la stdin a unor informații mai complexe ce pot fi interpretate
  - pagină de manual: **`man scanf`**
  - antet: **`int scanf(const char *format, ...);`**
  - parametrul *format* are o semnificație similară ca și la printf – specifică formatul datelor de la intrare. Cu ajutorul tiparelor de formatare (ce incep cu %) se pot specifica tipul datelor ce se vor citi în argumentele ca acel tip ce urmează dupa format
  - argumentul **...** – semnifică faptul ca urmează un număr variabil de argumente – acestea sunt date în funcție de tiparele ce apar în constanta de tip șir de caractere reprezentată de *format*.
  - returnează numărul de elemente citite

Funcții standard de intrare-ieșire

Funcția printf. Scriere formatată

- Directive de formatare pentru scriere formatată cu printf pentru tip

%d, %i: întreg zecimal cu semn
%o: întreg în octal, fără 0 la început
%u: întreg zecimal fără semn
%x, %X: întreg hexazecimal, fără 0x/0X; cu a-f pt. %x, A-F pt. %X
%c: caracter (se interpretează un întreg dat ca și argument ca și caracter – corespondență ASCII)
%f, %F: real fără exp.; precizie implicită 6 poz.; la precizie 0: fără punct
%e, %E: real, cu exp.; precizie implicită 6 poz.; la precizie 0: fără punct
%g, %G: real, ca %e, %E dacă exp. < -4 sau precizia; altfel ca %f. Nu tipărește zerouri sau punct zecimal inutil
%a, %A: real hexazecimal cu exponent zecimal de 2: 0xh.hhhhp±d
%%: caracterul procent

%p: pointer, în format dependent de implementare (tipic: hexazecimal)
*%n: scrie în argument (int *) nr. de caractere scrise până în prezent;*
%s: șir de caractere, până la '\0' sau nr. de caractere dat ca precizie

- Adaptare după: Marius Minea, curs "Programarea calculatoarelor 2", AC, 2005

Funcții standard de intrare-ieșire

Funcția printf. Scriere formatată

- Directive de formatare pot conține si alte componente opțională. Sintaxă generală:

```
% fanion dimensiune . precizie modificator tip
```

  - **Fanion**:
    -: aliniază valoarea la stânga într-un câmp de dimensiune dată
    +: pune + înainte de număr pozitiv de tip cu semn (implicit la număr pozitiv nu pune +)
    spațiu: pune spațiu înainte de număr pozitiv de tip cu semn
    #: format alternativ (0X/0x/0 pt. hex/octal, alte zecimale pt. reali)
    0: completează cu 0 la stânga până la dimensiunea dată
  - **Dimensiune**: un număr întreg ce reprezintă numărul minim de caractere pe care se scrie argumentul (aliniat la dreapta și completat cu spații sau conform modificatorilor)
  - **Precizie:** punct . urmat de un număr întreg opțional (dacă apare doar punct – precizia este 0)
    numărul minim de cifre pentru %diouxX (completate cu 0)
    numărul de cifre zecimale pentru %Eef
    numărul de cifre semnificative pentru %Gg
    numărul maxim de caractere de tipărit dintr-un șir (pentru %s)

- Adaptare după: Marius Minea, curs “Programarea calculatoarelor 2”, AC, 2005

Funcții standard de intrare-ieșire

Funcția printf. Scriere formatată

- Directive de formatare pot conține si alte componente opțională. Sintaxă generală:

```
% fanion dimensiune . precizie modificator tip
```

  - **Modificatori**:

hh: argumentul este char (pt. diouxXn)
h: argumentul este short (pt. diouxXn)
l: argumentul este long (pt. diouxXn) sau double (pt. aAeEfFgG)
ll: argumentul este long long (pt. diouxXn)
L: argumentul este long double (pt. aAeEfFgG)

- Adaptare după: Marius Minea, curs “Programarea calculatoarelor 2”, AC, 2005

Funcții standard de intrare-ieșire

Funcția printf. Scriere formatată

- Exemple de scriere formată:

```
// Scriere de numere reale in diverse formate:
printf("%f\n", 1.0/1100); /* 0.000909 : 6 poz. zecimale */
printf("%g\n", 1.0/1100); /* 0.000909091 : 6 poz. semnificative */
printf("%g\n", 1.0/11000); /* 9.09091e-05 : 6 poz. semnificative */
printf("%e\n", 1.0); /* 1.000000e+00 : 6 cifre zecimale */
printf("%f\n", 1.0); /* 1.000000 : 6 cifre zecimale */
printf("%g\n", 1.0); /* 1 : f˘ar˘a punct zecimal, zerouri inutile */
printf("%.2f\n", 1.009); /* 1.01: 2 cifre zecimale */
printf("%.2g\n", 1.009); /* 1: 2 cifre semnificative */
// Scriere de numere intregi in forma de tabel:
printf("|%6d|", -12);         /* |   -12| */
printf("|%-6d|", -12);        /* |-12   | */
printf("|%+6d|", 12);         /* |   +12| */
printf("|% d|", 12);          /* | 12|    */
printf("|%06d|", -12);        /* |-00012| */
```

- Adaptare după: Marius Minea, curs “Programarea calculatoarelor 2”, AC, 2005

Funcții standard de intrare-ieșire

Funcția scanf. Citire și interpretare

- Funcția scanf
  - citește conform tiparului specificat prin parametrul *format*
  - datele de intrare sunt în format text și sunt convertite în funcție de tiparul specificat și asignate variabilelor date ca și parametru
  - citește până când a epuizat tiparul sau până când datele de intrare nu mai corespund formatului – in acest caz varibilele rămân neatribuite iar caracterele de la intrare neconsumate. Exemplu scanf("politehnica"); intrare: policlinica\n
    - citește **poli** iar **clinica\n** rămâne neconsumat în buffer-ul de intrare
    - se testează valoarea returnată pentru detecția unei citirii corecte și se consumă (eventual datele înainte de o nouă citire)
  - spațiile albe din intrare – separatori impliciți
  - spațiu în format consumă oricâte spații din intrare până la un caracter ce nu e spațiu

```
#include <stdio.h>

int main(void)
{
     int a,b;
     scanf("%d%d", &a, &b);
     scanf("%d %d", &a, &b); // acelasi comportament cu linia precedenta
}
```

  - momentan: absolut necesară utilizarea caracterului & înainte de parametru (doar așa de poate scrie în variabila respectivă)
  - lipsa caracterului & determină ne-scrierea în variabilă, rezultatul nu se scrie

- Adaptare după: Marius Minea, curs "Programarea calculatoarelor 2", AC, 2005

Funcții standard de intrare-ieșire

Funcția scanf. Citire și interpretare

- Directive de formatare pentru citire formatată cu scanf pentru tip

%d: întreg zecimal cu semn
%i: întreg zecimal, octal (0) sau hexazecimal (0x, 0X)
%o: întreg în octal, precedat sau nu de 0
%u: întreg zecimal fără semn
%x, %X: întreg hexazecimal, precedat sau nu de 0x, 0X
%c: orice caracter; nu sare peste spațiii (doar " %c")
%a, %A, %e, %E, %f, %F, %g, %G: real (posibil cu exponent)
%%: caracterul procent

%s: șir de caractere, până la primul spațiu alb. Se adaugă '\0'.
%p: pointer, în formatul tipărit de printf
%n: scrie în argument (int *) nr. de caractere citite până în prezent, nu citește nimic; nu incrementează nr. de câmpuri convertite/atribuite
%[· · ·]: șir de caractere din mulțimea indicată între paranteze
%[^· · ·]: șir de caractere exceptând mulțimea indicată între paranteze

- Adaptare după: Marius Minea, curs "Programarea calculatoarelor 2", AC, 2005

Funcții standard de intrare-ieșire

Funcția scanf. Citire și interpretare

- Exemple de scriere formată:

```
// Citire de numere diverse formate:
Format scanf                    Intrare      Rezultat
scanf("%d%d", &m, &n);          12 34        12 34
scanf("%2d%2d", &m, &n);        12345        12 34
scanf("%d.%d", &m, &n);         12.34        12 34
scanf("%f", &x);                12.34        12.34
scanf("%d%x", &m, &n);          123a         123 0xA
```

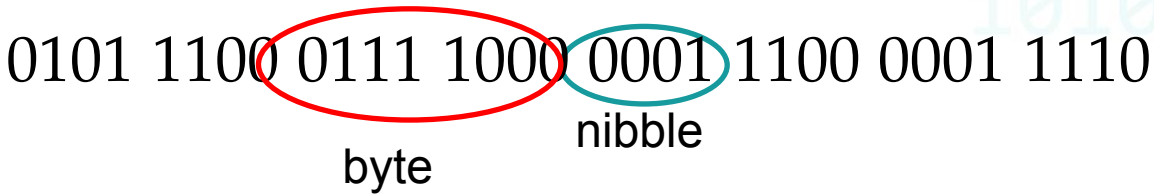
- Adaptare după: Marius Minea, curs “Programarea calculatoarelor 2”, AC, 2005

Secțiunea VII

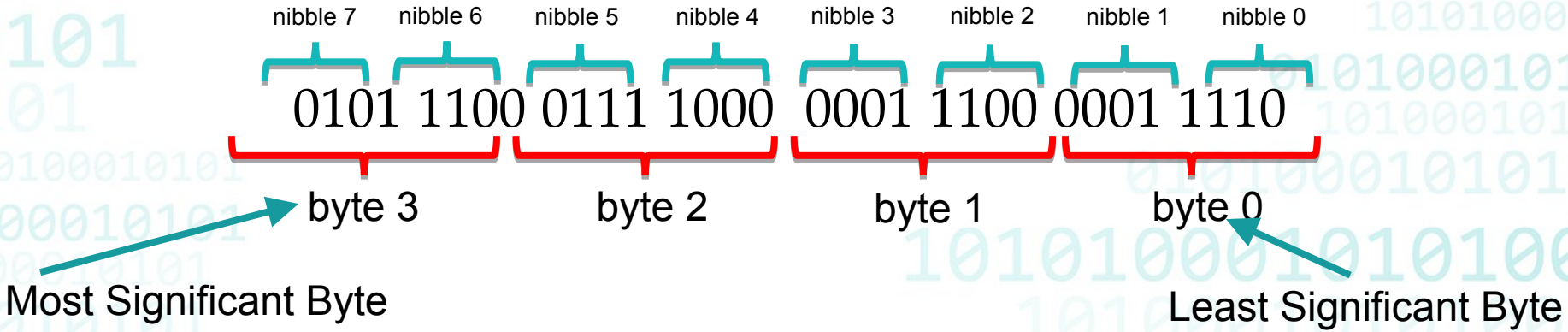
Operatori pe biți

Operatori pe biți

Bits, bytes, nibbles ?



- Bit – cifra binară elementară
- Nibble – un grup de 4 biți
- **Byte (octet) – un grup de 8 biți**
- numerotare biți, bytes, nibble – începând de la 0 din dreapta spre stânga (cu offset 0 sau cu un offset dat)



Operatori pe biți

Definiții și exemple

- Operatori pe biți – utilizați pentru a putea prelucra binar date la nivel de bit
- se pot aplica doar operanzilor de tipuri întregi cu sau fără semn
- se folosesc cu precădere în programarea pe sisteme embedded
- pot optimiza anumite operații (este mult mai eficientă deplasarea la la stânga sau la dreaptafață de inmulțirea, respective împărțirea cu 2)

Clasă de precedență	Operator	Descriere	Tip operator	Asociativitate
2	~	Complement față de 1 pe biți Negare pe biți	unar	dreapta->stânga
5	<< si >>	Deplasare stânga/ dreapta a biților	binar	stânga->dreapta
8	&	ŞI pe biți	binar	stânga->dreapta
9	^	SAU-EXCLUSIV pe biți	binar	stânga->dreapta
10	|	SAU pe biți	binar	stânga->dreapta

Operatori pe biți

Operații uzuale de bază

- operații cu operatori simpli, binari

Operație	Cod	Semnificație
testare de bit	`n & (1 << k)`	valoare este nenulă dacă bitul *k* din *n* este 1
setare de bit	`n = n | (1 << k)`	bitul *k* din *n* devine 1, restul rămân neschimbați
resetare de bit	`n = n & ~(1 << k)`	bitul *k* din *n* devine 0, restul rămân neschimbați
inversare	`n = n ^ (1 << k)`	bitul *k* din *n* se schimbă, restul rămân neschimbați

- operații cu operatori compuși, unari

Operație	Cod	Semnificație
setare de bit	`n |= 1 << k`	bitul *k* din *n* devine 1, restul rămân neschimbați
resetare de bit	`n &= ~(1 << k)`	bitul *k* din *n* devine 0, restul rămân neschimbați
inversare	`n ^= 1 << k`	bitul *k* din *n* se schimbă, restul rămân neschimbați

Operatori pe biți

Exemplu de program

- Să se scrie o funcție ce afișează pe biți un număr pe 16 biți fără semn.

```c
#include <stdio.h>
#include <stdint.h>

void print_bit_16(uint16_t nr)
{
  uint16_t mask = 0x8000; // 0b1000000000000000
  uint8_t i = 0;
  for (i = 0; i < 16; i++)
    {
      if ((nr & mask) == 0)
        {
          printf ("0");
        }
      else
        {
          printf ("1");
        }
      mask = mask >> 1; // mask >>= 1;
    }
  printf ("\n");
}

int main(void)
{
  print_bit_16(0x8001);
  return 0;
}
```

Operatori pe biți

Probleme

- Să se scrie o funcție ce adună pe biți 2 numere fără semn pe o lungime de 32 biți. Se va implementa algoritmul după metoda “pencil and paper”
- Să se scrie o funcție ce primește ca argument un număr întreg fără semn pe 64 de biți (uint64_t) și returnează paritatea pe biți a acestuia. Prin paritate se înțelege: paritate pară dacă numărul conține un număr par de biți de 1 și impară dacă numărul conține un număr impar de biți de 1.
- Să se scrie o funcție ce primește ca argumente 2 numere fără semn pe 1 byte si returnează un număr fără semn pe 2 bytes compus din cele 2 numerele primite ca argument. Primul număr reprezintă partea cea mai semnificativă (MSW – most significant word) a numărului returnat iar cel de-al doilea reprezintă partea cea mai puțin semnificativă (LSW – least significant word) a numărului returnat
- Să se scrie o funcție similară cu cea de la exercițiul anterior dar care primește 4 argumente reprezentate prin numere fără semn pe un byte și returnează un număr pe 32 de biți fără semn obținut prin compunerea celor 4 argumente după același algoritm

Secțiunea VIII

Pointeri
Funcții cu pointeri
Tablouri
Matrici

Preprocesorul C

Preprocesorul C

Definiții. Directive.

- un editor de texte ce preia un cod C și realizează anumite operații de transformare de cod C în funcție de niște directive (macro-uri) speciale dedicate
- **directiva #include**: sintaxa: `#include <nume_fişier_header>` sau `#include "nume_fişier_header"`
  - include textual (copiază) fișierul numit. <…> - fișierul este căutat în calea implicită de incluziuni "…" – cautat în directorul curent
- **directiva #define:** realizează o substituție textuală simplă

```c
#include <stdio.h>

#define COUNT 50

int main(void)
{
      int n;
      n = COUNT;
      for (int i = 0; i < COUNT; i++)
      {


      }
      return 0;
}
```

C preprocessor

```c
#include <stdio.h>


int main(void)
{
      int n;
      n = 50;
      for (int i = 0; i < 50; i++)
      {


      }
      return 0;
}
```

  - se înlocuiește textual fiecare apariție a lui COUNT cu 50

Preprocesorul C

Directiva #define

- **directiva #define:** forma generală
- alte exemple

```
#define nume(arg1, arg2,… ,argn
```

```
#define max(A, B) ((A) > (B) ? (A) : (B))

int main(void)
{
      int n;
      n = max(n, 2);
}
```

C preprocessor

```
int main(void)
{
      int n;
      n = n > 2 ? n : 2;
}
```

- substituție textuală – fără vreo verificare prealabilă de sintaxă – pot apărea erori
- se recomandă utilizarea parantezelor în jurul argumentelor
- de obicei nu se pune ; după vreo definiție de directivă #define – pot aparea erori de sintaxă în urma substituțiilor

- **operatorul #** aplicat unui argument din directiva define produce o constantă de tip șir de caractere
  - `#define to_string(str)  # str` – apelat `to_string(un_string)` devine `"un_string"`
- operatorul ## (binar) aplicat asupra a două argumente din directiva define produce o constantă de tip șir de caractere rezultată în urma concatenării celor 2 argumente
  - `#define to_string_2(a, b) a ## b` – apelat `to_string_2(poli, UPT)` devine `"poliUPT"`

Preprocesorul C

Directiva #if #ifdef #ifndef #else #endif

- **directivele #if #ifdef #ifndef #else #endif** se folosesc pentru compilarea selectivă a unor bucăți de cod
- exemplu: se poate traduce ușor prin “limbaj natural”
  - dacă macro-ul DEBUG este definit (există prima linie) atunci se va compila linia cu printf(“cod cu debug”) altfel se va compila linia printf (“cod fara debug”);
- definirea unui macro din argument dat compilatorului
  - in loc de prima linie din codul alaturat se poate scrie ca și argument dat compilatorului prin argumentul -D

```
gcc -DDEBUG prog.c
```

```
#define DEBUG

int main(void)
{
      #ifndef DEBUG
      printf ("cod fara debug\n");
      #endif


      #ifdef DEBUG
      printf ("cod cu debug\n");
      #endif


      #ifdef DEBUG
      printf ("cod cu debug\n");
      #else
      printf ("cod fara debug\n");
      #endif
}
```

Preprocesorul C

Efectuare de calcule statice

- **Efectuare de calcule statice**
  - un alt rol al preprocesorului C – a efectua calcule statice ce nu depind de execuția programului spre a degreva sistemul de calcul țintă de a efectua calcule inutile
  - exemple:

```c
#define NR 10

#define MYNR (NR + 2)

int main(void)
{
    int n = 2 + 3;
    n = NR + 5;
    n = MYNR - 9;
}
```

C preprocessor

```c
int main(void)
{
    int n = 5;
    n = 15;
    n = 3;
}
```

Pointeri

Pointeri

Noțiuni elementare

- Orice variabilă în C este stocată la o anumită adresă în memorie
- Adresa: reprezentată de o valoare numerică pe 32 de biți (pentru procesoare cu arhitectura pe 32 de biți) sau pe 64 de biți (pentru procesoare cu arhitectura pe 64 de biți) care indică o poziție dintr-un segment de memorie
- Generic: dimensiunea unei adrese de memorie depinde de dimensiunea magistralei de adrese a procesorului sistemului de calcul
- Adresa unei variabile se poate obține folosind operatorul unar **&** pus în fața unei variabile
- Exemplu (folosind directiva %p pentru printf pentru afișare) :

```c
#include <stdio.h>

int global;

int main(void)
{
  int local;
  static int local_static;
  printf ("address of local \t\t%p\n", &local);
  printf ("address of local_static \t%p\n", &local_static);
  printf ("address of global \t\t%p\n", &global);
  return 0;
}
```

```
valy@staff:~/teaching$ ./p
address of local                0x7ffebdf7835c
address of local_static         0x5613f0500034
address of global               0x5613f0500038
valy@staff:~/teaching$
```

Pointeri

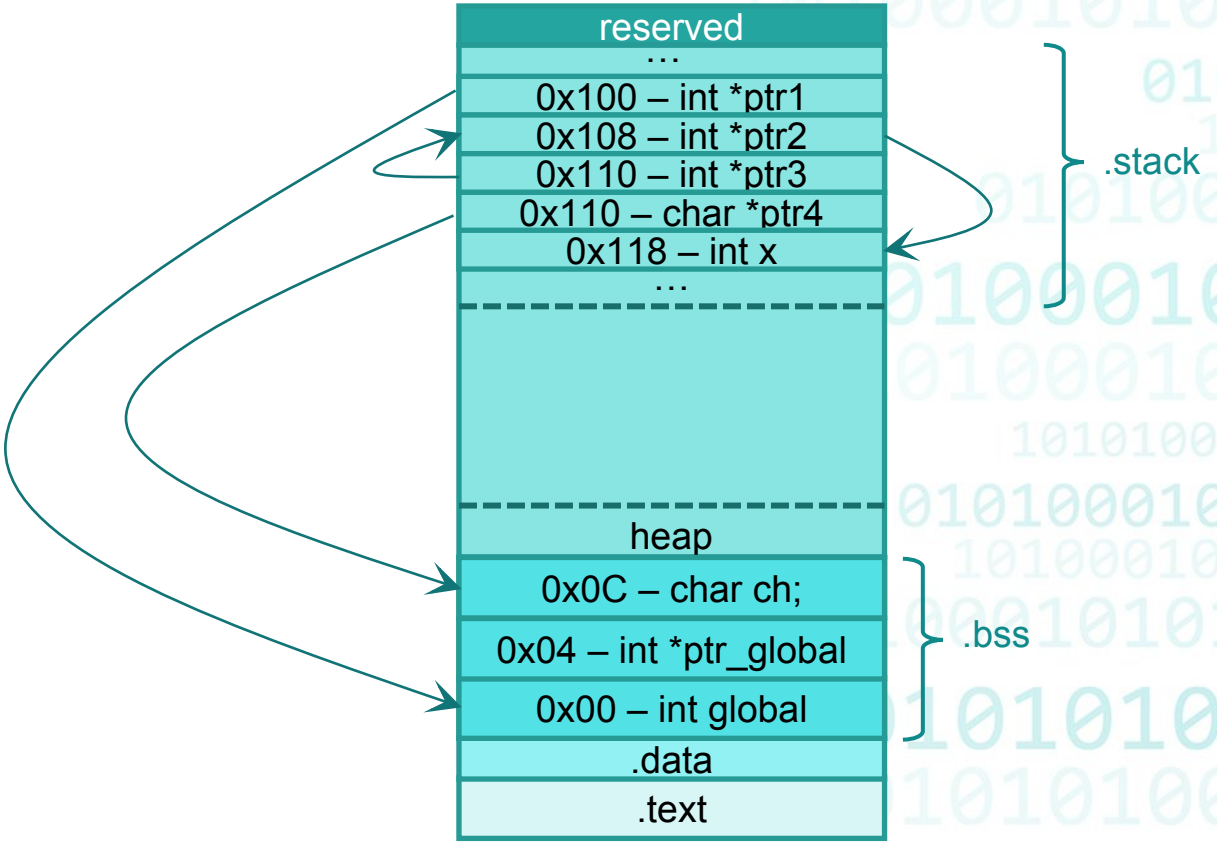
Noțiuni elementare. Definiții

- Considerăm variabila `int n`
  - `int n;` are tipul `int;` `&n` reperezintă adresa variabilei `int n` și are tipul `int *` (adresa de int)
  - `float f;` are tipul `float;` `&f` reperezintă adresa variabilei `float f` și are tipul `float *` (adresa de float)
- Limbajul C permite utilizarea tipurilor de date de tip adresa; orice tip de date în limbajul C poate "avea" și un tip de date de tip adresă de acel tip
- Limbajul C oferă posibilitatea de a declara o variabilă de tip adresa (de tip pointer) către un anumit tip. Sintaxă: `tip * nume_variabilă;`
  - nume_variabila reprezintă un pointer la o valoare de tip *tip*
  - pointer - o variabilă ce conține adresa unei alte variabile, a unei zone de memorie de un anumit tip
  - tipul este important, zona de memorie referențiată de variabila de tip pointer va fi interpretată în operații ca având tipul specificat

Pointeri

Noțiuni elementare. Exemplu

```c
int global;
int *ptr_global;
char ch;

int main(void)
{
      int *ptr1;
      int *ptr2;
      int *ptr3;
      char *ptr4;
      int x;


      ptr1 = &global;
      ptr2 = &x;


      ptr3 = ptr2;
      ptr4 = &ch;
      return 0;
}
```



Pointeri

Noțiuni elementare. Definiții. Exemplu

Clasă de precedență	Operator	Descriere	Tip operator	Asociativitate
2	*	Dereferențiere	unar	dreapta->stânga
	&	Operator adresă	unar	

- Operatorul prefix unar * - obține obiectul (valoarea) de la adresa dată de pointer
  - `int *p` – p are tipul `int*` , `*p` are tipul `int`

```c
#include <stdio.h>

int global = 44;

int main(void)
{
      int *ptr1;
      int *ptr2;
      int x = 22;
      ptr1 = &global;
      ptr2 = &x;
      int s = 0;
      s = *ptr1 + *ptr2;
      printf ("%d\n", *ptr1);
      printf ("%d\n", *ptr2);
      printf ("%d\n", s);
      return 0;
}
```

```
valy@staff:~/teaching$ ./p
44
22
66
valy@staff:~/teaching$
```

Pointeri

Modificatorul const

- Modificatorul **const** pus în fața declarației unei variabile de tip pointer are efect asupra adresei referențiate de pointer și nu permite modificarea conținutului acesteia prin variabila de tip pointer declarată ca și const
- dacă se încearcă scrierea zonei de memorie a unui pointer declarat ca și const - eroare de compilare
- exemplu:

```c
#include <stdio.h>

int global = 44;

int main(void)
{
  int x = 22;
  const int *ptr3 = NULL;
  printf ("%p\n", ptr3);
  ptr3 = &x;
  printf ("%p\n", ptr3);
  *ptr3 = 9;
  printf ("%d\n", *ptr3);
  printf ("%d\n", x);
  return 0;
}
```

```
valy@staff:~/teaching$ gcc -Wall -o p ptr5.c
ptr5.c: In function ‘main’:
ptr5.c:12:9: error: assignment of read-only location ‘*ptr3’
   *ptr3 = 9;
         ^
```

Pointeri

Consecințe

- Secțiunea III – Variabile – o variabilă neinițializată are o valoare nedefinită - reprezintă un risc major
- Pointer-ul este o variabilă de tip adresă de un anumit tip
- o variabilă de tip pointer neinitilizată
  - reprezintă un risc extrem de mare – produce coruperea memoriei - se va citi sau scrie de la o adresă necunoscută
  - **compilatorul nu generează eroare ci doar warning** !

```
valy@staff:~/teaching$ gcc -Wall -o p ptr2.c
ptr2.c: In function ‘main’:
ptr2.c:18:3: warning: ‘ptr3’ is used uninitialized in this function [-Wuninitialized]
   printf ("%d\n", *ptr3);
   ^~~~~~~~~~~~~~~~~~~~~~
valy@staff:~/teaching$
```

- **ignorarea** unui astfel de warning duce la **coruperea memoriei**
- Pointerii, ca și orice altă variabilă, trebuie inițializați înainte de utilizare

Pointeri

Consecințe

- Se pot atribui între ei doar pointeri de același tip
- sub nici o formă nu se **atribuie pointeri de tip diferiți** – conținutul din memorie va fi interpretat greșit și duce la **coruperea memoriei**
- Exemplu: compilatorul genereaza doar warning !

```
int main(void)
{
	char ch;
	char *p = &ch;
	int *a;
	a = p;
	(*a)++;
	return 0;
}
```

```
valy@staff:~/teaching$ gcc -Wall -o p ptr3.c
ptr3.c: In function 'main':
ptr3.c:6:5: warning: assignment to 'int *' from incompatible pointer type 'char *' [-Wincompatible-pointer-types]
   a = p;
     ^
valy@staff:~/teaching$
```

Pointeri

Consecințe

- la o atribuire de pointeri de tipuri diferite compilatorul poate genera warning
- un astfel de warning este extrem de periculos să fie ignorat duce la coruperea memoriei. Cum ?
  - ch are dimensiunea de 1 byte
  - char *p este un pointer către un tip de date char cu dimensiune de 1 byte
  - int *a este un pointer către un tip de date int cu dimensiune de 4 bytes
  - a = p – pointerul a primește adresa lui ch (are memorie alocată pentru 1 byte)
  - (*a)++ - se incrementează cu 1 conținutul de la adresa referită de a – această adresă este de tip int (pe 4 bytes) dar memorie alocată este doar pentru 1 byte - incrementare a unei zone de memorie incorectă - coruperea memoriei

Pointeri

Noțiuni elementare. Definiții

- Tipul de date `void *`
  - Nu indică nimic – nu are tip
  - poate fi atribuit
  - nu poate fi referențiat fără o convesie explicită (pentru ca nu se stie **ce** indică)
- Un pointer se poate atribui și cu adresa nulă (0)
  - se definește valoarea NULL (în stddef.h – inclus și prin stdlib.h) ca fiind #define NULL (void *)0
  - se poate atribui și inițializa orice pointer cu valoarea NULL
  - un pointer cu valoarea NULL nu se poate dereferenția – un program nu are drepturi sa acceseze adresa 0

Funcții cu pointeri

Funcții cu pointeri

Funcții cu argumente de tip pointer

- Problema: în limbajul C argumentele unei funcții de transmit doar prin valoare
  - se copiază valoarea argumentelor de la apel
- Ⓟ consecință: un parametru local modificat în corpul unei funcții nu modifică și valoarea varibilei cu care a fost apelată
- Problema: Se se scrie o funcție care interschimbă valorile a 2 argumente (swap)

```c
#include <stdio.h>

void swap(int x, int y)
{
  int aux;
  aux = x;
  x = y;
  y = aux;
}

int main(void)
{
  int a = 3;
  int b = 5;
  printf ("%d %d\n", a, b);
  swap(a,b);
  printf ("%d %d\n", a, b);
  return 0;
}
```

```
valy@staff:~/teaching$ ./p
3 5
3 5
valy@staff:~/teaching$
```

- la apelul swap se copiază valorile lui a și b pe stivă iar parametri locali vor primt aceste valori
- intern, in funcție valorile parametrilor x și y se modifică dar în afara funcției a și b nu se modifică – modificarea nu se vede și în exteriorul funcției – este normal și corect fiind doar o copiere de valori

Funcții cu pointeri

Funcții cu argumente de tip pointer

- Problema: Se se scrie o funcție care interschimbă valorile a 2 argumente (swap)
- SOLUȚIE: folosirea în funcție a argumentelor de tip pointer și trimiterea adreselor în apelul swap

```c
#include <stdio.h>

void swap(int *x, int *y)
{
  int aux;
  aux = *x;
  *x = *y;
  *y = aux;
}

int main(void)
{
  int a = 3;
  int b = 5;
  printf ("%d %d\n", a, b);
  swap(&a,&b);
  printf ("%d %d\n", a, b);
  return 0;
}
```

```
valy@staff:~/teaching$ ./p
3 5
5 3
valy@staff:~/teaching$
```

- la apelul swap se copiază valorile adreselor lui a și b pe stivă iar parametri locali vor primt aceste valori, ce reprezintă adresele variabilelor a și b
- În continuare adresele lor, valorile pointerilor (ce sunt adrese) nu pot fi modificate, fiind situația precedentă
- pe de altă parte, având acces la adresă se poate modifica conținutul de la acea adresa folosind pointeri cu dereferențiere

Funcții cu pointeri

Funcții cu argumente de tip pointer

- Când se folosesc argumente de tip pointer la funcții ?
  - când se dorește modficarea valorii unui parametru în interiorul funcției cu efect și în exteriorul acesteia
  - când se dorește transmiterea ca argument a unui tip de date de mari dimensiuni: se transmite cu pointer pentru a se evita copierea pe stivei a valorii de mari dimensiuni. Transmiterea cu pointer va duce doar la copierea pe stivă a adresei
  - se poate folosi modificatorul **const** pentru a se restricționa modificarea zonei de memorie referită de pointer… eventual pentru a transmite o valoare de dimensiune mare fără drept de scriere

Funcții cu pointeri

Funcții cu argumente de tip pointer – Consecințe

- Explicații suplimentare de utilizare a funcției scanf

```c
#include <stdio.h>

int main(void)
{
     int a,b;
     scanf("%d%d", &a, &b);
     scanf("%d %d", &a, &b); // acelasi comportament cu linia precedenta
}
```

- Situația este absolut identică cu problema precedentă
- daca nu se folosește operatorul & pentru a se trimite adresa variabilelor a și b, funcția scanf va scrie doar în copia locală iar variabilele a și b rămân neschimbate
- folosirea operatorului & duce la trimiterea către scanf a adreselor variabilelor a și b si astfel funcția scanf va modifica conținutul de la aceste adrese, adică conținutul variabilelor a și b

Funcții cu pointeri

Funcții cu valoare returnată de tip pointer

- O funcție poate să returneze o valoare de tip pointer
- este total greșit ca o funcție să returneze un pointer către o variabilă locală sau parametru local – la momentul returnării variabilele locale și parametri unei funcții **nu mai există**
- funcția poate returna o valoare de tip pointer doar dacă variabila spre care pointerul referă mai există după return
  - returnare de pointer către o variabilă declarată cu static
  - returnare de pointer către o variabilă declarată global
  - returnare de pointer către o variabilă alocată dinamic

Tablouri

Tablouri

Definiții

- Tablou (șir, array, vector) – reprezintă o structură de date ce definește o secvență de elemente de același tip alocate în mod continuu și consecutiv în memorie
- Sintaxă de declarație statică:
  `tip nume_variabilă[dimensiune];`
- Accesarea elementelor din tablou – prin iterație folosind un iterator întreg pozitiv. Iterație se face de la 0 la (dimensiune - 1)
  - Sintaxă: `nume_variabila[iterator]`
- Exemplu: program ce citește un tablou de intregi de la tastatură si il afișează la sfâșit

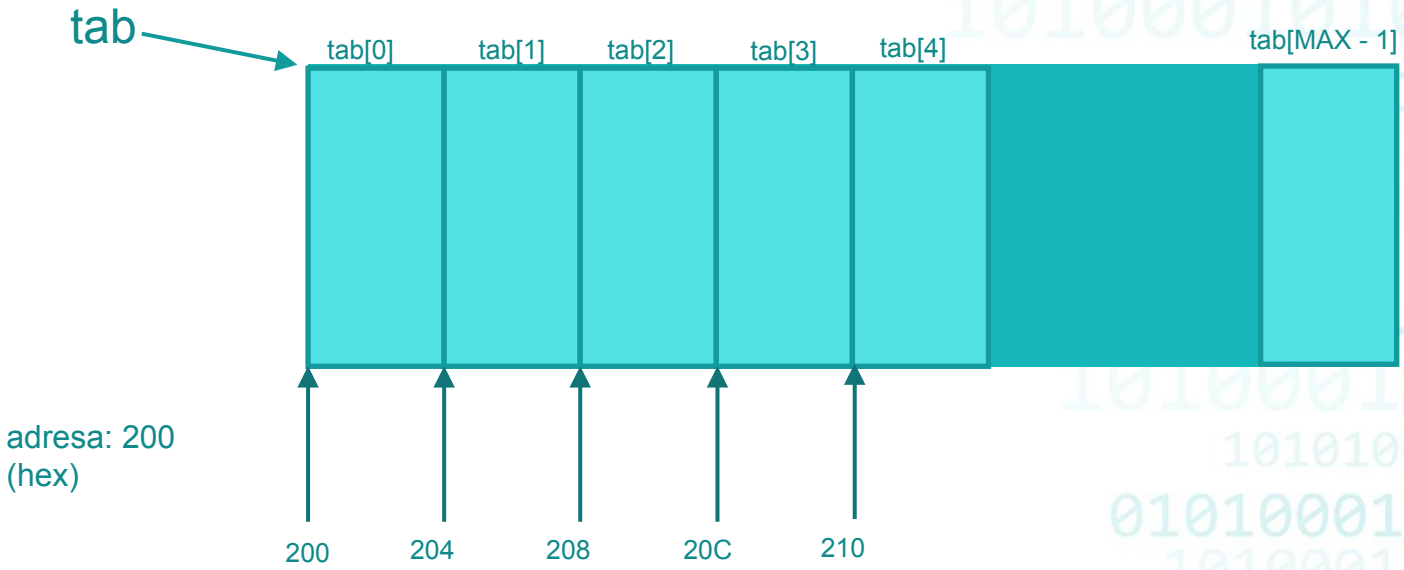
```
valy@staff:~/teaching$ ./p
Dati numar elemente (<=128):4
Dati elementul 0:2
Dati elementul 1:3
Dati elementul 2:4
Dati elementul 3:5
Tabloul este:
2 3 4 5
valy@staff:~/teaching$
```

```c
#include <stdio.h>

#define MAX 128

int main(void)
{
  int tab[MAX];
  int n = 0;
  int i = 0;
  printf ("Dati numar elemente (<=%d):", MAX);
  scanf("%d", &n);
  if (n > 128)
    {
      printf ("Eroare\n");
      return -1;
    }
  for (i = 0; i < n; i++)
    {
      printf ("Dati elementul %d:", i);
      scanf ("%d", &tab[i]);
    }
  printf ("Tabloul este:\n");
  for (i = 0; i < n; i++)
    {
      printf ("%d ", tab[i]);
    }
  printf ("\n");
  return 0;
}
```

Tablouri

Reprezentare în memorie

- `int tab[MAX]`
- `sizeof(int) = 4`



- $adresa\,(tab\,[i]) = adresa\,(tab\,[0]) + i{*}sizeof\,(tab\,[0])\,, i = 0...MAX{-}1,\ tab\,[0] = tab$
- pentru accesarea unui tablou este necesar doar primul element și un iterator întreg fără semn
- $sizeof\,(tab) = MAX{*}sizeof\,(tab\,[0]) = MAX{*}sizeof(tip)$

Tablouri

Tablouri și pointeri

- În limbajul C – există o relație extrem de strânsă între pointeri si tablouri

`int tab[MAX];`

  - tab este echivalent cu &tab[0] – tabloul este echivalent cu adresa primului element
  - &tab[0] este echivalent cu *tab;
  - eclarația de parametru - int tab[] este echivalent cu int *tab;
  - &tab[i] este echivalent cu tab + i
- iterație tablou cu iterator vs. iterație cu pointeri

```c
int main(void)
{
    int tab[10];
    int x = 0;
    int i = 0;
    for (i = 0; i < 10; i++)
    {
        x = tab[i];
    }
    return 0;
}
```

```c
int main(void)
{
    int tab[10];
    int x = 0;
    int i = 0;
    for (i = 0; i < 10; i++)
    {
        x = *(tab + i);
    }
    return 0;
}
```

- Adaptare după: Marius Minea, curs “Programarea calculatoarelor 2”, AC, 2005

Tablouri

Tablouri și aritmetică cu pointeri

- Pointer – un număr întreg fără semn ce specifică o adresa - se pot efectua operații aritmetice

1. Adunarea unui întreg cu pointer
   - folosită de obicei în scop iterativ
   - considerăm: **`tip a[MAX]`**
     - **`a + i echivalent cu &a[i]`**
     - **`*(a + i) echivalent cu a[i]`**
     - **`a + i`** – se obține o adresa mai mare decât *a* cu dimensiunea a *i* elemente de tip *tip* – se avansează în i elemente și nu bytes
     - se poate avansa și si cu număr efectiv de octeți prin forțarea tipului tabloului la un tip cu dimensiune de 1 octet (byte) `(char *)a + k * sizeof(tip)`
   - parcurgere tablou doar folosind aritmetică de pointeri

```
int main(void)
{
	int tab[10];
	int *p;
	int x;
	for (p = tab; p < tab + 10; p++)
	{
		x = *p;
	}
	return 0;
}
```

- Adaptare după: Marius Minea, curs “Programarea calculatoarelor 2”, AC, 2005

Tablouri

Tablouri și aritmetică cu pointeri

2. Diferența dintre 2 pointeri de același tip
   - folosit pentru a obține numărul de obiecte de tip *tip* ce încap între cele 2 adrese
   - se poate obține și diferența în octeți dacă se convertesc explicit ambii pointeri la un tip de date pe 1 byte

```
#include <stdio.h>

int main(void)
{
  int tab[10];
  int *p, *q;
  p = &tab[2];
  q = &tab[7];
  printf ("%ld\n", q - p);
  return 0;
}
```

- Adaptare după: Marius Minea, curs “Programarea calculatoarelor 2”, AC, 2005

Tablouri

Transmitere de tablouri ca parametri

- Se consideră afirmațiile anterioare
  - întregul tablou este reprezentat de adresa primului element
  - adresa primului element este un pointer
  - tabloul poate fi văzut ca un pointer
- Sintaxa:

```
void functie(int tab[]);
void functie(int *tab);

void functie(int tab[10]);
```

- ultima variantă – nu are rost să se specifice dimensiunea, nu se poate efectiv folosi
- se obișnuiește să se transmită si dimensiunea tabloului ca și parametru – dimensiunea este necesară pentru a cunoaște limita de iterație

```
void functie(int tab[], unsigned int size);
void functie(int *tab, unsigned int size);
```

- Adaptare după: Marius Minea, curs “Programarea calculatoarelor 2”, AC, 2005

Tablouri

Transmitere de tablouri ca parametri - Exemplu

```c
#include <stdio.h>

#define MAX 128

void readArray(int *a, int size)
{
	for (int i = 0; i < size; i++)
	{
		printf ("Dati elementul %d:", i);
		scanf ("%d", &a[i]);
	}


}

void printArray(int *a, int size)
{
	printf("Tabloul este: \n");
	for (int i = 0; i < size; i++)
	{
		printf ("%d ", a[i]);
	}
	printf ("\n");
}

int main(void)
{
	int tab[MAX];
	int n = 0;
	printf ("Dati numar elemente (<=%d):", MAX);
	scanf("%d", &n);
	readArray(tab, n);
	printArray(tab, n);
	return 0;
}
```

```
valy@staff:~/teaching$ ./p
Dati numar elemente (<=128):4
Dati elementul 0:2
Dati elementul 1:3
Dati elementul 2:4
Dati elementul 3:5
Tabloul este:
2 3 4 5
valy@staff:~/teaching$
```

Tablouri

Inițializarea tablourilor

- Inițializarea se face prin specificarea tuturor sau a unor valori ale elementelor tabloului între paranteze acolade { } (braces)
- Sintaxă:

```
tip nume_tablou[dimensiune] = {val0, val1, val2 ... }
```

- Exemplu – inițializare completă

```
int tab[4] = {1, 3, 4, 9};
```

- Exemplu – inițializare incompletă – se vor inițializa doar primele valori specificate

```
int tab[10] = {1, 3, 4, 9};
```

- La inițializare se poate omite dimensiunea tabloului – in acest caz dimensiunea va fi dată de numărul de valori specificate la inițializare

```
int tab[] = {1, 3, 4, 9, 0};
```

  - tabloul tab va avea dimensiunea 5

- Atenție: sintaxa de inițializare nu se poate folosi pentru atribuire de valori – va genera eroare de compilare

```
int tab[5];
tab = {1, 3, 4, 9, 0}; // eroare de compilare
```

Tablouri

Copierea tablourilor

- Problema: să se scrie o funcție care să copieze un conținutul unui tablou într-un alt tablou de aceeași dimensiune

```
void copyArray(int *src, int *dst, int size)
{
     for (int i = 0; i < size; i++)
     {
          dst[i] = src[i];
     }
}
```

- O variantă greșită: dst = src
  - prin această variantă pointerul dst va referenția spre aceeași zonă de memorie ca si src, deci conținutul nu s-a copiat, doar adresele vor fi identice - o modificare a lui dst va modifica și src (și invers)

Tablouri

Greșeli frecvente grave

- Depășire tablou – duce la coruperea memoriei – execuție impredictibilă, terminare program cu excepții

```
int main(void)
{
	int tab[SIZE];
	for (int i = 1; i <= SIZE; i++)
	{
		tab[i] = 0;
	}
	return 0;
}
```

- se iterează greșit tabloul de la 1 (tab[0] rămâne nefolosit)
- se accesează tab[SIZE] ce nu există, apare depășire, se scrie în memorie în afara tabloului - **corupere de memorie**

```
int main(void)
{
	int tab[SIZE];
	for (int i = 0; i <= SIZE; i++)
	{
		tab[i] = 0;
	}
	return 0;
}
```

- se iterează correct tabloul de la 0 dar se accesează tab[SIZE] ce nu există, apare depășire, se scrie în memorie în afara tabloului - **corupere de memorie**

```
int main(void)
{
	int tab[SIZE];
	for (int i = 0; i <= sizeof(tab); i++)
	{
		tab[i] = 0;
	}
	return 0;
}
```

- greșit ! sizeof(tab) returnează dimensiunea in bytes – in cazul exemplului este SIZE * sizeof(int) – apare depășire foarte mare

Tablouri

Observații

```
void functie1(void)
{
    int tab[10000000];
    // restul codului functiei
}

void functie2(void)
{
    int tab[10000];
    // restul codului functiei
}

void functie3(void)
{
    static int tab[10000];
    // restul codului functiei
}
```

- functie1 – variabila tab este pe stiva – programul se va termina cu eroare de memorie – **stiva este depășita – o abordare total greșită**
- functie2 – variabila tab este pe stiva – programul va rula, dar apelul de functie va fi foarte incet – pentru a pune pe stiva tab sunt necesare multe operații de PUSH iar la revenirea din apel multe operatii de POP
- functie3 – variabila tab este in .bss dar accesibilă doar functiei – codul se execută rapid fără operații pe stiva

- se folosește o variantă potrivită în funcție de context (mai puțin functie1 care va depăși stiva)

Secțiunea IX

Șiruri de caractere

Șiruri de caractere

Definiții

- Șirul de caractere (string) în limbajul C reprezintă un tablou de elemente de tip char ce conțin caractere și care, prin convenție, se termină cu caracterul '\0' – 0x00 (NULL)
- un șir de caractere ce nu se termină cu caracterul '\0' – **nu** e string și nu poate fi tratat ca atare – este doar un simplu tablou de tip char
- toate funcțiile standard de prelucrare de string-uri din C precum și printf și scanf se folosesc de convenția și definiția precedenta !
- sintaxă, exemplu și inițializare șir de caractere

```
char nume_string[dimensiune];

char mystring1[] = {'s','a','l','u','t', 0};
char mystring2[] = "salut";
```

  - mystring1 – declarație ca și tablou și initializare ca și tablou
  - mystring2 – declarație ca și tablou și inițializare cu constantă de tip șir de caractere
- cele 2 declarații (mystring1, mystring2) au un rezultat identic – toate reprezintă același lucru

Șiruri de caractere

Definiții

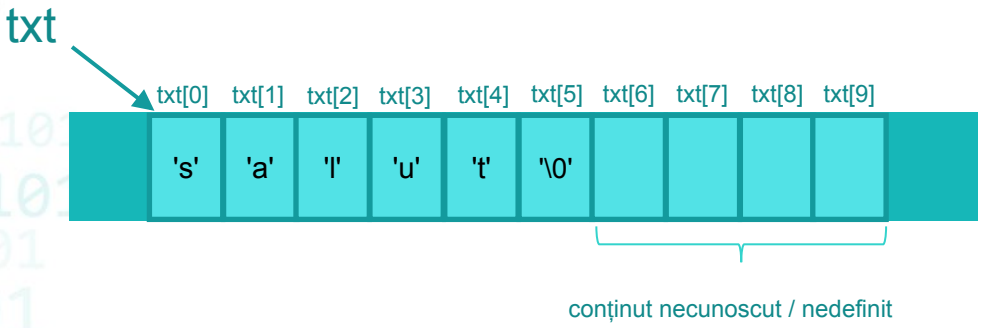
```
char *mystring3 = "salut";
```

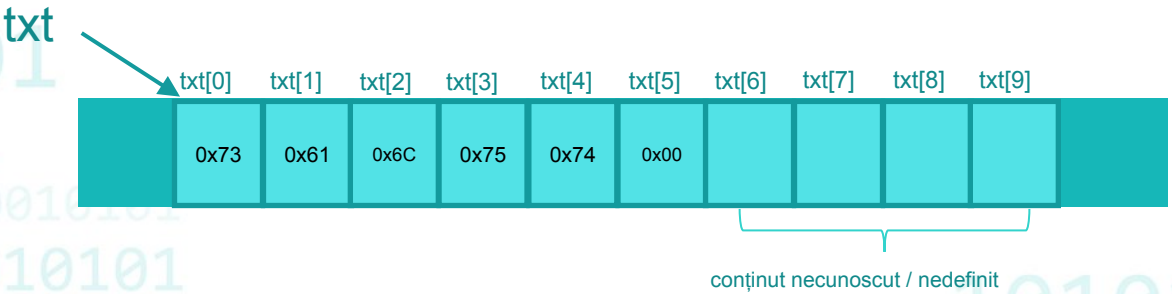
- Declarația și inițializarea variabilei mystring3 NU este același lucru cu mystring1 și mystring2
- mystring3 – variabilă de tip pointer ce se inițializează cu o constantă ce se află socată în zona de cod (.code).
- mystring3 va avea valoarea unei zone de memorie read-only – modifcarea șirului “salut” în acest caz va genera eroare de runtime (Segmentation Fault)

Șiruri de caractere

Reprezentarea în memorie

- `char txt[10] = "salut";`





Șiruri de caractere

Observații

- string-ul vid (gol): **"" –** ocupă un octet, conține doar caracterul NULL \0
- declarația `char t[3] = "tx";` - este corectă dar declarația `t[3] = "txt";` - nu este corectă, nu rămâne loc pentru \0
- declarație `char t[] = "txt";` va aloca static și inițializa pe *t* pe o dimensiune de 4 bytes (txt + \0), este echivalent cu `char t[4] = "txt";` Ⓟ accesarea lui `t[4], t[5],`... etc va produce depășire
- declarațiile (1) `char t[] = "text"` și (2) `char *p = "text"` nu sunt echivalente
  - declarația (1) aloca static pe *t* la 5 bytes și inițializează această locație cu constanta “text”
  - declarația (2) nu aloca nimic, asignează variabilei de tip adresa (pointer) la char **p** valoarea adresei constantei “text”. Această constantă se află în zona de cod (.code sau .text)

Șiruri de caractere

Funcții standard de citire șiruri de caractere

- Funcțiile scanf și printf cu directiva de formatare %s

```c
#include <stdio.h>

int main(void)
{
      char text[30];
      printf ("Scrieti un sir:");
      scanf ("%s", text);
      printf ("Sirul introdus este: %s\n", text);
      return 0;
}
```

- apelul scanf de mai sus citește de la stdin un șir de caractere până întâlnește un spațiu alb, EOF sau linie nouă iar la sfârșit adaugă caracterul '\0'
- la apelul *scanf* – se transmite variabila text fără **&**. De ce? *text* este deja o adresa, este un pointer, fiind un tablou, deci nu este necesară o transmitere cu adresa
- Apelul scanf este funcțional și corect dar este extrem de periculos – utilizatorul poate introduce **oricâte** caractere – poate duce la buffer overflow (depășirea zonei de memorie alocate). Când ? atunci cand introduce mai mult de 30 de caractere

Șiruri de caractere

Funcții standard de citire șiruri de caractere

- Citire sigură folosind scanf

```
#include <stdio.h>

int main(void)
{
     char text[30];
     printf ("Scrieti un sir:");
     scanf ("%29s", text);
     printf ("Sirul introdus este: %s\n", text);
     return 0;
}
```

- în acest caz citirea este sigură – prin folosirea unei dimensiuni în plus la %s – nu se poate ajunge la buffer overflow
- dimensiunea este cu 1 mai mică decât memoria alocată ☾ este necesar să se rezerve loc pentru '\0' !!!!!!
- scanf va citi maxim atâtea caractere câte sunt specificate în directiva de formatare – se va opri
  - când ajunge la numărul maxim
  - când întâlnește un spațiu
  - când intâlnește EOF (end of file) – nu mai are de unde citi

Șiruri de caractere

Funcții standard de citire șiruri de caractere

- Citire tablou de caractere fără \0 cu scanf – nu se citește string ci tablou de caractere

```c
#include <stdio.h>

int main(void)
{
    char text[80];
    scanf("%80c", text);
}
```

- în acest caz, citirea este sigură dar nu se citește un string ci un tablou de caractere, nu se adaugă \0 – I se spune lui scanf să citească maxim 80 de caractere și nu un string de maxim 80 caractere
- asupra lui text nu se pot efectua operații cu string-uri (text NU E STRING) – deci nu se poate folosi nici printf cu directiva %s

Șiruri de caractere

Funcții standard de citire șiruri de caractere

- Funcția gets: `char *gets(char *s);`
  - citește de la stdin un șir de caractere până la EOF sau linie nouă iar la sfârșit adaugă caracterul '\0'
  - este extrem de periculoasă – utilizatorul poate introduce **oricâte** caractere – se poate ajunge la buffer overflow prin depășirea zonei de memorie alocate
  - nu se recomandă utilizarea acestei funcții fiind extrem de nesigură (nici nu se mai poate compila)

Șiruri de caractere

Funcții standard de citire șiruri de caractere

- Funcția fgets: `char *fgets(char *s, int size, FILE *stream);`
  - citește un șir de caractere până la EOF sau linie nouă iar la sfârșit adaugă caracterul '\0'
  - citește maxim size-1 caractere pe care le stochează în zone de memorie referită de *s*
  - adaugă la sfârșit caracterul \0 (motivul pentru care citește size-1 caractere)
  - pentru a citi de la standard input se pune **stdin** la argumentul **stream**
  - dacă se oprește pentru ca a întâlnit linie nouă adaugă și caracterul linie nouă \n
  - returnează *s* dacă s-a citit cu succes minim un caracter sau NULL dacă a apărut o eroare și nu s-a citit nici un caracter

```
#include <stdio.h>

int main(void)
{
  char text[20];
  printf ("Scrieti o linie: ");
  if (fgets(text, 20, stdin) != NULL)
    {
      printf ("Linia: %s\n", text);
    }
  else
    {
      printf ("error\n");
    }
  return 0;
}
```

Șiruri de caractere

Funcții standard de prelucrare șiruri de caractere (string)

- Noțiuni preliminare
  - toate funcțiile standard de prelucrare de string-uri în C (cu puține excepții) se așteaptă ca șirul de caractere să aibă la sfârșit terminatorul NULL (caracterul \0)
  - dacă nu există caracterul NULL la sfârșitul șirului de caractere – funcțiile vor itera și vor căuta în toată memoria disponibilă până vor găsi un caracter NULL Ⓟ coruperea memorie prin depășire de buffer și buffer overflow - extrem de nesigur și periculos
  - funcțiile standard de prelucrare string-uri sunt declarate în header-ul <string.h>
  - funcțiile standard de prelucrare string-uri sunt în general implementate cu pointeri
  - toate informațiile despre aceste funcții se găsesc în paginile de manual (în Linux folosind comanda `man`)
  - este responsabilitatea programatorului să folosească funcțiile standard de prelucrare de string-uri doar asupra unor șiruri de caractere terminate cu \0 – funcții nu au vreo măsură de siguranță
  - funcțiile de prelucrare de string-uri reprezintă un risc dacă nu sunt folosite corect conform documentației
  - nu este permis ca zonele de memorie ale stringurilor implicate in apelul funcțiilor de prelucrare să se suprapună

Șiruri de caractere

Funcții standard de prelucrare șiruri de caractere (string)

- Funcția strlen: `size_t strlen(const char *s);`
  - primește ca parametru un string și returnează dimensiunea acestuia: numărul de caractere până la caracterul \0
  - este total diferită de operatorul sizeof – acesta returnează dimensiune zonei de memorie (alocată static) a argumentului
  - nu verifică transmiterea unui pointer NULL – programul va crăpa cu segmentation fault – se va încerca citirea de la adresa 0
  - tipul `size_t` este definit în <stddef.h> ca fiind un număr întreg fără semn (dimensiunea – dependentă de arhitectură)

```c
#include <stdio.h>
#include <string.h>

int main(void)
{
  char t[64] = "text";
  int n = 0;
  n = strlen(t);
  printf ("strlen - %d\n", n);
  n = sizeof(t);
  printf ("sizeof - %d\n", n);
  return 0;
}
```

```
valy@staff:~/teaching$ ./p
strlen - 4
sizeof - 64
valy@staff:~/teaching$
```

Șiruri de caractere

Funcții standard de prelucrare șiruri de caractere (string)

- Funcția strcpy: `char *strcpy(char *dest, const char *src);`
  - copiază string-ul *src* până la \0 în zona de memorie referită de *dest* și pune \0 la final
  - nu verifică (și nici nu are cum) dacă există suficient loc în *dest* pentru a copia conținutul lui *src*
  - nu verifică transmiterea unui pointer NULL nici pentru *src* nici pentru *dest* – programul va crăpa cu segmentation fault – se va încerca citirea/scrierea de la adresa 0
  - nu este permisă suprapunerea zonelor *dest* si *src*

```c
#include <stdio.h>
#include <string.h>

int main(void)
{
  char text[] = "ana are mere";
  char text2[128] = "text";
  printf ("%s\n", text2);
  strcpy(text2, text);
  printf ("%s\n", text2);
  return 0;
}
```

```
valy@staff:~/teaching$ ./p
text
ana are mere
valy@staff:~/teaching$
```

Șiruri de caractere

Funcții standard de prelucrare șiruri de caractere (string)

- Funcția strcat: `char *strcat(char *dest, const char *src);`
  - concatenează, adaugă la finalul lui dest conținut lui src și pune \0 la final
  - dest trebuie să se termine cu \0 (dar nu se verifică)
  - nu verifică (și nici nu are cum) dacă există suficient loc în *dest* pentru concatena conținutul lui *src*
  - nu verifică transmiterea unui pointer NULL nici pentru *src* nici pentru *dest* – programul va crăpa cu segmentation fault – se va încerca citirea/scrierea de la adresa 0
  - nu este permisă suprapunerea zonelor *dest* si *src*

```c
#include <stdio.h>
#include <string.h>

int main(void)
{
  char text[128] = "ana are mere";
  char text2[128] = " si pere";
  printf ("%s\n", text);
  strcat(text, text2);
  printf ("%s\n", text);
  return 0;
}
```

```
valy@staff:~/teaching$ ./p
ana are mere
ana are mere si pere
valy@staff:~/teaching$
```

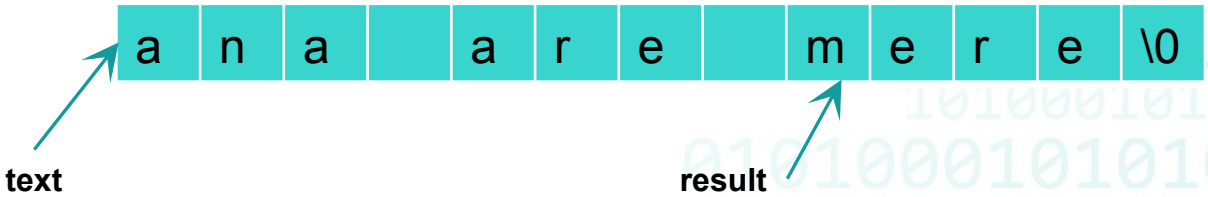
Șiruri de caractere

Funcții standard de prelucrare șiruri de caractere (string)

- Funcția strchr: `char *strchr(const char *s, int c);`
  - caută prima apariție a caracterului c în string-ul s și returnează un pointer la prima aparție a caracterului c în s sau NULL dacă nu există caracterul c în string-ul s
  - nu returnează pointerul într-o zonă de memorie nouă, diferită de s ci din zona de memorie ce conține string-ul s
  - nu verifică transmiterea unui pointer NULL programul va crăpa cu segmentation fault – se va încerca citirea de la adresa 0

```c
#include <stdio.h>
#include <string.h>

int main(void)
{
  char text[128] = "ana are mere";
  char *result = NULL;
  result = strchr(text, 'm');
  if (result == NULL)
    {
      printf ("not found\n");
    }
  else
    {
      printf ("found: %s\n", result);
    }
  return 0;
}
```

```
valy@staff:~/teaching$ ./p
found: mere
valy@staff:~/teaching$
```



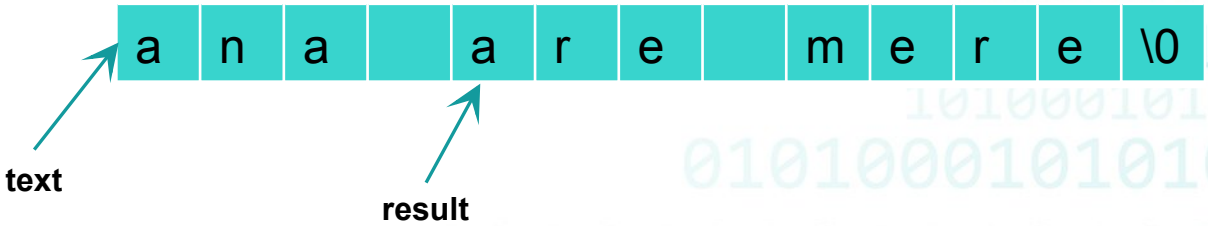
Șiruri de caractere

Funcții standard de prelucrare șiruri de caractere (string)

- Funcția strstr: `char *strstr(const char *haystack, const char *needle);`
  - caută prima apariție a stringului *needle* în stringul haystack și returnează un pointer la această prima apariție din haystack sau NULL daca nu găsește
  - nu returnează pointerul într-o zonă de memorie nouă, diferită de *haystack* ci din zona de memorie ce conține string-ul *haystack*
  - nu verifică transmiterea unui pointer NULL programul va crăpa cu segmentation fault – se va încerca citirea de la adresa 0

```c
#include <stdio.h>
#include <string.h>

int main(void)
{
  char text[128] = "ana are mere";
  char *result = NULL;
  result = strstr(text, "are");
  if (result == NULL)
    {
      printf ("not found\n");
    }
  else
    {
      printf ("found: %s\n", result);
    }
  return 0;
}
```

```
valy@staff:~/teaching$ ./p
found: are mere
valy@staff:~/teaching$
```



Șiruri de caractere

Funcții standard de prelucrare șiruri de caractere (string)

- Funcția strcmp: `int strcmp(const char *s1, const char *s2);`
  - compară string-urile s1 și s2 si returnează un întreg ce reprezintă și are valorile:
    - 0 – dacă cele 2 string-uri sunt egale
    - > 0 – dacă stringul *s1* este “mai mare” (nu mai lung !!) decât stringul *s2*
    - < 0 – dacă stringul s1 este “mai mic” (nu mai scurt !!) decât stringul s2
  - se compară practic caracter cu caracter și se returnează diferența dintre caractere (ASCII id) când acestea nu sunt egale
  - nu verifică transmiterea unui pointer NULL programul va crăpa cu segmentation fault – se va încerca citirea de la adresa 0

```c
#include <stdio.h>
#include <string.h>

int main(void)
{
  char text[128] = "ana are mere";
  int result = 0;
  result = strcmp(text, "ana are dere");
  printf ("%d\n", result);
  result = strcmp(text, "ana are mere");
  printf ("%d\n", result);
  result = strcmp(text, "ana are pere");
  printf ("%d\n", result);
  return 0;
}
```

```
valy@staff:~/teaching$ ./p
9
0
-3
valy@staff:~/teaching$
```

Șiruri de caractere

Funcții standard de prelucrare șiruri de caractere (string)

- Funcția strncpy: `char *strncpy(char *dest, const char *src, size_t n);`
  - similar cu strcpy dar copiază maxim *n* caractere din *src* în *dest*
  - dacă în cele *n* caractere din src nu există caracterul \0 atunci acesta nu se adaugă la finalul lui *dest*
- Funcția strncat: `char *strncat(char *dest, const char *src, size_t n);`
  - similar cu strcat, dar concatenează cel mult *n* caractere din *src* și pune \0 la final
  - în *dest* trebuie să fie suficient spațiu să încapă cele *n* caractere plus \0
- Funcția strncmp: `int strncmp(const char *s1, const char *s2, size_t n);`
  - similar cu strcmp, dar se compară doar primele n caractere din s1 și s2

Șiruri de caractere

Funcții standard de manipulare a zonelor de memorie

- sunt declarate tot în <string.h>
- nu interpretează zonele de memorie implicate ca și stringuri
- sunt implementate în general cu pointeri
- toate informațiile despre aceste funcții se găsesc în paginile de manual (în Linux folosind comanda `man`)
- este responsabilitatea programatorului să folosească aceste funcții corect nu se fac teste de existență a zonelor de memorie, de pointeri NULL, etc

Șiruri de caractere

Funcții standard de manipulare a zonelor de memorie

- Funcția memset: `void *memset(void *s, int c, size_t n);`
  - setează *n* octeți din zona de memorie referită de *s* cu valoarea *c*
  - returnează un pointer către zona de memorie s
  - nu verifică dacă *s* este NULL sau dacă nu este alocat sau neinițializat

```c
#include <string.h>

int main(void)
{
  char tab[10];
  int tab_int[10];
  memset(tab, 0, 10);
  memset(tab_int, 0, 10 * sizeof(int));
  return 0;
}
```

Șiruri de caractere

Funcții standard de manipulare a zonelor de memorie

- Funcția memcpy: `void *memcpy(void *dest, const void *src, size_t n);`
  - copiază n octeți din zona de memorie referită de src în zona de memorie referită de dest
  - src este pointer const – funcția nu va putea modifica sub nici o forma zona referită de src
  - nu verifică dacă *dest,src* sunt NULL sau ne-alocați sau neinițializați
  - nu se permite ca cele 2 zone de memorie dest și src să se suprapună

```c
#include <string.h>

int main(void)
{
  char array1[5] = {0, 1, 2, 3, 4};
  char array2[10];
  memcpy(array2, array, 5);
  return 0;
}
```

- Funcția memmove: `void *memmove(void *dest, const void *src, size_t n);`
  - similar cu funcția memcpy dar se permite ca zonele de memorie să se suprapună

Șiruri de caractere

Funcții standard de manipulare a zonelor de memorie

- Funcția memcmp: `int memcmp(const void *s1, const void *s2, size_t n);`
  - compară zonele de memorie s1 și s2 si returnează un întreg ce reprezintă și are valorile:
    - 0 – dacă cele 2 zone de memorie sunt egale
    - > 0 – dacă zona de memorie *s1* este “mai mare” (nu mai lungă !!) decât zona de memorie *s2*
    - < 0 – dacă zone de memorie s1 este “mai mică” (nu mai scurtă !!) decât zona de memorie s2
  - se compară practic byte cu byte și se returnează diferența dintre bytes când acestea nu sunt egale
  - nu verifică transmiterea unui pointer NULL programul va crăpa cu segmentation fault – se va încerca citirea de la adresa 0
  - similar cu strcmp dar în acest caz cele 2 zone de memorie nu trebuie sa reprezinte string-uri terminate cu \0 ci niște zone de memorie oarecare

Șiruri de caractere

Conversii de tipuri între stringuri si tipuri de date numerice

- Stringul în C reprezintă un tablou de caractere terminat cu \0 Ⓟ nu se poate face o conversie directă între un string și un tip numeric (int, float, etc…)
- sunt necesare niște funcții specializate pentru a realiza conversii din/în stringuri
- Funcția atoi: `int atoi(const char *nptr);`
  - primește ca argument un string ce conține un număr în format text (in baza 10) și returnează un intreg ce conține valoarea din string
  - declarată în header-ul <stdlib.h>
  - se poate spune: convertește un string numeric în baza 10 într-un intreg
  - convertește până când întâlnește un caracter ce nu e digit, caracterul linie noua, spațiu, terminatorul \0

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
  char text[10];
  int nr = 0;
  printf ("introduceti un numar:");
  scanf("%9s", text);
  nr = atoi(text);
  nr = nr + 1;
  printf ("numarul incrementat este: %d\n", nr);
  return 0;
}
```

```
valy@staff:~/teaching$ ./p
introduceti un numar:17
numarul incrementat este: 18
valy@staff:~/teaching$
```

Șiruri de caractere

Conversii de tipuri între stringuri si tipuri de date numerice

- Funcția strtol: `long int strtol(const char *nptr, char **endptr, int base);`
  - primește ca argument un string (*nptr*) ce conține un număr în format text, baza *base* și returnează un intreg ce conține valoarea din string în baza de numerație specificată
  - conversia se termină la primul caracter ce nu este digit în baza specificată, caracter spațiu, linie noua sau \0
  - argumentul endptr reprezintă o adresa a unei locații de tip sir de caractere (char*) și dacă nu este NULL atunci strtol va scrie acolo adresa primului caracter ce nu corespunde formatului

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  char text_dec[]="156320";
  char text_hex[]="A09BD25";
  char text_bin[]="01001001";

  int n = 0;
  n = strtol(text_dec, NULL, 10);
  printf ("%d\n", n);
  n = strtol(text_hex, NULL, 16);
  printf ("%d - %X\n", n, n);
  n = strtol(text_bin, NULL, 2);
  printf ("%d\n", n);
  return 0;
}
```

```
valy@staff:~/teaching$ ./p
156320
168410405 - A09BD25
73
valy@staff:~/teaching$
```

Șiruri de caractere

Conversii de tipuri între stringuri si tipuri de date numerice

- Alte funcții similare de transformat din string într-un format numeric:

```
long long int strtoll(const char *nptr, char **endptr, int base);
long atol(const char *nptr);
long long atoll(const char *nptr);
double atof(const char *nptr);
double strtod(const char *nptr, char **endptr);
float strtof(const char *nptr, char **endptr);
long double strtold(const char *nptr, char **endptr);
```

Șiruri de caractere

Conversii de tipuri între stringuri si tipuri de date numerice

- Funcția sprintf: `int sprintf(char *str, const char *format, ...);`
  - similar cu funcția *printf* dar rezultatul nu este printat la stdout ci este scris în zona de memorie referită de *str* sub forma unui string terminat cu \0
  - este necesar ca zona de memorie referită de *str* să fie alocată static sau dinamic înainte de apelul sprintf
  - nu face verifică de existență a zone de memorie referită de *str*
  - zona de memorie referită de *str* trebuie să fie suficient de mare să încapă string-ul rezultat – nu se face nici un fel de verificare ☾ poate provoca buffer overflow
  - se aplică absolut aceleași reguli și directive de formatare ca și la *printf*
  - poate fi folosită ori pentru a converti dintr-un format numeric într-un string ori pentru realizarea unui string formatat
  - nu concatenează, nu scrie la sfârșit, nu păstrează nimic din zona referită de str, nu adaugă la început – suprascrie zona de memorie referită de *str*
  - retrunează numărul de caractere scrise în zona refererită de *str* fără \0 – practic returnează dimensiunea string-ului scris în zona referită de *str*

Șiruri de caractere

Conversii de tipuri între stringuri si tipuri de date numerice

- Exemple utilizare sprintf

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(void)
{
  const char *some_text="some text";
  char t[128];
  char *text = NULL;
  int n = 3;
  float f = 3.14;
  unsigned int x = 780;
  sprintf(t, "%d", x);
  sprintf(t, "%X", x);
  sprintf(t, "%08X", x);
  sprintf(t, "%f", f);
  if ((text = malloc(1024 * sizeof(char))) == NULL)
    {
      printf ("no more memory\n");
      return -1;
    }
  sprintf(text, "String-ul meu custom: un sir %s, un numar in hex %04X si un float %f\n", some_text, n, f);
  return 0;
}
```

Secțiunea X

Alocare dinamică

Alocare dinamică

Recapitulare

- alocarea variabilelor s-a făcut până acum static, la compilare
- memoria variabilelor globale se alocă static și fix la pornirea programului (la runtime)
- varibilele locale sunt pe stivă

- cu acestea toate nu se poate realiza un program dinamic, totul fiind limitat la memoria alocată la runtime

Alocare dinamică

Definiții

- procesul prin care, în mod dinamic, în timpul rulării unui program, se alocă blocuri de memorie ce sunt mai apoi referențiate prin pointeri
- blocurile de memorie alocate dinamic se află în zona de memorie heap
- blocurile se pot accesa doar prin pointeri
- pierderea referinței către un bloc duce la pierderea totală a oricărei legături cu acea zona
- blocurile de memorie alocate rămân alocate până cand programatorul le eliberează explicit
- evidența blocurilor de memorie alocată este ținută tot în heap
- se folosește atunci când nu se cunoaște de la început dimensiunea datelor ce vor fi prelucrate
- în limbajul C alocarea dinamică și zona de heap sunt gestionate de funcții dedicate din <stdlib.h>

```
void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

- tipul `size_t` este definit în <stddef.h> ca fiind un întreg fără semn (>=0) cu dim. dependentă de arch. hw

Alocare dinamică

Funcții dedicate și exemple

- Funcția malloc: `void *malloc(size_t size);`
  - alocă un bloc de memorie continuu de dimensiune size bytes (octeți). Blocul de memorie este neinițializat
  - returnează o adresă (de tip void *) către blocul de memorie alocat sau NULL dacă a apărut o eroare și blocul nu a fost alocat

```c
#include <stdio.h>
#include <stdlib.h>

#define LEN 16

int main(void)
{
  int *int_ptr = NULL;
  int *int_dyn_array = NULL;
  int_ptr = malloc(sizeof(int));
  int_dyn_array = malloc(LEN * sizeof(int));
  *int_ptr = 5;
  printf ("value of int_ptr: %d\n", *int_ptr);
  for (int i = 0; i < LEN; i++)
    {
      int_dyn_array[i] = i * i;
    }
  printf ("array values: ");
  for (int i = 0; i < LEN; i++)
    {
      printf ("%d ", int_dyn_array[i]);
    }
  printf ("\n");
  return 0;
}
```

```
valy@staff:~/teaching$ ./p
value of int_ptr: 5
array values: 0 1 4 9 16 25 36 49 64 81 100 121 144 169 196 225
valy@staff:~/teaching$
```

- am folosit *int_ptr* pentru a aloca un singur întreg în heap
  iar *intr_dyn_array* pentru a aloca un tablou ☾ se observă că nu există nici o diferență – este tot un pointer – contează cum interpretez ulterior
- malloc aloca un număr de octeți deci un tablou de *tip* este de dimensiune LEN * sizeof(tip)
- nu se obișnuiește să se inlocuiască numeric operatorul sizeof(…) codul pierde portabilitatea si compatibilitatea

Alocare dinamică

Funcții dedicate și exemple

- Funcția calloc: `void *calloc(size_t nmemb, size_t size);`
  - alocă un bloc de memorie continuu format din *nmemb* elemente fiecare de dimensiune *size*
  - blocul de memorie alocat va fi inițializat cu 0 (fiecare byte din acest bloc va avea valoarea 0)
  - returnează o adresă (de tip void *) către blocul de memorie alocat sau NULL dacă a apărut o eroare și blocul nu a fost alocat
  - este echivalent cu `malloc(nmemb * size);`

```c
#include <stdio.h>
#include <stdlib.h>

#define LEN 16
int main(void)
{
  int *int_dyn_array = NULL;
  int_dyn_array = calloc(LEN, sizeof(int));
  for (int i = 0; i < LEN; i++)
    {
      int_dyn_array[i] = i * i;
    }
  printf ("array values: ");
  for (int i = 0; i < LEN; i++)
    {
      printf ("%d ", int_dyn_array[i]);
    }
  printf ("\n");
  return 0;
}
```

```
valy@staff:~/teaching$ ./p
array values: 0 1 4 9 16 25 36 49 64 81 100 121 144 169 196 225
valy@staff:~/teaching$
```

- ca și funcționalitate linia cu calloc este identică următoare secvență de cod

```c
int *t;
t = malloc(LEN * sizeof(int));
memset(t, 0, LEN * sizeof(int));
```

- funcția *calloc* are un timp de execuție mult mai mare deoarece pe lângă alocarea dinamică a blocului de memorie mai are loc și inițializarea acestuia cu 0.
- se recomandă utilizarea acestei funcții doar atunci când este absolut necesar ca noul bloc de memorie să fie inițializat cu 0

Alocare dinamică

Funcții dedicate și exemple

- Funcția realloc: `void *realloc(void *ptr, size_t size);`
  - realocă un bloc de memorie continuu deja alocat, referențiat prin ptr la o noua dimensiune *size*
  - returnează o adresă (de tip void *) către blocul de memorie alocat sau NULL dacă a apărut o eroare și blocul nu a fost alocat
  - size poate să fie mai mare sau mai mic decât dimensiunea inițială

```c
#include <stdio.h>
#include <stdlib.h>

#define LEN 16

int main(void)
{
  int *int_dyn_array = NULL;
  int_dyn_array = calloc(LEN, sizeof(int));
  for (int i = 0; i < LEN; i++)
    {
      int_dyn_array[i] = i * i;
    }
  printf ("array values before realloc: ");
  for (int i = 0; i < LEN; i++)
    {
      printf ("%d ", int_dyn_array[i]);
    }
  printf ("\n");
  int_dyn_array = realloc(int_dyn_array, (LEN + 1) * sizeof(int));
  int_dyn_array[LEN] = 99;
  printf ("array values after realloc:  ");
  for (int i = 0; i < (LEN + 1); i++)
    {
      printf ("%d ", int_dyn_array[i]);
    }
  printf ("\n");
  return 0;
}
```

```
valy@staff:~/teaching$ ./p
array values before realloc: 0 1 4 9 16 25 36 49 64 81 100 121 144 169 196 225
array values after realloc:  0 1 4 9 16 25 36 49 64 81 100 121 144 169 196 225 99
valy@staff:~/teaching$
```

- realloc poate fi apelat și cu *ptr* fiind NULL. În acest caz, realloc devine echivalent cu malloc astfel:

```c
int *t = NULL;
t = realloc(t, LEN*sizeof(int));
```

devine echivalent cu:

```c
t = malloc( LEN * sizeof(int))
```

din exemplul precedent

Alocare dinamică

Funcții dedicate și exemple

- Funcția free: `void free(void *ptr);`
  - eliberează (șterge) zona de memorie din heap alocată referențiată prin adresa dată de ptr
  - după eliberarea zonei de memorie, funcția *free* nu modifică valoarea lui ptr (nici nu are cum !)

```c
#include <stdio.h>
#include <stdlib.h>

#define LEN 16

int main(void)
{
  int *int_dyn_array = NULL;
  int_dyn_array = calloc(LEN, sizeof(int));
  for (int i = 0; i < LEN; i++)
    {
      int_dyn_array[i] = i * i;
    }
  printf ("array values: ");
  for (int i = 0; i < LEN; i++)
    {
      printf ("%d ", int_dyn_array[i]);
    }
  printf ("\n");
  free(int_dyn_array);
  return 0;
}
```

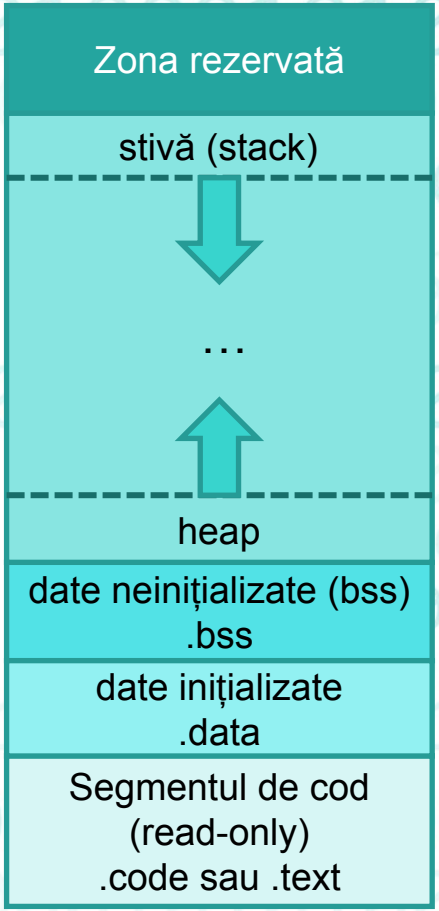
Alocare dinamică

Consecințe și observații

- Funcțiile prezentate alocă dinamic blocuri de memorie în zona heap – nu în altă parte !

```c
#include <stdio.h>
#include <stdlib.h>

int global;

int main(void)
{
  int local;
  static int local_static;
  int *to_local = &local;
  int *to_global = &global;
  int *p = NULL;
  p = malloc(sizeof(int));
  printf ("address of local \t\t\t%p\n", &local);
  printf ("address of local_static \t\t%p\n", &local_static);
  printf ("address of global \t\t\t%p\n", &global);
  printf ("adresss of pointer to_local: \t\t%p\n", to_local);
  printf ("address of pointer to_global: \t\t%p\n", to_global);
  printf ("adresss of dynamic alloc pointer: \t%p\n", p);
  return 0;
}
```

```
valy@staff:~/teaching$ ./p
address of local                        0x7fff11ce7444
address of local_static                 0x558fd037203c
address of global                       0x558fd0372040
adresss of pointer to_local:            0x7fff11ce7444
address of pointer to_global:           0x558fd0372040
adresss of dynamic alloc pointer:       0x558fd075a260
valy@staff:~/teaching$
```



Alocare dinamică

Consecințe și observații

- Nu se poate aloca, realoca un tablou/zona de memorie alocată static – în unele cazuri compilatorul generează eroare, în altele doar warning Ⓟ este extrem de periculos

```c
#include <stdlib.h>

int main(void)
{
  int tab[10];
  tab = malloc(20 * sizeof(int));
  free(tab);
  return 0;
}
```

```
dyn1.c: In function 'main':
dyn1.c:6:7: error: assignment to expression with array type
   tab = malloc(10 * sizeof(int));
       ^
valy@staff:~/teaching$
```

- Nu se poate șterge (elibera) cu free o zonă de memorie alocată static – compilator **poate** genera warning
  - se verifică doar dacă zona de memorie se află sau nu în heap (dacă nu, se poate genera warning)

```c
#include <stdlib.h>

int main(void)
{
  int tab[10];
  free(tab);
  return 0;
}
```

```
valy@staff:~/teaching$ gcc -Wall -o p dyn2.c
dyn2.c: In function 'main':
dyn2.c:6:3: warning: attempt to free a non-heap object 'tab' [-Wfree-nonheap-object]
   free(tab);
   ^~~~~~~~~
valy@staff:~/teaching$
```

Alocare dinamică

Consecințe și observații

- După apelul lui free zona de memorie referențiată de pointer este eliberată, ștearsă și nu mai există
- Ce se întâmplă cu valoarea adresei referită de pointer după free() ? pe scurt: depinde de compilator
  - valoarea rămâne aceeași, ca și cea de dinainte de free (deși blocul respecitv de memorie nu mai există)
  - se asignează o valoare random care poate să fie sau nu din spațiul de adrese al programului / sistemului
  - se asignează o valoare bine definită, clara, de obicei în afara spațiului de adrese (Visual Studio)
  - se asignează NULL (extrem de rar)

```c
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
  int *p = NULL;
  printf ("adresa lui p dupa init: \t%p\n", p);
  p = malloc(sizeof(int));
  printf ("adresa lui p dupa malloc: \t%p\n", p);
  free(p);
  printf ("adresa lui p dupa free: \t%p\n", p);
  return 0;
}
```

```
valy@staff:~/teaching$ ./p
adresa lui p dupa init:         (nil)
adresa lui p dupa malloc:       0x555891f18670
adresa lui p dupa free:         0x555891f18670
valy@staff:~/teaching$
```

Alocare dinamică

Consecințe și observații

- apelarea lui free() pe un pointer nealocat, NULL sau după ce a fost deja șters (free) duce la un comportament nedefinit

```c
#include <stdlib.h>
int main(void)
{
  int *p;
  free(p);
  p = NULL;
  free(p);
  p = malloc(sizeof(int));
  free(p);
  free(p);
  return 0;
}
```

- compilatorul nu generează nici un fel de warning – eventual la primul free fiind folosire de variabilă neinițializată
- comportamentul în toate situațiile este nedefinit și duce la coruperea memoriei
- extrem de periculos – de evitat

Alocare dinamică

Consecințe și observații

- pierderea referinței către un bloc de memorie – nu se mai poate recupera – blocul rămâne în heap fără vreo posibilitate de a fi accesat

```c
#include <stdlib.h>
int main(void)
{
    int *p = NULL;
    int x = 0;
    p = malloc(sizeof(int));
    p = &x;
    // blocul initial referit de p este pierdut
    return 0;
}
```

Alocare dinamică

Consecințe și observații

- **memory leaks** – blocuri de memorie (de obicei din heap) ce rămân alocate la terminarea programului (nu au fost eliberate de programator înainte de eliberarea programului)
- reprezintă un risc mare de acumulare de memorie până la blocarea sistemului de calcul
- reprezintă o sursă de bug-uri
- este foarte greu de depanat, dar este ușor de evitat
- exemplu de generare de memory leaks:

```c
#include <stdlib.h>
int main(void)
{
    int *p = NULL;
    p = malloc(sizeof(int));
    return 0;
}
```

Alocare dinamică

Consecințe și observații

- **memory leaks** (cont.)
- se pot detecta ușor folosind programul **valgrind** (un instrument de debug și evaluare de cod)
- **valgrind** se apelează în linie de comanda și primește ca argument calea programului de depanat

```
valy@staff:~/teaching$ valgrind ./p
==12696== Memcheck, a memory error detector
==12696== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==12696== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==12696== Command: ./p
==12696==
==12696==
==12696== HEAP SUMMARY:
==12696==     in use at exit: 4 bytes in 1 blocks
==12696==   total heap usage: 1 allocs, 0 frees, 4 bytes allocated
==12696==
==12696== LEAK SUMMARY:
==12696==    definitely lost: 4 bytes in 1 blocks
==12696==    indirectly lost: 0 bytes in 0 blocks
==12696==      possibly lost: 0 bytes in 0 blocks
==12696==    still reachable: 0 bytes in 0 blocks
==12696==         suppressed: 0 bytes in 0 blocks
==12696== Rerun with --leak-check=full to see details of leaked memory
==12696==
==12696== For counts of detected and suppressed errors, rerun with: -v
==12696== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Alocare dinamică

Consecințe și observații

- foarte utilă pentru a returna o valore dintr-o funcție

```
#include <string.h>

char *paritate(int n)
{
            char *rezultat = NULL;
            rezultat = malloc(8 * sizeof(char));
            if ((n % 2) == 0)
            {
                        strcpy(rezultat, "par");
            }
            else
            {
                        strcpy(rezultat, "impar");
            }
            return rezultat;
}
```

Alocare dinamică

Consecințe și observații

- umplerea și fragmentarea zonei de heap – fragmentarea – situația în care există multe blocuri alocate și multe zone libere ce nu sunt continue – se poate ajunge la situația în care deși mai există memorie liberă aceasta nu mai poate fi alocată
- nu se face automat defragmentarea memoriei
- alocarea dinamică este ineficientă ca timp – nu este bine să se folosească în exces
- realloc(…) este cea mai ineficientă operație – se folosește doar dacă nu există altă soluție

Secțiunea XI

Tipuri de date complexe

Tipul enumerare

Tipul enumerare

Definiții și exemple

- Tipul enumerare – tip de date folosit pentru a asocia constante (nume) simbolice unui șir de valori numerice cu scopul de a facilitata scrierea unui cod cu un grad mai mare de lizibilitate
- Sintaxă: **enum** [identificator] { lista-constante } [lista-decl] **;**
  - identificator reprezintă numele tipului enum (poate să lipsească – tip anonim)
  - lista-constante – constantele simbolice separate prin virgulă
  - lista-decl – listă declarații de variabile (poate să lipsească)
  - trebuie sa fie prezent minim un identificator ca nume sau un identificator în lista de declarații de variabile. Dacă lipsesc ambele nu are sens
- tipul enum reprezintă de fapt un tip întreg iar constantele simbolice sunt valorile
- implicit sirul constantelor simbolice începe de la 0 și cresc cu pasul 1
- se poate modifica explicit valoarea unei constante punând semnul egal dupa ea, apoi valoarea și apoi virgulă
- numele de constante trebuie să fie unice pe tot parcursul programului, nu pot exista 2 constante simbolice cu același nume
- se poate folosi direct ca și intreg
- se poate defini și ca pointer
- keyword: **enum**

```c
#include <stdio.h>

enum MyEnum
  {
   Luni,
   Marti,
   Miercuri,
   Joi,
   Vineri,
   Sambata,
   Duminica,
  };

int main(void)
{
  enum MyEnum zi_saptamana;
  int n = 0;
  zi_saptamana = Joi;
  printf ("Dati o zi: (0-7):");
  scanf("%d", &n);
  if ((n >= 0) && (n <= 6))
    {
      zi_saptamana = (enum MyEnum)n;
    }
  switch (zi_saptamana)
    {
    case Luni:
    case Marti:
    case Miercuri:
    case Joi:
    case Vineri:
      {
        printf ("Weekday %d\n", zi_saptamana);
        break;
      }
    case Sambata:
    case Duminica:
      {
        printf ("Weekend %d\n", zi_saptamana);
        break;
      }
    default:
      {
        printf ("unknown\n");
      }
    }
  return 0;
}
```

Tipul enumerare

Definiții și exemple

```c
#include <stdio.h>

enum MyEnum
{
	Luni = 4,
	Marti,		// 5
	Miercuri,	// 6
	Joi,		// 7
	Vineri,		// 8
	Sambata,	// 9
	Duminica,	// 10
};

enum
{
	val_1,
	val_2,
}x;

int main(void)
{
	enum MyEnum zi_saptamana;
	zi_saptamana = Vineri;
	int n = Vineri + 2;
	x = val_2;
	x = 0;
	return 0;
}
```

Tipul struct (structura)

Tipul struct

Definiții și exemple

- Tipul structură – tip de date folosit pentru a grupa mai multe elemente (declarații) de tipuri de date diferite în scopul de a facilita dezvoltarea de programe mai complexe și cu un grad mai mare de lizibilitate
- Sintaxă: `struct [identificator] { lista-campuri } [lista-decl] ;`
  - identificator reprezintă numele tipului struct (poate să lipsească – tip anonim)
  - lista-câmpuri – declarații de câmpuri (similar cu declarațiile de variabile)
  - lista-decl – listă declarații de variabile (poate să lipsească)
  - trebuie sa fie prezent minim un identificator ca nume sau un identificator în lista de declarații variabile ☾ lipsesc ambele – nu are sens
- elementele unei structuri se numesc câmpuri (fields)
- numele de câmpuri pot fi refolosite în structuri diferite dar nu și în cazul aceleiași structuri
- declarația unui câmp se termină cu punct-și-virgulă **;**
- declarația unei structuri se termină cu punct-și-virgulă chiar dacă lipsește lista-decl
- keyword: **struct**
- declararea unei variabile de o structură definită anterior se face după sintaxa:

```
struct nume_struct lista_identif_var;
```

- accesul la câmpurile unei structuri se face folosind semnul punct (.) după variabila de tipul structurii (operatorul selecție)

Tipul struct

Definiții și exemple

```c
#include <stdio.h>
#include <string.h>
enum MyEnum
{
	val_1,
	val_2,
};

struct MyStruct
{
	int n;
	char text[10];
	float f;
	enum MyEnum e;
};


int main(void)
{
	struct MyStruct str;
	str.n = 4;
	str.f = 3.14;
	str.e = val_2;
	strcpy(str.text, "Salut");
	struct MyStruct str2;
	str2.n = str.n;
	int n;
	n = str.n + 1;
	return 0;
}
```

Tipul struct

Dimensiunea unei structuri

- Dimensiunea unei structuri este dată de dimensiunea cumulată a tuturor câmpurilir plus spații goale de memorie pentru realizarea unei alinieri eficiente în memorie (de obicei se aliniază la multiplu de 4 bytes)

```c
#include <stdio.h>
struct MyStruct
{
  int n;
  char text[10];
};

int main(void)
{
  struct MyStruct str1;
  printf ("%ld\n", sizeof(str1.n));
  printf ("%ld\n", sizeof(str1.text));
  printf ("%ld\n", sizeof(struct MyStruct));
  return 0;
}
```

```
valy@staff:~/teaching$ ./p
4
10
16
valy@staff:~/teaching$
```

Tipul struct

Dispunerea câmpurilor unei structuri

- Câmpurile unei structuri se alocă în memorie continuu, unul după celalalt dar aliniat de obicei la dimensiunea câmpului in bytes (practic la sizeof(tip_camp) – între câmpuri este posibil să existe și spațiu liber de memorie (neutilizat) pentru a realiza alinierea
  - char, unsigned char 1 byte – aliniat la 1 byte
  - short – 2 bytes – aliniat la 2 bytes
  - int, unsigned int, uint32_t, int32_t 4 bytes – aliniat la 4 bytes
- Se poate obține deplasamentul unui câmp

1. prin diferență de pointeri între adresa câmpului și adresa unei variabile de tipul structurii

```c
#include <stdio.h>
struct MyStruct
{
  int n;
  char text[10];
  float f;
};

int main(void)
{
  struct MyStruct str1;
  printf ("deplasament n: %ld\n", (char*)&str1.n - (char*)&str1);
  printf ("deplasament text: %ld\n", (char*)&str1.text - (char*)&str1);
  printf ("deplasament f: %ld\n", (char*)&str1.f - (char*)&str1);
  printf ("dimensiune structura: %ld\n", sizeof(str1));
  return 0;
}
```

```
valy@staff:~/teaching$ ./p
deplasament n: 0
deplasament text: 4
deplasament f: 16
dimensiune structura: 20
valy@staff:~/teaching$
```

Tipul struct

Dispunerea câmpurilor unei structure (2)

2. prin utilizarea macro-ului `offsetof(tip_struct, nume_camp)` din `<stddef.h>`

```c
#include <stdio.h>
#include <stddef.h>
struct MyStruct
{
  int n;
  char text[10];
  float f;
};

int main(void)
{
  struct MyStruct str1;
  printf ("deplasament n: %ld\n", offsetof(struct MyStruct, n));
  printf ("deplasament text: %ld\n", offsetof(struct MyStruct,
text));
  printf ("deplasament f: %ld\n", offsetof(struct MyStruct, f));
  printf ("dimensiune structura: %ld\n", sizeof(str1));
  return 0;
}
```

```
valy@staff:~/teaching$ ./p
deplasament n: 0
deplasament text: 4
deplasament f: 16
dimensiune structura: 20
valy@staff:~/teaching$
```

Tipul struct

Inițializare

- O structură se inițializează dând valori între paraneteze acolade pentru câmpurile structurii, în ordinea apariției lor în declarație. Nu este obligatoriu să se inițializeze toate câmpurile dar nici nu se poate sări un câmp în inițializare

```
struct MyStruct
{
	int n;
	char text[10];
};

int main(void)
{
	struct MyStruct str3 = {78, "test"};
	return 0;
}
```

- Sintaxa de **inițializare nu** se poate folosi și pentru **asignare** de valori unei structure. Următoarea secvență de cod va genera eroare de compilare

```
struct MyStruct
{
	int n;
	char text[10];
};

int main(void)
{
	struct MyStruct str3;
	str3 = {65, "abc"};
	return 0;
}
```

```
valy@staff:~/teaching$ gcc -Wall -o p struct1.c
struct1.c: In function ‘main’:
struct1.c:10:10: error: expected expression before ‘{’ token
   str3 = {65, "abc"};
          ^
struct1.c:9:19: warning: variable ‘str3’ set but not used [-Wunused-but-set-variable]
   struct MyStruct str3;
                   ^~~~
valy@staff:~/teaching$
```

Tipul struct

Asignare

- Asignarea de valori pentru o structură se poate face doar prin asignarea de valori individuale pentru fiecare câmp în pare, respectând regulile pentru tipul respectiv

```
struct MyStruct
{
	int n;
	char text[10];
};

int main(void)
{
	struct MyStruct str1;
	str1.n = 123;
	strcpy(str1.text, "salut");
	struct MyStruct str2;
	return 0;
}
```

- Limbajul permite asignarea de valori între structuri de același tip (identice):

```
1.  struct MyStruct
2.  {
3.     int n;
4.     char text[10];
5.  };

6.  int main(void)
7.  {
8.     struct MyStruct str1;
9.     str1.n = 123;
10.    strcpy(str1.text, "salut");
11.    struct MyStruct str2;
12.    str2 = str1;
13.    return 0;
14. }
```

- Linia 12 are ca efect copierea întregii zone de memorie a variabilei str1 peste zona de memorie ocupată de variabila str2
- este echivalent cu:

```
memcpy(&str2, &str1, sizeof(str1));
```

Tipul struct

Comparare structuri

- Nu se pot aplica operatori logici asupra structurilor Ⓟ structurile nu se pot compara în mod direct și explicit
- Există 2 metode de comparare

1. comparare a două structuri prin compararea individuală a câmpurilor după anumiți algoritmi / necesități
2. prin folosirea funcției memcmp de comparare a două zone de memorie

```c
#include <stdio.h>
#include <string.h>
struct MyStruct
{
	int n;
	char text[10];
	float f;
};

int main(void)
{
	struct MyStruct str1;
	struct MyStruct str2;
	// some code
	if (memcmp(&str1, &str2, sizeof(str1)) == 0)
	{
		// cele 2 structuri sunt egale
	}
	else
	{
		// cele 2 structuri nu sunt egale
	}
	// some code
	return 0;
}
```

Tipul struct

Pointeri la structuri

- Structurile fiind un tip de date în limbajul C – acestea pot fi folosite ca tipuri de date pointeri – se pot declara pointeri la structuri
- se păsrează absolut toate elementele de teorie, observațiile și consecințele discutate la pointeri ☾ este doar un caz particular
- Exemplu:

```c
#include <stdlib.h>
#include <string.h>
struct MyStruct
{
	int n;
	char text[10];
};

int main(void)
{
	struct MyStruct *str1;
	str1 = malloc(sizeof(struct MyStruct));
	(*str1).n = 123;
	strcpy((*str1).text, "salut");
	struct MyStruct str2;
	str2 = *str1;
	return 0;
}
```

Tipul struct

Pointeri la structuri

- pentru a facilita scrierea codului s-a făcut următoarea convenție de echivalență

`(*p).camp` ⇔ `p->camp`

- codul precedent se rescrie astfel:

```c
#include <stdlib.h>
#include <string.h>
struct MyStruct
{
        int n;
        char text[10];
};

int main(void)
{
        struct MyStruct *str1;
        str1 = malloc(sizeof(struct MyStruct));
        str1->n = 123;
        strcpy(str1->text, "salut");
        struct MyStruct str2;
        str2 = *str1;
        return 0;
}
```

Tipul struct

Pointeri la structuri

- Operatorul ->
  - are un dublu rol simultan: de a accesa membru unei structuri și rol de dereferențiere
  - se poate folosi pentru a accesa membrul unei structuri declarate ca și pointer.
  - folosit în orice alt context generează eroare de compilare
  - este un operator unar

Clasă de precedență	Operator	Descriere	Tip operator	Asociativitate
1	. şi ->	Selecție membru (prin structură, respectiv pointer)	unar	stânga->dreapta

- Operatorii . și -> au precedență ridicată, rezultă următoarele afirmații

```
p->x++      <=>    (p->x)++
++p->x      <=>    ++(p->x)
*p->x       <=>    *(p->x)
*p->s++     <=>    *((p->s)++)
```

Tipul struct

Tablouri de structure. Structuri in functii

- Se pot defini tablouri de structuri - o organizare eficienta a datelor
- Se pot folosi tipuri de date structură ca și argumente la funcții sau ca si valoare returnată
- tablourile de structuri pot ajunge destul de mari - declarate pe stivă pot duce la stack overflow
- structurile transmise ca și parametru la funcții sau returnate ca și valoare de funcții pot fi uneori destul de mari - se poate ajunge ușor la stack overflow - se recomandă utilizarea pointerilor (eventual cu const la funcții dacă nu se dorește și modificarea valorilor zonelor de memorie)

Tipul struct

Structuri cu câmpuri pe biți

- Limbajul C permite specificarea dimensiunii în biți a unui câmp dintr-o structură. se poate atfel folosi pentru a defini câmpuri pe biți de dimensiuni date de utilizator
- sintaxă: `tip_camp nume_camp : dim_biti` sau `tip_camp : dim_biti`

```c
#include <stdio.h>
#include <stdint.h>

struct MyBitField
{
  uint16_t field1 : 4;
  uint16_t field2 : 2;
  uint16_t field3 : 6;
  uint16_t field_not_needed : 1;
  uint16_t field4 : 3;
};

int main(void)
{
  struct MyBitField f;
  f.field2 = 3;
  f.field2++;
  printf ("%d\n", f.field2);
  return 0;
}
```

Tipul union (uniune)

Tipul union

Definiții și exemple

- Tip de date similar cu tipul struct cu mențiunea ca nu se alocă memorie pentru toate câmpurile componente ci doar pentru cel mai mare dintre ele
- are aceeași sintaxă și utilizare ca și tipul de date struct
- Sintaxă: `union [identificator] { lista-campuri } [lista-decl] ;`
  - identificator reprezintă numele tipului union (poate să lipsească – tip anonim)
  - lista-câmpuri – declarații de câmpuri (similar cu declarațiile de variabile)
  - lista-decl – listă declarații de variabile (poate să lipsească)
  - trebuie sa fie prezent minim un identificator ca nume sau un identificator în lista de declarații variabile ☾ lipsesc ambele – nu are sens
- elementele unei uniuni se numesc câmpuri (fields)
- numele de câmpuri pot fi refolosite în structuri/union diferite dar nu și în cazul aceleiași structuri/union
- declarația unui câmp se termină cu punct-și-virgulă **;**
- declarația unei uniuni se termină cu punct-și-virgulă chiar dacă lipsește lista-decl
- keyword: **union**
- declararea unei variabile de o structură definită anterior se face după sintaxa:

```
union nume_union lista_identif_var;
```

- accesul la câmpurile unei uniuni se face folosind semnul punct (.) după variabila de tipul structurii (operatorul selecție)
- nu se alocă o zonă de memorie care să cuprindă toate câmpurile ci se alocă o zonă de memorie care să cuprindă cel mai mare câmp component declarat în union
- toate câmpurile impart aceeași zonă de memorie
- inițializarea, utilizarea, asignarea, restricțiile, utilizarea cu pointeri sunt identice cu cele din cazul structurilor

Tipul union

Exemplu

```
1.  #include <stdio.h>
2.  #include <stdint.h>
3.  #include <string.h>

4.  union MyUnion
5.  {
6.    uint32_t n;
7.    char text[4];
8.  };

9.  int main(void)
10. {
11.   union MyUnion u;
12.   printf ("size: %ld\n", sizeof(union MyUnion));
13.   strcpy(u.text, "abc");
14.   printf ("u.text = %s\n", u.text);
15.   printf ("u.n = %08X\n", u.n);
16.   return 0;
17. }
```

```
valy@staff:~/teaching$ ./p
size: 4
u.text = abc
u.n = 00636261
valy@staff:~/teaching$
```

- câmpurile *n* și *text* împart aceeași zonă de memorie de 4 bytes – se observă rezultatul lui sizeof
- prin linia 13 se scrie în zona comună de memorie – intrepretată de funcția strcpy ca și text (string) și se primeste valoarea ce compune string-ul Ⓟ 0x61 (‘a’) 0x62 (‘b’) 0x63 (‘c’) 0x00 (‘\0’) ☾ 0x00636261 (Intel little endian)
- prin linia 14 se intrerpretează aceeași zonă de memorie tot ca și string (prin printf)
- prin linia 15 se interpretează aceeași zona de memorie ca și unsigned int (uint32_t) tot prin printf
- zona de memorie este aceeași pentru toate câmpurile ☾ totul depinde de interpretare ☾ union permite un mecanism ușor de interpretare diferită a datelor dintr-o zonă de memorie

Tipuri de date utilizator

Tipuri de date utilizator

Definiții și exemple

- Limbajul C oferă posibilitatea ca utilizatorul să-și poată defini numele tipurilor de date utilizate – duce la o lizibilitate ridicată a codului și facilitează scrierea unor programe de o complexitate ridicată
- Sintaxă: `typedef definitie_tip identificator_nume_tip ;`
  - definitie_tip reprezintă orice definiție de tip (struct, union, enum) sau nume de tip deja existent
  - identificator_nume_tip – reprezintă numele noului tip de date
- tipul nou definit va putea fi apoi referit direct prin numele lui în orice declarație de variabilă/parametru al acelui tip
- tipul nou definit poate fi folosit ca pointer de acel tip
- tipul nou definit poate fi folosit ca paremetru local al unei functii si/sau ca tip de date de return
- tipul nou definit poate fi folosit în definiția unui alt tip nou
- poate fi si de tip tablou

Tipuri de date utilizator

Definiții și exemple

```c
#include <stdio.h>
#include <stdint.h>
#include <string.h>

typedef unsigned int MyUnsignedIntType;
typedef uint32_t MySpecialType;

typedef union MyUnion
{
  uint32_t n;
  char text[4];
}MyUnionType;

typedef struct
{
  uint16_t x;
  MyUnionType t;
  double d;
}MyStructType;

int main(void)
{
  MyUnsignedIntType n;
  MySpecialType special;
  MyUnionType myunion;
  MyStructType str;
  MyStructType *pstr;
  pstr = &str;
  pstr->t.n = 3;
  pstr->x = 16;
  return 0;
}
```

Secțiunea XII

Coduri de eroare standard

Coduri de eroare

Standard errno

- Orice program scris în limbajul C “moștenește” o variabilă globală, denumită errno, ce va conține codul de eroare al ultimului apel a unei funcții din bibliotecile standard
- variabila errno este declarată în header-ul <errno.h>: `extern int errno;`
- informații despre variabila errno și despre codurile de eroare suportate se găsește în pagina de manual:

  `man errno`
- variabila errno va lua valoarea 0 atunci când ultimul apel a funcției standard a fost efectual cu succes și o valoare diferită de 0 ce va specifica codul de eroare apărut
- se poate folosi în programe prin includerea header-ului si verificarea valorii

```c
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

int main(void)
{
  char text[] = "1234";
  strtol(text, NULL, -1);
  printf ("%d\n", errno);
  return 0;
}
```

```
valy@staff:~/teaching$ ./p
22
valy@staff:~/teaching$
```

Coduri de eroare

Standard errno

- Există metode de utilizare mai ușoară a valorii lui errno
  - prin constante de preprocesor definite în bibliotecile standard și explicate în pagina de manual `man errno`
  - prin funcții puse la dispoziție de biblioteca standard
- Funcția perror: `void perror(const char *s);`
  - scrie un string “human readable” la ieșirea standard de eroare (stderr) ce explică ultima valoare a lui errno
  - primește ca argument un string la care va fi concatenat mesajul de eroare după semnul :
  - poate fi și NULL și astfel se va printa doar mesajul de eroare
  - pentru folosirea lui perror nu este necesară includerea header-ului <errno.h>

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  char text[] = "1234";
  strtol(text, NULL, -1);
  perror("my error");
  return 0;
}
```

```
valy@staff:~/teaching$ ./p
my error: Invalid argument
valy@staff:~/teaching$
```

Coduri de eroare

Standard errno

- Funcția strerror: `char *strerror(int errnum);`
  - returnează un string “human readable” ce explică o valoare a lui errno primită prin argumentul errnum

```c
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
  char text[] = "1234";
  strtol(text, NULL, -1);
  char *error_string;
  error_string = strerror(errno);
  printf ("%s\n", error_string);
  return 0;
}
```

```
valy@staff:~/teaching$ ./p
Invalid argument
valy@staff:~/teaching$
```

Coduri de eroare

Funcția exit(…)

- `void exit(int status);`
  - determină terminarea programului cu returnarea codului specificat ca argument
  - un proram când își termină execuția va returna către sistemul de operare ce l-a apelat un cod de eroare ori prin valoarea de return a funcției main(…) ori prin argument la apelul funcției exit(…)
  - determină terminarea programului în momentul apelului fără a mai continua execuția
  - se poate folosi împreuna cu funcțiile de verificare de eroare
  - declarată în headerul `<stdlib.h>` cu pagina de manual: `man 3 exit`

```c
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
  char text[] = "1234";
  strtol(text, NULL, -1);
  if (errno != 0)
    {
      perror("my error");
      exit(-1);
    }

  printf ("message\n");
  return 0;
}
```

```
valy@staff:~/teaching$ ./p
Invalid argument
valy@staff:~/teaching$
```

Secțiunea XIII

Fluxuri de date – fișiere
Argumente în linie de comandă

Fișiere

Definiții

- Fișierul – este o formă elementară de organizare a datelor și reprezintă o secvență, un flux de date ce sunt stocate de obicei pe un mediu de stocare non-volatil
- este referențiat printr-o cale absolută sau relativă
- în fișiere, datele sunt stocate în mod binar și nestructurat – o eventuală structurare și interpretare a datelor este dată de programul ce folosește aceste fișiere
- fișierele sunt gestionate exclusiv de către sistemul de operare și accesul la acestea este controlat de sistemul de operare
- sistemul de operare oferă anumite funcții ce pot fi folosite pentru accesul la fișiere din codul programelor
- Limbajul C, prin bibliotecile standard, oferă funcții specializate pentru accesul la fișiere – operații de bază:
  - deschiderea unui fișier – operațiunea prin care se obține accesul la un fișier pentru citire/scriere
  - scriere/citire fișier – operațiunea prin care se scrie/citește efectiv în/din fișier – pentru a fi citit/scris un fișier trebuie în prealabil deschis
  - închiderea unui fișier – operațiunea prin care un fișier este eliberat de program – este absolut necesar ca un fișier să fie închis de către program după ce a terminat cu toate operațiunile de scriere/citire
- Limbajul C oferă funcții specializate pentru accesul la fișiere atât pentru fișiere ce sunt interpretate ca și text cât și pentru fișiere ce nu sunt interpretate, oferă acces de citire/scriere binară a fișierelor

Fișiere

Deschiderea unui fișier

- Este prima operațiune necesară înainte de orice prelucrare asupra fișierului
- Funcția fopen: `FILE *fopen(const char *pathname, const char *mode);`
  - primește ca argumentul 1 un string ce reprezintă calea relativă sau absolută către fișier
  - primește ca argumentul 2 un string ce reprezintă modul în care să deschidă fișierul. String-ul este format din următoarele caractere (puse în orice ordine):
    - r (read): fișierul se deschide pentru citire – fișierul trebuie să existe, dacă nu există rezultă eroare
    - w (write): deschidere pentru scriere – dacă fișierul nu există atunci este creat – se va trunchia conținut precedent
    - a (append): deschidere pentru scriere – dacă fișierul nu există atunci este creat – se va scrie la capătul fișierului, se adaugă la sfârșit
    - + : permite și celalalt mod pe lângă cel specificat (r+ ☾ rw , w+ ☾ rw)
    - b: deschide fișierul în mod binar (implcit este în mod text) – în mod binar nu se face nici un fel de interpretare asupra conținutului din fișier. în mod text se consideră că există doar text în fișier și se pot face anumite interpretări/transformări
  - returnează un pointer către o structură de tip FILE a cărei câmpuri nu sunt vizibile –
  - pointer-ul către structura FILE returnat va reprezenta un mod unic de identificare a fișierului deschis în cadrul programului până la sfârșitul programului sau închiderea explicită a fișierului. Acest pointer returnat va fi folosit apoi ca și argument pentru alte funcții specializate ce vor efectua operații asupra fișierului
  - returnează NULL în caz de eroare (fișier inexistent, permission denied,… etc)
  - deschide un fișier și returnează un pointer către o structură FILE ce va identifica în mod unic fișierul până la inchiderea acestuia

Fișiere

Închiderea unui fișier

- este ultima operațiune necesară după orice prelucrare asupra fișierului
- este total greșit ca programatorul să nu închidă fișierul după utilizare, deși la sfârșitul programului, după execuția funcției main() sistemul de operare va închide el toate fișierele găsite deschise
- Funcția fclose: `int fclose(FILE *stream);`
  - primește ca argument un pointer de tip FILE ce reprezintă un fișier deschis în prelabil cu funcția fopen
  - retrunează 0 dacă fișierul a fost închis cu succes sau -1 (EOF) dacă este o eroare
  - fclose are ca efect nu doar închiderea fișierului dar și scrierea efectivă a datelor în fișier. Sistemul de operare nu scrie datele fizic imediat în fișier ci încearcă optimizarea scrierilor. Apelul lui fclose determină si operația de flush prin care cere sistemului de operare scrierea efectivă a datelor

Fișiere

Operațiuni de citire scriere

- se realizează doar asupra fișierelor ce au fost deschise în prealabil cu fopen
- nu se pot realiza asupra unor fișiere ce au fost închise în prealabil cu fclose
- se realizează prin intermediul unor funcții specializate
- în momentul deschiderii unui fișier se asociază fișierului și un cursor (care la început are valoarea 0)
- cursorul are rolul de a prezenta poziția curentă în fișier
- o operațiune de citire din fișier va obține un număr de octeți din fișier și va avansa cursorul cu numărul de octeți obținuți (citiți)
- la o operațiune de citire, când cursorul a ajuns la capătul fișierului se spune că s-a ajuns la EOF (end of file) și din acest punct nu se mai poate citi (nu mai sunt date disponibile)
- la o operațiune de scriere, după ce s-a scris un număr de octeți în fișier, va avansa cursorul cu numărul de octeți scriși în fișier
- nu există cursori diferiți pentru scriere și citire ci un singur cursor care evidențează poziția curentă în fișier
- cursorul poate fi manipulat fără operațiuni de citire/scriere folosind funcții dedicate

Fișiere

Secvența tipică de lucru cu fișiere

1. se deschide fișierul folosind funcția fopen(…)
2. se testează valoarea returnată a funcției fopen(…)
3. se prelucrează fișierul (citire/scriere)
4. la final se închide fișierul cu fclose(…)
5. se testează valoarea returnată de funcția fclose(…)

```c
#include <stdio.h>

int main(void)
{
  FILE *f = NULL;
  char *filename = "myfile";
  if ((f = fopen(filename, "r")) == NULL)
    {
      // tratare eroare deschidere
    }
  else
    {
      // prelucrare fisier
    }

  if (fclose(f) != 0)
    {
      // tratare eroare inchidere
    }
  return 0;
}
```

Fișiere

Secvența tipică de lucru cu fișiere

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  FILE *f = NULL;
  char *filename = "myfile";
  if ((f = fopen(filename, "r")) == NULL)
    {
      perror(NULL);
      exit(-1);
    }
  else
    {
      // prelucrare fisier
    }

  if (fclose(f) != 0)
    {
      perror(NULL);
      exit(-1);
    }
  return 0;
}
```

Fișiere

Fișiere text

- în realitate nu există fișiere text sau binare - date oricum sunt stocate binar în memorie și în fișiere
- există funcții specializate prin care se prelucrează mai ușor fișiere ce conțin doar text – aceste fișiere se pot prelucra și binar, fără a se interpreta și considera conținut lor ca fiind text
- se consideră un fișier text acel fișier ce conține doar caractere printabile conform tabelei ASCII (litere, cifre, caractere speciale de puncuație, spații, tab-uri, … etc)
- limbajul C, prin bibliotecile standard, oferă anumite funcții specializate de operare asupra fișierelor text
- funcțiile interpretează de obice datele din fișier și le oferă programatorului sub formă de string-uri
- atentie ! funcțiile de lucru cu fișiere text nu vor scrie/citi niciodată din fișier caracterul \0 – caracterul \0 nu este caracter printabil deci el nu există în fișier

Fișiere

Fișiere text

- Funcția fgetc: **`int fgetc(FILE *stream);`**
  - citește un caracter din fișierul reprezentat printr-un pointer FILE * dat ca și argument
  - returnează caracterul citit sau EOF în caz de eroare sau sfârșit de fișier
  - după fiecare citire avansează cursorul din fișier
- Exemplu: sa se scrie o funcție care numără caracterele din fișierul a cărui cale este primită ca argument

```
uint32_t count_chars(const char *file)
{
  FILE *f = NULL;
  uint32_t result = 0;
  int ch;
  f = fopen(file, "r");
  if (f == NULL)
    {
      perror(NULL);
      return 0;
    }
  while ((ch = fgetc(f)) != EOF)
    {
      result++;
    }
  fclose(f);
  return result;
}
```

Fișiere

Fișiere text

- Funcția fgetc: **`int fgetc(FILE *stream);`**
  - citește un caracter din fișierul reprezentat printr-un pointer FILE * dat ca și argument
  - returnează caracterul citit sau EOF în caz de eroare sau sfârșit de fișier
  - după fiecare citire avansează cursorul din fișier
- Exemplu: sa se scrie o funcție care numără caracterele din fișierul a cărui cale este primită ca argument

```
uint32_t count_chars(const char *file)
{
  FILE *f = NULL;
  uint32_t result = 0;
  int ch;
  f = fopen(file, "r");
  if (f == NULL)
    {
      perror(NULL);
      return 0;
    }
  while ((ch = fgetc(f)) != EOF)
    {
      result++;
    }
  fclose(f);
  return result;
}
```

Fișiere

Fișiere text

- Funcția fgets: `char *fgets(char *s, int size, FILE *stream);`
  - citește din fișierul referențiat prin stream un șir de caractere până la EOF sau linie nouă iar la sfârșit adaugă caracterul '\0'
  - citește maxim size-1 caractere pe care le stochează în zone de memorie referită de *s*
  - adaugă la sfârșit caracterul \0 (motivul pentru care citește size-1 caractere)
  - dacă se oprește pentru ca a întâlnit linie nouă adaugă și caracterul linie nouă \n
  - returnează *s* dacă s-a citit cu succes minim un caracter sau NULL dacă a apărut o eroare și nu s-a citit nici un caracter

```c
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

int main(void)
{
  FILE *f = NULL;
  char text[20];
  if ((f = fopen("file.txt", "r")) == NULL)
    {
      perror(NULL);
      return 0;
    }
  if (fgets(text, 20, f) != NULL)
    {
      printf ("Linia: %s\n", text);
    }
  else
    {
      perror(NULL);
    }

  fclose(f);
  return 0;
}
```

Fișiere

Fișiere text

- Funcția fputc: `int fputc(int c, FILE *stream);`
  - scrie caracterul dat ca și argument prin c în fișierul referențiat prin stream
  - avansează cursorul fișierului un 1
  - returnează caracterul scris sau EOF dacă a apărut o eroare
- Funcția ungetc: `int ungetc(int c, FILE *stream);`
  - funcția pune înapoi în fișierul referențiat prin stream caracterul dat de argumentul c
  - doar o singură operație este garantată
- Funcția fputs: `int fputs(const char *s, FILE *stream);`
  - scrie în fișierul referențiat prin stream string-ul reprezentat prin argumentul s (evident fără a scrie și terminatorul NULL \0)
  - avansează cursorul fișierului cu numărul de caractere scrise
  - returnează un număr >= 0 în caz de succes sau EOF în caz de eroare

Fișiere

Fișiere text

- Funcția feof: `int feof(FILE *stream);`
  - funcția testează dacă cursorul fișierului referențiat prin stream a ajuns la EOF și în acest caz retrunează o valoarea diferită de 0, în caz contrar returnează 0

```c
#include <stdio.h>
#include <stdint.h>

uint32_t count_chars(const char *file)
{
  FILE *f = NULL;
  uint32_t result = 0;
  int ch;
  f = fopen(file, "r");
  if (f == NULL)
    {
      perror(NULL);
      return 0;
    }
  while (!feof(f))
    {
      ch = fgetc(f);
      result++;
    }
  fclose(f);
  return result;
}
```

Fișiere

Funcții de poziționare a cursorului

- Funcția ftell: `long ftell(FILE *stream);`
  - funcția returnează poziția curenta a cursorului din fișierul refernțiat de stream față de începutul fișierului
- Funcția rewind: `void rewind(FILE *stream);`
  - funcția poziționează cursorul fișierului referențiat prin stream la începutul fișierului
- Funcția fseek: `int fseek(FILE *stream, long offset, int whence);`
  - funcția poziționează cursorul fișierului referențiat prin stream la valorea dată prin *offset* față de parametrul whence
  - parametrul offset specifică incrementului (pozitiv sau negativ)
  - whence – specifică față de ce poziție se calculează noul cursor prin adăugarea lui offset. Poate fi:
    - SEEK_SET – noul cursor se calculează prin adăugarea lui offset față de începutul fișierului
    - SEEK_CUR – noul cursor se calculează prin adăugarea lui offset față de poziția curentă a cursorului
    - SEEK_END – noul cursor se calculează prin adăugarea lui offset față de sfârșitul fișierului

- aceste funcții lucrează la nivel de byte, cursorul fiind la nivel de byte

Fișiere

Fișiere binare

- în realitate nu există fișiere text sau binare ☾ date oricum sunt stocate binar în memorie și în fișiere
- bibliotecile standard C oferă funcții de scriere și citire din fișier la nivel de byte, în mod binar fără vreo interpretare prealabilă
- Funcția fread: `size t fread(void *ptr, size t size, size t nmemb, FILE *stream);`
  - citește din fișierul referențiat prin *stream* un număr de *nmemb* elemente de dimensiune *size* și le scrie în zona de memorie ptr
  - returnează numărul de elemente citite
- Funcția fwrite: `size t fwrite(void *ptr, size t size, size t nmemb, FILE *stream);`
  - scrie în fișierul referențiat prin *stream* un număr de *nmemb* elemente de dimensiune *size* din zona de memorie ptr
  - returnează numărul de elemente scrise în fișier

Fișiere

Fișiere binare

- program ce scrie un număr de intregi fără semn pe 4 bytes într-un fișier

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

#define COUNT 100

int main(void)
{
  FILE *out = NULL;
  uint32_t n = 0;
  if ((out = fopen("output.bin", "wb")) == NULL)
    {
      perror(NULL);
      exit(-1);
    }
  for (int i = 0; i < COUNT; i++)
    {
      n = i * 176549 + 3;
      if (fwrite(&n, sizeof(uint32_t), 1, out) != 1)
        {
          perror(NULL);
          exit(-1);
        }
    }
  if (fclose(out) != 0)
    {
      perror(NULL);
      exit(-1);
    }
  return 0;
}
```

Fișiere

Fișiere binare

- program ce citește un număr necunoscut de întregi pe 4 bytes dintr-un fișier binar și le scrie în format text într-un fișier text în format hexazecimal pe 8 cifre cu padding
de 0

```c
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

int main(void)
{
  FILE *in = NULL;
  FILE *out = NULL;
  uint32_t n = 0;
  if ((in = fopen("output.bin", "rb")) == NULL)
    {
      perror(NULL);
      exit(-1);
    }
  if ((out = fopen("output.txt", "w")) == NULL)
    {
      perror(NULL);
      exit(-1);
    }
  while (fread(&n, sizeof(uint32_t), 1, in) == 1)
    {
      fprintf (out, "%08X ", n);
    }
  if (fclose(in) != 0)
    {
      perror(NULL);
      exit(-1);
    }
  if (fclose(out) != 0)
    {
      exit(-1);
    }
  return 0;
}
```

Fișiere

Observații și consecințe

- Fiecare program are acces la 3 fișiere standard

```
FILE *stdin;
FILE *stdout;
FILE *stderr;
```

- se pot folosi în programe în mod explicit
- Exemplu:
  - printare a unui mesaj la standard error: `fprintf(stderr, "mesaj");`
- Funcția `fflush: int fflush(FILE *stream);`
  - la o cerere de scriere, datele nu se scriu direct în fișier ci până la fișier există o serie de buffere iar scrierea efectivă se face de obicei când bufferele intermediare sunt pline sau când decide sistemul de operare
  - cere sistemului de operare să scrie datele fizic în fișier
- Fișierul standard stdout este și el supus aceleiași implementări: există buffere intermediare – scrierea nu se face neaparat în momentul apelului unei funcții.
  - pentru a forța scrierea in stdout (implicit pe consolă) se poate folosi ori caracterul rând nou (\n) ori apelul `fflush(stdout)`
- Fișierul standard de eroare stderr este "conectat" direct la consolă și nu sunt buffere intermediare
  - o scriere la stderr se va face în momentul apelului fără a fi nevoie de caracterul rând nou sau apel de fflush

Argumente în linie de comandă

Argumente în linie de comandă

Definiții și exemple

- Sistemele de operare asigură suport astfel încât programele executabile să poată accepta argumente în linie de comandă
- Sintaxă exemplu: `program_executabil arg1 arg2 arg3 arg4`
- În limbajul C există suport astfel încât aceste argumente să se poată prelucra de către programator
- Se implementează în limbajul C printr-o variantă a funcției main cu argumente

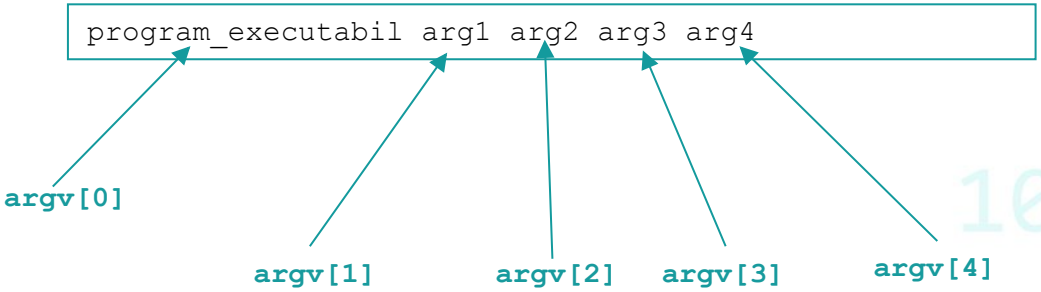
```
int main(int argc, char **argv)
{


}
```

- argumentul *argc* – reprezintă numărul de argumente primite de program în linie de comandă
- argumentul *argv* – este implementat ca un tablou de string-uri ce reprezintă efectiv argumentele, fiecare în format string (cu terminator \0). Dimensiunea tabloului de string-uri este *argc.*

Argumente în linie de comandă

Definiții și exemple

- Considerăm următorul exemplu:



- numele programului executabil reprezintă întotdeauna argv[0], argc > 1 – argc nu poate fi 0
- numărul efectiv de argumente în linie de comandă este dat de argc – 1
- tinând cont de modul de declarare a funcției main(int argc, char **argv) în program argumentele vor fi interpretate ca și string-uri
- în programe se va testa de fiecare argc și se vor valida argumentele astfel încât să corespundă cerinței programului

Argumente în linie de comandă

Definiții și exemple

- Program exemplu: printarea argumentelor date în linie de comandă

```c
#include <stdio.h>

int main(int argc, char **argv)
{
  for (int i = 0; i < argc; i++)
    {
      printf ("argumentul %d : %s\n", i, argv[i]);
    }
  return 0;
}
```

```
valy@staff:~/teaching$ ./p ana are mere
argumentul 0 : ./p
argumentul 1 : ana
argumentul 2 : are
argumentul 3 : mere
valy@staff:~/teaching$
```

Argumente în linie de comandă

Definiții și exemple

- Program exemplu: program ce primește ca argumente în linie de comanda operant1 semn_operație operand2 și printează rezultatul operației

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv)
{
  int op1, op2;
  char semn;
  if (argc < 4)
    {
      fprintf(stderr, "Eroare argumente\n");
      exit(-1);
    }
  op1 = strtol(argv[1], NULL, 10);
  op2 = strtol(argv[3], NULL, 10);
  if ((strlen(argv[2]) == 0) || (strlen(argv[2]) > 1))
    {
      fprintf(stderr, "eroare semn\n");
      exit(-1);
    }
  semn = argv[2][0];
  switch (semn)
    {
    case '+':
      {
        printf ("%d + %d = %d\n", op1, op2, op1 + op2);
        break;
      }
    case '-':
      {
        printf ("%d - %d = %d\n", op1, op2, op1 - op2);
        break;
      }
    default:
      {
        fprintf(stderr, "semn necunoscut\n");
      }
    }
  return 0;
}
```

Secțiunea XIV

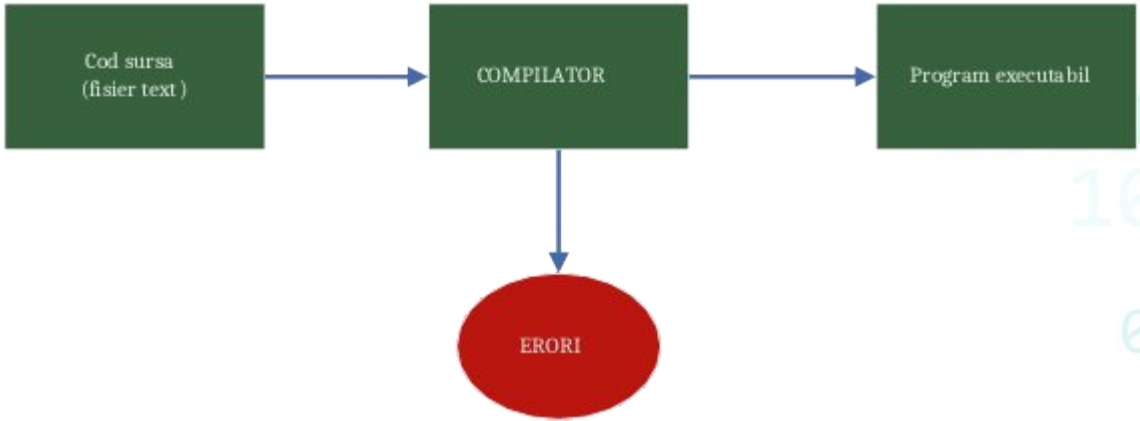
Etapele compilării
Preprocesorul C
Tratarea erorilor

Etapele compilării

Etapele compilării

Recapitulare

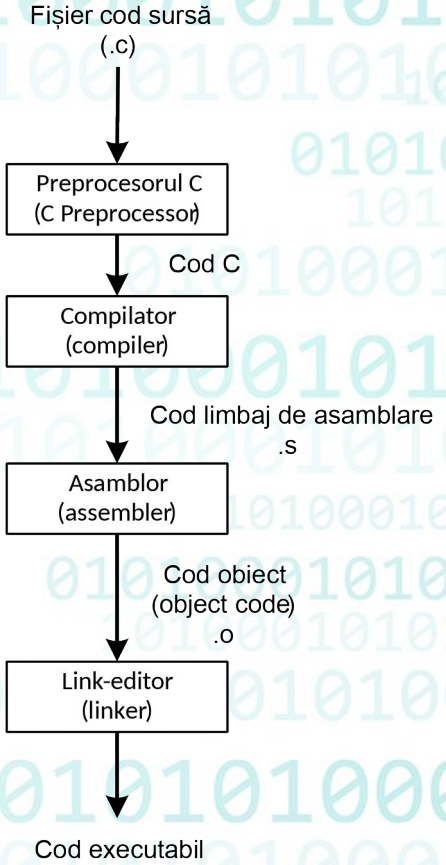
- Compilarea – procesul prin care un program scris într-un limbaj de programare este translatat în cod obiect și mai apoi în cod executabil pentru sistemul de calcul țintă



Etapele compilării

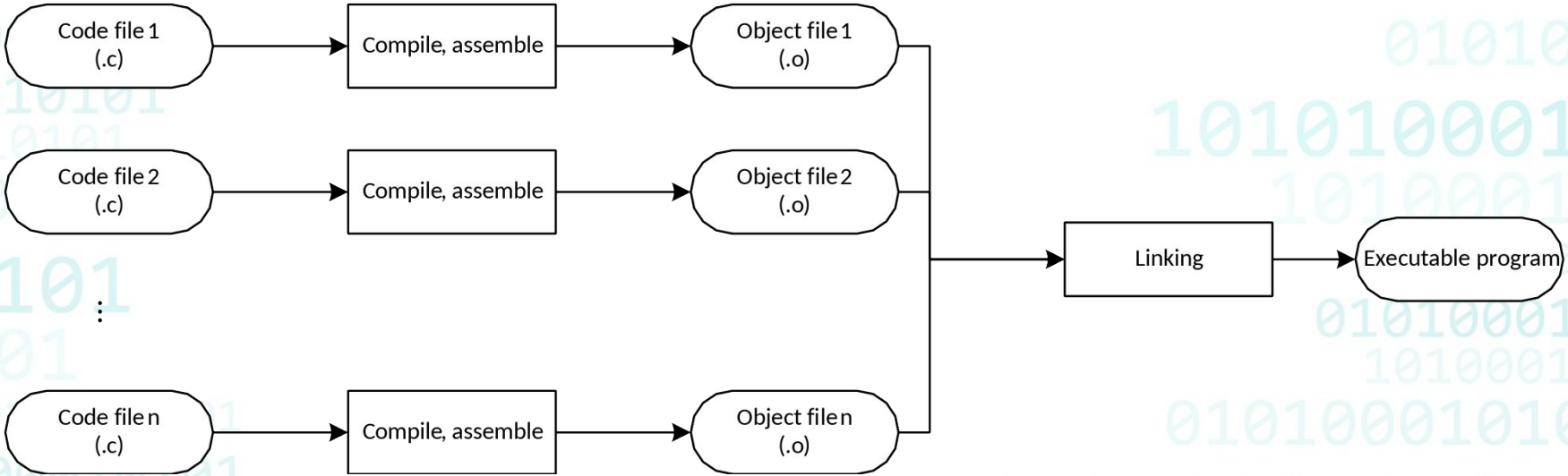
Etapele compilării

- Preprocesorul C – simplu editor de texte; preia codul scris în limbajul C și rezolvă anumite directive ce ii sunt asociate rezultând tot un cod C
- Compilatorul C – preia codul C pre-procesat de preprocesorul C și generează un cod intermediar în limbaj de asamblare specific mașinii țintă (target).
  Codul generat este într-un format lizibil pentru programator (human readable)
- Asamblorul – preia codul în limbaj de asamblare generat de compilator și generează un fișier de cod obiect ce conține codul mașină binar rezultat
- Link-editorul – preia toate fișierele obiect și le combină într-un singur program executabil (sau bibliotecă – o colecție de funcții fără funcția main)
- **Observații**
  - Un program C – poate fi compus din mai multe fișiere de cod (.c) și fișiere header (.h)
  - pentru fiecare fișier de cod (.c) rezultă un fișier obiect (.o)
  - toate fișierele obiect (.o) sunt link-editate într-un singur program executabil (sau bibliotecă)
  - Doar ișierele de cod (.c) sunt trecute prin procesul de compilare
  - fișierele header (.h) sunt incluse de preprocesorul C prin directive #include
  - în mod uzual compilarea se face de la cod C *direct* la cod executabil prin ascunderea și ștergerea fișierelor intermediare (totuși pașii se respectă și fișierele intermediare se crează temporar în /tmp)
  - la proiecte software de mari dimensiuni compilarea se face pe pași și se păstrează fișierele intermediare



Etapele compilării

Etapele compilării



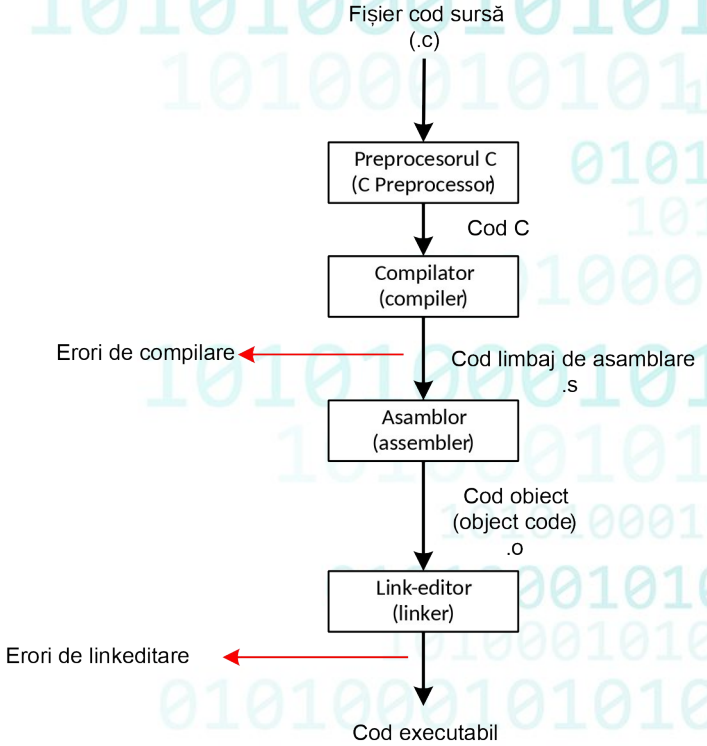
Tratarea erorilor

Tratarea erorilor

Clasificarea erorilor

- **Erori de compilare**
  - apar în momentul translatării codului C în limbaj intermediar
  - erori de sintaxă
  - se specifică tipul erorii, fișierul de cod, linia și coloana din fișierul de cod
  - la apariția unei erori nu se generează fișier obiect și procesul se oprește
- Exemplu:

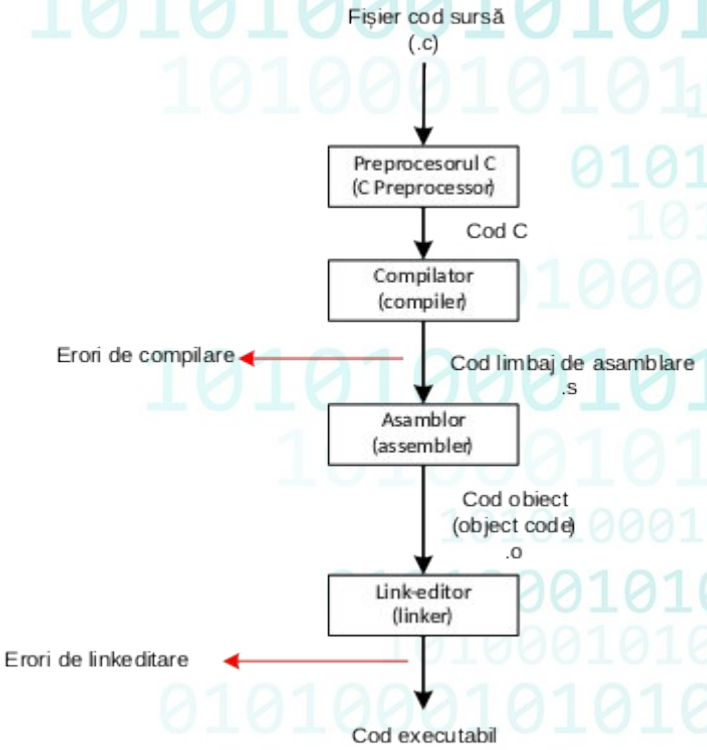
```
simple.c: In function 'sum':
simple.c:7:15: error: expected ';' before '}' token
   return a + b
               ^
               ;
 }
 ~
simple.c: In function 'main':
simple.c:15:15: error: expected expression before ')' token
   a = sum("a",);
               ^
```



Tratarea erorilor

Clasificarea erorilor

- **Erori de link-editare**
  - erori apărute în urma generării de cod intermediare
  - nu sunt erori de sintaxă ci erori de logică minimă de cod
  - specifică erori de conexiuni între simboluri (de ex: o funcție doar declarată dar nedefinită – fără corp de cod)
  - nu se poate da fișierul de cod, linia si coloana erorii – nu mai e la nivel de cod C
  - se dă de obicei fișierul obiect în care a apărut eroarea si adresa din cadrul acestuia unde a apărut eroarea
  - sunt mai greu de rezolvat decât erorile de compilare
  - la apariția unei erori nu se generează fișier executabil și procesul se oprește
- Exemplu:




```
valy@staff:~/teaching$ gcc -Wall -o p simple.c
/usr/bin/ld: /tmp/ccAS8Dsx.o: in function `main':
simple.c:(.text+0x40): undefined reference to `dif'
collect2: error: ld returned 1 exit status
```

Tratarea erorilor

Mesaje de atenționare - Warnings

- **Warning**
  - mesaje de atenționare date de compilator
  - procesul complet de compilare se realizează cu succes
  - mesajele de atenționare (warning) nu reprezintă erori dar pot duce la o funcționare incorectă a programului
  - în unele standarde warning se consideră eroare și compilarea este terminată fără să producă fișier obiect și/sau fișier executabil
- **Un program bun este un program ce nu are warning-uri la compilare**

Tratarea erorilor

Ordinea tratării și rezolvării erorilor

1. **Erori de compilare**
   - primele erori ce se tratează și se încearcă rezolvarea lor
   - se rezolvă în ordinea apariției mesajelor de eroare date de compilator începând cu **prima** eroare
   - după rezolvarea primei erori dată de compilator se **reia procesul de compilare**
   - sub nici o formă nu se tratează prima dată ultima eroare sau o eroare oarecare *i* din cod
2. **Erori de link-editare**
   - se tratează după ce s-au tratat toate erorile de compilare cu compilări succesive, atunci când nu mai sunt alte erori de compilare
   - se rezolvă în ordinea apariției mesajelor de eroare date de compilator începând cu **prima** eroare
   - după rezolvarea primei erori dată de compilator se **reia procesul de compilare**
   - sub nici o formă nu se tratează prima dată ultima eroare sau o eroare oarecare *i* din cod
3. **Mesaje de tip warning**
   - se rezolvă doar după ce codul este compilabil (fără nici un fel de erori de compilare sau link-editare) și s-a produs fișier obiect și/sau program executabil
   - se rezolvă în ordinea apariției mesajelor de warning date de compilator începând cu primul mesaj și reia procesul de compilare după fiecare warning rezolvat
   - sub nici o formă nu se lasă programe compilabile dar cu mesaje de warning !!!! chiar daca se obține fișier executabil
   - implicit compilatorul poate suprima și evita tipărirea mesajelor warning. Se recomandă folosirea argumentului –Wall dat compilatorului pentru a tipări toate mesajele warning

Tratarea erorilor

Exemplu erori multiple și warning-uri

```
valy@staff:~/teaching$ gcc -Wall -o p simple.c
simple.c:3:1: warning: data definition has no type or storage class
 a;
 ^
simple.c:3:1: warning: type defaults to ‘int’ in declaration of ‘a’ [-Wimplicit-int]
simple.c: In function ‘sum’:
simple.c:7:15: error: expected ‘;’ before ‘}’ token
   return a + b
               ^
               ;
 }
 ~
simple.c: In function ‘main’:
simple.c:15:15: error: expected expression before ‘)’ token
   a = sum("a",);
               ^
simple.c:15:11: warning: passing argument 1 of ‘sum’ makes integer from pointer without a cast [-Wint-conversion]
   a = sum("a",);
           ^~~
simple.c:5:13: note: expected ‘int’ but argument is of type ‘char *’
 int sum(int a, int b)
         ~~~~^
simple.c:13:7: warning: variable ‘a’ set but not used [-Wunused-but-set-variable]
   int a;
```

Secțiunea XV

Programe C complexe

Fișiere header
Programe cu mai multe fișiere C

Fișiere header

Definiții și exemple

- Pe langă fișierele .c în limbajul C mai sunt definite și fișiere header .h
- sunt fișiere ce conține doar declarații de tipuri, variabile, funcții și macro-uri
- nu conțin definiții și cod – nu conțin corpul funcțiilor
- fișierele header se includ cu directive de #include
- fișierele header nu se compilează și nu se transmit la compilator – ele doar se includ
- fișierele C se transmit compilatorului – fișierele C NU SE INCLUD (este conceptual greșit !!!)
- de obicei un fișier header insoțește un fișier C dar pot să existe și mai multe fișiere header fără vreun fișier C asociat
- un program C mai complex nu se scrie într-un singur fișier C ci se separă în mai multe fișiere C și header în funcție de rol și funcționalitate
- unul sau mai multe fișiere .c cu unul sau mai multe fișier .h pot forma o bibliotecă – o colecție de funcții și tipuri de date asociate

Fișiere header

Definiții și exemple

- Să se implementeze o bibliotecă ce gestionează operații minimale pe numere complexe. Să se implementeze de asemenea un program principal, separat care să testeze și să folosească biblioteca de numere complexe

```c
#ifndef __COMPLEX_H
#define __COMPLEX_H

typedef struct
{
  int re;
  int im;
}COMPLEX;

COMPLEX complex_add(COMPLEX a, COMPLEX b);
COMPLEX complex_sub(COMPLEX a, COMPLEX c);

#endif
```

complex.h

```c
#include <stdio.h>
#include "complex.h"

int main(void)
{
  COMPLEX a = {1,-2};
  COMPLEX b = {10,20};
  COMPLEX t;
  t = complex_add(a,b);
  t = complex_sub(a,b);
  return 0;
}
```

complex_test.c

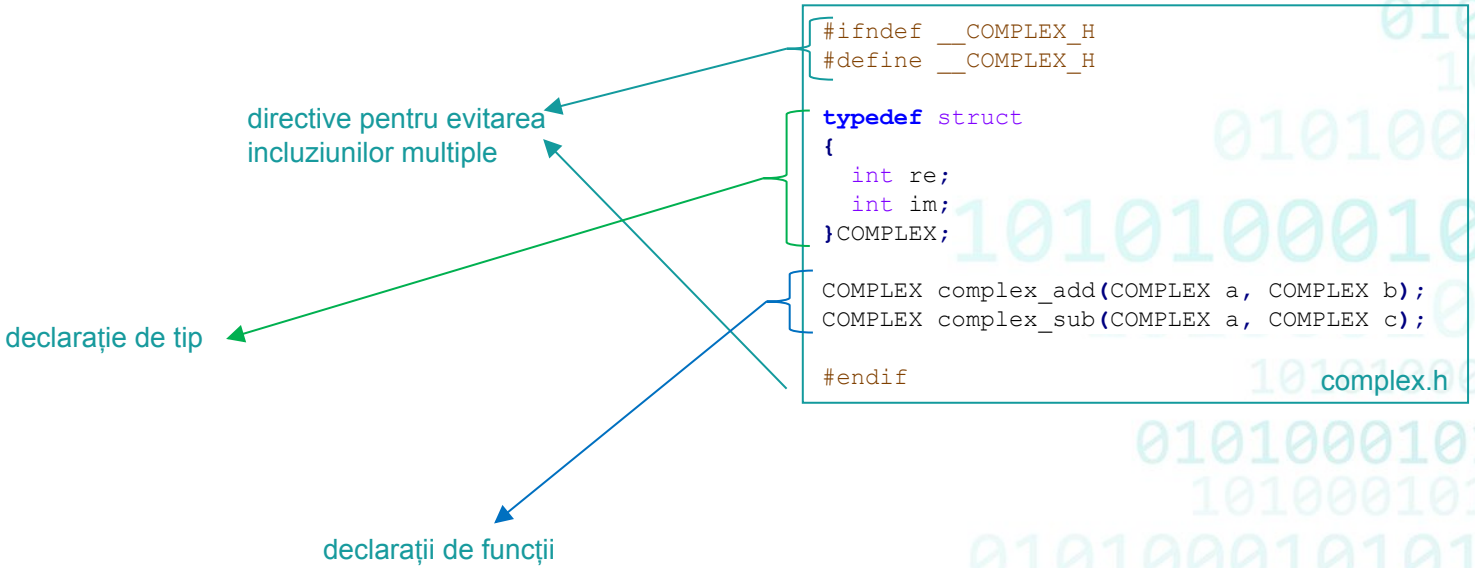
```c
#include "complex.h"

COMPLEX complex_add(COMPLEX x, COMPLEX y)
{
  COMPLEX r;
  r.re = x.re + y.re;
  r.im = x.im + y.im;
  return r;
}

COMPLEX complex_sub(COMPLEX x, COMPLEX y)
{
  COMPLEX r;
  r.re = x.re - y.re;
  r.im = x.im - y.im;
  return r;
}
```

complex.c

```
gcc -Wall -o p complex.c complex_test.c
```

Fișiere header

Definiții și exemple



Fișiere header

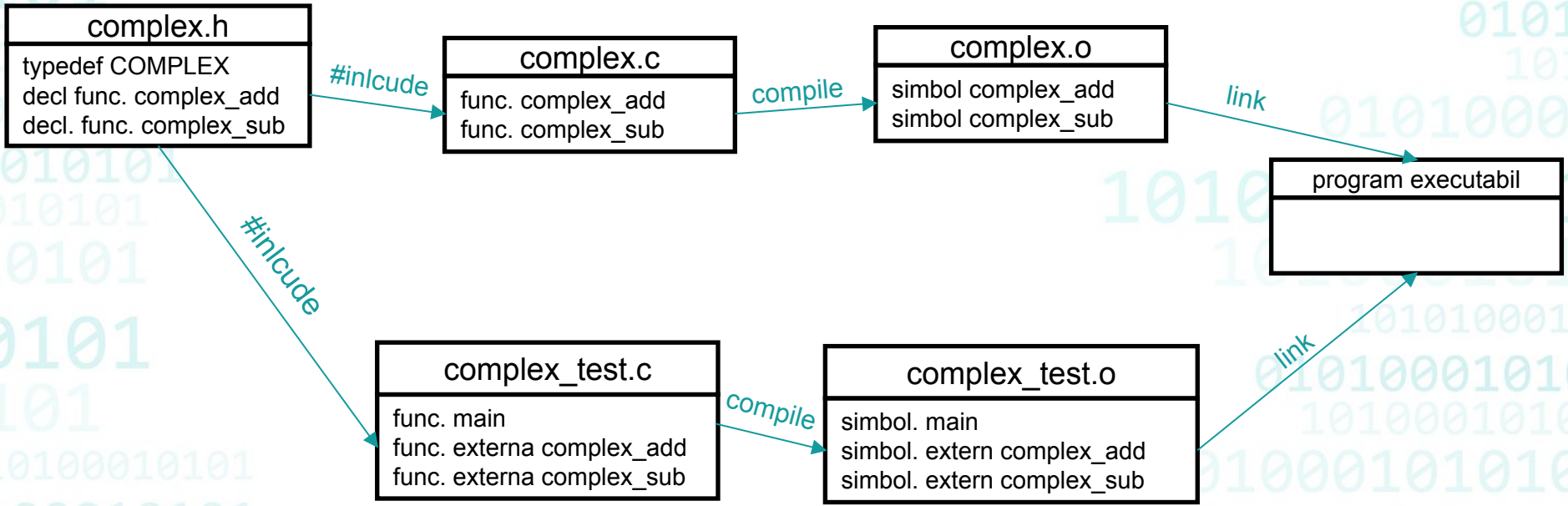
Definiții și exemple

- Directivele pentru evitarea incluziunilor multiple (once-only headers)
  - un fișier header poate ajunge să fie închis direct sau indirect de mai multe ori în același fișier C fiind practic parcurs de mai multe ori de către preprocesor si compilator ☾ pot rezulta erori de definiții și declarații multiple
  - directivele rezolva problema: doar prima dată va intra în corpul #indef…#endif – la a doua parcurgere va găsi ca fiind deja definit macro-ul
  - macro-urile din definiții multiple sunt realizate prin niște convenții de cod – este necesar doar să fie unice între fișiere
  - în standard-ul POSIX se recomandă să se declare folosind numele fișierului header precedat de 2 caractere underscore (__) iar în loc de punct se va pune tot un caracter underscore. ex: __COMPLEX_H
  - toate declarațiile se vor scrie în corpul #ifndef…#endif
- În fișierele header nu se definesc funcții (nu se implementează) și nu se definesc variabile

Fișiere header

Definiții și exemple

- Compilare exemplu precedent:



Fișiere header

Definiții și exemple

- Definiții de funcții și variabile în fișiere header ☾ poate duce la erori de tipul *multiple definition*
- Definiția unei variabile este implicită cu declarație
  - declarația unei variabile ☾ specificarea tipului și a numelui variabilei
  - definiția unei variabile ☾ alocarea de memorie pentru variabilă
- Cum se poate face declarație de variabilă fără definiție?
- Soluție: modificatorul **extern** (keyword) – specifică compilatorului că acea variabilă nu se va defini în fișierul obiect curent ci ea este definită intr-un alt fișier obiect
- în complex.h *n* este doar declarat iar când se include în complex.c nu va genera eroare de tipul *multiple definition*
- daca nu se compilează și complex.c se va genera eroare de linkeditare: *unreferenced symbol*

```c
#include "complex.h"
int n;
```

complex.c

```c
#ifndef __COMPLEX_H
#define __COMPLEX_H

extern int n;

#endif
```

complex.h

```c
#include "complex.h"

int main(void)
{
  n = 1;
  return 0;
}
```

complex_test.c

Fișiere header

Definiții și exemple

- în cazul anterior: variabila n este definită în fișierul obiect complex.o și accesată din fișierul obiect complex_test.o
- există și posibilitatea ca variabila n să poată fi blocată la nivelul fișierului obiect din care face parte prin folosirea modificatorului static
- în acest caz, folosirea modificatorului extern va genera eroare la includerea în complex_test.c, variabila n nefiind vizibilă în afara fișierului obiect complex.o
- Ⓟ folosirea modificatorului static în fața unei declarației unei variabile globale limitează utilizarea acestei variabile la nivelul fișierului obiect din care va face parte – se va permite link-editare în exterior

```c
#include "complex.h"
static int n;
```

complex.c

```c
#ifndef __COMPLEX_H
#define __COMPLEX_H

extern int n; // va genera eroare la link

#endif
```

complex.h

```c
#include "complex.h"

int main(void)
{
  n = 1;
  return 0;
}
```

complex_test.c

Funcții cu număr variabil de argumente

Funcții cu număr variabil de arg

Definiții și exemple

- Se poate observa că funcțiile precum printf, scanf, sscanf, sprintf,… etc nu au un număr finit de argumente ci depinde de directivele de formatare
- Limbajul C permite programatorului declararea de funcții cu număr variabil de argumente
- Funcția cu număr variabil de argumente trebuie sa aiba cel puțin un argument cunoscut iar argumentele variabile se pun ultimele și se reprezintă prin …
- Funcție cu număr variabil de argumente (variadic function)
- Sintaxă `tip_return nume_funcție(tip arg, …);`
- biblioteca <stdarg.h> oferă niște macro-uri prin care se pot obține argumentele de pe stivă ce urmează după ultimul argument cunoscut
- nu se poate obține numărul de argumente variabile cu care a fost invocată funcția, programatorul trebuie să-și implementeze propriile mecanisme și convenții
- macro-urile din <stdarg.h> sunt foarte nesigure și nu se oferă nici un mecanism de protecție

Funcții cu număr variabil de arg

Definiții și exemple

- Macro `void va_start(va_list ap, last);`
  - inițializează argumentul ap de tip va_list cu last, ce reprezintă ultimul argument stabil al funcției înainte de argumentele variabile
- Macro `type va_arg(va_list ap, type);`
  - returnează valoarea argumentului curent din lista de argumente variabile și castează valoarea la tipul *type*
  - după apel se avansează în lista *va_list* la următorul argument din lista variabilă
- Macro `void va_end(va_list ap);`
  - eliberează lista va_list

Funcții cu număr variabil de arg

Definiții și exemple

- Exemplu: să se scrie o funcție cu număr variabil de argumente ce calculează suma argumentelor variabile ce se consideră ca sunt de tip int. Funcția va avea un prim argument cunoscut de tip int ce determină numărul de argumente variable ce urmează

```c
#include <stdio.h>
#include <stdarg.h>

int sum(int n, ...)
{
  int result = 0;
  va_list arg_list;
  va_start(arg_list, n);
  for (int i = 0; i < n; i++)
    {
      result = result + va_arg(arg_list, int);
    }
  va_end(arg_list);
  return result;
}

int main(void)
{
  printf ("sum: %d\n", sum(3,10,10,10));
  printf ("sum: %d\n", sum(2, 4, 3));
  return 0;
}
```

```
valy@staff:~/teaching$ ./p
sum: 30
sum: 7
valy@staff:~/teaching$
```

Funcții de timp și numere aleatoare

Funcții de timp

Definiții și exemple

- Biblioteca <time.h> definește niște funcții ce permit obținerea anumitor elemente de timp
- Funcția clock_t clock(void);
  - returnează un număr ce reprezintă timpul scurs de la lansarea în execuție a programului în unități de timp definite de constanta CLOCKS_PER_SEC. Pentru obținerea timpului în secunde rezultatul va împărți la CLOCKS_PER_SEC
- Funcția time_t time(time_t *tloc);
  - returnează un număr ce reprezintă clasicul UNIX time – numărul de secunde de la Epoch – numărul de secunde de la data de 1970-01-01 00:00:00 +0000 (UTC).
  - daca tloc este nenull valoarea returnată va fi scrisă și la adresa reprezentată de tloc

Funcții de numere aleatoare

Definiții și exemple

- Un sistem de calcul NU POATE genera numere perfect ALEATOARE – există totuși mecanisme pe baza unor algoritmi prin care se pot genera numere pseudo-aleatoare – numere aleatoare ce se obțin pe baza unui algoritm și funcții matematice
- algoritmul de generare de numere pseudo-aleatoare începe de obicei de la o valoare inițiață denumită seed
- Pentru obținere de numere aleatoare în limbajul C există funcțiile rand și srand
- Funcția: `void srand(unsigned int seed);`
  - inițializează algoritmul de generare de numere aleatoare cu valorea seed dată ca argument
  - această funcție se apelează de obicei o singura dată la începutul programului
- Funcția: `int rand(void);`
  - la fiecare apel a funcției rand se va obține un număr pseudo-aleator pe baza algoritmului intern ce a pornit de la valoarea seed inițializată cu funcția srand(…)

Matrici

Matrici

Definiții

- Limbajul C oferă facilitatea de a declara tablouri multidimensionale. Sinaxa:

```
tip nume_tablou[DIM1][DIM2][DIM3]…[DIMn];
```

- În practică, de obicei, se folosesc doar 2 domensiuni - matrice. Sintaxa:

```
tip nume_matrice[DIM1][DIM2];
```

- Exemplu: `int mat[M][N];`
  - s-a declarat astfel matrice de M linii și N coloane
  - se adresează tot ca un tablou: `mat[i][j]`
  - `mat[i]` reprezintă linia i, `&mat[i]` reprezintă adresa liniei i – de tip int (*)[N]
  - `mat[i][j]` reprezintă elementul matricii de pe pozitia (i, j)

```c
// cod de afisare a unei matrici
#include <stdio.h>

#define M 16
#defien N 32

int main(void)
{
	int matrix[M][N];
	...
	for (int i = 0; i < M; i++)
	{
		for (int j = 0; j < N; j++)
		{
			printf ("%4d ", matrix[i][j]);
		}
		printf ("\n");
	}
	return 0;
}
```

Matrici

Transmitere matrici ca parametru către funcții. Inițialzare matrici

- Prima dimensiune nu trebuie specifică, însă următoarele dimensiuni se specifică (a doua)

```c
#include <stdio.h>

#define M 4
#define N 4

void printMatrix(int m[][N])
{
  for (int i = 0; i < M; i++)
    {
      for (int j = 0; j < N; j++)
        {
          printf ("%4d ", m[i][j]);
        }
      printf ("\n");
    }
}

int main(void)
{
  int matrix[M][N];
  // some code
  printMatrix(matrix);
  // some code
  return 0;
}
```

- În cazul tablourilor:

```c
int matrix[2][3] = { {1,2,3}, {2, 3, 4 } };
```