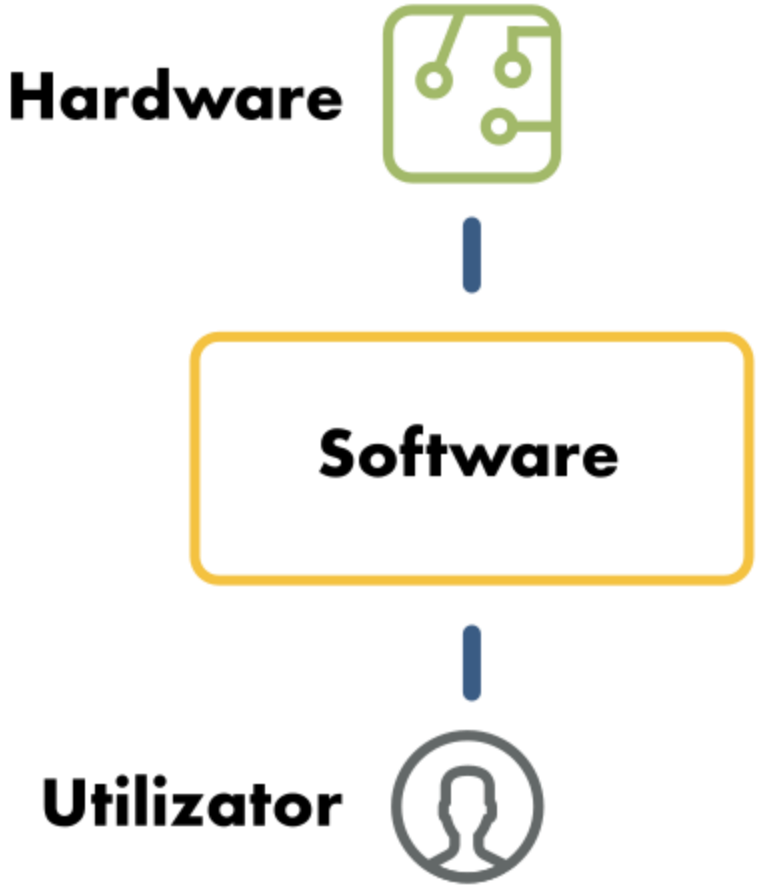
# Programe (C) și sistemul de calcul

Modificat: 5-Oct-20

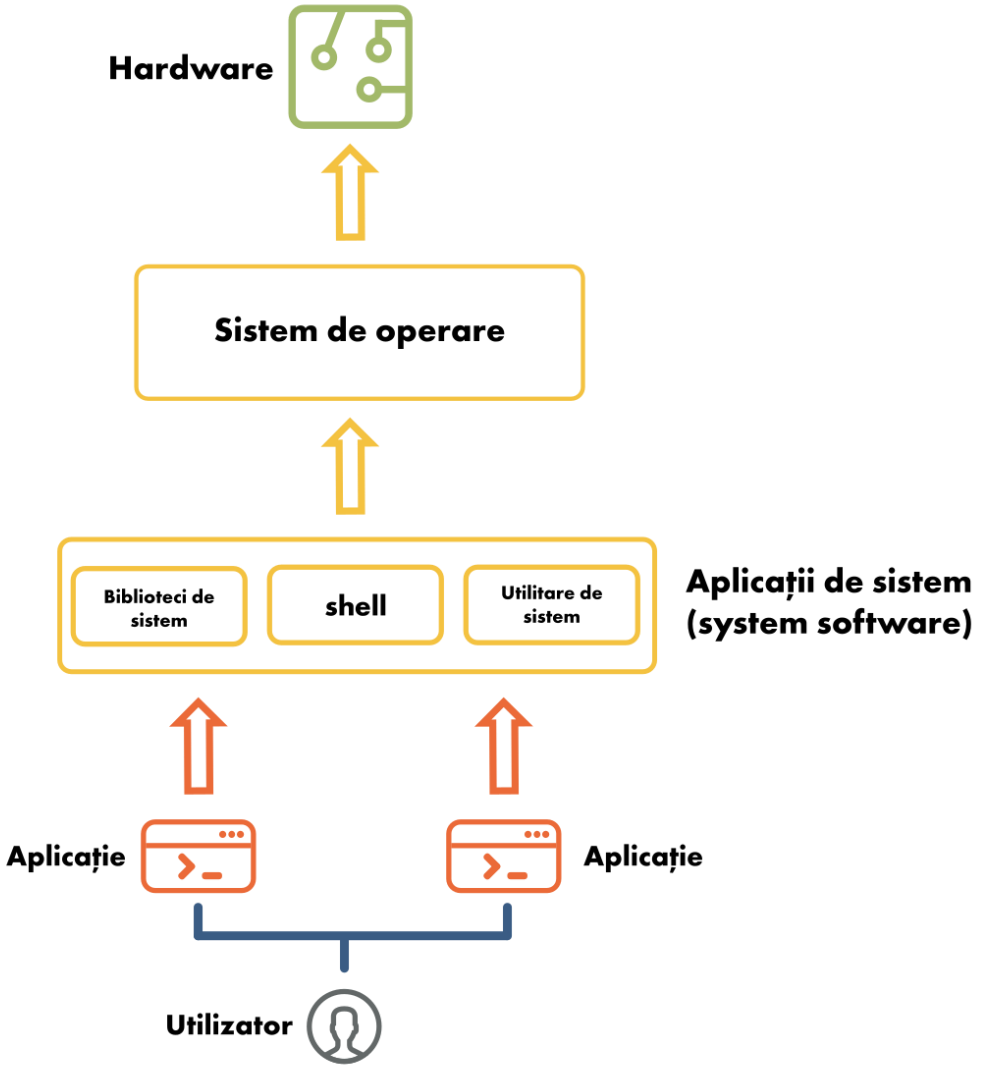
# Cuprins curs 1

- Software și hardware
- Cod sursă
- Cod mașină
- Fișier executabil
- Compilator
- ISA
- Memorie
- Procesor
- De ce să știm interfața hardware / software?
- Date și cod
- Variabile în C
- Pointeri în C

# Utilizator – software – hardware



# Utilizator – software - OS - hardware



# Software

- Programe, aplicații
- Instrucțiuni și date
  - Uzual într-un fișier (executabil)
- Specifică ce trebuie să execute hardware-ul

- Firefox, GCC, MS Office, MS Teams

- Avantaje: ușor de dezvoltat, ușor de extins
- Dezavantaje: foarte divers, probleme de portabilitate, funcționare

# Hardware

- Componente electronice
- Rulează instrucțiuni, stochează date din sofware
- Asigură interfțare cu utilizatorul și alte sisteme

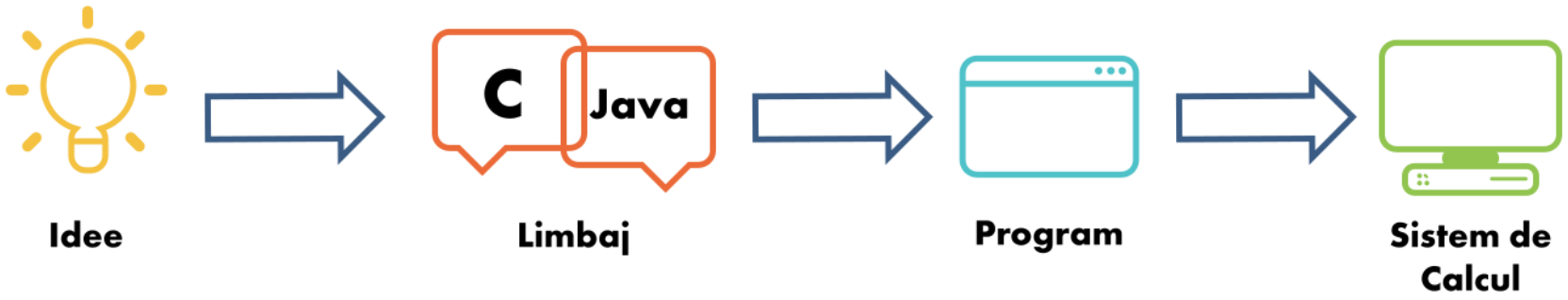
- Compute / calcul (CPU, procesor)
- Stocare date / cod (RAM, memorie)
- Interfațare cu exteriorul (I/O, dispozitive periferice)

- Avantaj: stabil, standard
- Dezavantaj: rigid, inflexibil

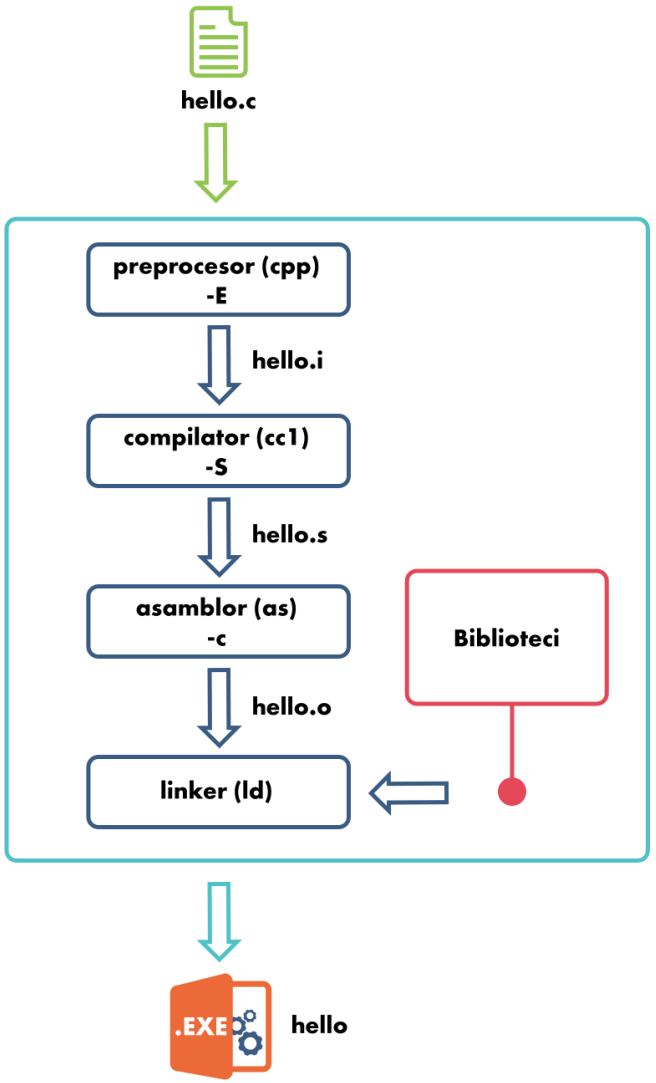
# De la hardware la software

- Utilizatorul are o nevoie / idee
- Dezvoltă o aplicație / program
  - Într-un limbaj de programare
- Programul este convertit (compilat) în cod mașină
- Programul în cod mașină este încărcat în memorie (loaded) și executat (run-time)

# Idee, program, sistem de calcul



# De la cod sursă la executabil



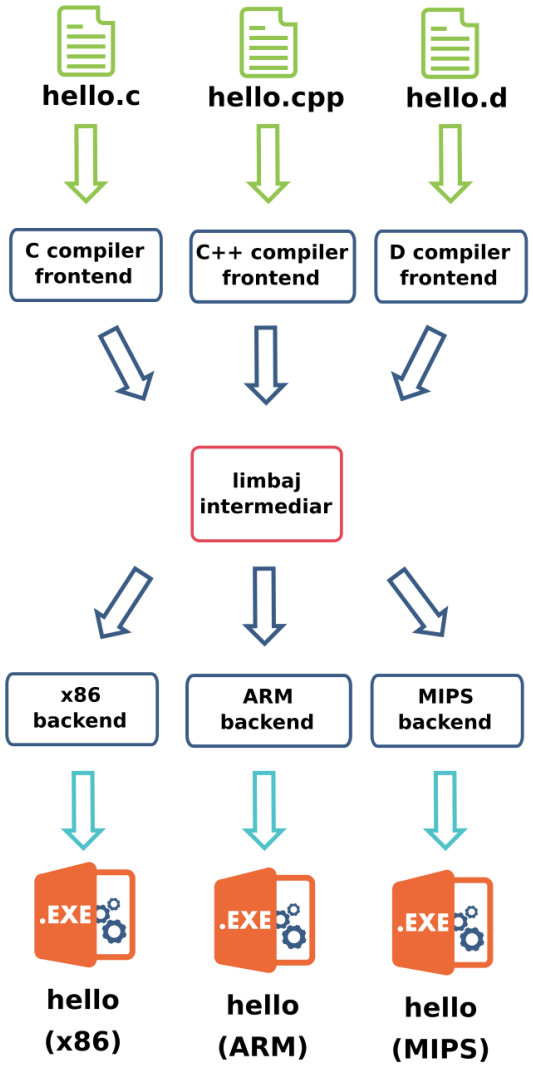
# Interfața software-hardware

- Software: colecție de date și instrucțiuni
- Instrucțiuni trebuie să fie înțelese de hardware (de procesor)
  - Procesorul interpretează instrucțiunile
  - O instrucțiune este decodificată și executată
- Instrucțiunile sunt secvențe binare cunoscute procesorului
  - cod mașină / machine code
- Fișierele executabile, fișierele obiect, bibliotecile conțin cod mașină

# Limbaje și compilatoare

- Scriem cod sursă într-un limbaj (de nivel înalt)
  - C, C++, Java, Python
- Codul nu este rulabil pe procesor
  - Trebuie convertit în cod mașină
- Compilatorul convertește / traduce codul sursă în cod mașină

# Funcționarea compilatorului



# Rolul compilatorului

- Ascunde complexitatea codului mașină
  - Dezvoltatorul este preocupat doar de limbajul de nivel înalt
- Portabilitate între diferite arhitecturi de procesor
- Optimizări

# Arhitecturi de procesor

- x86, ARM, MIPS, PowerPC
- Interfața expusă de procesor către software
- În general numite ISA
  - Instruction Set Architecture
- Specificația codului mașină
- Un compilator poate genera fișiere cod mașină pentru diferite ISA de la același cod sursă
- Mai multe în următorul curs

# Memoria și procesorul

- Memoria stochează date și instrucțiuni cod mașină
- Procesorul:
  - Preia instrucțiuni din memorie
  - Decodifică instrucțiunile
  - Dacă este cazul, preia date din memorie
  - Execută instrucțiunile decodificate
  - Dacă este cazul scrie rezultatul în memorie
- Datele din memorie provin de la
  - Program (la încărcarea programului – loading)
  - I/O (citire de la / scriere la dispozitive periferice)

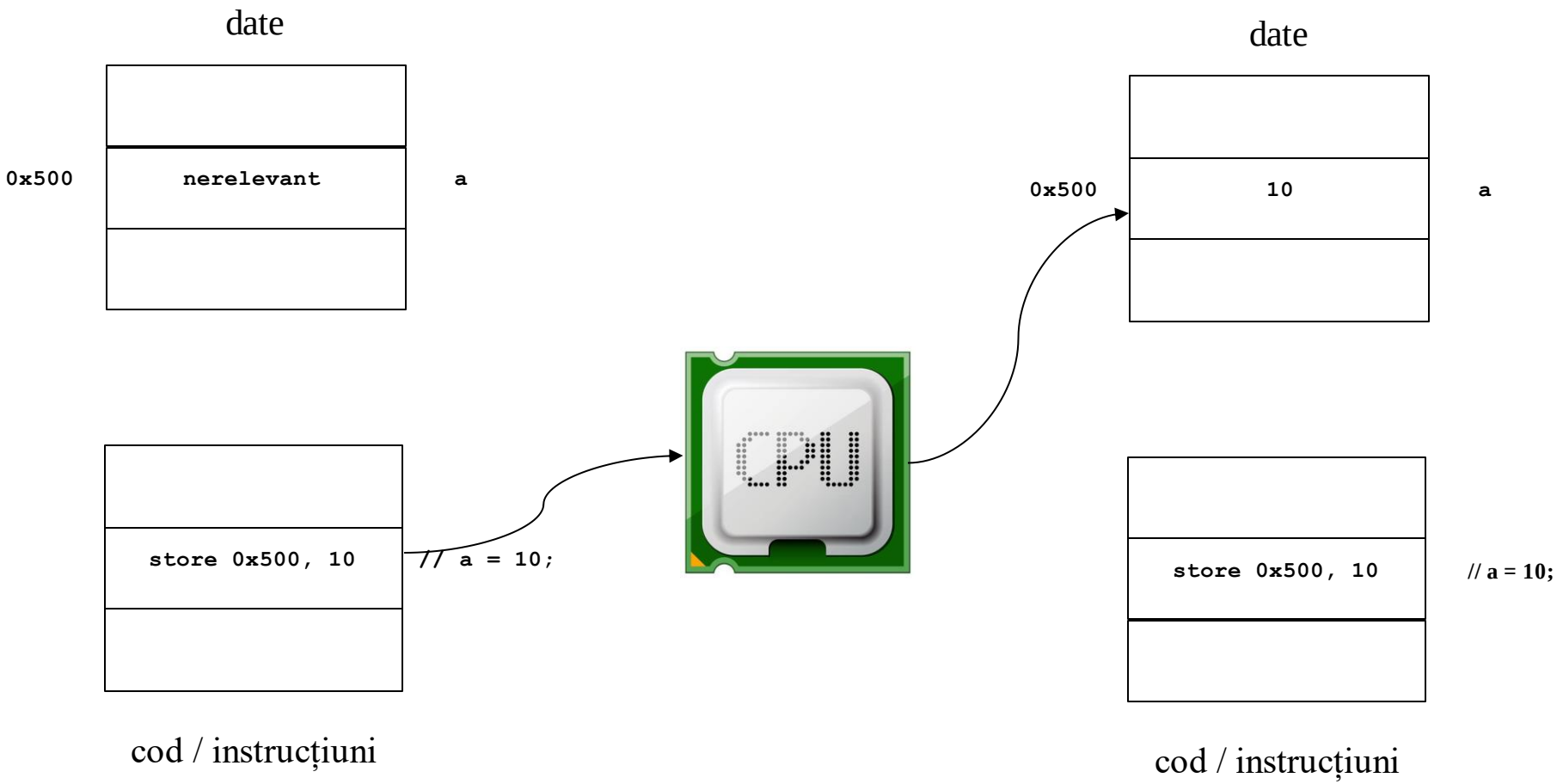
# De la C la memorie și procesor

```
a = 10;
```

- Cum se "prezintă" instrucțiunea C de mai sus în memorie și procesor?

- În zona de date din memorie există un loc pentru variabila a (memory location)
- În zona de cod există o instrucțiune care atribuie valoarea 10 în zona de memorie a variabilei a
- Procesorul va citi și va executa instrucțiunea
- La final, în zona de memorie va fi valoarea 10

# De la C la memorie și procesor (2)



# De ce să știu asta?

- Informal: Pentru că ești inginer de calculatoare, în pana mea!
- Mai formal: Pentru că vei înțelege ce se întâmplă cu aplicația ta, cum folosește resursele hardware.
  - Devii un programator mai bun.
  - Piloții foarte buni sunt și mecanici foarte buni.
  - Te vei simți mai confortabil cu aplicații mari, cu sisteme complexe: procesoare multiple, calcul eterogen
- Vei crea aplicații / sisteme mai sigure, mai robuste, mai performante
- Vei depana mai ușor ce nu funcționează

# Law of Leaky Abstractions

- Joel Spolsky, 2002
- https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/

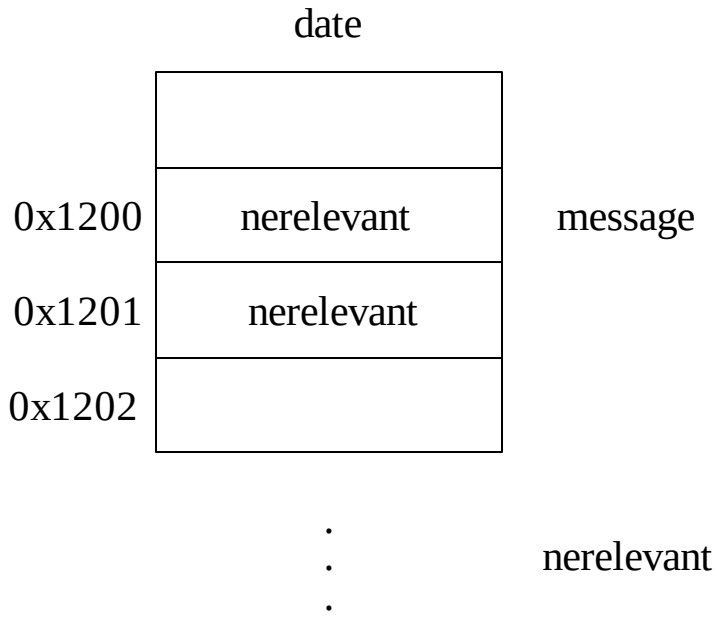
"And while these great tools, like modern OO forms-based languages, let us get a lot of work done incredibly quickly, suddenly one day we need to figure out a problem where the abstraction leaked, and it takes 2 weeks."

# `char message[128];`

- Ce efect are în memorie și procesor?

- Nici un efect în procesor, nu execută nimic.
- 128 de octeți în memorie, în zona de date

# char message[128]; (2)



# 0x500, 0x1200

- Adrese de memorie
- `0x500` – adresa variabilei `a`
- `0x1200` – adresa de început a vectorului `message`

# Ce este, din punct de vedere logic, memoria?

- Un vector de octeți
- Fiecare octet are o adresă

- Ce este o adresă de memorie?
  - Un index în vectorul de octeți ce reprezintă memoria
  - Adresa 0x500 este al 0x500-le octet din memorie
  - În general adresele le scriem în hexazecimal

# Ce este o variabilă?

- O zonă de memorie
  - O adresă (de start)
  - O dimensiune (număr de octeți)

- `int a;`
  - Începe de la o adresă dată
  - Ocupă 4 octeți

- char c;
  - Începe de la o adresă dată
  - Ocupă 1 octet

# Ce este o variabilă (2)

- `char message[128];`
  - Începe de la o adresă dată
  - Ocupă 128 de octeți

- `int key[256];`
  - Începe de la o adresă dată
  - Ocupă 256 * 4 = 1024 de octeți

# Valoarea, adresa și dimensiunea unei variabile

- `a = 10;`
  - În locul ocupat de variabila `a` (adresă, dimensiune) scriem 10
- `b = a;`
  - În locul ocupat de variabila `b` (adresă, dimensiune) scriem valoarea din locul ocupat de variabila `a` (adresă, dimensiune)
- `&a`
  - adresa variabilei `a`
- `sizeof(a)`
  - dimensiunea variabilei `a`

# Demo: Vizualizarea variabilelor

- https://github.com/systems-cs-pub-ro/iocla
- curs-01-prog/demo/inspect_vars.c
- make
- ./inspect_vars

# Valoarea unui vector

```
int key[256];
```

- Ce este `key`?
- Ce este `&key`?
- Ce este `key + 10`?
- Ce este `key[10]`?
- Ce este `&key[10]`?
- Cât este `sizeof(key)`?
- Cât este `sizeof(key[10])`?

# Demo: Adrese și valori de vectori

- https://github.com/systems-cs-pub-ro/iocla
- curs-01-prog/demo/inspect_array.c
- make
- ./inspect_array

# Ce este un pointer?

- O variabilă (adresă de start, valoare, dimensiune)
- Valoarea pointerului este o adresă
- int a = 10;
  - a este o variabilă întreagă ce are valoarea 10
- int *p = 10;
  - p este o variabilă de tip pointer ce are valoarea 10
  - adică referă adresa de memorie 10
- int *p = &a;
  - p este o variabilă de tip pointer ce are ca valoare adresa variabilei a
  - adică referă variabila a (zona ei de memorie)

# Pointer vs Adresă

- Adresa este un index în memorie
  - orice informație stocată în memorie are o adresă
  - variabile, funcții
- Un pointer este o variabilă care reține o adresă
  - Pentru că un pointer este o variabilă, are și acesta o adresă

- int *p = &a;
  - p este o variabilă pointer ce reține adresa variabilei a
- &p
  - adresa variabilei p
  - ca orice variabilă, p are o adresă

# Demo: Adrese și pointeri

- https://github.com/systems-cs-pub-ro/iocla
- curs-01-prog/demo/inspect_ptr.c
- make
- ./inspect_ptr

# &a = 10; - De ce nu funcționează?

- x = y;
  - La adresa variabilei x se pune valoarea de la adresa variabilei y
- x = 10;
  - La adresa variabilei x se pune valorea 10
- 10 = x;
  - Nu se poate, 10, nu este o variabilă, nu are o zonă de memorie
- &a
  - este o valoare, nu este o variabilă
- &a = x;
  - similar cu 10 = x; nu funcționează

# Ce este o funcție?

- O adresă într-o zonă de memorie de cod
- Echivalentul unui vector/pointer read-only
- main
  - adresa funcției main
- &main
  - Tot adresa funcției main
- void (*f)(void) = main;
  - f este variabilă de tip pointer de funcție
  - reține adresa functiei main
- jmp main (în limbaj de asamblare)
  - se duce execuția la adresa funcției main

# Demo: Funcții

- https://github.com/systems-cs-pub-ro/iocla
- curs-01-prog/demo/inspect_func.c
- make
- ./inspect_func

# Adrese valide / nevalide

- "alocarea" memoriei
  - zona respectivă devine "validă"
- validă
  - accesele la acea zonă funcționează
- nevalidă
  - accesul cauzează eroare
  - "excepție de acces la memorie"
  - "invalid memory access"
  - segmentation fault

# Segmentation fault

- Acces nevalid la memorie
- Dereferențiem o variabilă de tip pointer cu adresă nevalidă
  - Cel mai simplu: int *p = NULL; *p = 10;

# Demo: Segmentation fault

- https://github.com/systems-cs-pub-ro/iocla
- curs-01-prog/demo/segfault.c
- make
- ./segfault
- dmesg

# Demo: godbolt.org

- https://godbolt.org/

- Putem vedea cum se traduce un program în limbaj de asamblare
  - wrapper peste cod mașină

# Demo: objdump

- Utilitar de inspecție a executabilelor
- Face dezasamblare
  - Traduce înapoi codul mașină în cod în limbaj de asamblare
  - Util pentru inginerie inversă (reverse engineering)

- Afișează adrese din executabil

Click to add text

- objdump -d -M intel inspect_vars | grep -A 30 'main>'

# Next on IOCLA

- Ce este limbajul de asamblare?
- De ce este nevoie de limbaj de asamblare?
- Arhitectura unui sistem de calcul
  - Modelul von Neumann
  - interacțiunea procesor-memorie
- Arhitectura x86

# Cuvinte cheie

- Hardware
- Software
- Cod sursă
- Cod mașină
- ISA
- Memorie
- Adresă de memorie
- Variabilă
- Pointer

# Intrebări?