STRUCTURI DE DATE

Limbajul C pentru structuri de date

EVALUARE

SEMESTRU: 4 puncte

1. Implementare practica prin cod sursa (2 puncte)

2. Evaluari pe parcurs (2 puncte)

EVALUARE

EXAMEN: 6 puncte

Test PRACTIC (oral la calculator): 6 puncte.

Evaluari acordate DOAR pentru aplicatii executabile (“duse” pana in faza de executie) cu urmatoarele caracteristici:

- Compilare (no compile-time errors).
- Executie (no run-time errors).
- Exemple de test (inclusiv valori “extreme” pentru validarea datelor de I/O).

BIBLIOGRAFIE

Marius Popa, Cristian Ciurea, Mihai Doinea, Alin Zamfiroiu – “Structuri de date – teorie si practica”, editura ASE, 2023

Ion Ivan, Marius Popa, Paul Pocatilu (coord.) - “Structuri de date”, vol. 1, vol. 2, 2009

Ion Smeureanu, Marian Dardala – “Programarea in limbajul C/C++”, Editura CISON, 2004

http://www.acs.ase.ro
https://github.com/mpopaeu/structuri

BIBLIOGRAFIE

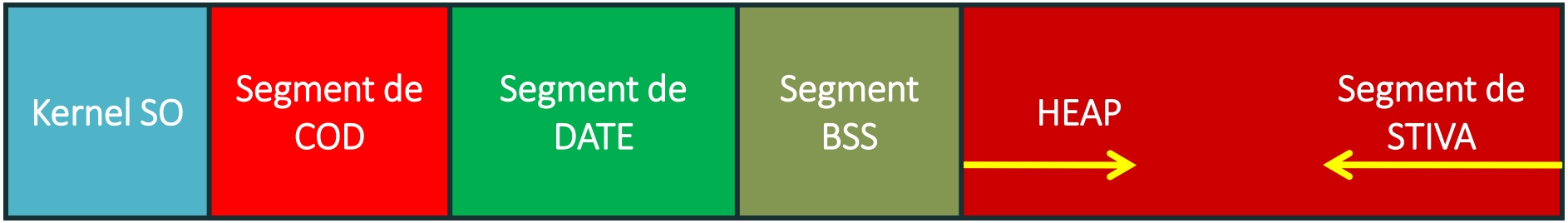
Bjarne Strastroup – The Creator of C++, “The C++ Programming Language” – 3rd Edition, Editura Addison-Wesley

http://www.acs.ase.ro

https://github.com/mpopaeu/structuri

MEMORIE

Organizarea memoriei la executia unui proces



Segment de COD

- Instructiuni executabile (binare).

- Read-Only – nu pot fi scrise secvente binare, deci nu poate fi o bresa de securitate (generare eroare acces memorie).

- Partajat intre utilizatori care executa concurent codul binar al aplicatiei.

Segment de DATE

- Date alocate static (clasa de memorie) si date globale initializate cu valori in codul aplicatiei.

- Fiecare proces contine propriul segment de date.

- Nu este segment executabil.

Segment BSS (Block Started by Symbol)

- Date alocate static (clasa de memorie) si date globale neinitializate prin codul aplicatiei (implicit au valori nule).

- Nu este segment executabil.

Segment STACK

- Variabile locale – definite in functii.

- Transfer parametri in functii.

- Localizata la capatul memoriei accesibile => alocare descrescatoare a adreselor.

- Management stack segment (registri procesor):

1. **SP (Stack Pointer** procesor de 16 biti) / **ESP (Extended Stack Pointer** procesor de 32 biti) / **RSP (Register Stack Pointer** procesor 64 biti).

2. **BP (Base Pointer) / EBP (Extended Base Pointer) / RBP (Register Base Pointer)**.

Segment STACK

- Utilizat pentru variabilele locale, valori temporare, argumente functii, adrese de return.

- Implementat pe prinicipiile structurii de tip stiva.

- Dimensiune cunoscuta la momentul compilarii app si alocata la momentul de incepere a executiei app.

- Stocare date la nivel de cuvant calculator – unitate de date de lungime fixa folosita de procesor conform setului de instructiuni ale acestuia. Registrii procesorului sunt lungime cuvant calculator (8, 16, 24, 32 sau 64 biti).

Segment HEAP

- Alocare memorie pe durata de executie a aplicatiei.

- Alocare gestionata de sistemul de operare.

- Zone de memorie gestionate prin variabile de tip **pointer**.

Segment HEAP

- Memorie alocata dinamic (pe dimensiunea cunoscuta la executia app).

- Accesat prin pointeri si are un continut anonim;

- Dimensiune pana la pointerul break. Poate fi modificata (adaugare) prin cerere catre sistemul de operare;

- Posibilitate de crestere a dimensiunii prin mutarea pointerului break spre adrese mai mari;

- Fragmentare ridicata prin operatii multiple de alocare / dealocare.

- Responsabilitate dezalocare (**memory leaks**).

Memory Leaks

- Apps consumatoare de memorie care nu este eliberata catre sistemul de operare si care devine inaccesibla pentru sistemul de operare.

- Determina cresterea cerintelor de memorie disponibilizata pentru app.

- Efecte: reducere performante computer prin reducerea cantitatii de memorie disponibila, app & system failures etc.

- Sisteme de operare moderne: memorie eliberata automata la terminarea executiei (in termen scurt) app.

CLASE DE MEMORIE ALE VARIABILELOR

AUTOMATICE (specificator **auto)**:

- Locale pentru blocul de instructiuni in care se definesc variabilele.

- Persistente pana la terminarea blocului de instructiuni in care se definesc.

- Zone de memorie distincte pentru cod recursiv sau multithreading.

- Stocate in ***segmentul de stack***.

Exemple:

```
auto a = 7;
auto b = "VariabileAuto";
```

CLASE DE MEMORIE ALE VARIABILELOR

REGISTRU (specificator **register)**:

- Specificator utilizat doar pentru variabile locale si parametri ai functiilor.

- Persistente pana la terminarea blocului de instructiuni in care se definesc (similar specificatorului auto).

- Decizia compilatorului de incarcare a unui registru cu continut variabila.

- Utile pentru operatii frecvente din punctul de vedere al reducerii timpului de acces si executie;

Exemplu:

```
register int vreg;
int d;
d = 8;
vreg = d;
```

CLASE DE MEMORIE ALE VARIABILELOR

EXTERNE (specificator **extern**):

- Utilizate pentru variabile declarate in mai multe fisiere sursa.
- Memorie alocata inainte de executia functiei main.
- Persistenta pana la terminare executiei programului.
- Definite in bloc de instructiuni cu accesibilitate in cadrul blocului; altfel, accesibila la nivel de fisier sursa;

Exemplu:

Fisierul 1

```
extern int i;
void f() {
   i++;
}
```

Fisierul 2

```
int i = 0;
extern void f();
void g() {
   f();
   printf("%d\n", i);
}
```

CLASE DE MEMORIE ALE VARIABILELOR

STATICE (specificator **static**):

- Persistenta continut si vizibilitate in bocul unde sunt definite (chiar si la nivel de fisier). La nivel de fisier individual, pot fi asimilate variabilelor globale, dar vizibilitatea este diferita intr-o colectie de fisiere sursa (abordare static vs global de catre linker).
- Alocate la inceperea executiei programului si dezalocate la terminare executiei.
- Declararea intr-o functie asigura persistenta continutului intre apeluri.

Exemplu:

```
int f() {
  static int x = 0;
  x++;
  return x;
}
```

```
void main() {
  int j;
  for (j = 0; j < 10; j++) {
     printf("Rezultat functie f: %d\n", f());
  }
}
```

MEMORIE

Sursa cod C/C++

```
#include<stdio.h>

char a = 7, b = 9;
short int c = 0;

void main()
{
        c = a+b;
}
```

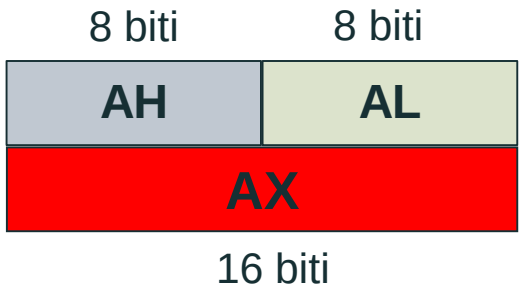
Reprezentare cod ASM

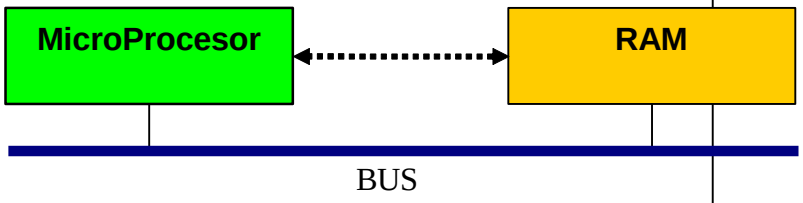
```
.model small
.stack 16
.data
        a db 7
        b db 9
        c dw 0
.code
start:
        mov AX, @data
        mov DS, AX

        mov AL, a
        add AL, b
        mov c, AX

        mov AX, 4C00h
        int 21h
end start
```

Segment cod

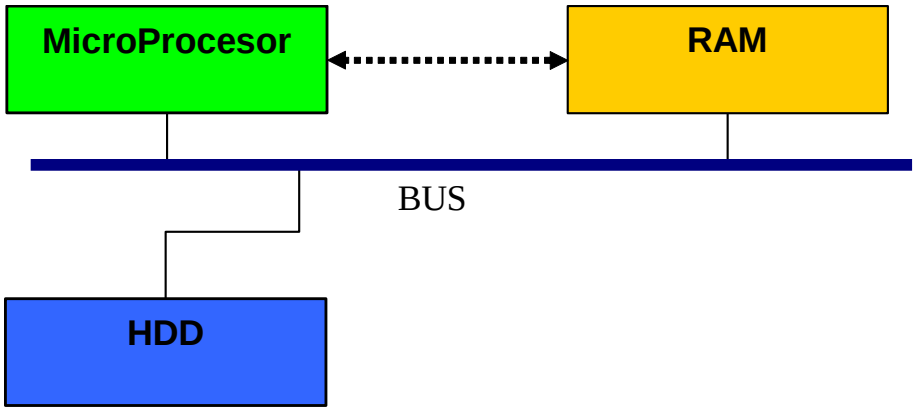
```
B8 02 00 8E D8
A0 00 00 02 06
01 00 A3 02 00
B8 00 4C CD 21
00 00 00…..00 00
07 09 00 00
```

Segment date



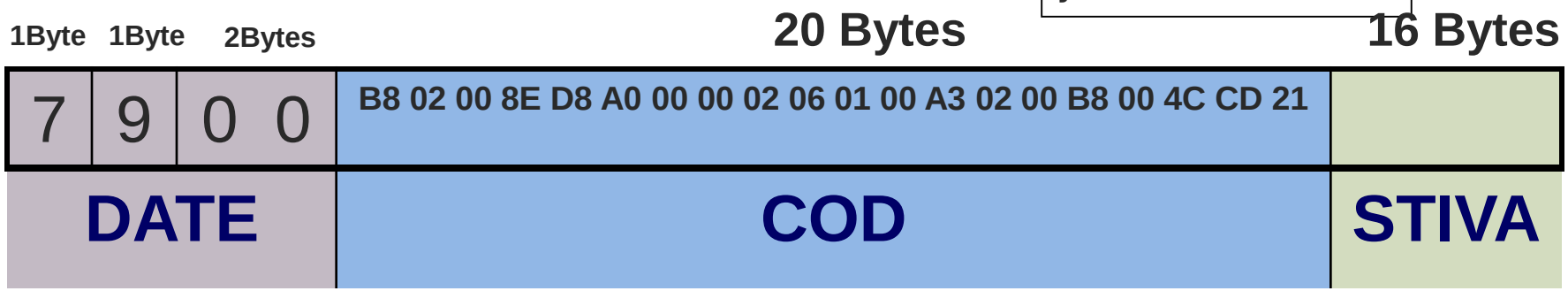


MEMORIE



Sursa C/C++

```
#include<stdio.h>

char a = 7, b = 9;
short int c;

void main()
{
        c = a+b;
}
```



Tipuri de date

Sursa definirii:

Fundamentale, definite in cadrul limbajului;

Definite de utilizator (programator/dezvoltator etc); exemple: structuri articol, clase de obiecte, structuri pe biti, uniuni etc.

Natura continutului:

- Simple, un element unic conform tipului de date.
- Masive, agregate si accesibile prin indecsi.
- Pointeri, acces explicit la zone de memorie.

Tipuri de date

Descriere tipuri fundamentale C/C++:

Denumire	Explicatie	Dimensiune in bytes	Interval valori posibile
char	Intreg de valoare mica, inclusive ASCII.	1 byte	cu semn: [-128, 127] fara semn: [0, 255]
short int (short)	Intreg short.	2 bytes	cu semn: [-32768, 32767] fara semn: [0, 65535]
int	Intreg.	4 bytes	cu semn: [-2147483648, 2147483647] fara semn: [0, 4294967295]
long int (long)	Intreg long.	4 bytes	cu semn: [-2147483648, 2147483647] fara semn: [0, 4294967295]
long long (int)	Intreg long long	8 bytes	cu semn: [–9223372036854775807, +9223372036854775807] fara semn: [0, 18446744073709551615]

Tipuri de date

Descriere tipuri fundamentale C/C++:

Denumire	Explicatie	Dimensiune in bytes	Interval valori posibile
float	Real virgula mobila precizie simpla.	4 bytes	+/- 3.4e +/- 38
double	Real virgula mobila precizie dubla.	8 bytes	+/- 1.7e +/- 308
long double	Real virgula mobila precizie extinsa.	8 bytes / **10 bytes** / 16 bytes (functie de compilator)	x86 10 bytes [3.4e-4932, 1.1e+4932]

Tipuri de date

Reprezentare interna tipuri reale (lungimi zone in biti):

Denumire	Semn	Exponent	Mantisa	Total biti	Bias exponent
float	1	8	23	32	127
double	1	11	52	64	1023
long double	1	15	64	80	16383

POINTERI

Date numerice utilizate pentru a gestiona valori reprezentand adrese.

Dimensiune este data de arhitectura procesorului. Pointeri **near** si **far**.

Definire:

```
tip_data * nume_pointer;
```

Initializare:

```
nume_pointer = & nume_variabila;
```

Utilizare:

```
nume_variabila = * nume_pointer;
```

POINTERI

Exemple:

```
int *pi; // pointer la int
char **ppc; // pointer la pointer de char
int *ap[10]; // vector de 10 pointeri la int
```

Valoarea 0 pentru un pointer este o valoare nula. Aceasta este asociata cu simbolul

```
#define NULL 0
```

sau cu constanta

```
const int NULL = 0;
```

POINTERI

Probleme initializare pointer cu 0 sau NULL:

- Imposibilitatea de a face diferenta dintre tip intreg (valoarea 0) si tip pointer in functiile supraincarcate.

- Imposibilitatea de a face diferenta dintre constanta 0 si macrodefinitia NULL in functiile supraincarcate.

```
void func(int* i) { printf("Call func(int*)\n"); }

void func(int i) { printf("Call func(int)\n"); }

int main() {
        func(NULL); // Call func(int);
}
```

POINTERI

Solutia: **nullptr** (C++11)

- **Constanta explicita** pentru pointer nul.

- **Diferenta** intre constanta intreaga 0 si constanta nullptr pentru functiile supraincarcate.

```
void func(int* i) { printf("Apel func(int*)\n"); }

void func(int i) { printf("Apel to func(int)\n"); }

int main() {
      func(nullptr); // Call func(int*);
      func(NULL);    // Call func(int);
}
```

POINTERI

Aritmetica pointerilor:

Pentru un pointer de tip T*, operatorii --/++ asigura deplasarea inapoi/inainte cu sizeof(T) bytes.

Pentru un pointer de tip T*, expresia pt + k sau pt – k este echivalenta cu deplasarea peste k * sizeof(T) bytes.

Diferenta dintre 2 pointeri din interiorul aceluiasi sir de valori reprezinta numarul de elemente (de tipul aferent pointerului) dintre cele doua adrese.

Adunarea dintre 2 pointeri nu este permisa.

POINTERI

Pointeri constanti

```
int * const p;          // pointer constant la int
int const *pint;        // pointer la int constant
const int *pint2;       // pointer la int constant
const int * const pint2;    // pointer constant la int constant
```

Exemplu:

Utilizare:

```
char* strcpy(char*p, const  char*q);
```

POINTERI

Alocare dinamica memorie:

- Functii: **malloc**
- Operatorul **new** sau **new [ ]** (C++)
- Rezerva memorie in **HEAP**

Dezalocare memorie:

- Functie: **free**
- Operatorul **delete** sau **delete [ ]** (C++)
- Elibereaza memoria rezervata in **HEAP**

FUNCTII

Caracteristici

- Secventa de cod sursa cu caracter general si repetitiv.
- Accepta parametri de intrare si returneaza rezultate.
- Definirea imbricata nu este permisa in C/C++.
- Transferul parametrilor de intrare: valoare, adresa, variabile globale, utilizare referinta.
- Parametri copiati in zone de memorie organizate ca stive.
- Rezultatul returnat: tip de retur, argumente transmise prin adresa.

FUNCTII

Declararea si construirea unei functii:

Declarare antet functie

```
TipRezultat DenFunctie([ListaParametriFormali]);
```

Standard/Utilizator

Implicit int/void

Masiv NU!

Identificator functie

```
TipRezultat DenFunctie([ListaParametriFormali]) {
    // corp functie
}
```

Parametrii formali sub forma

[tip$_i$ p$_i$[,…]]

- **declaratii locale**
- **instructiuni**
- **apeluri subprograme**
- **instructiune *return***

FUNCTII

Exemplu functie:

Sursa C/C++

```
#include<stdio.h>

double Suma1(float x, float y)
{
        double s;
        s = x + y;
        return s;
}
```

```
#include<stdio.h>

void Suma2(float x, float y,
            float *z)
{
          *z = x + y;
}
```

Apel functii C/C++

```
…
float a = 1.2, b = 4.7, c;
…
c = Suma1(a, b);
…
```

```
…
float a = 1.2, b = 4.7, c;
…
Suma2(a, b, &c);
…
```

POINTERI LA FUNCTII

Definire

```
tip_return (* den_pointer) (lista_parametri);
```

Initializare

```
den_pointer = den_functie;
```

Apel functie prin pointer

```
den_pointer (lista_parametri);
```

POINTERI LA FUNCTII

```
float (*fp)(int *);  // pointer la functie ce primeste un
                     // pointer la int si ce returneaza un float

int * f(char *);  // functie ce primeste char* si returneaza
                  // un pointer la int

int * (*fp[5])(char *);  // vector de 5 pointeri la functii ce
                         // primesc char* si returneaza
                         // un pointer la int
```

PREPROCESARE

- Etapa ce precede compilarea.
- Bazata pe simboluri definite prin operatorul #
- NU reprezintă instrucţiuni executabile.
- Determina compilarea condiţionata a unor instrucţiuni.
- Substituire simbolica.
- Tipul enumerativ.
- Macrodefinitii.

PREPROCESARE

Substituire simbolica:

Bazata pe directiva #define

```
#define NMAX 1000
#define then
#define BEGIN {
#define END }

void main()
BEGIN
int vb = 10;
int vector[NMAX];
if(vb < NMAX) then printf(“mai mic”);
else printf(“mai mare”);
END
```

PREPROCESARE

Substituire simbolica:

Valabilitate simbol:

Sfarsit sursa.

Redefinire simbol.

Invalidare simbol:

```
#define NMAX 1000
....
#define NMAX 10
...
#undef NMAX
```

PREPROCESARE

Tipul enumerativ:

enum denumire {lista simboluri} lista variabile

Valorile sunt in secventa.

Se poate preciza explicit valoarea fiecarui simbol

```
enum rechizite {carte , caiet , creion = 4, pix = 6, creta}
```

PREPROCESARE

Macrodefinitii:

#define nume_macro(lista simboluri) expresie

Exemplu:

```
#define PATRAT(X) X*X
#define ABS(X) (X) < 0 ? - (X) : (X)
```

Sursa C/C++

```
...
int x=PATRAT(3);
int y=PATRAT(3+2);
...
```

PREPROCESARE

Macrodefinitii generatoare de functii:

```
#define SUMA_GEN(TIP) TIP suma(TIP vb2, TIP vb2) \
        { return vb1 + vb2; }
```

Compilare conditionata:

```
#if expresie_1
                secventa_1
#elif expresie_2
                secventa_2
...
#else
                secventa_n
#endif
```

PREPROCESARE

Compilare conditionata:

```
#ifdef nume_macro
        ...
#else
        ...
#endif
```

sau

```
#ifndef nume_macro
        ...
#endif
```

PREPROCESARE

Operatorii # si ##:

Utilizati impreuna cu **#define**

Operatorul # (de insiruire) transforma argumentul intr-un sir cu “”

```
#define macro1(s) # s
```

Operatorul ## (de inserare) concateneaza 2 elemente

```
#define macro2(s1, s2)  s1 ## s2
```