

## **Curs 2 – Programare modulară în C**

- Funcții – Test driven development, Code Coverage**
- Module – Programare modulara, TAD**
- Gestiunea memoriei in C/C++**

## **Curs 1**

- Introducere - OOP
- C Programming language
  - sintaxa
  - tipuri de date, variabile, instrucțiuni

## Funcții

### Declarare (Function prototype)

<result type> name ( <parameter list>);

<result-type> - tipul rezultatului, poate fi orice tip sau *void* *daca funcția nu returnează nimic*

<name> - numele funcției

<parameter-list> - parametrii formali

Corpul funcției nu face parte din declarare

```
/**
 * Computes the greatest common divisor of two positive integers.
 * a, b integers, a,b>0
 * return the the greatest common divisor of a and b.
 */
int gcd(int a, int b);
```

# Funcții

## Definiție

```
<result type> name(<parameter list>){  
//statements - the body of the function  
}
```

- **return** <exp> rezultatul expresiei se returnează, execuția funcției se termina
- o funcție care nu este void trebuie neapărat să returneze o valoare prin expresia ce urmează după **return**
- declararea trebuie să corespundă cu definiția (numele parametrilor poate fi diferit)

```
/**  
 * Computes the greatest common divisor of two positive integers.  
 * a, b integers, a,b>0  
 * return the the greatest common divisor of a and b.  
 */  
int gcd(int a, int b) {  
    if (a == 0 || b == 0) {  
        return a + b;  
    }  
    while (a != b) {  
        if (a > b) {  
            a = a - b;  
        } else {  
            b = b - a;  
        }  
    }  
    return a;  
}
```

Funcția **main** este executat când lansăm în execuție un program C/C++

## Specificații

- Nume sugestiv
- O scurtă descriere a funcției (ce face)
- Semnificația parametrilor
- condiții asupra parametrilor (precondiții)
- ce se returnează
- relația dintre parametri și rezultat (post condiții)

```
/*  
* Verify if a number is prime  
* nr - a number, nr>0  
* return true if the number is prime (1 and nr are the only dividers)  
*/  
int isPrime(int nr);
```

**precondiții** - sunt condiții care trebuie să fie satisfăcute de parametrii actuali înainte de a executa corpul funcției

**postcondiții** - condiții care sunt satisfăcute după execuția funcției

## Apelul de funcții

name (<parameter list>);

- Toate expresiile date ca parametru sunt evaluate înainte de execuția funcției
- Parametrii actuali trebuie sa corespundă cu parametri formal (număr, poziție, tip)
- declarația trebuie sa apară înainte de apel

```
int d = gcd(12, 6);
```

## **Vizibilitate (scope)**

Locul unde declarăm variabila determină vizibilitate lui (unde este variabila accesibilă).

### **Variabile locale**

- variabila este vizibilă doar în interiorul instrucțiunii compuse (`{ }`) unde a fost declarată
- variabilele declarate în interiorul funcției sunt vizibile (accesibile) doar în funcție
- Încercarea de a accesa o variabilă în afara domeniului de vizibilitate generează eroare la compilare.
- Ciclul de viață a unei variabile începe de la declararea lui și se termină când execuția iese din domeniul de vizibilitate a variabilei (variabila se distruge, memoria ocupată se eliberează)

### **Variabile globale**

- Variabilele definite în afara funcțiilor sunt accesibile în orice funcție, domeniul lor de vizibilitate este întreg aplicația
- Se recomandă evitarea utilizării variabilelor globale (există soluții mai bune care nu necesită variabile globale)

## Transmiterea parametrilor : prin valoare sau prin referință

### Transmitere prin valoare: `void byValue(int a);`

La apelul funcției se face o copie a parametrilor.

Schimbările făcute în interiorul funcției nu afectează variabilele exterioare.

Este mecanismul implicit de transmitere a parametrilor în C

### Transmitere prin „referință” : `void byRef(int* a);`

La apelul funcției se transmite o adresă de memorie (locația de memorie unde se află valoarea variabilei).

Modificările din interiorul funcției sunt vizibile și în afară (modificam valorile de la același adresa de memorie).

```
void byValue(int a) {
    a = a + 1;
}
void byRef(int* a) {
    *a = *a + 1;
}
void testArrayParam(int a[]){
    a[0] = 3;
}
int main() {
    int a = 10;
    byValue(a);
    printf("Value remain unchanged a=%d \n", a);
    byRef(&a);
    printf("Value changed a=%d \n", a);
    int a[] = {1,2,3};
    testArrayParam(a);
    printf("value is changed %d\n",a[0]);
    return 0;
}
```

Vectorul este transmis prin **referință** (se transmite adresa de început al vectorului)

## Valoarea returnată de funcție

### Built in types (se returnează o valoare simplă)

Pentru tipurile predefinite (int, char, double, etc.) se returnează o copie.

### Pointer (se returnează o adresă de memorie)

**Nu returnați adresa unei variabile locale.** Memoria alocată de compilator pentru o variabilă este eliberată (devine invalid) în momentul în care se termină execuția funcției (variabila nu mai este vizibilă)

### Vector

nu se poate returna un vector (**int**[]) dintr-o funcție. Se poate returna un pointer **int\*** (adresa primului element). Obs. Nu returnați adresa de memorie de la variabile locale (alocate de compilator și distruse la ieșirea din funcție)

### Struct

Se comportă ca și valorile simple (int, char, double, etc)

Dacă tipul returnat este un struct, se creează o copie și acesta se returnează

Dacă folosim operatorul de asignment (=) se face o copie a struct-ului din dreapta

Dacă struct-ul conține pointeri (char\*), se copiază adresa, nu și memoria referită de pointer. După copiere cele două struct-uri vor referi același zonă de memorie.

Dacă struct-ul conține vectori (char[20]) se copiază întreg vectorul (20 de caractere). După copiere cele două struct-uri au doi vectori independenți.



## Copiere de valori in C

O valoare se poate copia:

- Folosind operatorul = (assignment) `a=b;`
- La transmitere ca parametru unei funcții
- La returnarea unei valori dintr-o funcție

Tipurile simple (`char`, `int` `double`): se copiază valoarea

**Pointeri** (`int*`, `char*`, etc): se copiază adresa, in urma copierii cele doua variabile refera același adresa de memorie. Valabil si daca avem un pointer in interiorul unui struct.

**Struct** (`struct {int a, int t[10], int* p}`): se copiază bit cu bit fiecare câmp din struct. Daca am o valoare se face o copie, daca am un pointer se copiază adresa, daca am un vector se copiază tot vectorul element cu element.

**Excepții de la regulile de copiere – vector static (`int[10]`,`char[10]`, etc):**

La transmiterea unui vector ca parametru la funcție se transmite adresa de început a vectorului. **Array decay**: vectorul e transmis ca un pointer. Astfel modificările vectorului in interiorul funcției se reflecta si in afara.

Nu se pot returna vectori dintr-o funcție

Nu se poate face atribuire (=) la vectori

## **Proces de dezvoltare incrementală bazată pe funcționalități**

- Se creează lista de funcționalități pe baza enunțului
- Se planifică iterațiile (o iterație conține una/mai multe funcționalități)
- Pentru fiecare funcționalitate din iterație
  - Se face modelare – scenarii de rulare
  - Se creează o lista de tascuri (activități)
  - Se implementează și testează fiecare activitate

## **Dezvoltare dirijată de teste (test-driven development - TDD)**

Dezvoltarea dirijată de teste presupune crearea de teste automate, chiar înainte de implementare, care clarifică cerințele

Pașii TDD pentru crearea unei funcții:

- Adaugă un test – creează teste automate
- Rulăm toate testele și verificăm ca noul test pică
- Scriem corpul funcției
- Rulăm toate testele și ne asigurăm că trec
- Refactorizăm codul

# Funcții de test

## Assert

```
#include <assert.h>
void assert (int expr);
```

expr – Se evaluează expresia. Dacă e fals ( $\neq 0$ ) metoda assert generează o eroare și se termină execuția aplicației

Mesajul de eroare depinde de compilator (pot fi diferențe în funcție de compilator), conține informații despre locul unde a apărut eroarea (fișierul, linia), expresia care a generat eroare.

Vom folosi instrucțiunea assert pentru a crea teste automate.

<pre>#include &lt;assert.h&gt; /*  * greatest common divisor .  * Pre: a, b &gt;= 0, a*a + b*b != 0  * return gcd  */ int gcd(int a, int b) {     a = abs(a);     b = abs(b);     if (a == 0) {         return b;     }     if (b == 0) {         return a;     }     while (a != b) {         if (a &gt; b) {             a = a - b;         } else {             b = b - a;         }     }     return a; }</pre>	<pre>/**  * Test function for gcd  */ void test_gcd() {     assert(gcd(2, 4) == 2);     assert(gcd(3, 27) == 3);     assert(gcd(7, 27) == 1);     assert(gcd(7, -27) == 1); }</pre>
---	---

# Acoperirea testelor – Test Code Coverage

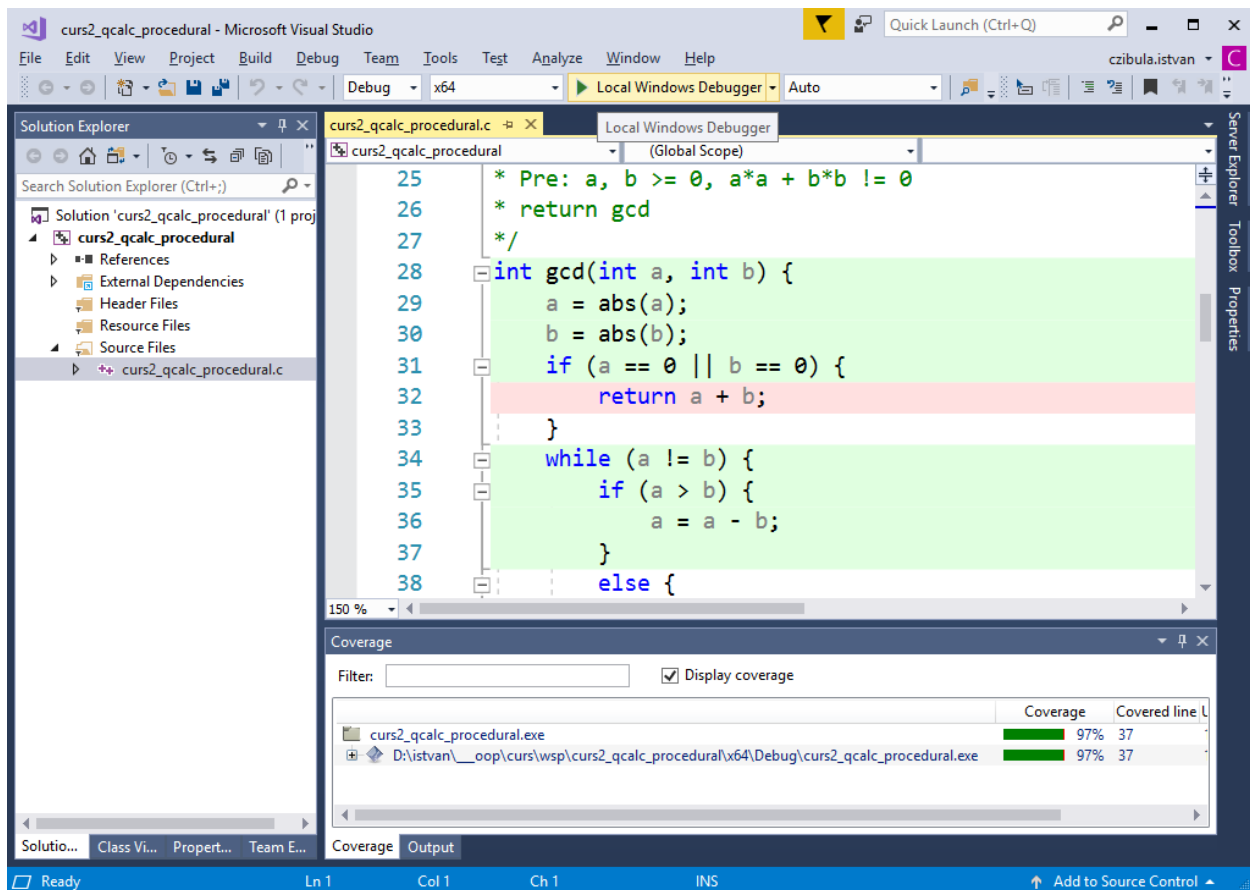
Ideea: măsoară procentul de cod executat (din totalul de cod din proiect/fișier) în urma rulării programului.

Code Coverage – porțiunea de cod executată la rularea aplicației

Test code coverage - porțiunea de cod executată la rularea testelor

## Test Code Coverage

- măsura pentru calitatea testelor (nu e singura)
- varianta simplă numără liniile de cod efectiv executate la rularea tuturor testelor
- există și alte variante: branch coverage, statement coverage, expression coverage, etc.



Pentru Visual Studio: se instalează plugin-ul OpenCPPCoverage

Din meniul: Tools->Extension and Updates -> OpenCPPCoverage plugin install

După instalare apare un nou element de meniu: Tools-> Run OpenCPPCoverage

VS 2022: încă nu apare pluginul oficial, dar găsiți o variantă pe teams-ul cursului.

## Review: Calculator – varianta procedurală

Problem statement: Profesorul are nevoie de un program care permite elevilor să învețe despre numere raționale. Programul ajută studenții să efectueze operații aritmetice cu numere raționale

```
/**
 * Test function for gcd
 */
void test_gcd() {+
    assert(gcd(2, 4) == 2);
    assert(gcd(3, 27) == 3);
    assert(gcd(7, 27) == 1);
    assert(gcd(7, -27) == 1);
}

/**
 * Add (m, n) to (toM, toN) - operation on rational numbers
 * Pre: toN != 0 and n != 0
 */
void add(int* toM, int* toN, int m, int n) {
    *toM = *toM * n + *toN * m;
    *toN = *toN * n;
    int gcdTo = gcd(abs(*toM), abs(*toN));
    *toM = *toM / gcdTo;
    *toN = *toN / gcdTo;
}

int main() {
    test_gcd();
    int totalM = 0, totalN = 1;
    int m, n;
    while (1) {
        printf("Enter m, then n to add\n");
        scanf("%d", &m);
        scanf("%d", &n);
        add(&totalM, &totalN, m, n);
        printf("Total: %d/%d\n", totalM, totalN);
    }
    return 0;
}
```

## **Principii de proiectare pentru funcții**

- **Fiecare funcție sa aibă o singură responsabilitate (Single responsibility principle)**
- **Folosiți nume sugestive (nume funcție, nume parametrii, variabile)**
- **Folosiți reguli de denumire (adauga\_rational, adaugaRational, CONSTANTA), consistent în toată aplicația**
- **Specificați fiecare funcție din aplicație**
- **Creați teste automate pentru funcții**
- **Funcția trebuie sa fie ușor de testat, (re)folosit, înțeles și modificat**
- **Folosiți comentarii în cod (includeți explicații pentru lucruri care nu sunt evidente în cod)**
- **Evitați (pe cât posibil) funcțiile cu efect secundar**

## **Programare Modulara in C/C++.**

Modulul este o colecție de funcții si variabile care oferă o funcționalitate bine definită.

### **Fișiere Header .**

Declarațiile de funcții sunt grupate într-un fișier separat – fișier header (.h).

Implementarea (definițiile pentru funcții) într-un fișier separat (.c/.cpp)

### **Scop**

Separarea interfeței (ce oferă modulul) de implementare (cum sunt implementate funcțiile)

Separare specificații, declarații de implementare

Modulele sunt distribuite in general prin: fișierul header + fișierul binar cu implementările (.dll,.so)

- Nu e nevoie să dezvălui codul sursă (.c/.cpp )

Cei care folosesc modulul au nevoie doar de declarațiile de funcții (fișierul header) nu si de implementări (codul din fișierele .c/.cpp)

## Directive de preprocessare

Preprocesarea are loc înainte de compilare.

cod sursă – preprocesare – compilare – linkeditare – executabil

Permite printre altele: includere de fișiere header, definire de macrouri, compilare condiționată

## Directiva Include

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

Pentru a avea acces la funcțiile declarate într-un modul (bibliotecă de funcții) se folosește directiva **#include**

Preprocesorul include fișierul referit în fișierul sursă în locul unde apare directiva

avem două variante pentru a referi un modul: < > sau “”

```
#include "local.h"      //caută fișierul header relativ la directorul curent al aplicației
```

```
#include <header>      // caută fișierul header între bibliotecile system (standard compiler include paths )
```



## Aplicații modulare C/C++

Codul este împărțit în mai multe fișiere header (.h) și implementare (.c)

- fișierele **.h** conțin declarații (interfața)
- **.c** conține definiția (implementarea) funcțiilor

se grupează funcții în module astfel încât modulul să ofere o funcționalitate bine definită (puternic coeziv)

- Când un fișier .h se modifică este nevoie de **recompilarea** tuturor modulelor care îl referă (direct sau indirect)
- Fișierele .c se pot compila separat, modificarea implementării nu afectează modulele care folosesc (ele referă doar definițiile din header)

Headerul este un **contract** între cel care dezvoltă modulul și cel care folosește modulul.

Detaliile de implementare sunt ascunse în fișierul .c

## **Review: Calculator versiune modulară**

### **Module:**

- **calculatorui.c – interfața utilizator**
- **calculator.h, calculator.c - TAD Calculator, operatii cu calculator**
- **rational.h, rational.c - TAD rational, operatii cu numere rationale**
- **util.h, util.c - funcții utile de operații cu numere (gcd)**

## Declarație multiplă – directivele `#ifndef` și `#define`

Într-un program mai complex este posibil ca un fișier header să fie inclus de mai multe ori. Asta ar conduce la declarații multiple pentru funcții

Soluție: se folosesc directivele de preprocesare

**`#ifndef`, `#ifdef`, `#define`, `#endif`**

Se poate verifica dacă modulul a fost deja inclus, respectiv să marcăm când un modul a fost inclus (prin definirea unei etichete)

```
#ifndef RATIONAL_H_    /* verify if RATIONAL_H_ is already defined, the rest
                        (until the #endif will be processed only if RATIONAL_H_ is
                        not defined*/
#define RATIONAL_H_    /* define RATIONAL_H_ so next time the preprocessor will not
                        include this */

/**
 * New data type to store rational numbers
 */
typedef struct {
    int a, b;
} Rational;

/**
 * Compute the sum of 2 rational numbers
 * a,b rational numbers
 * rez - a rational number, on exit will contain the sum of a and b
 */
void sum(Rational nr1, Rational nr2, Rational &rez);

#endif /* RATIONAL_H_ */
```

## Principii de proiectare pentru module

- **Separați interfața de implementare**
  - **Headerul conține doar declarații, implementările în fișierul .c**
- **Includeți la începutul fișierului header un comentariu, o scurta descriere a modulului**
- **Creați module puternic cohezive**
  - **fiecare modul o singură funcționalitate, are o singură responsabilitate**
- **Șablonul - Arhitectură stratificată**
  - **Straturi: ui, service, model, validation, repository**
  - **Controlul dependențelor - Fiecare nivel depinde doar de nivelul următor**
- **Tip abstract de date – TAD**
  - **operațiile definite in header (interfață) /implementarea in .c**
  - **ascundere detalii de implementare**
  - **specificații abstracte (independent de implementare, ce face nu cum)**

## Biblioteci standard

`#include <stdio.h>`

Operații de intrare/ieșire

`#include <math.h>`

Funcții matematice – abs, sqrt, sin, exp, etc

`#include <string.h>`

sirul de caractere in C - vector de char care se termina cu caracterul `'\0'`

`strncpy` - copiază string

`strcat` - concatenează string

`strcmp` - compară stringuri

`strlen` - lungimea stringului

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

int main(void) {
    char arr[4]; // for accommodating 3 characters and one null '\0' byte.
    char *ptr = "abc"; //a string containing 'a', 'b', 'c', '\0'

    memset(arr, '\0', sizeof(arr)); //reset all
    strncpy(arr, ptr, sizeof("abc")); // Copy the string

    printf("\n %s \n", arr);

    arr[0] = 'p';

    printf("\n %s \n", arr);
    return 0;
}
```

## Pointeri

Pointer este un tip de date folosit pentru a lucra cu adrese de memorie - poate stoca adresa unei variabile, adresa unei locații de memorie

Operatori: '&', '\*'

```
#include <stdio.h>

int main() {
    int a = 7;
    int *pa;

    printf("Value of a:%d address of a:%p \n", a, &a);
    //assign the address of a to pa
    pa = &a;
    printf("Value of pa:%d address of pa:%p \n", *pa, pa);

    //a and pa refers to the same memory location
    a = 10;
    printf("Value of pa:%d address of pa:%p \n", *pa, pa);
    return 0;
}
```

## Null pointer

- valoare specială (0) pentru a indica faptul ca pointerul nu referă o memorie validă

## Pointer invalid (Dangling pointer)

Adresa referită de pointer e invalid

```
#include <stdio.h>

int main() {
    //init to null
    int *pa1 = NULL;
    int *pa2;
    //!!! pa2 refers to an unknown address
    *pa2 = 6;

    if (pa1==NULL){
        printf("pa1 is NULL");
    }
    return 0;
}
```

```
#include <stdio.h>
int* f() {
    int localVar = 7;
    printf("%d\n", localVar);
    return &localVar;
}

int main() {
    int* badP = f();
    //!!! *badP refera o adresa de memorie
    //care a fost deja eliberata
    printf("%d\n", *badP);
}
```

## Vectori / pointeri - Aritmetica pointerilor

O variabila de tip vector - un pointer la primul element al vectorului

- vectorul este transmis prin referința (se transmite adresa de memorie al primului element din vector – nu se face o copie).
- Indicele pornește de la 0 – primul element este la distanță 0 față de începutul vectorului.
- Expresia `array[3]` – compilatorul calculează care este locația de memorie la distanță 3 față de începutul vectorului.
- Cu funcția **sizeof**(var) se poate afla numărul de bytes ocupat de valoarea din var (depinde de tipul lui var)

Array decay – orice vector cu elemente de tip T poate fi folosit în locul în care se cere un pointer la T. Obs. Informațiile despre dimensiune se pierd în acest proces (nu putem afla dimensiunea folosind sizeof)

## Aritmetica pointerilor

Folosirea de operații adăugare/scădere pentru a naviga în memorie (adrese de memorie)

```
#include <stdio.h>

int main() {
    int t[3] = { 10, 20, 30 };
    int *p = t;
    //print the first elem
    printf("val=%d adr=%p\n", *p, p);

    //move to the next memory location (next int)
    p++;
    //print the element (20)
    printf("val=%d adr=%p\n", *p, p);
    return 0;
}
```

**p++** în funcție de tipul valorii referite de pointer, compilatorul calculează următoarea adresa de memorie.



## Gestiunea memoriei

Pentru variabilele declarate într-o aplicație, compilatorul alocă memorie pe **stivă** (o zonă de memorie gestionat de compilator)

```
int f(int a) {  
    if (a>0){  
        int x = 10; //memory for x is allocated on the stack  
    }  
    //here x is out of scope and the memory allocated for x is no longer reserved  
    //the memory can be reused  
    return 0;  
}
```

```
int f(int a) {  
    int *p;  
    if (a>0){  
        int x = 10;  
        p = &x;  
    }  
    //here p will point to a memory location that is no longer reserved  
    *p = 5; //!!! undefined behavior, the program may crash  
    return 0;  
}
```

Memoria este automat eliberată de compilator în momentul în care execuția părăsește domeniul de vizibilitate a variabilei.

La ieșire dintr-o funcție memoria alocată pentru variabile locale este eliberată automat

## Alocare dinamică

Folosind funcțiile **malloc**(size) și **free**(pointer) programatorul poate alocă memorie pe Heap – zonă de memorie gestionat de programator

**<stdlib.h>** header file.

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

int main() {
    //allocate memory on the heap for an int
    int *p = malloc(sizeof(int));

    *p = 7;
    printf("%d \n", *p);
    //Deallocate
    free(p);
    //allocate space for 10 ints (array)
    int *t = malloc(10 * sizeof(int));
    t[0] = 0;
    t[1] = 1;
    printf("%d \n", t[1]);
    //deallocate
    free(t);
    return 0;
}

/**
 * Make a copy of str
 * str - string to copy
 * return a new string
 */
char* stringCopy(char* str) {
    char* newStr;
    int len;
    len = strlen(str) + 1; // +1 for the '\0'
    newStr = malloc(sizeof(char) * len); // allocate memory
    strcpy(newStr, str); // copy string
    return newStr;
}
```

Programatorul este responsabil sa dea loca memoria

OBS: Pentru fiecare **malloc** trebuie sa avem exact un **free**

## Memory management

`void *malloc(int num);` - alocă num byte de memorie, memoria este neinițializată

`void *calloc(int num, int size);` - alocă num\*size memorie, inițializează cu 0

`void *realloc(void *address, int newsize);` - resize the memory

`void free(void *address);` - eliberează memoria (este disponibilă pentru următoarele alocări)

## Memory leak

### Programul alocă memorie dar nu dealocă niciodată, memorie irosită

```
int main() {
    int *p;
    int i;
    for (i = 0; i < 10; i++) {
        p = malloc(sizeof(int));
        //allocate memory for an int on the heap
        *p = i * 2;
        printf("%d \n", *p);
    }
    free(p); //deallocate memory
    //leaked memory - we only deallocated the last int
    return 0;
}
```

**void\***

O funcție care nu returnează nimic

```
void f() {  
}
```

Nu putem avea variabile de tip **void** dar putem folosi pointer la void - **void\***

```
#include <stdio.h>  
#include <stdlib.h>  
  
int main() {  
    void* p;  
    int *i=malloc(sizeof(int));  
    *i = 1;  
    p = i;  
    printf("%d /n", *((int*)p));  
    long j = 100;  
    p = &j;  
    printf("%ld /n", *((long*)p));  
    free(i);  
    return 0;  
}
```

Se pot folosi **void\*** pentru a crea structuri de date care funcționează cu orice tip de elemente

Probleme: verificare egalitate între elemente de tip **void\*** , copiere elemente

## Vector dinamic

```
typedef void* Element;

typedef struct {
    Element* elems;
    int lg;
    int capacitate;
} VectorDinamic;

/**
 * Creaza un vector dinamic
 * v vector
 * post: vectorul e gol
 */
VectorDinamic * creazaVectorDinamic();

/**
 * Initializeaza vectorul
 * v vector
 * post: vectorul e gol
 */
VectorDinamic * creazaVectorDinamic() {
    VectorDinamic *v =
        malloc(sizeof(VectorDinamic));
    v->elems = malloc(INIT_CAPACITY *
        sizeof(Element));
    v->capacitate = INIT_CAPACITY;
    v->lg = 0;
    return v;
}

/**
 * Elibereaza memoria ocupata de vector
 */
void distruge(VectorDinamic *v) {
    int i;
    for (i = 0; i < v->lg; i++) {
        //!!!!functioneaza corect doar daca
        //elementele din lista NU refera
        //memorie alocata dinamic
        free(v->elems[i]);
    }
    free(v->elems);
    free(v);
}

/**
 * Adauga un element in vector
 * v - vector dinamic
 * el - elementul de adaugat
 */
void add(VectorDinamic *v, Element el);

/**
 * Returneaza elementul de pe pozitia data
 * v - vector
 * poz - pozitie, poz>=0
 * returneaza elementul de pe pozitia poz
 */
Element get(VectorDinamic *v, int poz);

/**
 * Aloca memorie aditionala pentru vector
 */
void resize(VectorDinamic *v) {
    int nCap = 2*v->capacitate;
    Element* nElems=
        malloc(nCap*sizeof(Element));
    //copiez din vectorul existent
    int i;
    for (i = 0; i < v->lg; i++) {
        nElems[i] = v->elems[i];
    }
    //dealocam memoria ocupata de vector
    free(v->elems);
    v->elems = nElems;
    v->capacitate = nCap;
}

/**
 * Adauga un element in vector
 * v - vector dinamic
 * el - elementul de adaugat
 */
void add(VectorDinamic *v, Element el) {
    if (v->lg == v->capacitate) {
        resize(v);
    }
    v->elems[v->lg] = el;
    v->lg++;
}
```

## Pointer la funcții

```
void (*funcPtr)(); // a pointer to a function
void *funcPtr();   // a function that returns a pointer
```

```
void func() {
    printf("func() called...");
}
int main() {
    void (*fp)(); // Define a function pointer
    fp = func; // Initialise it
    (*fp)(); // Dereferencing calls the function
    void (*fp2)() = func; // Define and initialize
    (*fp2)(); // call
}
```

Putem folosi pointer la funcții în structurile de date generice

```
typedef elem (*copyPtr)(elem&, elem);

typedef int (*equalsPtr)(elem, elem);
```