



UNIVERSITATEA BABEŞ-BOLYAI  
Facultatea de Matematică și Informatică



# Programare orientată obiect

---

Curs 02

Laura Dioşan

# POO

---

- Elemente de bază ale limbajului C++
  - Referințe și pointeri
  - Vectori
- TAD-uri

# Tipul referință

- Dacă T este un tip de date, atunci T& este o referință la acel tip
- Utilizare:
  - Variabilă referință (alias pentru o variabilă)
    - o variabilă referință trebuie inițializată la declarare
  - apelul prin referință a param. unei fc.
  - operatorul referință
- avantaje:
  - eficiență în utilizarea memoriei
    - apelul funcțiilor
  - claritate codului

```
void same(int x){
    x++;
}
void change(int& x){
    x++;
}
void main(){
    int v = 5;
    int &a = v; //alias a = 5

    same(v);    //v = 5
    change(v);  //v = 6

    int* r = &v; //r = 0012FF60
}
```

```
void change(int& x){
    x++;
    cout << x;    // 6
    cout << &x;  // 0012FF60
}
void main(){
    int v = 5;
    cout << v;    // 5
    cout << &v;  // 0012FF60
    change(v);   // v = 6
}
```

# Tipul pointer

---

- Dacă T este un tip de date → T\* este un pointer spre tipul T

```
void main(){
    int i;
    int *pi;
    float *pf;
    char *pc;
    int *pa, pb;
}
```

- Utilizare → pentru a reține adresa unei variabile sau NULL (0) din:
  - segmentul de date – variabile globale și statice
  - stivă – variabile locale/automatice
  - segmentul de cod – funcții
  - HEAP (memoria din timpul rulării) – variabile cu memorie alocată dinamic

# Tipul pointer

## □ Inițializare

### ■ cu adresa unei variabile

- Dereferențiere – accesarea conținutului de la o adresă dată

- pointer generic (***void\****) – pentru a reține adresa oricărui tip de date

- conversii explicite

```
void main(){
    int x = 5;
    int *px = &x;
    cout << x;        //5
    cout << px;      //0012FF60
    cout << *px;     //5
}
```

```
void main(){
    int x = 5;
    char c = 'a';
    int *pi = &x;
    cout << pi << *pi;
        //0012FF60 5
    char* pc = &c;
    cout << hex << (int)pc << *pc;
        //0012FF60 a
    void *pg = &x;
    cout << hex << (int)pg << *(int*)pg;
        //0012FF60 5
    pg = &c;
    cout << hex << (int)pg << *(char*)pg;
        //0012FF60 a

    int y = 7;
    int *py = &y;
    void *pg2;
    pg2 = py;
    //py = pg2; error
    py = (int*) pg2;
```

# Tipul pointer

## □ Inițializare

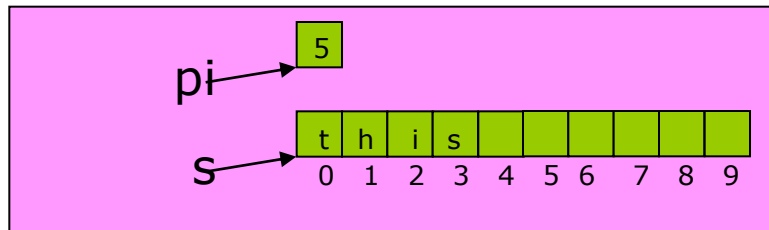
### ■ prin alocare dinamică a memoriei

- **new** – pentru alocare în Heap
- **delete** – pentru dealocare

```
T* pt = new T;  
delete pt;  
pt = NULL;
```

### □ e.g.

- alocare de 1 spațiu →
- alocare de mai multe spații →



```
void main(){  
    int *pi = new int;  
    *pi = 5;  
    cout << pi << *pi; //003434B8 5  
    delete pi;  
    pi = NULL;  
  
    char* s = new char[10];  
    s[0] = 't'; s[1] = 'h';  
    s[2] = 'i'; s[3] = 's';  
    delete[] s;  
    s = NULL;  
}
```

# Tipul pointer

---

## □ operatorul **new** se poate utiliza:

- `new type`
- `new type( initial_value_for_dynamic_Variable )`
- `new type[ numer_of_dynamic_variables_alocated ]`

## □ dealocarea

- `delete reference_variable;`
- `delete[] reference_variable;`

## □ exemplu:

- `int *p1=new int; //value from address p1 is not initialised`
- `int *p2=new int(5); //value from address p2 is 5;`
- `int *p3=new int[4]; //a vector of 4 integer elements is alocated`
  
- `delete []p3;`
- `delete p2;`
- `delete p1;`

## □ observații:

- pentru fiecare **new**, trebuie realizat un **delete**
- dealocarea trebuie realizată în ordinea inversă alocării

# Tipul pointer

## □ Inițializare

### ■ prin alocare dinamică a memoriei

□ **malloc** – alocarea în Heap

□ **free** – dealocarea

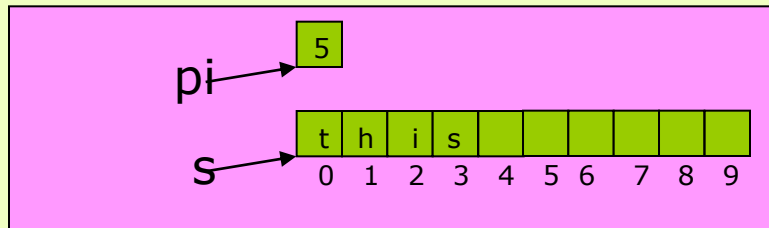
□ **realloc, calloc**

□ `<stdlib.h>`

□ e.g.

■ alocare de 1 spațiu

■ alocare de mai multe spații



```
T* pt = (T*)malloc(size of T);  
free(pt);
```

```
void main(){  
    int* pi = (int *)malloc(sizeof(int));  
    *pi = 5;  
    cout << pi << *pi;    //003434B8 5  
    free(pi);  
    pi = NULL;  
  
    char* s = (char *)malloc(10 * sizeof(char));  
    strcpy(s, "this");  
    cout << s << *s;    //this t  
    free(s);  
    s = NULL;  
}
```



# Tipul pointer

## □ Operații

- relaționale:  $<$ ,  $<=$ ,  $>$ ,  $>=$ ,  $==$ ,  $!=$ 
  - pentru adrese de memorie
- aritmetice:  $+$ ,  $-$ ,  $++$ ,  $--$ ,  $+=$ ,  $-=$ 
  - pentru vectori

```
T* pt = new T[5];  
T* q = new T[9];  
int n;
```

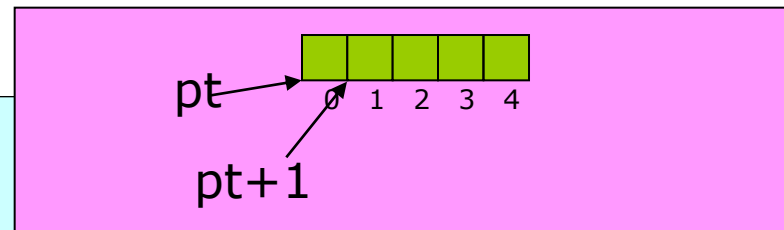
```
pt + n; ⇔ pt + n * sizeof(T);
```

```
pt++; ⇔ pt + sizeof(T);
```

```
pt - n; ⇔ pt - n * sizeof(T);
```

```
pt--; ⇔ pt - sizeof(T);
```

```
pt - q = mem. locations between pt and q
```



# Tipul pointer

---

## □ Variabile pointer ca și parametri

- când adresa reținută de pointer se modifică -> param. apelat prin referință

```
void allocation(int nrLin, int nrCol, int** &mat){
    mat = new int*[nrLin];
    for(int i = 0; i < nrLin; i++)
        mat[i] = new int[nrCol];
}
```

- Când informația de la adresa referită de variabila pointer se modifică -> param. apelat prin valoare

```
void init(int nrLin, int nrCol, int** mat, int val){
    for(int i = 0; i < nrLin; i++)
        for(int j = 0; j < nrCol; j++)
            mat[i][j] = val;
}
```

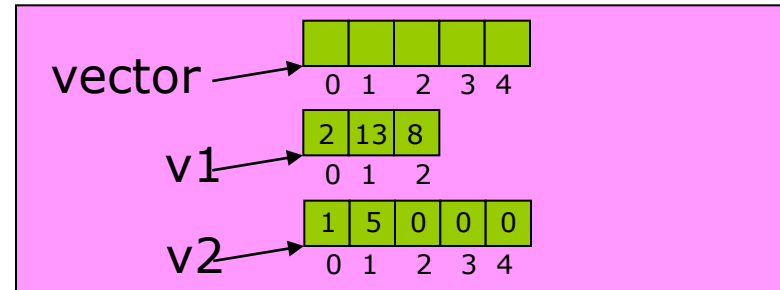
# Tipul pointer

## □ vectori

### ■ ca variabile

```
int vector[5];
int matrix[3][5];
int v1[] = {2, 13, 8};
int v2[5] = {1, 5};
```

```
void main(){
    int v1[] = {2, 13, 8};
    int n = sizeof(v1) / sizeof(int);
    int *vlp = v1;
    for(int i = 0; i < n; i++)
        cout << vlp[i] << " ";
    for(int i = 0; i < n; i++)
        cout << *(vlp + i) << " ";
    for( ;vlp - v1 < n; vlp++)
        cout << *vlp << " ";
}
```



```
void main(){
    int *vlp = new int[3];

    vlp[0] = 2;
    *(vlp + 1) = 13;
    *(vlp + 2) = 8;
    for(int i = 0; i < 3; i++)
        cout << vlp[i] << " ";

    if (vlp != NULL){
        delete[] vlp;
        vlp = NULL;
    }
}
```

### ■ ca param.

- a se consulta *vectorSimple.cpp* și *vectorPointer.cpp*

# Tipul pointer

## □ Vectori de caractere

- similar vectorilor de numere
- funcții speciale
  - <cstring.h> or <string> headers

```
void main(){
    char* flower1 = "tulip";
    char* flower2 = new char[strlen(flower1) + 1];
    strcpy(flower2, flower1);
    cout << flower2 << endl;
}
```

- strlen → lungimea unui string

```
size_t strlen(const char* s);
```

- strcmp → compararea a 2 stringuri

```
int strcmp (const char* s1, const char* s2);
```

- strncmp → compararea a 2 stringuri

```
int strncmp(const char* s1, const char* s2, size_t num);
```

- strcpy → copierea unui string in alt string

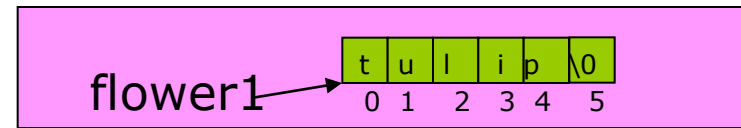
```
char* strcpy (char* destination, const char* source);
```

- strncpy → copierea unui string in alt string

```
char* strncpy(char* dest, const char* source, size_t n);
```

- strcat → concatenarea a 2 stringuri

```
char* strcat(char* destination, const char* source );
```



# Tipul pointer

## □ matrici

### ■ ca variabile

```
void main(){
    int linNo = 3;
    int colNo = 2;
    int m1[3][2];
    int** m2 = new int*[linNo];
    for(int i = 0 ;i < linNo; i++)
        m2[i] = new int[colNo];

    for(int i = 0; i < linNo; i++)
        for(int j = 0; j < colNo; j++)
            m2[i][j] = (i + 1) * (j + 1);

    for(int i = 0; i < linNo; i++){
        for(int j = 0; j < colNo; j++)
            cout << m2[i][j] << " ";
        cout << endl;
    }

    for(int i = 0; i < linNo; i++){
        delete[] m2[i];
        m2[i] = NULL;
    }
    delete[] m2;
    m2 = NULL;
}
```

### ■ ca param.

- a se consulta *matrixSimple.cpp*, *matrixPointer.cpp* și *matrixPointer2.cpp*

# Tipul pointer

## □ structuri

```
struct Flower{
    char* name;
    int price;
};
typedef Flower* FlowerPtr;

void main(){
    Flower f;
    f.name = new char[10];
    strcpy(f.name, "daisy");
    f.price = 10;
    cout << f.name << endl;
    if (f.name != NULL){
        delete[] f.name;
        f.name = NULL;
    }

    //Flower *pf = new Flower;
    FlowerPtr pf = new Flower;
    pf->name = new char[10];
    strcpy(pf->name, "rose");
    pf->price = 5;
    cout << pf->name << endl;
    if (pf != NULL){
        if (pf->name != NULL){
            delete[] pf->name;
            pf->name = NULL;
        }
        delete pf;
        pf = NULL;
    }
}
```

# Tipul pointer

Pointeri constanți – `const` ⇔ *cel mai apropiat de*

- ❑ `const int* u;`
- ❑ `int const* v;` → **u/v** este un pointer care refră un **întreg constant** (conținutul de la adresa **u/v** nu se poate modifica)
- ❑ `int d = 1;`
- ❑ `int* const w = &d;` → **w** este un pointer **constant** care referă un **întreg** (adresa **w** nu poate fi modificată)
- ❑ `int d = 1;`
- ❑ `const int* const x = &d;`
- ❑ `int const* const x2 = &d;` → **x(x2)** este un pointer constant care referă un **întreg constant** (adresa **x/x2** și conținutul ei nu pot fi modificate)

```
void main(){
    int i = 7;
    const int* u = &i; //int is a const
    //*u = 10;      // you cannot assign to a variable that is const

    int const* v = &i; //int is a const
    //*v = 9;      //you cannot assign to a variable that is const

    int j = 5;
    int* const w = &j; //pointer w is a const
    int k = 9;
    //w = &k;      //you cannot assign to a variable that is const

    int l = 1;
    const int* const x = &l; //const pointer to a const int
    int const* const x2 = &l;
    int m = 3;
    //x = &m;      //you cannot assign to a variable that is const
    //*x = m;      //you cannot assign to a variable that is const
}
```

# Tipul pointer

## □ Pointer către funcții

- o funcție ocupă o zonă de memorie (care are o adresă)
- numele fc. = începutul adresei
- variabile de tip pointer la fc.
  - `int (*funcPtr)(int, int)` = o fc. care returnează un întreg și are 2 param. Întreg
  - `int *funcPtr(int, int)` = o fc. care returnează un pointer către un întreg și are 2 param. Întregi

```
int max(int a, int b){
    return (a > b ? a : b);
}

int (*funcPtr)(int, int);

void main(){
    funcPtr = max;
    cout << funcPtr(3,4);
    cout << (*funcPtr)(3, 4);
}
```

- tipul de date pointer la fc
  - a se consulta `sortVector.cpp`

```
typedef int (*orderRel)(int, int);

int max(int a, int b){
    return (a > b ? a : b);
}

void main(){
    orderRel rel = max;
    cout << rel(3, 4);
}
```



# Cursul următor

---

- Elemente de bază ale limbajului C++
  - Fișiere de IO
  
- Clase
  - Declarație
  - Constructor/destructor
  - Metode
  - Utilizare (static și dinamic)
  - Clase ca și date membre
  - UML