

# Curs 02

## RECAPITULARE (CONT.)

Pointeri, masive, transmiterea parametrilor



# Pointerii



- Variabile responsabile cu salvarea unor adrese de memorie
- De cele mai multe ori sunt salvați în stack și pointează către zone din heap (acesta este cel mai întâlnit scenariu, dar nu înseamnă că nu pot pointa către zone din stack sau că nu pot fi definiți în heap)
- Ocupă 4 bytes indiferent de tipul de pointer (ce se regăsește la adresa de memorie către care pointează) pentru un procesor/compiler pe 32 de biți

# Definire



```
//la definire de obicei valoarea este una arbitrara  
//nu este recomandat ca un pointer sa ramana neinitializat  
tip_data* nume_pointer;  
  
tip_data* nume_pointer = nullptr;
```

# Inițializare



```
//pointerul poate salva doar adrese unde se gaseste  
//tipul de data specificat  
//operatorul & extrage adresa de memorie a unei variabile  
tip_data nume_variabila = valoare;  
tip_data* nume_pointer = &nume_variabila;
```

# Utilizare



```
//va afisa o adresa de memorie  
cout << nume_pointer;
```

```
//va afisa continutul de la acea adresa de memorie  
cout << *nume_pointer;
```

# Operații cu pointeri



- Definire
- Inițializare
- Dereferențiere
- Incrementare/decrementare
- Diferența a doi pointeri



# Incrementare/decrementare

```
int* nume_pointer = nullptr;  
nume_pointer++;  
//ce va afisa?  
cout << nume_pointer;  
//dar acum?  
nume_pointer--;  
cout << nume_pointer;
```

# Incrementare/decrementare



- Incrementarea unui pointer de tip  $T^*$  va duce la mărirea adresei cu `sizeof(T)` sau, cu alte cuvinte, deplasarea în memorie înainte către adresa următoarei variabile de tip  $T$
- Decrementarea unui pointer de tip  $T^*$  va duce la scăderea adresei cu `sizeof(T)` sau, cu alte cuvinte, deplasarea în memorie înapoi către adresa precedentei variabile de tip  $T$

# Caz particular: adunarea unui întreg



- Cum expresia *variabilă*++ este echivalentă cu *variabilă* = *variabilă* + 1, deducem că putem aduna sau scădea orice întreg la/dintr-un pointer
- Deci expresia:

```
nume_pointer = nume_pointer + 3;
```

- Va duce la incrementarea adresei salvate în *nume\_pointer* cu  $3 * \text{sizeof}(T)$ , unde *T* este tipul de dată utilizat la definire



# Scăderea a doi pointeri

- Așa cum am văzut expresia de mai jos este una validă

```
nume_pointer2 = nume_pointer + 3;
```

- Asta înseamnă că și expresia de mai jos este una validă

```
nume_pointer2 - nume_pointer = 3;
```

- Deci scăderea a doi pointeri este posibilă, iar rezultatul este diferența efectivă dintre adrese împărțită la `sizeof(T)` sau, cu alte cuvinte, câte variabile de tip `T` încap între cei doi pointeri

# Atenție!



Adunarea a doi pointeri nu este posibilă pentru că nu ar avea sens  
(suma a două adrese nu este neapărat o adresă validă)!

# Identificatorul const în cazul pointerilor



- Pointerii făcând referire la adresa unei variabile există 3 situații posibile de folosire a identificatorului const:
  - pointeri constanți: odată salvată o adresă, nu mai poate fi modificată
  - pointeri la o zonă de memorie constantă: valoarea de la adresa către care pointează nu poate fi modificată prin intermediul lor
  - pointeri constanți la o zonă de memorie constantă: combinație între cele două situații de mai sus

# Identificatorul const în cazul pointerilor



```
int x = 5; int y = 10;
```

```
//Pointer constant la intreg
```

```
int* const p = &x;
```

```
//adresa catre care pointeaza nu poate fi schimbata
```

```
//p = &y;
```

```
//Pointer la o zona de memorie constanta (pointer la int constant)
```

```
const int* q = &x;
```

```
//valoarea lui x nu poate fi modificat prin intermediul lui q
```

```
/*q = 4;
```

```
//Pointer constant la o zona de memorie constanta
```

```
const int* const w = &x;
```

```
//nu se poate schimba nici adresa, nici valoarea
```

# Vectorii



- Tipuri de date ce folosesc o zonă de memorie contiguă pentru a salva mai multe valori de același tip
- Pot fi alocați static sau dinamic
- Cei alocați static sunt salvați în stack și trebuie să aibă un număr de elemente cunoscut în momentul compilării
- Cei alocați dinamic se alocă și se dezalocă în heap și pot avea un număr de elemente cunoscut la momentul execuției

# Definire



```
int array1[5];
```

```
float array2[4];
```

```
int* array3 = (int*)malloc(5 * sizeof(int));
```

```
float* array4 = new float[4];
```

```
free(array3); delete[] array4;
```

```
array4 = nullptr;
```