

# Pointeri

# Pointeri

Memoria calculatorului este organizată ca un șir lung de locații de memorie.

Fiecare locație de memorie este caracterizată prin 2 entități: adresă și valoarea stocată.

Valori:	5	42	-31	221	15	4	32	84	167	201
Adrese:	100	101	102	103	104	105	106	107	108	109

# Pointeri

 Pointer = variabilă care stochează adresa unei alte variabile.

Sintaxă:

```
<tip_date> * <nume_pointer>; // declarare pointer
```

Exemplu:

```
float *a; // a este un pointer către un float  
const int *b; // b este un pointer către un int constant
```

# Operatorul & (referențiere)

Operatorul & - returnează adresa operandului său

```
int num = 0;
```

```
int *p = nullptr; // initializare pointer cu valoare nula
```

Valori:	0	0	...
Adrese:	100	104	108
Identificatori:	num	p	

# Operatorul & (referențiere)

Operatorul & - returnează adresa operandului său

```
int num = 0;
```

```
int *p = nullptr;
```

```
p = &num; // atribuim pointerului p adresa lui num
```

Valori:	0	100	...
Adrese:	100	104	108
Identificatori:	num	p	

# Operatorul \* (dereferențiere)

Operatorul \* - returnează valoarea de la adresa stocată în operandul său

```
int num = 5;  
int *p = nullptr;
```

Valori:	5	0	...
Adrese:	100	104	108
Identificatori:	num	p	

# Operatorul \* (dereferențiere)

Operatorul \* - returnează valoarea de la adresa stocată în operandul său

```
int num = 5;  
int *p = nullptr;  
p = &num;
```

Valori:	5	100	...
Adrese:	100	104	108
Identificatori:	num	p	

# Operatorul \* (dereferențiere)

Operatorul \* - returnează valoarea de la adresa stocată în operandul său

```
int num = 5;  
int *p = nullptr;  
p = &num;  
*p = 200;
```

Valori:	200	100	...
Adrese:	100	104	108
Identificatori:	num	p	



# Operatorul `new` (alocare dinamică de memorie)

Operatorul `new` – alocă în mod dinamic (la momentul rulării) un bloc de memorie de o dimensiune dată

```
int *vec = new int[5];
```

Valori:	?	?	?	?	?
Adrese:	100	104	108	112	116
Identificatori:	vec				

# Operatorul `new` (alocare dinamică de memorie)

Operatorul `new` – alocă în mod dinamic (la momentul rulării) un bloc de memorie de o dimensiune dată

```
int *vec = new int[5];  
for (int i=0; i<5; i++){  
    *(vec+i) = i; // incrementarea adresei se face cu sizeof(int)*i  
}
```

Valori:	0	1	2	3	4
Adrese:	100	104	108	112	116
Identificatori:	vec	vec+1	vec+2	vec+3	vec+4

# Operatorul delete (eliberare dinamică de memorie)

Operatorul `delete` – șterge un bloc de memorie care a fost alocat în mod dinamic și pointerul care indica către el

```
int *val = new int;  
int *vec = new int[5];  
...  
delete [] vec;  
delete val;
```

# Operatorul delete (eliberare dinamică de memorie)

În lipsa apelării explicite a operatorului delete, pointerul va fi șters atunci când se termină domeniul de vizibilitate, însă zona de memorie către care acesta indica va fi lăsată intactă => o zonă de memorie rezervată sau chiar inițializată care nu va mai putea fi accesată = memory leak.

# Operatorul delete (eliberare dinamică de memorie)

În lipsa apelării explicite a operatorului delete, pointerul va fi șters atunci când se termină domeniul de vizibilitate, însă zona de memorie către care acesta indica va fi lăsată intactă => o zonă de memorie rezervată sau chiar inițializată care nu va mai putea fi accesată = memory leak.

# Operatorul delete (eliberare dinamică de memorie)

```
#include <iostream>

void f(int x){
    int *p = new int[x];
    for (int i=0; i<x; ++i){
        *(p+i) = i;
    }
}


int main(){
    f(5);
}
```

# Operatorul delete (eliberare dinamică de memorie)

```
#include <iostream>
```

```
void f(int x){  
    int *p = new int[x];  
    for (int i=0; i<x; ++i){  
        *(p+i) = i;  
    }  
}
```

```
int main(){  
    f(5);  
}
```



# Operatorul delete (eliberare dinamică de memorie)

```
#include <iostream>
```

```
void f(int x){  
    int *p = new int[x];  
    for (int i=0; i<x; ++i){  
        *(p+i) = i;  
    }  
}
```

```
int main(){  
    f(5);  
}
```

Valori:	?	?	?	?	?
Adrese:	100	104	108	112	116
Identificatori:	p				



# Operatorul delete (eliberare dinamică de memorie)

```
#include <iostream>
```

```
void f(int x){  
    int *p = new int[x];  
    for (int i=0; i<x; ++i){  
        → *(p+i) = i;  
    }  
}
```

```
int main(){  
    f(5);  
}
```

Valori:	0	1	2	3	4
Adrese:	100	104	108	112	116
Identificatori:	p				

# Operatorul delete (eliberare dinamică de memorie)

```
#include <iostream>
```

```
void f(int x){  
    int *p = new int[x];  
    for (int i=0; i<x; ++i){  
        *(p+i) = i;  
    }  
    }  
→ }
```

```
int main() {  
    f(5);  
}
```

zonă de memorie ocupată, însă inaccesibilă ⇔ memory leak

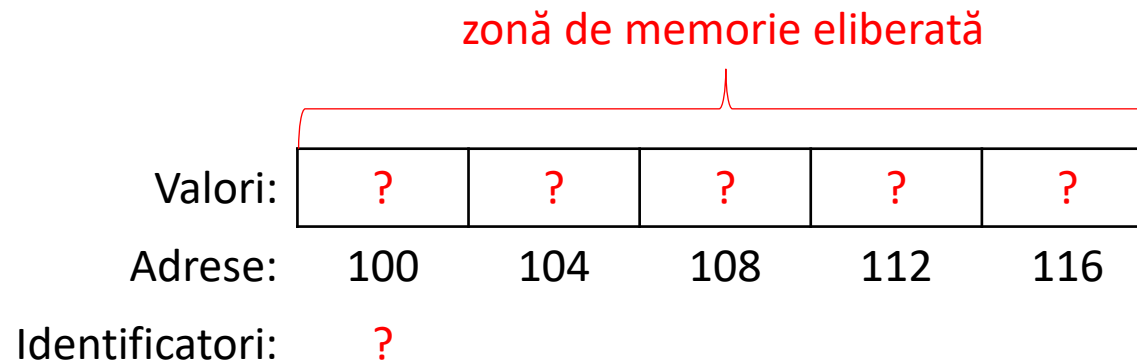
Valori:	0	1	2	3	4
Adrese:	100	104	108	112	116
Identificatori:	?				

# Operatorul delete (eliberare dinamică de memorie)

```
#include <iostream>
```

```
void f(int x){  
    int *p = new int[x];  
    for (int i=0; i<x; ++i){  
        *(p+i) = i;  
    }  
    delete [] p;  
}
```

```
int main(){  
    f(5);  
}
```



# Pointeri către obiecte

```
#include <iostream>

class Example{
private:
    int a;
public:
    Example(int a=0) {
        this->a = a;
    }
    void display() {
        std::cout << a << "\n";
    }
};
```

pointer implicit către obiectul a cărui funcție se execută.

```
int main() {
    Example e1(1);
    Example *e2 = &e1;
    Example *e3 = new Example(2);
    Example *e4 = new Example[5];

    e1.display();
    e2->display();
    e3->display();
    for (int i=0; i<5; ++i){
        (e4+i)->display();
        e4[i].display();
    }
    delete e2, e3;
    delete [] e4;

    return 0;
}
```

# Sfârșit capitol pointeri