

# P00

Curs-3

Gavrilut Dragos

- ▶ Functii “Friend”
- ▶ Operatori
  - ▶ Tipuri de operatori
  - ▶ Ordinea operatorilor
  - ▶ Operatori si clase (supraincercarea operatorilor)
- ▶ Operatii cu obiecte

- ▶ Functii “Friend”
- ▶ Operatori
  - ▶ Tipuri de operatori
  - ▶ Ordinea operatorilor
  - ▶ Operatori si clase (supraincercarea operatorilor)
- ▶ Operatii cu obiecte

# Functii “friend”

- ▶ O functie “friend” pentru o clasa este o functie care poate accesa membri si metodele private din acea clasa
- ▶ O functie “friend” nu este o functie care sa apartina clasei (in cazul de fata clasei Date). Din acest punct de vedere specificatorul de access nu se aplica (nu conteaza daca functia “friend” era scrisa in sectiunea private sau in cea public)

## App.cpp

```
class Date
{
    int x;
public:
    Date(int value) : x(value) {}
    void friend PrintDate(Date &d);
};

void PrintDate(Date &d)
{
    printf("X = %d\n", d.x);
}

void main()
{
    Date d1(1);
    PrintDate(d1);
}
```

# Functii “friend”

## App.cpp

```
class Date
{
    int x;
public:
    Date(int value) : x(value) {}
    friend class Printer;
};
class Printer
{
public:
    void PrintDecimal(Date &d);
    void PrintHexazecimal(Date &d);
};
void Printer::PrintDecimal(Date &d)
{
    printf("x = %d\n", d.x);
}
void Printer::PrintHexazecimal(Date &d)
{
    printf("x = %x\n", d.x);
}
void main()
{
    Date d1(123);
    Printer p;
    p.PrintDecimal(d1);
    p.PrintHexazecimal(d1);
}
```

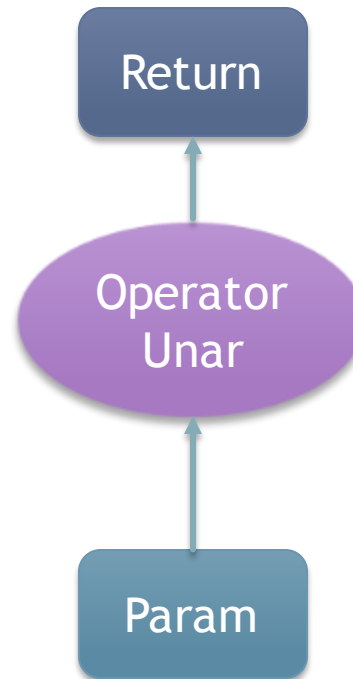
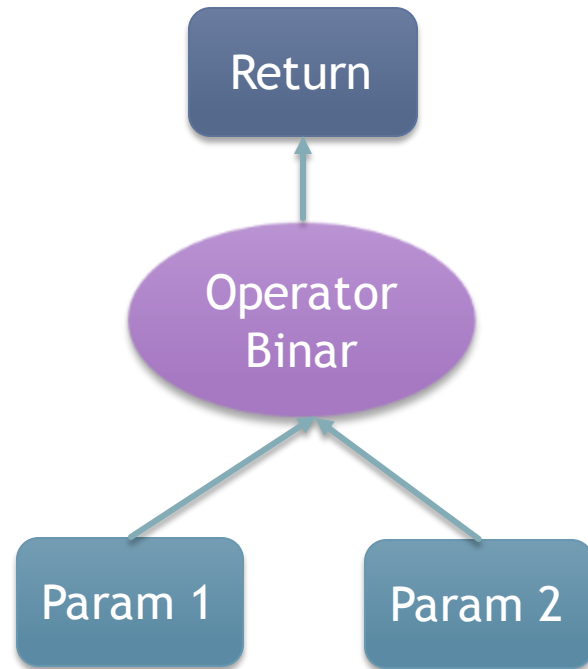
- ▶ O clasa “friend” se poate aplica pentru o clasa intreaga
- ▶ In cazul de fata se specifica ca tot membri din clasa **Printer** pot accesa datele private din clasa **Date**

- ▶ Functii “Friend”
- ▶ Operatori
  - ▶ Tipuri de operatori
  - ▶ Ordinea operatorilor
  - ▶ Operatori si clase (supraincercarea operatorilor)
- ▶ Operatii cu obiecte

# Operatori (tipuri)

- ▶ In functie de numarul de parametric necesari acelui operator pot fi:
  - ❖ Unari
  - ❖ Binari
  - ❖ Trinari
  - ❖ Multi parametru
- ▶ In functie de operatia pe care o realizeaza pot fi
  - ❖ Aritmetici
  - ❖ Relationari
  - ❖ Logici
  - ❖ Operatori pe biti
  - ❖ De asignare
  - ❖ Altii
- In functie de posibilitatea de suprascriere
  - ❖ Care pot fi suprascrisi
  - ❖ Care nu pot fi suprascrisi

# Operatori (tipuri)





# Operatori aritmetici

Operator	Tip	Suprascrisiere	Format	Returneaza
+	Binar	Da	$A + B$	Valoare/referinta
-	Binar	Da	$A - B$	Valoare/referinta
*	Binar	Da	$A * B$	Valoare/referinta
/	Binar	Da	$A / B$	Valoare/referinta
%	Binar	Da	$A \% B$	Valoare/referinta
++ (post/pre-fix)	Unar	Da	$A++$ sau $++A$	Valoare/referinta
-- (post/pre-fix)	Unar	Da	$A--$ sau $--A$	Valoare/referinta

# Operatori relationari

Operator	Tip	Suprascriptie	Format	Returneaza
==	Binar	Da	A == B	bool sau Valoare/referinta
>	Binar	Da	A > B	bool sau Valoare/referinta
<	Binar	Da	A < B	bool sau Valoare/referinta
<=	Binar	Da	A <= B	bool sau Valoare/referinta
>=	Binar	Da	A >= B	bool sau Valoare/referinta
!=	Binar	Da	A != B	bool sau Valoare/referinta

# Operatori logici

Operator	Tip	Suprascriptie	Format	Returneaza
&&	Binar	Da	A && B	bool sau Valoare/referinta
	Binar	Da	A    B	bool sau Valoare/referinta
!	Unar	Da	!	bool sau Valoare/referinta

# Operatori pe biti

Operator	Tip	Suprascrisiere	Format	Returneaza
&	Binar	Da	A & B	Valoare/referinta
	Binar	Da	A   B	Valoare/referinta
^	Binar	Da	A ^ B	Valoare/referinta
<<	Binar	Da	A << B	Valoare/referinta
>>	Binar	Da	A >> B	Valoare/referinta
~	Unar	Da	~A	Valoare/referinta

# Operatori de asignare

Operator	Tip	Suprascriere	Format	Returneaza
=	Binar	Da	A = B	Valoare/referinta
+=	Binar	Da	A += B	Valoare/referinta
-=	Binar	Da	A -= B	Valoare/referinta
*=	Binar	Da	A *= B	Valoare/referinta
/=	Binar	Da	A /= B	Valoare/referinta
%=	Binar	Da	A %= B	Valoare/referinta
>>=	Binar	Da	A >>= B	Valoare/referinta
<<=	Binar	Da	A <<= B	Valoare/referinta
&=	Binar	Da	A &= B	Valoare/referinta
^=	Binar	Da	A ^= B	Valoare/referinta
=	Binar	Da	A  = B	Valoare/referinta

# Operatori (altii)

Operator	Tip	Suprascriere	Format	Returneaza
sizeof	Unar	NU	sizeof(A)	valoare
new	Unar	Da	new A	pointer (A*)
delete	Unar	Da	delete A	<None>
Condition (?)	Ternar	NU	C ? A:B	A sau B in functie de C
:: (scope)		NU	A::B	
Cast (type)	Binar	Da	(A)B sau A(B)	B convertit la A
-> (pointer)	Binar	Da	A->B	B din A
. (membru)	Binar	Da	A.B	B din A
[] (index)	Binar	Da	A[B]	Valoare/referinta
() (function)	Multi	Da	A(B,C,...)	Valoare/referinta
, (lista)	Binar	Da	(A,B)	Val/ref pt. (A urmat de B)

- ▶ Functii “Friend”
- ▶ Operatori
  - ▶ Tipuri de operatori
  - ▶ **Ordinea operatorilor**
  - ▶ Operatori si clase (supraincercarea operatorilor)
- ▶ Operatii cu obiecte

# Operatori (ordinea de evaluare)

1. :: (scope)
2. () [] -> . ++ --
3. + - ! ~ ++ -- (type)\* & sizeof
4. \* / %
5. + -
6. << >>
7. < <= > >=
8. == !=
9. &
10. ^
11. |
12. &&
13. ||
14. ?:
15. = += -= \*= /= %= >>= <<= &= ^= |=
16. ,



- ▶ Functii “Friend”
- ▶ Operatori
  - ▶ Tipuri de operatori
  - ▶ Ordinea operatorilor
  - ▶ Operatori si clase (supraincercarea operatorilor)
- ▶ Operatii cu obiecte

# Operatori (clase)

- ▶ O clasa poate defini o serie de functii speciale care sa se comporte exact ca un operator - mai exact sa permita programatorului sa explice cum ar trebui compilatorul sa inteleaga anumite operatii intre clase
- ▶ Se foloseste cuvantul cheie “**operator**”
- ▶ Aceste functii pot avea si diversi operatori de access (si se supun regulilor impuse de acestia - daca un operator este declarat privat atunci nu poate fi accesat decat din interiorul clasei)
- ▶ Operatorii pot fi implementati si in afara claselor - in acest caz daca e nevoie se pot declara ca si functii “friend” pentru a putea accesa membri private dintr-o clasa.

# Supraincarcarea operatorilor

- In cazul de fata se supraincarca operatorul+ pentru operatia de adunare intre un Integer si un alt Integer

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    int operator+ (const Integer &i);
};
int Integer::operator+(const Integer &i)
{
    return value + i.value;
}
void main()
{
    Integer n1(100);
    Integer n2(200);
    int x = n1 + n2;
}
```

# Supraincarcarea operatorilor

- ▶ In cazul de fata se supraincarca operatorul+ pentru operatia de adunare intre un Integer si un alt Integer
- ▶ Operatia se aplica pentru parametrul din stanga iar parametrul din dreapta este dat ca si parametru

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    int operator+ (const Integer &i);
};
int Integer::operator+(const Integer &i)
{
    return value + i.value;
}
void main()
{
    Integer n1(100);
    Integer n2(200);
    int x = n1 + n2;
}
```

200

100

# Supraincarcarea operatorilor

- ▶ Parametrii nu trebuie sa fie neaparat de tipul const sau o referinta
- ▶ In cazul de fata (const Integer &) poate fi fie const Integer sau doar Integer simplu

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    int operator+ (Integer &i);
};
int Integer::operator+(Integer &i)
{
    return value + i.value;
}
void main()
{
    Integer n1(100);
    Integer n2(200);
    int x = n1 + n2;
}
```

# Supraincarcarea operatorilor

- ▶ Valoarea de return nu are neapărat un tip predefinit.
- ▶ În cazul de mai jos, adunarea între două obiecte de tipul Integer da tot un obiect de tipul Integer

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    Integer operator+ (Integer i);
};

Integer Integer::operator+(Integer i)
{
    Integer res(value+i.value);
    return res;
}

void main()
{
    Integer n1(100);
    Integer n2(200);
    Integer n3(0);
    n3 = n1 + n2;
}
```

# Supraincarcarea operatorilor

- ▶ Operatorii functioneaza ca o functie. Pot fi si ei supraincarcati.
- ▶ In cazul de fata clasa Integer suporta operatie de adunare intre doua obiecte de tip Integer, sau intre un obiect de tip Integer si o variabila de tip float

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    int operator+ (const Integer &i) { ... };
    int operator+(float nr);
};
int Integer::operator+(float nr)
{
    return value + (int)nr;
}
void main()
{
    Integer n1(100);
    Integer n2(200);
    int x = n1 + n2;
    int y = n1 + 1.2f;
}
```

# Supraincarcarea operatorilor

- ▶ Operatorii functioneaza ca o functie. Pot fi si ei supraincarcati.
- ▶ In cazul de fata insa, codul nu compileaza pentru ca exista deja o functie “operator+” cu parametru de tip “float”

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    int operator+ (const Integer &i) { ... };
    int operator+(float nr);
    float operator+(float nr);
};
int Integer::operator+(float nr)
{
    return value + (int)nr;
}
void main()
{
    Integer n1(100);
    Integer n2(200);
    int x = n1 + n2;
    int y = n1 + 1.2f;
}
```



# Supraincercarea operatorilor

- ▶ Atentie la ordinea de utilizare a operatorilor.
- ▶ In cazul de fata codul NU compileaza. Clasa Integer suporta adunare intre un Integer si un float, dar nu suporta si invers (intre un float si un Integer). Acest lucru nu e posibil printr-o functie din clasa.

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    int operator+ (const Integer &i) { ... };
    int operator+(float nr);
};
int Integer::operator+(float nr)
{
    return value + (int)nr;
}
void main()
{
    Integer n1(100);
    Integer n2(200);
    int x = n1 + n2;
    int y = 1.2f + n1;
}
```

# Supraincarcarea operatorilor

- Codul compileaza - functiile friend ne rezolva ambele cazuri (Integer+float, si float + Integer)

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    friend int operator+ (const Integer &i, float val);
    friend int operator+ (float val, const Integer &i);
};

int operator+(const Integer &i, float val)
{
    return i.value + (int)val;
}

int operator+(float val, const Integer &i)
{
    return i.value + (int)val;
}

void main()
{
    Integer n1(100);
    Integer n2(200);
    int y = (1.2f+n1)+(n2+1.5f);
}
```

# Supraincarcarea operatorilor

- Codul compileaza - functiile friend ne rezolva ambele cazuri (Integer+float, si float + Integer)

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    friend int operator+ (const Integer &i, float val);
    friend int operator+ (float val, const Integer &i);
};

int operator+(const Integer &i, float val)
{
    return i.value + (int)val;
}

int operator+(float val, const Integer &i)
{
    return i.value + (int)val;
}

void main()
{
    Integer n1(100);
    Integer n2(200);
    int y = (1.2f+n1) + (n2+1.5f);
}
```

# Supraincarcarea operatorilor

- Codul NU compileaza pentru ca sunt 2 functii (una apartinand clasei si alta in afara ei) care se pot folosi pentru a modela adunarea intre obiecte de tipul Integer. Acest caz este ambiguu.

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    int operator+ (Integer i);
    friend int operator+ (Integer n1, Integer n2);
};

int Integer::operator+(Integer i)
{
    return this->value + i.value;
}

int operator+ (Integer n1, Integer n2)
{
    return n1.value + n2.value;
}

void main()
{
    Integer n1(100);
    Integer n2(200);
    n3 = n1 + n2;
}
```

# Supraincercarea operatorilor

- Operatorii de relatie se definesc exact la fel ca si cei aritmetici. Similar cu ei pot fi definiti si ca functii friend in afara clasei.

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    bool operator > (const Integer & i);
};
bool Integer::operator > (const Integer & i)
{
    if (value > i.value)
        return true;
    return false;
}
void main()
{
    Integer n1(100);
    Integer n2(200);
    if (n2 > n1)
        printf("n2 mai mare ca n1");
}
```

# Supraincarcarea operatorilor

- Operatorii de relatie nu trebuie sa returneze un bool chiar daca asta e intelesul lor in mod normal. In cazul de fata, operatorul > returneaza un object.

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    Integer operator > (const Integer & i);
    void PrintValue();
};

void Integer::PrintValue()
{
    printf("Value is %d", value);
}

Integer Integer::operator > (const Integer & i)
{
    Integer res(this->value + i.value);
    return res;
}

void main()
{
    Integer n1(100);
    Integer n2(200);
    (n1 > n2).PrintValue();
}
```

# Supraincarcarea operatorilor

- Acelasi lucru e valabil si pentru operatorii logici (&&, ||, etc)

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    Integer operator && (const Integer & i)
    void PrintValue();
};
void Integer::PrintValue()
{
    printf("Value is %d", value);
}
Integer Integer::operator && (const Integer & i)
{
    Integer res(this->value + i.value);
    return res;
}
void main()
{
    Integer n1(100);
    Integer n2(200);
    (n1 && n2).PrintValue();
}
```

# Supraincarcarea operatorilor

- In cazul operatorilor unari metoda functioneaza simiar. Un operator unar nu are parametru (daca e definit in interiorul clasei) sau un parametru daca e definit ca o functie “friend”
- Deasemenea, nu este nici o obligativitate pentru ce returneaza astfel de functii. In cazul de mai jos x va avea valoare 80.

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    int operator ! ();
};
int Integer::operator ! ()
{
    return 100 - this->value;
}
void main()
{
    Integer n1(20);
    int x = !n1;
}
```



# Supraincarcarea operatorilor

- ▶ Metodele prezentate pot fi aplicate la fel la urmatorii operatori:

+	-	*	/	%	>	<	>=	<=	!=
==	&		&&		^	!	~		

- ▶ Pentru aceste cazuri, recomandarea e sa utilizam functii friend si nu sa creem supraincarcari in interiorul clasei
- ▶ Pe cat posibil, e indicat sa se adauge astfel de functii pentru diverse combinatii de parametru ( clasa cu int, int cu clasa, clasa cu double, double cu clasa, etc)
- ▶ Operatorii pot returna si obiecte si/sau referinte la un obiect. In aceste cazuri obiectul respective este folosit mai departe in evaluarea expresiei din care face parte.

# Supraincercarea operatorilor

- ▶ Functionarea este similara si pentru cazul operatorilor de asignare (=, +=, -=, \*=, etc)
- ▶ Recomandarea e sa se intoarca o referinta la obiectul pentru care se face asignarea

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    Integer& operator = (int val);
};
Integer& Integer::operator = (int val)
{
    value = val;
    return (*this);
}
void main()
{
    Integer n1(20);
    n1 = 20;
}
```

# Supraincarcarea operatorilor

- ▶ Acest lucru insa nu este necesar. In cazul de mai jos, operatorul = returneaza un **true** daca valoarea care se asigneaza in campul Value este para sau **false** altfel
- ▶ Dupa executia codului, n1.value va fi 30, iar res va fi true

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    bool operator = (int val);
};
bool Integer::operator = (int val)
{
    value = val;
    return (val % 2) == 0;
}
void main()
{
    Integer n1(20);
    bool res = (n1 = 30);
}
```

# Supraincarcarea operatorilor

- Diferenta mare este insa ca nu pot exista functii friend prin care sa se suprascrie operatorul de asignare (=) (acesta trebuie sa fie implementat in interiorul clasei din care face parte)

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    friend bool operator = (Integer i, int val);
};

void main()
{
    Integer n1(20);
    bool res = (n1 = 30);
}
```

# Supraincercarea operatorilor

- ▶ Sunt premise insa functii friend pentru operatorii derivati din operatorul de asignare (+=, -=, \*=, etc)
- ▶ In cazul de fata codul compileaza, res va avea valoarea true, iar n1.value va fi 30

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    friend bool operator += (Integer i, int val);
};
bool operator += (Integer i, int val)
{
    i.value = val;
    return true;
}
void main()
{
    Integer n1(20);
    bool res = (n1 += 30);
}
```

# Supraincercarea operatorilor

- ▶ Pentru aceste cazuri, inclusiv ordinea de apel este permisa
- ▶ In cazul de fata, codul compileaza, chiar daca `30 &= n1` nu are sens.
- ▶ In urma executie, `n1.value` va fi 30, iar `res` va fi `true`

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    friend bool operator &= (int val, Integer i);
};
bool operator &= (int val, Integer i)
{
    i.value = val;
    return true;
}
void main()
{
    Integer n1(20);
    bool res = (30 &= n1);
}
```

# Supraincarcarea operatorilor

- Un caz mai diferit se refera la postfix/prefix operators (++ si --)

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    bool operator++ ();
    bool operator++ (int value);
};
bool Integer::operator++ ()
{
    value++;
    return true;
}
bool Integer::operator++ (int val)
{
    value += 2;
    return false;
}
void main()
{
    Integer n1(20);
    bool res = (n1++);
}
```

# Supraincarcarea operatorilor

- In cazul de fata se executa forma postfix (n1.value = 22, res = false)

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    bool operator++ ();
    bool operator++ (int value);
};
bool Integer::operator++ ()
{
    value++;
    return true;
}
bool Integer::operator++ (int val)
{
    value += 2;
    return false;
}
void main()
{
    Integer n1(20);
    bool res = (n1++);
}
```

val=0, trebuie sa fie de  
tip int

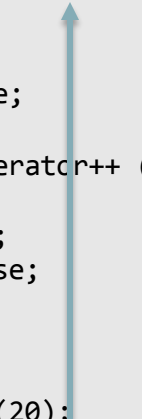


# Supraincarcarea operatorilor

- In cazul de fata se executa forma prefix (n1.value = 21, res = true)

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    bool operator++ ();
    bool operator++ (int value);
};
bool Integer::operator++ ()
{
    value++;
    return true;
}
bool Integer::operator++ (int val)
{
    value += 2;
    return false;
}
void main()
{
    Integer n1(20);
    bool res = (++n1);
}
```



# Supraincarcarea operatorilor

- Operatorii de prefix/postfix pot fi si functii “friend”. In mod normal primul parametru din functia friend trebuie sa fie de tipul referinta. Dupa executie n1.value = 22, res = false

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    friend bool operator++ (Integer &i);
    friend bool operator++ (Integer &i,int value);
};
bool operator++ (Integer &i)
{
    i.value++;
    return true;
}
bool operator++ (Integer &i,int val)
{
    i.value += 2;
    return false;
}
void main()
{
    Integer n1(20);
    bool res = (n1++);
}
```

# Supraincarcarea operatorilor

- ▶ Operatorii postfix/prefix au o semnificatie speciala
  - ▶ PostFix - mai intai se returneaza valoarea si apoi se executa operatia
  - ▶ Prefix - se executa mai intai operatia si apoi se returneaza valoarea

## App.cpp

```
void main()
{
    int x = 3;
    int y;
    y = x++;
    int z;
    z = ++x;
}
```

- ▶ In primul caz, mai intai y ia valoarea lui x, si apoi se face operatia de increment pentru x. Mai exact, y va fi egal cu 3, iar x cu 4.

```
mov    eax,dword ptr [x]
mov    dword ptr [y],eax
mov    ecx,dword ptr [x]
add    ecx,1
mov    dword ptr [x],ecx
```

# Supraincercarea operatorilor

- ▶ Operatorii postfix/prefix au o semnificatie speciala
  - ▶ PostFix - mai intai se returneaza valoarea si apoi se executa operatia
  - ▶ Prefix - se executa mai intai operatia si apoi se returneaza valoarea

## App.cpp

```
void main()
{
    int x = 3;
    int y;
    y = x++;
    int z;
    z = ++x;
}
```

- ▶ In primul caz, mai intai y ia valoarea lui x, si apoi se face operatia de increment pentru x. Mai exact, y va fi egal cu 3, iar x cu 4.
- ▶ In al doilea caz, mai intai se face operatie de increment pentru x si apoi se face asignarea catre z. Mai exact z va fi egal cu 5, iar x tot cu 5

```
mov    eax,dword ptr [x]
add    eax,1
mov    dword ptr [x],eax
mov    ecx,dword ptr [x]
mov    dword ptr [z],ecx
```

# Supraincercarea operatorilor

- Operatorii de prefix/postfix pot fi facuti sa aiba comportamentul asteptat (postfix, prefix) in felul urmator:

## App.cpp

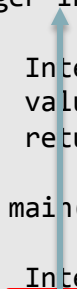
```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    Integer& operator++ ();
    Integer operator++ (int value);
};
Integer& Integer::operator++ ()
{
    value += 1;
    return (*this);
}
Integer Integer::operator++ (int)
{
    Integer tempObject(value);
    value += 1;
    return (tempObject);
}
void main()
{
    Integer n1(20);
    n1++;
}
```

# Supraincercarea operatorilor

- Operatorii de prefix/postfix pot fi facuti sa aiba comportamentul asteptat (postfix, prefix) in felul urmator:

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    Integer& operator++ ();
    Integer operator++ (int value);
};
Integer& Integer::operator++ ()
{
    value += 1;
    return (*this);
}
Integer Integer::operator++ (int)
{
    Integer tempObject(value);
    value += 1;
    return (tempObject);
}
void main()
{
    Integer n1(20);
    n1++;
}
```

A blue arrow originates from the `n1++` expression in the `main` function and points upwards to the `Integer operator++ (int)` method definition in the `Integer` class, illustrating the resolution of the postfix increment operator.

# Supraincercarea operatorilor

- Operatorii de prefix/postfix pot fi facuti sa aiba comportamentul asteptat (postfix, prefix) in felul urmator:

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    Integer& operator++ ();
    Integer operator++ (int value);
};
Integer& Integer::operator++ ()
{
    value += 1;
    return (*this);
}
Integer Integer::operator++ (int)
{
    Integer tempObject(value);
    value += 1;
    return (tempObject);
}
void main()
{
    Integer n1(20);
    ++n1;
}
```

# Supraincarcarea operatorilor

- ▶ Un operator mai special este **new**. New are un format mai special (trebuie sa returneze void\* si are primul parametru de tipul size\_t).
- ▶ Operatorul **new** nu poate fi folosit ca o functie friend.
- ▶ Parametru de tipul size\_t reprezinta dimensiunea obiectului de alocat.
- ▶ Operatorul **new** nu apeleaza si constructorul ci doar alocata o zona de memorie pentru obiectul current. In cazul de fata, dupa executie, GlobalValue = 100

## App.cpp

```
int GlobalValue = 0;
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    void* operator new(size_t t);
};
void* Integer::operator new (size_t t)
{
    return &GlobalValue;
}
void main()
{
    Integer *n1 = new Integer(100);
}
```



# Supraincercarea operatorilor

- Daca operatorul new are mai multi parametrii, restul parametrilor pot fi apelati in felul urmator:

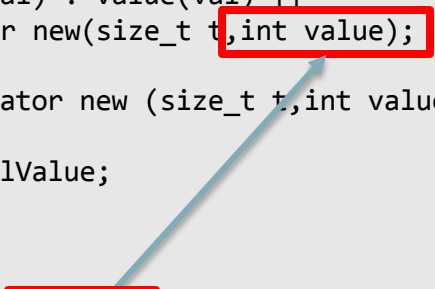
## App.cpp

```
int GlobalValue = 0;

class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    void* operator new(size_t t, int value);
};

void* Integer::operator new (size_t t, int value)
{
    return &GlobalValue;
}

void main()
{
    Integer *n1 = new(100) Integer(123);
}
```



# Supraincercarea operatorilor

- ▶ Un comportament similar il are operatorul new[]. Este folosit pentru alocarea mai multor obiecte de acelasi tip.
- ▶ Formatul e acelasi: returneaza void\*, primeste minim un parametru (primul este obligatoriu de tipul size\_t). In acest caz in acest parametru reprezinta numarul total de octeti de alocat pentru toate elementele din vector
- ▶ Trebuie sa existe si un constructor default pentru a functiona. Dupa executie toate elementele din GlobalValue vor avea valoarea 1

## App.cpp

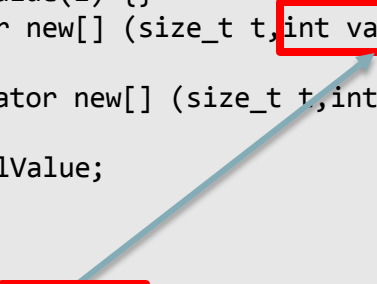
```
int GlobalValue[100];
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    Integer() : value(1) {}
    void* operator new [](size_t t);
};
void* Integer::operator new[] (size_t t) { return &GlobalValue[0]; }
void main()
{
    Integer *n1 = new Integer[100];
}
```

# Supraincercarea operatorilor

- Daca se doreste un operator de tipul new[] cu mai multi parametric, apelul trebuie facut in felul urmator

## App.cpp

```
int GlobalValue[100];
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    Integer() : value(1) {}
    void* operator new[] (size_t t, int value);
};
void* Integer::operator new[] (size_t t, int value)
{
    return &GlobalValue;
}
void main()
{
    Integer *n1 = new(123) Integer[100];
}
```



# Supraincercarea operatorilor

- ▶ La modul general, comportamentul normal pentru operatorii care asigura alocarea este urmatorul:

Operator
<code>void* operator new (size_t size)</code>
<code>void* operator new[] (size_t size)</code>
<code>void operator delete (void* obiect)</code>
<code>void operator delete[] (void* obiects)</code>

- ▶ Este recomandat ca operatorii new si delete sa arunce exceptii

# Supraincercarea operatorilor

- ▶ Un alt tip de operator este operatorul de cast.
- ▶ Mai exact, acest operator permite transformarea unui obiect dintr-un tip in altul
- ▶ Fiind un operator de cast, nu trebuie sa specificam tipul de return (e subinteles)

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    operator float();
};
Integer::operator float()
{
    return float(value * 2);
}
void main()
{
    Integer n1(2);
    float f = (float)n1;
}
```

# Supraincercarea operatorilor

- ▶ Cast-urile se aplica si daca nu le chemam noi explicit
- ▶ Ca si in cazul precedent, valoarea pentru f va fi 4.0
- ▶ Operatorii de cast nu pot fi si functii friend

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    operator float();
};
Integer::operator float()
{
    return float(value * 2);
}
void main()
{
    Integer n1(2);
    float f = n1;
}
```

# Supraincarcarea operatorilor

- Atentie la toti operatorii suprainacarcati. In cazul de fata  $f = 4.2$

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    operator float();
};
Integer::operator float()
{
    return float(value * 2);
}
void main()
{
    Integer n1(2);
    float f = n1 + 0.2f;
}
```

# Supraincercarea operatorilor

- In cazul de fata  $f = 20.2$  pentru ca se apeleaza operatorul de adunare.

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    operator float();
    float operator + (float f);
};
Integer::operator float()
{
    return float(value * 2);
}
float Integer::operator+ (float f)
{
    return value * 10.0f + f;
}
void main()
{
    Integer n1(2);
    float f = n1+0.2f;
}
```



# Supraincercarea operatorilor

- ▶ Operatorii de indexare permit utilizarea [] pentru un anumit obiect.
- ▶ Au o singura restrictie si anume ca au un singur parametru - insa acest parametru poate fi orice, iar valoarea de return poate deasemenea fi de orice tip. Deasemenea, operatorul de indexare nu poate fi functie friend/in afara obiectului curent
- ▶ In cazul de fata, ret = true, pentru ca bitul 1 din n1.value este setat

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    bool operator [](int index);
};
bool Integer::operator [](int index)
{
    return (value & (1 << index)) != 0;
}
void main()
{
    Integer n1(2);
    bool ret = n1[1];
}
```

# Supraincercarea operatorilor

- Exemplul de mai jos foloseste ca si cheie pentru operatorul de indexare o variabila de tipul `const char * name`

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    bool operator [] (const char *name);
};

bool Integer::operator [] (const char *name)
{
    if ((strcmp(name, "first") == 0) && ((value & 1) != 0))
        return true;
    if ((strcmp(name, "second") == 0) && ((value & 2) != 0))
        return true;
    return false;
}

void main()
{
    Integer n1(2);
    bool ret = n1["second"];
}
```

# Supraincercarea operatorilor

- Operatorul de indexare poate exista in mai multe forme (de exemplu cu cheie de tipul `const char*` sau cu cheie de tipul `int`)

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    bool operator [] (const char *name);
    bool operator [] (int index);
};

bool Integer::operator [] (int index)
{
    ...
}

bool Integer::operator [] (const char *name)
{
    ...
}

void main()
{
    Integer n1(2);
    bool ret = n1["second"];
    bool v2 = n1[2];
}
```

# Supraincarcarea operatorilor

- ▶ Operatoriul de apel de functie, functioneaza aproape la fel ca si operatorul de indexare.
- ▶ Ca si operatorul de indexare, operatul de apel de functie () nu poate fi decat o functie membru in cadrul clasei

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    bool operator()(int index);
};

bool Integer::operator()(int index)
{
    return (value & (1 << index)) != 0;
}

void main()
{
    Integer n1(2);
    bool ret = n1(1);
}
```

# Supraincercarea operatorilor

- Diferenta mare e ca operatorul de apel de functie poate avea mai multi parametric (inclusive nici un parametru)

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    int operator()(int start,int end);
};

int Integer::operator()(int start,int end)
{
    return (value >> start) & ((1 << (end - start)) - 1);
}

void xxxmain()
{
    Integer n1(122);
    int res = n1(1,3);
}
```

# Supraincercarea operatorilor

- Operatorul de apel de functie poate exista si fara nici un parametru:

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    int operator ()();
};

int Integer::operator ()()
{
    return (value*2);
}

void main()
{
    Integer n1(122);
    int res = n1();
}
```

# Supraincercarea operatorilor

- ▶ Operatorul de access la membri (->) poate fi deasemenea suprascris.
- ▶ In acest caz, chiar daca nu exista restrictii impuse de compilator, acest operator trebuie sa returneze un pointer catre un obiect.

## App.cpp

```
class MyData
{
    float value;
public:
    void SetValue(float val) { value = val; }
};
class Integer
{
    MyData data;
public:
    MyData* operator -> ();
};
MyData* Integer::operator ->()
{
    return &data;
}
void main()
{
    Integer n1;
    n1->SetValue(100);
}
```

# Supraincarcarea operatorilor

- ▶ Atentie la folosirea lui. Acest operator nu trebuie folosit pe un pointer (in acest caz se aplica forma default).
- ▶ Exemplu de mai jos nu compileaza → SetValue nu e o functie din Integer

## App.cpp

```
class MyData
{
    float value;
public:
    void SetValue(float val) { value = val; }
};
class Integer
{
    MyData data;
public:
    MyData* operator -> ();
};
MyData* Integer::operator ->()
{
    return &data;
}
void main()
{
    Integer n1;
    Integer *n2 = &n1;
    n2->SetValue(100);
}
```



# Supraincercarea operatorilor

- ▶ Daca convertim la obiect pointerul se poate insa folosi:
- ▶ Operatorul “->” nu poate fi definit decat intr-o clasa (nu poate fi definit in afara ei ca si functie friend)

## App.cpp

```
class MyData
{
    float value;
public:
    void SetValue(float val) { value = val; }
};
class Integer
{
    MyData data;
public:
    MyData* operator -> ();
};
MyData* Integer::operator ->()
{
    return &data;
}
void main()
{
    Integer n1;
    Integer *n2 = &n1;
    (*n2)->SetValue(100);
}
```

# Supraincercarea operatorilor

- ▶ Alti operatori care sunt foarte similari cu operatorul “->” ca si comportament si mod de folosinta sunt:
  - ❖ . (A.B)
  - ❖ ->\* (A->\*B)
  - ❖ .\* (A.\*B)
  - ❖ \* (\*A)
  - ❖ & (&A)

# Supraincercarea operatorilor

- ▶ Operatorul pentru lista “,” este folosit in cazurile listelor
- ▶ De exemplu urmatoarea lista se evalueaza de la stanga la dreapta si in lipsa unui operator specific se returneaza ultima valoare:

```
int x = (10,20,30,40)
```

- ▶ Evaluarea se face in felul urmator:
  - ❖ Se evalueaza mai intai expresia “10,20” → care returneaza 20
  - ❖ Apoi se evalueaza expresia “20,30” (20 returnat din expresia precedent) care returneaza 30
  - ❖ Si in final se evalueaza “30,40” care va returna 40

# Supraincercarea operatorilor

- ▶ In cazul de fata, se apeleaza mai intai “operatorul ,” pentru n1 si 2.5f, care returneaza valoarea  $30 * 2.5 = 75$ , si se intoarce aceasta valoare
- ▶ In res la finalul executiei o sa avem valoarea 75

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    int operator ,(float f);
};

int Integer::operator ,(float f)
{
    return (int)(value*f);
}

void main()
{
    Integer n1(30);
    int res = (n1, 2.5f);
}
```

# Supraincercarea operatorilor

- In cazul de fata, se apeleaza mai intai “operatorul ,” pentru n1 si 2.5f, care returneaza valoarea  $30 * 2.5 = 75$ , apoi se aplica “operatorul ,” default intre 75 si 10, care returneaza 10. In final in res vom avea valoarea 10

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    int operator ,(float f);
};
int Integer::operator ,(float f)
{
    return (int)(value*f);
}

void main()
{
    Integer n1(30);
    int res = (n1, 2.5f, 10);
}
```

# Supraincercarea operatorilor

- Se recomanda utilizarea functiilor friend / declararea operatorilor in afara clasei (in acest fel se pot obtine si combinatii de genu (clasa,numar) sau (numar,clasa), etc).

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    friend int operator , (Integer&, float f);
    friend int operator , (int value, Integer&);
};
int operator , (Integer& i, float f)
{
    return (int)(i.value*f);
}
int operator , (int value, Integer &i)
{
    return value + i.value;
}
void main()
{
    Integer n1(30);
    int res = (n1, 2.5f, n1);
}
```

# Supraincarcarea operatorilor

- Se recomanda utilizarea functiilor friend / declararea operatorilor in afara clasei (in acest fel se pot obtine si combinatii de genu (clasa,numar) sau (numar,clasa), etc).

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    friend int operator , (Integer&, float f);
    friend int operator , (int value, Integer&);
};
int operator , (Integer& i, float f)
{
    return (int)(i.value*f);
}
int operator , (int value, Integer &i)
{
    return value + i.value;
}
void main()
{
    Integer n1(30);
    int res = (n1, 2.5f, n1);
}
```

(n1,2.5f)=75

# Supraincarcarea operatorilor

- Se recomanda utilizarea functiilor friend / declararea operatorilor in afara clasei (in acest fel se pot obtine si combinatii de genu (clasa,numar) sau (numar,clasa), etc).

## App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    friend int operator , (Integer&, float f);
    friend int operator , (int value, Integer&);
};
int operator , (Integer& i, float f)
{
    return (int)(i.value*f);
}
int operator , (int value, Integer &i)
{
    return value + i.value;
}
void main()
{
    Integer n1(30);
    int res = (n1, 2.5f, n1);
}
```

(75,n1) = 105



- ▶ Functii “Friend”
- ▶ Operatori
  - ▶ Tipuri de operatori
  - ▶ Ordinea operatorilor
  - ▶ Operatori si clase (supraincercarea operatorilor)
- ▶ Operatii cu obiecte

# Operatii cu obiecte

- Fie urmatorul cod:

## App.cpp

```
class Date
{
    int X,Y,Z,T;
public:
    Date(int value) : X(value), Y(value + 1),
                     Z(value + 2), T(value+3) {}
    void SetX(int value) { X = value; }
};
void Set(Date d,int value)
{
    d.SetX(value);
}
void main()
{
    Date d(1);
    Set(d,123);
}
```

```
push    123
sub     esp,10h
mov     eax,esp
mov     ecx,dword ptr [d.X]
mov     dword ptr [eax],ecx
mov     edx,dword ptr [d.Y]
mov     dword ptr [eax+4],edx
mov     ecx,dword ptr [d.Z]
mov     dword ptr [eax+8],ecx
mov     edx,dword ptr [d.T]
mov     dword ptr [eax+0Ch],edx
call    Set
add     esp,14h
```

# Operatii cu obiecte

- Fie urmatorul cod:

## App.cpp

```
class Date
{
    int X,Y,Z,T;
public:
    Date(int value) : X(value), Y(value + 1),
                     Z(value + 2), T(value+3) {}
    void SetX(int value) { X = value; }
};
void Set(Date &d,int value)
{
    d.SetX(value);
}
void main()
{
    Date d(1);
    Set(d,123);
}
```

push 123  
lea eax,[d]  
push eax  
call Set  
add esp,8

# Operatii cu obiecte

- ▶ Cand se da un parametru unei functii avem urmatoarele cazuri:
  - a) Daca parametrul e de tipul referinta/pointer - se copie pe stiva doar adresa lui
  - b) Daca parametrul e un obiect, se copie tot acel obiect pe stiva sii va putea fi accesat ca orice parametru copiat pe stiva (relative la  $[EBP+xxx]$  )

# Operatii cu obiecte

- Fie urmatorul cod:

## App.cpp

```
class Date
{
    int X,Y,Z,T;
public:
    Date(int value) : X(value), Y(value + 1),
                    Z(value + 2), T(value+3)
    void SetX(int value) { X = value; }
};
Date Get(int value)
{
    Date d(value);
    return d;
}
void main()
{
    Date d(1);
    d = Get(100);
}
```

```
Date Get(int value)
{
    push    ebp
    mov     ebp,esp
    Date d(value);
    mov     eax,dword ptr [value]
    push    eax
    lea     ecx,[d]
    call    Date::Date
    return d;
    mov     eax,dword ptr [ebp+8]
    mov     ecx,dword ptr [d.X]
    mov     dword ptr [eax],ecx
    mov     edx,dword ptr [d.Y]
    mov     dword ptr [eax+4],edx
    mov     ecx,dword ptr [d.Z]
    mov     dword ptr [eax+8],ecx
    mov     edx,dword ptr [d.T]
    mov     dword ptr [eax+12],edx
    mov     eax,dword ptr [ebp+8]
}
    mov     esp,ebp
    pop     ebp
    ret
```

# Operatii cu obiecte

- Fie urmatorul cod:

## App.cpp

```
class Date
{
    int X,Y,Z,T;
public:
    Date(int value) : X(value), Y(value + 1),
                     Z(value + 2), T(value+3)
    void SetX(int value) { X = value; }
};
Date Get(int value)
{
    Date d(value);
    return d;
}
void main()
{
    Date d(1);
    d = Get(100);
}
```

```
Date d(1);
push    1
lea     ecx,[d]
call    Date::Date
d = Get(100);
push    64h
lea     eax,[ebp-0F0h]
push    eax
call    Get
add     esp,8
mov     ecx,dword ptr [eax]
mov     dword ptr [d.X],ecx
mov     edx,dword ptr [eax+4]
mov     dword ptr [d.Y],edx
mov     ecx,dword ptr [eax+8]
mov     dword ptr [d.Z],ecx
mov     edx,dword ptr [eax+0Ch]
mov     dword ptr [d.T],edx
}
```

# Operatii cu obiecte

- Fie urmatorul cod:

## App.cpp

```
class Date
{
    int X,Y,Z,T;
public:
    Date(int value) : X(value), Y(value + 1),
                     Z(value + 2), T(value+3) {}
    void SetX(int value) { X = value; }
};
Date Get(int value)
{
    Date d(value);
    return d;
}
void main()
{
    Date d(1);
    d = Get(100);
}
```

## App.pseudocode

```
class Date
{
    ...
};
Date* Get(Date *tempObject, int value)
{
    Date d(value);
    memcpy(tempObject,&d,sizeof(Date));
    return tempObject;
}
void main()
{
    Date d(1);
    unsigned char temp[sizeof(Date)]
    Date* tmpObj = Get(temp,100);
    memcpy(d,tmpObj,sizeof(Date))
}
```

# Operatii cu obiecte

## ► Constructor de copiere

### App.cpp

```
class Date
{
    int X,Y,Z,T;
public:
    Date(int value) : X(value), Y(value + 1),
                    Z(value + 2), T(value+3) {}
    void SetX(int value) { X = value; }
    Date(const Date & obj) { X = obj.X; }
};
Date Get(int value)
{
    Date d(value);
    return d;
}
void main()
{
    Date d(1);
    d = Get(100);
}
```

### App.pseudocode

```
class Date
{
    ...
};

Date* Get(Date *tempObject, int value)
{
    Date d(value);
    tempObject->Date(d);
    return tempObject;
}

void main()
{
    Date d(1);
    unsigned char temp[sizeof(Date)]
    Date* tempObj = Get(temp,100);
    memcpy(d,tempObj,sizeof(Date))
}
```



# Operatii cu obiecte

## ► Constructor de copiere

### App.cpp

```
class Date
{
    int X,Y,Z,T;
public:
    Date(int value) : X(value), Y(value + 1),
                    Z(value + 2), T(value+3) {}
    void SetX(int value) { X = value; }
    Date(const Date & obj) { X = obj.X; }
    Date& operator = (Date &d)
    {
        X = d.X;
        return (*this);
    }
};
Date Get(int value)
{
    Date d(value);
    return d;
}
void main()
{
    Date d(1);
    d = Get(100);
}
```

### App.pseudocode

```
class Date
{
    ...
};
Date* Get(Date *tempObject, int value)
{
    Date d(value);
    tempObject->Date(d);
    return tempObject;
}
void main()
{
    Date d(1);
    unsigned char temp[sizeof(Date)]
    Date* tmpObj = Get(temp,100);
    d.operator=(*tmpObj);
}
```

# Operatii cu obiecte

- Care este problema de la codul de mai jos ?

## App.cpp

```
class Date
{
    int X,Y,Z,T;
public:
    Date(int value) : X(value), Y(value + 1),
                     Z(value + 2), T(value+3) {}
    void SetX(int value) { X = value; }
    Date(const Date & obj) { X = obj.X; }
    Date& operator = (Date &d)
    {
        X = d.X;
        return (*this);
    }
};
Date& Get(int value)
{
    Date d(value);
    return d;
}
void main()
{
    Date d(1);
    d = Get(100);
}
```

## App.pseudocode

```
class Date
{
    ...
};
Date* Get(int value)
{
    Date d(value);
    return &d;
}
void main()
{
    Date d(1);
    Date* tmpObj = Get(100);
    d.operator=(*tmpObj);
}
```

# Operatii cu obiecte

- Care este problema de la codul de mai jos ?

## App.cpp

```
class Date
{
    int X,Y,Z,T;
public:
    Date(int value) : X(value), Y(value + 1),
                     Z(value + 2), T(value+3) {}
    void SetX(int value) { X = value; }
    Date(const Date & obj) { X = obj.X; }
    void operator = (Date &d)
    {
        X = d.X;
    }
};

Date& Get(int value)
{
    Date d(value);
    return d;
}

void main()
{
    Date d(1);
    d = Get(100);
}
```

```
void operator = (Date &d)
{
    push    ebp
    mov     ebp,esp
    sub     esp,0CCh
    push    ebx
    push    esi
    push    edi
    push    ecx
    lea     edi,[ebp-0CCh]
    mov     ecx,33h
    mov     eax,0CCCCCCCCh
    rep stos dword ptr es:[edi]
    pop     ecx
    mov     dword ptr [this],ecx
    X = d.X;
    mov     eax,dword ptr [this]
    mov     ecx,dword ptr [d]
    mov     edx,dword ptr [ecx]
    mov     dword ptr [eax],edx
    }

    pop     edi
    pop     esi
    pop     ebx
    mov     esp,ebp
    pop     ebp
    ret     4
```