

The background features abstract, overlapping green geometric shapes, primarily triangles and polygons, in various shades of green, creating a modern, layered effect.

# P00

Curs-2

Gavrilut Dragos

- ▶ Trecerea de la C la C++
- ▶ Referinte si pointeri
- ▶ Clase
  - ▶ Modificatori de access
  - ▶ Date membru
  - ▶ Functii membru (metode)
  - ▶ Constructori
  - ▶ Destructori

- ▶ Trecerea de la C la C++
- ▶ Referinte si pointeri
- ▶ Clase
  - ▶ Modificatori de access
  - ▶ Date membru
  - ▶ Functii membru (metode)
  - ▶ Constructori
  - ▶ Destructori

# C to C++

- Fie urmatorul cod scris in C

## App.cpp

```
struct Person
{
    int Varsta;        // ANI
    int Inaltime;      // Centimetri
}
void main()
{
    Person p;
    printf("Varsta = %d",p.Varsta);
    p.Varsta = -5;
    p.Inaltime = 100000;
}
```

- Ce problem observam la acest cod (de natura logica) ?

# C to C++

- Fie urmatorul cod scris in C

## App.cpp

```
struct Person
{
    int Varsta;      // ANI
    int Inaltime;    // Centimetri
}
void main()
{
    Person p;
    printf("Varsta = %d",p.Varsta);
    p.Varsta = -5;
    p.Inaltime = 100000;
}
```

- Programul e correct din punct de vedere sintactic, dar din punct de vedere logic valorile pentru campurile **Varsta** si **Inaltime** nu au sens !
- Nu exista nici o forma de initializarea a variabilei **p**. Functia printf va afisa o valoarea **nedefinita** !!!

# C to C++

- Solutia e sa cream functii care initializeze si sa valideze aceste valori

## App.c

```
struct Person
{
    int Varsta;      // ANI
    int Inaltime;    // Centimetri
}
void main()
{
    Person p;
    printf("Varsta = %d",p.Varsta);
    p.Varsta = -5;
    p.Inaltime = 100000;
}
```



## App.c

```
struct Person
{
    int Varsta;      // ANI
    int Inaltime;    // Centimetri
}
void Init(Person *p)
{
    p->Varsta = 10;
    p->Inaltime = 100;
}
void SetVarsta(Person *p,int value)
{
    if ((value>0) && (value<200))
        p->Varsta = value;
}
void SetInaltime(Person *p,int value)
{
    if ((value>50) && (value<300))
        p->Inaltime = value;
}
void main()
{
    Person p;
    Init(&p);
    SetVarsta(&p, -5);
    SetInaltime(&p, 100000);
}
```

# C to C++

- Aceasta abordare desi are o serie de avantaje vine si cu o serie de probleme:

## App.c

```
struct Person
{
    int Varsta;      // ANI
    int Inaltime;    // Centimetri
}
void Init(Person *p)
{
    p->Varsta = 10;
    p->Inaltime = 100;
}
void SetVarsta(Person *p,int value)
{
    if ((value>0) && (value<200))
        p->Varsta = value;
}
void SetInaltime(Person *p,int value)
{
    if ((value>50) && (value<300))
        p->Inaltime = value;
}
void main()
{
    Person p;
    Init(&p);
    SetVarsta(&p, -5);
    SetInaltime(&p, 100000);
}
```

# C to C++

- Aceasta abordare desi are o serie de avantaje vine si cu o serie de probleme:

## App.c

```
struct Person
{
    int Varsta;      // ANI
    int Inaltime;    // Centimetri
}
void Init(Person *p) { ... }

void SetVarsta(Person *p,int value)
{
    if ((value>0) && (value<200) && (p!=NULL))
        p->Varsta = value;
}
void SetInaltime(Person *p,int value) { ... }

void main()
{
    Person p;
    Init(&p);
    SetVarsta(&p, -5);
    SetInaltime(&p, 100000);
}
```

- a) Pointerul p din functiile SetVarsta si SetInaltime trebuie validat (trebuie sa fie valid)



# C to C++

- Aceasta abordare desi are o serie de avantaje vine si cu o serie de probleme:

## App.c

```
struct Person
{
    int Varsta;      // ANI
    int Inaltime;    // Centimetri
}
void Init(Person *p) { ... }

void SetVarsta(Person *p,int value) { ... }

void SetInaltime(Person *p,int value) { ... }

void main()
{
    Person p;
    Init(&p);
    SetVarsta(&p, -5);
    SetInaltime(&p, 100000);
    p.Varsta = -1;
    p.Inaltime = -2;
}
```

- a) Pointerul p din functiile SetVarsta si SetInaltime trebuie validat (trebuie sa fie valid)
- b) In continuare valorile pentru campurile Varsta si Inaltime se pot seta direct fara nici o validare

# C to C++

- Aceasta abordare desi are o serie de avantaje vine si cu o serie de probleme:

## App.c

```
struct Person
{
    int Varsta;        // ANI
    int Inaltime;      // Centimetri
}
void Init(Person *p) { ... }

void SetVarsta(Person *p,int value) { ... }

void SetInaltime(Person *p,int value) { ... }

void main()
{
    Person p;
    Init(&p);
    printf("Varsta = %d",p.Varsta);
}
```

- a) Pointerul p din functiile SetVarsta si SetInaltime trebuie validat (trebuie sa fie valid)
- b) In continuare valorile pentru campurile Varsta si Inaltime se pot seta direct fara nici o validare
- c) Initializarea nu este **implicita** (daca uitam sa o apelam **explicit** din cod, valoarea campurilor Varsta si Inaltime o sa fie nedefinita)

# C to C++

- Aceasta abordare desi are o serie de avantaje vine si cu o serie de probleme:

## App.c

```
struct Person
{
    int Varsta;      // ANI
    int Inaltime;    // Centimetri
}
void Init(Person *p) {...}
void SetVarsta(Person *p, int value) {...}
void SetInaltime(Person *p, int value) {...}
void AddYear(Person *p, int value) {...}
void AddHeight(Person *p, int value) {...}
int GetVarsta(Person *p) {...}
int GetInaltime(Person *p) {...}
```

- a) Pointerul p din functiile SetVarsta si SetInaltime trebuie validat (trebuie sa fie valid)
- b) In continuare valorile pentru campurile Varsta si Inaltime se pot seta direct fara nici o validare
- c) Initializarea nu este **implicita** (daca uitam sa o apelam **explicit** din cod, valoarea campurilor Varsta si Inaltime o sa fie nedefinita)
- d) Daca avem multe functii asociate unei structuri trebuie sa avem grija sa pasam pointerul catre un obiect al acelei structuri de fiecare data

# C to C++

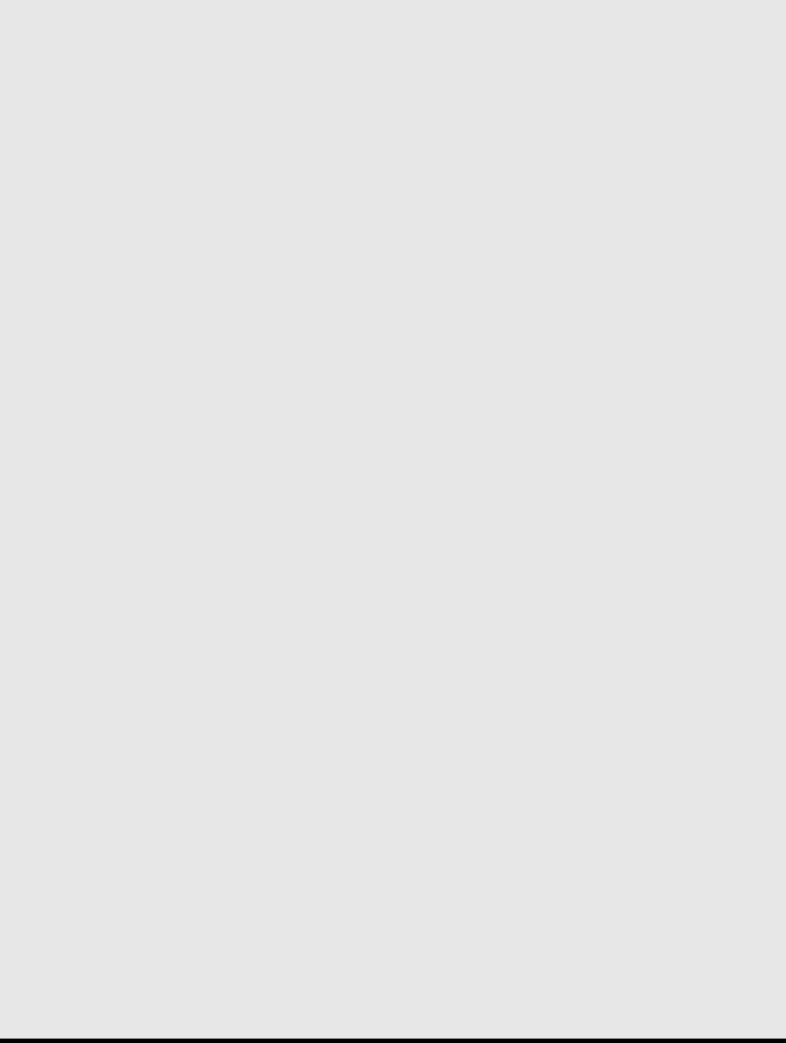
- ▶ Practic ne trebuie o solutie la nivel de limbaj care sa asigure urmatoarele:
  - ▶ Sa se poata restrictiona accesul la unele campuri ale structurii
  - ▶ Sa exista macar o functie de initializare care sa fie apelata automat in momentul in care se creaza un obiect de tipul structurii in cauza
  - ▶ Sa nu trebuiasca sa dam acel pointer catre obiectul structurii de fiecare data cand apelam o functie care modifica campuri ale acelei structure
  - ▶ Sa nu trebuiasca sa validam acel pointer (sa fie validat de catre compilator implicit)

# C to C++ (Conversion)

## App.c

```
struct Person
{
    int Varsta;    // ANI
}
void SetVarsta(Person *p,int value)
{
    if (p==NULL)
        return;
    if ((value>0) && (value<200))
        p->Varsta = value;
}
void Init(Person *p)
{
    if (p==NULL)
        return;
    p->Varsta = 10;
}
void main()
{
    Person p;
    Init(&p);
    SetVarsta(&p,10);
}
```

## App.cpp



# C to C++ (Conversion)

## App.c

```
struct Person
{
    int Varsta;    // ANI
}
void SetVarsta(Person *p,int value)
{
    if (p==NULL)
        return;
    if ((value>0) && (value<200))
        p->Varsta = value;
}
void Init(Person *p)
{
    if (p==NULL)
        return;
    p->Varsta = 10;
}
void main()
{
    Person p;
    Init(&p);
    SetVarsta(&p,10);
}
```

## App.cpp

```
class Person
{
```

# C to C++ (Conversii)

## App.c

```
struct Person
{
    int Varsta;    // ANI
}
void SetVarsta(Person *p,int value)
{
    if (p==NULL)
        return;
    if ((value>0) && (value<200))
        p->Varsta = value;
}
void Init(Person *p)
{
    if (p==NULL)
        return;
    p->Varsta = 10;
}
void main()
{
    Person p;
    Init(&p);
    SetVarsta(&p,10);
}
```

## App.cpp

```
class Person
{
```

**private:**

Modificator de access  
(specifica cine poate  
accesa variabilele care  
urmeaza dupa el)

# C to C++ (Conversii)

## App.c

```
struct Person
{
    int Varsta; // ANI
}
void SetVarsta(Person *p,int value)
{
    if (p==NULL)
        return;
    if ((value>0) && (value<200))
        p->Varsta = value;
}
void Init(Person *p)
{
    if (p==NULL)
        return;
    p->Varsta = 10;
}
void main()
{
    Person p;
    Init(&p);
    SetVarsta(&p,10);
}
```

## App.cpp

```
class Person
{
    private:
        int Varsta;
```



# C to C++ (Conversii)

## App.c

```
struct Person
{
    int Varsta;    // ANI
}
void SetVarsta(Person *p,int value)
{
    if (p==NULL)
        return;
    if ((value>0) && (value<200))
        p->Varsta = value;
}
void Init(Person *p)
{
    if (p==NULL)
        return;
    p->Varsta = 10;
}
void main()
{
    Person p;
    Init(&p);
    SetVarsta(&p,10);
}
```

## App.cpp

```
class Person
{
    private:
        int Varsta;
    public:
        void SetVarsta(int value);
}
```

# C to C++ (Conversii)

## App.c

```
struct Person
{
    int Varsta;    // ANI
}
void SetVarsta(Person *p,int value)
{
    if (p==NULL)
        return;
    if ((value>0) && (value<200))
        p->Varsta = value;
}
void Init(Person *p)
{
    if (p==NULL)
        return;
    p->Varsta = 10;
}
void main()
{
    Person p;
    Init(&p);
    SetVarsta(&p,10);
}
```

## App.cpp

```
class Person
{
    private:
        int Varsta;
    public:
        void SetVarsta(int value):
        Person();
}
```

Constructor

# C to C++ (Conversii)

## App.c

```
struct Person
{
    int Varsta;    // ANI
}
void SetVarsta(Person *p,int value)
{
    if (p==NULL)
        return;
    if ((value>0) && (value<200))
        p->Varsta = value;
}
void Init(Person *p)
{
    if (p==NULL)
        return;
    p->Varsta = 10;
}
void main()
{
    Person p;
    Init(&p);
    SetVarsta(&p,10);
}
```

## App.cpp

```
class Person
{
    private:
        int Varsta;
    public:
        void SetVarsta(int value);
        Person();
}
void Person::SetVarsta(int value)
{
    if ((value>0) && (value<200))
        this->Varsta = value;
}
```

# C to C++ (Conversii)

## App.c

```
struct Person
{
    int Varsta;    // ANI
}
void SetVarsta(Person *p,int value)
{
    if (p==NULL)
        return;
    if ((value>0) && (value<200))
        p->Varsta = value;
}
void Init(Person *p)
{
    if (p==NULL)
        return;
    p->Varsta = 10;
}
void main()
{
    Person p;
    Init(&p);
    SetVarsta(&p,10);
}
```

## App.cpp

```
class Person
{
    private:
        int Varsta;
    public:
        void SetVarsta(int value);
        Person();
}
void Person::SetVarsta(int value)
{
    if ((value>0) && (value<200))
        this->Varsta = value;
}
```

# C to C++ (Conversii)

## App.c

```
struct Person
{
    int Varsta;    // ANI
}
void SetVarsta(Person *p,int value)
{
    if (p==NULL)
        return;
    if ((value>0) && (value<200))
        p->Varsta = value;
}
void Init(Person *p)
{
    if (p==NULL)
        return;
    p->Varsta = 10;
}
void main()
{
    Person p;
    Init(&p);
    SetVarsta(&p,10);
}
```

## App.cpp

```
class Person
{
    private:
        int Varsta;
    public:
        void SetVarsta(int value);
        Person();
}
void Person::SetVarsta(int value)
{
    if ((value>0) && (value<200))
        this->Varsta = value;
}
Person::Person()
{
    this->Varsta = 10;
}
```

# C to C++ (Conversii)

## App.c

```
struct Person
{
    int Varsta;    // ANI
}
void SetVarsta(Person *p,int value)
{
    if (p==NULL)
        return;
    if ((value>0) && (value<200))
        p->Varsta = value;
}
void Init(Person *p)
{
    if (p==NULL)
        return;
    p->Varsta = 10;
}
void main()
{
    Person p;
    Init(&p);
    SetVarsta(&p,10);
}
```

## App.cpp

```
class Person
{
    private:
        int Varsta;
    public:
        void SetVarsta(int value);
        Person();
}
void Person::SetVarsta(int value)
{
    if ((value>0) && (value<200))
        this->Varsta = value;
}
Person::Person()
{
    this->Varsta = 10;
}
void main()
{
    Person p;
```

Constructorul se apeleaza implicit cand se creaza un obiect de tipul Person

# C to C++ (Conversii)

## App.c

```
struct Person
{
    int Varsta;    // ANI
}
void SetVarsta(Person *p,int value)
{
    if (p==NULL)
        return;
    if ((value>0) && (value<200))
        p->Varsta = value;
}
void Init(Person *p)
{
    if (p==NULL)
        return;
    p->Varsta = 10;
}
void main()
{
    Person p;
    Init(&p);
    SetVarsta(&p,10);
}
```

## App.cpp

```
class Person
{
    private:
        int Varsta;
    public:
        void SetVarsta(int value);
        Person();
}
void Person::SetVarsta(int value)
{
    if ((value>0) && (value<200))
        this->Varsta = value;
}
Person::Person()
{
    this->Varsta = 10;
}
void main()
{
    Person p;
    p.SetVarsta(10);
}
```

# C to C++ (Conversii)

## App.c

```
struct Person
{
    int Varsta;    // ANI
}
void SetVarsta(Person *p,int value)
{
    if (p==NULL)
        return;
    if ((value>0) && (value<200))
        p->Varsta = value;
}
void Init(Person *p)
{
    if (p==NULL)
        return;
    p->Varsta = 10;
}
void main()
{
    Person p;
    Init(&p);
    SetVarsta(&p,10);
    p.Varsta = -1;
}
```

Compileaza si modifica  
valoarea campului  
Varsta

## App.cpp

```
class Person
{
    private:
        int Varsta;
    public:
        void SetVarsta(int value);
        Person();
}
void Person::SetVarsta(int value)
{
    if ((value>0) && (value<200))
        this->Varsta = value;
}
Person::Person()
{
    this->Varsta = 10;
}
void main()
{
    Person p;
    p.SetVarsta(10);
    p.Varsta = -1;
}
```

Eroare la compilare -  
campul Varsta este  
declarant ca si privat



- ▶ Trecerea de la C la C++
- ▶ Referinte si pointeri
- ▶ Clase
  - ▶ Modificatori de access
  - ▶ Date membru
  - ▶ Functii membru (metode)
  - ▶ Constructori
  - ▶ Destructori

# Referinte si Pointeri

## App-Pointer

```
void SetInt(int *i)
{
    (*i) = 5;
}
void main()
{
    int x;
    Set(&x);
}
```

## App-Referinta

```
void SetInt(int &i)
{
    i = 5;
}
void main()
{
    int x;
    Set(x);
}
```

## App-Pointer (asm - SetInt)

```
SetInt:
    push    ebp
    mov     ebp,esp
    mov     eax,[ebp+8]
    mov     [eax],5
    mov     esp,ebp
    pop     ebp
    ret
```

## App-Referinta (asm - SetInt)

```
SetInt:
    push    ebp
    mov     ebp,esp
    mov     eax,[ebp+8]
    mov     [eax],5
    mov     esp,ebp
    pop     ebp
    ret
```

# Referinte si Pointeri

- Din perspectiva codului final care rezulta in urma compilarii, nu exista diferente intre conceptual de pointer si cel de referinta (ambele se traduc in exact acelasi cod in limbaj masina)
- Din perspectiva programatorului, referinta rezolva o serie de probleme , poate cea mai cunoscuta dintre ele fiind faptul ca nu mai e nevoie sa folosim operatorul “->” ci putem folosi “.”

## Pointer

```
struct Date
{
    int X;
}
void SetInt(Date *d)
{
    d->X = 5;
}
```

## Referinta

```
struct Date
{
    int X;
}
void SetInt(Date &d)
{
    d.X = 5;
}
```

# Referinte si Pointeri

- Referintele si pointerii se creaza/initializeaza in felul urmator:

## Pointer

```
int i = 10;  
int *p = &i;
```

## Referinta

```
int i = 10;  
int &refI = i;
```

- Diferenta e ca pointerii pot sa ramana neinitilzati, in timp ce referintele nu.

## Pointer

```
int i = 10;  
int *p;
```

## Referinta

```
int i = 10;  
int &refI;
```

Eroare la compilare -  
referinta neinitializata

Acest lucru practice forteaza programatorul sa initilizeze o referinta. Mai exact, avem garantia ca o referinta duce la o zona de memorie valida.

# Referinte si Pointeri

- Pointerii isi pot schimba valoarea (pot sa puncteze la mai multe variabile) pe parcusul executiei unui program. Referintele pot puncta doar la o singura variabila cu care au fost initializati.

## Pointer

```
int i = 10;  
int j = 20;  
int *p = &i;  
p = &j;
```

- Un pointer poate avea o valoare NULL.  
O referinta trebuie sa puncteze la o zona de memorie valida.

## Referinta

```
int i = 10;  
int j = 20;  
int &refI = i;  
&refI = j;
```

Eroare la compilare - o data ce a fost initializata referinta nu isi mai poate schimba valoarea

# Referinte si Pointeri

- Pointerii accepta anumite operatii aritmetice (+, -, ++, etc). Referintele nu accepta acest lucru.

## Pointer

```
int i = 10;  
int j = 20;  
int *p = &i;  
p++;  
(*p) = 30;
```

## Referinta

```
int i = 10;  
int j = 20;  
int &refI = i;  
refI++;  
(&refI)++;
```

Eroare la compilare

- In cazul pointerilor, variabilele “i” si “j” sunt consecutive pe stiva. Operatia “p++” muta pointerul p de la variabila “i” la variabila “j”. La finalul executiei codului de mai sus, j va avea valoarea 30

# Referinte si Pointeri

- Pointerii accepta cast-uri intre ei. In particular orice pointer accepta implicit un cast la un pointer de tipul void (void\*). Referintele NU accepta cast-uri.

## Pointer

```
int i = 10;  
char *p = (char *)&i;
```

## Referinta

```
int i = 10;  
char &refI = i;
```

Eroare la compilare

- Acest lucru are rolul de a garanta ca referinta puncteaza la o variabila de un anumit tip

# Referinte si Pointeri

- Pointerii accepta si multiple indirectari (un pointer poate puncta catre un alt pointer). O referinta insa puncteaza tot timpul catre un singur obiect de un anumit tip.

## Pointer

```
int i = 10;  
int *p = &i;  
int *p_to_p = &p;  
**p_to_p = 20;
```

## Referinta

```
int i = 10;  
int &refI = i;  
int & &ref_to_refI = refI;
```

Eroare la compilare



# Referinte si Pointeri

- Pointerii pot fi utilizati in array-uri (si initializati dinamic in acestea). Referintele insa nu pot fi legate de un array (nu se poate crea un array de referinte):

## Pointer

```
int *p[100];
```

## Referinta

```
int &ref[100];
```

Eroare la compilare

- In schimb o referinta poate sa puncteze la o valoarea temporara.

## Pointer

```
int *p = &int(10);
```

## Referinta

```
const int &redI = int(12);
```

Pentru cazul de fata a trebui sa utilizam si cuvantul cheie **const**. 12 (chiar daca e o valoarea temporara) e considerate o valoarea constanta.

- ▶ Trecerea de la C la C++
- ▶ Referinte si pointeri
- ▶ **Clase**
  - ▶ Modificatori de access
  - ▶ Date membru
  - ▶ Functii membru (metode)
  - ▶ Constructori
  - ▶ Destructori

# Clase (format)

## ► Date membru

- ❖ Variabile definite in clasa
- ❖ Fiecare data membru poate avea propriul ei modificador de access
- ❖ Datele membru pot fi si statice (in acest caz apartin clasei si nu instantei)
- ❖ O clasa poate sa nu aiba nici o data membru

## ► Functii membru (metode)

- ❖ Functii definite intr-o clasa (mai poarta si numele de metode)
- ❖ Fiecare metoda poate avea propriul ei modificador de access
- ❖ Orice metoda poate accesa orice data membru definita in clasa din care face parte indiferent de modificadorul ei de access
- ❖ O clasa poate sa nu aiba nici o metoda

## ► Constructori

- ❖ Functii care nu au tip (se considera implicit ca sunt de tipul void) care se apeleaza cand se creaza un obiect de tipul clasei din care fac parte
- ❖ Pot lipsi
- ❖ Pot avea diversi modificatori de access

## ► Destructor

- ❖ Functii care nu au tip (se considera implicit ca sunt de tipul void) care se apeleaza cand se distruge un obiect de tipul clasei din care fac parte
- ❖ Pot lipsi

## ► Operatori

- ▶ Trecerea de la C la C++
- ▶ Referinte si pointeri
- ▶ Clase
  - ▶ **Modificatori de access**
  - ▶ Date membru
  - ▶ Functii membru (metode)
  - ▶ Constructori
  - ▶ Destructori

# Clase (modificatori de access)

- ▶ C++ accepta 3 modificatori de access:
  - ▶ public (permite accesul la acel membru atat din interiorul clasei cat si din afara ei)
  - ▶ private (accesul la acel membru se poate face doar din interiorul clasei). Este modifierul de access implicit pentru clase daca nu se specifica altceva.
  - ▶ protected

# Clase (modificatori de access)

## App.cpp

```
class Person
{
    public:
        int Varsta;
}
void main()
{
    Person p;
    p.Varsta = 10;
}
```

- ▶ Codul se compileaza si ruleaza corect.
- ▶ Membru Varsta este public in clasa Person si poate fi accesat si din afara clasei

# Clase (modificatori de access)

## App.cpp

```
class Person
{
    private:
        int Varsta;
}
void main()
{
    Person p;
    p.Varsta = 10;
}
```

- Codul nu se compileaza (membrul Varsta din clasa Person este privat si nu poate fi accesat din afara clasei).

# Clase (modificatori de access)

## App.cpp

```
class Person
{
    private:
        int Varsta;
    public:
        void SetVarsta(int val);
}
void Person::SetVarsta(int val)
{
    this->Varsta = val;
}
void main()
{
    Person p;
    p.SetVarsta(10);
}
```

- ▶ Codul este correct si se compileaza.
- ▶ Din afara clasei se apeleaza doar metoda SetVarsta care este publica in clasa Person
- ▶ Orice metoda definite intr-o clasa (fie publica, fie private) poate accesa orice alt membru (variabila) definita in acea clasa indiferent daca specificatorul de access este public sau privat.



# Clase (modificatori de access)

## App.cpp

```
class Person
{
    int Varsta;
}
void main()
{
    Person p;
    p.Varsta = 10;
}
```

- ▶ Codul nu se compileaza (membrul Varsta din clasa Person este privat si nu poate fi accesat din afara clasei).
- ▶ In lipsa unui specificator de access, implicit se considera private (din acest motiv membru Varsta este privat)

- ▶ Trecerea de la C la C++
- ▶ Referinte si pointeri
- ▶ Clase
  - ▶ Modificatori de access
  - ▶ **Date membru**
  - ▶ Functii membru (metode)
  - ▶ Constructori
  - ▶ Destructori

# Clase (date membru)

## App.cpp

```
class Person
{
    private:
        int Varsta, Inaltime;
    public:
        char *Name;
}
void main()
{
    Person p;
}
```

- ▶ Datele membru reprezinta variabilele care sunt definite in cadrul unei clase
- ▶ In cazul de fata **Varsta** si **Inaltime** sunt private, iar **Name** este public

# Clase (date membru)

## App.cpp

```
class Person
{
    private:
        int Varsta, Inaltime;
    public:
        char *Name;
}
void main()
{
    Person p;
    p.Varsta = 10;
}
```

- Codul nu compileaza pentru ca Varsta este data membru privata

# Clase (date membru)

## App.cpp

```
class Person
{
    private:
        int Varsta, Inaltime;
    public:
        char *Name;
}
void main()
{
    Person p;
    p.Name = "Popescu";
}
```

► Codul compileaza pentru ca Name este public.

# Clase (date membru)

## App.cpp

```
class Person
{
    private:
        int Varsta, Inaltime;
        static int X;
    public:
        char *Name;
        static int Y;
}
```

- Datele membru pot sa fie si statice. In acelasi timp pot sa aiba si un modificador de access

# Clase (date membru)

## App.cpp

```
class Person
{
    private:
        int Varsta, Inaltime;
        static int X;
    public:
        char *Name;
        static int Y;
}
int Person::X;
int Person::Y = 10;
```

- ▶ Datele membru pot sa fie si statice. In acelasi timp pot sa aiba si un modificador de access
- ▶ Orice variabila statica dintr-o clasa trebuie sa fie definite si in afara clasei (ca o variabila globala). Daca nu se defineste linkerul nu poate linka.
- ▶ Optional se poate si initializa. Daca nu se initializeaza o variabila statica are valoarea 0.

# Clase (date membru)

## App.cpp

```
class Person
{
    private:
        int Varsta, Inaltime;
        static int X;
    public:
        char *Name;
        static int Y;
}
int Person::X;
int Person::Y = 10;

void main()
{
    Person p;
    p.Y = 5;
    Person::Y++;
}
```

- ▶ Membri statici dintr-o clasa pot sa fie accesati fie din orice variabila de tipul acelei clase, fie ca o referinta pentru numele clasei
- ▶ In cazul de fata dupa exacutia codului variabila statica Y va avea valoarea 6.



# Clase (date membru)

## App.cpp

```
class Person
{
    private:
        int Varsta, Inaltime;
        static int X;
    public:
        char *Name;
        static int Y;
}
int Person::X;
int Person::Y = 10;

void main()
{
    Person p;
    p.X = 6;
}
```

- ▶ Codul nu compileaza pentru X este membru privat
- ▶ Este nevoie sa creem o functie ca sa putem accesa acea valoare

# Clase (date membru)

## App.cpp

```
class Person
{
    private:
        int Varsta, Inaltime;
        static int X;
    public:
        char *Name;
        static int Y;
        void SetX(int value);
}
int Person::X;
int Person::Y = 10;

void Person::SetX(int value)
{
    X = value;
}

void main()
{
    Person p;
    p.SetValue(6);
}
```

- ▶ Codul compileaza si seteaza valoarea lui X la 6.
- ▶ Functia care seteaza valoarea lui X poate sa fie definite ca o functie membru (metoda) dar poate fi definita si ca o functie statica si apelata ca referinta a clasei direct.

# Clase (date membru)

## App.cpp

```
class C1
{
    int X,Y;
};
class C2
{
    int X,Y;
    static int Z;
};
class C3
{
    static int T;
};
class C4
{
};
int C2::Z;
int C3::T;
void main()
{
    printf("sizeof(C1)=%d",sizeof(C1));
    printf("sizeof(C2)=%d",sizeof(C2));
    printf("sizeof(C3)=%d",sizeof(C3));
    printf("sizeof(C4)=%d",sizeof(C4));
}
```

- ▶ Codul compileaza
- ▶ Variabilele statice sunt considerate variabile globale si nu conteaza in calculul dimensiunii unui obiect de tipul unei clase anume
- ▶ Pot exista clase care sa nu aiba nici o variabila membru - in acest caz dimensiunea unei variabile de acest tip este de 1 octet
- ▶ La executie programul va afisa:

sizeof(C1) = 8

sizeof(C2) = 8

sizeof(C3) = 1

sizeof(C4) = 1

# Clase (date membru)

## App.cpp

```
class Date
{
public:
    int X,Y;
    static int Z;
};
int Date::Z;
void main()
{
    Date d1,d2,d3;
}
```

## Address

## Name

## Value

100000

Date::Z

0

300000

d1.X

?

300004

d1.Y

?

300008

d2.X

?

300012

d2.Y

?

300016

d3.X

?

300020

d3.Y

?

# Clase (date membru)

## App.cpp

```
class Date
{
public:
    int X,Y;
    static int Z;
};
int Date::Z;
void main()
{
    Date d1,d2,d3;
    d1.Z = 5;
}
```

Address	Name	Value
100000	Date::Z	5
300000	d1.X	?
300004	d1.Y	?
300008	d2.X	?
300012	d2.Y	?
300016	d3.X	?
300020	d3.Y	?

# Clase (date membru)

## App.cpp

```
class Date
{
public:
    int X,Y;
    static int Z;
};
int Date::Z;
void main()
{
    Date d1,d2,d3;
    d1.Z = 5;
    d1.X = 7;
}
```

Address	Name	Value
100000	Date::Z	5
300000	d1.X	7
300004	d1.Y	?
300008	d2.X	?
300012	d2.Y	?
300016	d3.X	?
300020	d3.Y	?

# Clase (date membru)

## App.cpp

```
class Date
{
public:
    int X,Y;
    static int Z;
};
int Date::Z;
void main()
{
    Date d1,d2,d3;
    d1.Z = 5;
    d1.X = 7;
    d2.Y = d3.Z + 1;
}
```

Address	Name	Value
100000	Date::Z	5
300000	d1.X	7
300004	d1.Y	?
300008	d2.X	?
300012	d2.Y	6
300016	d3.X	?
300020	d3.Y	?

# Clase (date membru)

## App.cpp

```
class Date
{
public:
    int X,Y;
    static int Z;
};
int Date::Z;
void main()
{
    Date d1,d2,d3;
    d1.Z = 5;
    d1.X = 7;
    d2.Y = d3.Z + 1;
    Date::Z = d2.Z + 1;
}
```

Address	Name	Value
100000	Date::Z	6
300000	d1.X	7
300004	d1.Y	?
300008	d2.X	?
300012	d2.Y	6
300016	d3.X	?
300020	d3.Y	?



# Clase (date membru)

## App.cpp

```
class Date
{
public:
    int X,Y;
    static int Z;
};
int Date::Z;
void main()
{
    Date d1,d2,d3;
    d1.Z = 5;
    d1.X = 7;
    d2.Y = d3.Z + 1;
    Date::Z = d2.Z + 1;
    d3.X = d2.Z+d1.Z-1;
}
```

Address	Name	Value
100000	Date::Z	6
300000	d1.X	7
300004	d1.Y	?
300008	d2.X	?
300012	d2.Y	6
300016	d3.X	11
300020	d3.Y	?

- ▶ Trecerea de la C la C++
- ▶ Referinte si pointeri
- ▶ Clase
  - ▶ Modificatori de access
  - ▶ Date membru
  - ▶ Functii membru (metode)
  - ▶ Constructori
  - ▶ Destructori

# Clase (metode)

## App.cpp

```
class Person
{
    private:
        int Varsta;
        bool CheckValid(int val);
    public:
        void SetVarsta(int val);
};
bool Person::CheckValid(int val)
{
    return ((val>0) && (val<200));
}
void Person::SetVarsta(int val)
{
    if (CheckValid(val))
        this->Varsta = val;
}
void main()
{
    Person p;
    p.SetVarsta(40);
}
```

- ▶ Metodele sunt functii definite in cadrul unei clase. De obicei rolul lor este sa opereze pe datele membru din clasa (in special pe cele private).
- ▶ La fel ca si datele membru, ele pot si publice sau private. In cazul de fata **SetVarsta** este o metoda publica iar **CheckValid** este private
- ▶ O metoda poate accesa orice alta metoda declarata in clasa respective indiferent de specificatorul de access

# Clase (metode)

## App.cpp

```
class Person
{
private:
    int Varsta;
public:
    static bool Check(int val);
    void SetVarsta(int val);
};

bool Person::Check(int val)
{
    return ((val>0) && (val<200));
}

void Person::SetVarsta(int val)
{
    if (Check(val))
        this->Varsta = val;
}

void main()
{
    Person p;
    if (Person::Check(40))
    {
        printf("40 is a valid age");
    }
}
```

- O metoda poate fi statica si in acelasi timp sa aiba si un modificador de access (public/private/ etc)

# Clase (metode)

## App.cpp

```
class Person
{
private:
    int Varsta;
    static bool Check(int val);
public:
    void SetVarsta(int val);
};
bool Person::Check(int val)
{
    return ((val>0) && (val<200));
}
void Person::SetVarsta(int val)
{
    if (Check(val))
        this->Varsta = val;
}
void main()
{
    Person p;
    if (Person::Check(40))
    {
        printf("40 is a valid age");
    }
}
```

► Codul nu compileaza pentru ca metoda Check este privata

# Clase (metode)

## App.cpp

```
class Person
{
private:
    int Varsta;
    static bool Check(int val);
public:
    void SetVarsta(int val);
};
bool Person::Check(int val)
{
    return ((val>0) && (val<200));
}
void Person::SetVarsta(int val)
{
    if (Check(val))
        this->Varsta = val;
}
void main()
{
    Person p;
    p.SetVarsta(40);
}
```

- ▶ Codul se compileaza - SetVarsta este metoda publica si poate fi apelata
- ▶ Orice metoda (chiar si privata cum este cazul lui **Check**) poate fi accesata de o alta metoda declarata in clasa respectiva (in cazul de fata **SetVarsta**)

# Clase (metode)

## App.cpp

```
class Date
{
private:
    int X;
    static int Y;
public:
    static void Increment();
};
int Date::Y = 0;
void Date::Increment()
{
    Y++;
}
void main()
{
    Date::Increment();
}
```

- ▶ O metoda statica poate accesa un membru static declarant in aceeași clasă indiferent de specificatorul de acces al acelui membru
- ▶ In acest exemplu, funcția Increment adaugă 1 la membrul static Y din clasă Date

# Clase (metode)

## App.cpp

```
class Date
{
private:
    int X;
    static int Y;
public:
    static void Increment();
};
int Date::Y = 0;
void Date::Increment()
{
    X++;
}
void main()
{
    Date::Increment();
}
```

- ▶ Codul nu compileaza
- ▶ O functie statica **NU POATE ACCESA** un membru care nu este si el static din clasa
- ▶ De asemenea, o functie statica nu poate accesa pointerul **this**



# Clase (metode) - operatorul const

- ▶ In momentul in care se definesc metode care apartin unei clase, pe langa specificatorii de access se mai poate folosi un operator special (**const**)
- ▶ Urmatorul cod compileaza fara problem. La sfarsitul executiei valoarea variabilei x din obiectul "d" va fi 1;

## App.cpp

```
class Date
{
    private:
        int x;
    public:
        int& GetX();
};
int& Date::GetX()
{
    x = 0;
    return x;
}
void main()
{
    Date d;
    d.GetX()++;
}
```

# Clase (metode) - operatorul const

- Codul nu mai compileaza pentru ca functia GetX() returneaza un numar constant. Ceea ce inseamna ca operatorul “++” din “d.GetX()++” ar trebui sa modifice un numar care este constant.

## App.cpp

```
class Date
{
    private:
        int x;
    public:
        const int& GetX();
};
const int& Date::GetX()
{
    x = 0;
    return x;
}
void main()
{
    Date d;
    d.GetX()++;
}
```

# Clase (metode) - operatorul const

- ▶ Codul compileaza. Metoda GetX() returneaza o referinta constanta a carei valori este 0 care se copie in variabila x locala care apoi poate fi modificata.
- ▶ Se recomanda folosirea acestei solutii daca vrem sa dam access read-only la o variabila membru.

## App.cpp

```
class Date
{
    private:
        int x;
    public:
        const int& GetX();
};
const int& Date::GetX()
{
    x = 0;
    return x;
}
void main()
{
    Date d;
    int x = d.GetX();
    x++;
}
```

# Clase (metode) - operatorul const

- Codul nu compileaza. Utilizarea const la finalul unei metode specifica faptul ca in acea metoda NU SE POT modifica date membru ale clasei din care face parte. Practic **const** de la final transforma obiectul current intr-un obiect constant care nu poate modifica valorile sale pana cand se iese din functie.

## App.cpp

```
class Date
{
    private:
        int x;
    public:
        const int& GetX() const;
};
const int& Date::GetX() const
{
    x = 0;
    return x;
}
void main()
{
    Date d;
    int x = d.GetX();
    x++;
}
```

# Clase (metode) - operatorul const

- Codul comileaza pentru x nu mai e un membru al unei instante ci un membru static global (el nu apartine obiectului).

## App.cpp

```
class Date
{
    private:
        static int x;
    public:
        const int& GetX() const;
};
int Date::x = 100;
const int& Date::GetX() const
{
    x = 0;
    return x;
}
void main()
{
    Date d;
    int x = d.GetX();
    x++;
}
```

# Clase (metode) - operatorul const

- Codul nu compileaza pentru ca modificatorul const de la final nu poate fi folosit pentru functii statice.

## App.cpp

```
class Date
{
    private:
        static int x;
    public:
        static const int& GetX() const;
};
int Date::x = 100;
Static const int& Date::GetX() const
{
    x = 0;
    return x;
}
void main()
{
    Date d;
    int x = d.GetX();
    x++;
}
```

# Clase (metode) - operatorul const

- ▶ “const” face parte din tipul obiectului.
- ▶ O metoda dintr-o clasa (chiar daca nu are “const” la finalul declaratiei) daca este apelata pe un obiect constant va esua.

## Fara const

```
class Date
{
private:
    int x;
public:
    void Inc();
};
void Date::Inc()
{
    x++;
}
void Increment(Date &d)
{
    d.Inc();
}
void main()
{
    Date d;
    Increment(d);
}
```

## Cu const

```
class Date
{
private:
    int x;
public:
    void Inc();
};
void Date::Inc()
{
    x++;
}
void Increment(const Date &d)
{
    d.Inc();
}
void main()
{
    Date d;
    Increment(d);
}
```

Eroare la compilare,  
d este const

# Clase (metode) - autoreferinte

- ▶ Uneori este foarte util sa putem sa returnam o referinta catre obiectul current (auto-referinta).
- ▶ Acest lucru permite apelul mai mai multor functii in serie;

## App.cpp

```
class Date
{
private:
    int x;
public:
    void Init();
    Date& Inc();
};
void Date::Init()
{
    x = 0;
}
Date& Date::Inc()
{
    x++;
    return (*this);
}
void main()
{
    Date d;
    d.Init();
    d.Inc().Inc().Inc().Inc();
}
```



# Clase (metode) - supraincarcare metode

- ▶ La modul formal, o metoda din cadrul unei clase are urmatoarea definitie:  
`<tip_return> nume_metoda ( <lista_parametri> )`
- ▶ Din perspectiva compilatorului insa, o metoda este combinatia intre `nume_metoda` si `<lista_parametri>`
- ▶ Implicit, acest lucru inseamna ca pot exista metode ale caror nume este identic cata vreme lista de parametri pentru fiecare dintre ele este diferita.
- ▶ Mai mult, valoarea de return nu are nici o semnificatie → mai exact putem avea functii cu acelasi nume, care au ca si tip de return tipuri de date diferite, cata vreme lista de parametri pentru fiecare functie este diferita.
- ▶ Crearea de mai multe metode cu acelasi nume, dar parametric diferiti poarta numele de supraincarcare (overloading)

# Clase (metode) - supraincarcare metode

## App.cpp

```
class Math
{
public:
    int  Add (int v1, int v2);
    int  Add (int v1, int v2, int v3);
    int  Add (int v1, int v2, int v3, int v4);
    float Add (float v1, float v2);
};

int Math::Add(int v1, int v2)
{
    return v1 + v2;
}

int Math::Add(int v1, int v2, int v3)
{
    return v1 + v2 + v3;
}

int Math::Add(int v1, int v2, int v3, int v4)
{
    return v1 + v2 + v3 + v4;
}

float Math::Add(float v1, float v2)
{
    return v1 + v2;
}
```

# Clase (metode) - supraincarcare metode

- ▶ Exista si o serie de cazuri de eroare in care supraincarcarea nu se poate realiza.
- ▶ In cazul de mai jos, exista doua metoda cu numele Add si cu doi parametric de tipul int

## App.cpp

```
class Math
{
public:
    int Add(int v1, int v2);
    long Add(int v1, int v2);
};
int Math::Add(int v1, int v2)
{
    return v1 + v2;
}
long Math::Add(int v1, int v2)
{
    return v1 + v2;
}
```

# Clase (metode) - supraincarcare metode

- Atentie si la parametric cu valori implicite. Din punct de vedere a compilatorului, parametru cu valoare implicita specifica ca nu trebuie completata acea valoare si NU CA acea valoare nu exista. Functia are din punct de vedere al compilatorului tot 2 parametri.

## App.cpp

```
class Math
{
public:
    int  Add(int v1, int v2);
    long Add(int v1, int v2 = 0);
};
int Math::Add(int v1, int v2)
{
    return v1 + v2;
}
long Math::Add(int v1, int v2)
{
    return v1 + v2;
}
```

# Clase (metode) - supraincarcare metode

- Functiile cu numar variabil de parametric sunt insa acceptate de compilator. Chiar si asa , ele nu sunt recomandate pentru ca pot crea problem de interpretare din partea compilatorului

## App.cpp

```
class Math
{
public:
    int  Add(int v1, int v2);
    long Add(int v1, ...);
};
int Math::Add(int v1, int v2)
{
    return v1 + v2;
}
long Math::Add(int v1, ...)
{
    return v1;
}
```

# Clase (metode) - supraincarcare metode

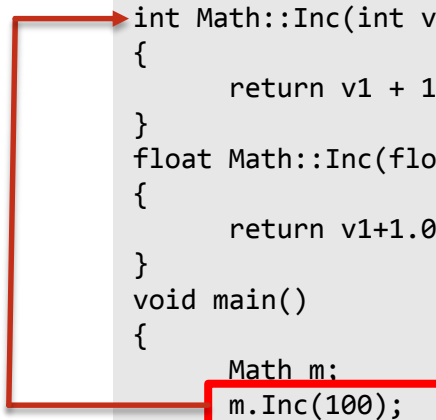
- ▶ Utilizarea supraincarcarii poate aduce vine si cu o serie de probleme. De exemplu utilizarea unor tipuri de date diferite de cele utilizate in parametrii metodelor supraincarcate.
- ▶ Pentru aceste cazuri compilatorul face urmatorii pasi:
  - ❖ Cauta o metoda pentru care sa aiba o potrivire exacta (parametric sa fie de exact acelasi tip)
  - ❖ Daca nu gaseste, **promoveaza** parametric existenti si verifica din nou. Promovarea presupune transformarea unor tipuri de baza in altele, astfel
    - ✓ char, short, unsigned char si unsigned short sunt convertite in int
    - ✓ float este convertit la double
    - ✓ Orice enumerare este convertita la int
  - ❖ Daca nu se gaseste un match - se incearca un cast standard
    - ✓ Pointerii se transforma in void\*
    - ✓ 0 ca si constanta numerica poate fi considera si pointer NULL
    - ✓ Valorile numerice se transforma in alte tipuri care le pot suporta valoarea (char in float, sau char in unsigned int, etc)
  - ❖ Daca nici asa nu se gaseste un match, se verifica daca nu exista o conversie explicita intre anumite tipuri de date (declarata de programator)

# Clase (metode) - supraincarcare metode

- ▶ 100 este considerat valoare de tip int. Cum exista metoda cu numele Inc si parametru de tipul (int) se foloseste acea metoda.

## App.cpp

```
class Math
{
public:
    int Inc(int v1);
    float Inc(float v1);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
float Math::Inc(float v1)
{
    return v1+1.0f;
}
void main()
{
    Math m;
    m.Inc(100);
}
```

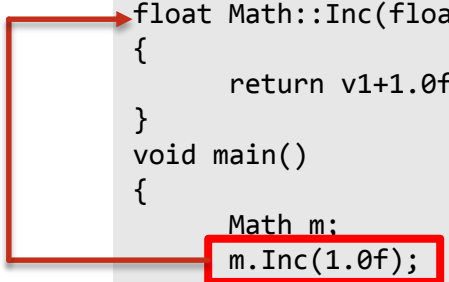
A red line with an arrow points from the `m.Inc(100);` call in the `main` function to the `int Math::Inc(int v1)` definition. Another red line with an arrow points from the same call to the `int Inc(int v1);` declaration inside the `Math` class, illustrating how the compiler resolves the method call to the correct overload based on the argument type (100 is an integer).

# Clase (metode) - supraincarcare metode

- 1.0f este considerat valoare de tip float. Cum exista metoda cu numele Inc si parametru de tipul (float) se foloseste acea metoda.

## App.cpp

```
class Math
{
public:
    int Inc(int v1);
    float Inc(float v1);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
float Math::Inc(float v1)
{
    return v1+1.0f;
}
void main()
{
    Math m;
    m.Inc(1.0f);
}
```

A red line originates from the `m.Inc(1.0f);` line in the `main()` function and points to the `float Math::Inc(float v1)` method definition in the `Math` class, illustrating the method resolution process based on the argument type.

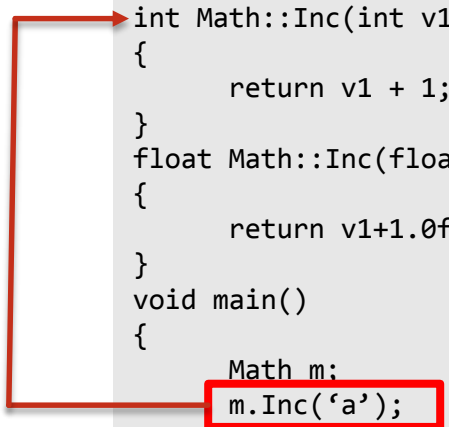


# Clase (metode) - supraincarcare metode

- 'a' este considerat valoare de tip char. Nu exista o metoda cu numele Inc si care sa primeasca un parametru de tipul char, asa ca valoare de tip char se promoveaza la int si se apeleaza metoda Inc cu parametru de tipul int.

## App.cpp

```
class Math
{
public:
    int Inc(int v1);
    float Inc(float v1);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
float Math::Inc(float v1)
{
    return v1+1.0f;
}
void main()
{
    Math m;
    m.Inc('a');
}
```

A red line originates from the `m.Inc('a');` call in the `main` function. It extends to the left and then turns upwards to point at the `int Math::Inc(int v1)` method definition, illustrating that the character 'a' is promoted to an integer and the integer version of the `Inc` method is called.

# Clase (metode) - supraincarcare metode

- 1.0 este de tipul double. Nu exista o metoda Inc care sa primeasca ca si parametru un double. Se incearca promovarea, dar cu double e pe mai multi biti decat int sau float nu se poate transforma. Nu se poate face nici cast in ceva mi mic, asa ca codul nu compileaza.

## App.cpp

```
class Math
{
public:
    int Inc(int v1);
    float Inc(float v1);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
float Math::Inc(float v1)
{
    return v1+1.0f;
}
void main()
{
    Math m;
    m.Inc(1.0);
}
```

# Clase (metode) - supraincarcare metode

- 'a' este de tipul char. Nu avem functie Inc cu parametric de tipul char asa ca se face promovarea la int. Nu avem nici functie Inc cu parametru int asa ca se incearca cast standard. "char" poate fi convertit atat in unsigned int cat si in float. Cum avem doua astfel de functii, avem o ambiguitate si nu va compila.

## App.cpp

```
class Math
{
public:
    int Inc(unsigned int v1);
    float Inc(float v1);
};
int Math::Inc(unsigned int v1)
{
    return v1 + 1;
}
float Math::Inc(float v1)
{
    return v1+1.0f;
}
void main()
{
    Math m;
    m.Inc('a');
}
```

# Clase (metode) - supraincarcare metode

- Atentie la cum folositi functiile cu numar variabil de parametric. Desi in cazul de mai jos ambele variante ale functiei Inc ar fi putut fi apelate, compilatorul foloseste cea care se potriveste.

## App.cpp

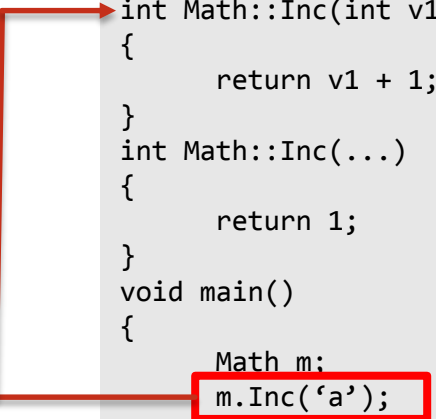
```
class Math
{
public:
    int Inc(int v1);
    int Inc(...);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
int Math::Inc(...)
{
    return 1;
}
void main()
{
    Math m;
    m.Inc(123);
}
```

# Clase (metode) - supraincarcare metode

- Similar, operatorul “...” nu se considera ca face match cu ‘a’ asa ca se face promovare si se apeleaza functia Inc(int)

## App.cpp

```
class Math
{
public:
    int Inc(int v1);
    int Inc(...);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
int Math::Inc(...)
{
    return 1;
}
void main()
{
    Math m;
    m.Inc('a');
}
```

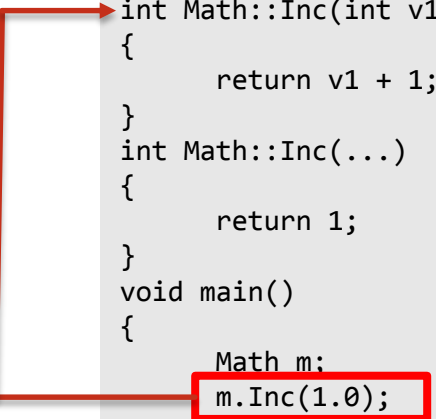


# Clase (metode) - supraincarcare metode

- ▶ 1.0 este double. Nici in acest caz nu se face match. Nu se poate face nici promovare asa ca se face cast la int (compilatorul afiseaza un warning) insa nu se apeleaza Int(...)

## App.cpp

```
class Math
{
public:
    int Inc(int v1);
    int Inc(...);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
int Math::Inc(...)
{
    return 1;
}
void main()
{
    Math m;
    m.Inc(1.0);
}
```



# Clase (metode) - supraincarcare metode

- Suntem pe cazul cu 2 parametric. Singura solutie este Inc(...) asa ca se apeleaza aceasta.

## App.cpp

```
class Math
{
public:
    int Inc(int v1);
    int Inc(...);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
int Math::Inc(...)
{
    return 1;
}
void main()
{
    Math m;
    m.Inc(1.0,2);
}
```

# Clase (metode) - supraincarcare metode

- ▶ Caz ambiguu. Atat `Inc(int)` cat si `Inc(int,...)` se potrivesc cu `Inc(123)`.
- ▶ Nu se compileaza in acest caz.

## App.cpp

```
class Math
{
public:
    int Inc(int v1);
    int Inc(int v1,...);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
int Math::Inc(int v1,...)
{
    return 1;
}
void main()
{
    Math m;
    m.Inc(123);
}
```



- ▶ Trecerea de la C la C++
- ▶ Referinte si pointeri
- ▶ Clase
  - ▶ Modificatori de access
  - ▶ Date membru
  - ▶ Functii membru (metode)
  - ▶ **Constructori**
  - ▶ Destructori

# Constructori

- ▶ Constructorii sunt functii fara tip (void) definite intr-o clasa care se apeleaza cand se initializeaza acea clasa
- ▶ Constructorii respecta conceptual de supraincarcare de la metoda (pot exista mai multi constructori cu diversi parametric)
- ▶ Constructorii nu sunt obligatorii insa daca este macar unul present crearea unui obiect trebuie sa se faca respectand parametrii constructorilor acestuia.
- ▶ O clasa care contine mai multe date membru, va apela constructorii pentru datele membru (daca acestia au fost definiti) in ordinea in care acele date membru au fost create in clasa.
- ▶ Constructorii nu pot fi static si nici constanti (utilizarea cuvintului const dupa numele constructorului)
- ▶ O clasa care contine o data membru constanta sau o data membru care este referinta **TREBUIE** sa aiba un constructor
- ▶ Constructorul fara nici un parametru se mai numeste si constructor implicit
- ▶ Un constructor poate sa aiba orice specificator de access (public/private/etc)

# Constructori

- Constructorul implicit este apelat si seteaza valoarea lui X la 10

## App.cpp

```
class Date
{
private:
    int x;
public:
    Date();
};
Date::Date()
{
    x = 10;
}
void main()
{
    Date d;
}
```

# Constructori

- ▶ "d" este construit folosind constructorul implicit
- ▶ "d2" este construit folosind constructorul cu un singur parametru

## App.cpp

```
class Date
{
private:
    int x;
public:
    Date();
    Date(int value);
};
Date::Date()
{
    x = 10;
}
Date::Date(int value)
{
    x = value;
}
void main()
{
    Date d;
    Date d2(100);
}
```

# Constructori

- Datele membru dintr-o clasa pot fi instantiate automat in cadrul constructorului adaugand dupa definirea constructorului numele datei membru urmat de valoare/valori. Daca data membru e un clasa asa se poate apela un constructor al acelei clase

## App.cpp

```
class Date
{
private:
    int x;
public:
    Date();
};
Date::Date() : x(100)
{
}
void main()
{
    Date d;
}
```

# Constructori

- Codul nu compileaza - clasa Date are un constructor si este privat.

## App.cpp

```
class Date
{
private:
    int x;
private:
    Date();
};
Date::Date() : x(100)
{
}
void main()
{
    Date d;
}
```

# Constructorii

- Codul nu compileaza - clasa Date are un membru const care trebuie sa fie initializat

## App.cpp

```
class Date
{
private:
    int x;
    const int y;
public:

};

void main()
{
    Date d;
}
```

# Constructori

- Codul nu compileaza - clasa Date are acum un constructor, dar nu initializeaza membrul y

## App.cpp

```
class Date
{
private:
    int x;
    const int y;
public:
    Date();
};
Date::Date() : x(100)
{
}

void main()
{
    Date d;
}
```



# Constructori

- Codul compileaza. Y este initializat cu valoarea 123

## App.cpp

```
class Date
{
private:
    int x;
    const int y;
public:
    Date();
};
Date::Date() : x(100), y(123)
{
}

void main()
{
    Date d;
}
```

# Constructorii

- ▶ Codul nu compileaza.
- ▶ Toti membri const al unei clase trebuie initializat in fiecare constructor al acelei clase

## App.cpp

```
class Date
{
private:
    int x;
    const int y;
public:
    Date();
    Date(int value);
};
Date::Date() : x(100), y(123)
{
}
Date::Date(int value) : x(value)
{
}
}
void main()
{
    Date d;
    Date d2(100);
}
```

# Constructorii

- Codul compileaza corect.

## App.cpp

```
class Date
{
private:
    int x;
    const int y;
public:
    Date();
    Date(int value);
};
Date::Date() : x(100), y(123)
{
}
Date::Date(int value) : x(value), y(value*value)
{

}
void main()
{
    Date d;
    Date d2(100);
}
```

# Constructori

- ▶ Constructorii sunt apelati cand se creaza un obiect de tipul clasei din care fac parte.
- ▶ Acest lucru inseamna ca sunt apelati cand se creaza un obiect pe stiva, sau cand se creaza un obiect pe heap (prin operatorul new)
- ▶ In cazul array-urilor, se apeleaza pentru fiecare element din array
- ▶ NU se apeleaza daca cream un pointer catre un obiect de tipul clasei din care face parte

## App.cpp

```
class Date
{
    ...
}
void main()
{
    Date d;                // se apeleaza constructorul
    Date *d2 = new Date(); // se apeleaza constructorul
    Date arr[100];         // se apeleaza constructorul de 100 de ori
    Date *d3;              // NU se apeleaza constructorul
}
```

- ▶ Trecerea de la C la C++
- ▶ Referinte si pointeri
- ▶ Clase
  - ▶ Modificatori de access
  - ▶ Date membru
  - ▶ Functii membru (metode)
  - ▶ Constructori
  - ▶ **Destructor**

# Destructorul

- ▶ Destructorul este apelat cand se doreste curatarea memoriei ocupate de un obiect
- ▶ Destructorul (daca exista) este unul singur pentru o clasa si nu are nici un parametru
- ▶ Destructorul nu poate fi static
- ▶ Destructorul poate avea modificatori de access (deobicei este public).

## App.cpp

```
class Date
{
private:
    int x;
public:
    Date();
    ~Date();
};
Date::Date() : x(100) { ... }
Date::~~Date() { ... }

void main()
{
    Date d;
}
```

# Destructorul

- Codul nu compileaza pentru ca in functia main nu se poate apela destructorul pentru obiectul “d” - destructorul este privat.

## App.cpp

```
class Date
{
private:
    int x;
public:
    Date();
    private:
        ~Date();
};
Date::Date() : x(100) { ... }
Date::~~Date() { ... }

void main()
{
    Date d;
}
```

# Destructorul

- Codul compileaza - destructorul nu se mai apeleaza implicit la iesirea din functia main pentru ca obiectul a fost alocat pe heap.

## App.cpp

```
class Date
{
private:
    int x;
public:
    Date();
private:
    ~Date();
};
Date::Date() : x(100) { ... }
Date::~~Date() { ... }

void main()
{
    Date *d = new Date();
}
```



# Destructorul

- Codul nu compileaza. La apelul “delete d” compilatorul nu poate accesa destructorul din Date pentru ca este private.

## App.cpp

```
class Date
{
private:
    int x;
public:
    Date();
private:
    ~Date();
};
Date::Date() : x(100) { ... }
Date::~~Date() { ... }

void main()
{
    Date *d = new Date();
    delete d;
}
```

# Destructorul

- Codul compileaza. Destructorul se apeleaza (insa dintr-o functie statica din cadrul clasei - DestroyData).

## App.cpp

```
class Date
{
private:
    int x;
public:
    Date();
    static void DestroyData(Date *d);
private:
    ~Date();
};
Date::Date() : x(100) { ... }
Date::~~Date() { ... }
void Date::DestroyData(Date *d)
{
    delete d;
}
void main()
{
    Date *d = new Date();
    Date::DestroyData(d);
}
```