

The background features abstract, overlapping geometric shapes in various shades of purple and blue, creating a modern, layered effect. The shapes are primarily triangles and polygons, some with thin white outlines.

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 apelează implicit când se creează 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
declarat 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;
    SetInt(&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

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

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 conceptul 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

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 numerica constanta. Daca nu folosim **const** codul nu va compila.

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];
```

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

Pointer

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

Referinta

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

Se pot crea insa pointeri catre o referinta care puncteaza la o valoarea temporara.

Referinte si Pointeri

- Pointerii catre o referinta numerica sunt pointeri pe stiva. Utilizarea lor poate produce rezultate inconsistene cu algoritmul.

App.cpp

```
int* GetPtr()
{
    const int &redI = int(12);
    return (int *)&redI;
}
void ProcessInt(int *i)
{
    int x[100];
    for (int tr = 0; tr < 100; tr++) {
        x[tr] = tr;
        (*i) += tr;
    }
}
void main()
{
    int *a = GetPtr();
    ProcessInt(a);
}
```

ASM - GetPtr

```
push    ebp
mov     ebp, esp
sub     esp, 0x8
mov     dword [ebp-0x4], 0xc
lea     eax, [ebp-0x4]
mov     [ebp-0x8], eax
mov     eax, [ebp-0x8]
mov     esp, ebp
pop     ebp
```

Functia ProcessInt suprascrie pe stiva valorile precedente (inclusiv pe cea a lui redI care a fost returnata de GetPtr). Rezultatul e ca in final valoarea lui (*a) va fi alta decat suma primelor 100 de numere + 12

- ▶ 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 modificatorul 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 corect si se compileaza.
- ▶ Din afara clasei se apeleaza doar metoda SetVarsta care este publica in clasa Person
- ▶ Orice metoda definita intr-o clasa (fie publica sai privat) 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:
        const 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:
        const 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:
        const 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:
        const 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:
        const 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:
        const 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:
        const char *Name;
        static int Y;
}
int Person::X;
int Person::Y = 10;

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

- ▶ Codul nu compileaza pentru ca X este membru privat
- ▶ Este nevoie sa cream 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 compilează → SetVarsta este metoda publică 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 declarat in aceeași clasă indiferent de specificatorul de acces al acelui membru
- ▶ In acest exemplu, funcția Increment adaugă 1 la membrul static Y din clasa 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 probleme. 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 compileaza pentru ca 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 modifierul 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();
}
```

- ▶ 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 o clasa → 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);
}
```

Constructori

- 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

- Codul NU compileaza. Constructorii trebuie sa initializeze valorile constant folosind lista de initializari si nu cod implicit. Acest lucru e necesar pentru a garanta ca accesul la y in acest caz se face doar dupa ce a fost initializat.

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

- Codul compileaza corect. Initializarea lui y se face la declaratie. Initializarea din declaratie functioneaza doar de la C++11 (standard) incoace.

App.cpp

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

Constructori

- ▶ Constructorii sunt utilizati pentru a crea 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
}
```


Tipuri de constructori

Constructorii:

- ▶ Implicit (constructorul fara nici un parametru)
- ▶ De copiere (copy constructor)
- ▶ De mutare (move constructor)

Constructorul de copiere

- Constructorul de copiere e un constructor care are ca si parametru o referință constantă sau nu la tipul de date din care face parte constructorul

App.cpp

```
class Date
{
public:
    Date(const Date & d);
};
```

Constructor de
copiere

- De obicei este utilizat in urmatoarele forme de initializare:

App.cpp

```
class Date
{
    int value;
public:
    Date(const Date &d) { value = d.value; }
    Date(int v) { value = v; }
};

int main()
{
    Date d(1);
    Date d2 = d;
    return 0;
}
```

Constructor de
copiere

Constructorul de copiere

- Constructorul de copiere e un constructor care are ca si parametru o referință constantă la tipul de date din care face parte constructorul

App.cpp

```
class Date
{
public:
    Date(const Date & d);
};
```

Constructor de
copiere

- De obicei este utilizat in urmatoarele forme de initializare:

App.cpp

```
class Date
{
    int value;
public:
    Date(const Date &d) {
    Date(int v) { value =
};
int main()
{
    Date d(1);
    Date d2 = d;
    return 0;
}
```

lea eax,[d]
push eax
lea ecx,[d2]
call Date::Date

Constructorul de copiere

- Constructorul de copiere mai e folosit si cand apelam o functie care are un parametru de tipul clasei in care a fost definit constructorul.

App.cpp

```
class Date
{
    int x,y,z,t;
public:
    Date(const Date &d) { x = d.x; y = d.y; z = d.z; t = d.y; }
    Date(int v) { x = y = z = t = v; }
};

void Process(Date d) { ... }

int main()
{
    Date d(1);
    Process(d);
    return 0;
}
```

In acest caz se face o copie a obiectului 'd' si acea copie este trimisa functiei Process

Constructorul de copiere

- In mod similar, daca o functie returneaza un obiect care are un constructor de copiere, acest din urma va fi utilizat pentru a copia continutul unui obiect definit local intr-un obiect temporar utilizat pentru a returna rezultatul.

App.cpp

```
class Date
{
    int x,y,z,t;
public:
    Date(const Date &d) { x = d.x; y = d.y; z = d.z; t = d.y; }
    Date(int v) { x = y = z = t = v; }
};

Date Process()
{
    Date d(1);
    return d;
}
```

In acest punct se apeleaza constructorul de copiere. Mai multe despre cum functioneaza aceste operatii in cursul urmator.

Constructorul de copiere

- Atenție la modul de declarație a unui constructor de copiere. Acesta poate fi declarat in doua feluri:

App.cpp

```
class Date
{
    int x;
public:
    Date(const Date &d) { x = d.x; }
    Date(Date &d) { x = d.x * 2; }
    Date(int v) { x = v; }
};
```

Ambele constructii reprezinta
un constructor de copiere

- Unele compilatoare produc un warning in aceste cazuri.
- Este recomandat sa se folosească varianta cu **const** care este mult mai generică.

Constructorul de copiere

- In acest caz se foloseste constructorul de copiere care nu este **const** (pentru ca si obiectul trimis ca si parametru nu este const).

App.cpp

```
class Date
{
    int x;
public:
    Date(const Date &d) { x = d.x; }
    Date(Date &d) { x = d.x * 2; }
    Date(int v) { x = v; }
};

int main()
{
    Date d(1);
    Date d2 = d;
    return 0;
}
```

Constructorul de copiere

- Daca constructorul de copiere care nu are parametru const nu ar exista, compilatorul ar alege varianta cu const.

App.cpp

```
class Date
{
    int x;
public:
    Date(const Date &d) { x = d.x; }

    Date(int v) { x = v; }
};

int main()
{
    Date d(1);
    Date d2 = d;
    return 0;
}
```


Constructorul de copiere

- In acest caz, `d` este `const` asa ca se va apela constructorul de copiere specific obiectelor constant.

App.cpp

```
class Date
{
    int x;
public:
    Date(const Date &d) { x = d.x; }
    Date(Date &d) { x = d.x * 2; }
    Date(int v) { x = v; }
};

int main()
{
    const Date d(1);

    Date d2 = d;

    return 0;
}
```

Constructorul de copiere

- In acest caz, codul nu compileaza pentru ca EXISTA un constructor de copiere, dar acel constructor nu accepta un parametru const.

App.cpp

```
class Date
{
    int x;
public:
    Date(Date &d) { x = d.x * 2; }
    Date(int v) { x = v; }
};

int main()
{
    const Date d(1);
    Date d2 = d;
    return 0;
}
```

Constructorul de copiere

- Compileaza. Compilatorul va genera un cod care copie octet cu octet datele din **d** in **d2** pentru ca nu exista nici un constructor de copiere.

App.cpp

```
class Date
{
    int x;
public:

    Date(int v) { x = v; }
};

int main()
{
    const Date d(1);
    Date d2 = d;
    return 0;
}
```

Constructorul pentru mutare

- Fie urmatorul cod:

App.cpp

```
class Date
{
    char * sir;
public:
    Date(const Date &d) {
        sir = strdup(d.sir);
        printf("COPY-CTOR: Copy sir from %p to %p \n", d.sir, sir);
    }

    Date(const char * tmp)
    {
        sir = strdup(tmp);
        printf("CTOR: Copy sir from %p to %p \n", tmp, sir);
    }
};

Date Get(Date d)
{
    return d;
}

int main()
{
    Date d = Get(Date("test"));
    return 0;
}
```

Constructorul pentru mutare

- Fie urmatorul cod:

App.cpp

```
class Date
{
    char * sir;
public:
    Date(const Date &d) {
        sir = strdup(d.sir);
        printf("COPY-CTOR: Copy sir from %p to %p\n", d.sir, sir);
    }

    Date(const char * tmp)
    {
        sir = strdup(tmp);
        printf("CTOR: Copy sir from %p to %p \n", tmp, sir);
    }
};

Date Get(Date d)
{
    return d;
}

int main()
{
    Date d = Get(Date("test"));
    return 0;
}
```

In realitate sirul "test" este copiat de 2 ori chiar daca in realitate copierea se face pentru un obiect temporar.

CTOR: Copy sir from 00F8689C to 00AB0610
COPY-CTOR: Copy sir from 00AB0610 to 00AB0648

Constructorul pentru mutare

- Fie urmatorul cod:

App.cpp

```
class Date
{
    char * sir;
public:
    Date(const Date &d) {
        sir = strdup(d.sir);
        printf("COPY-CTOR: Copy sir from %p to %p \n", d.sir, sir);
    }
    Date(const Date &&d) { printf("Move from %p to %p \n", &d, this); sir = d.sir; d.sir = NULL; }
    Date(const char * tmp)
    {
        sir = strdup(tmp);
        printf("CTOR: Copy sir from %p to %p \n", tmp, sir);
    }
};

Date Get(Date d)
{
    return d;
}

int main()
{
    Date d = Get(Date("test"));
    return 0;
}
```

Consstructor pentru mutare

Constructorul pentru mutare

- Fie urmatorul cod:

App.cpp

```
class Date
{
    char * sir;
public:
    Date(const Date &d) {
        sir = strdup(d.sir);
        printf("COPY-CTOR: Copy sir from %p to %p \n", d.sir, sir);
    }
    Date(const Date &&d) { printf("Move from %p to %p \n", &d, this);sir = d.sir;d.sir = NULL; }
    Date(const char * tmp)
    {
        sir = strdup(tmp);
        printf("CTOR: Copy sir from %p to %p \n", tmp, sir);
    }
};

Date Get(Date d)
{
    return d;
}

int main()
{
    Date d = Get(Date("test"));
    return 0;
}
```

Consstructor pentru mutare

CTOR: Copy sir from 00E36870 to 00FB0610
Move from 00D8FD24 to 00D8FE04

Constrangeri

- Fie urmatorul cod:

App.cpp

```
class Date
{
public:
    static int Suma(int x, int y) { return x + y; }
    static int Dif(int x, int y) { return x + y; }
    static int Mul(int x, int y) { return x * y; }
};

int main()
{
    Date d;
    printf("%d\n", Date::Suma(10, 20));
}
```

- Ce am putea face ca sa nu permitem initializarea unei instante a clasei Date ? (in acest context clasa Date este clasa care contine doar metode statice)

Constrangeri

- Solutia 1 - facem constructorul default privat.

App.cpp

```
class Date
{
    Date();
public:
    static int Suma(int x, int y) { return x + y; }
    static int Dif(int x, int y) { return x + y; }
    static int Mul(int x, int y) { return x * y; }
};

int main()
{
    Date d;
    printf("%d\n", Date::Suma(10, 20));
}
```

- Codul nu mai compileaza - insa in continuare se pot crea instante ale clasei Date doar ca din interiorul clasei.

Constrangeri

- Solutia 2 - utilizam cuvantul cheie **delete**

App.cpp

```
class Date
{
public:
    Date() = delete;
    static int Suma(int x, int y) { return x + y; }
    static int Dif(int x, int y) { return x + y; }
    static int Mul(int x, int y) { return x * y; }
};

int main()
{
    Date d;
    printf("%d\n", Date::Suma(10, 20));
}
```

- In aceasta situatie ii spunem explicit compilatorului ca nu exista un constructor implit si ca nu poate crea o instanta a clasei Date in acest mod.

Constrangeri

- Fie urmatorul cod:

App.cpp

```
class Date
{
    int value;
public:
    Date(int x) { value = x; }
};

int main()
{
    Date d('0');
    return 0;
}
```

- Codul functioneaza, doar ca utilizarea este incorecta. Daca cineva da parametru '0' (caracterul ASCII '0') si se asteapta ca valoarea lui x din clasa Date sa devina 0 (numarul 0), atunci acest cod nu va functiona corect (valoarea lui x va fi 48 → codul ASCII a caracterului 0).
- Acest lucru e posibil implicit din cauza promovarii !

Constrangeri

- Solutia in acest caz este similara:

App.cpp

```
class Date
{
    int value;
public:
    Date(char x) = delete;
    Date(int x) { value = x; }
};

int main()
{
    Date d('0');
    return 0;
}
```

- Codul nu compileaza. In acest caz ii spunem implicit compilatorului ca nu poate crea o instanta a unui obiect de tipul Date folosind un parametru de tip char.

Constructor pentru conversii

- Fie urmatorul cod:

App.cpp

```
class Date
{
    int value;
public:
    Date(int v) { value = v; }
};

int main()
{
    Date d = 100;
    return 0;
}
```

push 64h
lea ecx,[d]
call Date::Date

- Codul compileaza. In realitate compilatorul apeleaza in spate constructorul cu un parametru si ii paseaza valoarea 100 pentru initializare.

Constructor pentru conversii

- In mod similar:

App.cpp

```
class Date
{
    int value;
public:
    Date(int v1, int v2, int v3) { value = v1+v2+v3; }
};

int main()
{
    Date d = { 1, 2, 3 };
    return 0;
}
```

Assembly instructions for the initialization of object 'd':

push	3
push	2
push	1
lea	ecx, [d]
call	Date::Date

- Codul compileaza. Obiectul “d” este creat utilizat liste de initializare {...}.
- Valabil de la C++11 incoace.
- Cum putem forta compilatorul sa nu accepte astfel de initializari pentru un obiect de tipul Date ?

Constructor pentru conversii

- In mod similar:

App.cpp

```
class Date
{
    int value;
public:
    explicit Date(int v1, int v2, int v3) { value = v1+v2+v3; }
};
int main()
{
    Date d = { 1, 2, 3 };
    return 0;
}
```

- Solutia e sa folosim cuvantul cheie **explicit**. In acest fel ii spunem compilatorului ca poate folosi Date(int,int,int) doar ca un constructor normal si nu si folosind liste de initializare !
- Codul de mai sus nu compileaza. Daca insa inlocuim cu “Date d(1, 2, 3);” va compila !!!

- ▶ 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);
}
```