Programare Orientată Obiect

ZURINI MĂDĂLINA

OBIECTIVE

Obiectivul principal:

- Prezentarea noțiunilor aferente paradigmei Orientate Obiect în limbajul C++

Obiective secundare:

- Modelarea obientată obiect
- Editarea și compilarea codului sursă C++
- Depanarea și rezolvarea erorilor de compilare și de execuție

EVALUARE

60% EXAMEN FINAL

- examen oral – subiect extras și rezolvat la calculator

40% ACTIVITATE SEMINAR

- 20% test la seminar
- 10% participare activă
- 10% test grilă (S14)

Condiții întrare în examen în sesiunea din iarnă: Obținerea a minim **1.5 pct** din cele 4 aferente activității de seminar

Condiții promovare disciplină: Minim 3 pct din 6 la examenul final și minim 4.5 pct din 10 per total la nivel de disciplină

BIBLIOGRAFIE

- Ion Smeureanu – “Programarea în limbajul C/C++”, Editura CISON, 2001
- Ion Smeureanu, Marian Dardala – “Programarea orientată obiect în limbajul C++”, Editura CISON, 2002
- acs.ase.ro/cpp

STRUCTURĂ CURS

- Recapitulare (Structuri, pointeri, masive, transmiterea parametrilor, tipuri de erori)
- Clase (Definire, atribute, metode, constructori, operatori)
- Fișiere (Lucrul cu stream-uri, fișiere standard, text și binare)
- Derivare (Ierahii de clase, polimorfism, funcții virtuale)
- STL (Clase template, Standard Template Library)

DE CE ORIENTAT OBIECT?

I. ABSTRACTIZARE
   - Modelarea mai ușoară a elementelor din lumea reală

II. INCAPSULARE
   - Ascunderea complexității/informațiilor/comportamentelor de alte entități din exterior

III. DERIVARE
   - Crearea de entități noi pe baza unora deja existente

IV. POLIMORFISM
   - Comportamente multiple în funcție de context

DE CE C++?

- Foarte rapid
- Top 5 limbaje la nivel mondial de peste 30 de ani (https://www.tiobe.com/tiobe-index/)
- Încorporează toate conceptele POO într-un mod pur
- Are sintaxa limbajului C

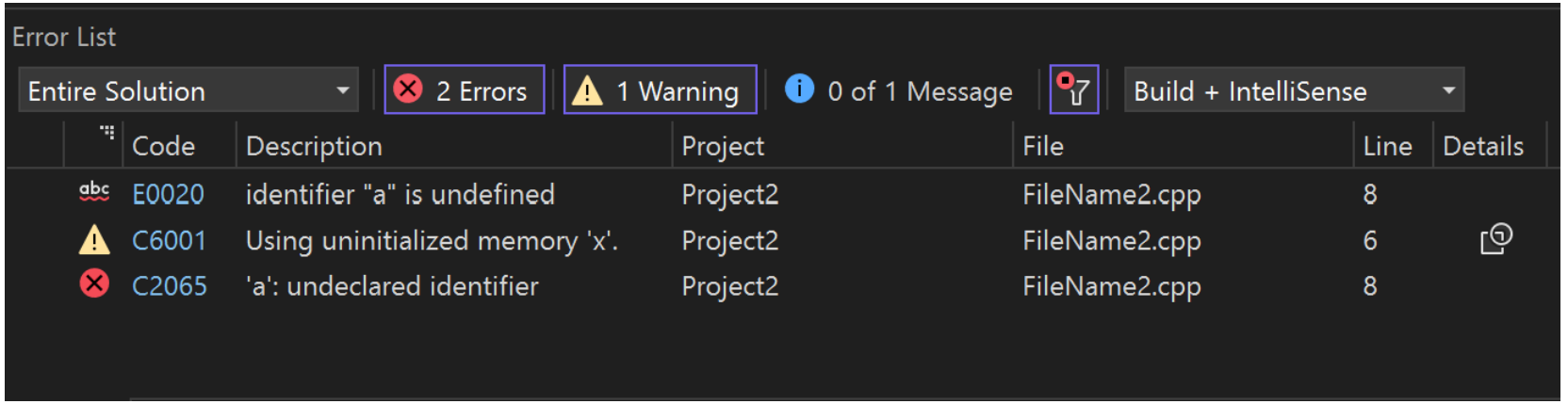
MEDIU DE DEZVOLTARE

- Microsoft Visual Studio 2022 Community la curs și laborator
- **Atenție!** Chiar dacă paradigma este aceeași, diferite compilatoare de C++ pot returna rezultate diferite pentru același cod sursă

RECAPITULARE C01+C02

TIPURI DE ERORI

ERORI DE COMPILARE



ERORI DE EXECUȚIE

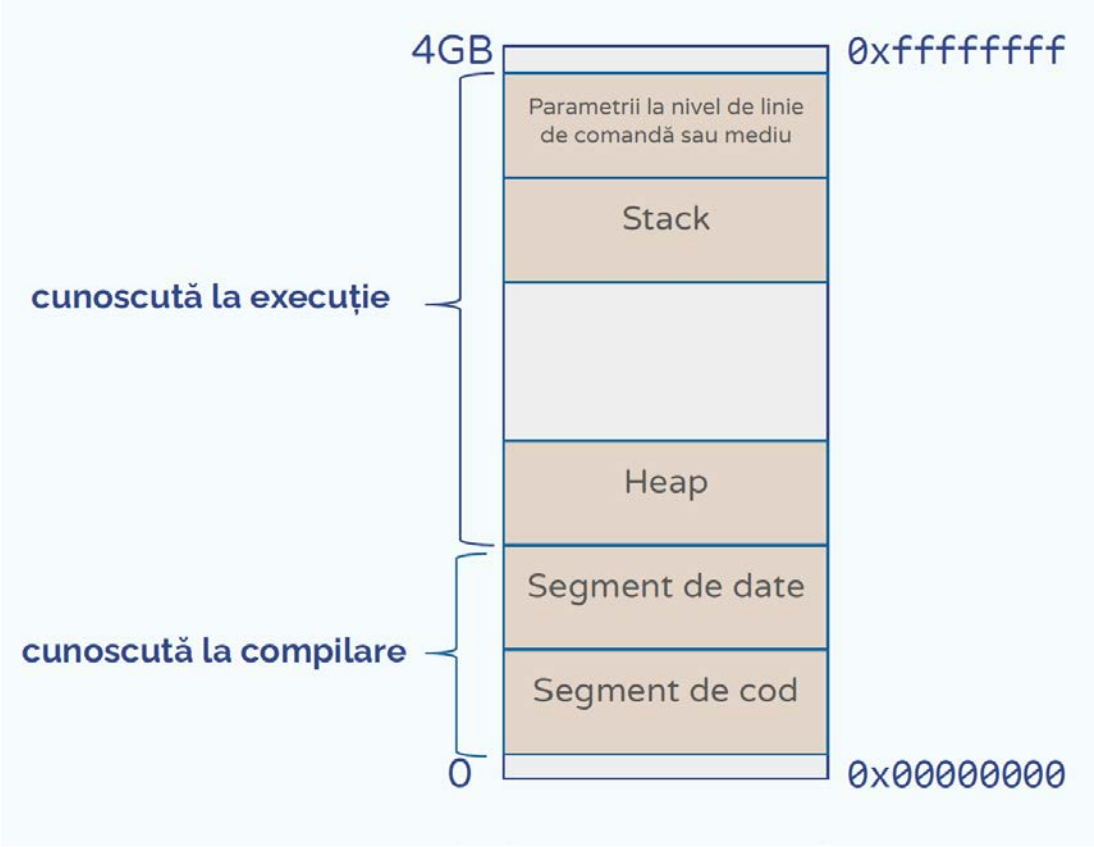
```
D:\Proiecte Visual\Project2\Debug\Project2.exe (process 25624) exited with code -1073741676.
Press any key to close this window . . .
```

NEXT TIME

Continuare recapitulare

a. Pointeri
b. Alocare statică și dinamică
c. Tipuri de memorie
d. Variabile constante
e. Masive uni și bidimensionale
f. Șiruri de caractere
g. Transmiterea parametrilor
h. Directive de preprocesare

MEMORIA


4GB
0xffffffff
Parametrii la nivel de linie de comandă sau mediu
Stack
cunoscută la execuție
Heap
Segment de date
cunoscută la compilare
Segment de cod
0
0x00000000

POINTERI

- Variabile responsabile cu salvarea unor adrese de memorie
- sunt salvați în stack și pointează către zone din heap (acesta este cel mai întâlnit scenario)
- Ocupă 4 bytes indiferent de tipul de pointer (ce se regăsește la adresa de
- memorie către care pointează) pentru un procesor/compilator pe 32 de biți

DEFINIRE

- la definire de obicei valoarea este una arbitrara
- nu este recomandat ca un pointer sa ramana neinitializat

tip_data* nume_pointer;

tip_data* nume_pointer = nullptr;

INITIALIZARE

- pointerul poate salva doar adrese unde se gaseste tipul de data specificat
- operatorul & extrage adresa de memorie a unei variabile

tip_data nume_variabila = valoare;

tip_data* nume_pointer = &nume_variabila;

UTILIZARE

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

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

INCREMENTARE/DECREMENTARE

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

IDENTIFICATORUL CONST LA POINTERI

1. pointeri constanți: odată salvată o adresă, nu mai poate fi modificată
2. pointeri la o zonă de memorie constantă: valoarea de la adresa către care pointează nu poate fi modificată prin intermediul lor
3. pointeri constanți la o zonă de memorie constantă: combinație între cele două situații de mai sus

IDENTIFICATORUL CONST LA POINTERI

```
int a;
const int* px1;
int* const px2 = &a;
const int* const px3 = &a;
int const* px4;
```

MASIVE UNIDIMENSIONALE

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

MASIVE BIDIMENSIONALE

- Masive bidimensionale ce permit accesul elementelor pe linii și coloane
- Memoria fiind liniară (unidimensională) pentru a putea fi salvate sunt liniarizate
- Cele alocate static sunt salvate în memorie ca vectori consecutivi ce conțin elementele de pe fiecare linie
- Cele alocate dinamic sunt salvate ca vectori de vectori (un vector ce conține adresele de memorie ale fiecărui vector corespunzător fiecărei linii)

OPERATORUL [] SI POINTERI

- vector[i] ⇔ *(vector + i)
- matrice[i][j] ⇔ *(*(matrice + i) + j)

VECTORI DE CARACTERE

- În limbajul C nu există un tip de dată special definit pentru șiruri de caractere
- Pentru a salva totuși astfel de șiruri se folosește o convenție: se utilizează un vector de caractere (vector de char) ce are drept ultim element ‘\0’ (cod ASCII 0) pentru a ști când vectorul de caractere se sfârșește
- Vectorii de caractere pot fi prelucrați ca orice alt vector sau, profitând de faptul că putem ști când ajungem la ultimul element, prin utilizarea de funcții specifice

SIRURI DE CARACTERE - STRING

- În limbajul C++ avem tip de dată special definit pentru șiruri de caractere - string
- string este o clasă în C++ așadar prelucrările se fac în mod direct prin operatori sau metode

CLASE C03

ENUMERARI

- Folosite atunci cand o data ia valori doar intr-un interval finit dat
- La compilare sunt transformate in constante
- Ajuta la o intelegere mai usoara a codului

```
enum formaStudiu {
ZI, ID, IDD };
  formaStudiu f = formaStudiu::ZI;
```

UNIUNI

- Date ce poate salva o singura valoare la un moment dat dintr-un set de valori disponibile
- Ocupa zona de memorie egal cu maximul dintre valorile posibile

```
union id {
    int cui;
    long cnp;
};
```

```
id idClient;
idClient.cnp = 1674546;
cout << idClient.cnp << " " << idClient.cui;
idClient.cui = 34567;
cout << endl << idClient.cnp << " " << idClient.cui;
```

STRUCTURI

- Permite crearea unui tip de data complex ce grupeaza mai multe caracteristici
- Inceputul conceptului de incapsulare
- In C, permite doar existenta atributelor

```
struct Student {
    string nume;
    int grupa;
    char serie;
};
```

```
Student s;
s.nume = "Gigel";
s.grupa = 1056;
s.serie = 's';
```

CLASE

- Asemanatoare structurilor, incapsuleaza caracteristici si comportamente
- Baza POO

```
class Masina {
    //atribute si metode
};
```

ATRIBUTE

- modeleaza caracteristicile, starea unui obiect

```
class Masina {
    int anProductie;
    string marca;
    string culoare;
    //....
};
```

METODE

- modeleaza comportamentul unui obiect

```
class Masina {
    int anProductie;
    string marca;
    string culoare;
    int kmParcursi;
    //....

    void adaugaKm(int nrKm) {
        kmParcursi += nrKm;
    }

};
```

OBIECTE

- instante ale claselor (variabile de tipul clasei)

```
Masina m;
m.culoare = "Galbena";
m.kmParcursi = 1500;
m.adaugaKm(100);
cout << endl << m.kmParcursi;
```

DOMENII DE VIZIBILITATE

- PRIVATE (tot ce este definit in aceasta zona poate fi accesat doar din interiorul clasei, deci nu poate fi accesat din exteriorul ei)
- PROTECTED (poate fi accesat din interiorul clasei precum si din clasele derivate din ea)
- PUBLIC (poate fi accesat de oriunde)

UTILIZARE DOMENII DE VIZIBILITATE

```
class Masina {

private:
    int anProductie;
    string marca;

public:
    string culoare;

    void adaugaKm(int nrKm) {
        kmParcursi += nrKm;
    }

protected:
    int kmParcursi;
};
```

CLASE VS STRUCTURI

CLASA	STRUCTURA
Membrii clasei sunt implicit privati	Membrii structurii sunt implicit publici
Se declara folosind cuvantul cheie class	Se declara folosind cuvantul cheie struct
Se foloseste de regula pentru abstractizare si mostenire	Se foloseste pentru a grupa diferite tipuri de date

CLASE C04

CONSTRUCTORI

- Metode cu rolul de creare de noi obiecte prin alocarea de zona de memorie si initializare a atributelor
- Particularitati implementare:
  - Metode care poarta numele clasei
  - Metode care nu intorc nimic (nu au tip returnat)

CONSTRUCTORI

- Daca nu exista implementari de constructori, este creat automat un constructor implicit fara parametri. Daca orice forma de constructor este implementat in clasa, atunci constructorul implicit fara parametri nu mai poate fi folosit, obligand implementarea si a constructorului fara parametri in caz de nevoie
- Pot exista oricati constructori in cadrul unei clase, insa doar o singura forma este folosita la crearea unui anume obiect in functie de forma apelata la declarare

CONSTRUCTOR IMPLICIT

```
class Student {
    string nume;
    string facultate;
    int anStudiu;
    int nrNote;
    int* note;

public:
    Student() {
        nume = "";
        facultate = "";
        anStudiu = 0;
        nrNote = 0;
        note = nullptr;
    }
};
```

```
int main() {
    Student s;
    Student* ps = new Student();
    return 0;
}
```

CONSTRUCTOR CU PARAMETRI

```
Student(string _nume, string _facultate, int _anStudiu, int _nrNote, int* _note) {
    nume = _nume;
    facultate = _facultate;
    anStudiu = _anStudiu;
    nrNote = _nrNote;
    note = new int[nrNote];
    for (int i = 0; i < nrNote; i++)
        note[i] = _note[i];
}
```

```
Student s2("Gigel", "CSIE", 2, 3, new int[3]{10, 6, 8});// e okai?
int* note = new int[3] { 10,6,8 };
Student s3("Costel", "CSIE", 2, 3, note);
```

CONSTRUCTOR COMBINAT

```
Student(string _nume="", string _facultate="", int _anStudiu=0, int _nrNote=0, int* _note=nullptr) {
    nume = _nume;
    facultate = _facultate;
    anStudiu = _anStudiu;
    nrNote = _nrNote;
    note = new int[nrNote];
    for (int i = 0; i < nrNote; i++)
        note[i] = _note[i];
}
```

```
Student s4("Marcel", "CSIE", 1, 3, note);
Student s5("Marcel", "CSIE", 1, 3); //??? este logic?
Student s6("Marcel", "CSIE", 1);
Student s7("Marcel", "CSIE");
Student s8("Marcel");
Student s9;
```

LISTA INITIALIZATORI

- Apel alte forme de constructor (din propria clasa sau din altele)

```
Student(string _nume, string _facultate, int _anStudiu) :Student() {
    nume = _nume;
    facultate = _facultate;
    anStudiu = _anStudiu;
}
```

```
Student(string _nume) :nume(_nume) {
    facultate = "";
    anStudiu = 0;
    nrNote = 0;
    note = nullptr;
}
```

CONSTRUCTOR CU 1 PARAMETRU

```
Student(string _nume) :nume(_nume) {
    facultate = "";
    anStudiu = 0;
    nrNote = 0;
    note = nullptr;
}
```

```
string nume = "Marcel";
Student s10 = nume;
```

! Aici nu este declarare/construire + atribuire, ci este construire prin atribuire

CONSTRUCTOR DE COPIERE

- Folosit pentru crearea de copii ale unor obiecte
- Situatii implicite de creare copii de obiecte:
  - Transmiterea unui obiect prin valoare
  - Returnarea unui obiect prin valoare
- Exista o forma implicita de constructor de copiere care aloca zona de memorie si copiaza octet cu octet => Ce facem cand avem membri de tip pointer?

SHALLOW COPY VS DEEP COPY

CONSTRUCTOR DE COPIERE

```
Student(const Student& s) {
    nume = s.nume;
    facultate = s.facultate;
    anStudiu = s.anStudiu;
    //SHALLOW COPY
    //note = s.note;

    //DEEP COPY
    nrNote = s.nrNote;
    note = new int[nrNote];
    for (int i = 0; i < nrNote; i++)
        note[i] = s.note[i];
}
```

```
Student f(Student s) {
    Student s2;
    //...
    return s2;
}
```

```
Student s11 = s10;
Student s12(s11);
```

DESTRUCTOR

- Metoda speciala care poarta acelasi nume cu clasa preceadat de caracterul ~
- Exista o singura forma de destructor
- Are rolul de a elibera zona de memorie ocupata de obiect prin distrugerea sa
- Exista o forma implicita de destructor dar care dezaloca zona de memorie doar din heap (adica cea alocata de acel constructor implicit al clasei)

DESTRUCTOR

```
~Student() {
    if (note != nullptr) {
        delete[] note;
        note = nullptr;
        nrNote = 0;//???
    }
}
```

```
Student* ps2 = new Student();
delete ps2;

{
    Student s13;
}

Student* vs = new Student[5];
delete[] vs;
```

METODE ACCESOR (GET SI SET)

- Metode speciale cu rol de interfata pentru zona private a unei clase
- Rol de consultare (get) si modificare (set)
- Sunt definite in zona public si ofera o interactiune controlata asupra membrilor private
  - read-only
  - validari

METODE ACCESOR

```
string getNume() {
    return nume;
}

void setNume(string _nume) {
    if (_nume.size() >= 5) {
        nume = _nume;
    }
}
```

```
s.setNume("Costel");
cout << s.getNume();
```

POINTERUL THIS

- In cadrul metodelor non-statice ale unei clase, pointerul this refera adresa obiectului apelator

```
void promovare() {
    this->anStudiu++;
}
```

```
int main() {
    Student s;

    s.promovare();

    return 0;
}
```

CLASE C05

METODE INLINE

- Metodele unei clase pot fi scrise in interiorul clasei in totalitate sau doar semnatura lor in clasa iar corpul in exteriorul clasei folosind operatorul de rezolutie ::
- Metodele scrise in interiorul clasei sunt considerate metode inline
- Metodele scrise in afara clasei pot fi transformate in metode inline folosind cuvantul cheie **inline** (depinde de compilator)

EXEMPLU

```
class Produs {
    string denumire;
    float pret;
    int cantitate;

public:
    float getPret() {
        return this->pret;
    }

    void reducerePret(float discount);
};

inline void Produs::reducerePret(float discount) {
    this->pret = this->pret * (1 - discount);
}

int main() {
    Produs p;
    p.reducerePret(0.2);
    return 0;
}
```

MEMBRI CONSTANTI

- Atribut pentru care nu dorim sa se modifice valoarea sa odata ce a fost initializat

```
class Produs {
    string denumire;
    float pret;
    int cantitate;

public:
    const int id = 1234;
};



int main() {
    Produs p;
    //p.id = 23; eroare de compilare
    return 0;
}
```

EXEMPLU UTILIZARE ATRIBUT CONSTANT

```
class Produs {
    string denumire;
    float pret;

public:
    const int id;//poate fi declarat in zona public

    Produs(int _id, string _denumire, float _pret):id(_id) {
        this->denumire = _denumire;
        this->pret = _pret;
    }

};
```

MEMBRI STATICI

- Un atribut care este comun tuturor obiectelor clasei sau o metoda care nu refera un anume obiect al clasei

```
class Student {
    string nume;
    static float taxaRestanta;
};

float Student::taxaRestanta = 100;
```

ATRIBUTE STATICE

- Se declara folosind cuvantul cheie **static**
- Sunt atribute ale clasei si nu ale unui obiect
- Au o valoare comuna pentru toate obiectele clasei
- Se initializeaza in afara clasei

METODE STATICE

- La definire se foloseste cuvantul cheie static
- Sunt metode care refera clasa si nu un anume obiect
- Nu primesc pointerul this
- Prelucreaza doar atributele statice ale unei clase

EXEMPLU MEMBRI STATICI

```
class Student {
    string nume;
    static float taxaRestanta;

public:
    static float getTaxaRestanta() {
        return Student::taxaRestanta;
    }
};

float Student::taxaRestanta = 100;



int main() {
    Student s;
    cout << s.getTaxaRestanta();
    cout << endl << Student::getTaxaRestanta();
    return 0;
}
```

IMPLICATII EXISTENTA MEMBRU POINTER

- Realizeaza extensii ale clasei in zona de memorie heap
- Pentru vectori numerici este necesara definirea unui atribut ce va reprezenta numarul de elemente ale vectorului (in cele mai multe cazuri)
- Cand in clasa exista un membru non-static pointer atunci este obligatorie definirea explicita a urmatoarelor metode:
  - Constructor de copier
  - Operator de atribuire (=)
  - Destructor

IMPLICATII EXISTENTA MEMBRU POINTER

- Copierea unui membru pointer se va face intr-o zona de memorie noua (deep copy)
- Folosirea operatorului de atribuire(=) intre adrese doar va realiza o copiere superficiala, astfel ambele obiecte vor partaja aceeasi zona de memorie
- Getter-ul membrului pointer NU trebuie sa returneze pointerul/adresa obiectului, ci trebuie sa faca o copie
- Setter-ul pentru membru tip pointer vector numeric primeste 2 parametri (noul vector reprezentat de adresa si noul numar de elemente)

OPERATORUL DE ATRIBUIRE (OPERATOR=)

- Apelat atunci cand se copiaza informatiile unui obiect existent in alt obiect existent
- Returneaza void (atunci cand nu este permis apelul in cascada) sau adresa obiectului creat (pentru a permite apelul in cascada)
- Precum constructorul de copiere, primeste ca parametru obiectul din care se face copia

```
Student& operator=(const Student& s) {

}
```

EXEMPLU OPERATOR=

```
class Student {
    string nume;
    int* note;
    int nrNote;

public:
    Student& operator=(const Student& s) {
        if (this != &s) {
            if (this->note != nullptr) {
                delete[] this->note;
                this->note = nullptr;
            }
            this->nume = s.nume;
            if (s.nrNote > 0 && s.note != nullptr) {
                this->nrNote = s.nrNote;
                this->note = new int[this->nrNote];
                for (int i = 0; i < this->nrNote; i++)
                    this->note[i] = s.note[i];
            }
            else {
                this->nrNote = 0;
                this->note = nullptr;
            }
        }
        return *this;
    }
};
```

CLASE C06

Metode accessor (get si set)

```
const int* getNote() {
    return this->note;
}

int getNrNote() {
    return this->nrNote;
}

int getNota(int index) {
    if (index >= 0 && index < this->nrNote)
        return this->note[index];
}
```

```
void setNota(int _index, int _nota) {
    if (_nota >= 1 && _nota <= 10) {
        if (_index >= 0 && _index < this->nrNote) {
            this->note[_index] = _nota;
        }
    }
}

//nu putem actualiza vectorul daca nu actualizam si noua dimensiune
void setNote(int* _note, int _nrNote) {
    if (_nrNote > 0 && _note != nullptr) {
        //se mai pot adauga validari pentru fiecare elem din vector (daca este nota)
        if (this->note != nullptr) {
            delete[] this->note;
            this->note = nullptr;
        }
        this->nrNote = _nrNote;
        this->note = new int[this->nrNote];
        for (int i = 0; i < this->nrNote; i++)
            this->note[i] = _note[i];
    }
}
```

SUPRAINCARCAREA FUNCTIILOR

- Polimorfism in POO
- Cel putin doua functii care au acelasi nume dar difera prin numarul si tipul parametrilor
- Identificarea functiei se face la compilare
- Conceptul de early binding sau polimorfism slab

EXEMPLU

```
float calculMedie() {
    int suma = 0;
    for (int i = 0; i < this->nrNote; i++)
        suma += this->note[i];
    if (this->nrNote > 0)
        return (float)suma / this->nrNote;
    return 0;
}

float calculMedie(int _index1, int _index2) {
    if (_index1 >= 0 && _index1 <= _index2 && _index2 < this->nrNote) {
        int suma = 0;
        for (int i = _index1; i < _index2; i++)
            suma += this->note[i];
        if (this->nrNote > 0)
            return (float)suma / this->nrNote;
        return 0;
    }
    return 0;
}
```

ETAPE ALEGEREA CORECTA A FUNCTIEI

1. Se cauta implementarea functiei care are exact aceeasi lista de parametri
2. Se cauta implementarea functiei realizand conversii nedegradante (int la long, char la int)
3. Se cauta implementarea functiei aplicand conversii degradante (float la int)
4. Se cauta implementarea functiei aplicand conversii implementate de programator

Reguli:

- Daca la oricare din pasii anteriori se gasesc mai multe functii care corespund, rezulta eroare de ambiguitate
- Daca nu se gaseste nicio functie parcurgand pasii, rezulta eroare de linkeditare

SUPRAINCARCAREA OPERATORILOR

- Operatorii existenti pot fi specializati sa gestioneze si tipuri de date definite de utilizator
- Acest lucru se realizeaza prin functii avand denumirea **operator<semn_grafic>**
- Supraincarcarea operatorilor pentru ca numere operatorului ramane acelasi si se schimba doar tipul parametrilor

TEHNICI SUPRAINCARCARE OPERATORI

1. **Metoda/functie membra**, atunci cand primul operand este obligatoriu de tipul clasei si va fi transpus in pointerul this
2. **Functie globala**, mai ales atunci cand primul operand nu este de tipul clasei

CONCEPT FRIEND

- Posibilitatea de a acorda acces anumitor functii sau clase asupra tuturor membrilor clasei
- Aceste functii sau clase se numesc prietene (friend) si se anunta folosind formatul:
  - **friend class Clasa;**
  - **friend tip_return functie(lista parametri);**

EXEMPLE

- Clasa Student are acces la toti membrii clasei Facultate
- Metoda afisare din clasa Student are acces la toti membrii clasei Facultate

```
class Facultate {
    string denumire;
    char* adresa;
    string contactSecretariat;

    friend class Student;

    friend void Student::afisare();
};
```

REGULI SUPRAINCARCARE OPERATORI

1. Nu se pot supraincarca decat operatori existenti
2. Unii operatori sunt exclusi de la supraincarcare (ex: . *. :: ?: sizeof())
3. Sensul unui operator se recomanda sa nu se schimbe prin supraincarcare
4. Supraincarcarea nu permite automat compunerea operatorilor
5. Nu se indeplineste comutativitatea implicit prin supraincarcare

OPERATORI UNARI

- Pot si implementati si prin functie membra si prin functie globala
- Daca sunt supraincarcati prin functie membra inseamna ca nu primesc niciun parametru iar prin functie globala primesc un unic parametru (obiectul de tipul clasei)

```
class Masina {
    string marca;
    int kilometraj;
    int nivelRezervor;//0-100

    bool operator!() {
        return this->nivelRezervor != 0;
    }
};
```

```
class Masina {
    string marca;
    int kilometraj;
    int nivelRezervor;//0-100

    friend bool operator!(Masina m);
};

bool operator!(Masina m) {
    return m.nivelRezervor != 0;
}
```

EXEMPLU OPERATORI UNARI

```
class Masina {
    string marca;
    int kilometraj;
    int nivelRezervor;//0-100

public:
    int getNivelRezervor() {
        return this->nivelRezervor;
    }
};

bool operator!(Masina m) {
    return m.getNivelRezervor() != 0;
}
```

```
if (!m)
    cout << "\nMasina m mai are benzina";
else
    cout << "\nMasina m nu mai are benzina";

//forma explicita apel operator! implementat prin functie globala
operator!(m);

//forma explicita apel operator! implementat prin functie membra
m.operator!();
```

ALTI OPERATORI UNARI

- Operatorii unari de pre si post-incrementare sau pre si post-decrementare sunt identificati de compilator prin utilizarea unui parametru de tip int suplimentar la nivelul celor postfixati.
- !!!Acest parametru nu este folosit efectiv, si doar utilizat pentru distictia implementarilor

```
//pre-incrementare
Masina operator++() {
    this->kilometraj++;
    return *this;
}

//post-incrementare
Masina operator++(int) {
    Masina copie = *this;
    this->kilometraj++;
    return copie;
}
```

```
int main() {
    Masina m;

    m++;
    m.operator++(2);

    ++m;
    m.operator++();

    return 0;
}
```

OPERATORI BINARI (Obj + int)

```
//operator+ (Masina + int)
Masina operator+(int _nivelSuplimentar) {
    if (_nivelSuplimentar > 0 && _nivelSuplimentar + this->nivelRezervor <= 100) {
        Masina rez = *this;
        rez.nivelRezervor += _nivelSuplimentar;
        return rez;
    }
}
```

```
m2 = m + 10;
m2 = m.operator+(10);
cout << endl << m2.getNivelRezervor();
```

OPERATORI BINARI (int + Obj)

```
    friend Masina operator+(int _nivelSuplimentar, Masina m);
};

Masina operator+(int _nivelSuplimentar, Masina m) {
    if (_nivelSuplimentar > 0 && _nivelSuplimentar + m.nivelRezervor <= 100) {
        Masina rez = m;
        rez.nivelRezervor += _nivelSuplimentar;
        return rez;
    }
}
```

In clasa!!!

```
Masina operator+(int _nivelSuplimentar, Masina m) {
    return m + _nivelSuplimentar;
}
```

Utilizand comutativitatea adunarii

```
m2 = 10 + m;
m2 = operator+(10, m);
cout << endl << m2.getNivelRezervor();
```

CLASE C07

OPERATORI DE CITIRE SI AFISARE LA CONSOLA

- Operatori binari care sunt supraincarcati printr-o functie globala pentru ca primul operand nu este de tipul clasei (este un obiect istream sau ostream)
- Pot fi declarati friend in clasa pentru a avea acces la membrii private sau protected; insa nu este obligatoriu

EXEMPLIFICARE <<

!!! Nu mai este necesara metoda afisare() folosita pana acum!

```
    friend ostream& operator<<(ostream& out, const Masina& m);
};

ostream& operator<<(ostream& out, const Masina& m) {
    cout << "\n------------------";
    cout << "\nModel: " << m.model;
    cout << "\nNr deplasari: " << m.nrDeplasari;
    cout << "\nDistanta deplasari: ";
    for (int i = 0; i < m.nrDeplasari; i++)
        cout << m.distantaDeplasari[i] << " ";
    cout << "\nPret: " << m.pret;
    cout << "\nAn fabricatie: " << m.anFabricatie;
    return out;
```

```
int main() {
    float deplasari[] = { 100,30,15.5,550 };
    Masina m1("Dacia", 4, deplasari, 12000, 2020);
    m1.afisare();
    cout << m1;
```

EXEMPLIFICARE >>

!!! Obiectul m se modifica in urma citirii, deci se transmite prin referinta fara a-l declara constant.

```
friend istream& operator>>(istream& in, Masina& m) {
    cout << "\nModel: ";
    in >> m.model;
    cout << "Nr deplasari: ";
    int nrDeplasari;
    in >> nrDeplasari;
    if (m.distantaDeplasari != nullptr) {
        delete[] m.distantaDeplasari;
        m.distantaDeplasari = nullptr;
    }
    if (nrDeplasari > 0) {

        m.nrDeplasari = nrDeplasari;
        m.distantaDeplasari = new float[m.nrDeplasari];
        cout << "Deplasari: ";
        for (int i = 0; i < m.nrDeplasari; i++) {
            in >> m.distantaDeplasari[i];
        }
    }
    else {
        m.nrDeplasari = 0;
        m.distantaDeplasari = nullptr;
    }
    cout << "Pret: ";
    in >> m.pret;
    return in;
}
```

OPERATOR INDEX []

- !!! Poate fi supraincarcat doar printr-o functie membra si mai primeste ca parametru, exceptand this, inca o valoara (nu este necesar sa fie numerica).

```
float operator[](int index) {
    if (index >= 0 && index < this->nrDeplasari)
        return this->distantaDeplasari[index];
}
```

```
cout << "\nDeplasarea 0: " << m1[0];
```

OPERATORUL CAST

- Este un operator unar (primeste un singur parametru -> this)
- Numele operatorului este ceea ce returneaza metoda
- Poate fi supraincarcat doar printr-o functie membra clasei
- Poate fi supraincarcat pentru diferite tipuri de date
- Exista o forma explicita sau implicita

EXEMPLIFICARE

```
operator int() {
    return this->anFabricatie;
}

explicit operator string() {
    return this->model;
}
```

```
int anFabricatie = m1;

string model = (string)m1;
```

OPERATOR FUNCTIE

- Supraincarcat doar printr-o functie membra
- Primeste un numar variabil de parametri (primul fiind un obiect de tipul clasei)

```
int operator()(int distantaMinima) {
    int ct = 0;
    for (int i = 0; i < this->nrDeplasari; i++)
        if (this->distantaDeplasari[i] >= distantaMinima)
            ct++;
    return ct;
}
```

```
float deplasari[] = { 100,30,15.5,550 };
Masina m1("Dacia", 4, deplasari, 12000, 2020);

int contor = m1(100); //apel operator functie
int ct = m1.operator ()(100);//apel explicit operator functie
```

CLASE C08

COMPUNEREA CLASELOR

- O clasa are ca membri de tipurile altor clase
- Relatia de compunere sau “has a”
- Este de mai multe tipuri: 1-1 sau 1-M
- Poate modela apartenenta de pointeri sau de obiecte ca membri de tipuri ale altor clase

EXEMPLE RELATII HAS A

- Proprietar has a Masina SAU Masina has a Proprietar???
- Grupa has a lista Studenti
- Angajat has a Contract sau Contract has a Angajat?

CLASE C09

MOSTENIRE

- Permite crearea de clase noi prin reutilizarea codului sursa
- Raspunde la intrebarea is – a
- Reflecta conceptul de specializare
- Clasa initiala poate numele de clasa de baza/parinte iar clasa derivata/specializata/extinsa poarta numele de clasa derivata/copil
- Clasa derivata mosteneste toti membrii clasei de baza

EXEMPLU MOSTENIRE

```
class Persoana {
    string nume;
    //....
};

class Student : public Persoana {
    //...
};
```

TIPURI DE DERIVARE

- PUBLIC
  - Membrii clasei de baza isi pastreaza vizibilitatea
- PROTECTED
  - Membrii publici devin protected in clasa derivata, ceilalti isi pastreaza vizibilitatea
- PRIVATE (implicit)
  - Toti membrii clasei de baza devin private in clasa derivata

PUBLICIZARE

```
class Persoana {
public:
    string nume;
    //....
};

class Student : private Persoana {
    //...
public:
    Persoana::nume;
};

int main() {
    Student s;
    s.nume = "Gigel";
    return 0;
}
```

PROPRIETATI DERIVARE

- Clasa derivata mosteneste toate atributele si metodele clasei de baza
- Chiar daca atributele private nu sunt accesibile in clasa derivata asta nu inseamna ca ele nu se mostenesc si nu formeaza clasa derivata

APEL CONSTRUCTOR CLASA DE BAZA

- Fiecare constructor este responsabil in initializarea partii sale din obiect
- Constructorul din clasa de baza initializeaza zona lui de obiect iar cel din clasa derivata noua zona adaugata obiectului
- Ordinea de apel este data de relatia Baza - Derivat

EXEMPLIFICARE APEL CONSTRUCTORI

```
class Persoana {
public:
    string nume;
    Persoana() { this->nume = "Anonim"; }
};

class Student : private Persoana {
    //...
public:
    string facultate;
    Student(){}
    Student(string nume, string facultate) :Persoana() {
        this->facultate = facultate;
    }
};
```

UPCASTING

- Conversia de la clasa derivata la clasa de baza
- Este realizata implicit prin limitarea in sizeof(clasa baza)
- Se aplica si pentru obiecte cat si pentru pointeri

EXEMPLIFICARE UPCASTING

```
int main() {
    Student s;
    Persoana p;
    p = s;
    Persoana* pp = new Persoana();
    Student* ps = new Student();
    pp = ps;
    return 0;
}
```

REDEFINIRE MEMBRI

- Clasa derivata poate defini membri cu acelasi nume cu clasa de baza
- Pentru a-i diferentia, este necesara utilizarea operatorului de rezolutie ::

```
class Persoana {
public:
    string nume;
};

class Student : public Persoana {
    //...
public:
    string nume;
};

int main() {
    Student s;
    cout << s.nume;
    cout << s.Persoana::nume;
    return 0;
}
```

EXTRA

- Functiile friend din clasa de baza raman friend si in clasa derivata
- Clasele friend din clasa de baza NU raman friend si in clasa derivata
- Operatorul = se mosteneste direct in clasa derivata insa se ocupa doar de parte de obiect din clasa de baza
- Operatorii pot fi redefiniti in clasa derivata

CLASE C10

EXCEPTII

- În C++ avem anumite situații când datele de intrare nu sunt valide
- În aceste situații putem returna excepții din anumite funcții
- Aruncarea unei excepții înlocuiește returnarea unei valori din acea funcție
- Excepțiile, dacă nu sunt gestionate, întrerup funcționarea normală a programului și se comportă precum erorile de execuție
- În C++ o mare parte din funcțiile standard nu folosesc excepții

EXCEPTII EXEMPLU

```
void setVarsta(int _varsta) {
    if (_varsta > 0) {
        this->varsta = _varsta;
    }
    else {
        throw new exception("varsta negativa");
    }
}
```

```
try {
    p1.setVarsta(-5);
    cout << p1;
}
catch (exception* ex) {
    cout << endl<<ex->what();//este un getter pentru mesaj
    delete ex;
}
```

EXCEPTII REGULI

- În C++ putem returna nu doar excepții de tipul exception ci și alte tipuri de date (coduri de eroare de exemplu)
- Pot exista mai multe blocuri de tip catch pentru un bloc try (cel puțin unul este obligatoriu), situație în care vor fi evaluate de sus în jos
- Întotdeauna un singur bloc catch este utilizat în cazul apariției unei excepții

EXCEPTII CUSTOM

- Pot fi definite excepții personalizate prin derivarea clasei **exception**
- Dacă dorim să specificăm și mesaje de eroare, atunci putem să definim un constructor cu un parametru de tip char* pe care să îl trimitem mai departe către constructorul cu un parametru al casei **exception**

```
class ExceptieCustom : public exception {
public:
    ExceptieCustom() {

    }

    ExceptieCustom(const char* mesaj):exception(mesaj){ }
};
```

EXEMPLU CATCH MULTIPLU

```
try {
    p1.setVarsta(-5);
    cout << p1;
}
catch (ExceptieCustom* ex) {
    cout << endl << ex->what();
    delete ex;
}
catch (exception* ex) {
    cout << endl<<ex->what();
    delete ex;
}
//ordinea este importanta, se gestioneaza de la cat mai specific spre cat mai general
```

EXCEPTII - CONCLUZII

- Excepțiile pot încetini execuția unui program și pot produce memory leaks dacă sunt generate de anumite metode (ex: constructor sau destructor)
- Nu sunt o alternativă pentru structura condițională
- Permit o modalitate standard și unitară de gestionare a situațiilor neprevăzute
- Nu sunt utilizate de către toate bibliotecile C++

FISIERE C11

STREAM

- Obiect ce permite gestionarea și/sau manipulare șirurilor de bytes
- Utilizate de obicei în C++ pentru formele supraîncărcate ale operatorilor **<<** și **>>**
- Obiecte standard existente:
  - **cout** - obiect de tip ostream (stream-ul standard de ieșire)
  - **cin** - obiect de tip istream (stream-ul standard de intrare)
  - **cerr** - obiect de tip ostream (asociat stream-ului standard de erori)

LUCRU CU FISIERE C++

- Stream-uri prezente în fișierul header **fstream**
- Denumite și stream-uri/fișiere non-standard
- Lucrul în mod text este asemănător cu cel pe fișiere standard
- Lucrul în mod binar presupune serializarea bit-cu-bit a atributelor clasei

CLASE UTILIZATE

- ifstream - fișiere de intrare (mod citire)
- ofstream - fișiere de ieșire (mod scriere)
- fstream - fișiere de intrare-ieșire (mod citire-scriere)

CITIRE SI SCRIERE FISIER BINAR

- Nu există funcții specifice în C++ pentru a serializa tipuri de date complexe
- Serializarea se face prin utilizarea metodei **write()** ce poate scrie un vector de caractere de lungime fixă
- Deserializarea se face prin utilizarea metodei **read()** ce poate citi un vector de caractere de lungime fixă
- Toate celelalte tipuri de date (cu excepția char*) vor fi convertite prin intermediul operatorului de cast explicit la char* pentru a putea fi scrise/citite binar

EXEMPLU SCRIERE FISIER BINAR

```
//serializare
void writeToFile(fstream& f) {
    //scriere int(id)
    f.write((char*) & this->id, sizeof(int));

    //scriere string(denumire)
    //1. scriere lungime sir
    int lg = this->denumire.length()+1;
    f.write((char*) & lg, sizeof(int));
    //2. scriere sir de caractere
    f.write(this->denumire.data(), lg);//data() returneaza chiar char*
```

EXEMPLU CITIRE FISIER BINAR

```
//deserializare
void readFromFile(fstream& f) {
    //dezalocare zona de memorie pentru memoria alocata anterior de obj this
    if (this->coordonator != nullptr) {
        delete[] this->coordonator;
        this->coordonator = nullptr;
    }
    if (this->bugetEtape != nullptr) {
        delete[] this->bugetEtape;
        this->bugetEtape = nullptr;
    }
    //citire int(id)
    f.read((char*) & this->id, sizeof(int));

    //citire string(denumire)
    int lg;
    f.read((char*) & lg, sizeof(int));
    char* buffer = new char[lg + 1];
    f.read(buffer, lg);
    this->denumire = buffer;
```

METODE UTILE

- eof() - returnează true dacă s-a ajuns la sfârșitul fișierului
- seekg() - poziționare relativă/absolută în fișiere de intrare
- seekp() - poziționare relativă/asbolută în fișiere de ieșire
- Poziționarea relativă se face față de ios::beg, ios::cur sau ios::end
- tellg() - poziționarea cursorului pentru fișiere de intrare
- tellp() - poziționarea cursorului pentru fișiere de ieșire

SUPRADEFINIRE C12 + C13

DOMENII DE NUME

- Utilizate pentru a organiza codul sursă
- Permit gruparea de clase, obiecte, funcții sau variabile globale într-o singură entitate
- Evită coliziunile de nume (și astfel erorile de ambiguitate)
- Organizarea în namespaces se poate face în funcție de diferite criterii precum functionalitate comuna, modul sau biblioteca specifica, tip aplicatie, etc

EXEMPLU NAMESPACE + UTILIZARE

```cpp
#include<iostream>
using namespace std;

namespace TEST {
    int variabilaTEST = 10;
}

namespace TEST2 {
    int variabilaTEST = 20;
}

using namespace TEST2;
//using namespace TEST;

int main() {
    cout << TEST::variabilaTEST << endl;
    cout << TEST2::variabilaTEST << endl;
    cout << variabilaTEST << endl;
    return 0;
}
```

DERIVARE MULTIPLA

- În C++ este permisă derivarea din mai multe clase simultan
- Utilizarea acesteia modelează faptul că tipul derivat este, în același timp, și oricare dintre tipurile de bază
- Se pot deriva oricât de multe clase, iar tipul de derivare se decide pentru fiecare clasă ce este derivată în parte
- Ordinea de apel a constructorilor/destructorilor în acest caz este dată de ordinea derivării (nu de cea definită explicit în cazul unor constructori)
- Upcasting-ul se poate face către oricare dintre clasele de bază

PROBLEME DERIVARI MULTIPLE

- Una dintre problemele ce poate apărea în cazul derivării multiple este aceea a claselor de bază ce conțin un membru cu aceeași denumire
- În acest caz, este obligatorie utilizarea rezoluției de clasă pentru a specifica în mod explicit la care dintre membrii moșteniți se face referire
- Definirea unui nou membru cu același nume în clasa derivată, va duce la ascunderea („hiding”) membrilor cu nume comun moșteniți

EXEMPLU DERIVARE MULTIPLA

```
class Persoana {
public:
    string nume;
    string adresa;
};

class Angajat {
    string adresa;
};

class Test :public Persoana, public Angajat {

};
```

PROBLEMA MOSTENIRII IN ROMB

Din cauza moștenirii multiple, poate apărea o situație bizară în C++ atunci când o clasă se derivează multiplu din cel puțin două clase cu o bază comună

- În aceste cazuri membrii clasei de bază sunt moșteniți de mai multe ori în clasa finală (de fiecare dată pe filiera unei clase de pe nivelul intermediar)
- Astfel se multiplică spațiul de memorie ocupat de aceștia în clasa finală din ierarhia de derivări
- În plus, upcasting-ul nu este permis direct către prima clasă din ierarhie, putându-se face doar etapizat
- Dacă acest lucru nu deranjează, lucrurile pot rămâne așa

RECOMANDARI

- Recomandarea generală este ca această situație să fie evitată
- Atunci când acest lucru nu este posibil, poate fi rezolvată prin derivarea virtuală a claselor de pe nivelul intermediar (a nu se confunda conceptul cu cel de funcție virtuală, despre care vom discuta puțin mai târziu)
- Dacă se utilizează derivarea virtuală, atunci membrii comuni sunt moșteniți o singură dată, iar upcasting-ul direct către clasa părinte este permis

SUPRADEFINIRE

- Cea de-a doua formă de polimorfism
- Considerată polimorfism pur sau puternic, deoarece funcțiile au exact aceeași signatură, iar alegerea funcției potrivite (aferente apelului) se face la momentul execuției („late binding”)
- Poate avea loc doar în contextul unei ierarhii de derivări, și doar dacă funcțiile/metodele sunt marcate ca virtuale
- Dacă funcția nu este virtuală, are loc doar ascunderea metodei din clasa de bază
- Se manifestă doar pe adrese, în funcție de conținutul de la acea zonă de memorie

FUNCTII VIRTUALE PURE

- Pot exista situații când o funcție nu are neapărat sens pentru o clasă de bază, ci doar pentru clasele derivate
- Sau, privind problema din punctul de vedere opus, situații când vreau să oblig clasele derivate să supradefinească anumite metode (astfel încât să nu le moștenească implementarea din clasa de bază)
- Dacă metoda nu conține cod (cazul în care returnează void, de exemplu) sau returnează o valoare aleatoare, problema de mai sus nu va fi rezolvată, deoarece această implementare va fi moștenită de clasele derivate

FUNCTII VIRTUALE PURE

- Din acest motiv au fost introduse funcțiile virtuale pure, funcții virtuale ce nu au corp și doar obligă clasele derivate să le supradefinească
- Pentru a nu fi confundate cu signaturi de metode ce vor fi implementate outline, funcțiile virtuale pure au un mod special de a fi declarate, de forma:

```
virtual tip_returnat nume_functie(parametri) = 0;
```

CLASE ABSTRACTE

- O clasă ce conține cel puțin o metodă virtuală pură poartă denumirea de clasă abstractă
- Clasele abstracte nu se pot instanția (nu se pot crea obiecte de tipul clasei abstracte), ele au rolul doar de bază pentru derivări
- Cu toate că nu se pot instanția, clasele abstracte pot conține atribute, constructori, destructori și alți membri; rolul acestora va fi de a fi moșteniți și utilizați de clasele derivate, nu a de a fi apelați în mod direct (lucru ce nu este posibil - va returna eroare de compilare)

CLASE ABSTRACTE

- Așadar singurul rol al claselor abstracte este de a fi derivate (constituie bază pentru derivări)
- Crearea lor, fără a le derivarea este în mare măsură inutilă
- Clasele abstracte, de obicei, modelează un comportament pe care îl impun apoi tuturor claselor derivate (prin intermediul metodelor virtuale pure)
- Un alt mod de a spune asta, pe care îl veți regăsi în literatura de specialitate: clasele abstracte reprezintă contracte - a deriva o clasă abstractă e același lucru cu a semna un contract (te obligi să supradefinești toate metodele virtuale pure)

INTERFETE

- Cu toate că în limbajul C++ nu există un cuvânt cheie sau o modalitate propriu-zisă de a defini interfețe, acestea sunt privite mai mult ca o convenție
- În C++ numim interfață o clasă abstractă ce conține doar metode virtuale pure (și nimic altceva)
- A nu se confunda noțiunea de interfață (tip special de clasă abstractă) cu noțiunea de interfață a clasei (mulțimea membrilor publici ai clasei respective)

INTERFETE IN ALTE LIMBAJE

- În alte limbaje de programare orientate obiect, interfețele poarta denumirea de actiune tocmai pentru a sugera un comportament (ex: Comparable, Cloneable, Runnable, etc.)
- În unele limbaje (C#) chiar încep cu „I” pentru a sublinia faptul că sunt interfețe
- De asemenea se utilizează cuvânt cheie separat (interface) pentru a fi definite
- Moștenirea multiplă este permisă în aceste limbaje doar dintr-o clasă și oricâte interfețe

CLASE TEMPLATE, STL
C13 + C14

FUNCTII TEMPLATE/SABLON

- Permit generalizarea codului sursă prin utilizarea de tipuri generice, ce sunt înlocuite la compilare de tipuri concrete de date
- În acest fel codul sursă poate fi refolosit pentru mai multe tipuri de date, fără a fi nevoie de duplicarea lui
- Unii autori consideră funcțiile/clasele șablon ca fiind tot o formă de polimorfism
- Asemănătoare noțiunii de funcție/clasă generică din alte limbaje (C#, Java), dar nu neapărat identice (în alte limbaje ce folosesc mașină virtuală înlocuirea tipului generic cu unul concret se poate realiza la momentul execuției)

EXEMPLU FUNCTIE SABLON

```
int sumaInt(int v1, int v2) {
    return v1 + v2;
}

//functie sablon/template
template<class T>
T suma(T v1, T v2) {
    return v1 + v2;
}

template<class T1, class T2>
T1 sumaGenerala(T2 v1, T2 v2) {
    return (T1)(v1 + v2);
}
```

CLASE SABLON/TEMPLATE

- În mod similar cu funcțiile șablon se pot defini și clase șablon (template)
- Clasa respectivă poate utiliza tipul șablon pentru a defini atribute, parametri, tipuri returnate de metode, etc.
- În faze de pre-compilare, compilatorul folosește șablonul definit pentru a înlocui tipul/tipurile șablon cu tipul/tipurile de dată(e) specificate la folosirea (instanțierea) șablonului

EXEMPLU CLASA TEMPLATE

```
template<class T>
class Container {
    T v[10];
    int nrElem;

public:
    Container() {
        this->nrElem = 0;
    }

    Container(T _v[10], int _nrElem) {
        this->nrElem = _nrElem;
        for (int i = 0; i < this->nrElem; i++)
            this->v[i] = _v[i];
    }
```

CLASE TEMPLATE

- Funcționarea template-ului pe anumite tipuri de date se verifică doar la instanțierea lui (acest lucru poate face că template-ul să nu funcționeze pe anumite tipuri de date definite de utilizator, dacă nu sunt definite/supraîncărcate anumite metode sau operatori)
- Funcțiile outline trebuie să specifice din nou definiția șablonului
- Atunci când template-ul nu funcționează pentru anumite tipuri (sau funcționează parțial) se pot defini specializări ale șablonului pentru acel tip de dată, ce vor avea o prioritate mai mare față de șablonul propriu-zis