PROGRAMARE ORIENTATĂ OBIECTE

Curs 3

Facilități ale limbajului C++

Principii POO

Pointeri către un obiect de tip necunoscut
Pointer nul
Struct/class layout
Forward declaration
Câmpuri de biți
Funcții inline

Class
Constructori de inițializare
Destructor

Pointeri către un obiect de tip necunoscut

- Într-o secvenţă de cod *low level* este necesar a transmite adresa unei locaţii de memorie fără a cunoaşte tipul datei/datelor stocate în acea locaţie.
- Pentru asta se foloseşte pointerul către tipul de date necunoscut ***void****
- Un pointer către orice tip de dată poate fi atribuit unei variabile de tip ***void****.
- Un pointer pe funcţie sau la un membru al unei clase nu poate fi atribuit unei variabile de tip ***void****.
- Un pointer de tip ***void**** poate fi atribuit/comparat unui/cu alt pointer de tip ***void****.
- Un pointer de tip ***void**** poate fi convertit explicit la orice tip de date.

Pointeri către un obiect de tip necunoscut

```
int f(int *p)
{
    void *pv = p;//conversie implicita
    *pv;//eroare, un pointer pe void* nu poate fi dereferentiat
    ++pv;//eroare, un pointer pe void* nu poate fi incrementat

    int *pi = (int*)pv; //conversie explicita la int*
}
```

- Sfaturi:
  - A nu se utiliza pointeri convertiţi explicit către un alt tip de date decât cel iniţial
  - A se utiliza pointeri la void* doar pentru a fi:
    - transmişi ca parametru funcţiilor iar tip acestora nu este necesar a fi cunoscut;
    - returnat de funcţii.
  - A se converti explicit pointierii la void* atunci când sunt folosiţi;

Pointeri către un obiect de tip necunoscut

- Având în vedere că pointeri la ***void**** sunt folosiţi în programarea *low level*, acolo unde resursele hardware sunt manipulate, apariţia unor astfel de pointeri la nivel *high level* trebuie privită cu suspiciune deoarece cu siguranţă se datorează unor erori de proiectare.

Pointer nul

- ***nullptr*** reprezintă un pointer nul, un pointer ce nu pointează către nici un obiect.
- Poate fi atribuit oricărui tip de pointer

```
int *pi = nullptr;
double *pd = nullptr;
int i = nullptr; // eroare : i nu este un pointer
```

Pointer nul

- Înainte de a fi introdus **`nullptr`**, era folosit zero (**`0`**) ca notaţie pentru pointer nul. `int *x = 0;`
- Nici un obiect nu este alocat la adresa **`0`**, iar zero (**`0`**) este cea mai comună reprezentare a **`nullptr`**.
- Zero (**`0`**) este un int. Totuşi în conversiile standard **`0`** este folosit ca o constantă a unui pointer sau pointer la membru.
- O altă reprezentare a unui pointer nul a fost macrodefiniţia **`NULL`**. `int *p = NULL;`
- Totuşi există diferenţe in definiţia **`NULL`** în diferite implementări:

```
#define NULL 0
#define NULL 0L
#define NULL (void*)0 /*C style*/

int *p = NULL; //eroare: nu poate fi atribuit un void* la un int*
```

Struct/class layout

- O structură/clasă îşi păstrează membri în ordinea în care au fost declaraţi:

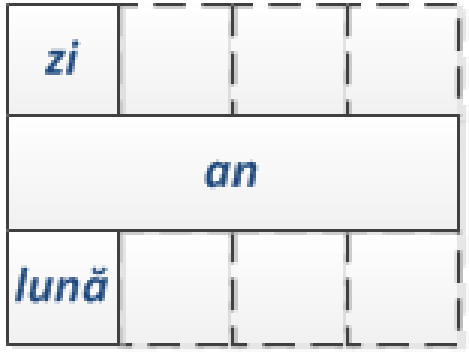
```
struct DataCalendaristica
{
    char zi;   // [1:31]
    int an;
    char luna; // [1:12]
};
```

- Aranjarea în memorie a membrilor ar putea fi aceasta:

- Totuşi, dimensiunea unui obiect nu este suma dimensiunilor membrilor

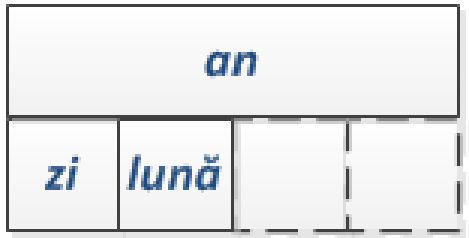
Struct/class layout

- Ex: pe 32 de biţi dimensiunea unui obiect de tip ***DataCalendaristica*** este de 12 octeţi şi nu de 6.



- Optimizare:

```
struct Data
{
    int an;
    char zi; // [1:31]
    char luna; // [1:12]
};
```



Definiția și utilizarea unei structuri

- Un nume (identificator) devine accesibil imediat ce a fost scris şi nu după declarare completă:

```
struct Link
{
    Link* previous;
    Link* successor;
};
```

- Totuşi nu se poate declara un obiect de un anumit tip dacă declaraţia tipului respectiv nu este finalizată

```
struct Link
{
        Link data;
};
```

- Eroare, deoarece compilatorul nu poate determina dimensiunea tipului ***Link***.

Forward declaration

- Referiri simultane:

```
struct List; // declararea numelui unei structuri: Va fi
             //definita mai tarziu

struct Link
{
    Link* prev;
    Link* next;
    List* member_of;
    int data;
};

struct List
{
    Link* head;
};
```

- Numele unei structuri poate fi folosit înainte de a fi definită structura, dar nu poate instanţia un obiect.

Câmpuri de biți

- Tipul de date ***char*** este cel mai mic obiect ce poate fi alocat independent
- Se poate considera risipă de spaţiu utilizarea unui ***char*** în declararea unei variabile binare (singurile valori pe care le poate avea sunt 0 şi 1)
- Există posibilitatea de împacheta astfel de variabile ca fiind câmpuri de biţi într-o structură/clasă.
- Un membru este definit ca fiind un câmp de biţi prin specificarea numărului de biţi pe care îl ocupă
- Un câmp de biţi trebuie sa fie de tip ***int*** sau ***enum***
- Nu se poate obţine adresa unui câmp de biţi
- Sunt permise câmpurile fără nume

Câmpuri de biți

```
struct student
{
    unsigned int nrMatricol;
    int : 8;
    bool promovat : 1; //[0..1]
    bool camin : 1; //[0..1]
    unsigned int grupa : 11; //[1101..1410]
    unsigned int nota : 4; // [1..10]
    unsigned int vârsta : 7; // [1..127]
};
```

- Utilizarea câmpurilor de biţi pentru împachetarea variabilelor salvează un câţi va octeţi în detrimentul dimensiunii şi vitezei de execuţie a unui program.
- Pot fi o alternativă convenabilă pentru lucru pe biţi.

Funcții inline

```
inline int factorial(int n)
{
    return (n<2) ? 1 : n*fac(n-1);
}
```

- Specificatorul inline sugerează compilatorului să nu apeleze funcţia ci să:
  - înlocuiască apelul cu secvenţa de cod conţinută de funcţie
  - ori să genereze cod
- Pentru un apel de genul ***factorial(6)*** un compilator „inteligent” va genera constanta ***720***
- Gradul de „inteligenţă” al unui compilator nu poate fi legiferat, motiv pentru care un compilator poate genera constanta ***720***, un altul ***6*factorial(5)***, un simplu apel de funcţie ***factorial(6)***
- Pentru a avea garanţia că o valoare este calculată la compilare se va declara funcţia ca fiind ***constexpr***
- Pentru a avea garanţia că o funcţie este ***inline*** se va defini (nu doar declara) în domeniul în care este utilizată.

Tehnica Programării Orientate pe Obiecte
Prezentare generală

- Orice structură poate fi definită printr-o clasă în care toate elementele sunt publice.

```
class Durata
{
private: //declara'ie implicita
    int ora, min, sec;
public:
    void Seteaza(int, int, int);
    void Scrie(void);
    void Citeste(void);
    void Aduna(void);
    int EsteEgalaCu(Durata);
};
```

- Tipurile abstracte de date se definesc folosind noţiunea (conceptul) de clasă (c++ ***class***).

Tehnica Programării Orientate pe Obiecte
Prezentare generală

- ***O clasă*** defineşte atât reprezentarea datelor cât şi funcţiile care au acces la date şi le pot prelucra.
- Utilizatorii clasei au acces numai la componentele ***publice***
- Datorită acestei încapsulări, componentele unei clase sunt de 3 tipuri, funcţie de ***nivele de protecţie*** specificate prin intermediul etichetelor ***private***, ***protected*** şi ***public***.

Tehnica Programării Orientate pe Obiecte
Prezentare generală

- Într-o definiţie de clasă pot exista mai multe secţiuni cu acelaşi nivel de protecţie

```
class X
{
public:
    //-----------membri publici
private:
    //-----------membri privati
public:
    //-----------membri publici
};
```

- Ideea de protejare deosebeşte de fapt structura de clasă.

```
struct X
{

    //-----------
};
```

```
class X
{
public:
    //-----------
};
```

Tehnica Programării Orientate pe Obiecte
Prezentare generală

- Asociind unei clase valori concrete pentru membrii săi se obţine un **obiect.**
- Un obiect este o ***instanţiere*** (sau instanţă) a clasei care defineşte tipul său.
- Variabilele membre definesc ***proprietăţile*** obiectului, în timp ce funcţiile definesc ***comportamentul***.
- Extinderea valabilităţii operatorilor existenţi în limbaj se numeşte ***supraîncărcarea operatorilor***
- Variabilele membre se numesc scurt membri iar funţiile membre metode

Tehnica Programării Orientate pe Obiecte
Prezentare generală

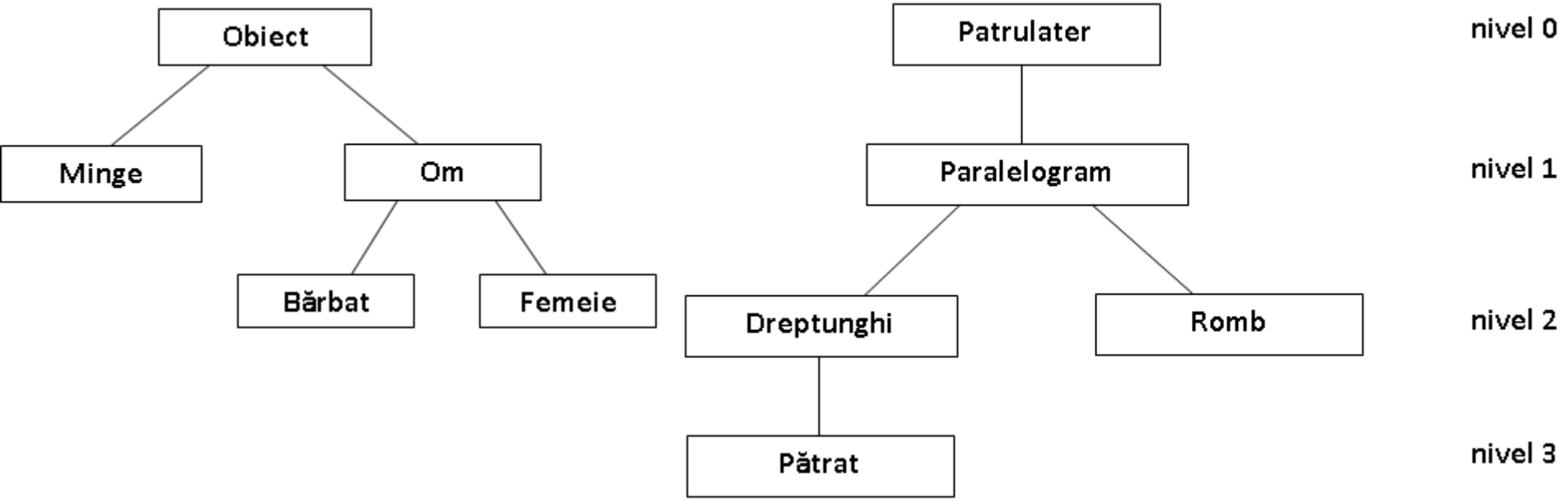
- Clasele pot conţine:
  - variabile de orice tip
  - funcţii de orice tip
  - metode de tip constructori – inţializează membri unei clase
  - metodă de tip constructor de copiere – implementează operaţia de copiere a unui obiect
  - metodă de tip destructor – realizează distrugerea obiectului (în special eliberarea memoriei alocate dinamic)
  - definiţia unor noi tipuri de date
  - funcţii operator
  - etc.

Tehnica Programării Orientate pe Obiecte
Prezentare generală

- Când se lucrează cu un tip abstract de date, el va fi descris prin prezentarea interfeţei şi a implementării sale.
- ***Interfaţa*** este reprezentată de **membrii publici** ai clasei.
- De obicei, diferite tipuri de date abstracte înrudite au elemente comune.
- Precizarea acestor elemente comune duce la o “ierarhizare” arborescentă a tipurilor abstracte de date.
- Prin ierarhizarea claselor, componentele unei clase rămân valabile şi pentru clasele de nivel ierarhic inferior. Această mecanism se numeşte ***moştenire.***
- Folosind moştenirea, se pot obţine clase noi, prin adăugarea de componente noi la o clasă deja existentă, numite clase derivate

Tehnica Programării Orientate pe Obiecte
Prezentare generală

- Clasa din care se obţin clasele derivate se numeşte ***clasă de bază***.



Tehnica Programării Orientate pe Obiecte
Prezentare generală

- Dintr-un tip abstract de date se poate crea un obiect. Cum un tip abstract de date este o colecţie de variabile rezultă:
  - Variabilele din tipul abstract de dată reprezintă informaţia care dă proprietăţile tipului
  - Funcţiile din tipul abstract de dată reprezintă operaţiile ce pot fi făcute cu datele acelui tip şi reprezintă comportamentul tipului
- Clasele abstracte nu se pot instanţia în mod direct deoarece nu exista suficiente informaţii pentru a construi acele instanţieri
- Un obiect poate fi descris folosind proprietăţile altui obiect la care se adaugă proprietăţi suplimentare. Astfel se pot defini clase grupatate ierarhic, procedeul fiind numit moştenire
- Polimorfism – o anumită operaţie (funcţie) poate funcţiona diferit cu obiecte diferite, dar înrudite

Tehnica Programării Orientate pe Obiecte
Moştenire, polimorfism

- Un obiect poate fi descris folosind proprietăţile altui obiect la care se adaugă proprietăţi suplimentare. Astfel se pot defini clase grupatate ierarhic, procedeul fiind numit moştenire
- Polimorfism – o anumită operaţie (funcţie) poate funcţiona diferit cu obiecte diferite, dar înrudite

Programarea orientată pe obiecte

- Scopul limbajului C++ este acela de a furniza programatorului o unealtă pentru crearea de noi tipuri de date.
- În plus, clasele derivate şi şabloanele (templates), furnizează căi, metode de organizare a claselor „înrudite”, astfel încât programatorul să dispună de avantajele oferite.
- Un tip de date este o reprezentare concretă a unui concept.
- Ex: tipul de date predefinit `float` cu operatorii `+`,`-`,`*`, etc. asigură o aproximare concretă a conceptului matematic a unui număr real.
- O clasă este un tip definit de utilizator. Se proiectează un nou tip pentru a furniza un concept care nu se regăseşte in lista tipurilor predefinite. Ex: tipul de date complex.

Programarea orientată pe obiecte

- Un program care furnizează tipuri de date cât mai apropiate de conceptele aplicaţiei tinde să fie mai uşor de înţeles şi uşor de modificat comparativ cu un program care nu face acest lucru.
- Un set foarte bine ales de tipuri definite de utilizator face programul mult mai concis.

Clasa – tip definit de utilizator

- Funcţii membre
  - Fie implementarea conceptului de dată calendaristică utilizând *struct* pentru a defini reprezentarea *Date* (dată calendaristică) şi un set de funcţii pentru manipularea variabilelor de acest tip.

```
struct Date
{
        int d, m, y;
};
void InitData(Date &d, int, int, int);    //initializare
void AddYear(Date &d, int n);
void AddMonth(Date &d, int n);
void AddDay(Date &d, int n);
```

  - Nu există o corespondenţă explicită între tipul Date şi aceste funcţii.

Clasa – tip definit de utilizator

- Această corespondenţă poate fi stabilită declarând funcţiile ca membre ale structurii:

```
struct Date
{
        int d, m, y;
        void Init(int dd,int mm,int yy);
        void AddYear(int n);
        void AddMonth(int n);
        void AddDay(int n);
};
```

- Funcţiile declarate în definiţia unei structuri/clase sunt numite funcţii membre şi pot fi invocate doar de tipul respectiv sau apropiat utilizând sintaxa standard de acces la membrii unei structuri

Clasa – tip definit de utilizator

```
Date myBirthday;
void f(void)
{
        Date today;
        today.Init(16, 10, 1996);
        myBirthday.Init(30, 12, 1950);
        Date tomorrow = today;
        tomorrow.AddDay(1);
}
```

- Deoarece diverse structuri pot avea funcţii membre cu acelaşi nume, trebuie specificat numele unei structuri când este definită o funcţie

```
void Date::Init(int dd, int mm, int yy)
{
        d = dd;
        m = mm;
        y = yy;
}
```

Clasa – tip definit de utilizator

- Într-o funcţie membră, numele membrilor pot fi folosiţi fără o referinţă explicită la un obiect.
- Construcţia:

class X { … };

este numită *definirea clasei* deoarece defineşte un nou tip de date. Din motive istorice definirea unei clase mai este numită şi *declararea clasei*. Din acest motiv definiţia clasei poate fi replicată în diverse surse utilizând *#include* fără să fie încălcată regula definiţiei unice.

Clasa – controlul accesului

- Declaraţia structurii *Date* de mai înainte pune la dispoziţie un set de funcţii pentru manipularea membrilor. Totuşi nu specifică faptul că sunt singurele funcţii care au acces direct la membri. Această restricţie poate fi exprimată folosind ***class*** în loc de ***struct***:

```
class Date
{
        int d, m, y;
public:
        void Init(int dd, int mm, int yy);
        void AddYear(int n);
        void AddMonth(int n);
        void AddDay(int n);
};
```

- Cuvântul cheie public separă corpul clasei în două părţi. Membri, *privaţi,* din prima parte pot fi folosiţi doar de funcţiile membre. Membri publici constituie interfaţa publică a obiectelor clasei.

Clasa – controlul accesului

- Structura este simplu o clasă a cărei membri sunt implicit publici.
- Funcţiile membre unei clase se definesc în acelaşi mod ca la o structură:

```
inline void Date::AddYear(int n)
{
        y += n;
}
```

- Totuşi funcţii nemembre nu pot accesa membri privaţi:

```
void Timewarp(Date &d)
{
        d.y= 200; //eroare: Date::y este privat
}
```

  - Avantaje: acces controlat, actualizare prin intermediul interfaţei

Clasa – specificatori de acces

- Modifică drepturile de acces către membrii clasei
  - `private`: membrii privaţi pot fi accesaţi numai de membrii aceleiaşi clase *sau de membrii claselor prietene*
  - `protected`: membrii protejaţi sunt accesibili în cadrul aceleiaşi clase *sau dintr-o clasă prietenă, dar şi din clasele derivate din acestea*
  - `public`: membrii publici sunt accesibili de oriunde din domeniul de vizibilitate al obiectului de tip clasă
- În cadrul unei clase, specificatorul de acces implicit este *private*
- În cadrul unei structuri, specificatorul de acces implicit este *public*

Clasa – constructori

- Utilizarea funcţiei *Init()* pentru a initializa obiectele clasei nu este elegantă şi generatoare de erori.
- O abordare potrivita este de a permite programatorului să declare o funcţie al cărei scop explicit este acela de iniţializare. Deoarece astfel de funcţii se construiesc obiecte au fost numite *constructori.*
- Un constructor are acelaşi nume ca şi clasa:

```
class Date
{
        //...
        Date(int, int, int);//constructor
};
```

Clasa – constructori

```
Date today = Date(8, 11, 2012);
Date Xmas(25, 12, 2012);      //formă abreviată
Date MyBirthday;      //eroare: lipsa parametri de initializare
Date Release1_0(10, 12);//eroare: lipsește al treilea argument
```

- Este de dorit a se furniza diferite moduri de iniţializare a unui obiect

```
class Date
{
        int d, m, y;
public:
        Date(int, int, int);  //day, month, year
        Date(int ,int);       //day, month, anul curent
        Date(int);            //day, luna si anul curent
        Date(void);           //valori implicite
        Date(const char*);    //data sir de caractere
};
```

Clasa – constructori

- Regulile de supraîncărcare a funcţiilor se aplică şi în cazul constructorilor

```
Date today(4);
Date july4(„1 Dec,2012");
Date guy(„8Nov");
Date now;
```

- Constructori pot avea argumente predefinite

```
class Date
{
        int d, m, y;
public:
        Date(int dd = 0, int mm = 0, int yy = 0);
        Date(void):d(0), m(0), y(0){};
};
```

Constructori şi destructori

- Constructorul:
  - metodă specială a unei clase
  - are acelaşi nume ca şi clasa
  - nu are un tip de return
  - este apelat automat la declararea obiectului
- Cu ajutorul constructorului:
  - se creează un obiect pentru care se face alocarea spaţiului de memorie necesar
  - se pot iniţializa variabilele membre ale clasei
- Constructor implicit
  - Se apelează atunci când nu există nicio declaraţie a unui constructor în clasă

Constructori

- Alte tipuri de constructori:
  - Constructor fără listă de argumente
  - Constructor de iniţializare (cu listă de argumente)
  - Constructor de copiere
- Dacă se defineşte cel puţin un constructor, nu se va mai genera constructorul implicit
- Pot exista mai mulţi constructori ai aceleiaşi clase
- Toate declaraţiile obiectelor trebuie să respecte un model din mulţimea constructorilor clasei

Clasa - Destructor

- Ajută la eliberarea zonelor de memorie
- Are rol opus constructorului
- Nu are tip de return
- Are numele clasei precedat de caracterul ~
- Nu primeşte nici un parametru
  - O clasă are un singur destructor!
- Se apelează automat când domeniul de vizibilitate a obiectului s-a terminat
- Se recomandă să se realizeze atunci când clasa conţine pointeri care sunt alocaţi dinamic în constructor sau în alte metode prezente în clasă

```
~Date(void);
```

Vă mulţumesc !