Tipuri de date abstracte în limbajul C++ - Clase

- Programarea orientată obiect are în vedere în principal organizarea globală a programului şi nu în mod primordial detaliile de funcţionare a programului. Procesul de abstractizare permite astfel ignorarea detaliilor de implementare

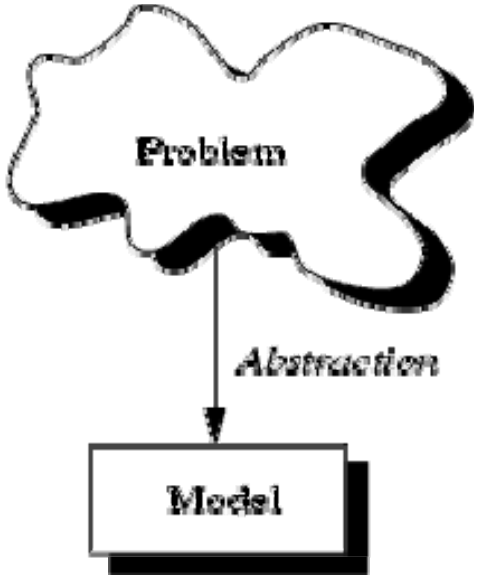

Figura 3.1. Procesul de abstractizare

- Modelul defineşte o viziune abstractă asupra problemei, care se focalizează pe aspectele legate de identificarea proprietăţilor esenţiale ce o caracterizează (date şi operaţii)

- Tipul de date abstract este o entitate manevrată doar prin operaţiile ce definesc acel tip. Avantajele utilizării tipurilor de date abstracte sunt:

    - Programele devin independente de modul de reprezentare a datelor. Modul de reprezentare poate fi modificat, fora însa a afecta restul programului
    - Se previne modificarea accidentală a datelor. Utilizatorul tipului abstract este forţat să manipuleze datele doar prin intermediul metodelor/operatorilor ce compun tipul abstract, astfel reducându-se riscul alterărilor/distrugerilor datelor.
    - Se permite verificarea riguroasă la compilare a utilizării corecte a datelor de tipul respectiv.

- Descrierea oricărui tip de date abstract presupune precizări asupra celor două elemente componente:

    - Date
    - Operaţii (inclusiv descrierea interfeţei)

Figura 3.1. Structura unui tip de date abstract (abstract data type - ADT)

- Limbajele orientate obiect permit programarea folosind tipuri abstracte de date, care, în acest context, se numesc **clase**
- Crearea unei instanţe a unei clase (instanţierea), cu proprietăţi bine definite, duce la crearea unui **obiect**
- În concluzie, ideea fundamentală care stă la baza limbajelor orientate obiect este aceea de a combina (încapsula) într-o singură entitate atât **date,** numite **date membru,** cât şi **funcţii,** numite **funcţii membru (metode)**, care operează asupra acelor date. O astfel de entitate se numeşte **obiect**.
- Abordarea unei probleme de programare din perspectiva unui limbaj orientat obiect nu mai presupune împărţirea problemei în funcţii, ci identificarea obiectelor implicate în dezvoltarea soluţiei

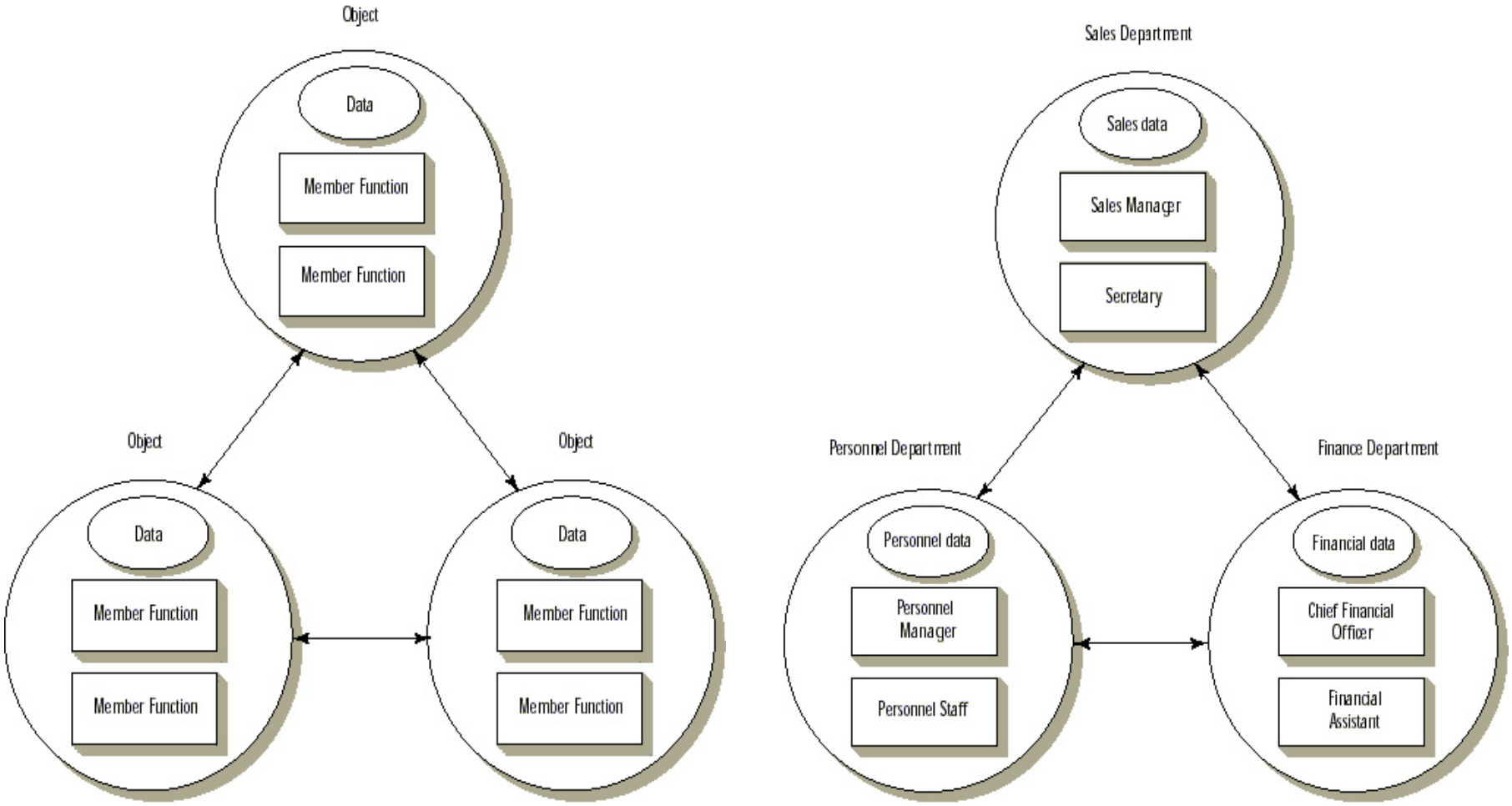

Figura 3.2. Paradigma programării orientate obiect şi analogia cu departamentele unei firme

- O clasă descrie unul sau mai multe obiecte ce pot fi precizate printr-un acelaşi set de atribute si metode.
- Se spune că un obiect este o instanţiere (instanţă) a unei clase.
- Un obiect este caracterizat de:
    - nume
    - atribute (date) - valorile atributelor la un moment dat definesc o stare
    - metode (funcţii, servicii, operaţii)
- Un obiect se identifică în mod unic prin numele său şi el defineşte o stare reprezentată de atributele sale la un moment particular de timp
- UML (Unified Modeling Language) este un limbaj de modelare, folosit pentru a reprezenta soluţiile sistemelor software, fundamental legat de metodologia OO (Object-Oriented) folosită pentru dezvoltarea de software, fără a fi doar un simplu limbaj de modelare orientat pe obiecte, ci în prezent, este limbajul universal standard pentru dezvoltatorii software
- Diagrama folosită în modelarea obiect se numeşte diagramă de clase şi ea oferă o notaţie grafică pentru reprezentarea claselor şi a relaţiilor dintre ele. Diagramele de clase descriu cazuri generale în modelarea sistemului.
- Clasele sunt reprezentate prin dreptunghiuri împărţite în trei compartimente şi care conţin numele clasei (în compartimentul superior), lista de atribute ale clasei (opţional) şi lista de operaţii (opţional) cu argumentele lor şi tipul returnat. Cele trei compartimente vor fi separate între ele prin câte o linie orizontală.

Nume clasă
Atribute (date membru)
Metode (Funcţii membru)

Figura 3.3. Diagramă de clasă UML

- Conceptele implementate prin clase au:
    - o parte protejată, numită şi privată, care exprimă partea specifică, proprie conceptului şi are rolul de a asigura **implementarea** clasei (conceptului)
    - o parte neprotejată, numită şi publică, ce reprezintă **interfaţa** cu celelalte clase ale programului.

Observaţie: În general, partea publică poate conţine şi date membru, dar atunci datele respective nu mai sunt protejate !

- Pentru a realiza protecţia datelor se utilizează modificatori de protecţie :
  - ***private*** - accesul la componentele private se face doar prin intermediul funcţiilor membru publice ale clasei
  - ***protected*** – modificator utilizat în contextul derivării claselor
  - ***public*** – acces nerestricţionat
- O clasă se specifică printr-o declaraţie de clasă :

```
class NumeClasa {
    // implicit privat ...
    // date şi funcţii membru

  public:
    // tot ce urmează este public până ...
    // date şi funcţii membru

  private:
    // ... aici, unde se comută din nou pe privat până ...
    // date şi funcţii membru

  protected:
    // ... aici, unde se comută pe protejat ...
    // date şi funcţii membru

  public:
    // ... şi din nou public.
    // date şi funcţii membru
};
```

- Numele clasei este un tip de dată abstract şi se poate folosi ca atare pe post de tip de date în continuare
- În cazul apelului unei funcţii membru, acesteia i se transmite implicit un pointer (constant!) către obiectul curent care a iniţiat apelul - denumit pointerul ***this***, ce are declaraţia implicită :

  NumeClasa ****const this***;

- Pointerul ***this*** poate fi utilizat explicit de către programator în funcţiile membru dacă este nevoie să se facă referire la obiectul spre care acesta pointează

- Funcţiile membru foarte simple pot fi definite în interiorul declaraţiei clasei, devenind astfel funcţii ***inline***
- Pentru funcţiile complexe, se declară în declaraţia de tip doar prototipul
- Funcţiile membru definite în exteriorul declaraţiei clasei pot fi ***inline*** dacă se declară explicit acest lucru
- O funcţie membru care se defineşte în afara declaraţiei clasei a cărei membră este se califică folosind numele tipului(clasei) prin intermediul operatorului de rezoluţie :

Tip_returnat NumeClasa :: nume_funcţie_membru(listă_de_parametri)

- Iniţializarea unui obiect este o problemă mult mai complexă decât iniţializarea datelor obişnuite, de aceea se foloseşte în acest scop o funcţie membru specială, numită **constructor**
- Constructorul unei clase are întotdeauna acelaşi nume ca şi numele clasei
- Constructorul unei clase nu returnează nici un tip
- Dacă o clasă are definit un constructor atunci acesta se apelează automat la instanţierea unui obiect al clasei respective, inclusiv în cazul alocării dinamice
- O clasă poate avea mai mulţi constructori care devin astfel **metode supraîncarcate**
- Constructorii pot avea sau nu parametri.
- Constructorul fără parametri se numeşte **constructor implicit**
- Dacă există definit un constructor implicit atunci nu se mai poate defini pentru clasa respectivă un constructor cu toţi parametrii impliciţi
- În cazul în care programatorul nu defineşte explicit nici un constructor, compilatorul generează în mod automat cod pentru un constructor implicit
- Constructorul implicit generat de compilator are doar rolul de a rezerva memorie pentru datele membru ale clasei respective, situaţie inadecvată atunci când la crearea unui obiect este necesară alocarea dinamică de memorie.
- Un constructor al unui obiect este apelat o singură dată pentru obiecte globale sau pentru cele locale alocate static. Pentru obiectele locale, constructorul este apelat de fiecare dată când se creează obiectul respectiv
- În cazul în care o clasă are cel puţin un constructor şi nici unul dintre constructori nu are parametri impliciţi, nu se pot instanţia obiecte neiniţializate (fapt cu consecinţe „fatale” în cazul definirii tablourilor de obiecte !
- Parametrul unui constructor poate fi de orice tip, cu excepţia tipului definit de clasa al cărei constructor este. Sunt permise în acest context doar parametri de tip referinţă sau pointer către clasa respectivă.

```
NumeClasa(NumeClasa *p);
NumeClasa(NumeClasa& p);
```

- Constructorul NumeClasa(const NumeClasa& p) este un constructor special ce permite copierea obiectelor şi se numeşte **constructor de copiere**
- Constructorul de copiere poate avea şi alţi parametri, care însă trebuie să aibă valori implicite
- Constructorul de copiere se apelează :
    - ✓ la crearea unui obiect nou din unul deja existent,
    - ✓ la transferul parametrilor prin valoare şi
    - ✓ în cazul funcţiilor ce returnează obiecte.
- Nu este obligatorie definirea explicită a unui constructor de copiere, acesta poate fi generat automat de către compilator; există însă situaţii în care este absolut necesară definirea unui constructor de copiere şi anume atunci când la crearea unui obiect este necesară alocarea dinamică de memorie.
- Constructorul de copiere generat implicit de compilator realizează o copie identică, bit cu bit (bitwise), a obiectului iniţial în obiectul ţintă
- Una din cele mai frecvente situaţii în care trebuie evitată copierea bit cu bit este cea în care, la crearea obiectului, se alocă memorie dinamic
- În cazul în care se defineşte explicit doar constructorul de copiere, compilatorul nu generează cod pentru constructorul implicit
- Eliberarea spaţiului de memorie ocupat de un obiect (distrugerea obiectului) se poate realiza folosind o altă funcţie membru specială, numită **destructor**.
- Orice clasă are un singur destructor.
- Destructorul nu are parametri şi nu returnează nici un tip
- Numele destructorului se construieşte din numele clasei căreia îi aparţine, prefixat de simbolul ~(tilda)

  ~NumeClasa()
- Destructorul este apelat automat de către un program atunci când încetează existenţa unui obiect (la încheierea execuţiei programului sau atunci când se iese din domeniul de valabilitate al obiectului); în cazul alocării dinamice, operatorul ***delete*** asociat unui pointer ce indică spre o zonă de memorie rezervată pentru un obiect, apelează implicit destructorul clasei
- În cazul în care programatorul nu defineşte explicit destructorul clasei, compilatorul generează în mod automat cod pentru destructor, ce eliberează memoria alocată datelor membru (nu şi memoria eventual alocată dinamic, caz în care trebuie definit explicit destructorul clasei)

În concluzie, declaraţia generică a unei clase poate fi descrisă astfel :

```
 class NumeClasa
{
        // date membru şi metode (implicit) private
        protected:
        // date membru şi metode protejate
        public:
                NumeClasa() ;                           // prototip constructor implicit
                NumeClasa(lista_de_parametri) ;         // prototip constructor cu parametri
                NumeClasa(const NumeClasa&) ;           // prototip constructor de copiere
                ~NumeClasa() ;                          // prototip destructor
        // date membru şi alte  metode publice
};
```

Exemplul 1. Clasa Complex

```
#include <stdio.h>
#include <iostream>
#include <math.h>

using namespace std;

class Complex
{
      double real;
      double imag;
    public:
      Complex()
      {
                  real=0.0;imag=0.0;
      }
      Complex(double x, double y=0){
         real = x;imag = y;
      }
      Complex(const Complex& c){
         real=c.real;imag=c.imag;
         cout<<"Constructor de copiere"<<endl;
     }
      double mod()  // definitie - implicit inline
      {
         return sqrt(real*real+imag*imag);
      }
      double& retreal()  // definitie - implicit inline
      {
         return real;
      }
double& retimag()  // definitie - implicit inline
 {
         return imag;
 }
 void afiscomplex(char* format);  // prototip
   ~Complex(){}
};

void Complex::afiscomplex(char* format)//UZ DIDACTIC!
{
        printf(format,real,imag);
}

int main(void)
{
         Complex z0;   // Numai cu constr de copiere=>ERR
         //:no appropriate default constructor available
          printf("z0=");
          z0.afiscomplex("%g+i%g\n");
          Complex z1(1.0,2.0);
           printf("z1=");  z1.afiscomplex("%g+i%g\n");
           Complex z2(z1);
           printf("z2="); z2.afiscomplex("%g+i%g\n");
           Complex z3=z2;
           printf("z3=");
           z3.afiscomplex("%g+i%g\n");
           //z3.real=100.0; cannot access private member
           //declared in class 'Complex'
           z3.retreal()=100.0;
           z3.retimag()=200.0;
           printf("z3=");   z3.afiscomplex("%g+i%g\n");
           return 0;
}
```