1. Clase. Abstractizarea datelor

Vom face în cursul de Programarea calculatoarelor şi limbaje de programare II o introducere în programarea orientată pe obiecte, cu aplicații în limbajul de programare C++. Cursul de Programarea calculatoarelor şi limbaje de programare I a realizat o prezentare a elementelor de programare structurată pe care le vom folosi acum pentru a crea aplicații care folosesc obiecte. Am introdus, totuşi, şi câteva concepte de programare obiectuală.

Am văzut că programarea orientată pe obiecte (*object oriented programming* – OOP) *încapsulează* date (atribute) şi funcții (comportamente) în pachete numite *clase*. O clasă poate fi asemănată cu un proiect, un şablon, o descriere. Având la dispoziție o clasă, un programator poate să creeze obiecte din acea clasă. Clasele au proprietatea de *ascundere a informației* (*information hiding*). Aceasta înseamnă că, deşi obiectele pot să comunice unele cu altele prin intermediul unor interfețe, claselor nu le este permis să acceseze detaliile de implementare ale altor clase. Se spune că detaliile de implementare sunt ascunse în interiorul claselor. Vom vedea de ce este atât de importantă ascunderea informației în ingineria software.

În programarea procedurală aplicațiile tind să fie orientate pe acțiuni, în timp ce în programarea obiectuală ele sunt, în mod ideal, orientate pe obiecte. În limbajul de programare C care este unul procedural, unitatea de program este *funcția*. În C++, în schimb, unitatea de program este *clasa* din care pot fi *instanțiate* sau create *obiecte*.

Programatorii de aplicații scrise în limbajul C se concentrează pe scrierea de funcții. Grupurile de acțiuni care realizează o operație formează o funcție, iar funcțiile sunt grupate pentru a forma un program. Datele sunt importante pentru o aplicație C, dar ele sunt privite ca un suport pentru acțiunile pe care le realizează funcțiile. Verbele din setul de specificații ale unei aplicații ajută programatorul C să determine care sunt funcțiile necesare implementării sistemului.

Programatorii C++ se concentrează pe crearea propriilor tipuri de date (*user-defined types*) numite *clase*. Clasele se mai numesc si tipuri de dată definite de programator (*programmer-defined types*). Fiecare clasă conține datele şi setul de funcții care le manipulează. Datele care intră în componența unei clase se numesc *date membre*. Funcțiile care intră în alcătuirea unei clase se numesc *funcții membre*. În alte limbaje de programare, datele membre se numesc *proprietăți*, iar funcțiile menbre se numesc *metode*. O instanță a unui tip de dată predefinit (*built-in type*), de exemplu `int`, se numeşte *variabilă*, în timp ce o instanță a unui tip de dată definit de utilizator, adică a unei clase, se numeşte *obiect*. Putem folosi termenul de obiect poate fi folosit şi pentru a ne referi la variabile. În proiectarea unei aplicații orientate pe obiecte ne focalizăm atenția în principal asupra claselor şi mai puțin asupra funcțiilor. Substantivele din specificațiile unui sistem software ne ajută să identificăm setul de clase care urmează a fi folosite pentru crearea obiectelor care, lucrând împreună, implementează aplicația.

Clasele în C++ pot fi privite ca o evoluție de la noțiunea de structură din limbajul C. Vom implementa mai întâi un tip de dată folosind o structură. Vom trece apoi la implementarea aceluiaşi tip de dată folosind o clasă.

1.1Definirea unei structuri

Structurile sunt tipuri de dată agregate definite pe baza altor tipuri de dată predefinite sau definite de programator.

Exemplu

```
struct Time
{
  int hour;
  int minute;
  int second;
};
```

Cuvântul rezervat `struct` se foloseşte pentru a defini o nouă structură. Identificatorul `Time` este numele ales de noi în exemplu pentru a denumi noul tip de dată definit prin această structură. Între acolade se găsesc membrii structurii. Membrii aceleiaşi structuri trebuie să aibă nume unice, însă două structuri diferite pot conține membri cu aceleaşi nume. Definiția unei structuri se încheie cu `;`.

O structură nu poate conține un membru al cărui tip să fie chiar cel definit de ea. În exemplul anterior, structura `Time` nu poate conține un membru de tip `Time`. Poate, însă, să conțină un membru care este pointer la tipul `Time`, fiind vorba, în acest caz, de o *structură autoreferențială*. Aceste structuri sunt utile, e exemplu, atunci când se definesc structuri de date precum listele înlănțuite, stivele sau arborii. Această observație este valabilă şi pentru clase.

Definiția unei structuri nu implică rezervarea de spațiu de memorie. În schimb, declarațiile de obiecte dintr-un tip definit printr-o structură presupun alocari de memorie. Declarația

```
Time timeObject, timeArray[10],
     *timePtr, &timeRef = timeObject;
```

declară `timeObject` ca un obiect de tip `Time`, `timeArray` ca tablou de 10 elemente de tip `Time`, `timePtr` ca pointer la un obiect de tip `Time` şi `timeRef` ca referință la un obiect de tip `Time` inițializat cu `timeObject`.

- *Accesarea membrilor unei structuri*

Membrii unei structuri sau ai unei clase pot fi accesați folosind operatorii de acces al membrilor. Aceştia sunt operatorul **.** şi operatorul **->**.

Operatorul **.** se foloseşte pentru a accesa membri ai structurilor sau ai claselor prin intermediul numelui obiectului sau al unei referințe la obiect.

Exemplu

```
cout << timeObject.hour;
cout << timeRef.hour;
```

În exemplul de mai sus, `timeObject` este un obiect de tip `Time`, iar `timeRef` este o referință la un obiect de tip `Time`.

Operatorul **->** se foloseşte pentru a accesa membrii unei structuri sau ai unei clase prin intermediul unui pointer la un obiect.

Exemplu

```
timePtr = &timeObject;
cout << timePtr->hour;
```

În acest exemplu, `timePtr` este un pointer la un obiect de tip `Time` şi este inițializat cu adresa de memorie a lui `timeObject`.

- *Implementarea tipului de dată `Time` folosind o structură*

Programul de mai jos tipăreşte în două moduri ora stocată într-un obiect de tip `Time`. Funcția `printShort` implementează tipărirea în formatul de 24 de ore fără să includă şi secundele, iar funcția `printLong` pe cea în formatul AM/PM.

Exemplu

```
#include <iostream>
using std::cout;
using std::endl;

struct Time //definitia structurii
{
  int hour;    //0-23
  int minute;  //0-59
  int second;  //0-59
};

void printShort(const Time&);
void printLong (const Time&);

int main()
{
  Time dinnerTime;  //obiect de tip Time
  //asignarea unor valori valide membrilor obiectului
  dinnerTime.hour = 18;
  dinnerTime.minute = 30;
  dinnerTime.second = 0;

  cout << "Cina va avea loc la ora ";
  printShort(dinnerTime);
  cout << " (format scurt), \nadica la ora ";
  printLong(dinnerTime);
  cout << " (format lung).\n" << endl;

  //asignarea unor valori invalide membrilor obiectului
  dinnerTime.hour = 29;
  dinnerTime.minute = 73;

  cout << "Obiect de tip Time cu valori invalide: ";
  printShort(dinnerTime);
  cout << endl;

  return 0;
}

void printShort(const Time &t)
{
  cout << (t.hour < 10 ? "0" : "") << t.hour << ":"
       << (t.minute < 10 ? "0" : "") << t.minute;
}

void printLong(const Time &t)
{
  cout << ((t.hour == 0 || t.hour == 12) ?
              12 : t.hour % 12)
       << ":" << (t.minute < 10 ? "0" : "") << t.minute
```

```
        << ":" << (t.second < 10 ? "0" : "") << t.second
        << (t.hour < 12 ? " AM" : " PM");
}
```

Rulând programul, vom obține următorul rezultat:

```
Cina va avea loc la ora 18:30 (format scurt),
adica la ora 6:30:00 PM (format lung).

Obiect de tip Time cu valori invalide: 29:73
```

Cele două funcții de tipărire au parametri de tip referință la un obiect constant de tip `Time`. Din acest motiv, ele vor primi referințe la obiect, evitându-se astfel dezavantajele legate de ocuparea memoriei şi a timpului necesar copierii întregului obiect. Parametrii fiind de tip `const`, funcțiile nu vor putea schimba conținutul obiectului.

Există mai multe dezavantaje legate de crearea unor noi tipuri de dată cu ajutorul structurilor. Inițializarea membrilor nu este cerută în mod explicit, astfel că am putea avea probleme din cauza folosirii unor date neinițializate. Chiar dacă membrii sunt inițializați, valorile lor pot să nu fie corecte, aşa cum am văzut în exemplu anterior. Lucrul acesta este posibil deoarece programatorul manipulează direct datele membre ale obiectului în lipsa unei interfețe care să asigure validarea valorilor.

O altă problemă legată de implementarea tipurilor de dată cu ajutorul structurilor este că acestea nu pot fi tipărite ca o entitate prin aplicarea operatorului `<<` asupra unui obiect. Clasele permit această facilitate prin supraîncărcarea operatorului `<<`, în felul acesta operatorul de inserare în stream-ul de scriere căpătând o nouă funcționalitate. În plus, obiectele de tip structură nu pot fi comparate prin operatori relaționali, în timp ce pentru obiecte de tip clasă aceşti operatori pot fi supraîncărcați.

- *Implementarea tipului abstract de dată `Time` folosind o clasă*

Vom reimplementa structura Time printr-o clasă C++ şi vom prezenta unele dintre avantajele creării *tipurilor abstracte de date* (*abstract data types*) ca şi clase. Un tip abstract de dată reprezintă un set de date şi un set de operații care se aplică asupra datelor. Clasele şi structurile pot fi folosite aproape identic in C++, diferența dintre cele două fiind legată de modul implicit de acces la membri.

Clasele permit programatorilor să modeleze obiecte care au *atribute*, reprezentate ca *date membre* şi *comportamente* sau *operații*, reprezentate ca *funcții membre*. Pentru definirea unei clase folosim în C++ cuvântul cheie `class`.

În alte limbaje de programare, funcțiile membre se numesc şi *metode* şi sunt invocate ca răspuns la *mesaje* trimise către un obiect. Un mesaj corespunde unui apel al unei funcții membre trimis dintr-un obiect altui obiect sau trimis dintr-o funcție unui obiect.

Odată definită o clasă, numele său poate fi folosit în declararea obiectelor acelei clase.

Exemplu

```
class Time
{
  public:
    Time();
    void setTime(int, int, int);
    void printShort();
```

```
    void printLong ();
  private:
    int hour;
    int minute;
    int second;
};
```

Definiția unei clase începe cu cuvântul cheie `class`. Corpul definiției clasei este delimitat de o pereche de acolade `{` şi `}`. Definiția clasei se termină prin semnul `;`. Clasa `Time` conține trei date membre de tip întreg, `hour`, `minute` şi `second`, ca şi structura `Time` descrisă mai devreme.

Specificatorii de acces la membri : `public`, `protected` şi `private`

Etichetele `public:` şi `private:` se numesc *specificatori de acces la membri*. Orice dată membră sau funcție membră declarată după specificatorul de acces `public` şi înaintea următorului specificator de acces la membri este accesibil oriunde în program unde poate fi utilizat un obiect din clasa `Time`. Orice dată membră sau funcție membră declarată după specificatorul de acces `private` şi înaintea următorului specificator de acces la membri este accesibilă doar funcțiilor membre ale clasei. Specificatorii de acces la membri sunt urmați întotdeauna de semnul `:` şi pot apărea de mai multe ori în definiția unei clase. În capitolele următoare, atunci când vom prezenta noțiunea de moştenire vom introduce un al treilea specificator de acces, `protected`.

Pentru claritatea programului, se preferă specificarea o singură dată a fiecărui specificator de acces la membri în definiția unei clase. În general, membrii publici se plasează primii.

Funcțiile membre

Definiția clasei `Time` din exemplul de mai sus conține prototipurile celor patru funcții membre. Acestea sunt aşezate după specificatorul `public` (în secțiunea `public` a clasei) şi se numesc `Time`, `setTime`, `printShort` şi `printLong`. Acestea se numesc *funcții membre publice* sau *servicii publice* sau *comportamente publice* sau *interfața clasei*. Aceste funcții pot fi folosite de *clienții clasei* pentru a manipula datele clasei. Clienții clasei sunt porțiunile de program care folosesc obiecte ale clasei. Datele membre ale clasei reprezintă suportul pentru *serviciile* oferite de clasă clienților săi prin intermediul funcțiilor membre. Serviciile permit codului client să interacționeze cu un obiect al casei.

În lista de funcții membre se găseşte una care are acelaşi nume cu cel al clasei. Ea este o funcție specială care se numeşte *constructor* al clasei şi care are rolul de a inițializa membrii unui obiect al clasei. Constructorul clasei este apelat automat atunci când se creează în memoria calculatorului un obiect al clasei. O clasă poate avea mai mulți constructori, acest lucru fiind posibil prin mecanismul supraîncărcării funcțiilor. Pentru funcțiile constructor nu se specifică niciun tip al valorii returnate, deci ele nu întorc valori.

Datele membre

În definiția clase `Time`, cele trei date membre, `hour`, `minute` şi `second`, apar după specificatorul de acces `private`. Acesta indică faptul că cei trei membri sunt accesibili doar funcțiilor membre sau, aşa cum vom vedea într-unul dintre capitolele următoare, „prietenilor” clasei. Datele membre poti fi accesate doar de cele patru

funcții membre care apar în definiția clasei. Se obişnuieşte ca datele membre să fie listate în secțiunea `private` a unei clase şi funcțiile membre în secțiunea `public` a clasei. Vom vedea că putem avea şi funcții private şi date membre publice. Cu toate acestea, este considerată o greşeală de proiectare a unei aplicații orientate pe obiecte folosirea datelor membre publice.

Obiectele

Un obiect constă dintr-o copie a setului de date membre declarate în clasă. Funcțiile membre sunt partajate între toate obiectele unei clase, neexistând copii ale funcțiilor pentru fiecare obiect în parte.

Odată definită o clasă, aceasta poate fi folosită ca tip de dată pentru declararea obiectelor, tablourilor, pointerilor sau referințelor. Tipul de dată introdus printr-o clasă poate fi utilizat în acelaşi fel ca şi tipurile predefinite. Putem avea într-un program mai multe obiecte dintr-o clasă, aşa cum putem avea mai multe variabile de tip `int`. Programatorul poate să creeze oricâte tipuri de dată are nevoie într-un program. Din acest motiv, limbajul C++ se spune că este un *limbaj extensibil*.

Exemplu

```
Time sunset,                //obiect de tip Time
     arrayOfTimes[5],       //tablou de obiecte Time
     *pointerToTime,        //pointer la un obiect de tip Time
     &dinnerTime = sunset;  //referinta la un obiect Time
```

În programul următor vom folosi clasa `Time` pentru a înlocui tipul de dată definit prin structura `Time`. Vom vedea cum se definesc funcțiile membre, cum se declară (instanțiază) obiecte ale clasei `Time` şi cum se apelează funcțiile membre.

Exemplu

```
#include <iostream>
using std::cout;
using std::endl;

//Declaratia tipului abstract de data Time
class Time
{
  public:
    Time();                          //constructor
    void setTime(int, int, int); //asignarea valorilor
    void printShort();           //tiparire in format scurt
    void printLong ();           //tiparire in format lung
  private:
    int hour;    //0-23
    int minute;  //0-59
    int second;  //0-59
};

//Constructorul clasei Time initializeaza
//toate datele membre cu valoarea 0.
//Ne asiguram astfel ca orice obiect de tip
//Time porneste dintr-o stare consistenta.
Time::Time()
{
```

```
  hour = minute = second = 0;
}

//Asignarea unor noi valori datelor membre din Time
//Este verificata validitatea datelor
//Valorile invalide sunt inlocuite cu 0
void Time::setTime(int h, int m, int s)
{
  hour = (h >= 0 && h < 24) ? h : 0;
  minute = (m >= 0 && m < 60) ? m : 0;
  second = (s >= 0 && s < 60) ? s : 0;
}

void Time::printShort()
{
  cout << (hour < 10 ? "0" : "") << hour << ":"
       << (minute < 10 ? "0" : "") << minute;
}

void Time::printLong()
{
  cout << ((hour == 0 || hour == 12) ?
           12 : hour % 12)
       << ":" << (minute < 10 ? "0" : "") << minute
       << ":" << (second < 10 ? "0" : "") << second
       << (hour < 12 ? " AM" : " PM");
}

int main()
{
  Time t; //instantiaza obiectul t de tip Time

  cout << "Valoarea initialia in format scurt este ";
  t.printShort();
  cout << "\nValoarea initialia in format lung este ";
  t.printLong();

  t.setTime(13, 27, 6);
  cout << "\n\nOra in format scurt dupa setTime este ";
  t.printShort();
  cout << "\nOra in format lung dupa setTime este ";
  t.printLong();

  //asignarea unor valori invalide membrilor obiectului
  t.setTime(99, 99, 99);
  cout << "\n\nDupa asignarea valorilor invalide:"
       << "\nOra in format scurt: ";
  t.printShort();
  cout << "\nOra in format lung: ";
  t.printLong();
  cout << endl;
```

```
  return 0;
}
```

Rulând programul, vom obține următorul rezultat:

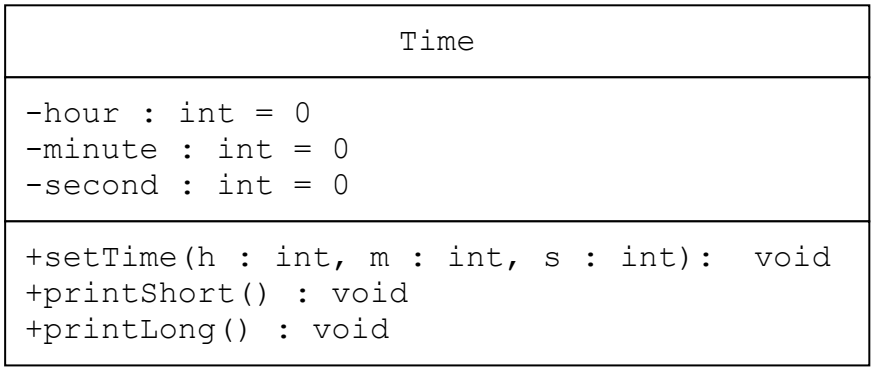
```
Valoarea initialia in format scurt este 00:00
Valoarea initialia in format lung este 12:00:00 AM

Ora in format scurt dupa setTime este 13:27
Ora in format lung dupa setTime este 1:27:06 PM

Dupa asignarea valorilor invalide:
Ora in format scurt: 00:00
Ora in format lung: 12:00:00 AM
```

Programul instanțiază obiectul `t` de tip `Time`. Atunci când este instanțiat acest obiect, constructorul clasei `Time` este apelat automat. Conform definiției acestei funcții, datele membre private `hour`, `minute` şi `second` sunt inițializate cu valoarea `0`. Funcția membră `setTime` asignează valori noi celor trei date membre, iar funcțiile `printShort` şi `printLong` tipăresc ora în format scurt şi în format lung.

Datele membre ale clasei `Time` sunt private, nefiind accesibile din afara clasei. Conform principiilor programării orientate pe obiecte, reprezentarea datelor din interiorul clasei este în afara zonei de interes a clientului clasei. De exemplu, o altă reprezentare a orei ar putea fi sub forma numărului de secunde scurs de la miezul nopții. Clienții ar putea folosi aceleaşi funcții membre publice, obținând aceleaşi rezultate fără să cunoască aceste detalii. Astfel, implementarea unei clase se spune că este *ascunsă* clienților. *Ascunderea informației* reprezintă imposibilitatea de a accesa detaliile de implementare ale unei clase prin metode din afara sa. Ascunderea informației promovează posibilitatea de a modifica programele şi de a simplifica percepția clienților asupra unei clase. Dacă implementarea unei clase se schimbă, de exemplu pentru a îi îmbunătăți performanțele, interfața clasei rămânând neschimbată, programul client nu trebuie modificat. Acest avantaj face ca modificarea unui sistem să fie mult mai uşoară.

În programul de mai sus, constructorul clasei `Time` inițializează datele membre cu valori 0. Acest lucru asigură faptul că obiectele sunt într-o stare consistentă atunci când sunt create. Obiectele de tip `Time` nu pot conține valori inițiate aleatoare invalide deoarece constructorul este apelat automat, conform mecanismelor implementate de orice limbaj de programare orientat pe obiecte. Nici operațiile ulterioare nu pot introduce valori incorecte în datele membre ale obiectelor `Time` pentru că funcția `setTime` a fost scrisă în aşa fel încât verifică încadrarea în domeniile dorite a întregilor care se vor stoca.

Diagrama UML a clasei `Time` este următoarea:



Membrii `private` sunt marcați prin semnul -, iar cei `public` prin +. Valorile inițiale are datelor membre sunt specificate în dreptul fiecăreia dintre ele după semnul =. Datele membre nu pot fi inițializate atunci când sunt declarate în corpul claselor. O astfel de operație produce o eroare de compilare. Datele membre trebuie inițializate de constructorii claselor sau li se pot asigna valori prin funcții de tip „set”.

O funcție care are acelaşi nume cu cel al clasei dar este precedată de caracterul tilda (~) se numeşte *destructor* al clasei. Cu ajutorul lor, programatorul poate implementa operațiile care consideră că trebuie realizate înainte de eliberarea memoriei care a fost alocată obiectului. Destructorii sunt funcții care nu pot avea argumente, deci nu pot fi supraîncărcați.

Adeseori, clasele nu trebuie create de la zero, ci pot fi *derivate* din alte clase care oferă atribute şi comportamente care pot fi refolosite. O altă posibilitate este posibilitatea de a include în noile clase obiecte altor clase ca membri. *Reutilizarea codului* în această manieră poate creşte semnificativ productivitatea în dezvoltarea software. Derivarea noilor clase din clase existente se numeşte *moştenire*, iar includerea obiectelor unei clase ca membri în alte clase se numeşte *compunere* sau *agregare*.

- *Domeniul clasă şi accesarea membrilor unei clase*

Datele şi funcțiile membre ale unei clase aparțin aceluiaşi domeniu, numit *domeniu clasă*.

Funcțiile membre pot fi definite în interiorul definiției clasei, dar se preferă definirea funcțiilor în afara definiției clasei. Atunci când funcțiile membre sunt definite în afara clasei, numele trebuie precedat de numele clasei şi de operatorul domeniu `::`. Acest lucru este necesar pentru că diversele clase definite în program pot conține funcții membre cu aceleaşi nume. Distincția este făcută de clasele cărora le aparțin aceste funcții.

Chiar dacă o funcție membră este declarată în afara clasei, ea face parte tot din domeniul clasei, adică ea poate fi accesată doar de celelalte funcții membre ale clasei sau prin intermediul unui obiect al clasei, o referință la un obiect al clasei sau un pointer la un obiect al clasei.

Funcțiile membre au acces direct la ceilalți membri, fără a fi nevoie să îi apeleze prin intermediul unor obiecte. Acest lucru poate fi observat comparând modul în care sunt scrise funcțiile `printShort` şi `printLong` din clasa `Time`, `care` nu au argumente, şi funcțiile `printShort` şi `printLong` din structura `Time` prezentată în secțiunea precedentă. Programarea orientată pe obiecte simplifică adeseori apelurile de funcții prin reducerea numărului de parametri care trebuie transmişi. Acest beneficiu derivă din faptul că *încapsularea* datelor şi a funcțiilor într-un obiect dă dreptul funcțiilor membre să acceseze în mod direct datele membre.

În interiorul domeniului clasă, toți membrii pot fi accesați de toate funcțiile membre şi pot fi referiți prin nume. În afara domeniului clasei, membrii sunt referiți printr-un obiect, o referință la un obiect sau un pointer la un obiect.

Funcțiile membre ale unei clase pot fi supraîncărcate doar de funcții membre din aceeaşi clasă.

Variabilele definite într-o funcție membră fac parte din *domeniul funcției*, fiind cunoscute doar acelei funcții. Dacă o funcție membră defineşte o variabilă cu acelaşi nume ca una din domeniul clasei, cea din urmă este ascunsă de prima. Într-o astfel de situație, variabila din domeniul clasei poate fi folosită doar dacă numele este precedat de operatorul domeniu `::`.

Operatorii folosiți pentru a accesa membrii unei clase sunt identici cu cei folosiți pentru accesul membrilor unei structuri. Operatorul punct **.** se foloseşte în combinație cu numele unui obiect sau cu cel al unei referințe la un obiect pentru a accesa membrii obiectului. Operatorul săgeată `->` se foloseşte în combinație cu un pointer la un obiect pentru a accesa membrii obiectului.

- ***Separarea interfeței de implementare***

Unul dintre principiile fundamentale ale ingineriei software este separarea interfeței unei clase de implementarea sa. Aceasta face ca programul să poată fi modificat mult mai uşor. De regulă, declarația unei clase se plasează într-un fişier header care va fi inclus de codul client în care urmează să fie folosită clasa. În felul acesta, furnizorii de software pot oferi biblioteci de clase pentru vânzare sau licențiere prin publicarea fişierelor header, implementările fiind sub forma modulelor obiect. Detaliile de implementare rămân astfel protejate, spre deosebire de varianta în care definiția clasei şi implementarea sa se află în acelaşi fişier.

Cu toate acestea, fişirele header pot conține segmente de cod care implementează unele acțiuni sau legături către astfel de segmente de cod. Funcțiile membre de tip `inline` sunt funcții, de preferat de dimensiuni reduse, definite în interiorul clasei, odată cu definiția funcției. Particular funcțiilor `inline` este că, la compilare, apelul lor este înlocuit cu corpul funcției. O funcție definită în afara definiției clasei poate deveni `inline` prin includerea cuvântului cheie `inline` în headerul său. Pe de altă parte, membrii privați sunt listați în fişierul header care conține definiția clasei, ei fiind vizibili utilizatorilor clasei chiar dacă nu le sunt accesibili.

Aplicând principiul separării interfeței de implementare, programul care implementează şi testează clasa `Time` pe care l-am scris mai devreme va fi separat în trei fişiere: `time.h` care conține definiția clasei `Time`, `time.cpp` care conține implementarea funcțiilor membre ale clasei şi `test_time.cpp` care testează clasa. Atunci când scriem un program C++, fiecare definiție a unei clase se plasează într-un fişier header şi definițiile funcțiilor membre ale clasei într-un fişier sursă care are acelaşi nume de bază cu numele fişierului header.

Exemplu

time.h

```
//Previne includerile multiple ale fisierului header
#ifndef TIME_H
#define TIME_H
//Declaratia tipului abstract de data Time
class Time
{
  public:
    Time();    //constructor
    void setTime(int, int, int);   //asignarea valorilor
    void printShort();   //tiparire in format scurt
    void printLong ();   //tiparire in format lung
  private:
    int hour;       //0-23
    int minute;     //0-59
    int second;     //0-59
};
#endif
```

time.cpp

```
#include <iostream>
using std::cout;

#include "time.h"

//Constructorul clasei Time initializeaza
//toate datele membre cu valoarea 0.
//Ne asiguram astfel ca orice obiect de tip
//Time porneste dintr-o stare consistenta.
Time::Time()
{
  hour = minute = second = 0;
}

//Asignarea unor noi valori datelor membre din Time
//Este verificata validitatea datelor
//Valorile invalide sunt inlocuite cu 0
void Time::setTime(int h, int m, int s)
{
  hour = (h >= 0 && h < 24) ? h : 0;
  minute = (m >= 0 && m < 60) ? m : 0;
  second = (s >= 0 && s < 60) ? s : 0;
}

void Time::printShort()
{
  cout << (hour < 10 ? "0" : "") << hour << ":"
       << (minute < 10 ? "0" : "") << minute;
}

void Time::printLong()
{
  cout << ((hour == 0 || hour == 12) ?
         12 : hour % 12)
       << ":" << (minute < 10 ? "0" : "") << minute
       << ":" << (second < 10 ? "0" : "") << second
       << (hour < 12 ? " AM" : " PM");
}
```

test_time.cpp

```
#include <iostream>
using std::cout;
using std::endl;

#include "time.h"

int main()
{
  Time t; //instantiaza obiectul t de tip Time

  cout << "Valoarea initialia in format scurt este ";
```

```
    t.printShort();
    cout << "\nValoarea initialia in format lung este ";
    t.printLong();

    t.setTime(13, 27, 6);
    cout << "\n\nOra in format scurt dupa setTime este ";
    t.printShort();
    cout << "\nOra in format lung dupa setTime este ";
    t.printLong();

    //asignarea unor valori invalide membrilor obiectului
    t.setTime(99, 99, 99);
    cout << "\n\nDupa asignarea valorilor invalide:"
         << "\nOra in format scurt: ";
    t.printShort();
    cout << "\nOra in format lung: ";
    t.printLong();
    cout << endl;

    return 0;
}
```

Definiția clasei este inserată în interiorul următorului cod destinat preprocesorului:

```
//Previne includerile multiple ale fisierului header
#ifndef TIME_H
#define TIME_H
...
#endif
```

În programele de dimensiuni mari, cu multe fişiere header care la rândul lor includ alte fişiere header, pot apărea situații în care unul dintre ele ar putea fi inclus de mai multe ori. Aceste directive de preprocesare se folosesc pentru a preveni includerea multiplă care conduce la erori de compilare.

- *Controlul accesului la membri*

Specificatorii de acces la membrii unei clase, `public`, `private` şi `protected` se folosesc pentru a controla accesul la datele membre şi la funcțiile membre. Specificatorul implicit de acces a membrii unei clase este `private`, astfel că membrii care se găsesc imediat după headerul clasei, în lipsa vreunui specificator sunt privați. În interiorul unei clase, specificatorii `public`, `private` şi `protected` se pot repeta, însă această modalitate de programare nu este uzuală pentru că este o sursă de confuzii.

Membrii `private` ai unei clase pot fi accesați doar de funcțiile membre sau de „prietenii” clasei. Membrii `public` ai unei clase pot fi accesați de orice funcție din program.

Scopul membrilor `public` este de a prezenta clienților clasei *serviciile* (comportamentele) pe care acestea le oferă. Setul de servicii formează interfața publică a clasei. Clienții clasei nu sunt interesați de modul în care sunt implementate aceste funcții.

În exemplul următor demonstrăm că membrii private nu pot fi accesați de codul client, iar compilatorul semnalează acest tip de erori.

Exemplu

test_private.cpp

```
#include <iostream>
using std::cout;

#include "time.h"

int main()
{
  Time t;
  //Eroare: Time::hour nu este accesibil
  t.hour = 7;

  //Eroare: Time::minute nu este accesibil
  cout << "minute=" << t.minute;

  return 0;
}
```

La compilare, apar următoarele erori:

```
C:\Dev-Cpp\Project\PCLPII\C1\Ex4\/time.h:  In  function  `int
main()':

C:\Dev-Cpp\Project\PCLPII\C1\Ex4\/time.h:13:    error:    `int
Time::hour' is private

C:\Dev-Cpp\Project\PCLPII\C1\Ex4\test_private.cpp:11:   error:
within this context

C:\Dev-Cpp\Project\PCLPII\C1\Ex4\/time.h:14:    error:    `int
Time::minute' is private

C:\Dev-Cpp\Project\PCLPII\C1\Ex4\test_private.cpp:14:   error:
within this context
```