6 Clase C++ (II)

6.1 Pointerul implicit `this`

Fiecare obiect al unei clase conține **câte o copie** a datelor membre.

Exemplu:

```
Screen myScreen, bufScreen;
// fiecare are propriul numar de linii (row), de coloane(col),
// *screen *cursor
```

Însă atât `myScreen` cât şi `bufScreen` folosesc **aceeaşi copie** a funcțiilor membre, pentru că funcțiile clasei nu sunt multiplicate pentru fiecare obiect declarat. Există o singură instanță pentru fiecare funcție membru.

Aceste 2 ipoteze creează două probleme:

1) dacă există o singură instanță a unei funcții membre, înseamnă că ea nu este memorată în interiorul clasei. Acest lucru ar însemna multiplicarea ei cu fiecare declarare de obiect nou.

2) dacă există o singură instanță a unei funcții membre, cum se face legătura între datele membre ale clasei şi datele membre asupra cărora se operează în corpul funcției?

Răspunsul este dat de pointerul `this`.

Fiecare funcție membră a unei clase conține un pointer de tipul clasei, care este inclus automat în lista de argumente.

Exemplu:

pentru o funcție membră a clasei `Screen`, `this` este de tipul `Screen*`
pentru o funcție membră a clasei `IntList`, `this` este de tipul `IntList*`

Pointerul `this` conține adresa obiectului prin care a fost făcut apelul de funcție. Pentru o mai bună înțelegere, să urmărim cum tratează compilatorul pointerul `this`:

1) prin compilare, fiecare funcție membră este transformată într-o funcție obişnuită, unică, adăugându-se ca argument pointerul `this`.

Exemplu:

```
   Screen::home()
```

devine

```
home__Screen(Screen *this)
    {
       this->cursor=this->screen;
          }
```

2) Fiecare apel relativ la un obiect este transformat într-un apel obişnuit.

Exemplu:

```
mySreen.home();
```

devine

```
home__Screen( &myScreen );
```

Programatorul poate utiliza explicit pointerul `this`. Există situații în care utilizarea acestui pointer este o necesitate. De exemplu, pointerul `this` este elementul cheie în implementarea sintaxei de concatenare.

Exemplu:

```
Screen ec1(3,3), ec2;
main()
{
   ec1.clear().move(2,2).set('*').display();
   // in loc de ec1.clear();  ec1.move(); ...
   ec2.reSize(5,5).display();
...
}
```

`ec1.clear()` este invocată prima. Pentru ca `move()` să fie invocată corect, `clear`() trebuie să returneze un obiect de tip `Screen`. Pentru ca această concatenare să fie posibilă, fiecare funcție membră implicată trebuie să returneze obiectul invocator, adică `*this`.

Exemplu:

```
// o varianta de implementare a functiei clear
Screen& Screen::clear(char backGround)
{
   // sterge ecranul
   char *p=cursor=screen;
    while(*p) *p++=backGround;
        return *this;
    }
```

Prin intermediul pointerului `this` se poate realiza schimbarea conținutului obiectului invocator. Un asemenea exemplu ilustrează funcția membru `reSize()`.

Exemplu:

```
Screen& Screen::reSize( int r, int c, char backGround)
{
   // redimensioneaza un ecran la r linii si c coloane
        Screen *ps=new Screen(r,c,backGround);
        char *pNew = ps->screen;
        if(screen)
        {
           // daca ecranul curent e alocat, se copiaza continutul vechi in
           // continutul nou
      char *pOld = screen;
           while( *pOld && *pNew )
           *pNew++ = *pOld++;
           delete screen;
        }
        *this=*ps;   // inlocuieste obiectul curent
        return *this;
     }
```

6.2 Prietenii unei clase

În unele situații, regulile de ascundere a informației sunt prea restrictive. Prin intermediul unor relații de tip prietenie (`friend`) se poate asigura accesul unor funcții nemembre la membri nepublici ai unei clase. Înainte de a discuta regulile de declarare a unui prieten, să ilustrăm un exemplu în care apare necesitatea unui prieten.

Exemplu:

Operatorii `iostream`-urilor , `<<` şi `>>`, pot fi supraîncărcaţi ca să trateze şi tipuri de clase. Odata ce operatorii aceştia sunt definiţi pentru o clasă, obiectele de tipul acelei clase vor putea fi afişate la fel ca şi tipurile predefinite.

```
Screen myScreen;
cout<<myScreen;
cout<<"myScreen :"<<myScreen<<"\n";
```

Observație: Operatorii de input/output solicită ca operand stâng un obiect de tip `iostream` şi întorc obiectul `iostream` invocator. Astfel, operația `cout<<s` trebuie implementată ca:

```
ostream& operator<<(ostream&, screen&);
```

Dacă operatorul trebuie să aiba acces la membrii nepublici din `Screen`, există următoarele variante:

1) să fie declarat membru: atunci poate avea un singur parametru, operandul stâng fiind obiectul invocator.

```
class Screen
{
   public: ostream& operator << ( ostream& );
```

```
...
};
```

Operandul stâng fiind obiectul invocator, operatorul trebuie folosit cu următoarea sintaxă:
`myScreen<<cout;`

2) să fie declarat `friend`: prietenii se declară în corpul clasei prin intermediul cuvântului cheie `friend`. Declarațiile `friend` se fac în general imediat după header-ul clasei.

```
class Screen
{
   friend ostream& operator << (ostream&, Screen& );
   public: ...
...
};
ostream& operator << (ostream& os, Screen& s )
{
   os<<"\nLinii=" << s.row << "\nColoane="<<s,col<<'\n';
   char *p=s.screen;
   while(*p) os.put(*p++);
   return os;
}
```

Toate supraîncărcările unei funcții `friend` trebuie declarate explicit `friend`. Dacă o funcție manipulează obiecte din două clase diferite, atunci fie este declarată `friend` în fiecare, fie este membră a uneia şi `friend` pentru cealaltă. O intregă clasă poate fi declarată `friend` (vezi clasa `IntList`).

6.3 Membri statici

Uneori este necesar ca toate obiectele unor clase să aibă acces la aceeaşi dată membră, (aceasta dată poate fi un contor sau un pointer la o funcție pentru tratarea excepțiilor), care nu e necesar să fie multiplicată pentru fiecare obiect. O astfel de variabilă va fi declarată statică. O dată membru statică se comportă ca o variabilă globală la nivelul clasei. Există o singură instanță a unui membru static al unei clase. Un membru este specificat static prin prefixarea declarației sale cu cuvântul cheie `static`. Membrii statici suportă toate regulile de acces: `public`, `protected` sau `private`. În corpul clasei, datele statice sunt doar declarate; ele vor fi definite în exterior.

Exemplu:

```
class CostOp   // operatii cu actiuni, costuri
{
   public:
      CostOp( int, char* );
      double rataLunara();
      void mCost(double incr);
      double getCost() { return costActiune; }
   private:
      static double costActiune;
      int actiuni;
      char *proprietar;
};
```

Declararea lui `costActiune` static are două motive: 1) economie de spatiu şi 2) evitarea greşelilor. Orice obiect de tip `CostOp` trebuie să aibă acces la `costActiune`. Valoarea curentă a variabilei `costActiune` e aceeaşi pentru toate obiectele. În timp, însă, se poate schimba. Din acest motiv, nu am declarat-o constantă. Fiind statică, daca se modifică, toate obiectele vor accesa aceeaşi valoare. Astfel se elimină posibilitatea ca fiecare obiect să poată modifica accidental aceasta valoare.

Datele membre statice se inițializează în afara corpului clasei şi, ca orice variabilă, poate fi inițializată o singură dată. Deci inițializarea nu ar trebui plasată în fişierul header, ci în fişierul `.cpp` care conține definiția funcțiilor membre ne-inline. Accesarea acestor variabile e cea normală.

Exemplu:

```
#include "CostOp.h"
double CostOp::costActiune=23.39;
   // unica pentru toate obiectele, poate fi accesata direct
double CostOp::rataLunara()
{
   return (costActiune*actiune);
}
void CostOp::mCost(double incr)
{
   costActiune += incr;
}
```

Funcțiile membru care accesează doar date statice nu au nevoie de pointerul `this`. Există o singură instanță pentru un membru static. Aceste funcții pot fi declarate statice:

Exemplu:

```
class CostOp
{
   public:
      static void mCost( double incr );
  static double getCost() { return costActiune; }
...
};
```

O funcție membru statică nu conține pointerul `this`, deci orice referire implicită sau explicită la acest pointer generează eroare de compilare (prin referire implicită întelegem încercarea de a referi un membru nestatic). De exemplu, funcția `rataLunara()` nu poate fi declarată statică pentru că citeşte variabila nestatică `actiuni`. O funcție membru statică poate fi invocată prin intermediul unui obiect sau a unui pointer la obiect, ca şi celelalte. Ea mai poate fi invocată direct, specificând sintaxa de membru al unei clase.

Exemplu:

```
f(CostOp& u1, CostOp* u2)
{
   double m1, m2;
   if( CostOp::getCost() == 0 ) return 0;
   m1=u1.getCost();
   m2=y2->getCost();
...
}
```

Observație: Un membru static poate fi accesat sau invocat direct chiar dacă nu au fost declarate obiecte de tipul clasei !

6.4 Pointeri la membrii clasei

Pointerii şi în special pointerii la funcții sunt deosebit de utili în creşterea generalității programelor. Unui pointer la o funcție nu i se poate atribui adresa unei funcții membre chiar dacă tipul returnat şi semnătura celor două se potrivesc exact. Spre deosebire de pointerii obişnuiți, pointerii la membrii claselor au un atribut suplimentar: numele clasei. Deci:

1) trebuie să se potrivească tipul
2) semnătura
3) clasa a carui membru este

Exemplu:

```
   int (*pfi)();
```

Clasa Screen are două funcţii membre care returnează int, fără parametri: int getCol(); şi int getRow();

Fie

```
int getX();
```

Atunci:

```
pfi=getX; // Ok
pfi = Screen::getCol; // eroare
```

Situaţia se repetă şi pentru cazul datelor membre. În clasa Screen avem membrul short col; Tipul unui pointer la col este de fapt un pointer la un membru short al clasei Screen, şi scriem:

```
short Screen::*
typedef short Screen::*pshort;
     pshort ps=&Screen::col;
     ps=&Screen::row;
```

Pointerii la funcţiile membru getCol() şi getRow() se definesc:

```
int (Screen::*)()
int (Screen::*pfi)()=0;
        // initializare, tuturor pointerilor la membrii clasei li se poate
        // atribui 0

     int (Screen::*pfi)()=Screen::getCol;
     pf2=Screen::getRow;
```

Un pointer la funcţiile membru home(), forward(), up(), ... va avea tipul:

```
Screen& (Screen::*)();
```

Pentru simplificarea sintaxei folosim instrucţiunea typedef:

```
typedef Screen& (Screen::*Actiune)():
     Actiune default=Screen::home;   // initializari
     Actiune next=Screen::forward;
```

Accesul pointerilor la membri se face întotdeauna printr-un obiect (operator de selecţie .*) sau printr-un pointer la obiect (operator de selecţie ->*).

Exemplu:

```
     int (Screen ::*pfi)()=Screen::getCol;
     Screen myScreen, *bufScreen=new Screen(10,10);
     // invocare directa a functiei membre
     if( myScreen.getCol() == BufScreen->getCol() )
     { ... }
     // invocare echivalenta prin pointeri la membri
     if( (MyScreen.*pfi)() == (BufScreen->*pfi)() )
     { ... }
```

Analog se tratează cazul pointerilor la datele membre:

```
     ff()
     {
        Screen s, *ps=new Screen(10,10);
        pshort pC=&Screen::col;
     ...
        s.*pC = ps->*pC;
     }
```

Vom scrie acum o funcţie membră pentru clasa Screen. Presupunem că un utilizator al clasei Screen doreşte să repete de mai multe ori o anumită acţiune.

```
     Screen& Screen::repeat( Actiune op, int nr )
     {
```

```
        for(int i=0; i<nr; ++i )
           (this->*op)();
        return *this;
     }
```

Declarația acestei funcții membre poate fi:

```
class Screen
{
   public:
          Screen& repeat( Actiune=Screen::forward, int i=1);
     ...
     };
```

Exemplu:

```
     Screen myScreen;
     myScreen.repeat();      // forward, int=1
     myScreen.repeat(Scree::down,20);
```

Pointerii la membrii statici se comportă ca şi pointerii obişnuiți.

Exemplu:

```
double *pd=&CoOp::costActiune; // nu double CoOp::*pd;
     double (*pf)() = CoOp::getCost(); // nu double CoOp::*pf
```

6.5 Domeniul de accesibilitate la nivel de clasă

Fiecare clasă determină câte un domeniu distinct. O clasă poate fi declarată în interiorul altei clase. Ea se consideră imbricată. Numele clasei imbricate este local clasei care o conține. Clasa imbricată nu este un membru al clasei care îi conține declarația! Funcțiile membre ale clasei imbricate nu au drepturi speciale asupra membrilor clasei care o conțin şi nici funcțiile clasei exetrioare asupra datelor imbricate.

Exemplu:

```
     class E
     {
        int x;
        class I
        {
           int y;
           void f(int i); // definita in afara clasei
           void f1( E *p, int i)
           {
              p->x=i;  // eroare, E::x privat
           }
        };
        int g( I* p)
        {
           return p->y;  // eroare, I::y privat
        }
     };
     void E::I::f(int i)
     {
     ...
     }
```

În acest mod, programatorul poate să indice clasa imbricată (interioară) ca fiind folosită doar de clasa exterioară.

Clasele pot fi definite şi la nivel local, dar în acest caz funcțiile membre trebuie definite în interiorul definiției clasei. În plus, variabilele locale nu sunt cunoscute clasei!

Exemplu:

```
const int bufSize=1024;
void functie()
```

```
{
   const int bufSize=512;
        char *ps=new char[bufSize];     // 512
        class String
        {
           public:
              String() { str=new char[bufSize]; } // 1024
         String& operator=(char *);
   };
   String& String::operator=(char *s) { ... } // ilegal
}
```

Definirea locală a unei clase permite programatorului să limiteze vizibilitatea unei clase la funcția sau blocul în care este definită clasa.

6.6 `Union` - o clasa pentru economisirea memoriei

Un `union` (uniune) este un tip special de clasă. Spațiul de memorie alocat unui union este spațiul necesar celui mai mare membru. Fiecare membru începe la aceeaşi adresă de memorie. La un moment dat, un union poate lucra doar cu un singur membru.

Exemplu:

```
union myValue
{
   int i;
   double d;
   char ch;
} myUnion;
//sizeof(myUnion)=8;
```

`myUnion` poate fi folosit[pentru a stoca 1 `int`, 1 `char` şi 1 `double`, dar numai unul la un moment dat. Valoarea curentă este ultima valoare încarcată.

Observații:

- membrii unei uniuni sunt implicit publici.
- o uniune nu poate conține membri statici.

Exemplu:

```
class Simbol
{
   public:
      char tip;
  myValue val;
  void f();
};
void Simbol::f()
{
   switch(tip)
   {
      case 'c': val.ch='x'; break;
      case 'd': val.d=3.14; break;
      case 'i': val.i=16;   break;
        }
     ...
     }
```

6.7 Câmp de biți - membru pentru economisirea memoriei

Câmpurile de biți sunt date membre speciale, de tip întreg (cu sau fără semn) reprezentate pe un număr de poziții specificat de utilizator. În măsura în care este posibil, câmpurile de biți consecutive sunt compactate în acelaşi întreg, realizându-se astfel o economie de spațiu.

Exemplu:

```
typedef unsigned int Bit;
class File
{
   public: // ...
   private:
      Bit mode:2;
      // identificatorul campului pe biti e urmat de ":" si de o
      // expresie constanta specificand numarul de biti.
      Bit modified:1;
      Bit protGrup:3;
      Bit protOwner:3;
      Bit protAll:3;
// ...
};
```

Câmpurile de biți se accesează la fel ca orice variabilă dar nu li se poate aplica operatorul & (adresă). Deci nu pot exista pointeri la nivel de biți.

Exemplu:

```
File::write()
{
   modified=1;
...
}
File::close()
{
   if(modified)
   // salveaza fisier
}
```

Iată şi un exemplu de folosire a câmpurilor de biți mai mari decât 1 bit:

```
enum {READ=01, WRITE=02}; // mod acces la fisier
main()
{
   File myFile;
   myFile.mode |= READ;
   if(myFile.mode & READ )
      cout<<"\n Acces la fisier Read/Only";
}
```