5 Clase C++

Prin intermediul mecanismului claselor, C++ oferă programatorului posibilitatea de a-şi defini propriile tipuri de date. Clasele se folosesc în mod tipic pentru a defini date abstracte care nu au un corespondent în setul predefinit sau derivat de tipuri de date. Tipurile de clase ce pot fi implementate nu sunt limitate.

Fiecărei clase i se asociază urmatoarele atribute:

1) *Reprezentarea clasei* (datele membre) este o colecție de zero sau mai multe date de orice tip.

2) *Interfața clasei* (funcțiile membre) este o colecție de operații aplicabile obiectelor clasei.

3) *Nivelele de acces* din exteriorul clasei la membrii acesteia. Există trei nivele de acces: `private`, `protected` şi `public`. În general, reprezentarea este privată iar interfața publică. Despre o reprezentare declarată `private` sau `protected` spunem că este încapsulată. Prin intermediul nivelelor de acces se realizează o *ascundere a informației*.

4) *Specificatorul de tip* (numele clasei respective) poate apărea oriunde pot apărea specificatorii de tip predefiniti.

Exemplu:

```
Screen myScreen;
Screen *tempScreen = &myScreen;
Screen& copy(const Screen[]);
```

O clasă cu o reprezentare încapsulată şi cu un set public de operații se numeste *tip abstract*.

5.1 Definirea claselor

O definiție a unei clase are două părți:

- antetul clasei
- corpul clasei

Antetul este format din cuvântul cheie `class` urmat de numele clasei. *Corpul clasei* este cuprins între `{}` şi conține *membrii clasei*, urmat de `;` sau de o *listă de declarare*.

Exemplu:

```
class Screen { /* ... */ };
class Screen { /* ... */ } ecranA, ecranB;
```

Datele membre se declară exact ca variabilele C obişnuite, exceptând faptul că nu pot fi inițializate explicit.

Exemplu:

```
class Screen
{
   short row, col;         // nr linii si coloane
   char *cursor;           // pozitia curenta a cursorului
   char *screen;           // matricea ecran, row*col
};
```

Observație: Este bine ca datele membre să se specifice în ordinea crescătoare a dimensiunii, acest fapt putând asigura o alocare optimă pe toate maşinile.

Un obiect al unei clase poate fi declarat membru doar dacă clasa a fost deja definită. Definirea se încheie în momentul apariției parantezei `}`. Nu se pot declara membri de tipul clasei. Prin intermediul declarațiilor forward se pot folosi pointeri la clase înca nedefinite. Antetul clasei se consideră declarație forward.

Exemplu:

```
class Screen;           // declaratie forward
class StackScreen       // lista de ecrane
{
   int topStack;
   Screen *stack;
};
```

Funcțiile membre se declară în corpul clasei (declararea însemnând specificarea prototipului). Funcțiile foarte scurte pot fi chiar definite, dar sunt considerate implicit `inline`.

Exemplu:

```
class Screen
{
   public:
     void Home(); {cursor=screen;}
     void Move( int, int);
     char Get() { return *cursor; }
     char Get(int, int);
     void CheckRange(int, int);
};
```

Pentru definirea funcțiilor membre în exteriorul clasei se va indica faptul că funcția este membră a clasei respective.

Exemplu:

```
void Screen::CheckRange(int row, int col)
//valideaza coordonatele
{
      if( row < 1 || row > height ||
            col < 1 || col > width)
      {
            cerr << "Coordonatele ecran (" << row << "," << col
                  << ") in afara limitelor." << endl;
            exit(-1);
      }
}
void Screen::Move(int row, int col)
//deplasarea absoluta a cursorului
{
      CheckRange(row, col); //validarea noii pozitii
      int rowMove = (row - 1) * width;
      cursor = screen + rowMove + col - 1;
}
```

Funcțiile membre se deosebesc de funcțiile obişnuite prin faptul că:

(i) au acces la toți membrii clasei (cele obişnuite doar la membrii publici)

Observatie: Funcțiile membre ale unei clase, în general, nu au acces privilegiat la membrii altei clase.

(ii) funcțiile membre sunt definite în domeniul clasei respective, funcțiile obişnuite în domeniul global (fişier).

Funcțiile membre pot fi supraîncărcate doar de funcții membre ale aceleiaşi clase (pentru că supraîncărcarea presupune un domeniu comun).

Exemplu:

```
char Screen::Get(int row, int col)
{
      Move(row, col); //pozitioneaza cursorul
      return Get();  //characterul aflat la noua pozitie a cursorului
}
```

Ascunderea informației este un mecanism formal pentru a restrânge accesul la reprezentarea clasei.

Un membru public este accesibil din orice punct al programului. Un membru `protected` se comportă ca un membru `public` pentru clasele derivate şi ca unul `private` pentru restul programului. Un membru `private` este accesibil doar funcțiilor membre.

Exemplu:

```
class Screen
{
   public:
      void Home() { Move(1,1); }
      char Get()  { return *cursor; }
      char Get(int, int);
      inline void Move(int, int);
...
   private:
      short row, col;
      char *cursor, *screen;
};
```

O clasa poate conține secțiuni multiple: `public`, `protected` sau `private`. Fiecare secțiune rămâne activă până la specificarea unui alt nivel de acces. Nivelul implicit imediat după `{` este `private`. Prin convenție, secțiunile publice se plasează la începutul, iar cele private spre sfârşitul corpului clasei.

5.2 Obiecte

Definirea claselor nu implică rezervări de memorie. Acestea vor fi făcute în momentul definirii obiectelor respective.

Exemplu:

```
Screen myScreen; // se aloca spatiu pentru 4 date membre
```

Obiectele aceleiaşi clase pot fi inițializate, atribuite între ele, pot fi transmise prin valoare şi pot fi returnate funcțiilor. Un pointer la un obiect poate fi inițializat (sau care se poate atribui) fie prin operatorul `new` fie prin operatorul adresă &.

Exemplu:

```
Screen ecranA=ecranB;   // implicit:
                        // ecranA.row=ecranB.row;
                        // ecranA.col=ecranB.col;
                        // ecranA.cursor=ecranB.cursor;
                        // ecranA.screen=ecranB.screen;
Screen *ptr=new Screen;
```

În exteriorul domeniului clasei, membrii sunt accesați prin selectorii `.` şi `->` (`.` cu obiectul sau referința, `->` cu pointer la obiect).

Exemplu:

```
#include "screen.h"
isEqual( Screen& s1, Screen *s2 )
{
   // returneaza 0 daca nu sunt egale si 1 in rest
   if( s1.GetRow() != s2->GetRow() || s1.GetCol() != s2.GetCol() )
      return 0;     // nu sunt egale
   for( int i=0; i<s1.GetRow(); ++i )
      for( int j=0; j<s2->GetCol(); ++j )
         if( s1.Get(i,j) != s2->Get(i,j) )
            return 0;
   return 1;
}
```

Funcția `isEqual` nu are nici un drept de acces la datele clasei `Screen`, motiv pentru care foloseşte funcțiile de acces `GetRow()` şi `GetCol()`.

Exemplu:

```
// sa definim functiile de acces
class Screen
{
```

```
   public:
      int GetRow() { return row; }
      int GetCol() { return col; }
...
   private:
      short row, col;
};
```

Membrii clasei pot fi accesaţi direct din domeniul clasei fără utilizarea operatorilor de selectie.

Exemplu:

```
#include <string.h>
void Screen::Copy(Screen& s)
//copiaza un obiect screen in alt obiect de tip screen
{
      delete screen;
      height = s.height;
      width = s.width;
      screen = cursor = new char[ height * width + 1];
      strcpy( screen, s.screen );
}
```

Observaţie: Funcţia de mai sus nu tratează cazul în care obiectele de tip `Screen` au dimensiuni diferite.

5.3 Funcţii membre

Succesul sau eşecul unei clase depinde de eficienţa şi completitudinea operaţiilor puse la dispoziţia programatorului prin intermediul funcţiilor membre (interfaţa publică).

Exista 4 categorii de funcţii membre:

- de administrare
- de implementare
- auxiliare
- de acces.

Funcţiile de administrare sunt folosite pentru activitatea de iniţializare, asignare, alocare a memoriei şi pentru conversii de tip. De obicei, aceste funcţii sunt automat invocate de către compilator.

Funcţia de initializare se numeşte *constructor* (are acelaşi nume ca şi numele clasei) şi este apelată automat ori de câte ori se defineşte sau se alocă (cu operatorul `new`) un obiect al clasei.

Exemplu:

```
Screen::Screen(int high, int wid, char background)
//constructor: functie de initializare
//a obiectelor de tip Screen
{
      int size = high * wid;
      height = high;
      width = wid;
      cursor = screen = new char[size + 1];
      char *startAddr = screen;
      char *endAddr = screen + size;
      while(startAddr != endAddr)
            *startAddr++ = background;
      *startAddr = '\0'; //sfarsitul ferestrei este marcat prin NULL
}
```

În clasă, constructorul va fi declarat

```
class Screen
{
   public: Screen( int=8, int=40, char='#' );
....
};
```

Declarații de obiecte de tip `Screen` pot fi:

```
Screen s1; // Screen( 8, 40, '#');
Screen *ps = new Screen(20);    // Screen(20, 40, '#');
main()
{
   Screen s(20,80,'*'); // Screen(20, 80, '*');
...
}
```

Funcțiile de implementare furnizează facilitățile tipului abstract. De exemplu, pentru clasa `Screen` acestea pot fi deplasări de cursor (`Home()`, `Move()`, `Forward()`, `Back()` (cu wrap_around), `Up()`, `Down()` (cu bell), `Bottom()`, `Top()`) sau căutări de text (`Find("text")`).

```
void Screen::Forward()
// avans in urmatoarea pozitie cu wrap_around
{
      ++cursor;
      if(*cursor == '\0')
            Home();
}
const char BELL = '\007';
void Screen::Up()
// avans o linie in sus, cu BELL
{
      if(Row() == 1)
            cout.put(BELL);
      else
            cursor -= width;
}
```

Funcțiile auxiliare nu sunt scrise pentru a fi apelate în mod normal, ci doar pentru a simplifica celelalte funcții. De cele mai multe ori sunt private.

Exemplu: pentru clasa `Screen`, funcții auxiliare pot fi `CheckRange()` - verificare coordonate, `Row()`, `Col()` - determinarea liniei şi coloanei curente, `RemainingSpace()` - calculul spatiului disponibil.

```
int Screen::Col()
// returneaza coloana curenta
{
      int position = cursor - screen + 1;
      return ((position + width - 1) % width) + 1;
}
int Screen::RemainingSpace()
// spatiul disponibil, exclusiv pozitia cursorului
{
      int size = width * height;
      return (screen + size - cursor - 1);
}
```

Funcțiile de acces permit accesul (strict controlat) la datele încapsulate. În general, ele sunt doar funcții de citire sau de scriere. Funcțiile de acces sunt FOARTE IMPORTANTE. Dacă apar erori, domeniul de căutare a erorii se limitează la aceste funcții.

Exemplu:

```
void Screen::Set(char *s)
// scrie un sir de caractere in pozitia curenta
{
      int space = RemainingSpace();
      int length = strlen(s);
      if(space < length)
      {
            cerr << "Atentie : trunchiere!" << endl
                  << "Spatiu disponibil : " << space << endl
                  << "Lungimea sirului : " << length << endl;
            length = space;
      }
      for(int i = 0; i < length; i++)
            *cursor++ = *s++;
}
```

sau

```
void Screen::Set(char ch)
// scrie un caracter la pozitia curenta
{
      if(ch == '\0')
            cerr << "Atentie!" << endl
                  << "Caracterul NULL este ignorat" << endl;
      else
            *cursor = ch;
}
```

Alte funcții de acces pentru clasa `Screen` ar putea fi:

```
IsEqual( char ch );     // verifica caracterul din pozitia curenta
IsEqual( char *s );     // verifica sirul din pozitia curenta
IsEqual( Screen& s);    // compara doua ecrane
```

Exemplu:

```
Screen::IsEqual(Screen& s)
{
   if( row!=s.row || col!=s.col )  // diferite
      return 0;
   char *p=screen;
   char *q=s.screen;
   if(p==q)        // adreseaza aceeasi zona de memorie
      return 1;
   while( *p && *p++ = *q++ );
   if( *p ) return 0; // testul de egalitate nu a ajuns la sfarsitul
                      // ecranului
   return 1;
}
```

În cazul manipulării obiectelor constante, compilatorului trebuie sa i se specifice dacă o funcție modifică sau nu obiectul respectiv. Am intâlnit până acum situații de forma:

```
const char blank=' ';
blank = '\0';   // eroare
```

Dar, prin definiția lui, un obiect nu poate fi direct modificat de un programator, ci doar prin invocarea setului de funcții care fac modificări asupra datelor acestuia.

Ce se întâmplă însă în cazul apelurilor:

```
const Screen blankScreen;
blankScreen.Display();  // sigura
blankScreen.Set('*');   // nesigura
```

Proiectantul clasei poate specifica faptul că o funcție membră nu modifică obiectul declarând-o şi definind-o `const`. Cuvântul `const` se plaseaza între lista de argumente şi corpul funcției. Pentru obiecte constante pot fi invocate doar funcțiile constante.

Exemplu:

```
class Screen
{
   public:
      char Get() const { return *cursor; }
      IsEqual( char ch ) const;
      void Ok( char ch) const { *cursor=ch; }
      // nu modifica cursor ci valoarea obiectului pe care il
      // adreseaza cursor
      void Er( char *pCh) { cursor=pCh; }
      // modifica datele membre
...
};
Screen::IsEqual(char ch) const
{
   return (ch==*cursor)
}
```

Funcțiile `const` şi non-`const` pot fi supraîncărcate. Ambiguitatea se elimină prin declarația `const` sau non-`const` a obiectului implicat.

Exemplu:

```
class Screen
{
   public:
      char Get( int x, int y);
      char Get( int x, int y) const;
...
};
```

Constructorii şi destructorii fac excepție de la aceste reguli. Ei nu trebuie declarați `const` pentru a putea fi aplicați obiectelor constante. În general, orice clasă care e intens folosită trebuie să asigure posibilitatea de lucru cu obiecte constante.

5 Clase C++ - Anexă

Programul de mai jos implementează şi exemplifică folosirea clasei `Screen`. Fişierul `Screen.h` conştine definiția clasei, `Screen.cpp` conține implementările funcțiilor membre ale clasei, iar `ScreenMain.cpp` este un exemplu de folosire a acestei clase.

Screen.h

```
#include <iostream.h>

class Screen
{
public:
 Screen(int = 8, int = 40, char = '#');
 void Home()
 {
       cursor = screen;
 }
 void CheckRange(int, int);
 void Move(int, int);
 char Get();
 char Get(int, int);
 int GetRow();
 int GetCol();
 int GetHeight();
 int GetWidth();
 void Copy(Screen&);
 void Forward();
 void Back();
 void Bottom();
 void Up();
 void Down();
 int RemainingSpace();
 int Row();
 int Col();
 void Set(char*);
 void Set(char);
 void Display();
private:
 short height; //numarul de linii
 short width;  //numarul de coloane
 char *cursor; //pozitia curenta pe ecran
 char *screen; //matricea ecran (height*width)
};
```

Screen.cpp

```
#include "Screen.h"
#include <string.h>
#include <stdlib.h>

Screen::Screen(int high, int wid, char background)
//constructor: functie de initializare
//a obiectelor de tip Screen
{
 int size = high * wid;
 height = high;
 width = wid;
 cursor = screen = new char[size + 1];
 char *startAddr = screen;
 char *endAddr = screen + size;
 while(startAddr != endAddr)
       *startAddr++ = background;
```

```
 *startAddr = '\0'; //sfarsitul ferestrei este marcat prin NULL
}

void Screen::CheckRange(int row, int col)
//valideaza coordonatele
{
 if( row < 1 || row > height ||
       col < 1 || col > width)
 {
       cerr << "Coordonatele ecran (" << row << "," << col
              << ") in afara limitelor." << endl;
       exit(-1);
 }
}

void Screen::Move(int row, int col)
//deplasarea absoluta a cursorului
{
 CheckRange(row, col); //validarea noii pozitii
 int rowMove = (row - 1) * width;
 cursor = screen + rowMove + col - 1;
}

char Screen::Get()
{
 return *cursor;
}

char Screen::Get(int row, int col)
{
 Move(row, col); //pozitioneaza cursorul
 return Get();   //characterul aflat la noua pozitie a cursorului
}


int Screen::GetHeight()
{
 return height;
}

int Screen::GetWidth()
{
 return width;
}

void Screen::Copy(Screen& s)
//copiaza un obiect screen in alt obiect de tip screen
{
 delete screen;
 height = s.height;
 width = s.width;
 screen = cursor = new char[ height * width + 1];
 strcpy( screen, s.screen );
}

void Screen::Forward()
// avans in urmatoarea pozitie cu wrap_around
{
 ++cursor;
 if(*cursor == '\0')
       Home();
}

void Screen::Back()
```

```
//mutarea cursorului pe pozitia anterioara
{
//implementati aceasta functie
}

void Screen::Bottom()
{
 int size = width * height - 1;
 cursor = screen + size;
}

const char BELL = '\007';

void Screen::Up()
// avans o linie in sus, cu BELL
{
//implementati aceasta functie
}

void Screen::Down()
{
 if(Row() == height)
       cout.put(BELL);
 else
       cursor += width;
}

int Screen::Row()
{
 int position = cursor - screen + 1;
 return (position + width - 1) / width;
}

int Screen::Col()
// returneaza coloana curenta
{
 int position = cursor - screen + 1;
 return ((position + width - 1) % width) + 1;
}

int Screen::RemainingSpace()
// spatiul disponibil, exclusiv pozitia cursorului
{
 int size = width * height;
 return (screen + size - cursor - 1);
}

void Screen::Set(char *s)
// scrie un sir de caractere in pozitia curenta
{
 int space = RemainingSpace();
 int length = strlen(s);
 if(space < length)
 {
       cerr << "Atentie : trunchiere!" << endl
              << "Spatiu disponibil : " << space << endl
              << "Lungimea sirului : " << length << endl;
       length = space;
 }
 for(int i = 0; i < length; i++)
       *cursor++ = *s++;
}

void Screen::Set(char ch)
```

```
// scrie un caracter la pozitia curenta
{
 if(ch == '\0')
       cerr << "Atentie!" << endl
              << "Caracterul NULL este ignorat" << endl;
 else
       *cursor = ch;
}

void Screen::Display()
{
 for(int i = 1; i <= height; i++)
 {
       for(int j = 1; j<= width; j++)
             cout << Get(i,j);
       cout << endl;
 }
}
```

ScreenMain.cpp

```
#include "Screen.h"

int main()
{
 Screen x(3,7,'*');
 x.Move(2,1);
 x.Set('#');
 x.Move(2,2);
 x.Set("Salut");
 x.Set('#');
 x.Display();
 x.Down();
 return 0;
}
```