7 Funcții de administrare

În acest capitol vom detalia următoarele trei categorii de funcții de administrare:

1) *constructorii* şi *destructorii*, destinați inițializării/deinițializării obiectelor;

2) *supraîncărcarea unor operatori* ca o alternativă la realizarea aceloraşi operatii prin funcții;

Exemplu:

În loc de invocarea explicită a functței membre

```
if( myScreen.isEqual(yourScreen) )
```

să fie posibilă invocarea echivalentă

```
if( myScreen == yourScreen )
```

În plus, o clasă poate să-şi promoveze propriul stil de administrare a memoriei, definind instanțe ale operatorilor `new` şi `delete`.

3) *operatorii de conversie* care definesc tipurile de conversii permise pentru o clasă.

Invocarea acestor funcții membre este, în general, transparentă utilizatorului clasei. Rolul lor este de a aduce sintaxa de utilizare a unei clase la nivelul de naturalețe al tipurilor predefinite.

7.1 Inițializarea obiectelor

Inițializarea unui obiect constă în inițializarea membrilor. Dacă aceştia sunt publici, ei pot fi inițializați explicit.

Exemplu:

```
class Word
{
   public:
      int occurs;
      char *string;
};
Word cuvant = {0, "test"};
```

C++ suportă un mecanism de inițializare automată a obiectelor. O funcție membră specială, numită *constructor*, este apelata automat de către compilator ori de câte ori se defineşte un obiect sau se alocă cu operatorul `new`. Numele constructorului este numele clasei.

Exemplu:

```
class Word
{
   public:
      Word( char*, int=0);
   private:
      int occurs;
      char *string;
};

#include<string.h>
Word::Word(char *sir, int cnt)
{
   string = new char[strlen(sir)+1];
   strcpy(string, str);
   occurs=cnt;
}
```

Iată câteva definiri de obiecte `Word` în prezența constructorului:

```
Word w=Word("test",5);
Word *pW=new Word("test");
// Notatii, prescurtari
```

```
Word w("test"); // Word::Word("test",0);
Word w="test"; // Word::WOrd("test",0);
```

Observație:

- constructorii nu au tip şi nu returnează nimic, în rest, ei nu sunt decât nişte funcții membre obişnuite;

- constructorii pot fi supraîncărcați.

Unul dintre cele mai utilizate tipuri de date este tipul şir de caractere. Pentru a ilustra semantica constructorilor, destructorilor şi a operatorilor supraîncărcați, vom construi o clasă pentru acest tip.

Exemplu:

```
class String
{
   public:
      String(int);
      String(char*);
   private:
      int len;
      char *str;
};
```

Observație:

- discutați existența membrului `len`: este nevoie de lungime destul de multe ori încât sa fie mai avantajoasa memorarea ei într-un membru decât calculul ei de fiecare dată?

- clasa poate fi folosita nu numai pentru lucrul cu siruri ci şi ca buffer - e de o lungime specificată.

```
String::String(char *s)
{
   len=strlen(s);
   str=new char[len+1];
   strcpy(str,s);
}
String::String(int ln)
{
   len=ln;
   str=new char[len+1];
   str[0]='\0';
}
```

În momentul definirii, fiecărui obiect i se alocă spațiul pentru datele membre nestatice. Constructorii inițializează aceste date. Se pot transmite argumente constructorului fie în forma explicită, fie în forma prescurtată.

Exemplu:

```
String searchWord=String("test");  // invocare explicita
String commonWord("cuvant");       // forma prescurtata 1
String inBuf=1024;                 // forma prescurtata 2
String *ptrBuf=new String(1024);   // new necesita forma explicita
```

Dacă prin `new` nu se poate aloca spațiul necesar datelor membre, atunci constructorul nu mai este apelat, iar pointerul la clasă este setat `0`.

Observație:

- este util să permitem definirea unui obiect `String` fără să cerem specificarea unor argumente.

```
String temStr;
```

Acest lucru poate fi făcut prin intermediul unui constructor implicit (a nu se confunda cu constructorii care au argumente implicite !). Constructorul implicit este constructorul cu lista de argumente vidă.

Exemplu:
```
String::String()        // constructor implicit
{
   len=0;
   st=0;
}
```

O greseala frecventă de programare este:
```
String st();
```
Observație:

- această declarație nu reprezintă definiția unui obiect `String` inițializat cu constructor implicit, ci reprezintă declarația unei funcții `st` care nu are argumente şi returnează un obiect de tip `String`.
- definiții corecte sunt:
```
String st;          // declaratii echivalente, se apeleaza
String st=String(); // constructorul implicit
```

Constructorii suportă toate nivelele de acces definite pentru o clasă. O clasă fără nici un constructor public se numeşte clasă privată (ex `IntItem`).

Destructorii sunt funcții membre speciale invocate ori de câte ori un obiect încetează să mai existe (ex. expirarea duratei de viață, aplicarea operatorului `delete` unui pointer de tipul clasei). Destructorii nu au argumente, nu returnează nimic şi numele lor este cel al clasei precedat de caracterul ~.

Exemplu:
```
class String
{
   public:
      ~String();
...
};
String::~String() { delete str; }
```

Destructorii nu acționează asupra spațiului alocat datelor membre nestatice ale obiectului. Pointerii şi referințele nu provoacă invocarea destructorilor în momentul în care işi încetează existența. Programatorul trebuie să invoce explicit operatorul `delete`. `Delete` invocă destructorul obiectului adresat de pointerul furnizat ca argument. Valoarea acestui pointer trebuie să fi fost obținută în urma unui apel de operator `new`.

Exemplu:
```
#include "String.h"
String cuvant("test");
f(){
   String *p = &cuvant;
   String *p2 = new String("sanie");
...
   delete p; // eronat ca idee, rezultat imprevizibil
   delete p2; // corect
}
```

Dacă pointerul căruia îi este aplicat `delete` este 0, deci nu adresează un obiect, destructorul nu e invocat. Deci nu e necesar să scriem:
```
if(p2 != 0)
   delete p2;
```

Pe scurt, destructorii pot efectua toate operațiile pe care programatorul le vrea efectuate înainte de încetarea existenței unui obiect.

Tablourile de obiecte se definesc la fel ca şi cele de tipuri predefinite.

Exemplu:

```
const int size=16;
String tab[size];
String *tab2 = new String[size];
   // ambele variante definesc un tablou de 16 obiecte String
```

Tablourile pot fi inițializate apelând explicit sau prescurtat constructorii respectivi.

```
String aS1[] = { "schi", "zapada" };
String aS2[] = { String(), String(1024), String("test") };
String aS3[] = { 1024, String(512) };
```

Dacă lista de inițializare este mai mică decât dimensiunea tabloului, atunci pentru elementele rămase neinițializate se apelează constructorul implicit. Tablourile de obiecte alocate dinamic nu pot fi inițializate explicit. În acest caz este invocat constructorul implicit (deci clasa trebuie să aiba constructor implicit). Pentru eliberarea memoriei alocate trebuie apelat explicit operatorul `delete`.

```
delete tab2;
   // nu e sufiecient, sterge doar primul element din tab2
```

Programatorul trebuie sa specifice dimensiunea tabloului

```
delete [size] tab2;
   // destructorul va fi invocat pentru fiecare din cele
   // size elemente ale tabloului tab2;
```

Dacă datele membre sunt obiecte ale altei clase:

- constructorii sunt apelați începând cu al celui mai interior obiect, în ordinea declarării acestora, terminând cu cel al clasei obiectului exterior (care conține datele membre).

Exemplu:

```
class Word
{
   public:
      Word();
      Word( char*, int=0 );
      Word( String&, int=0 );
   private:
      int occurs;
      String name;
};
```

Constructorilor datelor membre li se pot transmite argumente prin listele de inițializare a membrilor.

```
Word::Word(char *s, int cnt):name(s)
{
   occurs=cnt;
}
```

Observație:

- listele de inițializare apar doar în definițiile constructorilor, nu şi în declarațiile lor;
- în listele de inițializare pot apare şi tipuri de date predefinite;

```
Word::Word(char *s, int cnt): name(s), occurs(cnt)
{
}
```

- listele de inițializare sunt singura modalitate de inițializare a datelor membre `const` sau referință.

Execuția constructorilor cuprinde două faze:

- inițializare conform listei specificate;
- asignare conform corpului functiei.

Dacă constructorul unui obiect necesită listă de argumente, atunci obiectul respectiv trebuie să apară în lista de inițializare. Argumentul de inițializare al unui membru care este obiect al unei alte clase poate fi un obiect din această clasă.

Exemplu:
```
String mesaj("Hello");
Word Salut(mesaj);
```

În faza de inițializare se vor apela toți constructorii obiectelor membre, în ordinea declarării acestora. Pentru obiectele nespecificate în lista de inițializare se vor invoca constructorii impliciți.
```
class SinAntonim
{
   public:
      SinAntonim(char *s) : wd(s) {}
      SinAntonim(char *s1, char *s2, char *s3)
         : wd(s1), sinonim(s2), antonim(s3) {}
      ~SinAntonim();
   private:
      String sinonim;
      Word wd;
      String antonim;
};
```

Să analizăm urmatoarele două situații:
1) `SinAntonim sa1("test");`
2) `SinAntonim sa2("cauza", "origine", "efect");`

Ordinea de apel a constructorilor este următoarea:
1) pentru membrul
```
String sinonim:   String()
Word wd:          String("test"), Word("test")
String antonim:   String()
```
2) pentru membrul
```
String sinonim:   String("origine");
Word wd:          String("cauza"), Word("cauza")
String antonim:   String("efect")
```
Destructorii sunt apelați în ordinea inversă constructorilor.

7.2 Inițializarea datelor membre

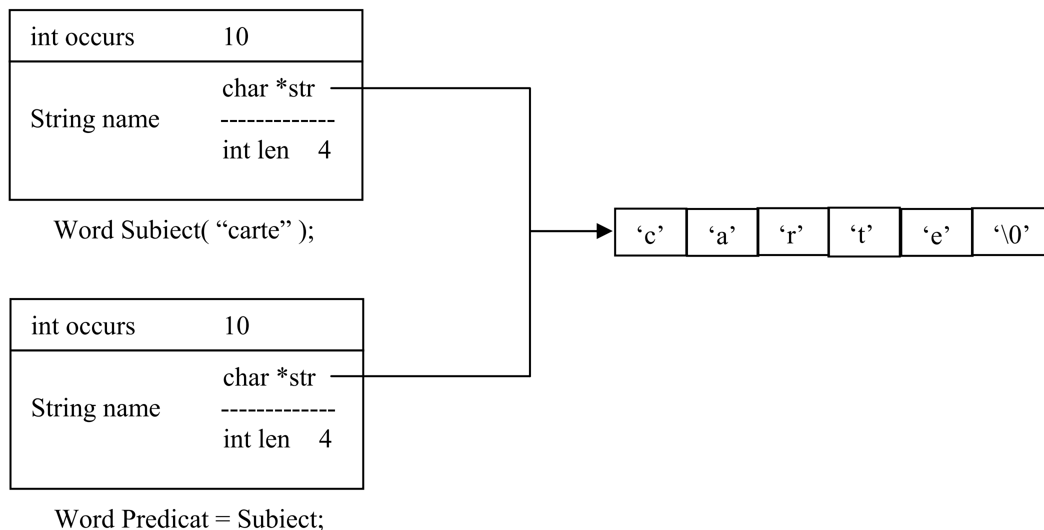
Atunci când un obiect este inițializat cu un obiect din aceeaşi clasă, nu se mai invocă constructorul definit de programator, ci unul implicit, `X::X(const X&)`, numit constructor de copiere, care realizează inițializarea membru cu membru.

Exemplu:
```
// constructorul de copiere implicit al clasei String
String::String(const String& s)
{
   len=s.len;
   str=s.str;
}
```

Acest tip de inițializare mai apare şi în următoarele două situații:
- transmiterea unui obiect ca argument (pentru crearea copiei locale);
- returnarea unui obiect de o funcție.

Observație:
- Inițializarea implicită membru cu membru este însă insificientă în special în cazurile când datele membre sunt pointeri.

```
Word Subiect( "carte" );
Word Predicat = Subiect;
```



Astfel pot apărea probleme serioase dacă obiectele nu există în acelaşi timp şi in acelaşi domeniu de accesibilitate. Programatorul poate adăuga control asupra unor asemenea situații, definind explicit o instanță a constructorului de copiere `X(const X&)`.

Exemplu:

```
String::String( const String& s )
{
   len=s.len;
   str=new char[len+1];
   strcpy(str,s.str);
}
```

Observație:

- dacă o clasă are definit constructorul `X::X(const&)`, atunci trebuie să inițializeze explicit toate obiectele membre, indiferent de tipul lor.

```
Word::Word( const  Word& w)
{
   occurs = 0;
   name = w.name;
}
```

Să analizăm acum pas cu pas ce se întâmplă în următoarea situație:

```
Word examen("scris");
Word test=examen;
```

(i) - `test` este recunoscut ca fiind inițializat cu un obiect de tip `Word`. Există constrcutor de copiere? Dacă da, el este invocat, dacă nu, are loc o inițializare membru cu membru.

(ii) - există listă de inițializare pentru `Word` ? Nu;

(iii) - există obiecte membre? Da, şi anume `String name`;

(iv) - clasa `String` a definit un constructor implicit? Dacă da, el este invocat, dacă nu, se semnalează eroare de compilare;

(v) - `String()` invocat pentru inițializarea lui `test.name`;

(vi) - se invocă `Word(const Word&)`, se execută `name=w.name`.

Observați că `String( const String& )` nu este invocat niciodată !

```
// varianta corecta
Word::Word( const Word& w) : name(w.name)
{
   occurs = w.occurs;
}
```