

Capitolul 4

Spațiul numelor

În cazul proiectelor de mari dimensiuni, gestionarea corectă a numelor din cadrul programelor reprezintă o problemă importantă a activității de programare. Variabilele și obiectele declarate în interiorul blocurilor nu prezintă un pericol deoarece sunt elemente locale și alocarea memoriei pentru ele se face pe stivă. Variabilele globale declarate în afara corpului oricărei funcții, au avantajul de a fi vizibile în corpul tuturor funcțiilor definite în același fișier ca și variabilele, dar au dezavantajul de a putea genera erori de multiplă definire.

În cazul limbajului C apare o problemă datorită faptului că spațiul numelor unui program este global întregului program; cu alte cuvinte, două variabile cu același nume definite în fișiere distincte ale unui program există în același spațiu al numelor și acest lucru poate genera conflicte.

De exemplu, se presupune că un program este dezvoltat de către două echipe de programatori în două module distincte. Dacă o variabilă este declarată în ambele fișiere la nivel de fișier cu același nume, compilatorul va genera un mesaj de eroare (variabilă multiplu definită). Întrucât fiecare echipă de programatori își gestionează propriile nume, trebuie să existe o coordonare la nivelul întregului proiect în ceea ce privește variabilele globale.

O variantă des utilizată a acestui tip de conflicte o constituie variabilele statice. O variabilă statică definită la nivelul de fișier, devine vizibilă doar în cadrul fișierului respectiv și este ascunsă pentru celelalte module ale programului.

În afara acestei tehnici, limbajul C++ permite utilizarea unor *spații de nume* distincte în cadrul acelui program. Astfel spațiul global al numelor unui program poate fi divizat în mai multe spații de nume, folosind facilitatea `namespace`.

4.1 Definirea unui spațiu al numelor

Definirea unui spațiu al numelor se face cu ajutorul următoarei sintaxe (asemănătoare sintaxei pentru `struct`, `class` și `union`):

```
namespace [<identificator>]
{
    <declaratii>
}
```

unde `<identificator>` reprezintă numele asociat spațiului respectiv de nume.

O asemenea definiție produce un nou spațiu al numelor, toate declarațiile dintre acolade fiind locale acestui spațiu.

Exemple:

```
namespace sp1 {
    int a, b;
    struct punct { double x, y; }
}

namespace sp2 {
    double a, b;
    int punct, punct1, punct2;
}

void main {
    // ...
}
```

Se observă faptul că spre deosebire de `struct`, `class` și `union`, o definiție `namespace` nu se încheie cu `;`.

Observatie: O definiție `namespace` poate să apară doar la nivel global, deci în afara definiției oricărei funcții. Este permisă însă includerea unui spațiu al numelor în interiorul altui spațiu. În mod uzual, definițiile `namespace` sunt plasate în cadrul fișierelor header și nu în fișierele de implementare.

Este posibil însă ca o anumită definiție `namespace` să conțină foarte multe declarații, caz în care ea se poate extinde peste mai multe fișiere header. În acest caz unul dintre fișiere reprezintă o definiție `namespace` iar celelalte doar *completări* ale definiției și nu *redefiniri* ale aceluiași spațiu. În mod uzual se folosește tehnica definirii *în cascadă* a fișierelor header: dacă sunt utilizate fișierele header f_1, f_2, \dots, f_k pentru definirea aceluiași spațiu al numelor, atunci în fișierul f_i trebuie inclus fișierul f_{i-1} .

Exemplul 4.1:

```
// fisierul header1.h
#ifndef HEAD1
#define HEAD1
namespace spa // definitia spatiului spa
{
    int m, n;
    int f(int, int);
    // ...
}
#endif

// fisierul header2.h
#ifndef HEAD2
#define HEAD2
```

```

#include "head1.h"
namespace spa // completarea spatiului spa
{
    // nu este eroare
    double x, y, z;
    double g(double);
    // ...
}
#endif

// fisierul header3.h
#ifndef HEAD3
#define HEAD3
#include "head2.h"
namespace spa //completarea spatiului spa
{
    char s1, s2;
    char h(const char *);
    // ...
}
#endif

//fisierul pr.cpp
#include "head3.h"
main()
{
    // ...
}

```

Rezultă astfel faptul că este permisă apariția mai multor construcții namespace ce referă același nume în același fișier: prima reprezintă definiția unui spațiu al numelor, iar celelalte completări ale primului spațiu.

Observatie: În afară de spațiile definite explicit de programatori, fiecare unitate de compilare are asociat în mod implicit un *spațiu al numelor anonim* (fără nume). Acesta este unic pentru o unitate de compilare și variabilele declarate în acest spațiu nu mai trebuie calificate în momentul utilizării lor. Pentru a adăuga variabile la spațiul numelor implicit al unei unități de compilare, în modulul respectiv trebuie adăugate construcții namespace fără nume.

Exemplul 4.2. În fișierul urmator:

```

// fisierul mod1.cpp
namespace // spatiu de nume anonim
{
    class A {
        // ...
    };
    class B {
        // ...
    };
}

```

```

    double p,q;
}
prelucrare()
{
    // ...
}

```

declarațiile claselor *A* și *B* precum și ale variabilelor *p* și *q* aparțin spațiului numelor implicit al fișierului *mod1.cpp*.

Un spațiu anonim de nume este unic pentru fiecare unitate de compilare, astfel încât toate numele din acest spațiu sunt *locale* modulului respectiv, fără să mai fie necesar să fie declarate *static*. Limbajul C++ încurajează utilizarea spațiilor anonime de nume în locul metodei folosirii alocării statice pentru ascunderea numelor în interiorul fișierelor.

Singura operație care se poate efectua asupra unui spațiu al numelor este cea prin care acestuia i se asociază un *alias* (un alt nume peste spațiul respectiv). Sintaxa prin care se asociază un alias unui spațiu al numelor, este următoarea:

```
namespace <nume spatiu1> = <nume spatiu2>;
```

De exemplu:

```

namespace spa1 {
    int a, b, c;
    // ...
}
namespace spa2 = spa1;

```

În exemplul anterior, *spa1* și *spa2* reprezintă același spațiu al numelor (definit inițial de *spa1*).

Un exemplu pertinent de utilizare a unui alias pentru un spațiu al numelor îl reprezintă situația în care se utilizează un spațiu al numelor deja definit dar al cărui nume este prea lung și dificil de utilizat pentru a prefixa utilizarea elementelor definite în cadrul lui.

4.2 Utilizarea spațiului numelor

Pentru a putea utiliza numele definite într-un spațiu al numelor, ele trebuie atașate la spațiul de definiție. Există trei metode de utilizare a acestor nume: folosirea operatorului de rezoluție, a *directivei using*, sau a *declarației using*.

A) Utilizarea operatorului de rezoluție.

Deoarece un spațiu al numelor reprezintă un domeniu de definiție pentru toate declarațiile interne, metoda clasică de specificare a numelor din cadrul acestor spații o reprezintă prefixarea numelor de numele spațiului, cu ajutorul operatorului de rezoluție, ca și în cazul membrilor unei clase.

Exemplul 4.3:

```

// fisierul spa.h
namespace spa {
    class A {
        int n;
        int f();
        // ...
    }
    double x, y;
    class B;
    // ...
}

class spa::B {
    char c1, c2;
    B(char);
    // ...
};

// fisierul pr.cpp
int spa::A::f() { return n; }
spa::B::B(char c) { c1 = c2 = c; }
void prelucrare () {
    spa::x = 7.5;
    // ...
}

```

Dezavantajul acestei metode constă în faptul că toate numele utilizate trebuie prefixate de numele spațiului din care face parte, ceea ce face dificilă scrierea programelor.

B) Utilizarea directivei `using`.

O metodă mult mai simplă constă în importarea tuturor numelor dintr-un spațiu al numelor cu ajutorul unei directive `using`. În acest fel, numele din spațiul respectiv pot fi folosite direct, fără prefixare.

Sintaxa directivei `using` este:

```
using namespace <nume>;
```

Efectul directivei `using` constă în importarea tuturor numelor din spațiul respectiv în domeniul de definiție în care apare directiva. De exemplu, dacă directiva este utilizată la nivelul unui fișier, toate numele din spațiul specificat devin nume globale în fișier, așa cum se observă din secvența următoare:

```

// fisierul pr0.cpp
#include "spa.h"
using namespace spa;
int A::f() {return n;}
B::B(char c) {c1 = c2 = c;}

```

```

void prelucrare () {
    x = 7.5;
    // ...
}

```

Domeniul de vizibilitate al numelor importate cu ajutorul directivei `using` este dat locul de plasare al directivei: la nivel de fișier, sau în cadrul unui anumit bloc. În cazul folosirii directivei `using` în cadrul unui bloc, numele importate din spațiul respectiv devin locale blocului în care există directiva `using`. De exemplu:

```

// fisierul pr1.cpp
#include "spa.h"
void h() {
    using namespace spa;
    x = y = h;
    A a;
    int k = a.f();
    // ...
}

```

Directiva `using` se poate utiliza chiar în interiorul unui alt spațiu al numelor, respectând aceleași reguli ale domeniului de vizibilitate.

Deoarece directiva `using` importă toate numele definite într-un spațiu de nume, este posibil ca un anumit nume să fie redefinit în domeniul de definiție respectiv. Există două situații distincte de redefinire:

- definirea unui obiect cu același nume ca și un nume existent într-un spațiu importat,
- utilizarea a două directive `using` referind două spații de nume ce conțin anumite nume comune.

În primul caz obiectul definit explicit îl acoperă pe cel definit în spațiul numelor. De exemplu:

```

// fisierul pr2.cpp
#include "spa.h"
using namespace spa;
float x = 7; // spa::x este acoperit de x
spa::X = 8.5;
// ...

```

În al doilea caz poate exista pericolul unei coliziuni de nume. Dar ambiguitatea poate apare doar în momentul referirii numelui, nu și al utilizării directivelor `using`. De exemplu:

```

// fisierul spa1.h
#ifndef SPA1
#define SPA1
namespace spa1 {
    int x;
    // ...
}

```

```

// fisierul pr3.cpp
#include "spa.h"
#include "spa1.h"
using namespace spa;
using namespace spa1; //Nu este o ambiguitate la definirea
x = 3;                // Eroare! Ambiguitate
spa1::x = 3.5;       // Corect!

```

B) Utilizarea declarației using.

A treia variantă de utilizare a numelor din cadrul spațiilor de nume o constituie declarația using. Spre deosebire de directiva using, o declarație using nu importă toate numele dintr-un spațiu, ci doar nume individuale.

Sintaxa este următoarea:

```
using <nume spatiu>::<nume>;
```

Se observă că nu trebuie specificat tipul de date asociat numelui respectiv, ci doar spațiul numelor la care aparține.

Ca și în cazul directivei, numele ce apar într-o declarație using nu mai trebuie calificate în domeniul de definiție în care apare declarația.

O declarație using poate apare într-un program exact în același locuri ca o declarație obișnuită. Datorită faptului că este o declarație, ea poate supraîncărca un obiect cu același nume importat dintr-un alt spațiu al numelor cu o directivă using, după cum se observă din secvența următoare:

```

//fisierul pr4.cpp
#include "spa.h"
#include "spa1.h"
void g()
{
    using namespace spa;
    using spa1::X;
    x = 4;                // spa1::X
    spa::X = 4.3;        // trebuie specificat spatiul
    // ...
}

```

Datorită faptului că într-o declarație using se specifică doar numele unui identificator, nu și tipul sau, în cazul funcțiilor supraîncărcate, o singură declarație relativă la numele unei funcții permite încărcarea tuturor funcțiilor cu același nume.

Exemplul 4.4:

```

namespace spa3
{
    void f(int,int);
}

```

```
    double f(double);
    int f(int);
    // ...
}

void spa3::f(int a, int b)
    { cout << a << b; }

double spa3::f(double x)
    { return x * x; }

int spa3::f(int n)
    { return 2 * n; }

void pr4() {
    using spa3::f;
    f(3, 4);
    double y = f(7.5);
    int k = f(2);
    // ...
}
```