10. Scope. Lifetime. Namespace

Pe măsură ce un program devine tot mai complex, numărul de identificatori creşte. Unii identificatori sunt declarați în interiorul blocurilor, alții în afara lor. Vom studia în acest capitol regulile C++ care permit unei funcții să acceseze identificatori declarați în afara propriului bloc.

Termenul s*cope* defineşte domeniul de accesibilitate al identificatorilor.

Lifetime este perioada de existență a unui identificator.

10.1 *Scope* – domeniul de accesibilitate al unui identificator

Variabilele locale sunt cele declarate în interiorul unui bloc de instrucțiuni. Corpul unei funcții este un bloc de instrucțiuni, iar variabilele declarate într-o funcție sunt variabile locale pentru acea funcție. Variabilele locale nu pot fi accesate în afara blocului în care au fost declarate. Această regulă se aplică şi constantelor.

Domeniul de accesibilitate, *scope*, este regiunea din program în care se poate folosi un identificator.

C++ defineşte patru categorii de domenii pentru un identificator:

1. *domeniul local* este domeniul unui identificator declarat într-un bloc, din punctul declarației până la sfârşitul blocului;
2. *domeniul global* (domeniul fişier) este domeniul unui identificator declarat în afara oricărei funcții sau clase, din punctul declarației până la sfârşitul fişierului care conține codul;
3. *domeniul clasă* se referă la tipurile de date introduse prin intermediul claselor;
4. al patrulea domeniu de existență este în legătură cu instrucțiunea `go to`.

În C++, numelor de funcții le corespunde domeniul global sau domeniul clasă. Funcțiile declarate în afara claselor sunt funcții globale. Odată declarată o astfel de funcție, ea poate fi folosită de orice altă funcție din program.

Variabilele globale şi constantele globale sunt declarate în afara tuturor funcțiilor sau claselor. Atunci când o funcție declară un identificator local cu acelaşi nume ca cel global, identificatorul local este cel folosit în funcție. Acest principiu se numeşte *precedența numelui* sau *ascunderea numelui*.

Exemplu

```
#include <iostream>
using namespace std;

void Functie(double);
const int a = 17;//constanta globala
int b;//variabila globala
int c;

int main()
{
  b = 4;//b: global
  c = 6;
  Functie(42.8);
  return 0;
}
```

```
void Functie(double c)//c: local
{
  double b;//b: local
  b = 2.3; //b: local
  cout << "a = " << a << endl;//a: global
  cout << "b = " << b << endl;//b: local
  cout << "c = " << c << endl;//c: local
}
```

Acest program va afişa :

```
a = 17
b = 2.3
c = 42.8
```

Reguli pentru domeniul de accesibilitate

Când scriem programe C++, declarăm rar variabile globale. Atunci când le folosim, trebuie să ştim exact cum manipulează C++ aceste declarații.

Suplimentar accesului global şi local, aceste reguli arată ce se întâmplă dacă într-un bloc includem alt bloc. Identificatorii declarați într-un bloc sunt *nelocali* pentru blocurile interioare lui. Spre deosebire de aceştia, identificatorii globali sunt nelocali pentru toate blocurile din program. Dacă un bloc accesează identificatori din afara sa, este vorba de *acces non-local*.

Iată regulile referitoare la domeniul de accesibilitate :

1. Un nume de funcție care nu este declarat într-o clasă are acces global. Definițiile de funcții nu pot fi imbricate în alte funcții;
2. Domeniul de accesibilitate al unui parametru formal este identic cu cel al unei variabile locale declarate în cel mai exterior bloc al corpului unei funcții;
3. Domeniul unei constante sau variabile globale se extinde din punctul declarației până la sfârşitul fişierului cu excepția cazului din regula 5;
4. Domeniul unei constante sau al unei variabile locale este din punctul declarației până la sfârşitul blocului, inclusiv blocurile interioare lui, cu excepția cazurilor din regula 5;
5. Domeniul de accesibilitate al unui identificator nu include blocurile interioare lui care conțin o declarație a unui identificator local cu acelaşi nume.

Exemplu

Acest program ilustrează regulile de accesibilitate din C++.

```
#include <iostream>
using namespace std;

void Bloc1(int, char&);
void Bloc2();
int a1 = 1;
char a2 = 'a';
int main()
{
  //Blocul 1
  Bloc1(a1, a2);
  Bloc2();
  return 0;
}
```

```
void Bloc1(int a1, char& b2)
{
  //Blocul 1
  //Parametrii a1 si b2 ai functiei fac si ei
  //parte din acest bloc
  int c1 = 2;
  int d2 = 3;
  cout << "Bloc1: c1 + d2 = " << c1 + d2 << endl;
}

void Bloc2()
{
  //Blocul 2
  int a1 = 4;
  int b2;
  cout << "Bloc2: b2=";
  cin >> b2;
  if(b2-a1)
  {
    //Blocul 3
    int c1 = 6;
    int b2;
    cout << "Bloc2: b2=";
    cin >> b2;
    cout << "Bloc2: b2-c1 = " << b2-c1 << endl;
  }
}
```

În acest exemplu, într-un bloc poate fi referit un identificator dintr-un alt bloc care îl cuprinde şi pe primul. De exemplu, în blocul 3 putem folosi un identificator declarat în blocul 2. Nu putem folosi, însă, în blocul 3 un identificator declarat în blocul 1 pentru că ar însemna un acces din exterior în interior. Sunt permise doar accese de la un nivel interior spre unul exterior.

Parametrii formali ai funcției `Bloc1()` sunt considerați ca aparținând blocului 1. Numele funcției, însă, se găseşte la nivelul global.

Să ne închipuim fiecare bloc din programul de mai sus ca fiind înconjurat de câte un contur. Aceste contururi ar reprezenta pereții unor camere. Pereții sunt oglinzi cu partea care reflectă orientată în afară, dar transparenți din interior. Dacă suntem în camera *Blocul 3*, vom putea vedea toate declarațiile variabilelor din camera *Blocul 2*, dar şi pe cele globale. Nu putem să vedem ce este în *Blocul 1*. Dacă suntem în camera *Blocul 2*, nu putem vedea declarațiile din *Blocul 3*, dar nici pe cele din *Blocul 1*. Datorită acestei analogii, în mod frecvent se foloseşte termenul de *vizibil* pentru a descrie domeniile de acces. Astfel, variabila `a2` este vizibilă în tot programul, iar variabila `c1` este vizibilă doar în blocul 3.

Trebuie să observăm că variabila `a1` este declarată în două locuri în program. Datorită precedenței numelui, *Blocul 2* şi *Blocul 3* accesează variabila `a1` declarată în *Blocul 2*, şi nu pe `a1` declarată la nivel global. În mod similar, domeniul de accesibilitate al variabilei `b2` declarată în *Blocul 2* nu include şi *Blocul 3* deoarece acolo este declarată o altă variabilă cu acelaşi nume, `b2`.

Compilatorul lucrează astfel atunci când este vorba de precedența numelor. Când o instrucțiune foloseşte un identificator, caută o declarație locală. Dacă

identificatorul nu este local, trece la un nivel superior până găseşte declarația, după care se opreşte. Dacă ajunge la nivelul global, incluzând şi identificatorii inserați cu directiva `#include` şi identificatorul nu este scris greşit, compilatorul generează mesajul de eroare `UNDECLARED IDENTIFIER`.

Declarații şi definiții de variabile

Terminologia C++ face diferența între o declarație de funcție şi o definiție de funcție. Limbajul C++ aplică aceeaşi terminologie şi variabilelor. Toate declarațiile de variabile pe care le-am folosit până acum au fost şi definiții. Să vedem la ce se referă această afirmație.

C++ permite scrierea unui program multifişier, adică a unui program pentru care codul este împărțit în mai multe fişiere. Se foloseşte cuvântul rezervat `extern` pentru a referi o variabilă globală localizată în alt fişier.

Dacă `var1` este o variabilă de tip întreg, o declarație a sa este următoarea:

```
int var1;
```

care are ca efect rezervarea de către compilator a unei locații de memorie de tip `int`. Pe de altă parte, declarația

```
extern int var1;
```

arată că variabila este globală şi este declarată în alt fişier, motiv pentru care nu mai trebuie rezervat spațiu de memorie pentru ea.

Fişierele header conțin declarații externe pentru variabilele importante folosite în programe.

Exemplu

În fişierul header `iostream` există declarațiile

```
extern istream cin;
extern ostream cout;
extern ostream cerr;
extern ostream clog;
```

care permit referirea în program a lui `cin` şi `cout` ca variabile globale.

În terminologia C++, instrucțiunea

```
extern int var1;
```

este o declarație a variabilei var1. Pe de altă parte, instrucțiunea

```
int var1;
```

este atât declarație cât şi definiție. O variabilă sau o funcție pot fi declarate de oricâte ori, dar pot fi definite doar o dată.

Spre deosebire de variabilele globale, folosirea constantelor globale este acceptabilă. Pentru că valorile lor nu pot fi schimbate, folosirea lor nu are comportamente neaşteptate de genul celor care apar atunci când modificarea unei valori a unei variabile globale produce efecte nedorite într-o altă funcție.

Există două avantaje ale folosirii constantelor globale:

- uşurința modificării valorii sale
- consistența.

Dacă dorim să modificăm valoarea unei constante globale folosită de mai multe ori în program, trebuie să modificăm doar declarația ei. Declarând o constantă şi folosind-o în program, suntem siguri că peste tot folosim aceeaşi valoare.

Nu trebuie să înțelegem că vom declara toate constantele globale. Dacă o constantă este folosită doar într-o funcție, este lipsit de sens să o declarăm global, ci local.

10.2 *Lifetime* – perioada de existență a unei variabile

Perioada de existență a unei variabile este perioada de timp în care un identificator are alocată o zonă de memorie.

Exemple

Variabilele locale rămân alocate din momentul intrării în funcție până la incheierea acesteia, când sunt dealocate.

Perioada de existență a variabilelor globale coincide cu perioada de existență a programului.

Putem face observația că domeniul de accesibilitate este un element legat de faza de compilare, în timp ce perioada de existență este legată de faza de execuție.

În C++ există mai multe categorii de variabile determinate de perioada de existență:

- *variabilele automatice* care sunt alocate la intrarea într-un bloc şi dealocate la ieşirea din bloc;
- *variabilele statice* care rămân alocate pe durata întregului program.

Variabilele globale sunt de tip static. Toate variabilele declarate într-un bloc sunt automatice.

Dacă folosim cuvântul cheie `static` atunci când declarăm o variabilă, aceasta devine statică şi perioada ei de existență nu se va încheia odată cu terminarea blocului. Ea va fi dispobiliă de la un apel al unei funcții la altul.

Exemplu

```
#include <iostream>
using namespace std;
void a();
void b();
void c();
int x = 1;//variabila globala

int main()
{
  int x = 5;//variabila locala functiei main
  cout << "variabila locala x din functia main "
       << "are valoarea " << x << endl;

  {
    //un nou bloc
    int x = 7;
    cout << "variabila locala x din blocul "
         << "interior functiei main are valoarea "
         << x << endl;
  }

  cout << "variabila locala x din functia main "
       << "are valoarea " << x << endl;

  a();//foloseste o variabila automatica locala x
  b();//foloseste o variabila statica locala x
  c();//foloseste variabila globala x
  a();//reinitializeaza variabila automatica locala x
  b();//variabila statica locala x retine valoarea
```

```
  c();//variabila globala x retine valoarea anterioara

  cout << "variabila locala x din functia main "
       << "are valoarea " << x << endl;

  return 0;
}

void a()
{
  int x = 25;//variabila locala automatica
             //se initializeaza la fiecare apel
             //al functiei a()
  cout << endl << "variabila locala x are valoarea "
       << x << " la intrarea in functia a()" << endl;
  ++x;
  cout << "variabila locala x are valoarea " << x
       << " inainte de iesirea din functia a()" << endl;
}

void b()
{
  static int x = 50;//variabila statica locala
                    //este initializata doar la primul apel
                    //al functiei b()

  cout << endl << "variabila locala statica x are valoarea "
       << x << " la intrarea in functia b()" << endl;
  ++x;
  cout << "variabila locala statica x are valoarea " << x
       << " inainte de iesirea din functia b()" << endl;
}

void c()
{
  cout << endl << "variabila globala x are valoarea " << x
       << " la intrarea in functia c()" << endl;
  x *= 10;
  cout << "variabila globala x are valoarea " << x
       << " inainte de iesirea din functia c()" << endl;
}
```

Rezultatul rulării programului este următorul:

```
variabila locala x din functia main are valoarea 5
variabila locala x din blocul interior functiei main are
valoarea 7
variabila locala x din functia main are valoarea 5

variabila locala x are valoarea 25 la intrarea in functia a()
variabila locala x are valoarea 26 inainte de iesirea din
functia a()
```

```
variabila locala statica x are valoarea 50 la intrarea in
functia b()
variabila locala statica x are valoarea 51 inainte de iesirea
din functia b()

variabila globala x are valoarea 1 la intrarea in functia c()
variabila globala x are valoarea 10 inainte de iesirea din
functia c()

variabila locala x are valoarea 25 la intrarea in functia a()
variabila locala x are valoarea 26 inainte de iesirea din
functia a()

variabila locala statica x are valoarea 51 la intrarea in
functia b()
variabila locala statica x are valoarea 52 inainte de iesirea
din functia b()

variabila globala x are valoarea 10 la intrarea in functia
c()
variabila globala x are valoarea 100 inainte de iesirea din
functia c()
variabila locala x din functia main are valoarea 5
```

Valoarea variabilei statice locale `x` din funcția `b()` este reținută de la un apel la altul al funcției, spre deosebire de valoarea variabilei locale automatice `x` din funcția `a()` care se pierde.

În general, se preferă declararea unei variabile locale statice decât folosirea unei variabile globale. Ca şi o variabilă globală, ea rămâne alocată pe durata întregului program, însă este accesibilă doar în interiorul funcției în care a fost declarată.

Variabilele statice fie globale, fie locale, sunt inițializate o singură dată, atunci când programul parcurge prima dată aceste instrucțiuni. Aceste variabile trebuie inițializate ca si constantele.

10.3 Namespaces (spații de nume)

Un program foloseşte identificatori incluşi în diverse domenii de accesibilitate. Uneori, o variabilă dintr-un domeniu se suprapune peste o variabilă cu acelaşi nume din alt domeniu. Suprapunerea numelor poate, uneori, să pună probleme, mai ales atunci când se folosesc în acelaşi program mai multe biblioteci care utilizează la nivel global identificatori cu aceleaşi nume. O astfel de situație conduce, de regulă, la erori de compilare.

Limbajul C++ rezolvă această problemă prin *spațiile de nume* (*namespaces*). Fiecare *namespace* defineşte un domeniu în care sunt plasați identificatorii. Pentru a folosi un membru al unui namespace, acesta trebuie să fie precedat de numele *namespace*-ului:

nume_namespace`::`*membru*

Alternativ, se poate folosi declarația `using` înaintea identificatorului care este folosit. În mod obişnuit, declarațiile `using` sunt plasate la începutul fişierului în care sunt folosiți membrii *namespace*-ului. De exemplu, instrucțiunea

`using namespace` *nume_namespace*`;`

la începutul unui fişier sursă speficică faptul că membrii lui *nume_namespace* pot fi folosiți în fişier fără a preceda fiecare identificator cu numele *namespace*-ului şi operatorul domeniu ::.

Exemplu

```
#include <iostream>
using namespace std;

int var = 98;

namespace Exemplu
{
  const double PI = 3.14159;
  const double E = 2.71828;
  int var = 8;
  void TiparesteValori();

  namespace Interior
  {
    enum Ani {FISCAL1 = 2004, FISCAL2, FISCAL3};
  }
}

namespace
{
  double d = 88.22;
}

int main()
{
  cout << "d= " << d;
  cout << "\n(global) var = " << var;
  cout << "\nPI = " << Exemplu::PI << "\nE = "
       << Exemplu::E << "\nvar = "
       << Exemplu::var << "\nFISCAL3 = "
       << Exemplu::Interior::FISCAL3 << endl;

  Exemplu::TiparesteValori();
  return 0;
}

void Exemplu::TiparesteValori()
{
  cout << "\nIn TiparesteValori():\n" << "var = "
       << var << "\nPI = " << PI << "\nE = "
       << E << "\nd = " << d << "\n(global) var = "
       << ::var << "\nFISCAL3 = "
       << Interior::FISCAL3 << endl;
}
```

Programul afişează pe ecran următorul rezultat:

```
d= 88.22
(global) var = 98
```

```
PI = 3.14159
E = 2.71828
var = 8
FISCAL3 = 2006

In TiparesteValori:
var = 8
PI = 3.14159
E = 2.71828
d = 88.22
(global) var = 98
FISCAL3 = 2006
```

Instrucțiunea

```
using namespace std;
```

informează compilatorul că va fi folosit *namespace*-ul `std`. Întreg conținutul fişierului header `iostream` este definit ca parte a lui `std`. Această declarație presupune că unii membri ai lui `std` vor fi folosiți în mod frecvent în program. Programatorul are, astfel, posibilitatea, să acceseze toți membrii acestui *namespace* şi să scrie instrucțiuni într-o variantă mai concisă:

```
cout << "d= " << d;
```

decât scriind

```
std::cout << "d= " << d;
```

Fără instrucțiunea de declarare a lui `std`, fiecare `cout` şi `endl` ar trebui să fie precedat de `std::`.

Prin declarațiile

```
namespace Exemplu
{
  const double PI = 3.14159;
  const double E = 2.71828;
  int var = 8;
  void TiparesteValori();

  namespace Interior
  {
    enum Ani {FISCAL1 = 2004, FISCAL2, FISCAL3};
  }
}
```

se definesc *namespace*-urile `Exemplu` şi `Interior`. Primul dintre ele conține constantele `PI` şi `E`, variabila `var`, funcția `TiparesteValori()` şi *namespace*-ul intitulat `Interior` care este declarat în interiorul său. Acesta, la rândul său, conține o enumerare. Aceasta reprezintă un tip de dată introdus prin cuvântul cheie `enum` urmat de un identificator şi este un set de constante întregi reprezentate de identificatorii dintre acolade. Valorile acestor constante încep cu `0` şi sunt incrementate apoi cu `1`, cu excpția cazului în care se specifică altfel. În exemplul de mai sus, constantele `FISCAL1`, `FISCAL2` şi `FISCAL3` au valorile `2004`, `2005` şi `2006`.

Un *namespace* poate fi declarat la nivel global sau în interiorul unui alt *namespace*.

Prin secvența

```
namespace
```

```
{
  double d = 88.22;
}
```

se declară un *namespace* fără nume care conține variabila d. Un *namespace* fără nume ocupă întotdeauna domeniul global, cel din care fac parte şi variabilele globale.

Accesul la membrii *namespace*-ului se face prin prefixarea acestora:

```
cout << "\nPI = " << Exemplu::PI << "\nE = "
     << Exemplu::E << "\nvar = "
     << Exemplu::var << "\nFISCAL3 = "
     << Exemplu::Interior::FISCAL3 << endl;
```

`PI`, `E` şi `var` sunt membri ai lui `Exemplu` şi sunt precedați de `Exemplu::`. Apartenența membrului `var` trebuie precizată pentru că altfel ar fi tipărită variabila globală cu acelaşi nume. `FISCAL3` trebuie precedat de `Exemplu::Interior::` pentru că face parte din *namespace*-ul Interior care este declarat în *namespace*-ul `Exemplu`.

Funcția `TiparesteValori()` este membru al lui `Exemplu` şi poate accesa în mod direct ceilalți membri ai *namespace*-ului. Variabila globală var este accesată prin folosirea operatorului domeniu unar `::`, iar `FISCAL3` este accesată prin folosirea înaintea sa a lui `Interior::`.

Cuvântul rezervat using poate fi folosit şi pentru a permite accesul la membri individuali dintr-un *namespace*. Instrucțiunea

```
using Exemplu::PI;
```

ar permite folosirea în instrucțiunile care urmează a identificatorului `PI` fără ca acesta să mai fie precedat de `Exemplu::`. Aceste declarații se fac doar atunci când sunt folosiți în mod intens într-un program un număr limitat de identificatori dintr-un *namespace*.