Separarea declarațiilor și definițiilor

Limbajul C++ distinge între *declararea* unei funcții sau a unei variabile și *definirea* acesteia. Declararea este momentul în care îi spunem compilatorului *ce tip de date* are funcția sau variabila; definiția este momentul în care îi spunem ce cod/implementare are funcția, respectiv ce valoare inițială are variabila.

Acest material o să vă ofere câteva exemple pentru cum puteți separa declarațiile funcțiilor și variabilelor voastre de definițiile lor.

Observație: VS Code nu oferă suport implicit pentru compilarea proiectelor de C++ formate din mai multe fișiere sursă. Va fi nevoie să modificați configurația din `tasks.json` sau să folosiți plugin-ul de CMake.

În cazul celorlalte editoare/IDE-uri menționate în PDF-ul de la primul laborator, ar trebui să fie suficient doar să adăugați fișierele sursă noi în proiectul vostru.

Definirea separată a funcțiilor

Înainte de a parcurge această secțiune ar fi bine să fiți familiari cu etapele compilării unui program de C++ (*preprocessing*, *compilation*, *linking*).

Dacă compilăm următorul fișier sursă, vom obține un executabil care afișează valoarea 25 la consolă:

```
1  #include <iostream>
2
3  int square(int x)
4  {
5      return x * x;
6  }
7
8  int main()
9  {
10     std::cout << square(5) << std::endl;
11 }
```

Dacă eliminăm corpul funcției `square` și îi lăsăm doar antetul, programul va trece faza de *compilation*, dar vom avea o eroare la faza de *linking*:

```
1  #include <iostream>
2
3  int square(int x);
4
5  int main()
```

```
6  {
7      std::cout << square(5) << std::endl;
8  }
```

Compilatorul compilează pe rând fiecare fișier cpp din proiectul nostru, iar apoi încearcă să le pună pe toate laolaltă pentru a obține binarul final. În acest caz, avem un singur fișier sursă, iar acesta conține un simbol care nu este definit nicăieri (funcția `square`).

Putem alege să lăsăm declararea funcției înainte de `main`, dar să o definim mai jos:

```
1  #include <iostream>
2
3  int square(int x);
4
5  int main()
6  {
7      std::cout << square(5) << std::endl;
8  }
9
10 int square(int x)
11 {
12     return x * x;
13 }
```

Din nou, fișierul sursă trece fără probleme de faza de *compilation*, deoarece avem declarată funcția `square` înainte de prima ei utilizare. La faza de *linking*, compilatorul va găsi funcția `square` (definită după `main`) și nu vom rămâne cu simboluri nedefinite.

Același principiu funcționează și dacă avem mai multe fișiere sursă în proiectul nostru. De exemplu, în `main.cpp` putem avea:

```
1  #include <iostream>
2
3  int square(int x);
4
5  int main()
6  {
7      std::cout << square(5) << std::endl;
8  }
```

Iar în fișierul `square.cpp` putem avea doar:

```
1 int square(int x)
2 {
3     return x * x;
4 }
```

Singura problemă rămasă care mai poate apărea este dacă signatura funcției declarate nu se potrivește cu signatura funcției definite. Dacă schimbăm tipul de date returnat sau parametrii funcției din `main.cpp`, compilatorul nu va mai reuși să o potrivească cu definiția funcției `square` din fișierul `square.cpp`. Problema va fi la faza de *linking*, nu la *compilation*, ceea ce poate duce la mesaje de eroare greu de înțeles.

Pentru a preveni această situație, dar și pentru a putea refolosi în mai multe fișiere sursă aceiași declarație, o putem muta într-un fișier *header*, denumit (de exemplu) `square.hpp`:

```
1 #ifndef SQUARE_HPP
2 #define SQUARE_HPP
3
4 int square(int x);
5
6 #endif
```

Directivele de preprocesare cu `#ifndef` și `#define` sunt un *include guard*, se pun de obicei în toate fișierele header pentru a preveni includerea multiplă a acestora. Alternativ, putem folosi `#pragma once`:

```
1 #pragma once
2
3 int square(int x);
```

Directiva de preprocesare `#pragma once` nu este standard, dar este suportată de majoritatea compilatoarelor de C++.

Definirea separată a metodelor

La fel ca și pentru funcții, putem declara metodele atunci când definim o structură/clasă în C++, și să le definim ulterior:

```
1 class Exemplu
2 {
3     int x;
4
```

```
5  public:
6      Exemplu(int valoare);
7      int get_x() const;
8  };
9
10 Exemplu::Exemplu(int valoare)
11 {
12     x = valoare;
13 }
14
15 int Exemplu::get_x() const
16 {
17     return x;
18 }
```

Definirea separată a metodelor este asemănătoare cu definirea separată a funcțiilor; trebuie să avem grijă ca signatura declarată în clasă să se potrivească cu cea folosită la definiție.

Principala diferență este că trebuie să includem numele clasei atunci când le implementăm (în cazul nostru, `Exemplu::`), ca să le putem distinge de funcțiile libere cu același nume.

Declararea înainte a claselor

Pe lângă faptul că putem defini separat metodele unei clase, putem doar să declarăm existența unei clase, definind-o ulterior.

```
1  class Exemplu;
2
3  void proceseaza(Exemplu ex);
```

În această situație, **nu** vom putea defini variabile concrete de tipul acestei clase până nu avem și definiția clasei în același fișier. Asta se întâmplă deoarece compilatorul nu poate determina cât spațiu să aloce pe stivă (dacă e vorba de o variabilă locală) sau în memoria globală (dacă e vorba de o variabilă globală) pentru obiectele din clasa respectivă.

```
1  class Exemplu;
2
3  Exemplu x; // eroare de compilare
4
5  void proceseaza(Exemplu ex) // eroare de compilare
```

```
{
    // ...
}
```

Definirea separată a variabilelor

Variabilele pot fi declarate și ele înainte de a fi definite. Problema este că atunci când declarăm o variabilă, compilatorul implicit o și instanțiază (îi apelează constructorul fără parametrii). De aceea, trebuie să folosim cuvântul cheie `extern` pentru a dezactiva acest comportament:

```
#include <iostream>

extern int x;

int main()
{
    std::cout << "Valoarea lui x este " << x << std::endl;
}

int x = 5;
```

Dacă nu am fi inclus definiția lui `x` după subprogramul principal, codul ar fi trecut de *compilation* dar am fi avut o eroare la faza de *linking*.

Observație: nu putem să folosim modificatorul `extern` pentru variabile locale, doar pentru cele globale.

Pentru variabilele statice dintr-o structură/clasă, comportamentul implicit este ca acestea să fie doar declarate în cadrul structurii/clasei, și definite undeva în afară:

```
class Exemplu {
    static int x;

public:
    static int get_x() { return x; }
};

int Exemplu::x = 15;
```

De fapt, dacă variabila statică nu este declarată cu `const`, nici măcar nu este permis să o *definim* în interiorul clasei.