9. Alte structuri de control

În capitolele precedente am văzut care instrucțiuni C++ implementează secvențele, selecțiile, buclele şi subprogramele. În unele cazuri am introdus mai multe metode de a implementa o structură, ca în cazul instrucțiunii `IF-THEN` sau `IF-THEN-ELSE`.

În acest capitol, vom introduce cinci noi instrucțiuni utile în programare. Prima, `switch`, face mai uşoară scrierea structurilor de selecție care au mai multe ramuri. Următoarele două instrucțiuni, `for` şi `do-while` facilitează implementarea anumitor tipuri de bucle. Ultimele două instrucțiuni, `break` şi `continue`, sunt folosite de asemenea în bucle şi instrucțiuni de selecție.

9.1 Instrucțiunea `switch`

Aceasta este o instrucțiune pentru implementarea structurilor de control cu ramificare multiplă. Valoarea *expresiei* `switch` determină ramura care se execută.

Instrucțiunea `switch` constă dintr-o serie de etichete `case` şi o variantă `default`. Programul de mai jos foloseşte instrucțiunea `switch` pentru a contoriza numărul de calificative din categoriile *foarte bine*, *bine*, *satisfăcător* şi *insuficient* pe care le-a primit un elev de-a lungul unui an şcolar. Vom codifica aceste calificative prin literele F – foarte bine, B – bine, S – satisfăcător şi I – insuficient.

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

int main()
{
  char calificativ;
  int nrF = 0,//numarul calificativelor F - foarte bine
      nrB = 0,//numarul calificativelor B - bine
      nrS = 0,//numarul calificativelor S - satisfacator
      nrI = 0;//numarul calificativelor I - insuficient

  cout << "Introduceti litera corespunzatoare"
       << "calificativului." << endl
       << "Tastati caracterul EOF pentru a incheia."
       << endl;

  while((calificativ = cin.get()) != EOF)
  {
    switch(calificativ)
    {
      case 'F':
      case 'f':
        ++nrF;
        break;
      case 'B':
      case 'b':
        ++nrB;
        break;
```

```
          case 'S':
          case 's':
            ++nrS;
            break;
          case 'I':
          case 'i':
            ++nrI;
            break;
          case '\n':
          case '\t':
          case ' ':
            break;
          default://orice alt caracter
            cout << "Ati introdus o litera incorecta."
                 << " Introduceti un nou calificativ." << endl;
        }
      }
      cout << "\n\nTotalurile pentru fiecare calificativ"
           << " sunt:"
           << "\nFoarte bine: " << nrF
           << "\nBine: " << nrB
           << "\nSatisfacator: " << nrS
           << "\nInsuficient: " << nrI << endl;

      return 0;
    }
```

În acest program, utilizatorului i se cere să introducă literele asociate calificativelor. Bucla `while` care implementează operația de citire şi prelucrare a datelor este scrisă astfel:

```
while((calificativ = cin.get()) != EOF)
{
  ...
}
```

Testul buclei constă dintr-o asignare urmată de o comparație. Asignarea

```
(calificativ = cin.get())
```

dintre parantezele rotunde este executată prima. Funcția `cin.get()` citeşte un caracter din stream-ul de intrare. Codul ASCII al acestui caracter este, apoi, asignat variabilei `calificativ`. Valoarea acestei variabile este comparată cu `EOF`.

Când caracterul este diferit de `EOF`, calculatorul trece la execuția instrucțiunii `switch`. Cuvântul rezervat `switch` este urmat, între paranteze, de o expresie integrală (ex. `char`, `int`) a cărei valoare este comparată cu fiecare etichetă `case` care trebuie să fie întotdeauna o expresie constantă integrală. Presupunând că am introdus litera `F` corespunzătoare calificativului *foarte bine*, `F` este comparat cu fiecare `case`. Dacă există o potrivire a valorilor (`case 'F':`), se execută secvența de instrucțiuni corespunzătoare acestui `case`. În exemplul nostru, se incrementează variabila `nrF`. Să notăm că, spre deosebire de alte structuri de control, nu este nevoie să folosim blocuri de instrucțiuni pentru a delimita secvențele asociate unui `case`. Instrucțiunea `break` are ca efect transferul controlului programului la prima instrucțiune de după instrucțiunea `switch`. Dacă omitem instrucțiunea `break`, atunci se execută toate instrucțiunile care urmează ramurii selectate. Dacă nu apare nicio

potrivire a valorilor între expresia din `switch` şi etichetele `case`, se execută varianta `default`.

Sintaxa instrucțiunii switch permite gruparea mai multor etichete case:

```
case 'I':
case 'i':
```

însemnând că pentru ambele cazuri se realizează acelaşi set de acțiuni.

O constantă poate apărea o singură dată în instrucțiunea `switch` pentru că în caz contrar compilatorul generează o eroare de sintaxă. De asemenea, se poate folosi o singură etichetă `default`.

Instrucțiunea switch are acelaşi efect ca instrucțiunea IF-THEN-ELSE, fapt ilustrat prin scrierea în două variante a structurii din programul anterior.

Exemplu

```
switch(calificativ)
{
  case 'F':
  case 'f':
    ++nrF;
    break;
  case 'B':
  case 'b':
    ++nrB;
    break;
  case 'S':
  case 's':
    ++nrS;
    break;
  case 'I':
  case 'i':
    ++nrI;
    break;
  case '\n':
  case '\t':
  case ' ':
    break;
  default://alt caracter
    cout << "Ati introdus o litera incorecta."
         << " Introduceti un nou calificativ." << endl;
}
```

sau

```
if(calificativ == 'F' || calificativ == 'f')
  ++nrF;
else if(calificativ == 'B' || calificativ == 'b')
  ++nrB;
else if(calificativ == 'S' || calificativ == 's')
  ++nrS;
else if(calificativ == 'I' || calificativ == 'i')
  ++nrI;
else if(calificativ == '\n' || calificativ == '\t'
        || calificativ == ' ')
{
}
```

```
else
  cout << "Ati introdus o litera incorecta."
       << " Introduceti un nou calificativ." << endl;
```

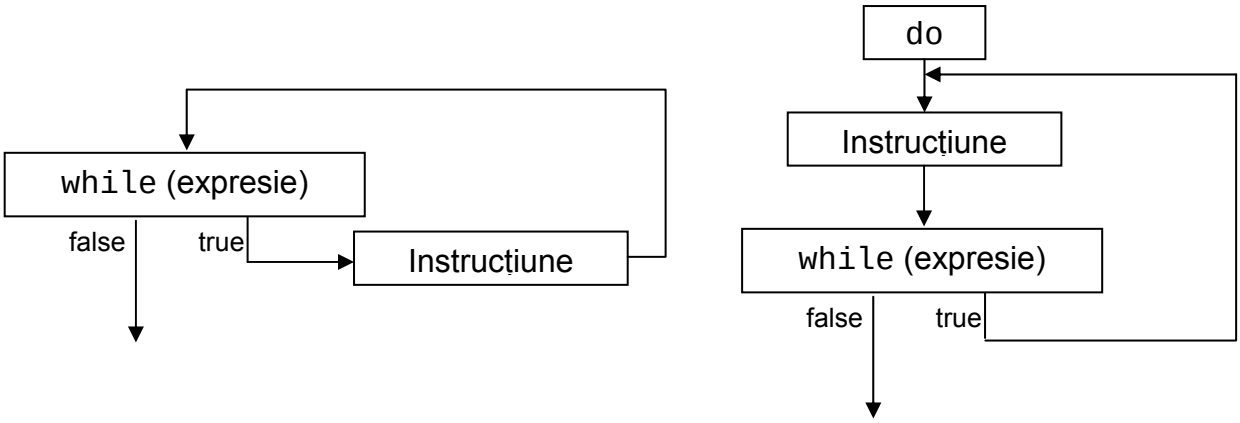
9.2 Instrucțiunea do-while

Aceasta este o instrucțiune de control al buclelor în care condiția de final se testează la sfârşitul buclei, garantând faptul că bucla se parcurge cel puțin o dată.

Şablonul sintactic al instrucțiunii `while` este

```
do
{
   Instrucțiune 1
   Instrucțiune 2
   ...
   Instrucțiune n
} while (expresie);
```

Bucla cuprinsă între `do` şi `while` se execută atâta timp cât expresia din `while` este nenulă sau `true`.



Să comparăm o buclă `while` şi una `do-while` care realizează aceeaşi acțiune: găsesc primul punct din stream-ul de intrare.

```
char inputChar;
cin >> inputChar;
while(inputChar != '.')
  cin >> inputChar;
```

```
char inputChar;
do
  cin >> inputChar;
while(inputChar != '.');
```

Soluția `while` are nevoie de o primă citire astfel încât `inputChar` să aibă o valoare înainte de intrarea în buclă. Acest lucru nu este necesar pentru soluția do-while datorită faptului că bucla este executată o dată înainte de a se evalua condiția.

Putem folosi structura `do-while` pentru a implementa o buclă controlată de un contor dacă ştim în avans că bucla este parcursă cel puțin o dată. Următorul exemplu prezintă două bucle care însumează întregii de la `1` la n.

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

int main()
{
  int n = 10;
  int sum = 0;
  int contor = 1;
  while(contor <= n)
```

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

int main()
{
  int n = 10;
  int sum = 0;
  int contor = 1;
  do
```

```
  {                                  {
    sum += contor;                     sum += contor;
    contor++;                          contor++;
  }                                  } while(contor <= n);

  cout << "Suma este "               cout << "Suma este "
       << sum << endl;                    << sum << endl;

  return 0;                          return 0;
}                                  }
```

Dacă n este un număr pozitiv, ambele versiuni sunt echivalente. Dar dacă n este 0 sau negativ, cele două bucle conduc la rezultate diferite. În versiunea `while`, valoarea finală a lui `sum` este `0` pentru că bucla nu se execută niciodată. În varianta `do-while`, `sum` are valoarea `1` pentru că bucla se execută o dată.

Instrucțiunea `while` testează condiția înainte de executarea corpului buclei şi se numeşte *buclă pretest*, în timp ce instrucțiunea `do-while` se numeşte *buclă posttest*.

9.3 Instrucțiunea `for`

Această instrucțiune simplifică scrierea buclelor controlate de contor, în C++ instrucțiunea `for` fiind formă mai compactă a buclei `while`.

Exemplu

```
for(int contor = 1; contor <= n; contor++)
  cout << contor << endl;

int contor = 1;
while(contor <= n)
{
  cout << contor << endl;
  contor++;
}
```

Ambele instrucțiuni tipăresc numerele de la `1` la n.

Instrucțiunea for din exemplul de mai sus inițializează variabila de control al buclei, `contor`, cu valoarea `1`. Atâta timp cât valoarea lui `contor` este mai mică sau egală cu n execută instrucțiunea de afişare şi incrementează `contor`.

Corespondența dintre componentele instrucțiunii `for` şi cele ale lui `while` este marcată prin caracterele **aldine**. Instrucțiunea `for` plasează inițializarea dinaintea buclei pe prima poziție dintre parantezele rotunde. Testul buclei este pe a doua poziție, iar incrementarea variabilei de contor este pe ultima poziție. Pe prima poziție se găsesc acțiunile care se execută înainte de prima iterație, iar pe ultima poziție sunt cele care se execută la sfârşitul fiecărei iterații. Dacă este vorba de mai mult de o acțiune, atunci acestea sunt separate prin virgulă.

Ca şi buclele `while`, şi buclele `do-while` sau `for` pot fi imbricate.

Exemplu

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

int main()
{
```

```
  for(int num = 1; num <= 5; num++)
  {
    for(int numToPrint =1; numToPrint <= num; numToPrint++)
      cout << numToPrint;
    cout << endl;
  }

  return 0;
}
```

Rezultatul rulării acestui program este

```
1
12
123
1234
12345
```

9.4 Instrucțiunile `break` şi `continue`

Instrucțiunea `break` introdusă odată cu instrucțiunea `switch` poate fi folosită şi în bucle. Ea provoacă ieşirea imediată din cel mai interior `switch`, `while`, `do-while` sau `for` în care apare.

Una dintre cele mai frecvente situații în care se foloseşte este ieşirea din buclele infinite.

Exemplu

```
#include <iostream>
#include <math.h>
using namespace std;

int main()
{
  int num1, num2;
  int contor = 1;
  while(true)
  {
    cin >> num1;
    if(!cin || (num1 >= 100))
      break;
    cin >> num2;
    if(!cin || (num2 <= 50))
      break;
    cout << sqrt(static_cast<double>(num1+num2)) << endl;
    contor++;
    if(contor > 10)
      break;
  }
  return 0;
}
```

Pentru exemplul acesta am fi putut folosi o buclă for de la 1 la 10. În tot cazul, bucla este controlată de contor şi de eveniment, deci preferăm bucla while.

Această buclă conține 3 puncte distincte de ieşire. Unii programatori se opun acestui stil de programare susținând că face codul mai greu de urmărit. Rescriind codul, vom vedea ca prima variantă este, totuşi, mai clară.

Exemplu

```
#include <iostream>
#include <math.h>
using namespace std;

int main()
{
  bool num1Valid = true;
  bool num2Valid = true;
  int num1, num2;
  int contor = 1;
  while(num1Valid && num2Valid && contor<=10)
  {
    cin >> num1;
    if(!cin || (num1 >= 100))
      num1Valid = false;
    else
    {
      cin >> num2;
      if(!cin || (num2 <= 50))
        num2Valid = false;
      else
      {
        cout << sqrt(static_cast<double>(num1+num2))
             << endl;
        contor++;
      }
    }
  }
  return 0;
}
```

O altă instrucțiune care afectează fluxul unui program C++ este `continue`. Ea termină iterația curentă şi se foloseşte numai în bucle.

Exemplu

```
#include <iostream>
#include <math.h>
using namespace std;

int main()
{
  double valIntrare;
  for(int i = 1; i <= 5; i++)
  {
    cin >> valIntrare;
    if(valIntrare < 0)
      continue;
    cout << sqrt(valIntrare) << endl;
  }
```

```
  return 0;
}
```

Dacă `valIntrare` este negativ, se trece la următoarea iterație. Şi această secvență de program poate fi rescrisă.

Exemplu

```
#include <iostream>
#include <math.h>
using namespace std;

int main()
{
  double valIntrare;
  for(int i = 1; i <= 5; i++)
  {
    cin >> valIntrare;
    if(valIntrare >= 0)
      cout << sqrt(valIntrare) << endl;
  }
  return 0;
}
```

Aşadar, diferența dintre `break` şi `continue` este că `break` întrerupe imediat bucla curentă, iar `continue` întrerupe iterația curentă.

9.5 Proiectarea orientată pe obiecte – abstractizarea în programare

Clase şi obiecte

Să ne imaginăm că dorim să realizăm imagini formate din pătrate. Fiecare pătrat este un obiect care este caracterizat prin atributele de culoare şi poziție. Este caracterizat, de asemenea, printr-un comportament pentru că, printre altele putem să îi schimbăm culoarea şi să il redesenăm.

Fiecare pătrat este diferit, dar are proprietăți şi comportamente comune cu alte pătrate. De aceea, putem să abstractizăm aceste elemente comune: toate pătratele au acelaşi comportament şi au acelaşi set de atribute. Abstractizând, ignorăm valorile particulare ale atributelor, adică nu ținem cont de faptul că un pătrat este roşu sau altul este verde, sau că unul este aşezat în mijlocul ecranului iar altul intr-un colț al său.

Prin abstractizare am realizat o *clasă*. O clasă este o mulțime de *obiecte* care au o structură comună şi care au comportamente comune. Clasele reprezintă şabloane ale obiectelor pentru că dacă dorim să introducem în imaginea noastră un nou pătrat, folosim clasa pătrat patricularizându-i proprietățile.

Reprezentarea UML a unei clase se face printr-un dreptunghi împărțit în trei secțiuni.

Patrat
culoare : int varfuri : Punct[] nrVarfuri : int = 4
schimbaCuloarea(c : int) : void deseneaza() : void

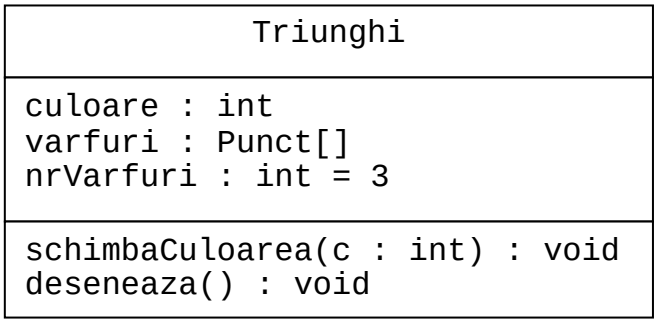
Prima parte conține numele clasei.

Partea din mijloc conține *atributele* clasei. Descrierea unui atribut este formată din numele atributului, tipul său şi valoarea inițială. Tipul depinde de limbajul de programare în care se va face implementarea. Putem folosi tipuri primitive de date, aşa cum am facut pentru atributele `culoare` şi `nrVarfuri`, sau tipuri de date definite prin alte clase, cum a fost definit atributul `varfuri`.

A treia secțiune conține comportamentele sau operațiile ca nume de funcții. Numele operației este urmat între paranteze rotunde de lista de parametri separați prin virgulă. Numele unui parametru este urmat de tipul acestuia. Lista poate să fie vidă dacă operația nu foloseşte parametri. Ultimul element al descrierii unei operații este tipul valorii returnate care este precedat de semnul `:`.

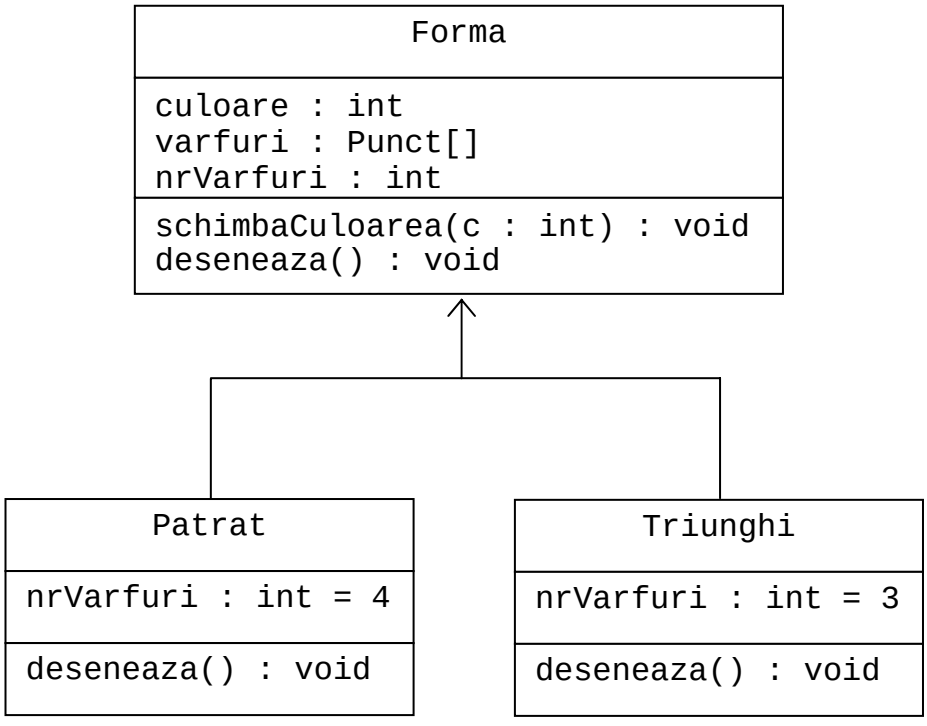
Moştenirea

Să ne imaginăm că pe lângă pătrate ne propunem să folosim şi triunghiuri în imaginile pe care le construim.



Comparând cele două clase, observăm că sunt câteva diferențe între ele pentru că pătratele au patru vârfuri, în timp ce triunghiurile au doar trei. Totodată, modul în care sunt desenate aceste două figuri diferă.

Pe de altă parte, sunt şi similarități. De exemplu, putem stabili culoarea ambelor figuri şi ambele figuri pot fi redesenate, chiar dacă modul de desenare este diferit. Pe baza asemănărilor, dorim să abstractizăm aceste clase eliminând detaliile specifice fiecărei forme şi amplificând faptul că ambele au culoare şi pot fi desenate. La un



nivel mai înalt de abstractizare, am putea să gândim un desen ca fiind alcătuit din forme, iar desenarea intregii imagini să se poată face desenând pe rând fiecare formă. Dorim să eliminăm detaliile nerelevante, deci nu suntem interesați de faptul că o formă este un pătrat sau că o altă formă este un triunghi, atâta timp cât pentru fiecare dintre ele vom concepe ulterior câte o metodă particulară de desenare.

Pentru a face acest lucru, vom muta elementele comune într-o nouă clasă pe care o numim `Forma`.

Această formă de abstractizare se numeşte *moştenire*. Clasele de la nivelul de jos se numesc *subclase* sau *clase derivate*. Ele moştenesc elemente care definesc starea şi comportamentul de la clasa de la nivelul superior numită şi *superclasă* sau *clasă de bază*.

Prin acest mecanism, un obiect de tip `Triunghi` va avea o `culoare`, o operație `schimbaCuloarea(c : int)`, un tablou de `varfuri`, o operație `deseneaza()`, iar `nrVarfuri` va fi trei.

O clasă poate moşteni următoarele trei tipuri de elemente:

- starea, de exemplu `culoare`
- operațiile, de exemplu `schimbaCuloarea(c : int)`
- *interfața* unei operații: Clasa `Forma` conține interfața operației `deseneaza()` pentru că interfața `deseneaza()` pentru `Triunghi` şi `Patrat` este aceeaşi. Implementarea operației rămâne în subclase, ea fiind diferită pentru fiecare dintre ele.

Atunci când abstractizăm interfața unei operații lăsând implementarea în subclase spunem că avem de a face cu o *operație polimorfică*.

Această metodă de abstractizare plasează elementele importante care definesc starea, comportamentul şi interfețele în clasele de bază. Putem recurge la abstractizare şi altfel, combinând obiecte în interiorul unui nou obiect. Un exemplu sugestiv este cel al autoturismului privit ca un obiect care foloseşte o baterie, un aparat de radio, un motor dar şi alte obiecte oferind o interfață simplă pentru conducerea sa. Există diverse moduri de abstractizare iar găsirea celei mai potrivite pentru o problemă este cheia proiectării orientate pe obiecte.

Atunci când, ca metodă de proiectare, se foloseşte abordarea orientării pe obiecte, putem folosi abstractizarea pentru a împărți o problemă în subprobleme mai uşor de rezolvat. Lucrând la un nivel de abstractizare potrivit, rezolvarea poate fi mult mai uşoară.