6. Bucle

În capitolul trecut am văzut cum putem selecta diferite instrucțiuni pentru execuție folosind instrucțiunea `if`. O *buclă* este o structură de control care provoacă executarea unei instrucțiuni sau a unei secvențe de instrucțiuni în mod repetat. Instrucțiunile se execută atâta timp cât sunt îndeplinite una sau mai multe condiții.

Vom descrie diferite tipuri de bucle şi vom vedea cum se pot implementa acestea folosind instrucțiunea `while`. Vom prezenta, de asemenea, buclele imbricate.

6.1 Instrucțiunea while

Această instrucțiune, ca şi `if`, testează o condiție. Sintaxa ei este următoarea:

`while(`***expresie***`)`
Instrucțiune

Exemplu

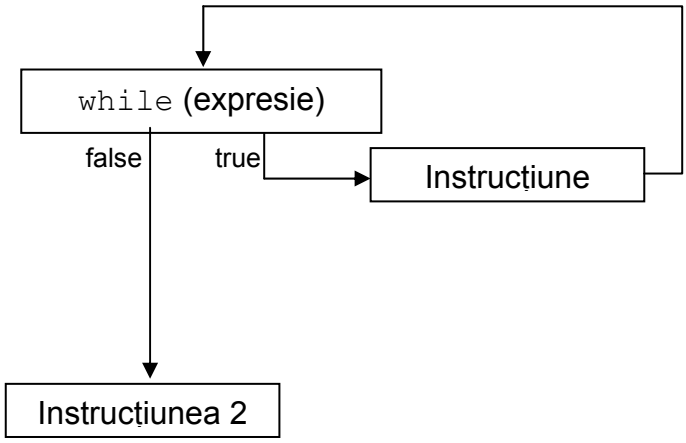
```
while(valIn != 25)
    cin >> valIn;
```

Instrucțiunea care se execută în mod repetat se numeşte *corpul buclei*.

Condiția din instrucțiunea `while` poate fi o expresie de orice tip de dată. Aproape întotdeauna ea este o expresie logică. Instrucțiunea while din exemplul de mai sus spune următorul lucru: *Dacă valoarea expresiei este `true` (nenulă), execută corpul buclei iar apoi revino şi testează din nou expresia. Dacă expresia este `false` (zero), treci de corpul buclei.*

Dacă expresia este falsă de la început, corpul nu se execută niciodată.

În figura de mai jos arătăm în mod schematic modul de execuție a instrucțiunii `while`.



Corpul buclei poate fi şi un bloc, fapt care ne permite să executăm mai multe instrucțiuni în mod repetat.

Exemplu

```
while(expresie)
{
...
}
```

Blocul se execută până când expresia devine falsă.

6.2 Fazele de execuție a unei bucle

1. **Intrarea în buclă**. Este punctul în care programul ajunge la prima instrucțiune din interiorul buclei.
2. **Iterarea**. De fiecare dată când se execută corpul buclei, spunem că facem câte o trecere prin buclă. Această trecere se numeşte *iterație*.
3. **Testul buclei**. Reprezintă punctul în care se evaluează expresia din instrucțiunea `while`. În urma acestei evaluări se poate lua decizia de a se începe o nouă iterație sau de a trece la instrucțiunea imediat următoare buclei.
4. **Condiția de terminare**. Este condiția care provoacă ieşirea din buclă, trecându-se la prima instrucțiune de după buclă. Această condiție apare în instrucțiunea `while`.
5. **Ieşirea din buclă**. Într-o buclă `while`, ieşirea din buclă apare când expresia din instrucțiunea `while` este `false` sau `0`. În acest moment, se întrerupe repetarea corpului buclei.

Deşi condiția de terminare poate deveni validă în mijlocul corpului buclei, iterația curentă este executată până la capăt şi numai după aceea calculatorul verifică din nou expresia din instrucțiunea `while`.

6.3 Implementarea buclelor folosind instrucțiunea `while`

În rezolvarea problemelor se pot întâlni două tipuri majore de bucle:

- bucla controlată de un contor;
- bucla controlată de un eveniment.

Dacă în timpul unui antrenament sportiv vi se cere să alergați *de 3 ori* în jurul stadionului, este vorba de o *buclă controlată de contor*. Dacă, în schimb, vi se cere să alergați *până când veți auzi sunetul fluierului*, avem de a face cu o *buclă controlată de un eveniment*.

Bucla controlată de un contor

O astfel de buclă foloseşte o variabilă numită *variabilă de control al buclei*. Înaintea buclei ea este inițializată, adică i se atribuie o valoare inițială. Apoi, în fiecare iterație a buclei ea trebuie incrementată.

Exemplu

```
int contorBucla = 1;      //initializare
while(contorBucla <= 10) //test
{
   ...                    //actiune care se repeta
   contorBuclă++;         //incrementare
}
```

În acest exemplu, `contorBucla` este variabila de control al buclei. Ea este inițializată cu valoarea `1` înainte de intrarea în buclă. Instrucțiunea while testează expresia

```
contorBucla <= 10
```

şi execută bucla atâta timp cât expresia este adevărată. Ultima instrucțiune a buclei incrementează variabila contorBucla. Variabilele folosite în acest fel se numesc *contoare*.

La folosirea acestor bucle, programatorul trebuie să urmărească inițializarea contorului înaintea instrucțiunii `while`. Trebuie, de asemenea, să urmarească dacă

în interiorul buclei valoarea lui se modifică în aşa fel încât la un moment dat condiția să devină falsă.

O buclă din care programul nu poate ieşi deloc se numeşte *buclă infinită*. Această situație apare atunci când în program se omite incrementarea contorului.

Bucla controlată de un eveniment

Pentru această categorie de bucle condiția de terminare depinde de un eveniment care poate apărea în timpul execuției corpului buclei. Vom studia două tipuri de bucle controlate de evenimente:

- bucla controlată de o valoare de semnalizare (valoare santinelă);
- bucla controlată de sfârşitul unui fişier (`EOF`).

Bucla controlată de o valoare de semnalizare (valoare santinelă)

Aceste bucle se folosesc în special atunci când se prelucrează volume mari de date. La fiecare iterație se citeşte şi se prelucrează câte un set de date. Anumite valori dintre cele citite vor semnaliza încheierea buclei `while`. Bucla `while` îşi continuă execuția atâta timp cât valorile citite nu sunt cele de semnalizare (santinelă).

Exemplu

```
int luna, ziua;
cin >> luna >> ziua;       //citeste primul set de date
while(!(luna == 2 && ziua == 31))
{
   ...                     //procesare
   cin >> luna >> ziua;    //urmatorul set de date
}
```

Este bine ca valorile santinelă să fie dintre cele care nu apar în mod obişnuit între datele valide de intrare.

Înainte de intrarea în buclă este citit primul set de date. Dacă nu este vorba despre valorile santinelă, ele sunt procesate. La sfârşitul buclei se citeşte următorul set de date, revenindu-se apoi la începutul buclei. Bucla se execută până la citirea valorilor santinelă. Acestea nu sunt prelucrate şi conduc la ieşirea din buclă.

Exemplu

Atunci când prelucrăm date de tip `char` putem folosi caracterul *newline* ca valoare santinelă:

```
char inChar;
cin.get(inChar);
while(inChar != '\n')
{
   cout << inChar;
   cin.get(inChar);
}
```

Ce se întâmplă dacă nu introducem valoare santinelă? Un program interactiv ne va cere în continuu noi valori. Dacă intrările în program se fac dintr-un fişier şi datele se epuizează înaintea apariției valorii santinelă, stream-ul intră in *fail state*.

O greşeală frecventă în urma căreia programul poate avea o evoluție nedorită este folosirea neintenționată a operatorului = în locul lui ==.

Exemplu

```
char inChar, valSemnal;
cin >> inChar >> valSemnal;
while(valSemnal = 1)
```

```
        //din greseala am folosit = in loc de ==
    {
        ...
        cin >> inChar >> valSemnal;
    }
```

Această eroare creează o buclă infinită. Expresia din instrucțiunea `while` este o asignare şi nu o expresie logică. Calculatorul evaluează valoarea variabilei `valSemnal` după asignare. Aceasta va fi `1` şi va fi interpretată ca fiind `true`. Astfel, expresia testată în exemplul de mai sus stochează valoarea `1` în `valSemnal` înlocuind valoarea care tocmai a fost citită. Pentru că expresia este tot timpul `true`, bucla nu se întrerupe niciodată.

Bucla controlată de sfârşitul unui fişier (EOF)

După ce programul citeşte şi ultimele date din fişierul de intrare, calculatorul ajunge la sfârşitul fişierului (`EOF`, *end of file*). În acest moment starea stream-ului este normală. Dar dacă încercăm să citim o nouă dată, stream-ul intră în *fail state*. Putem folosi acest comportament în avantajul nostru în buclele `while` în care se citeşte un număr necunoscut de valori. Starea de eroare a stream-ului poate fi interpretată ca valoare santinelă pentru că numele stream-ului poate apărea într-o expresie logică la fel ca o variabilă booleeană. Într-un astfel de test, rezultatul este `true` dacă ultima operație de intrare/ieşire a avut succes, sau este `false` dacă aceasta a eşuat.

Să presupunem că avem un fişier de date care conține valori întregi. Putem scrie:

```
int inVal;
inData >> inVal;
while(inVal)
{
    cout << inVal << endl;
    inData >> inVal;
}
```

Dacă fişierul de date conține numerele `10`, `20` şi `30`, primele 3 citiri se vor realiza corect. Chiar şi după citirea lui `30` starea stream-ului este normală. Dacă dorim să citim după sfârşitul fişierului, însă, stream-ul va intra în stare de eroare. Aceasta înseamnă că valoarea expresiei logice din `while` va fi `false` provocând ieşirea din buclă. Trebuie spus că orice eroare de citire conduce la intrarea stream-ului *în fail state*.

Similar, se poate folosi şi stream-ul de intrare cin. În sistemele UNIX, combinația de taste CTRL-D, iar în sistemele DOS combinația de taste CTRL-Z au semnificația EOF pentru intrări interactive.

6.4 Operații în buclă

Pentru a avea sens, o buclă trebuie să realizeze o operație. Vom discuta despre

- contorizare;
- însumare;
- păstrarea unei valori anterioare.

Contorizarea

O operație comună este memorarea numărului de iterații executate. Programul care urmează citeşte şi numără caracterele dintr-o propoziție, până la apariția punctului.

Exemplu

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

int main()
{
  char inChar;
  int count = 0;       //initializarea contorului
  cin.get(inChar);     //citirea primului caracter
  while(inChar != '.')
  {
    count++;           //incrementarea contorului
    cin.get(inChar);   //citirea urmatorului caracter
  }
  cout << "Propozitia are " << count
       << " caractere" << endl;
  return 0;
}
```

După terminarea buclei, `count` va conține cu 1 mai puțin decât numărul de caractere citite, adică nu numără şi valoarea santinelă (`'.'`). Facem o primă citire înaintea buclei pentru că aceasta este controlată de un caracter de semnalizare.

O variabilă care numără câte iterații se execută se numeşte *contor de iterații* . În exemplul nostru, variabila `count` este un contor de iterații.

Însumarea

O altă operație care se poate implementa cu ajutorul buclelor este însumarea unui set de valori.

Exemplu

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

int main()
{
  int numar;
  int suma = 0;
  int contor = 1;
  while(contor <= 5)
  {
    cin >> numar;
    suma = suma + numar;
    contor++;
  }
  cout << "Suma este " << suma << endl;
  return 0;
}
```

Inițializăm suma cu 0 înainte de începutul buclei. Atunci când se execută prima dată instrucțiunea

```
suma = suma + numar;
```

se adaugă valoarea curentă a variabilei `suma` la valoarea variabilei `numar` pentru a forma noua valoare a variabilei `suma`. După executarea buclei, variabila `suma` va conține suma celor 5 valori citite, `contor` va conține valoarea 6 şi `numar` ultima valoare citită.

Păstrarea unei valori anterioare

Avem uneori nevoie în program de o valoare anterioară a unei variabile. Să presupunem că dorim să scriem un program care contorizează numărul de operatori `!=` dintr-un fişier sursă C++. Va trebui să numărăm de câte ori apare semnul `!` urmat de `=`. De fiecare dată vom citi din fişierul de intrare un caracter păstrând ultimele două valori în două variabile diferite. La fiecare iterație valoarea curentă devine valoare anterioară şi apoi se citeşte o nouă valoare. Bucla se termină când se ajunge la `EOF`.

Exemplu

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

int main()
{
  int contor = 0;
  char carAnterior;
  char carCurent;
  ifstream inFisier;
  inFisier.open("main.cpp");

  inFisier.get(carAnterior);
  inFisier.get(carCurent);
  while(inFisier)
  {
    if(carCurent == '=' && carAnterior == '!')
      contor++;
    carAnterior = carCurent;
    inFisier.get(carCurent);
  }
  cout << contor
       << " operator(i) != au fost gasiti in fisier"
       << endl;
  return 0;
}
```

Contorul din acest exemplu este un *contor de evenimente*. El este o variabilă care se incrementează atunci când apare un anumit eveniment. Este inițializat cu valoarea 0 spre deosebire de contorul de iterații din exemplul precedent care este inițializat cu 1.

6.5 Instrucțiuni `while` imbricate

Am studiat în capitolul anterior modul în care se pot scrie instrucțiunile `if` imbricate. Este posibil să folosim şi instrucțiuni `while` imbricate.

Exemplu

Ne propunem să numărăm câte caractere `;` sunt pe fiecare linie dintr-un fişier.

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

int main()
{
  ifstream inFisier;
  inFisier.open("main.cpp");

  char inChar;
  inFisier.get(inChar);
  while(inFisier)
  {
    int contorPunctVirgula = 0;
    while(inChar != '\n')
    {
      if(inChar == ';')
        contorPunctVirgula++;
      inFisier.get(inChar);
    }
    cout << contorPunctVirgula << endl;
    inFisier.get(inChar);
  }
  return 0;
}
```

Să notăm că am omis prima citire pentru bucla interioară. Aceasta a fost deja făcută înaintea buclei exterioare. Dacă o făceam, primul caracter citit s-ar fi pierdut înainte de a-l testa.

Şablonul sintactic al buclelor imbricate este:

```
Inițializarea_buclei_exterioare
while(condiția_buclei_exterioare)
{
  ...
  Inițializarea_buclei_interioare
  while(condiția_buclei_interioare)
  {
    Procesarea_şi_actualizarea_buclei_interioare
  }
  ...
  Actualizarea_buclei_exterioare
}
```

Fiecare buclă are propria inițializare, testare şi actualizare. Se poate ca bucla externă să nu facă nicio procesare. Pe de altă parte, bucla interioară poate fi o mică parte a procesării realizate de bucla exterioară.