Cuprins

2. Instrucțiunile limbajului C/C++

2.1. *Instrucțiunea expresie*

```
expresie;
```

Efect

Se evaluează expresia.

Exemple

```
i++;        //se incrementeaza valoarea variabilei i
p*=2;       //se inmulteste cu 2 valoarea variabilei p
clrscr();  //se sterge ecranul
```

Observații

1. Ultima expresie este constituită dintr-un apel de funcție. Pentru a apela funcția standard `clrscr()`, trebuie să includem la începutul programului fișierul antet `conio.h`, în care se găsește prototipul (declarația) funcției:

```
void clrscr(void);
```

ceea ce înseamnă că funcția nu are nici un parametru și nu întoarce nici un rezultat.
2. Este permis ca expresia să fie vidă. Instrucțiunea devine `;` (instrucțiunea vidă).
3. Este permisă utilizarea oricărei expresii sintactic corecte, chiar și atunci când instrucțiunea nu generează nici un efect, de exemplu: `2+5;`
4. Orice instrucțiune din limbajul C/C++ se termină cu caracterul `';'`.

2.2. *Instrucțiunea compusă*

```
{
 declaratii
 instructiune_1
 instructiune_2
   ...
 instructiune_n
}
```

Efect

Se execută în ordine instrucțiunile specificate.

Observații

1. Utilizarea instrucțiunilor compuse este necesară atunci când sintaxa permite executarea unei singure instrucțiuni, dar este necesară efectuarea mai multor operații (de exemplu, într-o instrucțiune `if`, `for`, `while` sau `do-while`).
2. Declarațiile care apar într-o instrucțiune compusă sunt **locale** instrucțiunii. Mai exact, ele sunt valabile numai în corpul instrucțiunii compuse, din momentul declarării lor până la sfârșitul instrucțiunii.

2.3. *Instrucțiunea* `if`

```
if (expresie) instructiune_1
    else instructiune_2
```

Efect

Se evaluează expresia. Dacă valoarea expresiei este diferită de `0`, se execută `instructiune_1`, altfel se execută `instructiune_2`.

Exemplu

```
if (a) cout << a << " este nenul" << endl;
    else cout << a << " este nul" << endl;
```

Observații

1. Expresia se încadrează obligatoriu între paranteze rotunde.
2. Dacă `instructiune_2` este vidă, ramura `else` poate să lipsească, obținându-se o formă simplificată a instrucțiunii `if`:

```
if (expresie) instructiune
```

Exemplu

Să calculăm în variabila `max` maximul dintre `a` și `b`:

```
max=a;
if (max<b) max=b;
```

2.4. *Instrucțiunea* `switch`

```
switch    (expresie)
{ case    expresie-constanta1: secventa-instructiuni1
  case    expresie-constanta2: secventa-instructiuni2
    ...
  case    expresie-constantan: secventa-instructiunin
  default:                     secventa-instructiunin+1
}
```

Efect

Se evaluează expresia.

Se compară succesiv valoarea expresiei cu valorile expresiilor constante care etichetează alternativele `case`. Dacă se întâlnește o alternativă `case` etichetată cu valoarea expresiei, se execută secvența de instrucțiuni corespunzătoare și toate secvențele de instrucțiuni care urmează, până la întâlnirea instrucțiunii `break` sau până la întâlnirea acoladei închise, care marchează sfârșitul instrucțiunii `switch`. Dacă nici una dintre valorile etichetelor alternativelor `case` nu coincide cu valoarea expresiei, se execută secvența de instrucțiuni de pe ramura `default`[13].

Exemplu

În funcție de valoarea variabilei de tip `char` `c` (`'+'`, `'-'`, `'*'` sau `'/'`), vom efectua operația corespunzătoare între variabilele `x` și `y`. Dacă variabila `c` are valoarea `'~'` sau `'!'`, vom da mesajul `"Nu e operator binar!"`, iar dacă `c` are orice altă valoare, vom da mesajul `"Eroare"`.

```
switch (c)
{
     case '*':   x *= y; break;
     case '/':   x /= y; break;
     case '+':   x += y; break;
     case '-':   x -= y; break;
     case '~':
     case '!':   cout << "Nu e operator binar!"; break;
     default :   cout << "Eroare!";
}
```

Observații

1. Expresia se încadrează obligatoriu între paranteze rotunde.
2. Pe fiecare alternativă `case` este permisă executarea mai multor instrucțiuni.
3. Dacă $\texttt{secventa-instructiuni}_{n+1}$ este vidă, ramura `default` poate lipsi (similar ramurii `else`).
4. Instrucțiunea `switch` este o generalizare a instrucțiunii `if`. Spre deosebire de `if`, care permite selectarea unei alternative din maximum două posibile, `switch` permite selectarea unei alternative din maximum `n+1` posibile. O altă diferență majoră constă în faptul că în `if` se execută instrucțiunea corespunzătoare valorii expresiei și atât, în timp ce în `switch` se execută și toate secvențele de instrucțiuni ale alternativelor `case` următoare.

13. În limba engleză, *default* înseamnă lipsă. Deci este alternativa care se alege în lipsă de altceva. Noi o vom numi alternativă implicită.

2.5. *Instrucțiunea* `break`

```
break;
```

Efect

Determină ieșirea necondiționată din instrucțiunea `switch`, `while`, `for` sau `do-while` în care apare.

2.6. *Instrucțiunea* `while`

```
while (expresie)
 instructiune
```

Efect

Pas 1: se evaluează expresia;
Pas 2: dacă valoarea expresiei este `0`, se iese din instrucțiunea `while`;
dacă valoarea expresiei este diferită de `0`, se execută instrucțiunea, apoi se revine la Pas 1.

Observații

1. Instrucțiunea se execută repetat cât timp valoarea expresiei este nenulă. Pentru ca ciclul să nu fie infinit, este obligatoriu ca instrucțiunea care se execută să modifice cel puțin una dintre variabilele care intervin în expresie, astfel încât aceasta să poată lua valoarea `0`, sau să conțină o operație de ieșire necondiționată din ciclu (de exemplu, `break;`).
2. Dacă expresia are de la început valoarea `0`, instrucțiunea nu se execută nici măcar o dată.
3. Sintaxa permite executarea în `while` a unei singure instrucțiuni, prin urmare, atunci când este necesară efectuarea mai multor operații, acestea se grupează într-o singură instrucțiune compusă.

Exemplu

Să calculăm răsturnatul numărului natural `n`. Răsturnatul unui număr se obține considerând cifrele numărului în ordine inversă (de la dreapta la stânga).

```
r=0;                  // in r vom obtine rasturnatul lui n
while (n)
  { r=r*10+n%10;      //"lipesc" ultima cifra din n la r
    n/=10;            //elimin ultima cifra din n
  }
```

Să urmărim pas cu pas execuția acestei secvențe de instrucțiuni, pentru `n=237`:

Linia	Efect	n	r
1	Se inițializează variabila `r` cu `0`.	237	0
2	Se testează dacă `n≠0`, apoi se execută instrucțiunea compusă.	237	0
3	Se adaugă la sfârșitul lui `r` ultima cifră din `n`.	237	7
4	Se elimină ultima cifră din `n`.	23	7
2	Se testează dacă `n≠0`, apoi se execută instrucțiunea compusă.	23	7
3	Se adaugă la sfârșitul lui `r` ultima cifră din `n`.	23	73
4	Se elimină ultima cifră din `n`.	2	73
2	Se testează dacă `n≠0`, apoi se execută instrucțiunea compusă.	2	73
3	Se adaugă la sfârșitul lui `r` ultima cifră din `n`.	2	732
4	Se elimină ultima cifră din `n`.	0	732
2	Se testează dacă `n≠0`; se iese din `while`.	0	732

2.7. *Instrucțiunea* `do-while`

```
do
  instructiune
while (expresie);
```

Efect

Pas 1: se execută instrucțiunea;
Pas 2: se evaluează expresia;
Pas 3: dacă valoarea expresiei este `0`, se iese din instrucțiunea repetitivă;
dacă valoarea expresiei este diferită de `0`, se revine la Pas 1.

Observație

Spre deosebire de `while`, instrucțiunea `do-while` execută instrucțiunea specificată cel puțin o dată, chiar dacă de la început valoarea expresiei este 0, deoarece evaluarea expresiei se face după execuția instrucțiunii.

În afară de momentul evaluării expresiei (înainte sau după executarea instrucțiunii), alte diferențe între cele două instrucțiuni repetitive nu există. Prin urmare, preferăm să utilizăm instrucțiunea `while` când este necesar să testăm o condiție înainte de efectuarea unor prelucrări. Preferăm să utilizăm `do-while` când condiția depinde de la început de prelucrările din ciclu și, prin urmare, este necesar să o testăm după executarea instrucțiunii.

Exemplu

Să numărăm cifrele numărului natural memorat în variabila `n`:

```
nr=0;              //in nr vom obtine numarul de cifre
do
  {n/=10;          //elimin ultima cifra din n
   nr++;}          //o numar
while (n); //actiunea se repeta cat timp n mai are cifre
```

Dacă în locul instrucțiunii `do-while` am fi utilizat o instrucțiune `while`, algoritmul nu ar fi funcționat și pentru `n=0`, deoarece testând condiția la început, nu s-ar fi executat nici o dată instrucțiunea din ciclu, deci valoarea variabilei `nr` ar fi rămas `0` (greșit! numărul `0` are o cifră!).

2.8. *Instrucțiunea* `for`

```
for (expresie_initializare; expresie_continuare; expresie_reinitializare)
 instructiune
```

Efect

Pas 1: se evaluează expresia de inițializare;
Pas 2: se evaluează expresia de continuare;
Pas 3: dacă valoarea expresiei de continuare este `0`, se iese din instrucțiunea repetitivă `for`;
dacă valoarea expresiei de continuare este diferită de 0:
– se execută instrucțiunea;
– se evaluează expresia de reinițializare;
– se revine la Pas 2.

Observații

Instrucțiunea `for` este tot o instrucțiune repetitivă, ca și `while` și `do-while`. Nu este o instrucțiune strict necesară, ca poate fi simulată cu instrucțiunea `while` astfel:

```
expresie_initializare;
while (expresie_continuare)
      {  instructiune
         expresie_reinitializare;  }
```

Totuși, majoritatea programatorilor preferă să utilizeze instrucțiunea `for`, deoarece este un instrument puternic, flexibil, care ne permite să exprimăm foarte concis prelucrări de natură repetitivă.

Exemplu

Fie `n` un număr natural (`n<10`) citit de la tastatură. Să se calculeze:

`n!=1*2*...*n` (`n` factorial). Prin definiție, `0!=1`.

Observăm că pentru orice `n>0` are loc relația:

`n!=1*2*...*(n-1)*n=(1*2*...*(n-1))*n=(n-1)!*n`