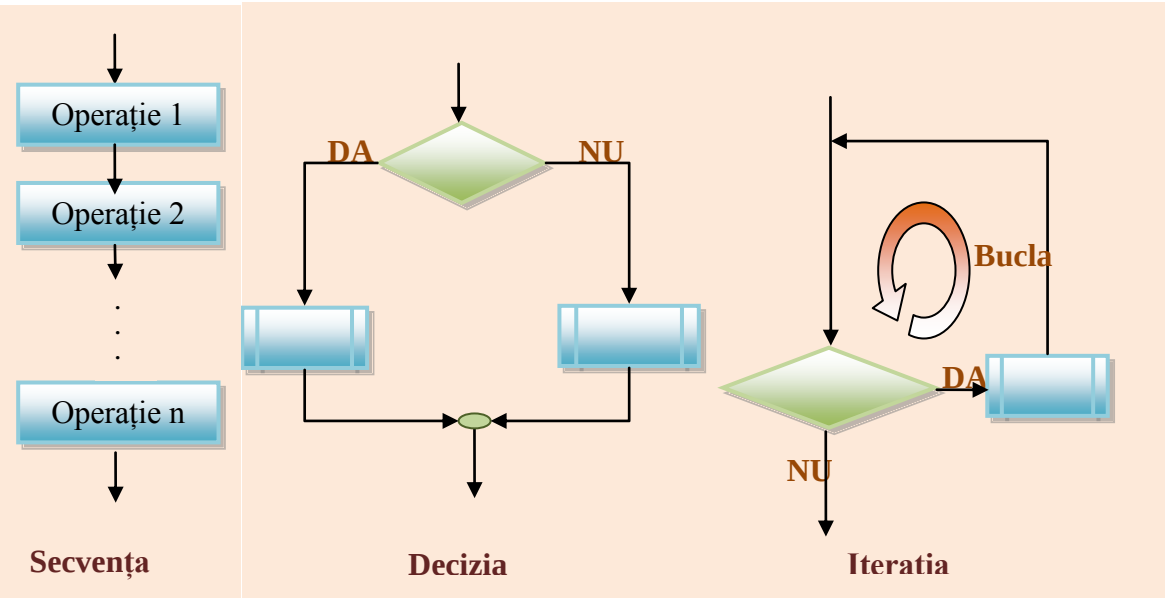
Modul 4 – Instrucțiunile limbajului C

Introducere

După cum spuneam, există trei tipuri de construcții de bază pentru controlul fluxului operațiilor: *secvența*, *decizia* (condiția) şi *iterația* (bucla, ciclu, repetiția), aşa cum sunt ilustrate mai jos.



Instrucțiunea expresie

O expresie urmată de caracterul terminator de instrucțiune ';' devine o instrucțiune expresie.

Sintaxa:
expresie;

Cele mai importante cazuri de instrucțiuni expresie sunt:

Tip instrucțiune expresie	Descriere	Exemple
Instrucțiunea vidă	• conține doar terminatorul ';' • este folosită pentru a marca absența unei prelucrări într-o altă instrucțiune (*if*, *while*, etc)	for (i=0; i<10000; i++) ; /* temporizare, de 10000 de ori nu fac nimic */
Instrucțiunea de apelare a unei funcții	un apel de funcție terminat cu ';'	getchar(); r=sqrt(a); system("pause");
Instrucțiunea de atribuire	o expresie de atribuire terminată cu ';'	a=1; ++a; c=r/(a+b); i=j=k=1;

Utilizarea neatentă a caracterului punct-şi-virgulă poate introduce uneori erori grave (nesemnalate de compilator), dar alteori nu afectează execuția (fiind interpretat ca o instrucțiune vidă).

Exemple:

```
char  a = '1', b ='c';
printf (" %c \n %c \n", a, b);

int a,b,c,m,n,p=2;
// liniile de mai jos reprezinta instructiuni expresie
scanf("%d",&a); // apel de functie, valoarea returnata nu este folosita
b = 5;
c = a > b ? a:b;
n = printf("%d %d %d\n",a,b,c); //valoarea returnata este memorata in n
p = a*b/c;
p++;
m = p+ = 5;
a+b; /* valoarea expresiei nu este folosita - apare un avertisment ( warning ) la compilare: Code has no effect */
```

Instrucțiunea compusă (bloc)

Pe parcursul elaborarii programelor, intervin situatii cand sintaxa impune folosirea unei singure instructiuni, iar codificarea necesita prezenta mai multora - instructiunile se intercaleaza intre acolade, formand un bloc ce grupează mai multe instrucțiuni (şi declarații).

Sintaxa:

```
{
        declarații_variabile_locale_blocului // opționale, valabile doar în fişiere cpp!
        instrucțiuni
}
```

Observații:

- *Corpul* oricărei funcții este un bloc;
- Instrucțiunile unui bloc pot fi de orice tip, deci şi alte instrucțiuni bloc; instrucțiunile bloc pot fi deci *incuibate*;
- Un bloc corespunde structurii de control *secvență* de la schemele logice;
- O instrucțiune bloc poate să nu conțină nici o declarație sau instrucțiune între acolade; în general un astfel de bloc poate apare în faza de punere la punct a programului (funcții cu corp vid);
- Acoladele nu modifică ordinea de execuție, dar permit tratarea unui grup de instrucțiuni ca o singură instrucțiune de către alte instrucțiuni de control (*if*, *while, do, for* ş.a). Instrucțiunile de control au ca obiect, prin definiție, o singură instrucțiune. Pentru a extinde domeniul de acțiune al acestor instrucțiuni la un grup de operații se foloseşte instrucțiunea compusă.
- Un bloc poate conține doar o singură instrucțiune, aceasta pentru punerea în evidență a acțiunii anumitor instrucțiuni.
- Un bloc nu trebuie terminat cu ';' dar nu este greşit dacă se foloseşte (este interpretat ca instrucțiune vidă).
- Dacă un bloc conține doar instrucțiuni expresie, el se poate înlocui cu o instrucțiune expresie în care expresiile inițiale se separă prin operatorul secvențial

Exemple:

```
{ int t; t=a; a=b; b=t; }      // schimba a şi b prin t
// sau:
{
      int t;
      t=a;
      a=b;
      b=t;
}

//Blocul:
{
      a++;
      c=a+ --b;
      printf("%d\n",c);
}
// este echivalent cu instructiunea expresie:
a++, c=a+ --b, printf("%d\n",c);
```

Instrucțiuni de decizie (condiționale)

Există următoarele tipuri de instrucțini de decizie *if-then*, *if-then-else*, *if încuibat* (*if-elseif-elseif-...-else*), *switch-case.*

Instrucțiunea *if*

Instrucțiunea introdusă prin cuvântul cheie *if* exprimă o decizie binară şi poate avea două forme: o formă fără cuvântul *else* şi o formă cu *else*:

Sintaxa:

```
If (expresie)
        instructiune1
else
        instructiune2

SAU:
If (expresie)
        instructiune
```

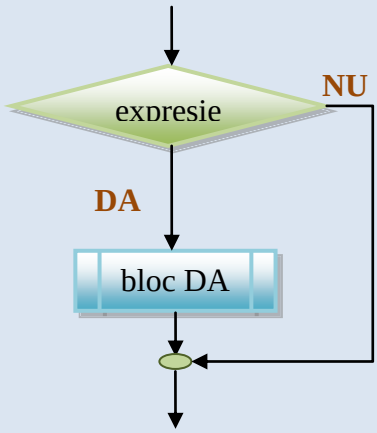
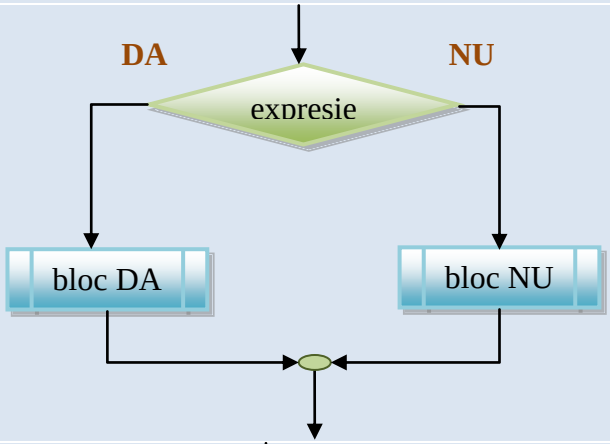
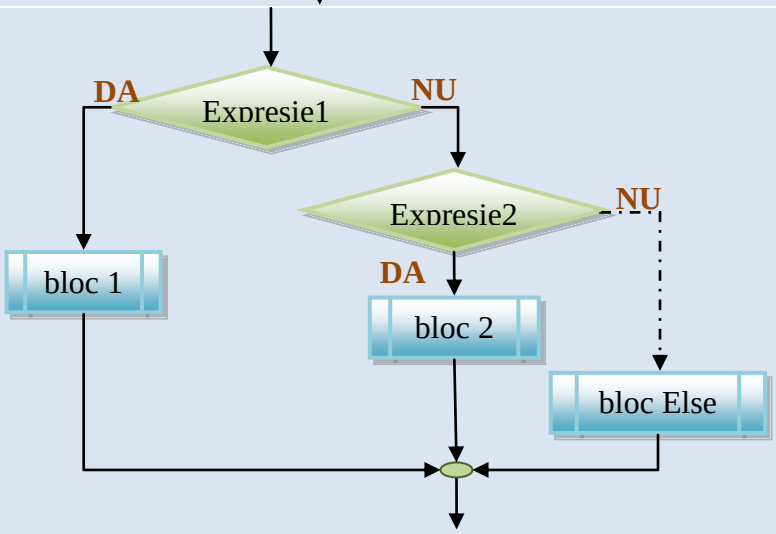
Semantica:

Se evaluează expresie; dacă valoarea ei este adevărat (diferită de 0) se execută instrucțiune1, altfel, dacă există ramura else, se execută instrucțiune2.

Observații:

- Instrucțiunea corespunde structurii de control *decizie* din schemele logice;
- Pentru ca programele scrise să fie cât mai clare este bine ca instrucțiunile corespunzătoare lui if si else sunt de obicei scrise pe liniile următoare şi sunt deplasate spre dreapta, pentru a pune în evidentă structurile şi modul de asociere între *if* şi *else*. Acest mod de scriere permite citirea corectă a unor cascade de decizii.
- Valoarea expresiei dintre paranteze se compară cu zero, iar instrucțiunea care urmează se va executa numai atunci când expresia are o valoare nenulă. În general expresia din instrucțiunea *if* reprezintă o condiție, care poate fi adevărată (valoare nenulă) sau falsă (valoare nulă). De obicei expresia este o expresie de relație (o comparație de valori numerice) sau o expresie logică care combină mai multe relații într-o condiție compusă dar poate fi orice expresie cu rezultat numeric.
- Instrucțiunea corespunzătoare valorii adevărat sau fals, poate fi orice instrucțiune C:
    - instrucțiune expresie terminată cu simbolul ;
    - instrucțiunea bloc (atunci când trebuie executate mai multe prelucrări)
    - alta instrucțiune de decizie - deci instrucțiunile if-else pot fi încuibate; fiecare else corespunde if-ului anterior cel mai apropiat, fără pereche.
- O problemă de interpretare poate apare în cazul a două (sau mai multe) instrucțiuni *if* incluse, dintre care unele au alternativa *else*, iar altele nu conțin pe *else*. Regula de interpretare este aceea că *else* este asociat cu cel mai apropiat *if* fără *else* (dinaintea lui).

O sinteză a celor trei moduri de utilizare a lui *if* precum şi fluxul de operații în schemă logică sunt date în cele ce urmează:

Sintaxă	Flux operații
// if-then **if (*expresie*) {** **bloc_DA;** **}**	
// if-then-else **if (*expresie*)** **bloc_DA ;** **else** **bloc_NU ;**	
// if incuibat **if (*expresie_1*)** ***bloc_1* ;** **else** **if (*expresie_2*)** ***bloc_2* ;** **else** **if (*expresie_3*)** ***bloc_3* ;** **else** **if (*expresie_4*)** **.....** **else** ***bloc_Else* ;**	

Exemple de utilizare a lui *if*:

Sintaxă	Exemple
// if-then **if (*expresie*)** **bloc_DA;**	if (nota >= 5) { printf("Congratulation!\n"); printf("Keep it up!\n"); }
// if-then-else **if (*expresie*)** **bloc_DA ;** **else** **bloc_NU ;**	if (nota >= 5) { printf("Congratulation!\n"); printf("Keep it up!\n"); } else printf("Try Harder!\n");
// if incuibat **if (*expresie_1*)** ***bloc_1* ;** **else** **if (*expresie_2*)** ***bloc_2* ;** **else** **if (*expresie_3*)** ***bloc_3* ;** **else** **if (*expresie_4*)** **.....** **else** ***bloc_Else* ;**	if (nota >= 80) printf("A\n"); else if (nota >= 7) printf("B\n"); else if (nota >= 6) printf("C\n"); else if (nota >= 5) printf("D\n"); else printf("E\n");

Exemple:

```
/* urmatoarele trei secvente echivalente verifica daca trei valori pot
reprezenta lungimile laturilor unui triunghi */
```

- ```
  if(a<b+c && b<a+c && c<a+b)
        puts("pot fi laturile unui triunghi");
  else
        puts("nu pot fi laturile unui triunghi");
  ```
- ```
  if(a<b+c && b<a+c && c<a+b)
        ;
  //pentru cond adevarata nu se executa nimic: apare instructiunea vida
  else
        printf("nu ");
  puts("pot fi laturile unui triunghi");
  ```
- ```
  if(!(a<b+c && b<a+c && c<a+b))  // sau if(a>=b+c || b>=a+c || c>=a+b)
        printf("nu ");
  puts("pot fi laturile unui triunghi");
  ```
  ```

```
// Pentru gruparea mai multor instrucțiuni folosim o instrucțiune bloc:
      if ( a > b) { t=a; a=b; b=t; }

/* Pentru comparația cu zero nu trebuie neapărat folosit operatorul de
inegalitate (!=), deşi  folosirea lui poate face codul sursă mai clar:*/
      if (d) return;          //  if (d != 0) return;

// determinare minim dintre doua numere
      if ( a < b)
            min=a;
      else
            min=b;


/* Pentru a grupa o instrucțiune if-else care conține un if fără else
utilizăm o instrucțiune bloc:*/
      if ( a == b ) {
            if (b == c)
                  printf ("a==b==c \n");
      }
      else
            printf (" a==b şi b!=c \n");


// Expresia conținută în instrucțiunea if poate include şi o atribuire:
      if ( d = min2 - min1) printf(„%d”,d);

/*Instrucțiunea anterioară poate fi derutantă la citire şi chiar este
semnalată cu avertisment de multe compilatoare, care presupun că s-a
folosit eronat atribuirea în loc de comparație la egalitate (o eroare
frecventă):*/

      if (i=0) printf( “Variabila i are valoarea 0”);
      else printf( “Variabila i are o valoare diferita de 0”);
```

Instrucțiunea de selecție *switch*

Selecția multiplă (dintre mai multe cazuri posibile), se poate face cu mai multe instrucțiuni *if* incluse unele în altele sau cu instrucțiunea *switch*. Instrucțiunea *switch* face o enumerare a cazurilor posibile (fiecare precedat de cuvântul cheie *case*) folosind o expresie de selecție, cu rezultat întreg. Forma generală este:

Sintaxa:

```
switch (expresie) {
        case c1: prelucrare_1
        case c2: prelucrare_2
        case cn: prelucrare_n
        default: prelucrare_x
}
```

Unde:

- **expresie - de tip întreg, numită expresie selectoare**

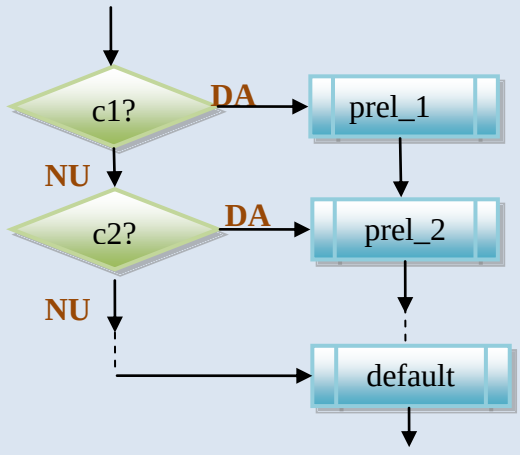
- **c1, cn - constante sau expresii constante întregi (inclusiv „char”)**
- **orice prelucrare constă din 0 sau mai multe instrucțiuni**
- **default e opțional, corespunde unor valori diferite de cele n etichete**

Semantica:
Se evaluează expresie; dacă se găseşte o etichetă având valoarea egală cu a expresiei, se execută atât secvența corespunzătoare acelui caz cât şi secvențele de instrucțiuni corespunzătoare tuturor cazurilor care urmează (chiar dacă condițiile acestora nu sunt îndeplinite) inclusiv ramura de *default*!
Această interpretare permite ca mai multe cazuri să folosească în comun aceleaşi operații. Cazul *default* poate lipsi; în cazul în care avem ramura *default*, se intră pe această ramură atunci când valoarea expresiei de selecție diferă de toate cazurile enumerate explicit.

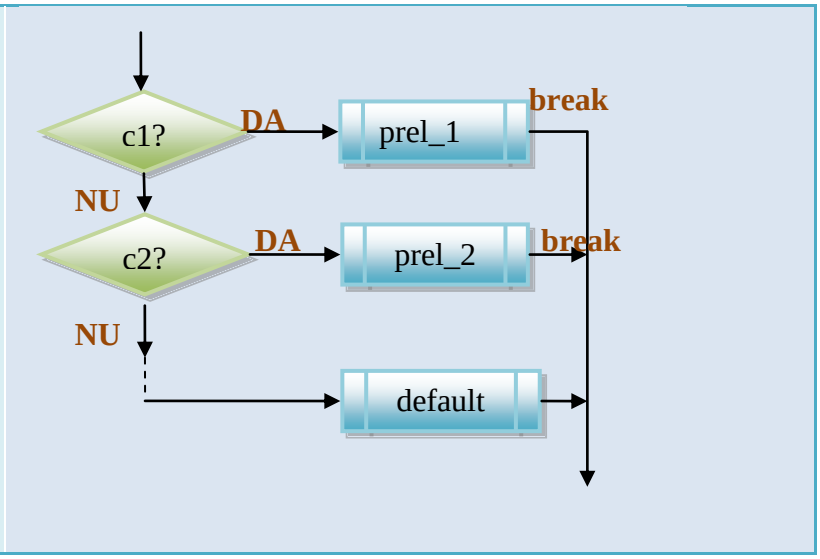
Observație:
Deseori cazurile enumerate se exclud reciproc şi fiecare secvență de instrucțiuni se termină cu *break*, pentru ca după selecția unui caz să se executa doar prelucrarea corespunzatoare unei etichete, nu şi cele următoare:

```
switch (expresie) {
        case c1:        prelucrare_1
                        break;
        case c2:        prelucrare_2
                        break;
        case cn:        prelucrare_n
                        break;
        default:        prelucrare_x ;
}
```

O sinteză a celor două moduri de utilizare a lui *switch* precum şi fluxul de operații în schemă logică este dat în cele ce urmează:

Sintaxă	Flux operații
`// switch-case` `switch (expresie) {` `  case c1:` `  prelucrare_1` `  case c2:` `  prelucrare_2` `  case cn:` `  prelucrare_n` `  default:` `  prelucrare_x` `}`	

```
// switch-case
switch (expresie) {
   case c1:
            prelucrare_1
            break;
   case c2:
            prelucrare_2
            break;
   case cn:
            prelucrare_n
            break;
   default:
            prelucrare_x
}
```



Exemple

```
//calculeaza rezulatul expresiei num1 oper num2
char oper; int num1, num2, result;
......
switch (oper) {
   case '+':
      result = num1 + num2; break;
   case '-':
      result = num1 - num2; break;
   case '*':
      result = num1 * num2; break;
   case '/':
      result = num1 / num2; break;
   default:
      printf("Operator necunoscut");
}


// determina nr de zile dintr-o lună a unui an nebisect
switch (luna) {
      // februarie
      case 2: zile=28; break;

      // aprilie, iunie,..., noiembrie
      case 4: case 6: case 9: case 11: zile =30; break;

      // ianuarie, martie, mai,.. decembrie, celelalte (1,3,5,..)
      default: zile=31;
```

Instrucțiuni repetitive

Există trei tipuri de instrucțiuni de ciclare (bucle, iterații): *while*, *for* şi *do-while*.

Instrucțiunea *while*

Instrucțiunea *while* exprimă structura de ciclu cu condiție inițială şi cu număr necunoscut de paşi şi are forma următoare:

Sintaxa:

while (expresie)
instructiune

Semantica
Se evaluează expresie; dacă valoarea ei este adevărat (diferită de 0) se execută instructiune, după care se evaluează din nou expresie; daca valoarea este 0, se trece la instrucțiunea următoare. Efectul este acela de executare repetată a instrucțiunii conținute în instrucțiunea *while* cât timp expresia din paranteze are o valoare nenulă (este adevărată). Este posibil ca numărul de repetări să fie zero dacă expresia are valoarea zero de la început.

Observatii:

- Instrucțiunea while corespunde structurii repetitive cu test inițial de la schemele logice;
- În general *expresie* conține variabile care se modifică în instructiune, astfel încât expresie să devină falsă, deci ciclarea să nu se facă la infinit;
- În unele programe se poate să apară
  ```
  while(1)
        instructiune
  ```
  Atunci, corpul ciclului poate să conțină o instrucțiune de ieşire din ciclu, altfel tastarea Ctrl/Break întrerupe programul;
- următoarele două instrucțiuni de ciclare sunt derivate din cea cu test inițial;
- Ca şi în cazul altor instrucțiuni de control, este posibil să se repete nu doar o instrucțiune ci un bloc de instrucțiuni;

Exemple:

```
// cmmdc prin incercari succesive de posibili divizori, presupunem a>b

d = b;                  // divizorul maxim posibil este minimul dintre a şi b
while ( a%d || b%d )    // repeta cat timp nici a nici b nu se divid prin d
      d = d -1;         // incearca alt numar mai mic
}
```

În exemplul anterior, dacă `a=8` şi `b=4` atunci rezultatul este `d=4` şi nu se execută niciodată instrucțiunea din ciclu (`d=d-1`).

```
// determinare cmmdc prin algoritmul lui Euclid. While cu instructiune bloc
while (a%b > 0) {
      r = a % b;
      a = b;
      b = r;
}// la ieşirea din ciclu b este cmmdc
```

Expresia din instrucțiunea *while* poate să conțină atribuiri sau apeluri de funcții care se fac înainte de a evalua rezultatul expresiei:

```
// algoritmul lui Euclid rescris
while (r=a%b) {
      a=b;
      b=r;
}
```

Instrucțiunea *for*

Instrucțiunea *for* din C permite exprimarea compactă a ciclurilor cu condiție inițială sau a ciclurilor cu număr cunoscut de paşi şi are forma:

Sintaxa:

for (expresie1; expresie2; expresie3)
instructiune

Semantica:
Se evaluează expresie1 care are rol de *inițializare*; se evaluează apoi *expresie2*, cu rol de condiție - dacă valoarea ei este adevărat (diferită de 0) se execută *instructiune* - *corpul ciclului*, după care se evaluează *expresie3*, cu rol de *actualizare*, apoi se evaluează din nou *expresie2*; dacă valoarea este 0, se trece la instrucțiunea următoare. Cu alte cuvinte, *instructiune* se execută atâta timp cât *expresie2* este adevărată, deci de 0 sau mai multe ori.
Efectul acestei instrucțiuni este echivalent cu al secvenței următoare:

```
expresie1;                  // operații de inițializare
while (expresie2) {         // cat timp exp2 !=0 repeta
 instrucțiune;              // instrucțiunea repetata
 expresie3;                 // o instrucțiune expresie
}
```

Observații:

- Instructiunea for permite o scriere mult mai compactă decât celelalte două instrucțiuni de ciclare, fiind foarte des utilizată în scrierea programelor;
- Oricare din cele trei expresii poate lipsi, dar separatorul ; rămâne. Absența expresie2 echivalează cu condiția adevărat, deci 1; in tabelul de mai jos sunt date echivalențele cu instrucțiunea while, pentru cazuri când expresii din sintaxa for lipsesc:

for	while
`for(;expresie;) instructiune`	`while(expresie) instructiune`
`for(;;) instructiune`	`while(1) instructiune`

- Cele trei expresii din instrucțiunea *for* sunt separate prin ';' deoarece o expresie poate conține operatorul virgulă. Este posibil ca prima sau ultima expresie să reunească mai multe expresii separate prin virgule;
- Este posibilă mutarea unor instrucțiuni din ciclu în paranteza instrucțiunii *for*, ca expresii, şi invers - mutarea unor operații repetate în afara parantezei.

- Instrucțiunea for are ca şi caz particular instrucțiunea de ciclare cu contor numărul de cicluri fiind (val_finala-val_initiala)/increment:

for (var_contor = val_initiala; var_contor <= val_finala; var_contor += increment)
instructiune

- Nu se recomandă modificarea variabilei *contor* folosită de instrucțiunea *for* în interiorul ciclului, prin atribuire sau incrementare;
- Pentru ieşire forțată dintr-un ciclu se folosesc instrucțiunile *break* sau *return*;

Exemple:

```
// ştergere linii ecran
for (k=1;k<=24;k++)
      putchar('\n');    // avans la linie noua

// alta secvența de ştergere ecran
for (k=24;k>0;k--)
      putchar('\n');

// calcul factorial de n
for (nf=k=1 ; k<=n ; k++)  nf = nf * k;

// alta varianta de calcul pentru factorial de n
for (nf=1, k=1 ; k<=n ; nf=nf * k, k++) ;  // repeta instr. vida
```

Instrucțiunea *do*

Instrucțiunea *do-while* se foloseşte pentru exprimarea ciclurilor cu condiție finală, cicluri care se repetă cel puțin o dată. Forma uzuală a instrucțiunii *do* este următoarea:

Sintaxa:

do
instructiune
while (expresie) ;

Semantica:

Se execută *instrucțiune* - *corpul ciclului*, apoi se evaluează *expresie* care are rol de condiție - dacă valoarea ei este adevărat (diferită de 0) se execută *instrucțiune*, după care se evaluează din nou expresie; dacă valoarea este 0, se trece la instrucțiunea următoare.
Cu alte cuvinte, *instrucțiune* se execută atâta timp cât *expresie* este adevărată; ca observație, *instrucțiune* se execută cel puțin o dată.

Observații:

- Instrucțiunea echivalează cu structura repetitivă cu test final de la scheme logice;

- Instrucțiunea do-while se utilizează în secvențele în care se ştie că o anumită prelucrare trebuie executată cel puțin o dată;
- Spre deosebire de *while*, în ciclul *do* instrucțiunea se execută sigur prima dată chiar dacă expresia este falsă. Există şi alte situații când instrucțiunea *do* poate reduce numărul de instrucțiuni, dar în general se foloseşte mult mai frecvent instrucțiunea *while*.

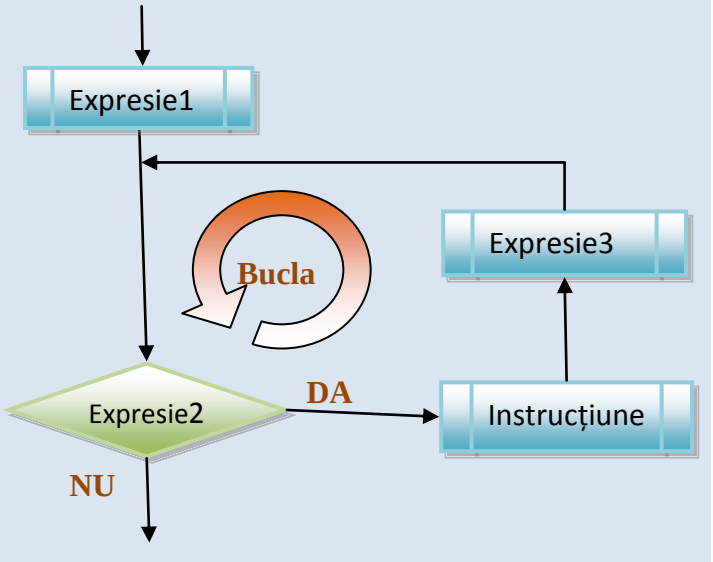
Exemplu:

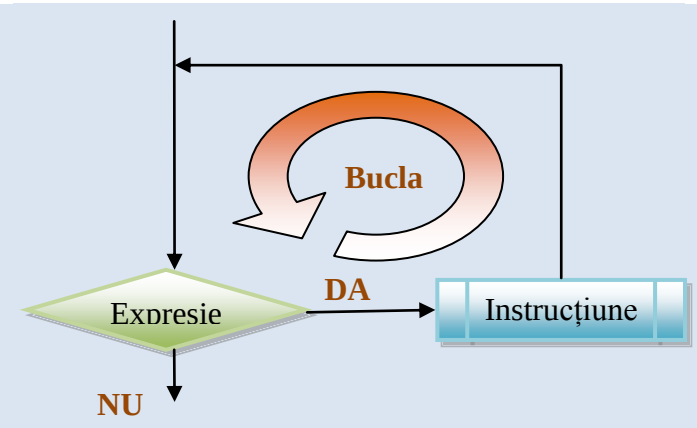
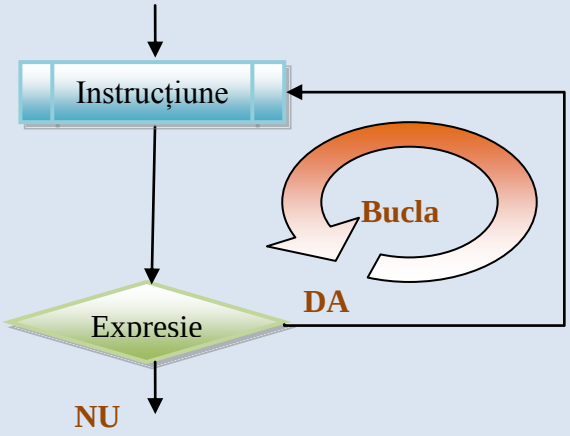
```
// calcul radical din x prin aproximatii succesive
r2=x;                          // aproximatia inițiala
do {
      r1=r2;                   // r1 este aproximatia veche
      r2=(r1+x/r1) / 2;        // r2 este aproximatia mai noua
} while ( abs(r2-r1)) ;        // pana cand r2==r1
```

Un ciclu *do* tipic apare la citirea cu validare a unei valori, citire repetată până la introducerea corectă a valorii respective:

```
do {
      printf ("n (<1000): "); // n trebuie sa fie sub 1000
      scanf("%d", &n);
      if ( n <=0 || n>=1000)
            printf (" Eroare la valoarea lui n ! \n");
} while (n>1000) ;
```

Sinteza instrucțiunilor repetitive

Sintaxă	Flux operații
// for **for (***expresie1; expresie2; expresie3***)** *instrucțiune*	

// while-do **while (** *expresie* **)** *instrucțiune*	
// do-while **do {** instrucțiune **}** **while (** expresie **) ;**	

Exemple scrise utilizând cele trei instrucțiuni repetitive:
1. Suma primelor 1000 de numere naturale
2. Secvențe echivalente care citesc cu validare o variabilă - în urma citirii, variabila întreagă trebuie să aparțină intervalului *[inf,sup]*:

Instrucțiune	Exemple
for	`// Suma de la 1 la 1000` `int suma = 0;` `for (int nr = 1; nr <= 1000; ++nr) {` `  suma += nr;` `}` `// Citire cu validare in intervalul [inf, sup]` `puts("Valoare");` `scanf("%d",&var);` `for(;var < inf || var > sup;){` `  puts("Valoare");` `  scanf("%d",&var);` `}` `// Citire cu validare in intervalul [inf, sup]` `for(puts("Valoare"),scanf("%d",&var);var < inf || var > sup;){` `  puts("Valoare");` `  scanf("%d",&var);` `}` `// Citire cu validare in intervalul [inf, sup]` `for(puts("Valoare"), scanf("%d",&var); var<inf || var>sup;`

	puts(“Valoare”), scanf("%d",&var)); **// Citire cu validare in intervalul [inf, sup]** for(;puts(“Valoare”), scanf("%d",&var), var<inf || var>sup;);
while	**// Suma de la 1 la 1000** int suma = 0, nr = 1; while (nr <= 1000) { suma += nr; ++nr; } **// Citire cu validare in intervalul [inf, sup]** puts(“Valoare”); scanf("%d",&var); while (var < inf || var > sup){ **//valoare invalida, se reia citirea** puts(“Valoare”); scanf("%d",&var); } **// Citire cu validare in intervalul [inf, sup]** while(puts(“Valoare”), scanf("%d",&var),var<inf ||var>sup);
do while	**// Suma de la 1 la 1000** int suma = 0, nr = 1; do { suma += nr; ++nr; } while (nr <= 1000); **// Citire cu validare in intervalul [inf, sup]** do{ puts(“Valoare”); scanf("%d",&var); }while(var<inf || var>sup);

Instrucțiunile *break* şi *continue*

Instrucțiunea *break* determină ieşirea forțată dintr-un ciclu - adică ieşirea din corpul celei mai apropiate instrucțiuni *while*, *for*, *do-while* - sau dintr-un *switch* care o conține, şi trecerea la execuția instrucțiunii următoare.
Sintaxa instrucțiunii este simplă:

Sintaxa:

break;

Semantica
Efectul instrucțiunii *break* este un salt imediat după instrucțiunea sau blocul repetat prin *while, do, for* sau după blocul *switch*.

Observații:

- Un ciclu din care se poate ieşi după un număr cunoscut de paşi sau la îndeplinirea unei condiții (ieşire forțată) este de obicei urmat de o instrucțiune *if* care stabileşte cum s-a ieşit din ciclu: fie după numărul maxim de paşi, fie mai înainte datorită satisfacerii condiției.

Exemplu:

```
// verifica daca un numar dat n este prim
for (k=2; k<n;k++)
      if ( n%k==0)  break;
      //daca gasim un divizor, iesim, n nu este prim!
if (k==n) printf ("prim \n"); /*s-a iesit normal din ciclu-nu are
                                                        divizor*/
else printf ("neprim \n"); /*s-a iesit fortat prin break - are divizor */
```

- Utilizarea instrucțiunii *break* poate simplifica expresiile din *while* sau *for* şi poate contribui la urmărirea mai uşoară a programelor, deşi putem evita instrucțiunea *break* prin complicarea expresiei testate în *for* sau *while*. Secvențele următoare sunt echivalente:

```
//se iese cand e este diferita de 0
for (k=0 ; k<n; k++)
            if (e) break;

for (k=0 ; k<n && !e ; k++);
```

Instrucțiunea *continue* este mai rar folosită față de *break.*

Sintaxa:

continue;

Semantica

Efectul instrucțiunii *continue* este opirea iterației curente a ciclului şi un salt imediat la prima instrucțiune din ciclu, pentru a continua cu următoarea iterație. Nu se iese în afara ciclului, ca în cazul instrucțiunii *break*.

În exemplu următor se citeşte repetat un moment de timp dat sub forma ora, minut, secundă până la introducerea unui momnet corect (care are toate cele 3 componente: h, m s şi pentru care ora (h) se incadrează între 0 şi 24, minutele şi secundele (m, s) între 0 şi 59. Este realizată validarea doar pentru oră:

```
int h,m,s;
int corect=0;  // initial nu avem date corecte - nu avem de fapt deloc date

while ( ! corect ) {
// atata timp cat nu s-au citit date corecte
      printf (" ore, minute, secunde: ");
      if ( scanf("%i%i%i", &h, &m, &s) !=3 ) {
      //nu s-au citit 3 nr intregi
            printf (" Eroare - insuficiente date numerice\n");
            fflush(stdin);    //stergere buffer de intrare
            continue;  // salt peste instructiunile urmatoare, reia citirea
      }
```

```
      if (h < 0 || h > 24) {
            printf (" Valoare incorecta pentru ora!\n");
            fflush(stdin);   //stergere buffer de intrare
            continue;  // salt peste instructiunile urmatoare, reia citirea
            }
      .... // testare m si s intre 0 si 59
      corect=1;
}
```

Observații:

Uneori se recomandă să evităm utilizarea instrucțiunilor *break* şi *continue* deoarece programele care le folosesc sunt mai greu de citit şi de înțeles. Întotdeauna putem scrie acelaşi program fără să folosim *break* şi *continue*.

Exemplu:

```
// Suma de la 1 la n, excluzand  11, 22, 33,...
int n = 100;
int suma = 0;
for (int nr = 1; nr <= n; nr++) {
   if (nr % 11 == 0) continue;  /* sare peste restul corpului buclei şi
                                   trece la urmat. iterație – nr+1 */
   suma += nr;          // aici ajung doar daca nr nu e divizibil cu 11
}

// Este mai bine să rescriem bucla for astfel:
for (int nr = 1; nr <= n; nr++) {
   if (nr % 11 != 0) suma += nr;
}
```

Terminarea programului

Un program se termină în mod normal în momentul în care s-au executat toate instrucțiunile sale. Dacă dorim să forțăm terminarea lui, putem folosi funcția *exit* sau instrucțiunea *return*.

Funcția ***exit*** are următoarea sintaxă:

Sintaxa:

exit();

sau:

exit(int codIesire);

Semantica:

Termină programul şi returnează controlul sistemului de operare (OS). Prin convenție, returnarea codului 0 indică terminarea normală a programului, în timp ce o valoare diferită de zero indică o terminare *anormală*.

Exemplu:

```
if (nrErori > 10) {
   printf( "prea multe erori!\n");
```

```
   exit(1);     // Terminarea programului
}
```

Instrucțiunea ***return*** are următoarea sintaxă:

Sintaxa:

return;
sau:
return expresie;

Semantica:
Se revine din funcția care conține instrucțiunea, în cea apelantă, la instrucțiunea următoare apelului; se returnează valoarea expresiei pentru cazul al doilea.
Putem folosi instrucțiunea "return *valoareReturnata;*" în funcția *main*() pentru a termina programul.

Exemplu:

```
int main() {
   ...
   if (nrErori > 10) {
      printf( "prea multe erori!\n");
      return 1;        // Termina programul si reda controlul OS
   }
   ...
}
```

În continuare, găsiți două anexe, una cu sfaturi practice pentru devoltarea programelor C (good practices) şi modul de depanare a programelor C în Netbeans şi CodeBlocks, iar cea de-a doua cu programele din cursul 1 rezolvate în C:

Anexa A
Sfaturi practice pentru devoltarea programelor C. Depanare

Este important să scriem programe care produc rezultate corecte, dar de asemenea este important să scriem programe pe care alții (şi chiar noi peste câteva zile) să le putem înțelege, astfel încăt să poată fi uşor întreținute. Acesta este ceea ce se numeşte un program bun.
Iată câteva sugestii:

- Respectă conveția stabilită deja la proiectul la care lucrezi astfel încăt întreaga echipă să respecte aceleaşi reguli.
- Formatează codul sursă cu indentare potrivită, cu spații şi linii goale. Foloseşte 3 sau 4 spații pentru indentare şi linii goale pentru a marca secțiuni diferite de cod.
- Alege nume bune şi descriptive pentru variabile si funcții: coloană, linie, xMax, numElevi. Nu folosiți nume fără sens pentru variabile, cum ar fi a, b, c, d. Evitați nume de variabile formate doar dintr-o literă (mai uşor de scris dar greu de înțeles), excepție făcând nume uzuale cum ar fi coordonatele x, y, z şi nume de index precum i.
- Scrie comentarii pentru bucățile de cod importante şi complicate. Comentează propriul cod cât de mult se poate.
- Scrie documentația programului în timp ce scrii programul.
- Evită construcțiile nestructurate, cum ar fi break şi continue, deoarece sunt greu de urmărit.

Erori în programare
Există trei categorii de erori în progamare:

- Erori de compilare (sau de sintaxă): pot fi reparate uşor.
- Erori de rulare: programul se opreşte prematur fără a produce un rezultat – de asemenea se repară uşor.
- Erori de logică: programul produce rezultate eronate. Eroarea este uşor de găsit dacă rezultatele sunt eronate mereu. Dar dacă programul produce de asemenea rezultate corecte cât şi rezultate eronate câteodată, eroare este foarte greu de identficat.

  Acest tip de erori devine foarte grav dacă nu este detectat înainte de utilizarea efectivă a programului în producție. Implementarea unor programe bune ajută la mimimizarea şi detectarea acestor erori. O startegie de testare bună este necesară pentru a certifica corectitudinea programului.

Programe de depanare
Exista câteva tehinci de depanare a programaelor:

1. Uită-te mult la cod! Din păcate, erorile nu o să-ți sară în ochi nici dacă te uiți destul de mult.
2. Nu închide consola de erori, când apar mesaje pretinzând că totul este în regulă. Analizează mesajele de eroare! Asta ajută de cele mai multe ori.

3. Inserează în cod afişări de variabile în locuri potrivite pentru a observa valori intermediare. Este folositor pentru programe mici, dar la programe complexe îşi pierde din eficienţă.
4. Foloseşte un depanator grafic. Aceaste este cea mai eficienta metodă. Urmăreşte execuția programului pas cu pas urmărind valorile variabilelor.
5. Foloseşte unelte avansate pentru a descoperi scurgeri de memorie sau nealocarea ei.
6. Testează programul cu valori de test utile pentru a elimina erorile de logică.

Testarea programului pentru a vedea dacă este corect

Cum te poți asigura că programul tău produce rezultate corecte mereu? Este imposibil să încerci toate variantele chiar şi pentru un program simplu. Testarea programului foloseşte de obicei un set de teste reprezentative, care sunt făcute pentru a detecta clasele de erori majore.

În continuarea acestui material găsiți modul de depanare a programelor C în Netbeans:
Ecuatia de grad 1
Suma primelor n numere naturale

şi CodeBlocks:
Inversul unui numar natural
Interschimbarea valorilor- Problema paharelor

Anexa B

Programele din laboratorul 1 rezolvate în C