Cursul 9

Instrucţiuni C\C++
(partea I)

Descrierea acţiunilor de prelucrare a datelor care au loc în corpul unei funcţii se realizează cu ajutorul *instucţiunilor*, care sunt executate în ordinea dată de *fluxul de control* al execuţiei. Instrucţiunile sunt executate în mod *secvenţial,* în ordinea în care sunt scrise în textul sursă, atât timp cât fluxul de control nu este ramificat de către o *selecţie*, de o *ciclare* sau de un *salt*.

O *instrucţiune simplă* este formată din *cuvinte cheie*, *expresii* şi eventual una sau două instrucţiuni corp. De exemplu instrucţiunea ***if*** are o singură instrucţiune corp iar ***if-else*** are două. Există un singur tip de *instrucţiune compusă*, ea începe cu *simbolul* '{' urmat de un *bloc de instrucţiuni* (posibil vid) şi se termină cu *simbolul* '}'. Cele două acolade sunt părţi componente ale instrucţiunii. Orice altă instrucţiune începe cu un cuvânt cheie sau cu o expresie şi se termină sau cu *simbolul punct şi virgulă* ';' sau cu o instrucţiune corp. In cazul în care apare, simbolul punct şi virgulă constitue *terminatorul* instrucţiunii şi este parte componenă a respectivei instrucţiuni. O situaţie de excepţie o constitue *instrucţiunea nulă*, care este formată numai din terminatorul punct şi virgulă.

In vederea realizării unui salt la o anumită instrucţiune, acea instrucţiune trebuie prefixată cu una sau mai multe *etichete*. O etichetă constă dintr-un identificator urmat de *simbolul două puncte* ':', etichetele au propriul spaţiu de nume (acelaşi identificator poate fi folosit pentru o etichetă şi pentru o variabilă fără să interfereze), au ca domeniu de vizibilitate întregul corp al funcţiei în care apar şi sunt recunoscute numai de instrucţiunea de salt ***goto***.

I. Instrucţiuni secvenţiale

1. Instrucţiunea declaraţie

Sintaxa: *declaraţie***;**

Iniţial, în limbajul C, declaraţiile nu aveau statut de instrucţiune, iar asupra declaraţiilor acţionau tot felul de restricţii; de exemplu, declaraţiile de variabile erau constrânse să apară numai la începutul blocului din care faceau parte. Odată cu trecerea la C++ aceste restricţii au fost ridicate, declaraţiile pot să apară acum în cele mai neaşteptate locuri, chiar şi în condiţia unui ***if***, de exemplu. Mai mult decât atât, declaraţiile de variabile din corpul unei funcţii au primit statut de instrucţiune, ele au dreptul să apară oriunde poate să apară oricare altă instrucţiune, şi pot fi etichetate pentru un eventual salt la ele cu instrucţiunea ***goto***.

Nu toate declaraţiile care apar în definiţia unei funcţii sunt instrucţiuni: de exemplu declaraţiile parametrilor formali ai unei funcţii nu sunt instrucţiuni.

In programul următor, care calculează şi afişează pe monitor primii 100000 de termeni ai şirului recurent $x_0 = 0.1$, $x_1 = 0.9$, $x_{n+1} = 1 - x_n x^3_{n-1}$, în funcţia ***afiseaza*** sunt declarate cinci variabile, dar dintre aceste declaraţii numai cele ale variabilelor ***i*** şi ***xViit*** sunt instrucţiuni declaraţie:

```
#include<iostream>
using namespace std;
void afiseaza(double xTrec, double xPrez, int nrTermeni){
	for(int i=0; i<nrTermeni; i++){
		double xViit=1-xPrez*xTrec*xTrec*xTrec;
		cout<<xViit<<endl;
		xTrec=xPrez;
		xPrez=xViit;
	}
}
int main(){
	afiseaza(0.1, 0.9, 100000);
	return 0;
}
```

O instrucţiune declaraţie introduce în program una sau mai multe date de un anumit tip. Declaraţiile pot fi simple (declară numai tipul datelor) sau cu iniţializare. Iniţializarea unei variabile simple la declarare poate fi efectuată cu *sintaxa semnului egal* (stilul C) sau cu *sintaxa apelului de funcţie* (stilul C++). In primul stil numele variabilei declarate este urmat de semnul egal şi apoi de o expresie a cărei valoare este asignată noii variabile, iar în al doilea stil numele variabilei este urmat de o pereche de paranteze rotunde care conţin expresia valoare:

```
int a=1, b=2;
cout<<a<<endl;//1
cout<<b<<endl;//2
double c(3), x(a+10*b);
cout<<c<<endl;//3
cout<<x<<endl;//21
```

Tablourile pot fi iniţializate numai în stilul C, cu semnul egal urmat de o pereche de acolade care conţin o listă de expresii sau, în cazul tablourilor de caractere, cu semnul egal urmat de o constantă string:

```
char text[5]={'a','b','c','\0'}, mesaj[]="ABC";
cout<<text<<mesaj<<endl; //abcABC
```

Stilul C++ de iniţializare este specific obiectelor de tip clasă şi va fi studiat odată cu acestea.

Standardele limbajului C++ precizează că o declaraţie în care sunt declarate mai multe variabile este echivalentă cu seria de declaraţii simple care se obţine prin declararea variabilelor din listă una câte una, citite de la stânga la dreapta. De exemplu:

```
int a=1, b=a, c=a+b;
```

este echivalentă cu aceste trei declaraţii

```
int a=1;
int b=a;
int c=a+b;
```

în această ordine. Prin urmare, declaraţia

```
double x=y, y=1;
```

va produce cu siguranţă o eroare de compilare:

```
error C2065: 'y' : undeclared identifier
```

In cazul în care o instrucţiune declaraţie obişnuită intră de mai multe ori în fluxul de execuţie al funcţiei în care apare, alocarea variabilelor declarate de ea se efectuează o singură dată, la intrarea în execuţie a funcţiei respective, spre deosebire de partea de iniţializare care se execută de fiecare dată când instrucţiunea intră în flux.

In programul următor variabila ***k*** este alocată de două ori de către cele două apeluri ale funcţiei ***test***, iar pe durata execuţiei fiecărui apel variabila ***k*** este iniţializată de 3 ori:

```
#include<iostream>
using namespace std;
void test(int a){
	for(int i=0; i<3; i++){
		int k=a;
		cout<<k++<<" ";
	}
	cout<<endl;
	return;
}
int main(){
	test(1);
	test(5);
	return 0;
}
/*REZULTAT
1 1 1
5 5 5
Press any key to continue . . .*/
```

Dacă dorim ca o variabilă să fie iniţializată numai o singură dată în tot programul, la prima intrare în fluxul de control al programului, ea trebuie declarată cu modificatorul ***static***, care schimbă *clasa de memorie* a variabilei:

```
#include<iostream>
using namespace std;
void test(int a){
	for(int i=0; i<3; i++){
		static int k=a;
		cout<<k++<<" ";
	}
	cout<<endl;
	return;
}
int main(){
	test(1);
	test(5);
	return 0;
}
```

```
/*REZULTAT
1 2 3
4 5 6
Press any key to continue . . .*/
```

Observaţi că, deşi ***k*** este variabilă locală funcţiei test, ea îşi conservă valoarea între apelurile funcţiei deoarece a fost declarată statică. Variabilele statice au domeniul de vizibilitate cât al blocului în care sunt declarate dar au timpul de viaţă cât al întregului program, deoarece ele nu sunt alocate pe stivă aşa cum sunt celelalte variabilele locale, aşa numitele *variabile automatice* (cu alocare/dealocare automată) .

Variabilele statice sunt de fapt nişte variabile globale cu restricţii de vizibilitate, ele sunt alocate din start în zona variabilelor globale şi iniţializate implicit cu zero. Iniţializarea explicită a unei variabile statice are loc o singură dată în tot programul, la prima intrare în execuţie a instrucţiunii care le declară. In cazul lor diferenţa dintre *iniţializare* (la declarare) şi *atribuirea unei valori iniţiale* (printr-o instrucţiune expresie de atribuire) este evidentă, comparaţi rezultatul precedent cu cel următor:

```
#include<iostream>
using namespace std;
void test(int a){
	for(int i=0; i<3; i++){
		static int k;
		cout<<k++<<" ";
		k=a;
	}
	cout<<endl;
	return;
}
int main(){
	test(1);
	test(5);
	return 0;
}
/*REZULTAT
0 1 1
1 5 5
Press any key to continue . . .*/
```

Clasa de memorie a unei variabile defineşte timpul de viaţă, domeniul de vizibilitate şi locul de stocare al acesteia. Limbajul C/C++ are definite patru clase de memorie: externă, statică, automatică şi registru, dintre care ultima este perimată, fiind menţinută numai pentru compatibilitate. Clasa de memorie a unei variabile se deduce din context în funcţie de locul declaraţiei (în corpul unei funcţii sau nu) şi din prezenţa sau absenţa următorilor specificatori de stocare: ***extern***, ***static***, ***auto*** şi ***register***.

Variabilele externe sunt cele declarate în afara oricărei funcţii fără nici un specificator de stocare sau cele redeclarate cu specificatorul ***extern***, ele sunt variabile cu vizibilitate globală, fiind vizibile din orice punct al fişierului sursă aflat după locul declaraţiei, şi au timpul de viaţă nelimitat. Sunt alocate la intrarea în execuţie a programului într-o zonă de memorie special destinată lor şi sunt iniţializate în mod implicit cu zero.

Pentru a fi vizibile şi din funcţii definite înaintea declaraţiei lor, variabilele externe pot fi anunţate cu specificatorul ***extern***:

```
#include<iostream>
using namespace std;
int main(){
	extern double pi;
	cout<<pi<<endl;
	return 0;
}
double pi=3.1415;
/*REZULTAT:
3.1415
Press any key to continue . . .*/
```

Domeniul de vizibilitate al unei variabile externe poate depăşi fişierul sursă în care este declarată, pentru a folosi o variabilă externă declarată în alt fişier aceasta trebuie redeclarată în fişierul curent cu specificatorul ***extern***. In acest caz, după compilare, editorul de legături trebuie să găsească declaraţia iniţială a variabilei.

Variabilele statice sunt cele declarate cu specificatorul ***static***. Ele sunt alocate la intrarea în execuţie a programului în zona de memorie a variabilelor externe, şi la fel ca acestea, au tipul de viaţă cât al programului şi sunt iniţializate implicit cu zero. Diferă de variabilele externe numai prin restrângerea vizibilităţii lor: domeniul de vizibilitate al unei *variabile statice globale* (declarată în exteriorul funcţiilor) nu poate depăşi fişierul sursă, iar domeniul de vizibilitate al unei *variabile statice locale* (declarată în interiorul unei funcţii) este restrîns la blocul în care este declarată. După cum am văzut, variabilele statice locale îşi conservă valoarea între apelurile succesive al funcţiei.

Variabilele registru sunt variabilele declarate cu specificatorul ***register***. Acesta dă numai o sugestie compilatorului de alocare a variabilei pe cât posibil în unul dintre regiştrii microprocesorului, pentru a fi accesată cât mai rapid, dar compilatoarele moderne ignoră sistematic această cerinţă. Cuvântul cheie ***register*** este menţinut pentru compatibilitatea cu programele mai vechi.

Variabilele automatice sunt variabilele locale unei funcţii declarate fără nici un specificator de stocare sau declarate explicit cu specificatorul ***auto*.** După cum ştim deja, ele sunt alocate pe stivă la apelul funcţiei şi dealocate automat (de unde şi denumirea) prin coborârea stivei la încetarea apelului. Au timpul de viaţă cât durata apelului şi sunt vizibile numai în blocul în care au fost declarate.

Observaţie: începând cu Microsoft Visual C++ 2010 cuvântul cheie ***auto*** nu mai desemnează o clasă de memorie ci este un specificator de tip implicit şi poate fi folosit numai la declaraţii cu iniţializare, când tipul variabilei poate fi dedus de către compilator din expresia de iniţializare. In următorul exemplu ***a*** este o variabilă de tip ***double***:

```
auto a = 1.1;
int b = a;
// warning C4244: 'initializing' : conversion
// from 'double' to 'int', possible loss of data
```

Revenirea la vechea semnificaţie a lui ***auto*** este posibilă cu ajutorul unei anumite opţiuni de compilare.

Utilizarea variabilelor statice locale în locul variabilelor globale face posibilă definirea unei "stări interne" care se conservă între apelurile succesive ale unei funcţii. In programul următor, în care este implementat un generator standard de numere pseudoaleatoare bazat pe congruenţa liniară $x_{n+1}=ax_n+c$ modulo 2^{32}, cu a = 134775813 şi c = 1, este esenţial faptul că variabila de stare ***x*** este statică:

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

double nextDouble(){
	static const double doiLa32=4294967296e0;
	static const unsigned a=134775813u;
	static const unsigned c=1u;
	static unsigned x=13;
	x=a*x+c;
	return x/doiLa32;
}
int main(){
	cout.precision(12);
	for(int i=0;i<100;i++)
		cout<<nextDouble()<<endl;
	return 0;
}
```

Variabila ***x*** a fost declarată statică pentru a-şi păstra valoarea între apeluri, iar cele trei constante au fost declarate statice pentru a nu fi reiniţializate în mod inutil la fiecare apel şi a-i mări astfel durata de execuţie.

2. Instrucţiunea expresie

Sintaxa: *expresie*;

O instrucţiune expresie provoacă evaluarea expresiei componente, fără să producă ramificaţii în fluxul de execuţie al programului (eventualele apeluri de funcţii din cadrul expresiei introduc o ramificare temporară, dar ele returnează controlul execuţiei în punctul în care s-a întrerupt evaluarea expresiei). Dacă expresia componentă nu are efecte secundare, instrucţiunea expresie respectivă este acceptată de compilator dar nu este tradusă în cod obiect (nu este nimic de executat), altfel se evaluează expresia şi sunt completate (executate) toate efectele secundare înainte de preluarea controlului de către urmatoarea instrucţiune din textul sursă.

Cele mai des utilizate instrucţiuni expresie sunt instrucţiunile expresie de atribuire şi instrucţiunile expresie apel de funcţie. Exemple:

```
int main(void){
	int i=1;
	int j,k;
	k=7;			//instr. expresie (atribuire)
	i<k ? j=0:j=1;		//instr. expresie
	i+j;			//instr. expresie (fără efecte secundare)
	j++;			//instr. expresie (incrementare)
	cout<<j<<endl;		//instr. expresie (apel de funcţie)
	return 0;
}
```

Să remarcăm şi următorul avertisment corespunzător instrucţiunii "`i+j;`" şi anume: `warning C4552: '+' : operator has no effect; expected operator with side-effect`

Instrucţiunea nulă este o instrucţiune expresie fără expresia componentă (sic!), ea se reduce la terminatorul punct şi virgulă, şi este utilă, de exemplu, în situaţia în care corpul unei instrucţiuni de ciclare nu trebuie să execute nimic. Astfel, în urmatorul program funcţia ***lgString*** returnează numărul de caractere al stringului `text`:

```
#include<iostream>
using namespace std;
const int dimMax=20;

int lgString(char text[dimMax]){
      int i;
      for(i=0; i<dimMax && text[i]!='\0'; i++)
            ;           //instrucţiunea nula
      return i;
}
int main (){
      char text[dimMax]="un text oarecare";
      cout<<"nr. de caractere="<<lgString(text)<<endl;
      return 0;
}
```

3. Instrucţiunea compusă

Sintaxa: **{**
instrucţiune
instrucţiune
……..
instrucţiune
}

O *instrucţiune compusă* constă din zero, una sau mai multe instrucţiuni cuprinse în interiorul unei perechi de acolade. Acoladele sunt părţi componente ale instrucţiunii şi nu trebuie urmate de simbolul punct şi virgulă (care ar introduce o instrucţiune nulă inutilă).

Instrucţiunea compusă formează un *bloc*, şi în consecinţă eventualele instrucţiuni declaraţie prezente în corpul instrucţiunii compuse introduc în mod implicit *variabile locale* blocului respectiv.

Exemplu: următoarea funcţie determină valoarea maximă a celor două argumente ale sale şi tipăreşte un numar de caractere, `'#'`sau `'*'`, egal cu valoarea returnată.

```
int max(int a,int b){
      if (a<b){
            for(int i=b; i>0; i--){
                  cout<<'#';
            }
            cout<<endl;
            return b;
      }
```

```
        else{
                for(int i=a; i>0; i--) cout<<'*';
                cout<<endl;
                return a;
        }
}
```

In exemplul de mai sus, ramurile instrucţiunii ***if*** – ***else*** sunt instrucţiuni compuse. Avem şi un exemplu de instrucţiune compusă formată dintr-o singură instrucţiune componentă (corpul primului ***for***), o astfel de instrucţiune este utilă pentru un stil unitar de scriere.

II. Instrucţiuni de selecţie

4. Instrucţiunea *if*

Sintaxa: ***if*** **(***expresie-condiţională***)** *instrucţiune-corp*

Instrucţiunea ***if*** îşi inserează în fluxul secvenţial de execuţie al programului *instrucţiunea sa corp*, dar numai în cazul în care *expresia-condiţională* este adevarată

Execuţia instrucţiunii ***if*** are loc în urmatoarele etape: se evaluează expresia condiţională, care trebuie să fie de tip scalar (aritmetic sau pointer), şi se completează toate efectele sale secundare, dacă rezultatul obţinut este egal cu zero (valoarea logică *fals*) controlul este preluat de instrucţiunea imediat următoare din textul sursă al programului, altfel controlul este dat instrucţiunii corp. După executarea instrucţiunii corp, dacă aceasta nu conţine nici o instrucţiune de salt, controlul este preluat tot de următoarea instrucţiune din program.

Exemplu: funcţia următoare returnează modulul diferenţei argumentelor:

```
int dist(int a,int b){
        int c=a-b;
        if ( a < b ) c=b-a;
        return c;
}
```

5. Instrucţiunea if – else

Sintaxa: ***if*** **(***expresie-conditionala***)** *instrucţiune-corp-DA*
else *instrucţiune-corp-NU*

Instrucţiunea ***if*** – ***else*** inserează în fluxul de execuţie al programului una şi numai una dintre cele două instrucţiuni corp componente, şi anume: dacă expresia conditională este adevarată controlul este preluat de *instrucţiunea-corp-DA*, altfel controlul este preluat de *instrucţiunea-corp-NU*. Analog instrucţiunii ***if***, execuţia începe cu evaluarea expresiei condiţionale şi completarea efectelor sale secundare, iar la terminarea execuţiei

controlul este preluat de instrucţiunea imediat urmatoare în textul sursă, dacă nu a apărut nici o instrucţiune de salt (***break***, ***continue***, ***return*** sau ***goto***).

Exemplul precedent rescris cu ***if*** – ***else***:

```
int dist(int a,int b){
      int c;
      if ( a < b )c=b-a;
      else c=a-b;
      return c;
}
```

De remarcat prezenţa terminatorului punct şi virgulă în cazul instrucţiunii corp DA din exemplul de mai sus: simbolul **';'** face parte din instrucţiunea expresie şi trebuie să fie prezent, el încheie numai instrucţiunea componentă, nu şi instrucţiunea ***if – else***.

Subliniem că exemplul de mai sus are numai scopul de a exemplifica instrucţiunea ***if - else***, funcţia ***dist*** se implementează mult mai concis cu ajutorul operatorului ***if*** aritmetic (care nu trebuie confundat cu ***if***-ul logic):

```
int dist(int a,int b){
      return a<b ? b-a : a-b;
}
```

Sintaxa instrucţiunilor ***if*** şi ***if – else*** nu prevede utilizarea unui terminator de instrucţiune, şi acest fapt poate crea ambiguităţi de interpretare în cazul unei secvenţe de instrucţiuni ***if*** şi ***if – else*** imbricate. Pentru ieşirea din ambiguitate limbajul prevede următoarea regulă de asociere: când întâlneşte un cuvant cheie ***else*****,** compilatorul îl asociază cu primul ***if*** aflat înaintea sa în textul sursă şi care nu este deja asociat cu un ***else***. Dacă rămâne un ***else*** pentru care nu există nici un ***if*** cu care să formeze o pereche ***if – else*****,** apare o eroare de compilare. Este recomandată utilizarea acoladelor în cazul instrucţiunilor ***if*** şi ***if - else*** imbricate: măreşte lizibilitatea programelor şi evită interpretarile eronate.

Următoarele două secvenţe de cod sunt echivalente:

```
int test1(int a,int b)
{
      int c=-0;
      if (a > 0)
      if (b > 0)
      if (a < b)  c=2;
      else
      if (a > 10) c=3;
      else c=4;
      else c=1;
      return c;
}

int test2(int a,int b)
{
      int c=-0;
      if (a > 0){
            if (b > 0){
                  if (a < b) c=2;
                  else{
```

```
                        if (a > 10) c=3;
                        else c=4;
                    }
                }
                else c=1;
        }
        return c;
}
```

O tehnică des utilizată pentru alegerea unei singure altenative din mai multe posibile este „*else-if în cascadă*”, schemă ilustrată de exemplul urmator:

```
#include<iostream>
#include<iomanip>
using namespace std;
// Azimutul geografic al unei directii
// este unghiul dintre aceasta si directia nord
// masurat in  sens anti-trigonometric
void arataDirectia(int azm){
        (azm%=360) >= 0? azm : azm+=360;
        cout<<"Azimut = "<<setw(3)<<azm<<" grade, mergem catre ";
        if (azm==0)
                cout<<"NORD.";
        else if (0<azm && azm<90)
                cout<<"NORD-EST.";
        else if (azm==90)
                cout<<"EST.";
        else if (90<azm && azm<180)
                cout<<"SUD-EST.";
        else if (azm==180)
                cout<<"SUD.";
        else if (180<azm && azm<270)
                cout<<"SUD-VEST.";
        else if (azm == 270)
                cout<<"VEST.";
        else if (270<azm && azm<360)
                cout<<"NORD-VEST.";
        else
                cout<<"ABSURD.";
        cout<<endl;
        return;
}

int main(){
        for(int a=0;a<=360;a+=45) arataDirectia(a);
        return 0;
}
```

Urmând principiul de programare de a declara variabilele în cel mai restrâns domeniu de vizibilitate posibil (pentru a nu fi uitate sau confundate) şi cât mai aproape de momentul când li se poate asigna o valoare iniţială (pentru a nu fi folosite din eroare înaintea iniţializării) în C++ s-a permis înlocuirea expresiei condiţionale a ***if***-ului cu declararea cu iniţializare a unei variabile. Această *variabilă condiţională* are ca domeniu de vizibilitate cele două ramuri ale ***if***-ului (dacă sunt prezente amândouă) iar valoarea sa este cea testată pentru salt. De exemplu, funcţia următoare afişează pe verticală stringul primit la apel:

```
void afiseazaString(char text[]){
      for(int i=0; ; i++)
            if(char ch=text[i]) cout<<ch<<endl;
            else return;
}
```

6. Instrucţiunea switch

Sintaxa: ***switch*** **(***expresie-selector***)** *instrucţiune-corp*

Instrucţiunea de comutare ***switch*** introduce diverse alternative în fluxul de execuţie, în funcţie de valoarea *expresiei selector*. Pentru aceasta corpul instrucţiunii ***switch*** trebuie sa fie un bloc de instrucţiuni dintre care măcar una trebuie să fie etichetată cu o *etichetă* ***case*** sau ***default*** (altfel controlul sare peste ***switch***). Forma efectivă a instrucţiunii este urmatoarea:

switch **(***expresie-selector***){**
case *expresie-constanta***:**
instrucţiune
……..
instrucţiune
case *expresie-constanta***:**
instrucţiune
……..
instrucţiune

……..
case *expresie-constanta***:**
instrucţiune
……
instrucţiune
***default*:**
instrucţiune
……..
instrucţiune
}

Fiecare expresie aflată între ***case*** şi simbolul " **:**" corespunzător trebuie să fie o expresie constantă (a carei valoare este determinată la compilare) de tip întreg. Aceste expresii au rolul de etichete pentru instrucţiunile care le urmează şi în consecinţă valorile lor trebuie să fie distincte. O instrucţiune este etichetată numai dacă urmează imediat după simbolul două puncte. Atât prezenţa cât şi ordinea etichetelor ***case*** şi ***default*** este opţională, este permisă cel mult o singură eticheta ***default***. O instrucţiune poate avea mai multe etichete.

Execuţia instrucţiunii ***switch*** începe cu evaluarea *expresiei-selector*, care trebuie sa fie de tip întreg şi care de obicei este formată doar din numele unei *variabile selector*. In funcţie de valoarea gasită se execută un salt la una dintre instrucţiunile etichetate din interiorul blocului component, sau se încheie execuţia instrucţiunii de comutare. Regulile sunt urmatoarele: dacă valoarea selectorului este exact cât valoarea unei etichete ***case***, se sare la respectiva etichetă, în caz contrar se sare la eticheta ***default*** dacă aceasta este prezentă, dacă nu, execuţia instrucţiunii de comutare se încheie şi controlul este preluat de instrucţiunea care urmează în textul sursă al programului. In cazul în care a avut loc un salt la una dintre instrucţiunile blocului ***switch***, execuţia acestora continuă în modul uzual (implicit secvenţial, "în cascadă"), fără ca prezenţa etichetelor ***case*** şi ***default*** să mai influenţeze ordinea lor de execuţie.

In corpul unui ***switch*** pot să apară următoarele instrucţiuni de salt: ***break***, ***return*** şi ***goto***. Dintre acestea, ***break*** are o utilizare specifică: încheie execuţia blocului ***switch*** în care apare şi, în consecinţă, este utilizată îndeosebi pentru selectarea unei singure alternative din mai multe posibile.

In exemplul urmator:

```
void selectie(int i,int j)
{
	switch (i+j){
		cout<<'0';
	case 1:
		cout<<'1';
		break;
	case 2:
		cout<<'2';
		return;
	case 3:
	case 4:
		cout<<'#';
	default:
		cout<<'D';
		break;
	case 2*3:
		cout<<'6';
	}
	cout<<'$';
	return;
}
```

apelul `selectie(0,1)` afişează secvenţa de caractere 1$, apelul `selectie(1,1)` afişează numai cifra 2, `selectie(1,2)` si `selectie(2,2)` imprima fiecare secventa de caractere #D$, `selectie(3,3)` afiseaza 6$ iar `selectie(0,17)` afiseaza D$. Instrucţiunea `cout<<'0';` , aflată înaintea oricărei etichete ***case*** sau ***default***, este aceptată de compilator dar nu poate fi atinsă de fluxul de execuţie al programului.

Instrucţiunea ***switch*** este deosebit de utilă în situaţia în care trebuie selectată o anumită acţiune pe baza unei codificari numerice stabilite de la bun început. Punctul slab îl constituie faptul că selecţia se efecuează numai după un singur criteriu: valoarea numerică a expresiei selector. Mai mult, cazurile etichetate cu ***case*** sunt de tip discret, valoarea selectorului trebuie să fie exact cât valoarea unei etichete, numai cazul ***default***

permitând o singură alegere pentru toate valorile selectorului aflate într-un anumit domeniu. Totuşi, orice selecţie multi-criterială, chiar şi cu cazuri de tip "domeniu de valori", poate fi transformată într-o selecţie de tip ***switch***, utilizând tehnica ilustrată de urmatoarea versiune a funcţiei ***arataDirectia*** din secţiunea precedentă:

```
void arataDirectia2(int azm){
      (azm%=360) >= 0? azm : azm+=360;
      cout<<"Azimut = "<<setw(3)<<azm<<" grade, mergem catre ";
      int nord =(270<azm && azm<360)||(0<=azm && azm<90);
      int est =(0<azm && azm<180);
      int sud =(90<azm && azm<270);
      int vest =(180<azm && azm<360);

      switch (1000*nord+100*est+10*sud+vest){
      case 1000:
            cout<<"NORD.";
            break;
      case 1100:
            cout<<"NORD-";
      case 100:
            cout<<"EST.";
            break;
      case 1001:
            cout<<"NORD-";
      case 1:
            cout<<"VEST.";
            break;
      case 10:
            cout<<"SUD.";
            break;
      case 110:
            cout<<"SUD-EST.";
            break;
      case 11:
            cout<<"SUD-VEST.";
            break;
      default:
            cout<<"ABSURD!";
      }
      cout<<endl;
}
```

Expresia-selector prezentată mai sus foloseşte scrierea poziţională a numerelor, baza zece fiind folosită aici numai din motive de comoditate. In mod uzual sunt folosite coduri binare şi operaţii pe biţi, rezultând astfel selecţii cu **switch** foarte eficiente.

Echivalentul binar al expresiei selector de mai sus este următorul:

```
switch (nord<<3 | est<<2 | sud<<1 | vest){...}
```

Analog ***if***-ului, şi în cazul ***switch***-ului este permisă înlocuirea expresiei selector cu declararea şi iniţializarea unei variabile, o adevărată *variabilă selector*, vizibilă numai în corpul ***switch***-ului şi a cărei valoare determină salturile la etichete.

In programul următor avem un exemplu de funcţie care interpretează stringurile de forma "123+56-89-3+2341" ca sume de întregi şi le evaluează numeric. Stringul este

parcurs de la dreapta la stânga, calculând mereu valoarea ultimului termen citit. Când este citit un semn, valoarea formată este adunată sau scăzută din sumă.

```
#include<iostream>
using namespace std;
const int inTERMEN = 0;     //suntem intr-un termen al sumei
const int inSEMN = 1;       //suntem in + sau in -
const int inAFARA = 2;      //suntem in afara stringului
const int inEROARE = 3;     //eroare de sintaxa

int evalueaza(char tab[]){
    int i;
    //cautam sfarsitul stringului
    for (i = 0; tab[i] != '\0'; i++){
    }
    //parcurgem stringul de la dreapta la stanga
    int stare = inSEMN, suma = 0, term = 0, ordin = 1;
    for (i--; stare == inSEMN || stare == inTERMEN; i--){
        switch (char ch = (i < 0 ? '\0' : tab[i])){
        case '0':case '1':case '2':case '3':case '4':
        case '5':case '6':case '7':case '8':case '9':
            term += (ch - '0')*ordin;
            ordin *= 10;
            stare = inTERMEN;
            break;
        case '+': case '-':
            if (stare == inSEMN) stare = inEROARE;
            else{
                suma += (ch == '+' ? term : -term);
                term = 0;
                ordin = 1;
                stare = inSEMN;
            }
            break;
        case '\0':
            suma += term;
            stare = inAFARA;
            break;
        default:
            stare = inEROARE;
        }
    }
    return stare != inEROARE ? suma : -666666;
}
int main(){
    cout << evalueaza("1234+4321-60+10") << endl;
    return 0;
}
```