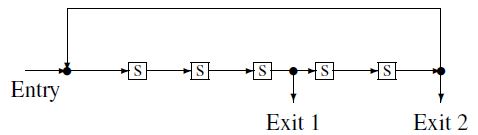
თემა 5. პროგრამის მართვის ინსტრუქციები

* განმეორების შეტყობინებები
* ციკლი წინაპირობით, შემდგომი პირობით, for -ციკლი
* ციკლის წყვეტის ოპერატორები

**განმეორების შეტყობინებები (ციკლები – მარყუჟები)**

განმეორების შეტყობინების ზოგადი სტრუქტურა შემდეგია:



ციკლს აქვს ერთადერთი შესვლის წერტილი, შესრულებადი შეტყობინებების თანმიმდევრობა და ციკლიდან გამოსვლის წერთილი. გამოსვლის წერტილი დამოკიდებულია ციკლის პირობაზე, რომელიც განსაზღვრავს, უნდა გაგრძელდეს ციკლის შესრულება, თუ ციკლი უნდა შეწყდეს.

იმპერატიული დაპროგრამების ენებში არსებობს სამი ტიპის ციკლი – ციკლი წინაპირობით, მისი კერძო შემთხვევა – for ციკლი და ციკლი შემდგომი პირობით.

**ციკლი წინაპირობით**

მისი შესრულების სქემა შემდეგია:

1. მოწმდება პირობა – გამოსახულება;
2. თუ პირობა მცდარია, ციკლის შესრულება წყდება და მართვა გადაეცემა პროგრამის შემდეგ ინსტრუქციას. პირობის ჭეშმარიტობის შემთხვევაში, სრულდება ციკლის შეტყობინებები (ციკლის ტანი).
3. პროცესი მეორდება პუნქტი 1–დან.

ე.ი. ციკლი წინაპირობით სრულდება მანამდე, სანამ პირობა ჭეშმარიტია. ამასთან, ციკლი შეიძლება საერთოდ არ შესრულდეს იმ შემთხვევაში, თუ პირობა თავიდანვე მცდარია.

მაგალითად, C/C++ -ში ასეთი ციკლის ზოგადი სახეა:

while (expression) statement;

სადაც expression არის პირობა – საზოგადოდ, შედარების, ლოგიკური ან არითმეტიკული გამოსახულე­ბა;

statement შეიძლება იყოს მარტივი – ერთი შეტყობინება, შედგენილი – ფიგურულ ფრჩხილებში ჩასმული რამდენიმე შეტყობინება, ან ცარიელი – შედგება **;** -საგან.

შემდეგი კოდი ითვლის  ():

int n, i; long long int fact = 1;

printf("Shemoitenet n: ");

scanf("%d", &n);

i = 1; // i = 2;

while (i <= n)

Shemoitenet n: 13

13 ! = 6227020800

Press any key to continue . . .

{

fact \*= i;

i++;

}

printf("%d ! = %lld\n", n, fact);

Pascal -ში while-ციკლის ზოგადი სახეა:

while B do statement;

სადაც B – ლოგიკური გამოსახულებაა.

Pascal -ის კოდი

**Program** Factorial(input,output);

**var** i,n:integer; fact:real;

**begin**

fact := 1; i := 1;

read(n);

**while** i <= n **do begin**

fact := fact \* i;

i := i + 1;

**end**;

writeln('factorial = ', fact);

**end**.

ასრულებს იგივე დავალებას. შედეგია: factorial = 6.2270208000000000E+009

**ციკლი შემდგომო პირობით**

სრულდება ერთხელ მაიც. მასში პირობა მოწმდება ციკლის ტანის ბოლოს. მაგალითად, C/C++ -ში ასეთი ციკლის ზოგადი სახეა:

do{

ციკლის ტანი: შესრულებადი შეტყობინებების თანმიმდევრობა

ან ერთი შეტყობინებება

}

while (<პირობა>);

C -ში –ის გამოთვლის კოდი შემდეგია

**int** n, i; **long** **long** fact = 1;

printf("Shemoitenet n: ");

scanf("%d", &n);

i = 1;

**do**{

fact \*= i;

i++;

}

**while** (i <= n);

printf("%d ! = %lld\n", n, fact);

ხოლო Pascal -ზე იგი ჩაიწერება ასე:

fact := 1; i : = 1;

read(n);

**repeat**

fact := fact \* i;

i : = i + 1;

**until** i > n;

პასკალში განმეორება მთავრდება, როდესაც ციკლის დასრულების პირობა ღებულიბს მნიშვნელობას true, C -ს do–while ციკლისაგან განსხვავებით, სადაც დასრულების პირობა არის მნიშვნელობა false.

რისთვისაა შემოღებული ორი თითქმის ერთნაირი ლოგიკის მქონე ციკლი – წინაპირობით და შემდგომი პირობით? ამ კითხვას პასუხობს შემდეგი მარტივი მაგალითი.

ვთქვათ, ვითვლით კლავიატურიდან შემოტანილი არაუარყოფითი რიცხვის ციფრთა ნამრავლს:

**unsigned** n, prod = 1;

printf("Shemoitanet mteli arauaryofiti ricxvi: ");

scanf("%u",&n);

**while**(n != 0) *// while ( n )*

{

prod \*= n%10;

n /= 10;

}

printf ("ricxvis cifrta namravli = %u\n",prod);

კოდი მუშაობს გამართულად ყველა მთელი ნატურალური რიცხვისთვის, მაგრამ n = 0 -ის შემთხვევაში გვაძლევს არასწორ პასუხს:

Shemoitanet mteli arauaryofiti ricxvi: 0

ricxvis cifrta namravli = 1

Press any key to continue . . .

არასწორი შედეგი უკავშირდება while -ის გამოყენებას. მართლაც, თუ n -ის შემოტანილი მნიშვნელობა ნულია, n != 0 ციკლის პირობა მცდარია, ციკლი არ შესრულდება და დაიბეჭდება prod ცვლადის მნიშვნელობა 1.

თუ შევცვლით while ციკლს do-while -ით, სიტუაცია გამოსწორდება. do-while ციკლში პირველი შესვლა ხდება უპირობოდ, ციკლის ტანი ერთხელ მაინც შესრულდება, რაც საკმარისია n = 0 -ის შემთხვევაში სწორი პასუხის მისაღებად:

**unsigned** n, prod = 1;

printf("Shemoitanet mteli arauaryofiti ricxvi: ");

scanf("%u",&n);

**do**{

prod \*= n%10;

n /= 10;

}

**while**(n != 0); *// while ( n )*

printf ("ricxvis cifrta namravli = %u\n",prod);

Shemoitanet mteli arauaryofiti ricxvi: 0

ricxvis cifrta namravli = 0

Press any key to continue . . .

***იმპლემენტაცია***

while ციკლის

while (expression)

statement;

ფსევდო ასემბლერზე ჩაწერილი იმპლემენტაცია შემდეგია:

L1: compute R1,expr გამოსახულების გამოთვლა, მისი მნიშვნელობა R1 –შია

jump\_zero R1,L2 თუ R1–ში ნულია (false), ციკლიდან გამოსვლა

statement წინააღმდეგ შემთხვევაში ციკლის ტანის შესრულება

jump L1 მართვის გადაცემა პირობის შემოწმებაზე

L2:

მივაქციოთ ყურადღება, რომ რეალიზება შეიცავს გადასვლის ორ ბრძანებას. ამავე დროს, თუ ციკლიდან გამოსვლა ხდება ციკლის ბოლოს, საჭიროა მხოლოდ ერთი გადასვლის ინსტრუქცია, როგორც do-while ციკლის იმპლემენტაციაში.

do {

statement;

} while (expression);

კომპილირდება ასემბლერის კოდში

L1: statement

compute R1,expr

jump\_nz R1,L1 არანული არის true

L2:

**for ციკლი**

C -ს შტოს ენებში for ციკლის ფორმატია:

for (გამოსახულება\_1; გამოსახულება\_2; გამოსახულება\_3) ციკლის\_ტანი;

სადაც

გამოსახულება\_1 -ში, როგორც წესი, ხდება ციკლის ცვლადებისთვის საწყისი მნიშვნელობების მინიჭება (ინიციალიზება).

გამოსახულება\_2 -ში მიოცემა პირობა, რომელიც განსაზღვრავს, უნდა შესრულდეს ციკლის ტანი, თუ არა.

გამოსახულება\_3 განმარტავს ციკლის ცვლადების ცვლილებას ციკლის ტანის ყოველი შესრულების შემდეგ.

for -ის შესრულების სქემა შემდეგია:

1. გამოითვლება გამოსახულება\_1;
2. გამოითვლება გამოსახულება\_2, რომელიც შეიძლება იყოს როგორც ლოგიკული, ისე არითმეტიკული ტიპის;
3. თუ გამოსახულება\_2-ის მნიშვნელობა ნულისაგან განსხვავებულია (ჭეშმარიტია), შესრულდება ციკლის ტანი, შემდეგ გამოითვლება გამოსახულება\_3 და ვუბრუნდებით მე-2 პუნქტს; თუ კი გამოსახულება\_2 უდრის 0-ს (მცდარია), ციკლი დასრულდება და მართვა გადაეცემა for -ის მომდევნო ინსტრუქციას.

ციკლის დასაწყისში ყოველთვის მოწმდება ციკლის პირობა. ამიტომაც, თუ პირობა თავიდანვე მცდარია, for ციკლი შეიძლება არც შესრულდეს.

თუ წინასწარ ცნობილია ციკლის იტერაციების რიცხვი, გამოგვდის ე.წ. for ციკლი მთვლელით (პარამეტრით). მთვლელის მნიშვნელობა მოიცემა ან მუდმივის სახით, ან გამოითვლება ციკლის დაწყებამდე. მაგალითად,

int i; /\* ციკლის მთვლელი \*/

int low, high; /\* მთვლელის საზღვრები \*/

i = low; /\* მთვლელის ინიციალიზება \*/

while (i <= high) /\* ციკლიდან გამოსვლის პირობის გამოთვლა \*/

{

statement;

i++; /\* მთვლელის გაზრდა \*/

};

ან for ციკლით:

int i;

int low, high;

for (i = low; i <= high; ++i)

statement;

ენა Ada -ზე თითქმის იგივეა

Low, High: Integer;

for I in Low .. High loop

statement;

end loop;

განსხვავება ისაა, რომ მთვლელის ცხადი ინიციალიზება არ ხდება: Ada -ში, ისევე როგორც Pascal -ში, ციკლის ცვლადი მკაცრად იზრდება ბიჯით 1 ან მცირდება ბიჯით –1.

Pascal -ის შემდეგი ფრაგმენტები for ციკლის გამოყენებით ითვლიან  (– ნატურალური რიცხვია):

fact := 1; read(n);

for i := 2 to n do fact := fact \* i;

ან

fact := 1; read(n);

for i := n downto 2 do fact := fact \* i;

ზოგ ადრეულ ენაში პროგრამისტს შეეძლო ციკლის პარამეტრის ცვლილების ბიჯის მითითება. მაგალითად, Algol –ენაზე კენტ რიცხვებთან სამუშაოდ შეგვიძლია დავწეროთ

for I := 1 to N step 2 do ...

სადაც I ციკლის პარამეტრია და ამავდროულად კენტი რიცხვის მნიშვნელობაც არის.

Ada -ზე კი აუცილებელია ცხადად დავაპროგრამოთ კენტი Odd რიცხვების გამოთვლა

for I in 1 .. (N + 1)/2 loop

Odd = 2 \* I - 1;

…

end loop;

სამაგიეროდ, Ada -ში დაშვებულია ციკლის საზღვრებისთვის გამოსახულებების გამოყენება, რომლებიც გამოითვლება მხოლოდ ერთხელ ციკლში შესვლისას და აღარ იცვლება. მაგალითად,

for I in expression\_1 .. expression\_2 loop

statement;

end loop;

შემდეგი ფრაგმენტის ეკვივალენტურია

I = expression\_1;

Temp = expression\_2;

while (I < Temp) loop

statement;

I := I + 1;

end loop;

C -ში

for (expression\_1; expression\_2; expression\_3) statement;

შემდეგი სახის while–ციკლისეკვივალენტურია

expression\_1;

while (expression\_2) {

statement;

expression\_3;

}

ამასთან, C -ს for -ის სამივე გამოსახულება შეიძლება იყოს საკმაოდ რთული. ვთქვათ, უნდა დავბეჭდოთ ფიბონაჩის მიმდევრობის წევრები დიაპაზონიდან [1; 100]. შესაბამისი კოდი ჩაიწერება ერთი for შეტყობინებით

**int** f0 = 1, f1 = 1, f2;

printf("%d %d ", f0, f1);

**for**(f2 = f0 + f1; f2 <= 100; f0 = f1, f1 = f2, f2 = f0 + f1)

printf("%d ", f2);

ან გამოვიყენოთ for ციკლი ტანის გარეშე, უფრო სწორად, ციკლის ტანში ჩავწეროთ ცარიელი ; ოპერატორი:

**int** f0 = 1, f1 = 1, f2;

printf("%d %d ", f0, f1);

**for**(f2 = f0 + f1; f2 <= 100; printf("%d ", f2), f0 = f1, f1 = f2, f2 = f0 + f1);

ორივე შემთხვევაში დაიბეჭდება

1 1 2 3 5 8 13 21 34 55 89 Press any key to continue . . .

***იმპლემენტაცია***

Ada -ს ციკლის

for I in expression\_1 .. expression\_2 loop

statement;

end loop;

იმლემენტაცია ჩვენს ფსევდო ასემბლერზე:

compute R1,expr\_1

store R1,I ციკლის ქვედა ინდექსი შევინახოთ I–ში

compute R2,expr\_2

store R2,High ციკლის ზედა ინდექსი შევინახოთ High –ში

L1: load R1,I R1–ში ჩავტვირთოთ I

load R2,High R2–ში ჩავტვირთოთ High

jump\_gt R1,R2,L2 თუ მეტია დავასრულოთ ციკლი

statement შევასრულოთ ციკლის ტანი

load R1,I გავზარდოთ I ერთით

incr R1

store R1,I

jump L1

L2:

ოპტიმიზებას მივაღწევთ, თუ ერთ–ერთ რეგისტრს მივამაგრებთ ცვლადს I და, თუ ეს შესაძლებელია, სხვა რეგისტრს – ცვლადს High

compute R1,ехрг\_1 R1–ში ციკლის ქვედა ზღვარია

compute R2,expr\_2 R1–ში ციკლის ზედა ზღვარია

L1: jump\_gt R1,R2,L2 თუ მეტია დავასრულოთ ციკლი

statement

incr R1 R1–ის ინკრემენტი

jump L1

L2:

ახლა განიხილეთ მარტივი C -ს ციკლი

for (i = expression\_1; expression\_2; i++)

statement;

რომლის იმპლემენტაცია იქნება

compute R1,expr\_1

store R1,i ციკლის ქვედა ინდექსი ჩავწეროთ i –ში

L1: compute R2,expr\_2 ციკლის ზედა ინდექსი გამოითვლება ციკლის ტანში

jump\_gt R1,R2,L2 თუ მეტია დავასრულოთ ციკლი

statement ციკლის ტანი

load R1,i i –ს ინკრემენტი

incr R1

store R1,i

jump L1

L2:

**ციკლის წყვეტის ოპერატორები**

სტრუქტურული პროგრამირების მიდგომა მოითხოვს, რომ ციკლიდან გამოსვლა ხდებოდეს მხოლოდ ციკლის დასაწყისში ან ციკლის ბოლოს. ეს საშუალებას იძლევა გაცილებით უფრო ადვილად გავაანალიზოთ და შევამოწმოთ პროგრამა. მაგრამ პრაქტიკაში, ხშირად საჭირო ხდება გამოსვლა ციკლის ტანიდან, განსაკუთრებით მაშინ, როდესაც გამოვლინდა შეცდომა.

დაპროგრამების ენაში ციკლის ტანის წყვეტას უზრუნველყოფს სპეციალური ოპერატორები: ასე C/C++, Java, C# - ში ესენია break; და continue;

Fortran - ში – exit და cycle

Pascal- ში – პროცედურები break, continue და ოპერატორი goto

Ada - ში – ოპერატორები exit, exit L, exit when (condition),

exit L when (condition)

მაგალითად, Pascal - ის კოდი

**var** num: real; i: integer;

**begin**

**for** i := 1 **to** 5 **do** **begin**

write ('Enter the number: ');

readln (num);

**if** num = 0 **then**

break;

writeln (num)

**end**;

დაბეჭდავს კლავიატურიდან შემოტანილ 5 რიცხვს, ან შეწყვეტს ციკლს, თუ შემოვიდა 0–ი.

ხოლო

**var**

num, sum: real;

i: integer;

**begin**

sum := 0;

**for** i := 1 **to** 5 **do** **begin**

write ('Enter the number: ');

readln (num);

**if** num <= 0 **then**

continue;

sum := sum + num

**end**;

  writeln (sum);

კლავიატურიდან შემოტანილი 5 რიცხვიდან შეკრებს მხოლოდ დადებითებს.

Ada - ს კოდი

Sum := 0;

**while** Sum <= 100 **loop**

Get(A); -- წავიკითხოთ A

**if** A = 0 **then** -- თუ А = 0, მაშინ

**exit**; -- შევწყვიტოთ ციკლი

**end** **if**;

Sum := Sum + A;

**end** **loop**;

დასრულდება ორიდან ერთ შემთხვევაში: ა) Sum ცვლადი გახდება 100–ზე მეტი; ან ბ) მომხმარებელი შემოიტანს 0 –ს.

თუკი გამოვიყენებთ წყვეტის ოპერატორს exit when (condition), გვექნება შემდეგი ფრაგმენტი

Sum := 0;

**while** Sum <= 100 **loop**

Get(A);

**exit** **when** A = 0;

Sum := Sum + A;

**end** **loop**;

**for-each ციკლი**

თანამედროვე პროგრამირების ენებში სულ უფრო დიდ გამოყენებას პოულობს ციკლების „for-each“ (თითოეულისათვის) კონცეფცია. ასეთი ციკლის დანიშნულებაა თანმიმდევრობით მოხდეს განმეორებადი მოქმედებები ობიექტების კოლექციის მიმართ.

ზოგიერთ ენაში (მაგალითად, C#) ასეთი ციკლი წარმოდგენილია საკვანძო სიტყვით foreach.

1.

var names = new List<string>() { "John", "Tom", "Peter" };

foreach (string name in names)

{

if (name == "Tom")

John

Peter

{

continue;

}

Console.WriteLine(name);

}

2.

var names = new List<string>() { "John", "Tom", "Peter" };

foreach (string name in names)

{

if (name == "Tom")

{

break;

John

OK

}

Console.WriteLine(name);

}

Console.WriteLine("OK");

Java -ში სპეციალური საკვანძი სიტყვა შემოღებული არაა. „for-each“ ციკლის ზოგადი ფორმა ასეთია:

for (<ტიპი> <იტერაციული ცვლადი> : <კოლექცია>) <ციკლის ტანი>

სადაც <იტერაციული ცვლადი> - ესაა ცვლადის სახელი, რომელიც მიმდევრობით მიიღებს მნიშვნელობებს კოლექციიდან, პირველი ელემენტიდან ბოლომდე, ხოლო <ტიპი> - იტერაციული ცვლადის ტიპია. ცხადია, <ტიპი> უნდა ემთხვევოდეს კოლექციის ელემენტების ტიპს.

1.

int sum = 0;

int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

*// use for to display and sum the values*

for(int x : nums)

{

System.out.println("Value is: " + x);

sum += x;

if(x == 5) break; *// stop the loop when 5 is obtained*

}

System.out.println("Summation of first 5 elements: " + sum);

**2.**

ArrayList<String> list=**new** ArrayList<String>();

list.add("aaaaaaa");

list.add("bbbbb");

aaaaaaa

bbbbb

cccccccc

list.add("cccccccc");

**for**(String s:list){

     System.out.println(s);

}

“for-each” სტილის ციკლი საშუალებას იძლევა მოვახდინოთ კოლექციის ელემენტებზე წვდომის პროცესის ავტომატიზაცია. მაგალითად, მასივებისატვის ასეთი ციკლის გამოყენებით აღარაა საჭირო ელემენტის ინდექსის გამოყენება, რომლისთვისაც საწყისი და საბოლოო მნიშვნელობები უნდა მიგვეთითებინა. ამის მაგივრად, პროგრამა ავტომატურად შეასრულებს ციკლს მასივის ყველა ელემენტზე და მიმდევრობით ამოიღებს თითოეული ელემენტის მნიშვნელობას, პირველიდან ბოლო წევრამდე. ასეთი ციკლის დროს მარტივდება პროგრამის ჩაწერის სინტაქსი და ასევე გამოირიცხება მასივის საზღვრებს გარეთ გასვლის შეცდომის შესაძლებლობა.

C++ -ის for–each ტიპის ციკლის - დიაპაზონიანი for-ის მაგალითები:

1.

for (auto m : "Ganivi")

cout << m << '\t';

G a n i v i

cout << endl;

2.

double sum(0.0);

for (auto a : { 2.0,10.5,-3.25,0.75 })

sum += a;

average = 2.5

cout <<"average = "<< sum / 4 << endl;

3.

#include <iostream>

#include <array>

using namespace std;

int main()

{

array<int, 10> vec;

for (int n = 1; n < 11; n++)

vec[n - 1] = (n \* n);

for (auto m : vec)

cout << m << '\t';

cout << endl;

}

შედეგია: 1 4 9 16 25 36 49 64 81 100