3. C++ kontroles konstrukcijas

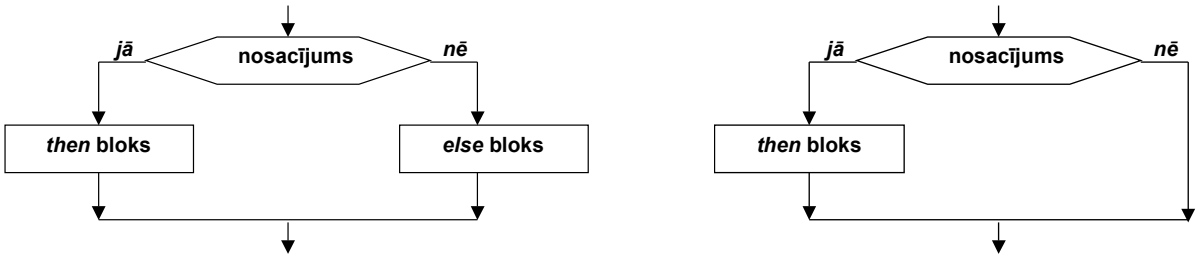
Nodaļas saturs:

3.1. Izvēles priekšraksts un loģiskas izteiksmes

Izvēle (*selection*) ir viena no 3 galvenajām vadības konstrukcijām (blakus secībai (*sequence*) un ciklam (*looping*). Izvēli valodā C++ realizē izvēles priekšraksti (*selection statements*) *if-then* un *switch-case*. Kaut kādā nozīmē ar izvēli nodarbojas arī nosacījuma operators *(?:)*, tomēr tas nenosaka izmaiņas programmas vadībā, bet tikai nodrošina noteiktas vērtības izvēli un ir uzskatāms par cita līmeņa konstrukciju.

3.1.1. Izvēles priekšraksts if-then-else

Programmas darbības gaitā var rasties nepieciešamība atkarībā no kāda nosacījuma izpildīt vai neizpildīt kādu programmas daļu. To parasti veic ar izvēles priekšrakstu *if-then-else*. *if-then-else* priekšraksta darbība aprakstīta attēlā 3.1 parādītajā blokshēmā, bet sintakse – sintaktiskajā diagrammā 1.1.



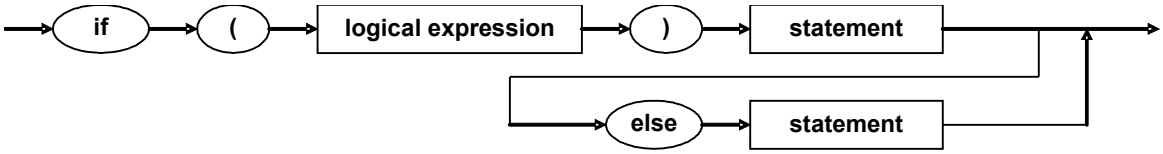
Attēls 3.1. if-then-else darbības shēma (ar un bez else zara)

Valodas C++ *if-then-else* priekšrakstā neparādās vārds *then* (to aizvieto obligātas iekavas ap izvēles nosacījumu). if-then-else priekšraksts sastāv no 3 blokiem, no kuriem trešais ir neobligāts:

- izvēles nosacījums (*selection condition*), kas ir loģiska izteiksme (par loģiskām izteiksmēm skatīt zemāk), visbiežāk salīdzināšanas operācija,
- darbība, kas tiek veikta, ja nosacījums izpildās (*then* bloks),
- (neobligāti) darbība, kas tiek veikta, ja nosacījums neizpildās (*else* bloks).

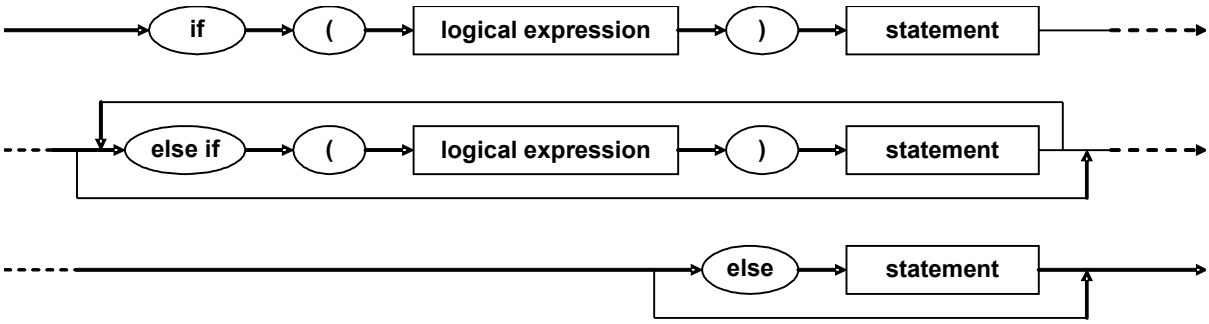
Darbību apraksta kāds priekšraksts, un ļoti bieži tas ir bloks.

Sintakse 1.1. *if-then-else statement* (if-then-else priekšraksts)



Formāli valodā C++ operatoram *if-then-else* nav speciāli izdalīta daļa *else if*. Ja programmā parādās *else if*, tad attiecīgais *if* tiek uzskatīts par augšējā *if-then-else* priekšraksta *else* zara sastāvdaļu. Tomēr praksē ir daudz vieglāk to uztvert kā kaskādes veida konstrukciju, nevis kā vairāku līmeņu konstrukciju (sintakse 1.2). Ievērojiet, ka sintaktiskās diagrammas 1.1 un 1.2 apraksta vienu un to pašu konstrukciju, tikai divos dažādos veidos!

Sintakse 1.2. *if-then-else statement 2* (if-then-else priekšraksts 2)

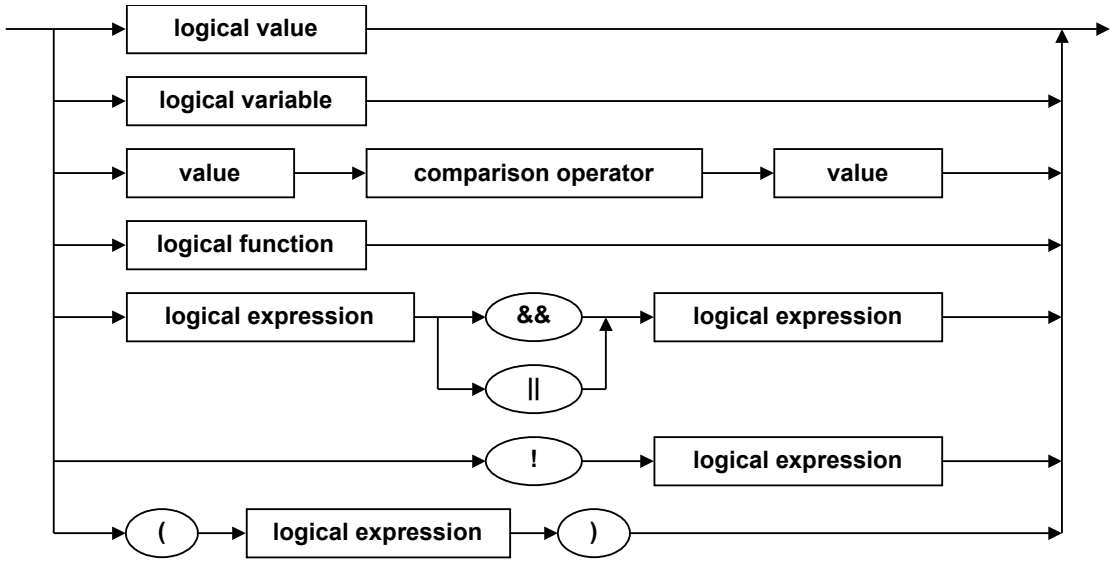


3.1.2. Loģiskas izteiksmes

Loģiska izteiksme (*logical expression*) ir konstrukcija, kas sastāv no loģiskām operācijām un loģiskām vērtībām, un kuru pēc šo operāciju izpildes reprezentē loģiska vērtība.

Vienkāršas loģiskas izteiksmes tipisks piemērs ir salīdzināšanas operācija, tomēr vispārīgāka loģisku izteiksmju sintakse parādītā diagrammā 1.3. Loģiskas izteiksmes ir priekšraksta *if-then-else*, kā arī cikla priekšrakstu sastāvdaļa.

Sintakse 1.3. *logical expression* (loģiska izteiksme)



Sintakse 1.4. *comparison operator* (salīdzināšanas operācija)

<comparison operator> ::= < | <= | > | >= | == | !=

Sintakse 1.5. *logical value* (loģiska vērtība)

<logical value> ::= true | false

Pirmkods 1.1. if-then-else un loģiskas izteiksmes (*ctr1if.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 int main ()
05 {
06     int a;
07     bool b;
08     cout << "Input integer value: ";
09     cin >> a;
10     if (a > 0) cout << a << " is positive" << endl;
11     else cout << a << " is not positive" << endl;
12     b = a >= 1 && a <=10;
13     if (b) cout << a << " is between 1 and 10" << endl;
14     else cout << a << " is not between 1 and 10" << endl;
15     return 0;
16 }
```

Programmas darbības piemērs #1:

```
Input integer value: 5
5 is positive
5 is between 1 and 10
```

Programmas darbības piemērs #2:

```
Input integer value: -5
-5 is not positive
-5 is not between 1 and 10
```

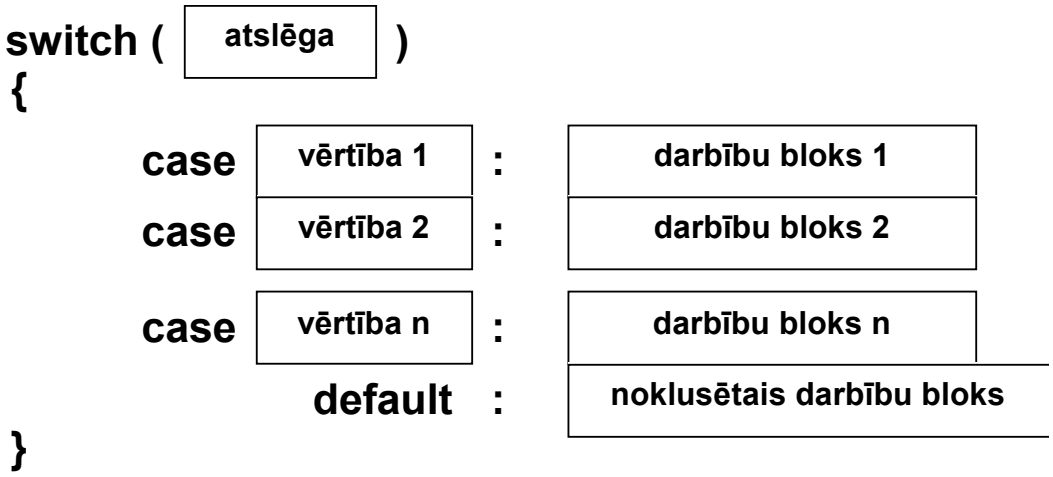
3.1.3. Aritmētisku un loģisku izteiksmju saistība

Valodā C++ pēc būtības nav atsevišķa loģiska tipa un loģisku vērtību. Tips bool un vērtības true, false ieviestas tikai jaunākajā C++ standartā. Pēc būtības tips *bool* ir tas pats *int* viena baita garumā, *true* ir 1, bet *false* – 0, un ne tikai pēc būtības atbilst, bet ir līdzvērtīgi lietojami. Loģiskās operācijas darbojas vēl plašāk – 0 apzīmē *false*, bet jebkurā cita vesela skaitļa vērtība (piemēram, 1, -2, 3) – *true*. Ņemot vērā to, ka loģiskas vērtības ir skaitliskas vērtības tad var uzskatīt, ka jebkura loģiska izteiksme ir arī aritmētiska izteiksme, bet jebkura aritmētiska izteiksme – arī loģiska, turklāt drīkst brīvi būvēt jauktu aritmētiski-loģisku izteiksmi.

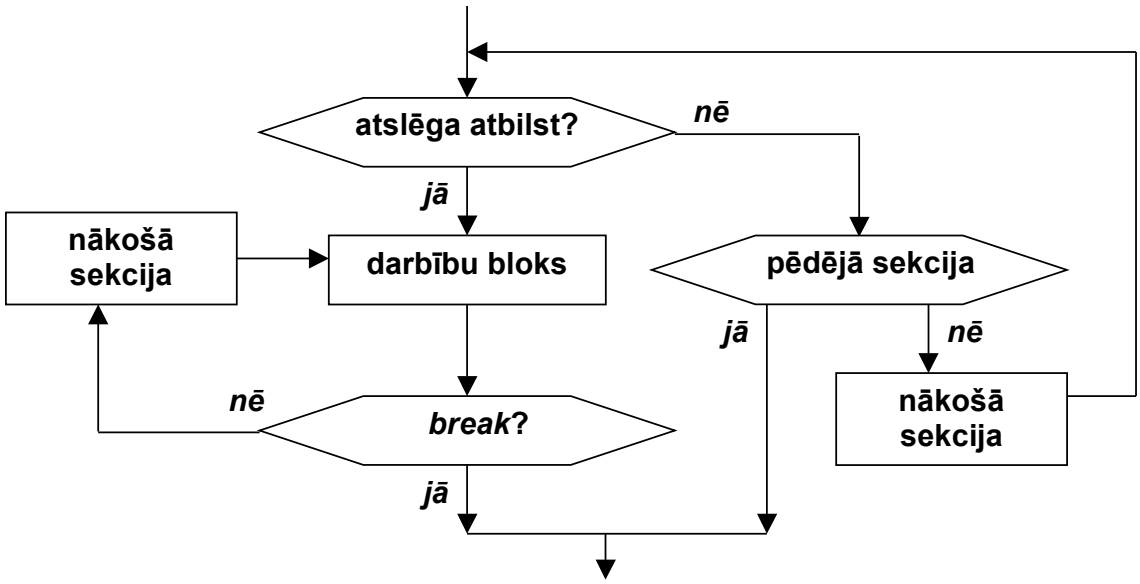
3.1.4. Izvēles priekšraksts switch-case

Ja izvēle notiek starp diskrētām vērtībām, kas nav īpaši lielā skaitā, tad ērtāk un uzskatamāk būtu lietot priekšrakstu *switch-case*. *switch-case* darbības princips ir lielā mērā līdzīgs *if-then-else* darbībai ar vienu būtisku izņēmumu – pēc atbilstošās darbības izpildes pie dotās izvēles programmas vadība automātiski **netiek** nodota uz priekšraksta beigām, bet gan nonāk nākošās izvēles apstrādes blokā, ja vien netiek lietots operators *break*. Tas nodrošina iespēju vairākām izvēles iespējām rakstīt vienu vienīgu apstrādes kodu. Šis izņēmums padara *switch-case*

priekšraksta darbības shēmu nedaudz neordināru un ne tik viegli saprotamu (attēls 3.2), tāpēc vieglāk saprast tā darbību būtu pēc piemēra (pirmkods 1.2).



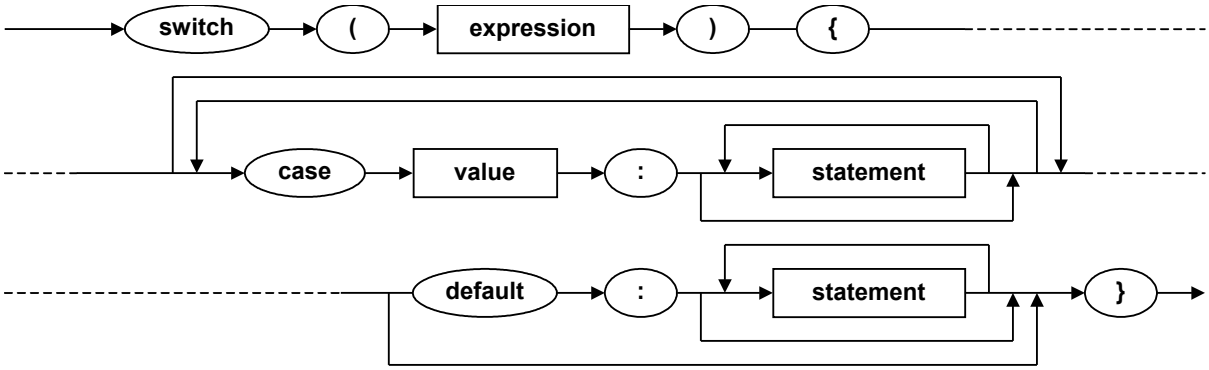
Attēls 3.2. switch-case loģiskā struktūra



Attēls 3.3. switch-case darbības shēma

Ja atslēga ir sakritusi ar kādu no uzskaitītajām vērtībām, nekāda turpmāka salīdzināšana vairs nenotiek, bet tiek izpildītas visas darbības pēc kārtas, līdz operatoram *break* vai līdz konstrukcijas beigām. Ja atslēga nav sakritusi ar nevienu no vērtībām, tiek izpildīts *default* bloks, bet jā tāda nav, neizpildās nekas.

Sintakse 1.6. ***switch-case statement*** **(switch-case priekšraksts)**



Sintakse 1.7. ***switch-case statement*** **(switch-case priekšraksts) (alternatīva BNF formā)**

<switch-case statement> ::= switch { (case <value>: (<statement>)*)* [default: (<statement>)*] }

Pirmkods 1.2. switch-case priekšraksts (*ctr2switch.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 int main ()
05 {
06     int a;
07     cout << "Input integer value: ";
08     cin >> a;
09     switch (a)
10     {
11         case 1:
12         case 2:
13             cout << "a is 1 or 2" << endl;
14             break;
15         case 3:
16             cout << "a is 3" << endl;
17         case 4:
18             cout << "a is 4" << endl;
19             break;
20         default:
21             cout << "a is bad" << endl;
22     };
23     return 0;
24 }
```

Programmas darbības piemērs #1:

```
Input integer value: 2
a is 1 or 2
```

Programmas darbības piemērs #2:

```
Input integer value: 3
a is 3
a is 4
```

Programmas darbības piemērs #3:

```
Input integer value: 4
```

```
a is 4
```

Programmas darbības piemērs #4:

```
Input integer value: 5
a is bad
```

Komentāri pie pirmkoda piemēra 1.2.

- *break* neizsaukšana aiz skaitļa 1 apstrādes (rinda 11) nodrošina to, ka programma gan uz skaitli 1, gan uz skaitli 2 izpilda rindas 13-14 (programmas darbības piemērs #1).
- *break* neizsaukšana aiz skaitļa 3 apstrādes (rinda 16) nodrošina to, ka programma uz skaitli 3 izpilda gan rindu 16, gan rindas 18-19 (programmas darbības piemērs #2).
- *default* zars izpildās, ja nav fiksēta atslēgas sakritība ar nevienu no definētajām vērtībām (rindas 20-21, programmas darbības piemērs #4).

3.1.5. Nosacījuma funktors ?:

Nosacījuma funktors darbojas pēc līdzīga principa kā *if-then-else* priekšraksts ar atšķirību, ka veicamo darbību (priekšrakstu) vietā ir izteiksmes, kas atgriež vērtību, tādējādi arī pats funktors atgriež noteiktu vērtību – to, kuru izrēķina attiecīgā izteiksme.

Sintakse 1.8. *conditional functor* (nosacījuma funktors)

Pirmkods 1.3. Nosacījuma funktors ?: (*ctr3condition.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 int main ()
05 {
06     int a;
07     cout << "Input integer value: ";
08     cin >> a;
09     cout << (a>0?"positive":"not positive") << endl;
10     cout << (a>0?"positive":(a==0?"null":"negative")) << endl;
11     return 0;
12 }
```

Programmas darbības piemērs #1:

```
Input integer value: 5
positive
positive
```

Programmas darbības piemērs #2:

```
Input integer value: -5
not positive
negative
```

3.2. Cikla konstrukcijas

Cikls (*looping*) ir vadības konstrukcija, kas atkārto noteiktas darbības izpildi līdz brīdim, kad pārstāj izpildīties noteikts nosacījums.

Programmēšanā mēdz izšķirt divu veidu ciklus:

- cikls ar skaitītāju,
- cikls ar nosacījumu (*conditional looping*).

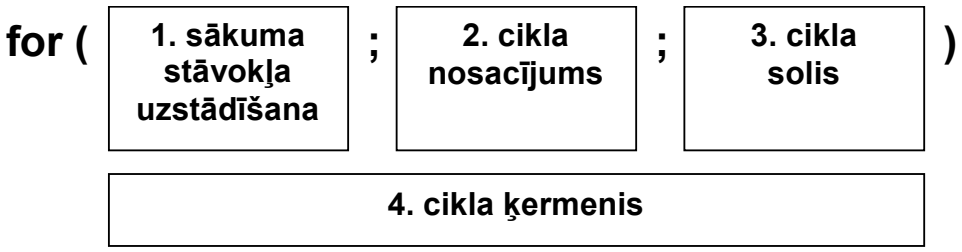
Ciklā ar skaitītāju pirms cikla izpildes jau ir zināms atkārtošanas reižu skaits, un tas uzskatāms par cikla ar nosacījumu speciālgadījumu.

Tomēr C++ šis dalījums nav tik izteikts, jo visas 3 pieejamās cikla konstrukcijas ir izmantojamas ciklu ar nosacījumu veidošanai, bet operators *for* ir ērtāks cikla ar skaitītāju konstruēšanai.

Divas obligātas jebkura cikla priekšraksta komponentes (atbilstoši cikla definīcijai) ir **cikla nosacījums** un **cikla ķermenis**. Viena cikla ķermeņa izpildīšanu vienu reizi sauc par **iterāciju**. Vispārīgā gadījumā cikla izpildē var būt neviena, viena vai vairākas iterācijas.

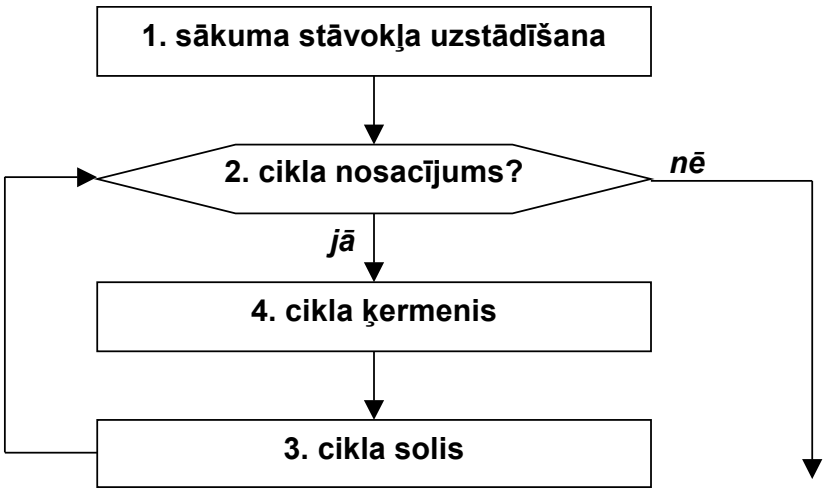
3.2.1. Cikls for

Lielā daļā programmēšanas valodu cikls *for* ir cikls ar skaitītāju, tomēr valodā C++ tas ir tikpat universāls kā abi pārējie cikli.



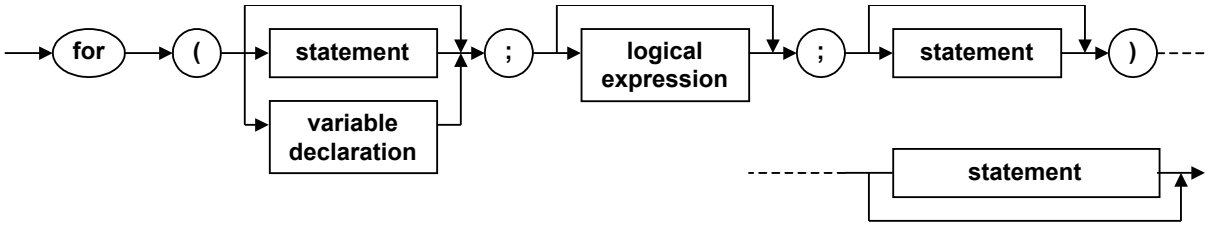
Attēls 3.4. for cikla loģiskā struktūra

Visas parējās cikla komponentes, kas nav cikla ķermenis, kopā mēdz saukt par cikla galvu.



Attēls 3.5. for cikla darbības shēma

Sintakse 1.9. *for looping* (for cikls)



Abi sekojošie piemēri (1.4 un 1.5) apraksta programmas, kas izrēķina visu veselu skaitļi 1..a summu.

Pirmkods 1.4. for cikls #1 (*ctr4for.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 int main ()
05 {
06     int a, s=0;
07     cout << "Input integer value: ";
08     cin >> a;
09     for (int i=1; i<=a; i++) s+=i;
10     cout << s << endl;
11     return 0;
12 }
```

Programmas darbības piemērs:

```
Input integer value: 9
45
```

Komentāri pie pirmkoda piemēra 1.4.

- Viss *for* cikls aizņem tikai rindiņu 9.
- Sākuma stāvokļa uzstādīšana var būt gan piešķiršanas (vai cits) priekšraksts, gan arī mainīgā deklarācija, kā šajā piemērā, tomēr, ja mainīgais tiek deklarēts cikla galvā (resp., sākuma nosacījuma uzstādīšanas blokā), tad tā redzamības apgabals ir tikai šis cikls un pēc šī cikla mainīgais vairs nav izmantojams (senākās C++ versijās šādi deklarēts mainīgais bija izmantojams arī aiz cikla).

Otrais piemērs parāda divas citas iespējas, kā pierakstīt šo pašu algoritmu izmantojot *for* ciklu (attiecīgi rinda 9 un rindas 11-15).

Pirmkods 1.5. for cikls #2 (*ctr5for.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 int main ()
05 {
06     int i, a, s;
07     cout << "Input integer value: ";
08     cin >> a;
09     for (i=1,s=0; i<=a; s+=i,i++);
10     cout << s << endl;
11     for (i=1,s=0; i<=a; )
12     {
```

```
13          s+=i;
14          i++;
15      };
16      cout << s << endl;
17      return 0;
18  }
```

Programmas darbības piemērs:

```
Input integer value: 9
45
45
```

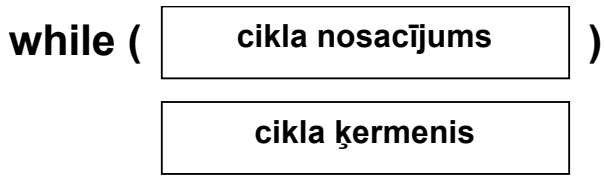
Komentāri pie pirmkoda piemēra 1.5.

- 9. rindā redzams, ka gan sākuma nosacījuma uzstādīšanā, gan solī var ievietot vairākas darbības, atdalot tās ar komatiem (sk. komata operatoru zemāk) – vienīgi jāievēro darbību izpildes secība (sk. attēlu 3.5).
- *for* priekšraksts pieļauj jebkuras no 4 komponentēm (sk. attēlu 3.4) izlaišanu – 9. rindā attēlotājā priekšrakstā izlaists cikla ķermenis (tā lomu pilda solis), bet 11 rindā attēlotajā cikla galvā ir izlaists solis (tā lomu pilnā mērā pārņēmis cikla ķermenis). Visu, kas ievietots sākuma nosacījuma uzstādīšanas blokā, var pārnest pirms cikls, vienīgā atšķirība tādā gadījumā iespējama tikai tad, ja šajā blokā tiek deklarēts mainīgais (tiktu ietekmēts mainīgā redzamības apgabals).
- *for* priekšrakstā teorētiski drīkst izlaist arī cikla nosacījumu – tādā gadījumā tas vienmēr būs *true*, un nebūs izmantojams iziešanai no cikla, kas būs jāveic ar operatora *break* palīdzību.

Šajā sadaļā parādījās jēdziens **mainīgā redzamības apgabals**. Tas izvērsti tiks aprakstīts nodaļā par funkcijām.

3.2.2. Cikls ar priekšnosacījumu while

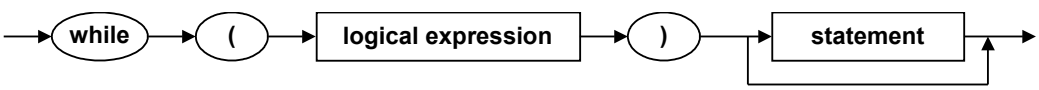
Cikls *while* ir funkcionāli identisks ciklam *for* ar divām sintaktiskām atšķirībām – sākuma stāvokļa uzstādīšana iznesta pirms cikla, bet cikla solis var tikt novietots cikla ķermeņa beigās.



Attēls 3.6. while cikla loģiskā struktūra

while priekšraksta sintakse parādīta diagrammā 1.10. Svarīgi ievērot, ka cikla nosacījumu aprakstošā loģiskā izteiksme obligāti jāliek iekavās.

Sintakse 1.10. *while looping* (while cikls)



Pirmkoda piemēri 1.6 un 1.7 apraksta programmas, kas ļauj no klaviatūras ievadīt veselus pozitīvus skaitļus un beigās izvada iepriekš ievadīto skaitļu summu. Kā ievadīšanas beigu pazīme kalpo nulles vai negatīva skaitļa ievadīšana.

Pirmkods 1.6. while cikls (*ctr6while.cpp*)

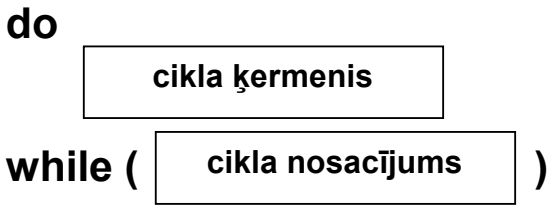
```
01 #include <iostream>
02 using namespace std;
03
04 int main ()
05 {
06     int a, s=0;
07     cout << "Input positive integer value (<=0 to end): ";
08     cin >> a;
09     while (a > 0)
10     {
11         s += a;
12         cout << "Input positive integer value (<=0 to end): ";
13         cin >> a;
14     };
15     cout << s << endl;
16     return 0;
17 }
```

Programmas darbības piemērs:

```
Input positive integer value (<=0 to end): 6
Input positive integer value (<=0 to end): 2
Input positive integer value (<=0 to end): 7
Input positive integer value (<=0 to end): 0
15
```

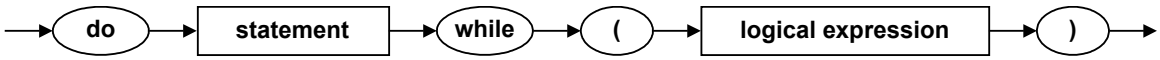
3.2.3. Cikls ar pēcnosacījumu do-while

Cikls *do-while* atšķiras no *while* ar to, ka ieeja ciklā notiek uzreiz cikla ķermenī (nevis caur cikla nosacījumu), tādējādi nodrošinot, ka cikla ķermenis tiks izpildīts vismaz vienu reizi.



Attēls 3.7. do-while loģiskā struktūra

Sintakse 1.11. *do-while looping* (do-while cikls)



Pirmkoda piemērā 1.7 aprakstītā programma ir funkcionāli identiska tai, kas aprakstīta 1.6. Ieguvums ir tas, ka, salīdzinot ar *while* ciklu, nebija 2 reizes jāatkārto skaitļa ievadīšanas kods (rindas 10-11 piemērā 1.7). Jebkura *while* cikla pārveidošana par *do-while* ir relatīvi vienkārša, tomēr ne vienmēr tik vienkārša, kā pārveidojot starp *while* un *for*, jo nosacījuma pārbaudes pārcelšana aiz cikla ķermeņa var potenciāli radīt arī blakus efektus, piemēram, pirmkoda piemērā 1.7 aprakstītā programma strādās pareizi tikai tad, ja pirms cikla mainīgais *a* būs inicializēts ar 0 (salīdzināt ar piemēru 1.6).

Pirmkods 1.7. while cikls (*ctr7dowhile.cpp*)

```
01 #include <iostream>
```

```
02 using namespace std;
03
04 int main ()
05 {
06     int a=0, s=0;
07     do
08     {
09         s += a;
10         cout << "Input positive integer value (<=0 to end): ";
11         cin >> a;
12     } while (a > 0);
13     cout << s << endl;
14     return 0;
15 }
```

Programmas darbības piemērs:

```
Input positive integer value (<=0 to end): 6
Input positive integer value (<=0 to end): 2
Input positive integer value (<=0 to end): 7
Input positive integer value (<=0 to end): 0
15
```

3.2.4. Operatori break un continue

Visām trīs C++ pieejamām cikla konstrukcijām kopīgs ir tas, ka izeja no cikla notiek vienā punktā – tajā, kur notiek cikla nosacījuma pārbaude. Šāda noteiktība nodrošina mazāku kļūdu ielaišanas iespējamību, veidojot ciklus, tomēr atsevišķos gadījumos, lai nodrošinātu īsāku un ērtāku pierakstu, būtu pieļaujams atkāpties no šī principa, izmantojot operatoru *break* (tiek izmantots arī priekšrakstā *switch-case*), bez tam dažkārt ļoti ērts izmantošanai ir arī operators *continue*.

break

Operators *break* nodrošina kārtējās cikla iterācijas pārtraukumu un izeju no cikla.

continue

Operators *continue* nodrošina kārtējās cikla iterācijas pārtraukumu un pāreju uz nākošo iterāciju, izlaižot atlikušās cikla ķermeņa daļas izpildi.

Pirmkoda piemērā 1.8 parādīta operatoru *break* un *continue* izmantošana. Tajā aprakstītā programma ir ļoti līdzīga kā tā, kas aprakstīta piemēros 1.6 un 1.7 – tiek saskaitīta pēc kārtas ievadītu skaitļu summa, tomēr ir divi papildus nosacījumi: (1) cikls beidzas ne tikai, ievadot 0 vai negatīvu skaitli, bet arī otrreiz pēc kārtas ievadot to pašu skaitli (sk. programmas darbības piemēru #2), to nodrošina operatora *break* izmantošana (rinda 12), (2) tiek saskaitīti tikai pāra skaitļi – to nodrošina operatora *continue* izmantošana (rinda 13), kas nepāra skaitļa gadījumā izlaiž pieskaitīšanas operācijas izpildi (rinda 14).

Pirmkods 1.8. break un continue izmantošana (*ctr8break.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 int main ()
05 {
06     int a, b=0, s=0;
```

```
07      do
08      {
09          cout << "Input positive integer value (<=0 to end): ";
10          b = a;
11          cin >> a;
12          if (a == b) break;
13          if (a%2 != 0) continue;
14          s += a;
15      } while (a > 0);
16      cout << s << endl;
17      return 0;
18  }
```

Programmas darbības piemērs #1:

```
Input positive integer value (<=0 to end): 2
Input positive integer value (<=0 to end): 3
Input positive integer value (<=0 to end): 4
Input positive integer value (<=0 to end): 0
6
```

Programmas darbības piemērs #2:

```
Input positive integer value (<=0 to end): 3
Input positive integer value (<=0 to end): 4
Input positive integer value (<=0 to end): 5
Input positive integer value (<=0 to end): 5
4
```

Komentāri pie pirmkoda piemēra 1.8.

- Lai tiktu nodrošināta izeja no cikla sastopot ievadā tādu pašu skaitli kā iepriekš, ir vajadzīgs papildus mainīgais iepriekš ievadītā skaitļa glabāšanai (*b*). Dotajā kontekstā ir svarīgi, lai tas būtu inicializēts ar 0 vai negatīvu skaitli.

Ikvienu *break* un *continue* operatoru var aizstāt ar citām konstrukcijām, papildus izmantojot *if* priekšrakstu un/vai papildinot cikla nosacījumu. Pirmkoda piemērā 1.9 aprakstītā programma ir funkcionāli identiska iepriekšējai, bet tajā šie abi operatori ir aizstāti (sk. komentārus pēc piemēra).

Pirmkods 1.9. break un continue apiešana (*ctr9nobreak.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 int main ()
05 {
06      int a, b=0, s=0;
07      do
08      {
09          cout << "Input positive integer value (<=0 to end): ";
10          b = a;
11          cin >> a;
12          if (a != b && a%2 == 0)
13          {
14              s += a;
15          }
16      } while (a > 0 && a != b);
17      cout << s << endl;
```

```
18      return 0;
19  }
```

Komentāri pie pirmkoda piemēra 1.9.

- Lai aizstātu *break* (rinda 12 piemērā **1.8**), tika papildus ieviesta nosacījuma pārbaude *a!=b* rindā 12, kā arī papildināts cikla nosacījums ar to pašu *a!=b* (rinda 16).
- Lai aizstātu *continue* (rinda 13 piemērā **1.8**) tika papildus ieviesta nosacījum pārbaude *a%2==0* (rinda 12).
- Kopumā ņemot, ieviestais *if* priekšraksts (rindas 12-15) kalpo gan *break*, gan *continue* aizvietošanai.