1. Ievads C++ programmēšanā

Nodaļas saturs:

1.1. Valodas C++ izcelšanās

Valoda C++ ir izcēlusies no valodas C, kuru 1970. gadā izstrādājis *AT&A Bell Laboratories* līdzstrādnieks **Deniss Ričijs** (*Dennis Ritchie*). Valodas C sākotnējais uzdevums bija operētājsistēmas UNIX veidošanai un uzturēšanai. Tomēr šī valoda kļuva ļoti populāra un to sāka izmantot lietojumprogrammu izstrādei, kā arī citās operētājsistēmās. Valoda C uzskatāma par kaut ko, kas ir pa vidu starp ļoti augsta un zema līmeņa programmēšanas valodām. Kaut arī sistēmprogrammu rakstīšanai C uzskatāms par ļoti piemērotu, tomēr lietojumprogrammatūras izstrādē valodai C piemīt vairāki trūkumi – tajā rakstītās programmas nav tik viegli uztveramas un saprotamas kā citu augsta līmeņa programmēšanas valodu programmas, turklāt tajā nav daudzu iebūvētu automātiskās pārbaudes iespēju (resp., augsta līmeņa konstrukciju) kā pierasts daudzās citās valodās.

Lai kompensētu valodas C nepilnības, 20. gs. 70. gadu beigās – 80. gadu sākumā *AT&A Bell Laboratories* līdzstrādnieks **Bjerns Stroustrups** (*Bjärne Stroustrup*) izstrādāja programmēšanas valodu C++, kas daudzos gadījumos ir izrādījusies labāka par savu priekšteci. Tajā ieviestas daudzas augsta līmeņa konstrukcijas, kas atvieglo programmētāja darbu. Bez tam C++ ir saglabājis savietojamību ar valodu C (lielākā daļa C programmu ir arī C++ programmas), kas savulaik visticamāk bija izšķiroši, lai C++ iegūtu savu plašo pielietojumu un popularitāti.

Uz valodu C++ vēl lielākā mērā attiecas tas, ko savulaik attiecināja uz valodu C – tajā ir kopā apvienotas gan zema, gan ļoti augsta līmeņa konstrukcijas, kas vēl lielākā mērā, kā tas bija valodai C, ir valodas C++ spēks un reizē arī tās vājums. Tās vājums galvenokārt izpaužas valodas C++ sarežģītībā un plašajā apjomā, kam ir divas galvenās sekas: ir salīdzinoši grūti izveidot kompilatoru (programmu, kas no pirmkoda teksta izveido izpildāmu kodu) un tā prasa augstāku profesionalitāti no programmētāja. Vairāk kā divus gadu desmitus pēc C++ izcelšanās var uzskatīt, ka pirmā problēma ir atrisināta – ir uzbūvēti pietiekoši kvalitatīvi un efektīvi kompilatori, un ērtas izstrādes vides, atliek tikai programmētāja profesionalitāte. Tāpēc, iegūstot nepieciešamās zināšanas un iemaņas, valoda C++ programmētāja rokās var kļūt par tik spēcīgu rīku, ar kuru var cerēt mēroties spēkiem tikai retā programmēšanas valoda.

1.2. Programma valodā C++

1.2.1. Failu struktūra un kompilēšana

Programma valodā C++ sastāv no viena vai vairākiem **C++ failiem** (faila paplašinājums parasti *.cpp* vai *.cc*), kā arī iespējami no citiem failiem, no kuriem tipiskākie ir t.s. **hedera** (jeb galvas) faili (faila paplašinājums *.h*). Tātad, triviālā C++ programma sastāv no viena C++ faila. Triviāla C++ programma parādīta pirmkoda piemērā 1.1.

Pirmkods 1.1. Triviāla C++ programma (*int1world.cpp*)

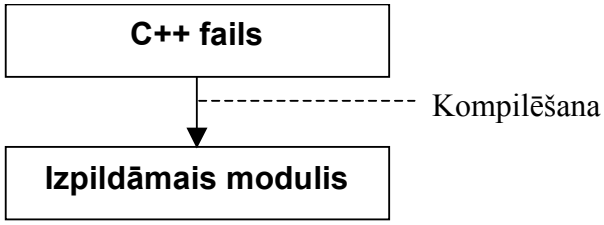
```
01 #include <iostream>
02 using namespace std;
03
04 int main ()
05 {
06     cout << "Hello, world!" << endl;
07     return 0;
08 }
```

Programmas darbības piemērs:

```
Hello, world!
```

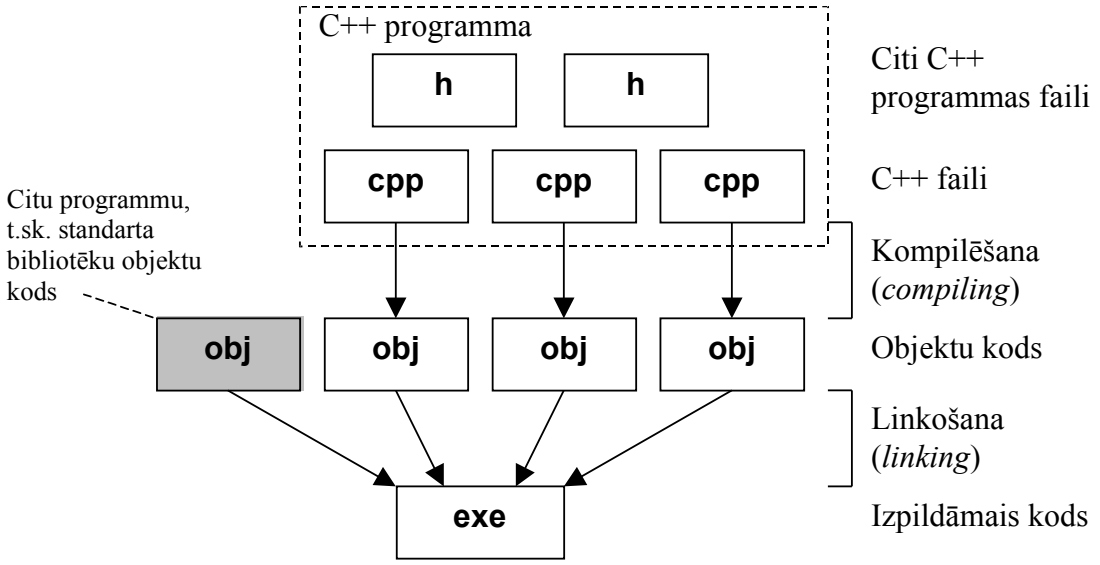
Programmas rakstīšanas mērķis parasti ir iegūt izpildāmu moduli (piemēram *.exe* failu), kas veic noteiktas darbības. Lai no programmas iegūtu izpildāmo moduli, nepieciešams kompilators.

Kompilators (*compiler*) ir datorprogramma, kas pārveido noteiktā programmēšanas valodā uzrakstītu programma par izpildāmu moduli (vispārīgā gadījumā – par programmu mašīnkodā, tādu, kuru dators var tiešā veidā izpildīt).



Attēls 1.1. Izpildāma moduļa iegūšana no triviālas C++ programmas

Lielākā daļa no C++ izstrādes vidēm dod iespēju veikt triviālas C++ programmas kompilēšanu (izpildāmā moduļa iegūšanu) tādā veidā kā parādīts attēlā 1.1. Tomēr dažas izstrādes vides pieprasa veidot t.s. projektu, kas apskatīts laboratorijas darbu daļā. Kā jau tika minēts, vispārīgā gadījumā C++ programma sastāv no vairākiem failiem, turklāt īpaša vieta tajā ir C++ failiem (sk. attēlu 1.2).



Attēls 1.2. Izpildāma moduļa iegūšana no C++ programmas vispārīgā gadījumā

Kaut arī sarunu valodā visu izpildāmā moduļa iegūšanas procesu sauc par kompilēšanu, tomēr šis process sastāv no divām fāzēm, no kurām par kompilēšanu sauc tikai pirmo:

1. fāze. Kompilēšana. Katram C++ failam tiek uzbūvēts atbilstošs **objektu fails** (faila paplašinājums parasti *.obj* vai *.o*), kopumā veidojot **objektu kodu**;

2. fāze. Linkošana. No iegūtā objektu koda, kā arī citu programmu, t.sk. standarta bibliotēku objektu koda tiek izveidots gala produkts – izpildāmais kods (izpildāmais modulis vai moduļi).

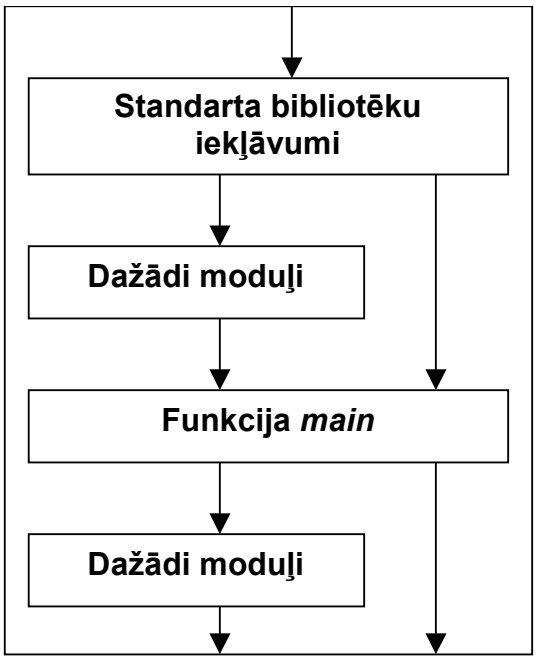
Pilnu izpildāmā koda iegūšanas procesu mēdz saukt arī par **uzbūvēšanu** (*build*), tomēr sarunu valodā, kā jau tika minēts, tiek lietots termins kompilēšana.

Abas izpildāmā koda uzbūvēšanas fāzes parasti tomēr veic viens un tas pats kompilators. Programmas failu organizāciju, kā arī pareizu kompilatora un citu palīgprogrammu izsaukšanas secību izpildāmā koda iegūšanai parasti nodrošina izstrādes vide (piemēram, *Dev-C++*, *Microsoft Visual C++*, *Borland C++*, *Anjuta*), izmantojot **projektu** mehānismu, tādējādi programmētājs var abstrahēties no šīs shēmas un "ar vienas pogas spiedienu" nonākt no programmas līdz rezultātam. Otrs bieži lietots variants programmas izpildāmā koda uzbūvēšanai, kas īpaši izplatīts *Unix/Linux* platformās, ir t.s. ***make*** mehānisms.

Turpmāk, apgūstot dažādas C++ konstrukcijas, parasti pietiks ar C++ programmām ar viena faila struktūru.

1.2.2. C++ programmas struktūra

C++ programma no valodas konstrukciju viedokļa ir galvenokārt dažādu C++ moduļu kopums, starp kuriem viens ir galvenais modulis jeb t.s. galvenā funkcija ***main***. Tādējādi triviāla C++ programma sastāv tikai no funkcijas *main*, kas atbilst jēdzienam "galvenā programma" dažās citās programmēšanas valodās (sk. pirmkoda piemēru 1.1). Bez tam, atšķirībā no daudzām citām programmēšanas valodām, ikviena C++ programma (arī pati vienkāršākā) ir praktiski neiedomājama bez vismaz kādas standarta bibliotēkas iekļāvuma. Tas tādēļ, ka daudzas tipiskas funkcijas (piemēram, ievade, izvade, failu apstrāde, simbolu virkņu apstrāde, matemātiskās operācijas) nav iekļautas valodas kodolā.



Attēls 1.3. C++ programmas vienkāršota struktūra

1.3. Valodas C++ galvenās konstrukcijas

Viena no programmēšanas valodas (t.sk. C++) svarīgākajām konstrukcijām ir **priekšraksts**, kas tradicionāli gan latviešu, gan krievu literatūrā ir ticis saukts par operatoru.

Deklarācija (*declaration*) ir konstrukcija, kas paziņo, ka programmā tiks izmantots noteikts elements (visbiežāk mainīgais) vai elementi, iespējami uzstādot šim elementam sākotnējās vai noklusētās vērtības.

Deklarācija pēc izskata ir ļoti līdzīga priekšrakstam, kas aprakstīts zemāk, un daudzos gadījumos deklarāciju var uzskatīt par priekšrakstu.

Priekšraksts (*statement*) ir pabeigta valodas konstrukcija, kas veic noteiktu darbību.

Priekšraksti var tikt apvienoti, veidojot **moduļus** vai **saliktos priekšrakstus**. Tipiska konstrukcija priekšrakstu apvienošanai ir **bloks**.

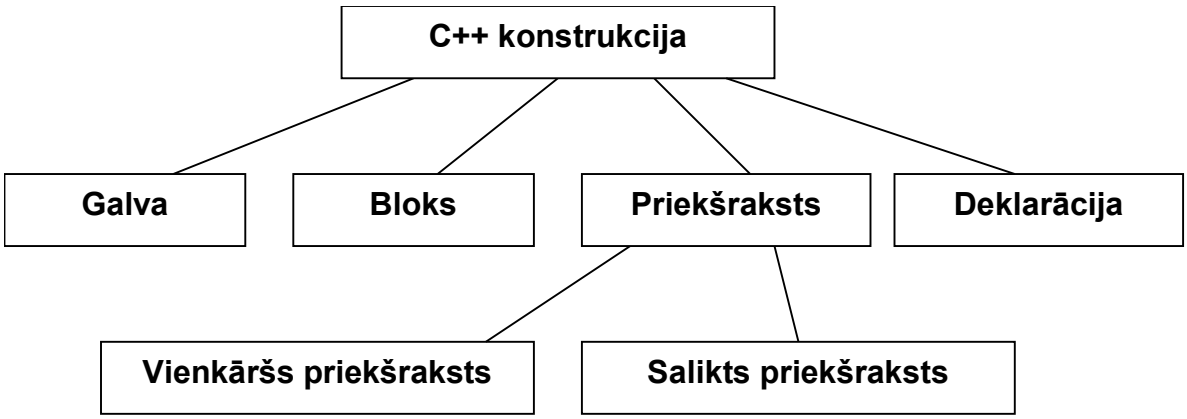
Bloks (*block*) ir priekšrakstu vai mainīgo deklarāciju virkne, kas ietverta figūriekavās.

Priekšrakstu apvienošana blokos var notikt daudzos līmeņos, tādējādi bloks var būt arī kāda priekšraksta sastāvdaļa, kā arī pats būt par priekšrakstu.

Par **moduli** (*module*) šajā mācību materiālā tiek saukts patstāvīgs programmas bloks.

Moduli ievada moduļa galva, kas lielā mērā to identificē.

Tipiskākie moduļu piemēri valodā C++ ir funkcijas, klases un datu struktūras.



Attēls 1.4. Valodas C++ konstrukciju klasifikācija

Salikti priekšraksti (*simple statements*) ir tādi priekšraksti, kuru sastāvā ietilpst citi priekšraksti. **Vienkāršu priekšrakstu** (*compound statements*) sastāvā nevar ietilpt citi operatori. Šajā mācību materiālā par saliktajiem priekšrakstiem sauksim tikai tādus, kuru sastāvā var ietilpt bloki, taču **kaskādes veidā** izsaucamus priekšrakstus (piemēram, piešķiršanas priekšrakstu) pieskaitīsim pie vienkāršajiem. Saliktu priekšrakstu piemēri ir izvēles un cikla priekšraksti, bet vienkāršu priekšrakstu piemēri – piešķiršanas, funkcijas izsaukuma, atdalīšanas priekšraksti.

Priekšrakstus, kuru pieraksts programmas tekstā nosacīti ievietojas vienā rindā (bet dažreiz arī vienkārši programmas teksta daļu, kas ievietojas vienā rindiņā), mēdz saukt par **instrukcijām** (*instruction*). Par instrukcijām kalpo vienkāršie priekšraksti un īsi pierakstāmi saliktie priekšraksti, un var teikt, ka programmas pamatmasu veido tieši instrukcijas – tās uzskatāmas par programmas struktūras ķieģeļiem.

Pirmkoda piemērā 1.2. modulis ir visa funkcija *main* (rindas 4-13), bloks ir šīs funkcijas pamatdaļa (rindas 5-13), galva ir funkcijas *main* galva (rinda 4), priekšraksti ir funkcijas *main* sastāvā (rindas 7-12), kā arī rinda 2 (kas gan neatbilst vienkāršotajai shēmai, kas redzama attēlā 1.3), deklarācija redzama rindā 6. Visi priekšraksti šajā piemērā ir arī instrukcijas, jo ietilpst vienā rindā.

Pirmkods 1.2. Vienkārša C++ programma izteiksmes izrēķināšanai (*int2program.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 int main ()
05 {
06     double x, y, z;
07     cout << "Input a numeric value:" << endl;
08     cin >> x;
09     y = 1.2;
10     z = x * y;
11     cout << x << '*' << y << '=' << z << endl;
12     return 0;
13 }
```

Programmas darbības piemērs:

```
Input a numeric value:
2.5
2.5*1.2=3
```

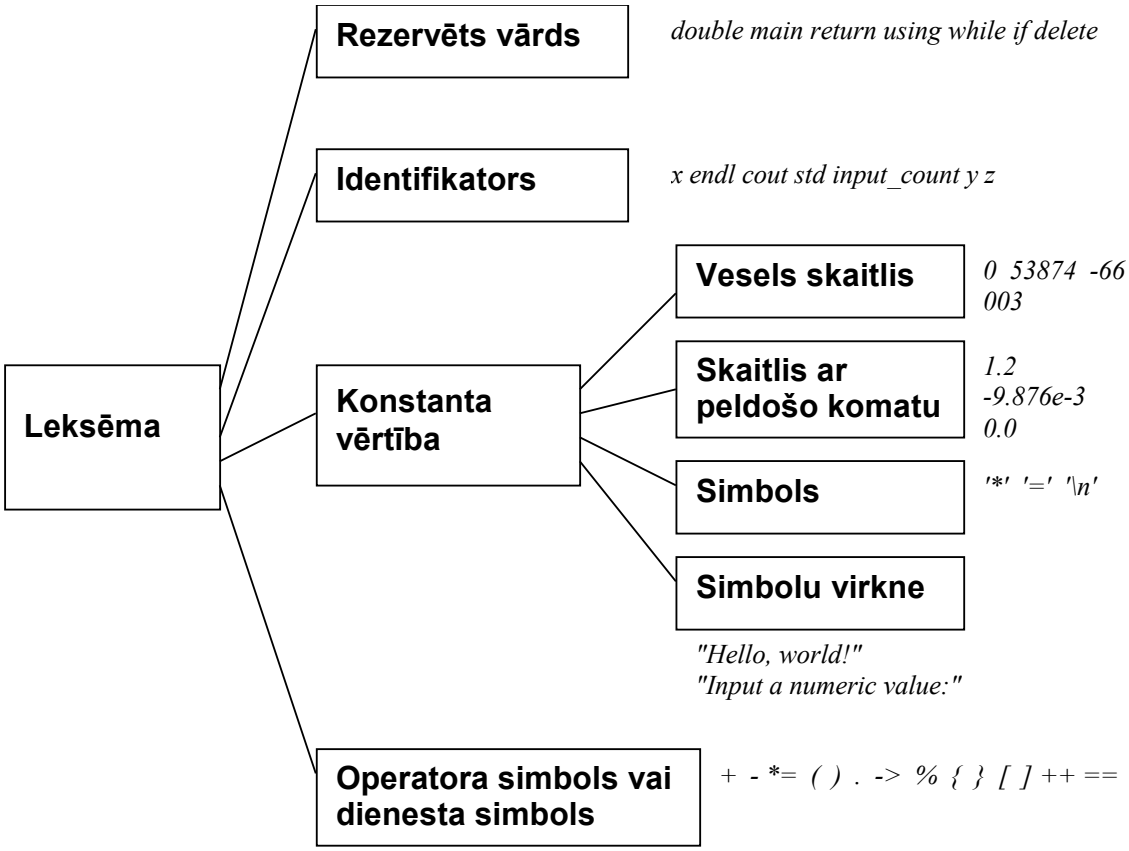
1.4. Valodas C++ leksiskās pamatvienības – leksēmas

C++ konstrukcijas, kas aprakstītas iepriekšējā sadaļā, sastāv no mazākām vienībām. Atkarībā no abstrakcijas pakāpes var izšķirt 2 līmeņu vienības – **leksiskās** un **sintaktiskās**. Leksiskā līmeņa vienības attiecas uz pareizu pierakstu (t.i., kā cilvēku valodā – vārdu pareizrakstību), bet sintaktiskā līmeņa vienības – jau uz valodas konstrukciju veidošanu (kā cilvēku valodā – teikuma priekšmets, izteicējs, palīgteikums).

No leksiskā viedokļa programma sastāv no **leksēmām**.

Leksēma (*lexeme*) (programmēšanas valodā) ir mazākā valodas vienība.

Piemēram, cilvēku valodā leksēmas (no programmēšanas valodu viedokļa) ir vārdi un pieturzīmes, bet viens burts kādā vārdā vairs nav leksēma.



Attēls 1.5. Valodas C++ leksēmu klasifikācija

*programmētāja ieviests vai standarta bibliotēkā sastopams vārds

Atšķirībā no priekšraksta (*statement*), leksēma nav pabeigta konstrukcija, un tās nozīme ir saprotama tikai kontekstā (līdzīgi, kā cilvēku valodā – ir grūti vai pat neiespējami saprast viena vārda jēgu tekstā, neredzot visu teikumu).

Praktiski leksēma ir viena rakstzīme vai vairāku pēc kārtas esošu rakstzīmju secība. Savukārt programma ir leksēmu secība.

Ir divi veidi, kā leksēmas var tikt atdalītas viena no otras:

- izmantojot atdalītājsimbolus (tukšums, tabulācija, jaunas rindiņas simbols (ENTER) u.c.),
- pēc konstrukcijas – simbola nepiederība dotajai leksēmai norāda uz nākošās leksēmas sākšanos (piemēram, simbolu virknē 1+2 ir 3 leksēmas, kaut arī nav atdalītājsimbolu).

Dažu leksēmu (leksēmu pāra) atdalīšanai obligāti jālieto atdalītājsimboli.

Valodā C++ atdalītājsimboliem visiem ir vienāda nozīme (nav svarīgi, vai atdalīšanai lietot vienu vai vairākus simbolus, un vai tukšumu vai jaunas rindiņas simbolu). (Programmēšanas valodā *Basic* jaunas rindiņas simbolu izmantojot cita atdalītāja vietā, var mainīties konstrukcijas sintakse.)

Identifikators.

Identifikators (*identifier*) ir mainīgā vai cita programmēšanas elementa vārds.

Rezervētajiem vārdiem valodā C++ ir noteikta nozīme, kuru nedrīkst mainīt, tomēr saviem mainīgajiem un citiem programmas elementiem programmētājs var brīvi izvēlēties identifikatorus, tomēr tie nedrīkst konfliktēt ar rezervētajiem vārdiem un citiem identifikatoriem. Identifikators var sākties ar burtu vai pasvītrojuma zīmi (_), bet pārējie simboli var būt arī cipari. Tipiskākais identifikatora piemērs ir mainīgā vārds.

C++ identifikatoru pieraksts (atšķirībā no daudzām programmēšanas valodām, kā, piemēram, *Basic*, *PASCAL*) ir reģistrjūtīgs (*case-sensitive*), t.i., piemēram, identifikatori *count* un *COUNT* ir divi dažādi.

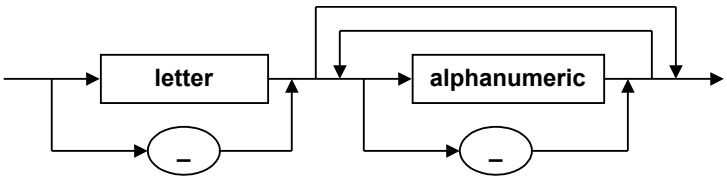
Korekti identifikatori ir, piemēram, šādi:

```
x sum input_count _x ExchangeRate_1 a2b3c
```

Šādas simbolu virknes nevar kalpot par identifikatoriem:

```
50 2b3c ExchangeRate-1 input.count %x
```

Sintakse 1.1. *identifier* (identifikators)



Sintakse 1.1a. *identifier* (identifikators) (alternatīva BNF formā)

<identifier> ::= (_ | <letter>) (_ | <alphanumeric>)*

Sintakse 1.2. *letter* (burts)

<letter> ::= a..z | A..Z

Sintakse 1.3. *alphanumeric* (burts vai cipars)

<alphanumeric> ::= <letter> | <digit>

Sintakse 1.4. *digit* (cipars)

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Vesels skaitlis.

Vesels skaitlis pierakstāms kā ciparu virkne, pirms kuras var atrasties plus vai mīnus zīme.

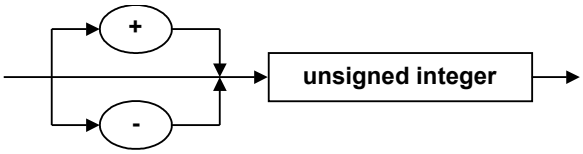
Korekti pierakstīti veseli skaitļi ir šādi:

```
1234 0 003 +28 -837
```

Šādas simbolu virknes neapzīmē veselus skaitļus:

```
50.0 3x
```

Sintakse 1.5. *integer* (vesels skaitlis)



Sintakse 1.5a. *integer* (vesels skaitlis) (alternatīva BNF formā)

<integer> ::= [+ | -] <unsigned integer>

Skaitlis ar peldošo komatu.

Skaitlis ar peldošo komatu apzīmē skaitlisku vērtību, kas nav vesels skaitlis. Vispārīgā gadījumā skaitlis ir vai nu vesels skaitlis vai skaitlis ar peldošo komatu.

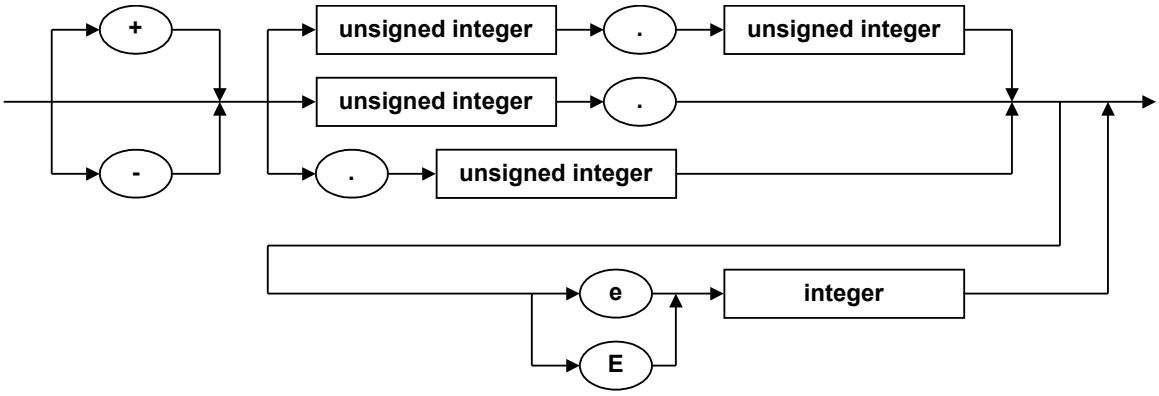
Sintakse 1.6. *number* (skaitlis)

<number> ::= <integer> | <floating point number>

Skaitlis ar peldošo komatu sastāv no veselās daļas un mantisas (daļas pēc decimālās zīmes), kam var sekot desmitnieku pakāpe, ko ievada simbols ‘e’: -1.2e-3 nozīmē $-1.2*10^{-3}$ jeb -0.0012.

Vai nu veselo daļu vai mantisu var izlaist: 0.3 var pierakstīt kā .3 , bet 4.0 kā 4. .

Sintakse 1.7. *floating point number* (skaitlis ar peldošo komatu)



Korekti skaitļi ar peldošo komatu ir:

```
1.0 –0.0 -.4 3. 1.23e4 2.e-3 .5e+2 6E1
```

Šādas simbolu virknes nevar kalpot par skaitļiem:

```
1.2.3 3..4 5.e 5.e1.1 . .e2
```

Konstante-simbols.

Konstante-simbols apzīmē vienu simbolu. To pieraksta, liekot atbilstošo rakstzīmi vienkāršajās pēdiņās (apostrofos):

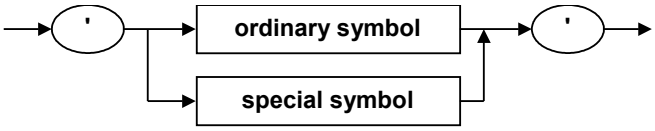
```
'a'  'B'  '1'  '@'  '/'  ' '
```

Izņēmums ir daži speciāli pierakstāmi simboli, kurus pieraksta, izmantojot 2 rakstzīmes, no kurām pirmā ir \ (**“atpakaļsvītra”**, *backslash*):

```
'\\'  '\''  '\"'  '\n'  '\t'
```

Šajā pierakstā attēloti tipiskākie speciāli pierakstāmie simboli: “atpakaļsvītra”, vienkāršā pēdiņa, dubultā pēdiņa, jaunas rindiņas simbols un tabulācija. “Atpakalsvītra” šajā pierakstā nodrošina speciālo simbolu pierakstu, un šajā “ampluā” to sauc par **atsoļa rakstzīmi** (*escape character*), kas darbojas pēc līdzības ar mēmo taustiņu uz latviskas klaviatūras.

Sintakse 1.8. ***constant-symbol*** **(konstante-simbols)**



Parastie simboli ir visas no klaviatūras iegūstamās rakstzīmes, izņemot "atpakaļsvītru", parasto pēdiņu un dubulto pēdiņu.

Sintakse 1.9. ***ordinary symbol*** **(parastais simbols)**

<ordinary symbol> ::= every keyboard character, except \ ' "

Sintakse 1.10. ***special symbol*** **(speciālais simbols)**

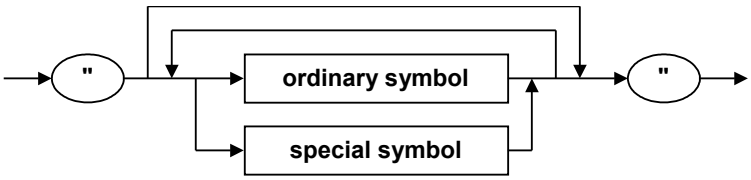
<special symbol> ::= \\ | \' | \" | \n | \t | \b | \f | \r | \v

Konstante-simbolu virkne.

Konstante-simbolu virkne apzīmē virkni no vairākiem simboliem. Virkne drīkst būt arī tukša (bez neviena simbola). To pieraksta, liekot atbilstošās rakstzīmes dubultajās pēdiņās, speciālos simbolus pierakstot tāpat kā konstantes- simbola gadījumā:

```
""  "a"  "Hello, World!"  "Hello,\nWorld!"  "\'Hello\'"
```

Sintakse 1.11. ***constant-string*** **(konstante-simbolu virkne)**

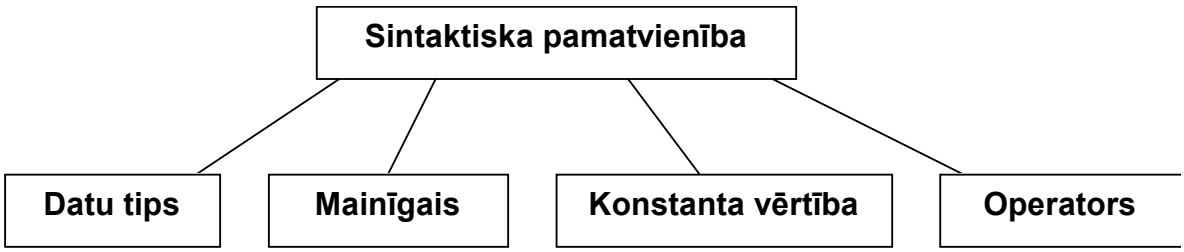


Citi simboli.

Citas leksēmas, kas sastopamas programmas tekstā ir operatoru simboli (piemēram, +, =, ::, &&, !=, %, [,]) un palīgsimboli, piemēram, parastās iekavas un figūriekavas.

1.5. Valodas C++ sintaktiskās pamatvienības

Galvenās valodas C++ sintaktiskās pamatvienības (vienības, no kurām sastāv priekšraksti) ir datu tipi, mainīgie, konstantas vērtības un operatori.



Attēls 1.6. Valodas C++ sintaktisko pamatvienību klasifikācija

Konstantās vērtības jau aprakstītas iepriekšējā nodaļā (1.4), bet datu tipi tiks aprakstīti vēlāk, tāpēc šajā nodaļā aprakstīti tikai mainīgie un operatori.

1.5.1. Mainīgie

Mainīgais (*variable*) ir vieta datora atmiņā, kur tiek glabāti noteikta tipa dati un kuru programmas tekstā identificē noteikts vārds – identifikators.

Mainīgais noteiktā programmas kontekstā **reprezentē** vienu no divām vai abas sekojošās ar programmas darbību saistītās komponentes:

- vietu, kurā var glabāt (ierakstīt) datus,
- vērtību, kas tiek glabāta mainīgajā.

Īpašs mainīgo veids (no izmantošanas viedokļa) ir norādes mainīgie, kuros glabātā vērtība ir kāda cita atmiņas apgabala adrese. Šajā gadījumā var teikt, ka mainīgais reprezentē atmiņas apgabala sākumu, kurā glabāt datus.

Tādējādi, var teikt, ka no datu veida viedokļa, ko glabā mainīgajā var izšķirt divu veidu mainīgos:

- parastie (datu) mainīgie,
- norādes mainīgie (*variables-pointers*).

No atmiņas izmantošanas viedokļa mainīgos iedala šādās grupās:

- parastie (automātiskie) (*authomatic variables*),
- statiskie (*static variables*),
- dinamiskie (*dynamic variables*).

No redzamības viedokļa dažādos programmas moduļos mainīgos iedala šādās grupās:

- globālie (*global variables*),
- nosaukumu telpu globālie,
- lokālie (*local variables*),
- datu struktūru iekšējie.

Dažādi mainīgo veidi tiks precīzāk aprakstīti tālākajās nodaļās.

1.5.2. Operatori

Operators (*operator*) ir programmas konstrukcijas elements, kas nosaka noteiktu darbību ar datiem.

Operatori kopā ar tajos iesaistītajiem datiem veido tādas valodas konstrukcijas, kā izteiksmes (aprakstītas nākamajās nodaļās) un priekšraksti. No pieraksta viedokļa operators ir viena (parasti) vai vairākas leksēmas, kas var būt izkliedētas vienas izteiksmes vai priekšraksta ietvaros (līdzīgi kā saiklis *gan..., gan* latviešu valodā).

Atkarībā no iesaistīto datu vienība skaita, operatori ir **unāri**, **bināri** vai **ternāri**. Lielākā daļa operatoru ir bināri (saistīti ar divām datu vienībām), tomēr tieši valodā C++ ir arī salīdzinošai daudz unāru operatoru.

Ar operatoriem neatraujami saistīts jēdziens ir prioritāte.

Prioritāte (*priority*) ir operatoru raksturojošs lielums, kas nosaka operatora izpildes vietu vairāku operatoru izpildes secībā.

Piemēram, reizināšanas augstāka prioritāte pār saskaitīšanu nodrošina, ka izteiksme *2+3*4* izpildīsies kā *2+(3*4)*.

Katram operatoram bez prioritātes ir noteikts arī **izpildes virziens**, kurš iegūst nozīmi gadījumos, kad blakusesošo operatoru prioritātes ir vienādas.

Unārajiem operatoriem, kā arī piešķiršanas operatoram izpildes virziens ir no labās uz kreiso (piemēram, *x=y=z* nozīmē *x=(y=z)*), bet pārējiem – no kreisās uz labo (piemēram, *x+y+z* nozīmē *(x+y)+z*)

Tabula 1.1. Galvenās C++ operatoru grupas

aritmētiskās darbības	`+ - * / %`
loģiskās operācijas	`&& || !`
piešķiršanas operatori	`= += -= *= /= %= ++ --`
salīdzināšanas operācijas	`== != < <= > >=`
struktūras elementa izvēle	`. ->`
masīva indeksācija	`[]`
funkcijas izsaukums	`()`
atmiņas vadības operatori	`new delete delete[]`
adreses iegūšanas operators	`&`
bitu operatori	`~ & ^ | << >>`
izņēmuma ģenerēšana	`throw`
nosacījuma operators	`?:`
redzamības apgabala izšķiršanas operators	`::`
secības operators	`,`
datu tipu apstrādes operatori	`typeid reinterpret_cast sizeof`

Pilns operatoru saraksts ar prioritātēm ir pieejams jebkurā C++ rokasgrāmatā.