Programmēšana un C++

Mācību materiāls programmēšanas apguvei uz valodas C++ bāzes

Saturs īsi

Saturs detalizēti

1. Datorzinātnes. Algoritms. Programmēšanas valoda

1.1. Datorzinātnes

> **Datorzinātnes** ir zinātņu disciplīna, kas nodarbojas ar tādām tēmām kā datoru uzbūve, programmatūras izstrāde, informācijas apstrāde, algoritmu izstrāde dažādu problēmu risināšanai.

Datorzinātnes radās 20. gs. 40. gadu sākumā uz algoritmu teorijas un matemātiskās loģikas bāzes, pievienojoties tikko izgudrotajām elektroniskajām skaitļošanas mašīnām. Lai darbotos ar datorzinātnēm saistītā sfērā, nepieciešamas zināšanas un/vai iemaņas šādos četros galvenajos virzienos: algoritmiskā domāšana, informācijas reprezentācija, programmēšana un sistēmu projektēšana.

Datorzinātne ir cieši saistīta ar matemātiku un mūsdienās to pat stipri ietekmē.

Datorzinātnes var iedalīt 12 galvenajās sadaļās:

- Algoritmi un datu struktūras.
- Programmēšanas valodas.
- Datoru arhitektūra.
- Operētājsistēmas un datortīkli.
- Programmatūras izstrāde.
- Datu bāzes un informācijas meklēšana.
- Mākslīgais intelekts un robotu tehnika.
- Datorgrafika.
- Cilvēka un datora sadarbība.
- Skaitļošanas matemātika.
- Lietišķā informātika.
- Bioinformātika.

Lai mūsdienās kļūtu par labu speciālistu informācijas tehnoloģiju jomā, būtu nepieciešams kompleksi apgūt visas datorzinātņu nozares.

1.2. Algoritms

Algoritms ir viens no programmēšanas un visas datoru zinātnes centrālajiem jēdzieniem.

> **Algoritms** ir instrukciju virkne, kas nosaka noteiktas problēmas risināšanas veidu.

> **Algoritms** ir precīzu, viennozīmīgu instrukciju virkne, kuru izpildes rezultātā tiek atrisināts noteikts uzdevums.

Algoritms ir stingra, precīza un galīga noteikumu sistēma, kas nosaka darbību virkni izpildīšanai ar noteiktiem objektiem un pēc galīga soļu skaita sasniedz uzstādīto mērķi.

Algoritms ir sakārtots viennozīmīgu un izpildāmu etapu kopums, kas apraksta noteiktu pabeigtu procesu.

Algoritma piemērs parādīts Att. 1.1.

```
Solis 1. Piešķirt mainīgajiem M un N divu ievadīto skaitļu
         vērtības.
Solis 2. Izdalīt M ar N un atlikumu piešķirt mainīgajam P.
Solis 3. Ja P vērtība nav vienāda ar 0,
               3.1. tad piešķirt mainīgajam M mainīgā N vērtību,
                    bet mainīgajam N – P vērtību, un doties
                    atpakaļ uz soli 2,
               3.2. pretējā gadījumā doties uz soli 4.
Solis 4. Algoritms apstājas. Lielākais kopīgais dalītājs ir
         vērtība, kas glabājas mainīgajā N.
```

Att. 1.1. Eiklīda algoritms divu veselu pozitīvu skaitļu lielākā kopīgā dalītāja atrašanai

No algoritma definīcijām tiešā vai netiešā veidā izriet vairākas algoritmu īpašības vai prasības pret algoritmiem:

- Instrukciju (etapu) sakārtojums. Tas norāda, ka algoritmā tā etapu izpildes kārtībai ir jābūt stingri noteiktai. Tas nenozīmē, ka algoritms vienmēr tiks izpildīts tieši kā instrukciju virkne, jo eksistē, piemēram, paralēlie algoritmi, kas ietver vairākas paralēlas izpildes virknes.
- Instrukciju (etapu) izpildāmība. Katram algoritma etapam un algoritmam kopumā jābūt izpildāmam, turklāt pietiekoši efektīvi izpildāmam. Citādi tas vispār nebūs algoritms, vai arī būs nosaucams par neefektīvu. Praksē bieži vien ir grūti izšķirt, vai noteikta instrukciju virkne ir uzskatāma par neefektīvu algoritmu vai vispār nav saucama par algoritmu.

Sekojošā instrukciju virkne (Att. 1.2) "algoritms lielākā naturālā skaitļa atrašanai" noteikti neatbilst algoritma definīcijai, jo pirmie divi soļi nav izpildāmi:

```
Solis 1. Izveidot visu naturālo skaitļu sarakstu.
Solis 2. Sakārtot sarakstu dilstošā secībā.
Solis 3. Izvēlēties pirmo skaitli no saraksta.
```

Att. 1.2. Darbību virkne, kas pretendē uz lielākā naturālā skaitļa atrašanu

- Instrukciju (etapu) viennozīmība. Katrā algoritma solī ir jābūt pietiekoši daudz informācijas, lai dotā etapa izpilde tiktu viennozīmīgi noteikta un neprasītu, piemēram, kāda radoša procesa iejaukšanos.
- Algoritmam ir ieeja (ieejā var tikt padoti noteikti objekti).
- Algoritmam ir izeja (izejā tiek iegūti noteikti objekti).

1.3. Algoritma formalizētas pieraksta metodes

Algoritmus var pierakstīt dažādos – vairāk vai mazāk formālos veidos. Att. 1.1 parāda daļēji formālu algoritma pierakstu. Tālāk tiks apskatītas arī citas algoritma pieraksta metodes – blokshēmas, UML aktivitāšu diagrammas, Nassi-Šneidermana struktogrammas, pseidokods un programmēšanas valodas.

1.3.1. Algoritma vadības konstrukcijas

Jebkura algoritma pieraksts sastāv no divu veidu informācijas:

- izpildāmās instrukcijas (etapi),
- **vadības konstrukcijas**, kas nosaka instrukciju secību vienai aiz otras.

Noteiktu algoritma pieraksta metodi raksturo tieši otrās, t.i., vadības konstrukcijas, bet pašu izpildāmo instrukciju pieraksts bieži vien vairāk atkarīgs no stila. Šajā materiālā instrukciju pierakstam grafiskajās metodēs un pseidokodā tiks lietota valodas PASCAL sintakse.

Algoritma pierakstā tiek izšķirtas 3 galvenās vadības konstrukcijas, bez jebkuras no tām nav iedomājama neviena algoritmu pieraksta metode:

- **Secība** (*sequence*). Nosaka vienas instrukcijas izpildi tieši aiz otras instrukcijas.
- **Zarošanās** (*branching*). Atkarībā no noteikta kritērija izpildīšanās nosaka, pa kuru no zariem turpināsies algoritma izpilde.
- **Cikls** (*looping*). Nosaka instrukciju kopuma vairākkārtēju izpildi.

Att. 1.1 pierakstītajā algoritmā secība parādās pārejā no soļa 1 uz 2 un no 2 uz 3, zarošanās ietverta solī 3, bet ciklu nodrošina lēkšana atpakaļ no soļa 3.1 uz 2.

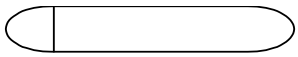
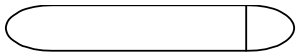
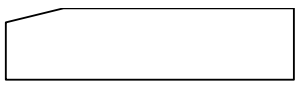
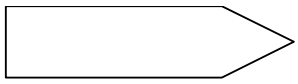
1.3.2. Blokshēma

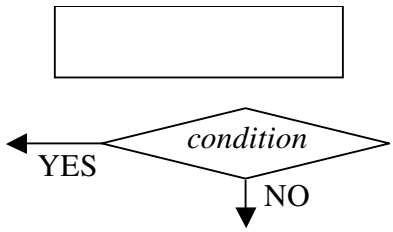
Blokshēma (*flowchart*) ir ļoti populāra algoritmu grafiskā pieraksta metode, kurai ir daudz dažādu paveidu. Klasiskajā variantā blokshēmas pieraksts neietver ciklu kā neatkarīgu konstrukciju – to nodrošina zarošanās konstrukcija kopā ar atgriezenisko saiti.

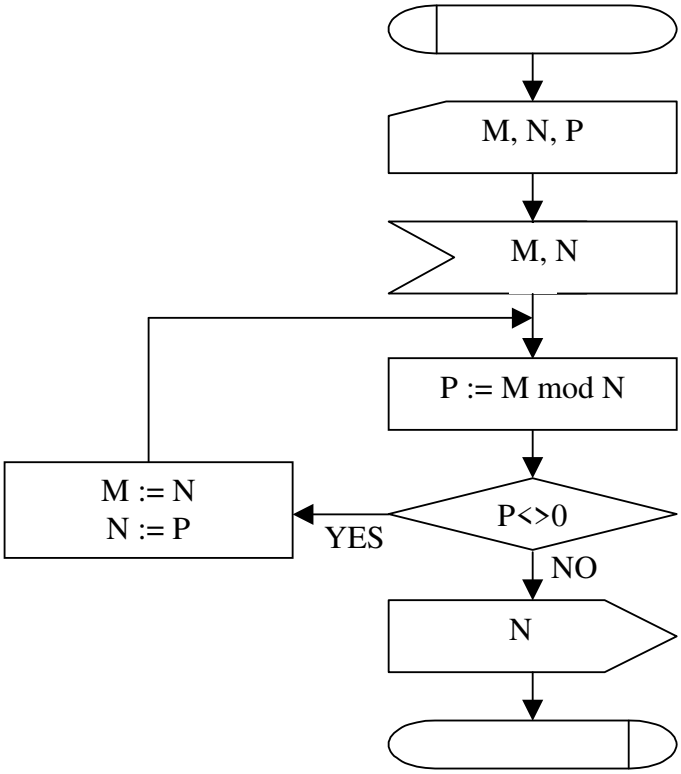
Piezīme. Šeit un tālāk algoritmu pieraksta piemēros, atbilstoši PASCAL sintaksei, ***mod*** apzīmē dalījuma atlikuma iegūšanas operāciju, bet **:=** ir piešķiršanas operators.

Tab. 1.1.

Blokshēmas konstrukciju apraksts

	Sākums
	Beigas
	Mainīgo deklarēšana
	Ievade
	Izvade





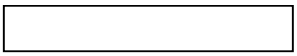
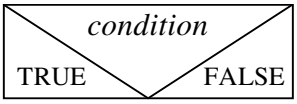
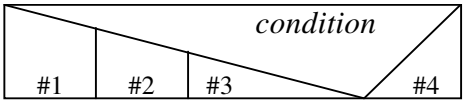
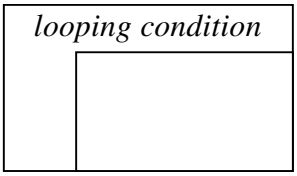
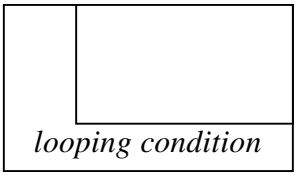
Att. 1.3. Blokshēmas piemērs. Eiklīda algoritms divu veselu pozitīvu skaitļu lielākā kopīgā dalītāja atrašanai

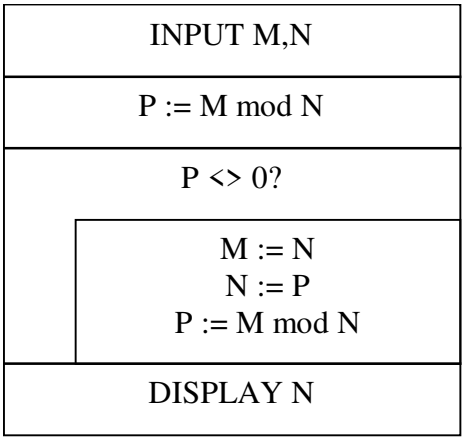
1.3.3. Nassi-Šneidermana struktogramma

Nassi-Šneidermana struktogramma (*Nassi-Schneidermann structogram*) nodrošina kompaktāku un elegantāku pierakstu, kas ir viegli lasāms, tomēr šīs metodes mīnuss ir neērtā uzturēšana – izmaiņu gadījumā visticamāk jāpārzīmē liela daļa diagrammas.

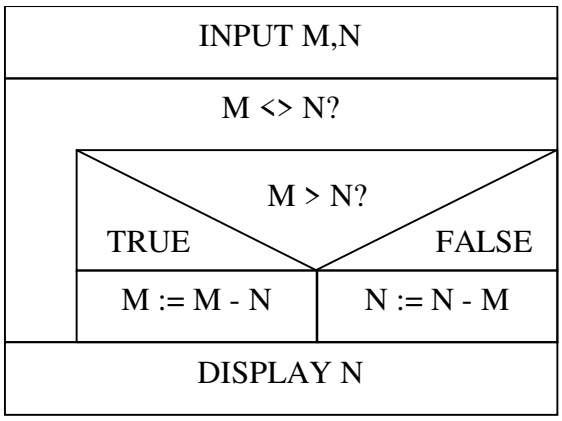
Tab. 1.2.

Nassi-Šneidermana struktogrammas konstrukciju apraksts

	Darbības
	Izvēle
	Izvēle ar vairākkārtēju zarošanos (*multiple branching*)
	Cikls ar priekšnosacījumu
	Cikls ar pēcnosacījumu



Att. 1.4. Nassi-Šneidermana struktogrammas piemērs. Eiklīda algoritms divu veselu pozitīvu skaitļu lielākā kopīgā dalītāja atrašanai



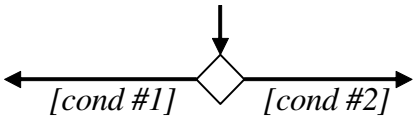
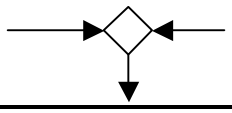
Att. 1.5. Nassi-Šneidermana struktogrammas piemērs. Eiklīda algoritma variants, dalīšanas vietā izmantojot atņemšanu

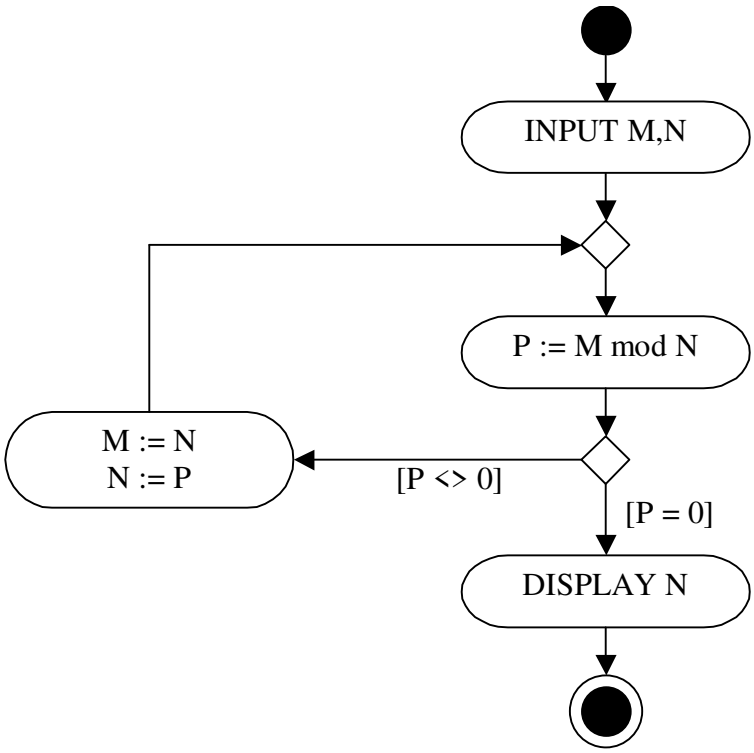
1.3.4. UML aktivitāšu diagramma

1.3.4. UML aktivitāšu diagramma (*UML activity diagram*) ir ***UML*** (*unified modeling language*) rīks, kas ir ļoti līdzīgs blokshēmai.

Tab. 1.3.

UML aktivitāšu diagrammas konstrukciju apraksts

	Sākums
	Beigas
	Darbības
	Zarošanās (*branching*)
	Saplūšana (*merge*)



Att. 1.6. UML aktivitāšu diagrammas piemērs. Eiklīda algoritms divu veselu pozitīvu skaitļu lielākā kopīgā dalītāja atrašanai

1.3.5. Pseidokods

Pseidokods (*pseudocode*) ir pusformāls algoritmu pieraksta veids, kas atgādina pierakstu kādā programmēšanas valodā. Pseidokods, atšķirībā no programmēšanas valodām, dod iespēju pierakstīt vairākas idejas, darbības un konstrukcijas neformāli, tādējādi nodrošinot, ka algoritms tiek pierakstīts īsāk un lasītājam vieglāk uztveramā formā, kā arī samazinās piesaiste kādai konkrētai programmēšanas valodai. Aritmētisku un citu darbību pieraksts notiek, izmantojot matemātisko notāciju, bet piešķiršanas operators bieži vien ir bultiņas formā no vērtības, kuru piešķir, uz mainīgo, kurā ieraksta ←.

Pseidokoda pieraksta sintakse balstās uz kādas labi zināmas programmēšanas valodas sintaksi (parasti PASCAL vai BASIC).

```
GREATESTCOMMONDENOMINATOR (M,N)
     P ← M MOD N
     WHILE P ≠ 0 DO
          M ← N
          N ← P
          P ← M MOD N
     END WHILE
     RETURN N
END FUNCTION
```

Att. 1.7. Pseidokoda piemērs. Eiklīda algoritms divu veselu pozitīvu skaitļu lielākā kopīgā dalītāja atrašanai, kas noformēts funkcijas formā

```
GREATESTCOMMONDENOMINATOR (M,N)
     WHILE M ≠ N DO
          IF M > N
               M ← M - N
          ELSE
               N ← N - M
          END IF
     END WHILE
     RETURN N
END FUNCTION
```

Att. 1.8. Pseidokoda piemērs. Eiklīda algoritma variants, dalīšanas vietā izmantojot atņemšanu

Ir pseidokoda pieraksta sintakse, kurā netiek lietoti bloku noslēdzošie apzīmējumi "END", tādējādi visu programmas strukturēšanu uzliekot atkāpju izmantošanai pirms komandām. Nākošais piemērs ir iepriekšējā piemēra pārveidojums tieši šādā veidā:

```
GREATESTCOMMONDENOMINATOR (M,N)
WHILE M ≠ N DO
     IF M > N
          M ← M - N
     ELSE
          N ← N - M
RETURN N
```

Att. 1.9. Pseidokoda piemērs. Eiklīda algoritma variants, dalīšanas vietā izmantojot atņemšanu. Tā pati programma, kas iepriekšējā, bet ar sintaksi bez noslēdzošajiem "END"

```
GREATESTCOMMONDENOMINATOR (M,N)
     WHILE M ≠ N DO
          Lielākajam no M un N
          piešķir lielākā un mazākā starpību
     END WHILE
     RETURN N
END FUNCTION
```

Att. 1.10. Pseidokoda piemērs. Eiklīda algoritma variants, dalīšanas vietā izmantojot atņemšanu un izmantojot arī neformālus pieraksta elementus (algoritms identisks Att. 1.8)

1.4. Programmēšanas valodas

Programmēšanas valoda ir algoritma pieraksta forma, kas nodrošina automātisku algoritma izpildi datorā vai arī ērti pārveidojama uz kādu datorā automātiski izpildāmu formu.

1.4.1. Programmēšanas valodu izcelšanās un agrīnās paaudzes

Sākotnēji visi algoritmi tika rakstīti datorā uzreiz **mašīnkodā** (*machine code*). Tāda pieeja padarīja sarežģītu jau tā grūto algoritmu izstrādes procesu. Mašīnkodā rakstiskas instrukcijas skaitliskā veidā reprezentēja gan komandas, gan vērtības un atmiņas adreses (Att. 1.11).

Att. 1.11. Mašīnkodā rakstītas programmas piemērs, pierakstīts sešpadsmitnieku sistēmā. Programma saskaita skaitļus, kas glabājas šūnās pēc adresēm 5A un 5B un rezultātu ievieto šūnā ar adresi 5C.

```
155A
165B
5056
305C
C000
```

Pirmais solis programmēšanas darba atvieglošanai bija atteikšanās no komandu un to operandu pieraksta tieši tajā formā, kādā tie tiek izmantoti mašīnā. Tādējādi skaitliskā komandu pieraksta vietā plaši sāka pielietot **mnemonisko** (jeb vārdisko) pierakstu. Arī atmiņas apgabalus noteiktā veidā nosauca ar vārdiem (**identifikatoriem**). Specifisks identifikatoru lietošanas piemērs ir procesoru reģistru nosaukšana, piemēram, R0, R1.

Att. 1.12. Mnemoniskā pierakstā (asemblerā) rakstītas programmas piemērs. Programma saskaita skaitļus, kas glabājas šūnās pēc adresēm 5A un 5B un rezultātu ievieto šūnā ar adresi 5C (tas pats, kas Att. 1.11).

```
LD R5,NUMBER1
LD R6,NUMBER2
ADDI R0,R5,R6
ST R0,TOTAL
HLT
```

Sākotnēji programmētāji izmantoja šādu pierakstu tikai programmu izstrādei uz papīra, taču vēlāk kļuva skaidrs, ka mnemoniskā programmas teksta pārveidošanu mašīnkodā varētu veikt pats dators. Šim nolūkam tika izstrādātas speciālas programmas, sauktas par **asembleriem** (*assembler*). Pašās pieraksta sistēmas mnemoniskā veidā sāka saukt par asembleru valodām. Savulaik asembleru valodu izstrāde tika uzskatīta par milzīgu soli uz priekšu programmēšanas tehnoloģiju attīstībā. Daudzi uzskatīja, ka tā ir pavisam cita programmēšanas valodu paaudze, un vēlāk asembleru sāka saukt par **otrās paaudzes programmēšanas valodām**, bet mašīnkodu pieskaitīja **pirmās paaudzes valodām**. Kaut arī asembleru valodas bija liels solis uz priekšu, tomēr jāatzīst, ka pēc būtības tā valodas konstrukcijas sakrita ar mašīnkodu, un atšķirība bija vien to sintaktiskajā pierakstā.

Asemblera valodā uzrakstīta programma ir cieši piesaistīta mašīnai, kurai tā ir rakstīta, tāpēc universālu programmu rakstīšanai bija nepieciešamas augstāka līmeņa – platformu neatkarīgas konstrukcijas. To piedāvāja trešās paaudzes programmēšanas valodas, no kurām tipiskākie agrīnie piemēri ir ***FORTRAN*** (*FORmula TRANslator*) un ***COBOL*** (*COmmon Business-Oriented Language*).

Lai augsta līmeņa programmēšanas valodā rakstītu programmu pārveidotu mašīnkodā, ir nepieciešama speciāla programma – **translators** (*translator*). Translators no vienas puses ir līdzīgs asembleram, tomēr translatoriem, atšķirībā no asembleriem, bieži nācās apvienot (*compile*) vairākas mašīnkoda instrukcijas, lai nodrošinātu vienas augsta līmeņa konstrukcijas izpildi, tāpēc translatorus bieži sauc arī par **kompilatoriem** (*compilers*).

Populāra kompilatoru alternatīva ir **interpretatori** (*interpreters*), programmas, kuras nodrošina augsta līmeņa programmēšanas valodā pierakstītas programmas izpildi tiešā veidā, tā vietā, lai izveidotu programmas kopiju mašīnkodā, kuru izpildīt vēlāk.

1.4.2. Programmēšanas paradigmas

Iepriekšējā sadaļa aprakstītā valodu attīstība gāja it kā pa lineāru skalu – no pirmās paaudzes uz priekšu, t.i., no pilnīgas mašīnu atkarības virzienā uz pielāgošanos cilvēka vajadzībām. Tomēr šāda programmēšanas valodu attīstības shēma tālāk nekā līdz trešajai paaudzei ir grūti piemērojama programmēšanas valodu klasifikācijā, jo tālāk valodas sāka attīstīties savādāk un attīstība notika dažādos virzienos. Šos dažādos virzienus mēdz saukt par paradigmām (*paradigms*). Paradigma ir virziens, veids vai pieeja plašā nozīmē. Populārākās programmēšanas valodu paradigmas ir imperatīvā, deklaratīvā, funkcionālā un objektorientētā.

Imperatīvā (jeb procedurālā) paradigma reprezentē tradicionālo veidu, kā notiek programmēšanas process. Imperatīvās paradigmas ietvaros programmēšanas process ir komandu virknes pierakstīšana noteiktu operāciju veikšanai. Imperatīvo paradigmu pārstāv mašīnkoda valodas, asambleri, kā arī, piemēram, FORTRAN, COBOL, BASIC, C, Pascal.

Deklaratīvā paradigma, atšķirībā no imperatīvās, nenosaka, kā izpildīt noteiktu uzdevumu, bet gan "kas ir problēma" vai "kā problēma varētu tikt reprezentēta". Tipiskākais deklaratīvās programmēšanas pārstāvis ir Prolog, kas no tās realizācijas aspekta tiek saukta arī par **loģisko programmēšanas valodu**. Kaut kādā nozīmē pie deklaratīvās paradigmas pieder arī SQL pieprasījumu valoda.

Funkcionālās paradigmas ietvaros programmēšanas process ir kā programmas konstruēšana no "melnajām kastēm", katra no kurām saņem noteiktus datus ievadā un izejā atgriež noteiktu rezultātu. Matemātikā tādas "melnās kastes" sauc par funkcijām. Šīs paradigmas pārstāvji ir Lisp un Haskell.

Objektorientētās programmēšanas paradigma kaut kādā mērā būtu uzskatāma par imperatīvās paradigmas apakšgadījumu. Atšķirība ir tā, ka galvenais datu elements šeit ir **aktīvs "objekts"**, atšķirībā no pasīva elementa imperatīvajā pieejā. Šīs paradigmas pārstāvji ir Smalltalk, C++, Visual Basic, Java.

2. C++ programmas uzbūves pamatprincipi

2.1. 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 veidošana un uzturēšana. 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** (*Bjarne 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, nekā 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.

Kaut arī sākta izstrādāt 70-gadu beigās, C++ parādījās tikai 80-gadu vidū, bet pirmais tās standarts **C++98** – tikai 1998. gadā.

Ilgu laiku C++ nekas nemainījās (ja neskaita nelielus uzlabojumus C++03 2003. gadā), un liela daļa funkcionalitātes palika ārpus standarta, un papildus funkcionalitātei tika veidotas trešo pušu bibliotēkas, piemēram, *boost*.

Tomēr "ledus beidzot izkustējās" 2011. gadā ar standartu C++11, kam sekoja C++14, C++17 un C++20, kur jau tika ieviesti būtiski papildinājumi (izstrādes stadijā ir C++23).

Šī mācību materiāla ietvaros tiks izmantota izstrādes vide *Code::Blocks* ar kompilatoru g++ un C++ versiju **C++14**.

2.2. C++ programmas failu struktūra

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 Att. 2.1. Šī programma, kā jau redzams tās darbības piemērā, uz ekrāna izdrukā tekstu "Hello, world!".

Sākot ar šo programmu un turpmāk, materiālā programmas vai to fragmenti tiks marķēti ar biezu svītru programmas koda kreisajā pusē, pēc tam var sekot **programmas darbības piemērs**, kurš marķēts ar trīskāršo svītru kreisajā pusē. Ja programmas darbības piemērā parādās lietotāja ievads no klaviatūras, tas tiek atzīmēts treknināti (*bold*).

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

int main ()
{
    cout << "Hello, world!" << endl;
    return 0;
}
```

```
Hello, world!
```

Att. 2.1. Triviāla C++ programma

Ko pēc šīs vienkāršās programmas var spriest vai sākt stāstīt par C++ programmas struktūru:

- Programmas galveno daļu (to, kas veic darbības) veido instrukcijas, kuru pieraksts beidzas ar semikolu.
- Programmas instrukcijas var tikt apvienotas blokos, ko ietver figūriekavas {}.
- Parasti figūriekavu bloks ir kādas funkcijas daļa, kuras nosaukums ir rindā pirms figūriekavu bloka – šajā gadījumā tā ir funkcija *main*.
- Programma var sastāvēt arī no vairākām funkcijām, bet vismaz viena funkcija – funkcija *main* noteikti atrodas programmā, un tieši no šīs funkcijas sākas programmas izpilde.
- Jebkurā funkcijā parasti atrodas vismaz viena (un parasti tieši viena) rindiņa, kas satur atslēgas vārdu *return*, un kas saistīta ar funkcijas izpildes beigšanu (t.i., izeju no funkcijas).
- C++ programma praktiski nav iedomājama bez kādu bibliotēku iekļaušanas (tas tādēļ, ka, atšķirībā no daudzām citām programmēšanas valodām, pat relatīvi ikdienišķas lietas, kā, piemēram, ievade, izvade un failu apstrāde, nav iekļautas valodas "kodolā", no programmēšanas viedokļa gan tas ir tikai sintakses jautājums). Vienas bibliotēkas iekļaušanu nodrošina komanda *#include*, pēc kā seko bibliotēkas vārds stūra iekavās <>:
  - šajā gadījumā tiek iekļauta bibliotēka *<iostream>*, kas atbild par standarta ievadi un izvadi (no klaviatūras, uz ekrāna),
  - Aiz bibliotēku iekļaušanas papildus jālieto rinda *using namespace std;* (visi standarta bobliotēku elementi ir ievietoti t.s. nosaukumu telpā std, un šāda rindiņa ļauj piekļūt elementiem tiešā veidā, bez *std* uzrādīšanas),
  - programmā bez standarta bibliotēkām var tikt izmantotas arī citas bibliotēkas, tādā gadījumā to failu nosaukumus liek nevis stūra iekavās, bet gan dubultpēdiņās, piemēram *#include "test.h"*.

Tātad, ja programmas sākumā esam uzrādījuši *using namespace std;*, varam rakstīt:

```
    cout << "Hello, world!" << endl;
```

Ja nē, tad *std* papildus jāpievieno pie katra no bibliotēkas elementiem:

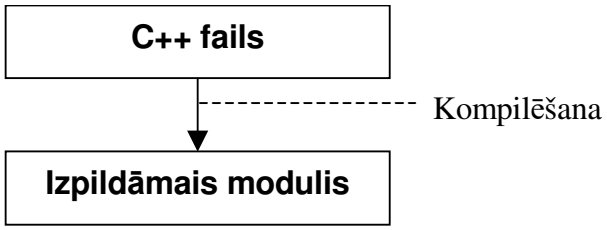
```
    std::cout << "Hello, world!" << std::endl;
```

2.3. C++ programmas kompilēšana un palaišana

2.3.1. Kompilēšanas pamatprincipi

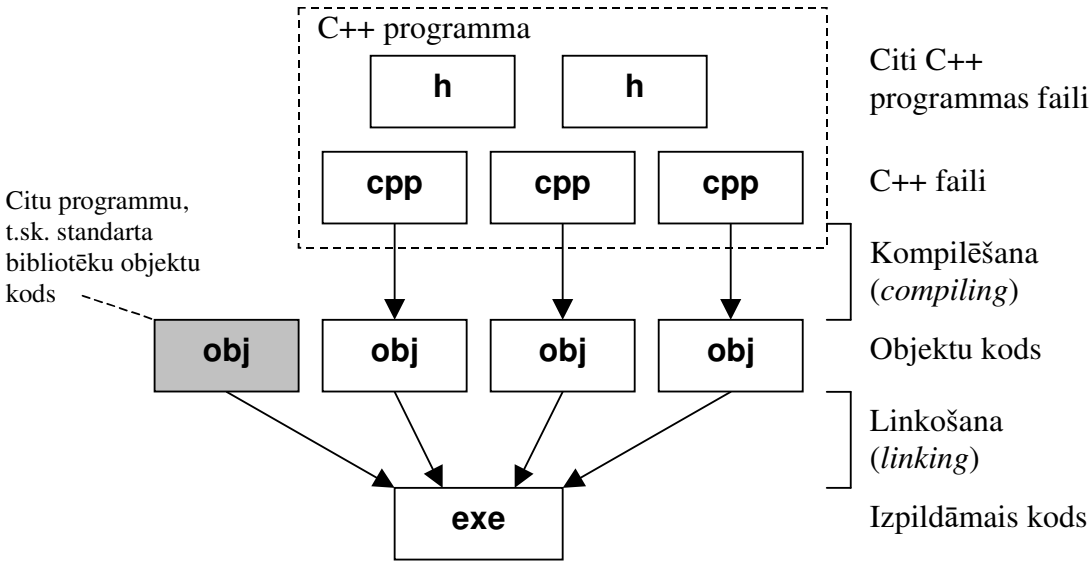
Programmas rakstīšanas mērķis parasti ir iegūt izpildāmu moduli (piemēram *.exe* failu), kas veic noteiktas darbības. Lai no programmas iegūtu izpildāmo moduli, nepieciešams kompilators.

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



Att. 2.2. Izpildāma moduļa triviāla iegūšana no C++ programmas

Lielākā daļa no C++ izstrādes vidēm dod iespēju veikt triviālas C++ programmas kompilēšanu (izpildāmā moduļa iegūšanu) tādā veidā kā parādīts Att. 2.2. 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. Att. 2.3).



Att. 2.3. 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++*, *Code::Blocks*), izmantojot **projektu** (*project*) 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.

2.3.2. Programmas kompilēšana

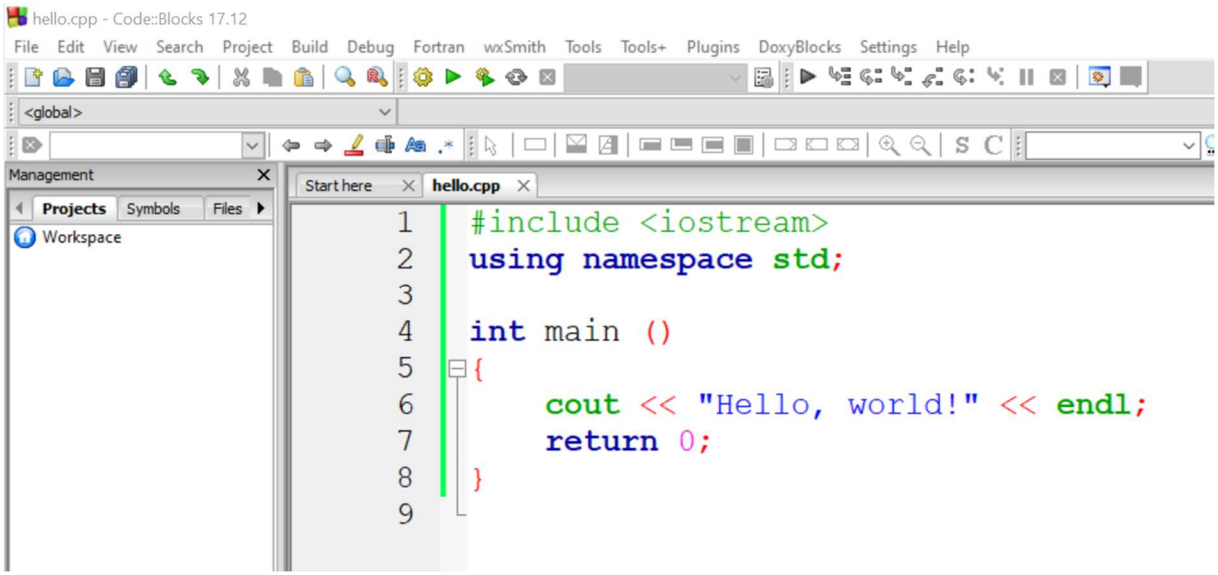
Programmas kompilēšana, izmantojot noteiktu izstrādes vidi, parasti iespējama vismaz divos variantos: **triviālajā** (bez projekta veidošanas) un **parastajā** (ar projekta veidošanu). Jebkura izstrādes vide, kompilējot programmu, vienu vai vairākas reizes izsauc kompilatoru (piemēram, izstrādes vide *Code::Blocks* pēc noklusēšanas izmanto kompilatorus *gcc* un *g++*).

2.3.2.1. Triviālais variants programmas kompilēšanā

Triviālais variants pieejams tikai programmām, kuras sastāv no viena *.cpp* faila (citi iekļautie faili, piemēram, *.h*, papildus drīkst būt). Bez tam, lai programmu kompilētu šajā variantā, iegūstamajam izpildāmajam modulim ir jābūt teksta režīmā jeb ar konsoles tipu (triviālais variants var nederēt, ja būs nepieciešama lietotne ar logu izmantošanu). Triviālais variants ir ļoti ērts izmantošanā, un to atbalsta gandrīz visas izstrādes vides.

Programmas kompilēšana triviālajā variantā (Att. 2.2, Att. 2.4):

- atvērt *.cpp* failu ar izstrādes vides teksta redaktoru,
- ar komandu *compile* vai *build* veikt kompilēšanu, iegūstot izpildāmo moduli.



Att. 2.4. Faila atvēršana izstrādes vidē Code::Blocks

Triviālajā variantā kompilētas programmas rezultāts parasti ir tikai izpildāmais modulis (piemēram, *.exe* fails).

2.3.2.2. Projekta veidošana un kompilēšana izstrādes vidē

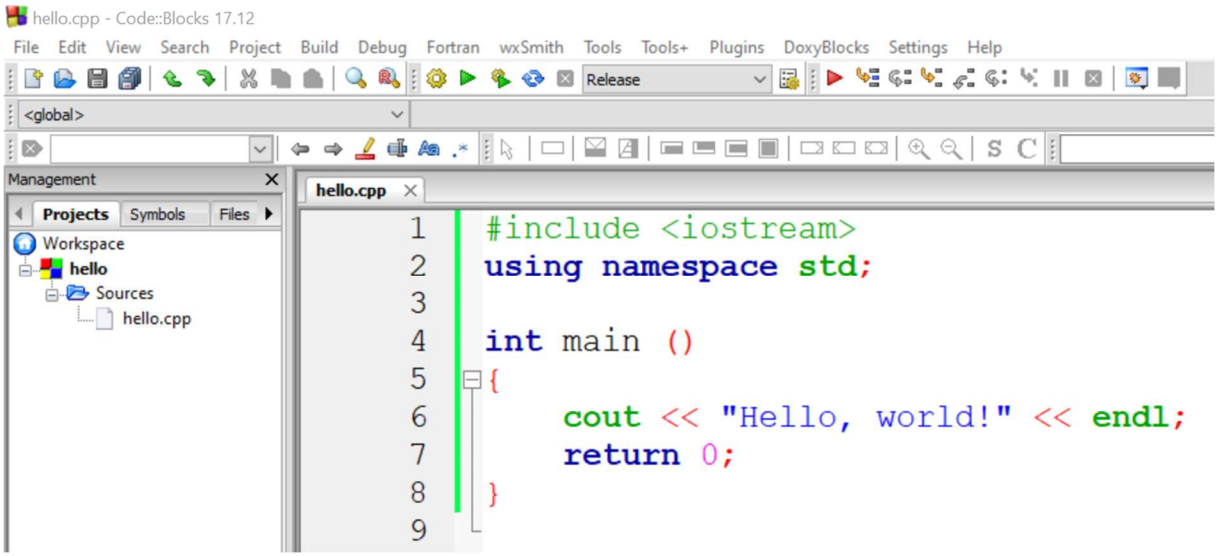
Ja programmu veido vairāki *.cpp* faili, tad izstrādes vides ietvaros jāveido t.s. projekts. Projekts ir konkrētās izstrādes vides specifisks jēdziens un nav C++ jautājums, tāpēc katrā izstrādes vidē darbs ar projektu varētu nedaudz atšķirties.

C++ programmas kompilēšana, veidojot projektu no jauna:

- izveidot jaunu projektu,
- piesaistīt visus programmas .cpp failus, kā arī neobligāti citus failus, projektam – tas jāveic ar izstrādes vides specifiskiem līdzekļiem – failus projektā parasti ir iespējams grupēt mapēs,
- ar komandu *compile* vai *build* veikt kompilēšanu, iegūstot izpildāmo moduli.

Pēc projekta izveidošanas to parasti reprezentē speciāls projekta fails, kura paplašinājums ir atkarīgs no izstrādes vides. Šajā failā (parasti teksta režīmā) ir uzskaitīti visi projektā ietilpstošie faili, kā arī dažādi projekta konfigurācijas parametri.

- C++ programmas kompilēšana izveidotam projektam:
- atvērt projekta failu ar izstrādes vides (teksta) redaktoru,
- ar komandu *compile* vai *build* veikt kompilēšanu, iegūstot izpildāmo moduli.



Att. 2.5. Projekts "hello" izstrādes vidē Code::Blocks

Projekta kompilēšanas rezultātā izveidojas vismaz šādi faili:

- katram C++ failam viens objekta fails (.obj vai .o)
- izpildāmais modulis (piemēram *.exe*).

2.3.2.3. Programmas kompilēšana komandrindas režīmā

Programmu iespējams nokompilēt arī, neizmantojot izstrādes vidi. Lai to izdarītu, ar kompilatoru tiešā veidā jāveic šādas darbības:

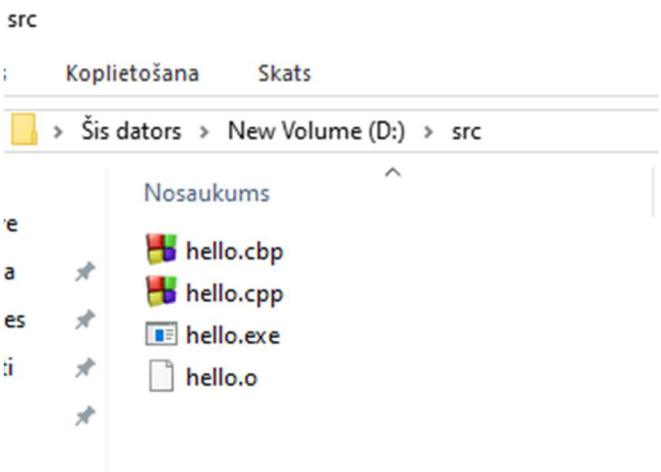
- kompilators kompilēšanas režīmā vienreiz jāizsauc katram C++ failam, lai izveidotu atbilstošo objekta failu,
- kompilators jāizsauc linkošanas režīmā, uzskaitot visus iepriekšējā solī izveidotos objekta failus un izveidojot izpildāmo moduli.

Ja programma sastāv no viena faila, tad kompilēšanu, lai iegūtu izpildāmo moduli, var izdarīt arī vienā solī.

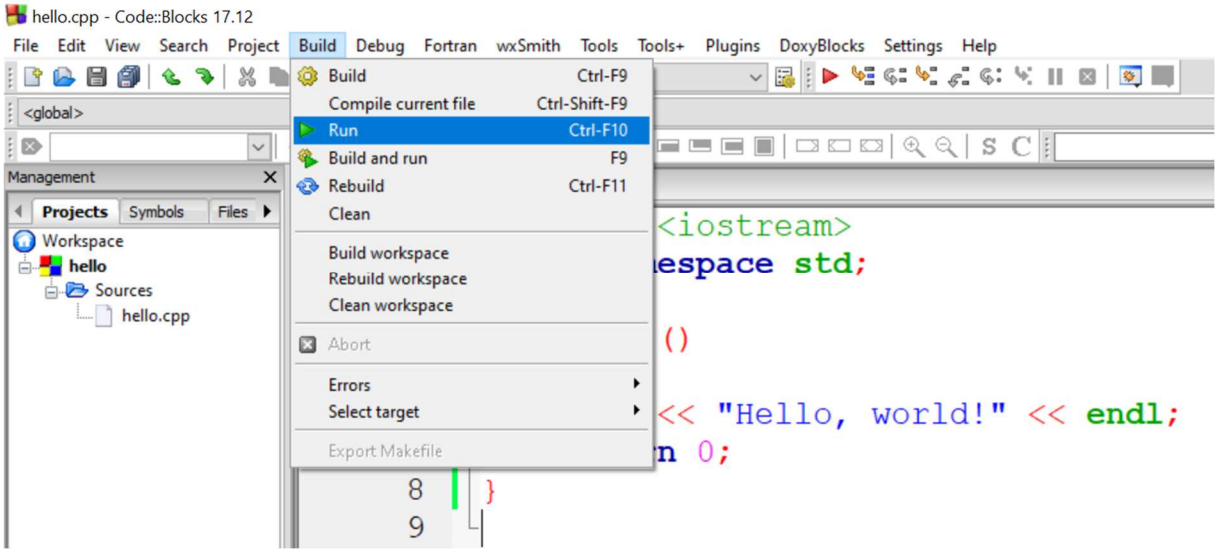
Izplatīti kompilatori dažādās platformās ir *gcc/g++*. Tos izmanto arī izstrādes vide *Code::Blocks*.

2.3.3. Programmas palaišana

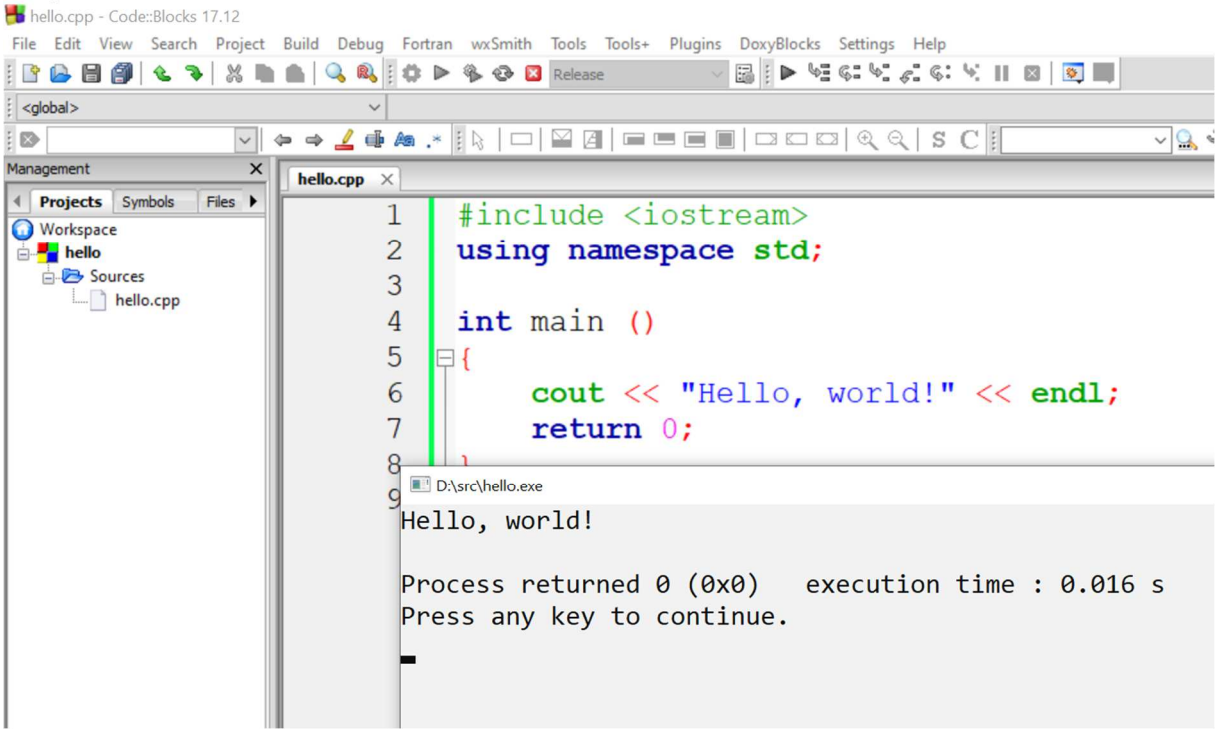
Programmu var palaist tikai tad, ja programma ir līdz galam nokompilēta un ir izveidots izpildāmais modulis:



Programmas palaišana no izstrādes vides parasti notiek ar programmu “Run” (Code::Blocks arī poga ▶):



Programma darbībā:



Tomēr, palaižot programmu, ir jāatceras, ka tiek palaists kompilētais izpildāmais modulis (nevis pa taisno programma), tāpēc, katrreiz pamainot kodu, vispirms programma ir jānokompilē un tad jāpalaiž.

Nokompilētu izpildāmo modili var palaist arī pa taisno no komandrindas:

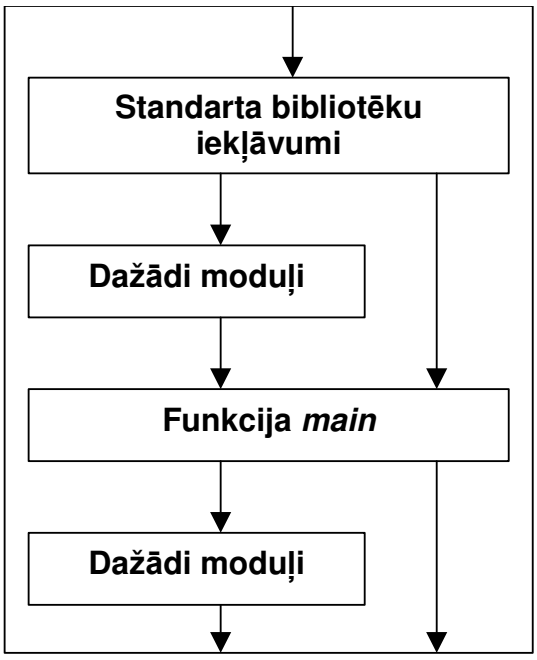
```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.18362.535]
(c) 2019 Microsoft Corporation. Visas tiesības paturētas.

d:\src>hello.exe
Hello, world!

d:\src>_
```

2.4. C++ programmas vispārējā struktūra

2.4.1. Programmas struktūras galvenā shēma

Valodas C++ programma (kā jebkurā citā programmēšanas valodā rakstīta programma) ir konstrukciju kopums, kas apraksta noteiktu algoritmu.

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. Att. 2.1). 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.



Att. 2.6. C++ programmas vienkāršota struktūra

No leksiskā viedokļa C++ programmas struktūru veido **leksēmas** (*lexemes*) jeb **marķieri** (*tokens*):

- identifikatori (*identifiers*),
- atslēgas vārdi (*keywords*),
- literāļi jeb konstantas vērtības (*literals*),
- operatoru marķieri (*operator tokens*) un
- dienesta simboli (*punctuators*).

Sintaktiskā līmenī programmu veido:

- mainīgie (*variables*),
- konstantes (*constants*), t.sk., literāļi,
- datu tipi (*data types*),
- funkcijas (*functions*),
- izteiksmes (*expressions*).

Konstrukciju līmenī varētu teikt, ka C++ programmas sastāvā ietilpst:

- priekšraksti (*statements*),
- deklaratori (*declarators*),
- bloki (*blocks*),
- (konstrukciju) galvas.

Bez tam ikviena C++ programma nav iedomājama bez **preprocesēšanas** (priekšapstrādes) **komandām**, no kurām tipiskākās ir bibliotēku iekļaušanas komandas *#include*.

2.4.2. Programmas piemērs

Vienkāršākajā gadījumā C++ programma ir funkciju kopums, kurā tieši viena saucas *main*. Tieši no funkcijas *main* sākās C++ programmas izpilde. Izņēmums ir vienīgi *Windows* specifiskas programmas, kur programmu ievadošā funkcija tiek saukta *WinMain*. Nezinot, kas ir C++ funkcija, sākumā pietiek uztvert funkciju kā figūriekavu blokā ietvertu instrukciju virkni, pirms kura ir funkcijas galva, kas cita starpā ietver arī funkcijas nosaukumu. Programma Att. 2.7 demonstrē vienkāršu C++ programmu, kas veic skaitļa ievadi, tā sareizināšanu ar citu skaitli un rezultāta izvadi.

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

int main ()
{
    double x, y, z;
    cout << "Input a numeric value:" << endl;
    cin >> x;
    y = 1.2;
    z = x * y;
    cout << x << '*' << y << '=' << z << endl;
    return 0;
}
```

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

Att. 2.7. Programmas piemērs valodā C++, kas izrēķina ievadītā skaitļa reizinājumu ar 1.2

No leksiskā viedokļa šī programma ietver šādus elementus:

- identifikatori (ieskaitot rezervētos vārdus): std, main, x, y, z, cout, endl, cin;
- atslēgas vārdi: using, namespace, int, double, return
- literāļi: [1.2] ["Input a numeric value:"] ['*'] ['='] [0]
- operatori un dienesta simboli: () { } , ; << >> = *

No sintaktiskā viedokļa programma ietver šādus elementus:

- mainīgie: x, y, z, cout, cin
- konstantes: visi literāļi un [endl]
- datu tipi: int double
- funkcijas: main
- izteiksmes: (piemēram) [x * y]

No konstrukciju viedokļa programmā ir izdalāmas šādas daļas:

- priekšraksti: (piemēram) [cin >> x;] [z = x * y;]
  [cout << "Input a numeric value:" << endl;]
- deklaratori: [using namespace std;] [double x, y, z;]
- bloki: [{...}]
- galvas [int main ()]

Bez tam programmā ir viena preprocesora komanda: bibliotēkas *iostream* iekļāvums.

2.4.3. Komentāri

Komentāri ir tādi programmas fragmenti, kas neietekmē programmas darbību (resp., izpildāmā koda veidošanu), jo kompilators tos ignorē. Komentārus programmētājs lieto, lai pierakstītu papildus informāciju pie programmas, kas, piemēram, vēlāk ļautu vieglāk lasīt programmu.

Valodā C++ ir divu veidu komentāri:

- Vienas rindas komentāri – sākas ar simbolu pāri // un beidzas līdz ar rindas beigām.
- Vairāku rindu komentāri – sākas ar simbolu pāri /* un beidzas ar simbolu pāri */.

Nākošais piemērs demonstrē abu veidu komentārus:

```
/*********************************************************
**
** Programma "Hello, World"
** Izveidota 2007.04.05.
** Labota 2007.04.09.
** Autors Mr. X
**
*********************************************************/

#include <iostream>
using namespace std;  /*
tā bija bibliotēku iekļaušana */

// galvenā funkcija
int main ()
{
    cout << "Hello, world!" << endl;
    return 0; // beidz funkcijas darbu
}
```

Att. 2.8. Komentāri C++ programmā

Šajā programmā ir četri komentāri – pa divi no katra veida:

```
/*********************************************************
**
** Programma "Hello, World"
** Izveidota 2007.04.05.
** Labota 2007.04.09.
** Autors Mr. X
**
*********************************************************/

                        /*
tā bija bibliotēku iekļaušana */

// galvenā funkcija

                // beidz funkcijas darbu
```

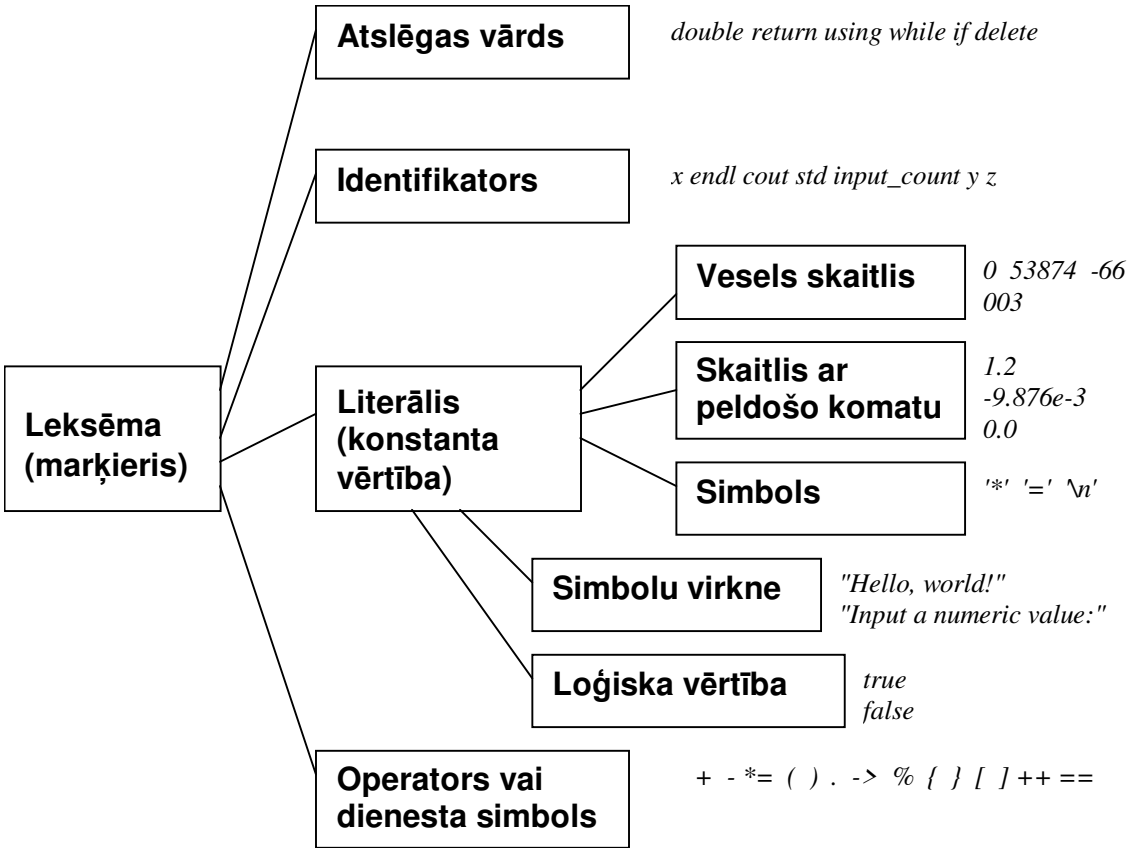
3. Valodas C++ struktūras elementi

3.1. C++ programmas leksiskā līmeņa elementi

No leksiskā jeb zemākā līmeņa pieraksta viedokļa C++ programma sastāv no leksēmām jeb marķieriem.

> **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.



Att. 3.1. C++ leksiskā līmeņa elementu klasifikācija

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, programmas teksta fragmentā, neskaitot kvadrātiekavas, [**1+2**], ir trīs leksēmas, kaut arī nav atdalītājsimbolu).

Dažu leksēmu atdalīšanai vienai no otras obligāti jālieto atdalītājsimboli.

Valodā C++ visiem atdalītājsimboliem ir vienāda nozīme (nav svarīgi, vai atdalīšanai lietot vienu vai vairākus simbolus, un – vai tukšumu vai jaunas rindiņas simbolu).

3.1.1. Identifikatori

> **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 uzskatāmi par diviem dažādiem.

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
```

Identifikatora sintaktiskais apraksts parādīts Att. 3.2.

identifier:
identifier_start *identifier_tail*$_{opt}$

identifier_start: one of
_ a b c d e f g h i j k l m
n o p q r s t u v w x y z
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z

identifier_tail:
identifier_symbol *identifier_tail*$_{opt}$

identifier_symbol:
identifier_start
digit

digit: one of
0 1 2 3 4 5 6 7 8 9

Att. 3.2. Identifikators (*identifier*)

3.1.2. ISO metavaloda – sintakses apraksta līdzeklis

Valodu sintakses aprakstam tiek plaši lietoti dažādi sintakses apraksta līdzekļi. Viens no līdzekļiem, kas pārsvarā tiks izmantots arī šajā mācību materiālā aprakstīto konstrukciju sintaktiskajam pierakstam, ir **ISO metavaloda**. Tālāk mācību materiālā tiks aprakstīts vēl viens sintaktiskā apraksta līdzeklis – sintaktiskā diagramma.

ISO metavaloda ir teksta formas apraksta metode, kas papildināta ar dažām teksta formatēšanas iespējām (šrifts, apakšraksts (*subscript*)), tādējādi izvairoties no metasimbolu lietošanas, kā tas ir, piemēram, tādai teksta formas sintaktiskā pieraksta metodei, kā Bekusa-Naura forma. Metasimbolu nelietošana formatēšanai būtiski atvieglo sintakses pierakstīšanu un lasāmību gadījumos, kad jāapraksta sintaktiskas konstrukcijas, kur metasimboli sakrīt ar terminālajiem simboliem.

Jebkura sintakses apraksta metode sastāv no divu veidu elementiem:

- **terminālie simboli**, t.i., tie, kas tiešā veidā parādās programmas tekstā, un
- **neterminālie simboli**, jeb jēdzieni, kas, izmantojot aprakstu, tiek noteiktā veidā izvesti, lai beigu beigās nonāktu līdz terminālajiem simboliem.

Bez elementiem sintaktiskajā pierakstā tiek izmantotas arī **sintaktiskā pieraksta operācijas**, kas nosaka, kādā secībā elementi tiek izkārtoti, lai atbilstu noteiktam neterminālajam simbolam. Tipiskākās operācijas ir:

- izvēle starp variantiem (*or*),
- konkatenācija (*and*),
- neobligātais statuss (*optional*),
- atkārtošanās.

Sintaktiskais pieraksts parasti ir organizēts **teikumos**, kur katrs **teikums** vai teikumu kopums **apraksta vienu neterminālo simbolu**. Ja neterminālo simbolu apraksta vairāki teikumi, tad tie būtu uzskatāmi par alternatīvām un būtu apvienojami vienā teikumā, izmantojot izvēles operāciju (*or*).

Tālāk apskatīsim ISO metavalodu detalizēti, par piemēru demonstrējot piecus apraksta teikumus Att. 3.2, kas tālāk tekstā tiks identificēti attiecīgi ar burtiem a, b, c, d, e.

- **Teikums** ISO metavalodā **sākas** ar aprakstāmā neterminālā simbola (jēdziena) nosaukumu kursīvā (*italic*), aiz kura seko kols. To demonstrē piemēri a, b, c, d, e:
  identifier:
- Attiecīgā neterminālā simbola **apraksts seko nākošajās rindās** aiz teikuma galvas ar papildus atkāpi no malas, piemēram:
  identifier_start
  digit

Apraksts sastāv no divu veidu elementiem – terminālajiem un neterminālajiem simboliem, kas atdalīti ar tukšumiem un var tikt izvietoti vairākās rindās, bez tam šiem elementiem var tikt piemērots papildus īpašību apraksts, kas dots apakšrakstā (*subscript*).

- **Neterminālie simboli** apzīmēti kursīvā parastajā šriftā, piemēram:
  identifier_start
- **Terminālie simboli** tiek pierakstīti fiksēta platuma simbolu šriftā (*Courier*), piemēram:
  `A B C D`

- Lai aprakstītu **secību** (konkatenāciju), elementi tiek rakstīti viens aiz otra vienā rindā, atdalot ar tukšumiem. To demonstrē piemērs a:

  identifier_start *identifier_tail*$_{opt}$

Šis pieraksts nozīmē, ka vispirms doto neterminālo simbolu veido *identifier_start*, pēc kura seko *identifier_tail*$_{opt}$.

- Lai aprakstītu **alternatīvās** iespējas, elementu secības tiek rakstītas dažādās rindās. To demonstrē piemērs d:

  identifier_start
  digit

Šis pieraksts nozīmē, ka ir divas alternatīvas iespējas – vai nu *identifier_start* vai *digit*.

- **Uzskaitījums** ir izvēles iespēju definēšanas vienkāršākais variants. Lai atvieglotu alternatīvo iespēju definēšanu gadījumos, kad alternatīvie varianti nav garas virknes, bet ir elementi, visticamāk termināļi, teikuma galvā tiek papildus pievienots teksts ***one of*** (ne kursīvā), kā rezultātā visi elementi apraksta daļā tiek uzskatīti par alternatīvām, kaut arī būtu izvietoti pa vairākiem vienā rindā. To demonstrē piemērs e:

  one of

  0 1 2 3 4 5 6 7 8 9

- **Neobligātu elementu** nosaka pazīme ***opt***, kas apakšrakstā (*subscript*) tiek pielikta aiz dotā elementa. To demonstrē piemērs c:

  identifier_symbol *identifier_tail*$_{opt}$

Šīs pieraksts nozīmē, ka doto neterminālo simbolu veido *identifier_symbol*, kam var sekot (bet ne obligāti) *identifier_tail*.

Pazīmes ***opt*** vietā var izmantot divu alternatīvu variantu uzskaitījumu – ar un bez neobligātā elementa. Piemēram, augstāk apskatītais *identifier_symbol* *identifier_tail*$_{opt}$ tad savādāk būtu pierakstāms šādi:

identifier_symbol
identifier_symbol *identifier_tail*

Šāds pieraksta veids dažos gadījumos ir labāk lasāms.

- **Atkārtojums** ISO meta pierakstā netiek pierakstīts ar kādu speciālu operatoru. To nodrošina rekursīvais pieraksta veids, kad teikumā aprakstītais neterminālais simbols (kura nosaukums ir teikuma galvā) parādās arī aprakstā, tādējādi veidojot rekursīvu ciklu. Šo konstrukciju demonstrē piemērs c:

  identifier_tail:

  identifier_symbol *identifier_tail*$_{opt}$

Šeit neterminālais simbols *identifier_tail* parādās gan galvā, gan apraksta daļā, tādējādi nodrošinot atkārtošanos.

3.1.3. Atslēgas vārdi

> **Atslēgas vārdi** (*keywords*) ir programmēšanas valodā iekšēji definēti vārdi.

Atslēgas vārdus apraksta identifikatori. Atslēgas vārdi apraksta iebūvētos datu tipus, dažādas konstrukcijas un citus valodas elementus. C++ pieejamie atslēgas vārdi uzskaitīti Tab. 3.1.

Tab. 3.1.

Atslēgas vārdi

asm	do	if	return	typedef
auto	double	inline	short	typeid
bool	dynamic_cast	int	signed	typename
break	else	long	sizeof	union
case	enum	mutable	static	unsigned
catch	explicit	namespace	static_cast	using
char	export	new	struct	virtual
class	extern	operator	switch	void
const	false	private	template	volatile
const_cast	float	protected	this	wchar_t
continue	for	public	throw	while
default	friend	register	true	
delete	goto	reinterpret_cast	try	

3.1.4. Literāļi

> **Literāļi** (*literals*) ir dažādu tipu konstantas vērtības, kas tiešā veidā parādās programmas tekstā.

Literāļi var būt veseli skaitļi, skaitļi ar peldošo komatu, simboli, simbolu virknes un loģiskās vērtības (sk. arī Att. 3.1).

3.1.4.1. Vesels skaitlis

Vesela skaitļa pieraksts decimālajā variantā dots Att. 3.3 (vēl C++ pieļauj pierakstu arī astotnieku un sešpadsmitnieku formā). Vesels skaitlis pierakstāms kā ciparu virkne (*unsigned_integer*), pirms kuras var atrasties plus vai mīnus zīme (*sign*).

integer:
sign$_{opt}$ *unsigned_integer*

unsigned_integer
digit
digit unsigned_integer

sign:
+
–

Att. 3.3. Vesels skaitlis (*integer*).

Korekti pierakstīti veseli skaitļi ir šādi:

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

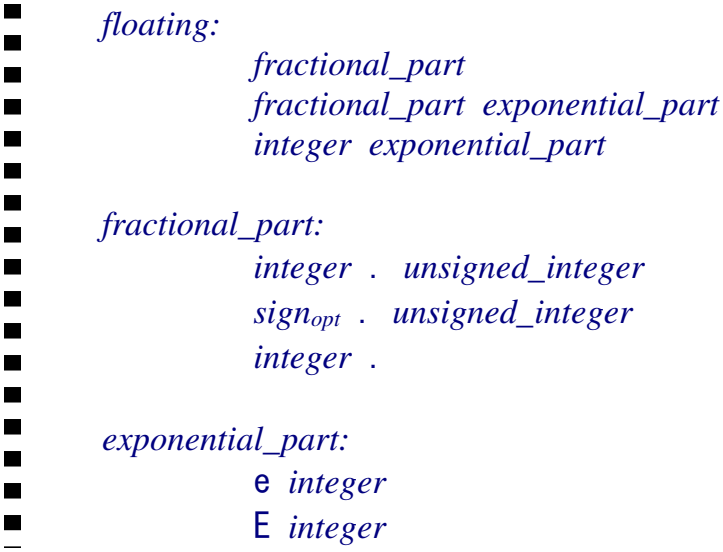
Šādas simbolu virknes neapzīmē veselus skaitļus:

```
50.0 3x
```

3.1.4.2. Skaitlis ar peldošo komatu

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

Vienu no divām – veselo daļu vai mantisu var izlaist, tādējādi ‘0.3’ var pierakstīt kā ‘.3’ , bet ‘4.0’ kā ‘4.’.



Att. 3.4. Skaitlis ar peldošo komatu (*floating*)

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
```

3.1.4.3. Simbola literālis

Simbola literālis 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*), kas kalpo par atsoļa (*escape*) simbolu:

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

character_literal:
    ' *c_char* '
    ' *simple_escape_sequence* '

c_char:
    any member of the source character set except
        the single-quote ', backslash \, or new-line character
    simple_escape_sequence

simple_escape_sequence: one of
    ¥' ¥" ¥? ¥¥ ¥a ¥b ¥f ¥n ¥r ¥t ¥v

Att. 3.5. Simbola literālis (*character_literal*)

3.1.4.4. Simbolu virknes literālis

Simbolu virknes literālis (*string literal*) 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 simbola literāļa gadījumā:

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

string_literal:
    " $s\_char\_sequence_{opt}$ "

s_char_sequence:
    s_char
    s_char s_char_sequence

s_char:
    any member of the source character set except
        the double-quote ", backslash \, or new-line character
    simple_escape_sequence

Att. 3.6. Simbola literālis (*string_literal*)

3.1.4.5. Loģiskās vērtības literālis

Loģiskās vērtības literālis (*Boolean literal*) apzīmē loģisku vērtību ar tipu *bool*.

boolean_literal:
    true
    false

Att. 3.7. Loģiskās vērtības literālis (*boolean_literal*)

3.2. Daži C++ programmas sintaktiskā līmeņa elementi

Galvenās valodas C++ sintaktiskās pamatvienības ir mainīgie, konstantes, datu tipi, kā arī funkcijas un izteiksmes.

3.2.1. Mainīgie

> **Mainīgais** (*variable*) ir vieta datora atmiņā, kur tiek glabāti noteikta tipa dati.

Mainīgā jēdziens būtu aplūkojams arī nedaudz citā aspektā:

> **Mainīgais** (*variable*) ir mehānisms, kas reprezentē noteiktu vērtību, kā arī nosaka darbības, kas var tikt veiktas ar šo vērtību.

Mainīgo programmā identificē noteikts vārds – identifikators. Pirms sākt strādāt ar mainīgo, tas ir jāinicializē. Nevajadzētu izmantot automātisku mainīgo inicializāciju, ko veic dažas izstrādes vides.

> **Inicializācija** (*initialization*) ir mainīgā pirmreizējā aizpildīšana ar vērtību.

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 (*pointer variables*).

No atmiņas izmantošanas viedokļa mainīgos iedala šādās grupās:

- parastie (automātiskie) (*automatic 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.

3.2.2. Konstantes

> **Konstante** (*constant*) ir vērtība, kas nemainās vai nu visas programmas, vai arī kāda tās moduļa darbības laikā.

Programmā konstantes var reprezentēt divu veidu mehānismi:

- literāļi,
- konstantes-mainīgie – speciāli definēti mainīgie, kuru saturu pēc inicializācijas nedrīkst mainīt.

3.2.3. 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 salīdzinoši 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 $(x+y)+z$).

Tab. 3.2.

Galvenās C++ operatoru grupas

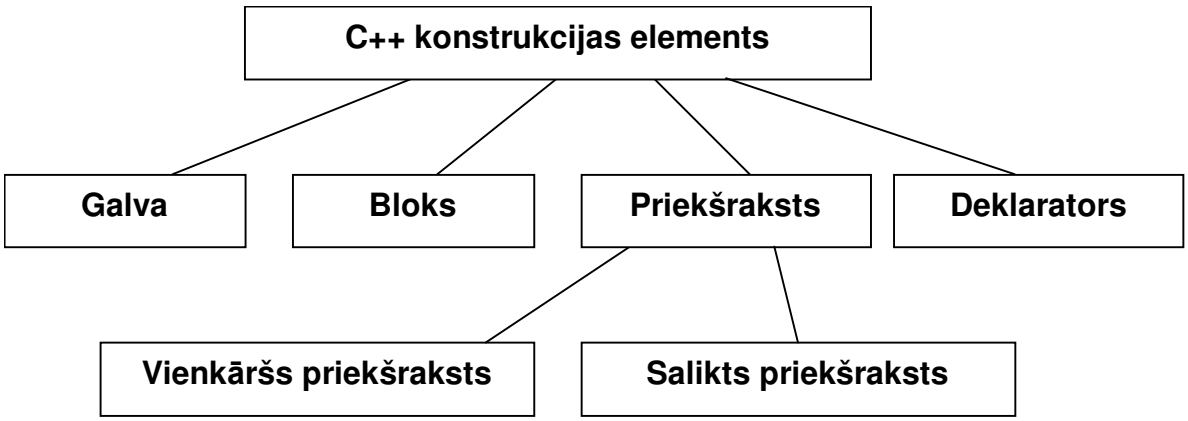
aritmētiskās darbības	`+ - * / %`
loģiskās operācijas	`&& || !`
piešķiršanas operatori	`= += -= *= /= %= ++ --`
salīdzināšanas operācijas	`== != < <= > >=`
struktūras elementa izvēle	`. ->`
masīva indeksācija	`[]`
funkcijas izsaukums	`()`
atmiņas vadības operatori	`new delete delete[]`
adreses iegūšanas 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.

3.3. C++ programmas konstrukcijas līmeņa elementi

Viena no programmēšanas valodas (t.sk. C++) svarīgākajām konstrukcijām ir priekšraksts. Varētu teikt, ka no konstrukciju viedokļa programma sastāv pamatā no **priekšrakstiem**.

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



Att. 3.8. Valodas C++ konstruktīvo elementu klasifikācija

Deklarators pēc izskata ir ļoti līdzīgs priekšrakstam, un daudzos gadījumos deklaratoru no konstruktīvā viedokļa var uzskatīt par priekšrakstu.

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

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

> **Bloks** (*block*) ir priekšrakstu vai deklaratoru 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.

Salikti priekšraksti (*compound statements*) ir tādi priekšraksti, kuru sastāvā ietilpst citi priekšraksti. **Vienkāršu priekšrakstu** (*simple statements*) sastāvā nevar ietilpt citi priekšraksti. 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.

4. C++ pamati

4.1. Vērtības un datu tipi

Programma kādā programmēšanas valodā realizē vienu vai vairākus algoritmus, kas ir darbību vai instrukciju virkne. Tomēr šīs darbības darbojas ar datiem, un programmas darba rezultāts ir tieši dati, respektīvi, kaut kādas vērtības. Parasti programmēšanas valodās dati nepastāv kā vērtības vien. Neatņemama vērtības sastāvdaļa ir datu tips. Piemēri datu tipiem ir vesels skaitlis, skaitlis ar peldošo komatu, teksts.

> **Datu tips** (*data type*) (no vispārēja izmantošanas viedokļa) ir datu veids.

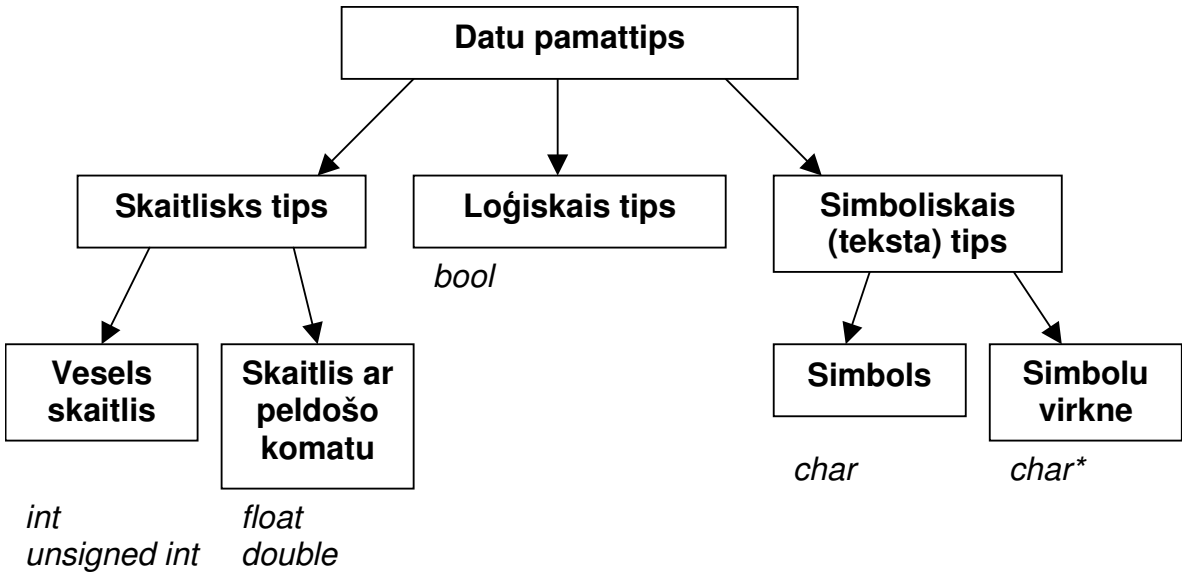
C++ programmā nav iespējams strādāt ar vērtību, ja nav zināms (nav noteikts) tās datu tips. Tam ir praktiski iemesli, ko lielā mērā izskaidro nākošā datu tipa definīcija.

> **Datu tips** (*data type*) (no datu apstrādes viedokļa) ir darbību kopums, ko var veikt ar noteiktu vērtību.

Tādējādi it kā viena un tā pati darbība ar dažādu datu tipu vērtībām var tikt veikta dažādi.

Tipiskākais piemērs valodā C++ tam ir dalīšana. Ir **parastā dalīšana** (piemēram, 16.0/5=3.2) un **veselo skaitļu dalīšana** (16/5=3, atlikumā 1), kas ir pilnīgi cita operācija, tomēr abas šīs dalīšanas operācijas tiek apzīmētas ar ‘/’, un to, kura no tām izpildīsies, nosaka tikai un vienīgi operandu tipi, uz ko attiecas dalīšana.

Valodā C++ ir vairāki datu pamattipi, kuru klasifikācija parādīta Att. 4.1.



Att. 4.1. Valodas C++ datu pamattipi

No datu pamattipiem var būvēt saliktus datu tipus – masīvus, struktūras, klases. Valodā C++ no realizācijas viedokļa datu tipu atšķirības nav tik ļoti izteiktas kā tas ir citās programmēšanas valodās. Piemēram, vesels skaitlis, loģiskais tips un simbols (*char*) pēc būtības ir ļoti līdzīgi.

Īpašs tips ir simbolu virkne *char**, kas, kaut arī pēc būtības ir salikts tips (*char* vērtību virkne), tomēr izmantošanas biežuma un svarīguma dēļ šeit ir pieskaitīts pie pamata tipiem.

Katru datu tipu raksturo

- atmiņas daudzums (veselos baitos), ko aizņem viena šāda tipa vērtība,
- vērtību diapazons, kas atbilst šim datu tipam,
- ar šo datu tipu saistītās darbības.

Nākošajā tabulā parādīti galvenie C++ pamattipi un dots īss to raksturojums.

Tab. 4.1.

C++ pamattipi

Datu tips	Izmērs	Vērtību diapazons	Piemēri
int	4B (32 bitu mašīnām parasti); atkarīgs no platformas	-2147483648.. 2147483647	`-8 4567 +20478 0`
short int	2B	-32768..32767	
long int	4B	-2147483648.. 2147483647	
long long int	8B	$-2^{64}..2^{64}-1$	
unsigned int	4B	0..65535	`4567 +20478 0`
float	4B	$\sim10^{-38}..10^{38}$ 7 zīmīgie cipari	`1.286 -1.2e+9`
double	8B	$\sim10^{-308}..10^{308}$ 15 zīmīgie cipari	
long double	10B (vismaz)	$\sim10^{-4932}..10^{4932}$ 19 zīmīgie cipari	
char	1B	-128..127 (simbolu kodi) (parasti); dažās kompilatoru realizācijās 0..255	`'A' '?'`
unsigned char	1B	0..255 (simbolu kodi)	
bool	1B	true (1), false (0)	`true false`
char*	mainīgs		`"abcdef" ""`
void	"tukšais" tips – tips ar speciālu nozīmi		

Vēl ir pieejami arī citi līdzīgi tipi, kā *signed int* (alternatīvs *int*), *signed char*, *unsigned long int*.

Bez tam vairākiem no tipiem var lietot to saīsinātos nosaukumus: *unsigned* (= *unsigned int*), *short* (= *short int*), *long* (= *long int*).

Kā jau redzams no konteksta, *signed* nozīmē "ar zīmi", t.i. gan negatīvās, gan pozitīvās vērtības, bet *unsigned* – "bez zīmes", t.i., tikai 0 un pozitīvās vērtības, kas dod iespēju saglabāt divas reizes vairāk pozitīvo vērtību.

Galvenie veidi, kā piekārtot vērtībai datu tipu:

- ja vērtība tiek glabāta mainīgajā, kuram pie deklarēšanas ir noteikts datu tips, tad vērtībai tiek automātiski piekārtots mainīgā datu tips, šis ir visizplatītākais veids, kā vērtībai tiek piekārtots datu tips;
- ja vērtība netiek glabāta mainīgajā (tā aprakstīta, izmantojot kādu no literāļiem), tad tai tiek piekārtots datu tips atkarībā no šīs vērtības pieraksta, pielietojot kādu no noklusētajiem datu tipiem (Tab. 4.2);
- ja vērtība atrodas dinamiskās atmiņas apgabalā, tās datu tips tiek noteikts, to uzrādot pie operatora *new*, rezervējot atmiņas apgabalu.

Tab. 4.2.

C++ noklusētie datu tipi. Kāds no šiem tipiem automātiski tiek piekārtots programmā sastopamajiem literāļiem

Literāļa tips	Datu tips	Piemēri
vesels skaitlis	int	`-8 4567 +20478 0`
skaitlis ar peldošo komatu	double	`1.286 -1.2e+9`
simbols	char	`'A' '?'`
simbolu virkne	char*	`"abcdef" ""`

Iepriekšējā tabula norāda, ka, piemēram, ja programmas tekstā sastopams skaitļa pieraksts 99, tad tam automātiski tiek piekārtots tips *int*. Lai panāktu, ka tas tiek interpretēts par *double*, tad būtu jāraksta, piemēram, 99.0.

Datu tipa *char* vērtība ir viens simbols, kas atmiņā aizņem vienu baitu. Praktiski atmiņā glabājas šī simbola kods un, atšķirībā no dažām citām programmēšanas valodām, nav vajadzīga nekāda speciāla pārveidošana (piemēram, funkcijas izsaukšana), lai iegūtu šī koda vērtību, pietiek ar parastu tipa pārveidošanu.

4.2. Mainīgo deklarēšana

Pirms mainīgo var izmantot programmā, tas ir jādeklarē.

> **Deklarators** (*declarator*) ir konstrukcija, kas paziņo, ka noteikts programmas elements (visbiežāk mainīgais) tiks izmantots programmā.

Mainīgā deklarācija ietver šādas informācijas uzrādīšanu par mainīgo:

- mainīgā vārds,
- mainīgā tips,
- sākuma vērtība (neobligāti).

Mainīgā deklarēšanu veic deklarators. Ar vienu šādu konstrukciju var deklarēt vienu vai vairākus viena tipa mainīgos. Mainīgo deklarēšanu var veikt jebkurā programmas vietā, galvenais, lai tas notiktu pirms pirmās šo mainīgo izmantošanas.

```
int x;
int x1, x2;
double d;
double e=2007;
int m=7, n=5;
int p=-1, q, r=89;
string s = "Hallo, World!";
char v, w='X';
```

Att. 4.2. Vienkāršas mainīgo deklarēšanas piemēri

Atbilstoši aprakstam un piemēriem redzams, ka mainīgos var deklarēt ar vai bez inicializēšanas deklarācijas laikā. Mainīgo deklarācija sintaktiski aprakstīta divos alternatīvos variantos Att. 4.3 un Att. 4.4.

variable_declarator:
data_type var_decl_unit_list ;

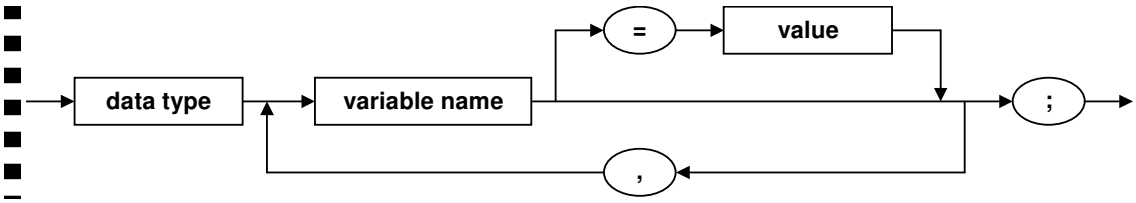
var_decl_unit_list:
var_decl_unit
var_decl_unit , var_decl_unit_list

var_decl_unit:
var_name
var_name = value

Att. 4.3. Mainīgo deklarācija (*variable_declarator*)

4.3. Sintaktiskā diagramma – sintakses apraksta līdzeklis

Iepriekš tika aprakstīta un šī mācību materiāla aprakstā lietota ISO metavaloda valodas konstrukciju pierakstam. Atšķirībā no ISO metavalodas, **sintaktiskā diagramma** nodrošina grafisku pieraksta formu. Vizuāli sintaktiskā diagramma atgādina blokshēmu, tomēr pēc būtības atšķirība ir liela, jo blokshēma domāta algoritma pierakstam, bet sintaktiskā diagramma – valodas sintakses aprakstam.



Att. 4.4. Mainīgo deklarācija sintaktiskās diagrammas formā (*variable_declarator*)

Sintaktiskās diagrammas lasīšanas un darbības secību nosaka **bultiņas** – izpilde sākas uzreiz aiz ieejošās bultiņas un beidzas izejošajā.

Bultiņas sintaktiskajā diagrammā var savienot elementus, zaroties un atkal saplūst kopā, un tas viss nosaka **secību** (konkatenāciju), **alternatīvās iespējas** un **atkārtošanos**.

Terminālie simboli ievietoti ovālas formas rāmītī, bet **neterminālie simboli** – taisnstūra formas kastītēs.

Salīdzinot ar ISO metavalodu, sintaktiskās diagrammas parasti labāk lasāmas un uztveramas, tomēr to izveidošanai vajag vairāk laika, nepieciešami speciāli rīki, turklāt ISO metavalodā pierakstītas konstrukcijas ir tuvāk tam, kā tās ir realizētas kompilatorā.

Turpmāk lielākā daļa konstrukciju tiks aprakstītas ISO metavalodā, tomēr atsevišķas, lai nodrošinātu labāku sapratni, būs pieejamas arī sintaktisko diagrammu formā.

4.4. Piešķiršana un tipa pārveidošana

4.4.1. Piešķiršanas priekšraksts

> **Piešķiršana** (*assignment*) ir vērtības ierakstīšana mainīgajā.

Piešķiršanu veic **piešķiršanas priekšraksts** (*assignment statement*), kas ietver sevī piešķiršanas operatoru (vienlīdzības zīmi).

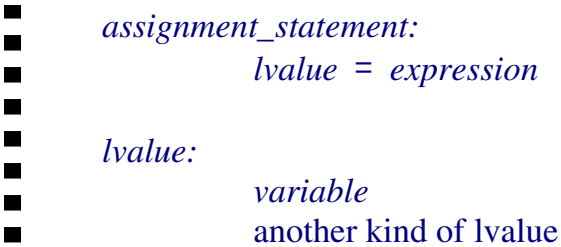
Kreisajā pusē no tā obligāti jābūt mainīgajam, vispārīgi runājot t.s. ***lvalue*** (jeb "kreisās puses") izteiksmei. *Lvalue* (nosaukums cēlies no *"left value"*) izteiksme ir tāda konstrukcija, kas nodrošina pieeju atmiņas apgabalam vai objektam, uz kuru attiecas piešķiršana. Kā jau iepriekš minēts, tipiskākais un izplatītākais *lvalue* piemērs ir mainīgais.

Labajā puse var būt jebkura konstrukcija, kas reprezentē vērtību (piemēram, literālis, mainīgais, funkcijas izsaukums vai izteiksme).

```
x = 5;
x = y = z;
x = (y + z) * 5;
arr[x] = sqrt(y);
```

Att. 4.5. Piešķiršanas priekšraksta piemēri

Kā redzams piemērā, piešķiršanas operatori ir izmantojami viens pēc otra kaskādes veidā, piemēram, (y=z; x=y) vietā var rakstīt x=y=z.



Att. 4.6. Piešķiršanas priekšraksts (*assignment_statement*)

4.4.2. Tipa pārveidošana

Parasti mainīgajam ir tāds pats tips kā tam piešķiramajai vērtībai. Tomēr dažreiz mainīgā tips atšķiras no tam piešķiramās vērtības tipa:

```
int x = 5;
double y = x;
```

Tādā gadījumā runā par tipa pārveidošanu.

> **Tipa pārveidošana** (*type casting*) ir process datu pārveidošanai no viena tipa otrā, saglabājot to pašu vai gandrīz to pašu vērtību.

Zināma informācijas zaudēšana vai vērtības pārveidošana notiek datu tipu dažādo vērtību diapazonu vai saglabājamās precizitātes dēļ. Datu tipa pārveidošana nepieciešama tādēļ, lai dotu iespēju papildus veikt noteiktas darbības ar datiem, jo datu tips nosaka arī iespējamās darbības, ko var veikt ar vērtību (sk. datu tipa definīciju no datu apstrādes viedokļa).

Daudzos gadījumos, kad tipa pārveidošanas rezultātā iespējama potenciāla informācijas zaudēšana, kompilators par to brīdina, pieprasot tipu pārveidošanas operatora pielietošanu.

Vecā stila tipa pārveidošana, izmantojot speciālo tipa pārveidošanas operatoru (kas pieejams arī valodā C), sintaktiski parādīta Att. 4.7. Tā tiek veikta vienkārši, pirms pārveidojamās vērtības priekšā pieliekot datu tipu, kā arī papildus pielietojot iekavas.

Tipa pārveidošanu, izmantojot tipa pārveidošanas operatoru, sauc par **tiešo tipa pārveidošanu** (*explicit conversion*), bet, ja operators nav nepieciešams – par **automātisko** jeb **netiešo tipa pārveidošanu** (*implicit conversion*).

Nākošajā piemērā uz ekrāna tiek izdrukāti burti ‘A’ un ‘B’, kuru atbilstošie kodi ir 65 un 66. Abos gadījumos mainīgajiem tiek piešķirtas skaitliskas vērtības, bet uz ekrāna izdrukāti simboli, nevis skaitļi. Pirmajā gadījumā, piešķirot mainīgajam c vērtību 65, notiek netiešā pārveidošana, bet otrā gadījumā – izdrukājot simbolu ‘B’, kad dots tā kods mainīgajā i – notiek netiešā pārveidošana.

```
int i=66;
char c;
c = 65;
cout << c << endl;
cout << (char)i << endl;
```

```
A
B
```

type_casting_old:
(*type_name*) *expression*

Att. 4.7. Vecā stila tipa pārveidošana (*type_casting_old*)

Valodā C++ ir pieejami papildus operatori tipa pārveidošanai, kas atšķirībā no klasiskā varianta papildus nodrošina dažādas pārbaudes. Att. 4.8 parāda vispārīgo tipa pārveidošanas sintaksi.

```
type_casting:
        type_casting_old
        type_casting_operator < type_name > ( object )

type_casting_operator:
        const_cast
        dynamic_cast
        reinterpret_cast
        static_cast
```

Att. 4.8. Tipa pārveidošana (*type_casting*)

Tipa pārveidošanas operators *const_cast* domāts, lai pārdefinētu modifikatorus *const* vai *volatile*, bet citādi tips paliktu tāds pats.

Operators *dynamic_cast* pārbauda, vai ir iespējama norādītā tipa pārveidošana.

Operators *reinterpret_cast* domāts viena tipa pārveidošanai par pilnīgi citu un būtu uzskatāms par vistuvāko klasiskajam variantam. Parasti šo operatoru lieto, lai pārveidotu savstarpēji nesaderīgus norādes tipus.

Korekta atbilstošā operatora izvēle ir specifisks jautājums un netiek apskatīts šī materiāla ietvaros.

Nākošajā piemērā parādīti divi alternatīvi varianti mainīgā tipa pārveidošanai uz *char*.

```
(char)i
reinterpret_cast<char> (i)
```

Lūk, daži tipiski gadījumi, kad tiek lietota tipa pārveidošana:

- Pārveidošana starp vesela skaitļa un skaitļa ar peldošo komatu tipu, lai nodrošinātu atbilstošu darbību izpildi.
- Norāžu tipa pārveidošana starp mantotas klases un bāzes klases norādes tipu.
- Norādes tipa pārveidošana uz tipiem *char** vai *void** specifisku funkciju izmantošanai.

Visi šie gadījumi tiks pieminēti tālāk mācību materiālā.

4.5. Izteiksmes

> **Izteiksme** (*expression*) ir konstrukcija, kas sastāv no operatoriem (jeb operācijām) un operandiem (parasti vērtībām), kas nosaka rēķināšanas procesu.

Parasti rēķināšanas procesa rezultāts ir noteikta tipa vērtība.

Vienkāršākais izteiksmju variants ir aritmētiskas izteiksmes, kuras atšķirībā no matemātikas pieraksta lineāri, izpildes secību ietekmējot ar papildus iekavām.

Piemēram, aritmētisku izteiksmi

$$\frac{\sin(x)+1}{y}+\frac{y}{x+1.5}$$

valodā C++ pieraksta:

```
(sin(x) + 1) / y + y / (x + 1.5)
```

Vispārējā programmēšanas teorijā parasti tiek izšķirtas aritmētiskas izteiksmes un loģiskas izteiksmes, tomēr valodā C++ izteiksme vispārīgā gadījumā ne tikai apvieno abus šos izteiksmju veidus, bet arī saturīgi krietni pārsniedz šo abu izteiksmju veidu apvienojumu. Neskatoties uz to, šajā mācību materiālā, lai papildus nodrošinātu lasītāju izpratni, gan aritmētiskās, gan loģiskās izteiksmes tiks apskatītas atsevišķi, lai uzskatāmāk parādītu to tipisko pielietojumu dažādās konstrukcijās.

Parasti izteiksmes lieto, lai izrēķinātu noteiktu vērtību, bet vispārīgā gadījumā izteiksme var atgriezt t.s. *lvalue* vērtību, kas nozīmē, ka tā var atrasties, piemēram, piešķiršanas operatora kreisajā pusē.

4.5.1. Aritmētiskas izteiksmes

> **Aritmētiska izteiksme** (*numeric expression*) ir konstrukcija, kas sastāv no skaitliskām operācijām un skaitliskiem operandiem, un kura pēc šo operāciju izpildes izrēķina skaitlisku vērtību.

Skaitliskas izteiksmes piemērs jau parādīts iepriekšējā nodaļā. Nākošais sintakses bloks apraksta aritmētisku izteiksmi ļoti vienkāršotā variantā.

```
numeric_expression:
        numeric_literal
        numeric_variable
        numeric_function_call
        numeric_expression  numeric_operator  numeric_expression
        ( numeric_expression )

numeric_literal:
        integer
        floating

numeric_operator:  one of
        + - * / %
```

Att. 4.9. Vienkāršota aritmētiska izteiksme (*numeric_expression*)

Aritmētisku izteiksmju "motors" ir aritmētiskas operācijas un funkcijas.

Galvenās C++ skaitliskās operācijas (*numeric operators*) parādītas Att. 4.9.

Svarīgi atcerēties, ka divas dažādas dalīšanas – veselo skaitļu dalīšana un parastā dalīšana tiek apzīmētas ar vienu un to pašu operāciju /. To, kura no darbībām tika izpildīta, nosaka konteksts – ja abi operandi ir veseli skaitļi, tiek izpildīta veselo skaitļu dalīšana, bet ja kaut viens tāds nav – tad parastā dalīšana.

Dalījuma atlikuma operācija izpildāma tikai veseliem skaitļiem.

Pārējās darbības (+ - *) attiecas un vienādi darbojas gan ar veseliem skaitļiem, gan skaitļiem ar peldošo komatu, atšķirības ir tikai vērtības tipā, kas tiek atgriezts pēc dotās operācijas izpildes – ja abi operandi ir veseli skaitļi, atgrieztās vērtības tips arī ir vesels skaitlis, bet, ja kaut viens ir skaitlis ar peldošo komatu, tad arī atgriežamā vērtība ir skaitlis ar peldošo komatu.

```
int a=13, b=5;
double aa=a;
a / b             // 2       veselo skaitļu dalīšana
aa / b            // 2.6     parastā dalīšana
a / 5.0           // 2.6     parastā dalīšana
a / (double)b     // 2.6     parastā dalīšana
a % b             // 3       dalījuma atlikums
a % (-b)          // 3       dalījuma atlikums
(-a) % b          // -3       dalījuma atlikums
(-a) % (-b)       // -3       dalījuma atlikums
```

Blakus aritmētiskām operācijām valodā C++ ir pieejamas bitu līmeņa (*bitwise*) operācijas (<<, >>, |, &, ^, ~), kas dod iespējas veikt dažādas manipulācijas ar veselu skaitļu vērtībām. Bitu līmeņa operācijas ir dažkārt ērtākas tādēļ, ka mazākā adresējamā vienība programmā ir baits, bet reizēm ir nepieciešama bitu līmeņa apstrāde (piemēram, taupot atmiņas vietu). Bitu līmeņa operāciju darbības princips ir labi saprotams, ja iztēlojas skaitļa attēlojumu atmiņā pa bitam. Bitu līmeņa operācijas sīkāk šajā materiālā netiks aprakstītas.

4.5.2. Skaitliskās funkcijas

Tālāk aprakstītas dažas skaitliskas funkcijas, kas pamatā atrodamas bibliotēkā *<math.h>*. Funkcijām, kas aprakstītas ar *float* tipu, parasti pieejamas arī *double* un *long double* versijas.

pow

```
float pow (float base, float p);
```

Funkcija *pow()* atgriež vērtību, ko iegūst, argumentu *base* kāpinot pakāpē *p*.

ceil

```
float ceil (float num);
```

Funkcija *ceil()* atgriež mazāko veselo skaitli, kas lielāks vai vienāds par argumentu *num* (noapaļošana uz augšu). Jāievēro, ka atgriežamā vērtība ir ar tipu "ar peldošo komatu", tāpēc, lai noapaļoto vērtību piešķirtu mainīgajam ar vesela skaitļa tipu, papildus jāveic tipa pārveidošana, piemēram, *a = (int)ceil (1.85)*.

floor

```
float floor (float num);
```

Funkcija *floor()* atgriež lielāko veselo skaitli, kas mazāks vai vienāds par argumentu *num* (noapaļošana uz apakšu). Papildus īpašības sk. pie funkcijas *ceil* apraksta.

round

```
float round (float num);
```

Funkcija *round()* atgriež noapaļotu argumenta *num* vērtību. Papildus īpašības sk. pie funkcijas *ceil* apraksta.

sin, cos, tan, asin, acos, atan, sinh, cosh, tanh

```
float sin (float arg);
```

Funkcijas *sin()* u.c. realizē attiecīgās trigonometriskās funkcijas vai to hiperboliskos variantus. Argumenta vērtība tiek mērīta radiānos.

exp

```
float exp (float arg);
```

Funkcija *exp()* atgriež argumenta *arg* eksponenti (skaitļa *e* kāpinājumu pakāpē *arg*).

log

```
float log (float arg);
```

Funkcija *log()* atgriež argumenta *arg* naturālo logaritmu.

log10

```
float log10 (float arg);
```

Funkcija *log10()* atgriež argumenta *arg* logaritmu pie bāzes 10.

sqrt

```
float sqrt (float arg);
```

Funkcija *sqrt()* atgriež argumenta *arg* kvadrātsakni.

fabs

```
float fabs (float arg);
```

Funkcija *fabs()* atgriež argumenta *arg* absolūto vērtību.

```
pow(4, 0.5)       // 2       kāpināšana 4 pakāpē 0.5
ceil(4.5)         // 5       noapaļošana uz augšu
ceil(-4.5)        // -4      noapaļošana uz augšu
floor(4.5)        // 4       noapaļošana uz apakšu
floor(-4.5)       // -5      noapaļošana uz apakšu
round(4.5)        // 5       noapaļošana matemātiski
round(-4.5)       // -5      noapaļošana matemātiski
fabs(4.5)         // 4.5     absolūtā vērtība (bez zīmes)
fabs(-4.5)        // 4.5     absolūtā vērtība (bez zīmes)
```

4.5.3. Piešķiršanas operatori ar izrēķināšanu

Valodā C++ pieejami un ir ļoti populāri izmantošanā specifiski piešķiršanas operatori, kas reizē veic arī noteiktu aritmētisku darbību, piemēram, *x=x+1* vietā var rakstīt *x+=1* vai vēl īsāk: *x++*.

Speciālie piešķiršanas operatori attiecas gan uz aritmētiskām operācijām, gan bitu līmeņa operācijām.

assignment_statement_extended:
assignment_statement
++ lvalue
lvalue ++
-- lvalue
lvalue --
lvalue special_assignment_operator expression

special_assignment_operator: one of
`+=  -=  *=  /=  %=`
`<<= >>= ^=  &=  |=`

Att. 4.10. Piešķiršanas priekšraksts paplašināti (*assignment_statement_extended*)

Speciālo piešķiršanas operatoru izmantošanas piemēri:

```
i++       //  i = i + 1
++i       //  i = i + 1
i--       //  i = i - 1
--i       //  i = i - 1
i+=k      //  i = i + k
i-=k      //  i = i - k
i*=k      //  i = i * k
i/=k      //  i = i / k
i%=k      //  i = i % k
```

5. Standarta ievade un izvade

Lai programma darbotos, tai vispirms ir jāpadod noteikti dati, bet programmas darbības beigās mums būtu jāsaņem rezultāts. Ar to nodarbojas ievade un izvade. Iepriekš apskatītajos programmu piemēros jau bija redzami izvades un ievades operatori << un >>.

Standarta ievade (no klaviatūras) un izvade (uz ekrāna) notiek, izmantojot attiecīgi objektus *cin* un *cout*, kas atrodas bibliotēkā *<iostream>*. No klaviatūras un uz ekrānu ievade/izvade parasti notiek **formatēti** (*formatted input/output*), izmantojot operatorus '>>' un '<<'. Par formatētu ievadi/izvadi sauc tādēļ, ka tā nodrošina ievadi/izvadi dažādā formātā neatkarīgi no šo datu glabāšanas formāta atmiņā. Tādējādi formatēšana (*formatting*) nozīmē datu pārveidošanu starp teksta formātu (klaviatūra vai ekrāns) un datu glabāšanas formātu atmiņā.

5.1. Formatēta izvade

5.1.1. Darbības princips

Formatēta izvade (uz ekrāna) notiek, izmantojot objektu *cout* un izvades operatoru <<. Izmantojamās bibliotēkas: *<iostream>* un (dažreiz) *<iomanip>*.

Viens izvades priekšraksts sastāv no objekta *cout*, kam seko viens vai vairāki pārīši, kur katrā pārīti ir izdrukas operators << un pēc tā vai nu izdrukājamā vērtība, vai formatēšanas manipulators.

Lai uz ekrāna izdrukātu skaitli 5, jāraksta

```
cout << 5;
```

Vairākas vērtības iespējams izdrukāt, pielietojot izdrukas operatoru << kaskādes veidā:

```
cout << "Five = " << 5;
```

Tas ir tas pats, kas veicot divas šādas darbības:

```
cout << "Five = ";
cout << 5;
```

Komanda *endl* reprezentē pāreju jaunā rindā:

```
cout << 5 << 6;
cout << "Hallo," << endl << "World!" << endl;
cout << "The End";
```

```
56Hallo,
World!
The End
```

Komanda *endl* ir identiska simbolam '\n', ko demonstrē nākošās divas rindiņas, kuras darbojas vienādi:

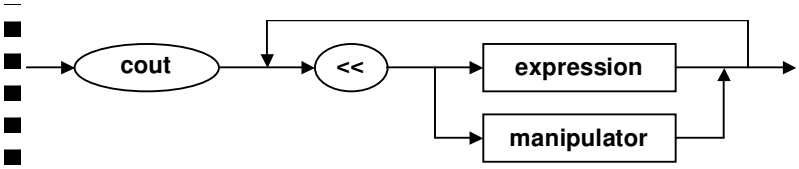
```
cout << "Hallo, World!" << endl;
cout << "Hallo, World!\n";
```

```
Hallo, World!
Hallo, World!
```

Izdrukas priekšraksta sintakse parādīta Att. 5.1 un Att. 5.2.

```
formatted_output:
        cout output_sequence

output_sequence:
        output_element
        output_element  output_sequence

output_element:
        << expression
        << manipulator
```

Att. 5.1. Formatēta izvade (*formatted_output*)



Att. 5.2. Formatēta izvade sintaktiskās diagrammas pierakstā

5.1.2. Izdrukājamās informācijas formatēšana

Izmantojot izdrukas objektu *cout* un izdrukas operatoru <<, vērtības var ne tikai vienkārši izdrukāt, bet arī uzstādīt noteiktas formatējuma īpašības. Formatēšana var tikt veikta ar divu mehānismu palīdzību (un daudzas īpašības ir uzstādāmas alternatīvi, izmantojot abus mehānismus):

- izmantojot **manipulatorus** (izdrukas priekšraksta ietvaros),
- izmantojot objekta *cout* **funkcijas** (pirms izdrukas priekšraksta).

Turpmākajos piemēros būs parādīti abi alternatīvie varianti.

cout objekta izmantošanai tiek ielādēta bibliotēka *iostream*:

```
#include<iostream>
```

Ja tiek izmantoti manipulatori, tad papildus tiek ielādēta bibliotēka *iomanip*:

```
#include<iostream>
#include<iomanip>
```

Formatēšanas komandu darbības apgabals ir divu veidu:

- lielākā daļa formatēšanas komandu darbojas no uzstādīšanas brīža līdz atsaukšanai vai programmas beigām (piemēram, *precision*),
- daļa komandu attiecas tikai uz nākošo izdrukas operatoru << (pat ne uz visu nākošo izdrukas priekšrakstu! Piemēram, *width*).

Svarīgākās operācijas formatētajā izvadē (aprakstā pirms slīpsvītras uzrādīta objekta *cout* funkcija, bet pēc slīpsvītras – atbilstošais manipulators):

- *precision*/*setprecision* – uzstāda skaitļa precizitāti vai nu ciparos aiz komata, vai nu zīmīgajos ciparos (pēc noklusēšanas 6 zīmīgie cipari).

Sekojošais piemērs parāda skaitļa izdruku 3 variantos: ar 6 zīmīgajiem cipariem (pēc noklusēšanas), ar 5 (ar manipulatora uzstādījumu) un ar 4 (ar funkciju).

```
cout << 271.84753 << endl;
cout << setprecision(5) << 271.84753 << endl;
cout.precision(4);
cout << 271.84753 << endl;
```

```
271.848
271.85
271.8
```

- *width*/*setw* – uzstāda izvades apgabala platumu simbolos (**darbojas tikai uz nākošo izvadāmo vērtību**, pēc noklusēšanas izslēgts, t.i., izmanto tik daudz vietas, cik nepieciešams).

Nākošajā piemērā ‘B’ tiek izdrukāts uzreiz aiz ‘A’, ‘C’ tiek izdrukāts 3-vietīgā apgabalā, bet ‘D’ 12-vietīgā apgabalā (pēc noklusēšanas apgabala ietvaros pielīdzinot pie labās malas).

```
cout << "A" << "B";
cout << setw(3) << "C";
cout.width(12);
cout << "D" << endl;
```

```
AB  C           D
```

Iepriekš demonstrētais precizitātes uzstādīšanas piemērs ar *precision*/*setprecision* pēc noklusēšanas strādā uz **zīmīgajiem** cipariem, tomēr bieži formatēšanas interese attiecas uz cipariem **aiz komata**, nevis zīmīgajiem. Šādā gadījumā noder nākošā komanda.

setf(ios::fixed)/*fixed* – uzstādītās precizitātes noteikšana, ka tā attiecas uz cipariem aiz komata (nevis zīmīgajiem). Lai saprastu piemēru, sk. literāļa “skaitlis ar peldošo komatu” pierakstu.

```
cout.precision(2);
cout.setf (ios::fixed);
cout << 271.84753 << endl;
```

```
271.85
```

```
cout.precision(2);
cout << fixed << 271.84753 << endl;
```

```
271.85
```

Ja pirms izdrukas operatora tiek uzstādīts platums un tas ir lielāks nekā izdrukājamai informācijai nepieciešamais, tad izdrukā starp vērtībām tiek likti tukšumi, tomēr ir iespējams tukšumus (kā noklusēto aizpildīšanas simbolu) nomainīt ar citu.

- *fill*/*setfill* – uzstāda aizpildīšanas simbolu (ir nozīme gadījumā, ja ir uzstādīts arī platums, pēc noklusēšanas – tukšums).

```
cout.fill('_');
cout << setw(10) << 123;
cout << setfill('^') << setw(10) << 456 << endl;
```

```
_______123^^^^^^^456
```

“Dīvainu” pāri veido funkcija un manipulators *setf(ios::scientific)*/*scientific*, kas, atšķirībā no citu pāru parauga, nedara precīzi vienu un to pašu.

- *scientific* – manipulators, kas nosaka zinātnisko skaitļa izvades formātu.
- *setf(ios::scientific)* – funkcija, kas nosaka, ka uzstādītā precizitāte attiecas uz zīmīgajiem cipariem (nevis cipariem aiz komata).

Sekojošais piemērs demonstrēs šo abu komandu darbību:

```
cout.precision(3);
cout.setf (ios::fixed);
cout << 271.84753 << endl;
cout.setf (ios::scientific);
cout << 271.84753 << endl;
cout << scientific << 271.84753 << endl;
```

```
271.848
272
2.718e+002
```

Komandas *precision(3)* un *setf(ios::fixed)* uzstāda precizitāti – 3 zīmes aiz komata. Komanda *setf (ios::scientific)* šo 3 ciparu precizitāti pārliek uz zīmīgajiem cipariem, bet *scientific* nosaka zinātnisko pierakstu un tā ietvaros 3 ciparus aiz komata.

left,*right* – manipulatori pielīdzināšanai pie izdrukas apgabala kreisās vai labās malas (ir nozīme tikai gadījumā, ja ir uzstādīts platums ar *width*/*setw*).

```
cout.fill('*');
cout << left << setw(10) << 123 << endl;
cout << right << setw(10) << 456 << endl;
```

```
123*******
*******456
```

5.2. Formatēta ievade

5.2.1. Darbības princips

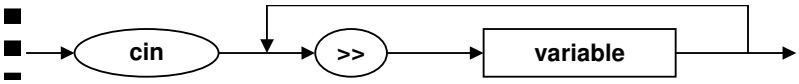
Formatēta ievade nodrošina informācijas pieņemšanu no klaviatūras un pārveidošanu par noteikta datu tipa vērtību, ja tas iespējams, un ierakstot to mainīgajā. Formatēta ievade notiek, izmantojot objektu *cin* un ievades operatoru >>. Izmantojamā bibliotēka: <*iostream*>. Sintakse ir visai līdzīga, kā tas ir formatētas izvades gadījumā:

formatted_input:
cin *input_sequence*

input_sequence:
input_element
input_element input_sequence

input_element:
>> *variable*
>> other lvalue

Att. 5.3. Formatēta input (*formatted_input*)



Att. 5.4. Formatēta izvade sintaktiskās diagrammas pierakstā

Kaut arī formatētas ievades sintakse līdzinās izvades sintaksei, tomēr ievadei ir vairākas būtiskas atšķirības, kuras tiek apskatītas tālāk.

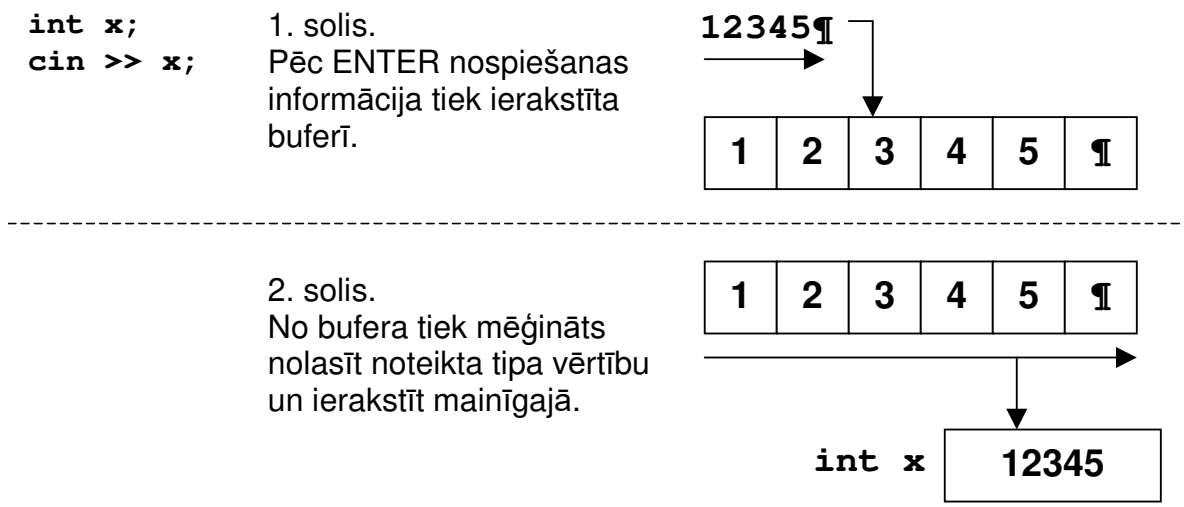
Līdzīgi kā izvadei, arī ievadi var noformēt vairākām vērtībām uzreiz vienā priekšrakstā kaskādes veidā. Nākošie divi programmas bloki divu mainīgo *i* un *k* ievadīšanai darbojas identiski.

```
cin >> i;
cin >> k;
```

```
cin >> i >> k;
```

Starp vairāku vērtību ievadīšanu var tikt likts viens vai vairāki atdalītājsimboli. Visi atdalītājsimboli ir līdzvērtīgi – tie nolasīšanas laikā tiek ignorēti, lai nolasītu "īstās" vērtības. Tomēr ENTER no programmas izpildes viedokļa ir nedaudz savādāks: lai kādus atdalītājsimbolus mēs liktu starp vērtībām, **vismaz pašās beigās ir jāliek ENTER**, jo tikai tad sākas pats nolasīšanas process (līdz tam notika tikai simbolu uzkrāšana). Tātad, ievades process notiek 2 posmos:

- informācijas nolasīšana no klaviatūras un uzkrāšana buferī līdz pat ENTER nospiešanai,
- bufera satura pārstaigāšana, mēģinot izvilkt noteikta tipa vērtību vai vērtības, lai ierakstītu mainīgajā vai mainīgajos.



Att. 5.5. Vienkāršota ievades divdaļīgā darbības shēma

To netieši demonstrē arī nākošais piemērs ar diviem programmas palaišanas variantiem, kuros tiks parādīti arī **neredzamie simboli** procesa uzskatāmībai ('·' tukšums, '¶' jaunas rindiņas simbols):

```
int i, k;
cin >> i;
cout << i << endl;
cin >> k;
cout << k << endl;
```

```
5¶
5
7¶
7
```

```
5·7¶
5
7
```

Dotais piemērs parāda, ka nolasīšanas process ir ne tikai divdaļīgs, bet tā izpilde notiek paralēli un asinhroni programmā nokodētā algoritma izpildei: ievērojiet, ka programmā mainīgā k nolasīšana notiek pēc mainīgā i izdrukāšanas uz ekrāna, taču otrajā programmas palaišanas piemērā tas it kā nostrādā otrādi, jo tikai pēc ENTER nospiešanas tā pa īstam sākas nolasīšanas process.

5.2.2. Ievades kļūdu apstrāde

Vēl viena atšķirība ir tāda, ka, ja izvade normālā gadījumā vienmēr notiks veiksmīgi (izņemot retus ārkārtas gadījumus), tad ievades komandai neveiksmīgs iznākums ir relatīvi parasta lieta, kas notiek ievadāmā teksta **formāta neatbilstības** dēļ mainīgā tipam, kurā notiek informācijas ielasīšana. Piemēram, ja komanda ir ielasīt datus mainīgajā ar skaitlisku tipu, bet uz klaviatūras tiek padoti burti, nevis cipari, protams, notiek **kļūda**. Ja ir iespēja notikt kļūdai, tad nepieciešami mehānismi kļūdas novērst vai apiet.

- Pēc katras lasīšanas operācijas ir iespējams pārbaudīt, vai tā bijusi **veiksmīga** vai nē. To veic ar objekta *cin* funkciju *good* (iespējams pārbaudīt arī uz neveiksmi ar funkcijām *bad* vai *fail*).

To demonstrē nākošais programmas fragments, kuram seko divi programmas palaišanas varianti – bez kļūdas un ar kļūdu (jāatgādina, ka darbības piemērā no klaviatūras ievadītais teksts iezīmēts treknināti (*bold*)).

Sekojošās programmas būtība: ja uz klaviatūras tiek padots korekts skaitlis, tad tas tiek veiksmīgi nolasīts un izdrukāts uz ekrāna aiz teksta "OK: ", bet, ja padotā vērtība neatbilst skaitlim, tiek paziņots par kļūdu.

```
int i;
cin >> i;
if (cin.good()) cout << "OK: " << i << endl;
else cout << "ERROR" << endl;
```

```
123
OK: 123
```

```
abc
ERROR
```

Ja nolasīšana bijusi neveiksmīga, objekts *cin* tiek nobloķēts, un, ja to neatbloķē, visas turpmākās darbības ar to tiek ignorētas.

Nākošajā piemērā redzams, ka pēc kļūdas komandā *cin >> i* (mēģinot ievadīt tekstu "abc" skaitliskā tipa mainīgajā), nākošā ievades komanda (*cin >> k*) tiek ignorēta (resp., otrreiz ievade no klaviatūras netiek prasīta). Abas izdrukas komandas nostrādā normāli, izdrukājot mainīgo *i* un *k* sākotnējās vērtības.

```
int i=5, k=7;
cin >> i;
cout << i << endl;
cin >> k;
cout << k << endl;
```

```
abc
5
7
```

Lai turpinātu darbu pēc neveiksmīgas ievades, ir jāveic divas operācijas:

- objekta *cin* atbloķēšana ar funkciju *clear*,
- ievadītās simbolu virknes ignorēšana līdz tuvākajam ENTER (funkcija *ignore*).

Otrā no veicamajām darbībām saistīta ar nolasīšanas procesa 2. soli (sk. Att. 5.5).

Ievades objekta *cin* atbloķēšanu pēc kļūdas iestāšanās parāda nākošais piemērs.

```
int i=5, k=7;
cin >> i;
cout << i << endl;
cin.clear ();
cin.ignore (256,'\n');
cin >> k;
cout << k << endl;
```

```
abc
5
99
99
```

Iepriekšējais piemērs nestrādās, ja pirms ENTER tiks ievadīti vairāk nekā 256 simboli, tāpēc 256 vietā var likt maksimālo bufera izmēru, tikai tādā gadījumā jāiekļauj papildus bibliotēka *<limits>*.

```
// #include <limits>
int i=5, k=7;
cin >> i;
cout << i << endl;
cin.clear ();
cin.ignore (std::numeric_limits<std::streamsize>::max(),'\n');
cin >> k;
cout << k << endl;
```

```
abc
5
99
99
```

Šajā piemērā, mēģinot nolasīt *i*, iestājās kļūda, pēc tam *cin* objekts tiek atbloķēts ar *clear*, bet starpbuferis "iztīrīts" ar *ignore*, lai ar nākošo komandu *cin >> k* atkal mēģinātu nolasīt skaitli.

Pilnīga ievades kļūdu apstrādes sistēma ir tad, kad pieprasījums pēc ievades tiek prasīts tik ilgi, kamēr tiek ievadīta pareiza vērtība. To parāda nākošais piemērs.

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

int main ()
{
    int i;
    cout << "Input integer value: ";
    cin >> i;
    while (cin.fail())
    {
        cin.clear ();
        cin.ignore (numeric_limits<streamsize>::max(), '\n');
        cout << "Try again: ";
        cin >> i;
    };
    cout << "Success: " << i << endl;
}
```

```
Input integer value: abc
Try again: +++
Try again: 123
Success: 123
```

Att. 5.6. Vesela skaitļa ievadīšana ar kļūdu apstrādi

Nākošajā piemērā parādīts veids, kā vērtības tiek ievadītas vairākās citās valodās – vispirms tieši simbolu virknē (mainīgais s), pēc tam mēģinot nolasīt veselu skaitli no šīs simbolu virknes (izmantojot stringstream objektu, kas dod iespēju lasīt no simbolu virknes kā no konsoles). Vienas int vērtības nolasīšanai tiek izmantota funkcija.

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

bool getInt(int &n)
{
    string s;
    getline(cin,s);
    stringstream ss(s);
    ss>>n;
    return ss.fail()==false;
}

int main ()
{
    int i;
    cout << "Input integer value: ";
    bool ok = getInt(i);
    while (!ok)
    {
        cout << "Try again: ";
        ok = getInt(i);
    };
    cout << "Success: " << i << endl;
}
```

```
Input integer value: abc
Try again: +++
Try again: 123
Success: 123
```

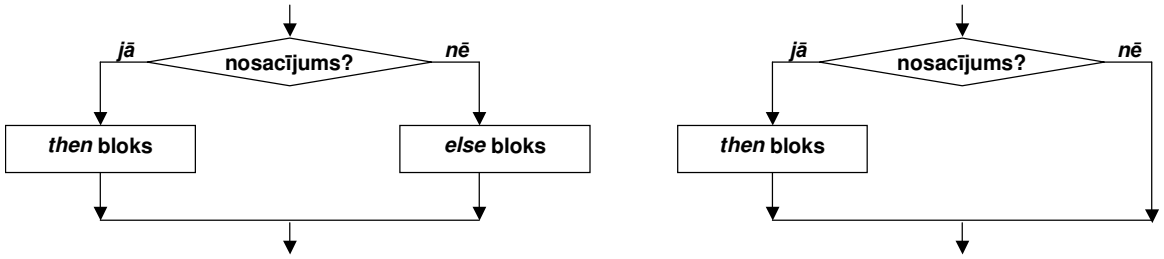
Att. 5.7. Vesela skaitļa ievadīšana ar kļūdu apstrādi, sākumā nolasot informāciju kā simbolu virkni

6. Zarošanās un loģiskās izteiksmes

Zarošanās (*branching*, *selection*) ir viena no 3 galvenajām vadības konstrukcijām (blakus secībai (*sequence*) un ciklam (*looping*). Zarošanos valodā C++ realizē zarošanās priekšraksti (*selection statements*) *if-then-else* 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.

6.1. Zarošanās priekšraksts if-then-else

Programmas darbības gaitā var rasties nepieciešamība atkarībā no kāda nosacījuma izpildīt vai neizpildīt kādu programmas daļu. To parasti veic ar zarošanās priekšrakstu *if-then-else*.



Att. 6.1. Priekšraksta *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.

if_statement:
`if` (*logical_expression*) *statement* *else_block*$_{opt}$

else_block:
`else` *statement*

Att. 6.2. Zarošanās priekšraksts *if-then-else* (*if_statement*)

if-then-else pielietojumu demonstrē nākošais piemērs:

```
if (a > 0) cout << "positive" << endl;
else cout << "not positive" << endl;
```

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. Tādējādi *if-then-else* sintakse ir aprakstāma arī šādi:

if_statement_v2:

if (*logical_expression*) *statement* *else_block*$_{opt}$

else_block:

else *statement*

else if (*logical_expression*) *then_block* *else_block*

Att. 6.3. Zarošanās priekšraksts *if-then-else*, 2. variants (*if_statement_v2*)

if-then-else pielietojums ar else-if komponenti redzams nākošajā piemērā (== nozīmē salīdzināšanu uz vienādību, nesajaukt ar =, kas valodā C++ nozīmē piešķiršanu!):

```
if (a > 0) cout << "positive" << endl;
else if (a == 0) cout << "zero" << endl;
else cout << "negative" << endl;
```

Gan *then* blokā, gan *else* blokā var atrasties arī cits *if-then-else* priekšraksts. Komplektā ar to, ka *if-then-else* priekšraksts var parādīties 2 veidos – ar un bez *else* zara – tas var it kā novest pie efekta, ka nav skaidrs, kuram *if-then-else* priekšrakstam īsti pieder *else* zars. Tāpēc, lai novērstu divdomību, ir noteikts, ka *else* attiecas uz tuvāko *if*. Sekojošajā piemērā *else* attiecas uz *if (a > 0)*. Tādējādi, dotais programmas kods vispār kaut ko izdrukā tikai tad, ja *a* ir pāra skaitlis. (*a%2 == 0* nozīmē: dalot *a* ar 2, atlikums sanāk 0)

```
if (a%2 == 0)
if (a > 0) cout << "Positive even" << endl;
else cout << "Negative even" << endl;
```

Iepriekšējais programmas fragments ir papildināts un apgādāts ar izpildes piemēriem tālāk (beidzamajos divos programmas izpildes piemēros netiek izdrukāts nekas):

```
int a;
cin >> a;
if (a%2 == 0)
    if (a > 0) cout << "Positive even" << endl;
    else cout << "Negative even" << endl;
```

```
8
Positive even
```

```
-8
Negative even
```

```
7
```

```
-7
```

Lai panāktu, ka *else* attiecas uz pirmo *if*, ir jālieto papildus figūriekavas:

```
int a;
cin >> a;
if (a%2 == 0)
{
    if (a > 0) cout << "Positive even" << endl;
}
else cout << "Odd" << endl;
```

```
8
Positive even
-8
7
Odd
-7
Odd
```

6.2. Loģiskas izteiksmes

6.2.1. Vispārīgs apraksts

Priekšraksta *if-then-else* zarošanos nosaka zarošanās nosacījums, kuru reprezentē loģiska izteiksme, kas vienkāršākajā un tipiskākajā gadījumā ir salīdzināšanas operācija, piemēram, *a>0*, tomēr vispārīgā gadījumā tā var būt krietni sarežģītāka.

> **Loģiska izteiksme** (*logical expression*) ir konstrukcija, kas sastāv no loģiskām operācijām un loģiskiem operandiem, un kura pēc šo operāciju izpildes izrēķina loģisku vērtību.

Loģiska izteiksme ir zarošanās priekšraksta *if-then-else*, kā arī visu cikla priekšrakstu obligāta sastāvdaļa.

logical_expression:
boolean_literal
boolean_variable
logical_function_call
logical_expression logical_operator logical_expression
unary_logical_operator logical_expression
expression comparison_operator expression
(*numeric_expression*)

boolean_literal: one of
`true false`

logical_operator: one of
`|| &&`

unary_logical_operator:
`!`

comparison_operator: one of
`< <= > >= == !=`

Att. 6.4. Vienkāršota loģiskā izteiksme (*logical_expression*)

Vispārīgā gadījumā loģiska izteiksme dod atbildi *"patiess"*/*"nepatiess"* (true/false) uz "jautājumu", kas pierakstīts ar loģisko operāciju un operandu palīdzību.

6.2.2. Salīdzināšanas operācijas – vienkāršākās loģiskās konstrukcijas

Tipiskākais vienkāršas loģiskas izteiksmes piemērs, kā jau tika minēts, ir salīdzināšana. Nākošā tabula parāda salīdzināšanas operatorus un to nozīmi.

Tab. 6.1.

Salīdzināšanas operatori

Matemātiskais simbols	Būtība	C++ apzīmējums	Piemērs	Matemātiskais ekvivalents
=	*vienāds*	`==`	`x + 7 == 2 * y`	x+7=2y
≠	*nav vienāds*	`!=`	`s != 'n'`	s ≠ 'n'
<	*mazāks*	`<`	`a < b - 5`	a < b-5
≤	*mazāks vai vienāds*	`<=`	`a <= max`	a ≤ max
>	*lielāks*	`>`	`a > b + 5`	a > b+5
≥	*lielāks vai vienāds*	`>=`	`a >= min`	a ≥ min

Valodā C++ ir viegli nokļūdīties un salīdzināšanas operatora == vietā ierakstīt vienkāršo =, kas nozīmē piešķiršanu – kompilators to pieļaus, bet kļūdu atrast būs ļoti grūti.

Jebkura salīdzināšanas operācija, kā jau loģiskai konstrukcijai pienākas, atgriež vērtību *true* vai *false*, un nevienu citu.

6.2.3. Loģiskie operatori

Dažādas loģiskās vērtības, piemēram, salīdzināšanas operācijas, var apvienot, veidojot sarežģītākas loģiskās izteiksmes. Šim nolūkam der loģiskie operatori `!` `&&` `||`. Loģisko operatoru operandi var būt jebkādas loģiskas vērtības, t.sk., salīdzināšanas operācijas, funkcijas, t.i., citas loģiskas izteiksmes. Loģisko operatoru aprakstā *true/false* vietā vai paralēli bieži tiks lietoti *1/0*, jo dažkārt tā ir uzskatamāk.

6.2.3.1. Operators &&

Operators *&&* (loģiskais "un") atgriež *true* tikai tajā gadījumā, ja abi operandi atgriež *true*, citādi atgriež *false*:

Tab. 6.2.

Loģiskais "un" (&&)

A	B	A && B
false (0)	false (0)	false (0)
false (0)	true (1)	false (0)
true (1)	false (0)	false (0)
true (1)	true (1)	true (1)

Nākošā loģiskā izteiksme pasaka, **vai** dotais skaitlis **ir** pozitīvs viencipara skaitlis, vai nav (ievērojiet, ka, izdrukājot loģisko vērtību *true* vai *false*, tiek izdrukāts attiecīgi 1 vai 0).

```
cin >> a;
cout << (a >= 0 && a <= 9) << endl;
```

```
-8
0
```

```
5
1
```

```
99
0
```

6.2.3.2. Operators ||

Operators || (loģiskais "vai") atgriež *true* tajā gadījumā, ja kaut vai viens (t.sk. abi) operandi atgriež *true*, citādi (ja abi operandi ir *false*) atgriež *false*:

Tab. 6.3.

Loģiskais "vai" (||)

A	B	A || B
false (0)	false (0)	false (0)
false (0)	true (1)	true (1)
true (1)	false (0)	true (1)
true (1)	true (1)	true (1)

Nākošā loģiskā izteiksme pasaka, **vai taisnība, ka** dotais skaitlis **nav** pozitīvs viencipara skaitlis (ievērojiet, ka, izdrukājot loģisko vērtību *true* vai *false*, tiek izdrukāts attiecīgi 1 vai 0).

```
cin >> a;
cout << (a < 0 || a > 9) << endl;
```

```
-8
1
```

```
5
0
```

```
99
1
```

6.2.3.3. Operators !

Operators *!* (noliegums) atgriež *true*, ja tā operands atgriež *false*, un *false*, ja operands atgriež *true*. Parasti nolieguma operators tiek izmantots komplektā ar kādu no abiem pārējiem loģiskajiem operatoriem, jo citādi no tā var izvairīties, izvēloties attiecīgu salīdzināšanas operatoru.

Tab. 6.4.

Noliegums (!)

A	!A
false (0)	true (1)
true (1)	false (0)

Nākošā loģiskā izteiksme pasaka, **vai** dotais skaitlis **ir** pozitīvs viencipara skaitlis, vai nav (salīdzināt ar piemēriem pie operatoriem && un ||, no kuriem pirmais dara to pašu, bet otrais pretēji).

```
cin >> a;
cout << !(a < 0 || a > 9) << endl;
```

```
-8
0
```

```
5
1
```

```
99
0
```

6.2.3.4. Loģisko operatoru prioritātes

Loģisko operatoru prioritātes rinda ir `! && ||`, kur stiprākais ir kreisajā pusē. Lai nomainītu izpildes secību, jā tā neatbilst prioritāšu dēļ, tad jālieto papildus iekavas.

Nākošais piemērs pasaka, vai mēneša *m* (1..12) datums *d* (1..31) ir īsā mēneša pēdējā diena. Papildus iekavas tiek lietotas, jo loģiskais "vai" ir vājāks par loģisko "un".

```
int m, d;
cin >> m >> d;
cout << ((m==4 || m==6 || m==9 || m==11) && d==30) << endl;
```

```
4 30
1
```

```
5 30
0
```

```
9 5
0
```

6.2.4. 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 jebkura 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.

6.3. Zarošanās priekšraksts switch-case

Ja izvēle notiek starp diskrētām vērtībām, kas nav īpaši lielā skaitā, tad ērtāk un uzskatāmāk būtu lietot priekšrakstu *switch-case*. Priekšraksta *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 sākotnēji padara *switch-case* priekšraksta darbības shēmu nedaudz neordināru un ne tik viegli saprotamu, tāpēc vieglāk saprast tā darbību būtu pēc piemēra.

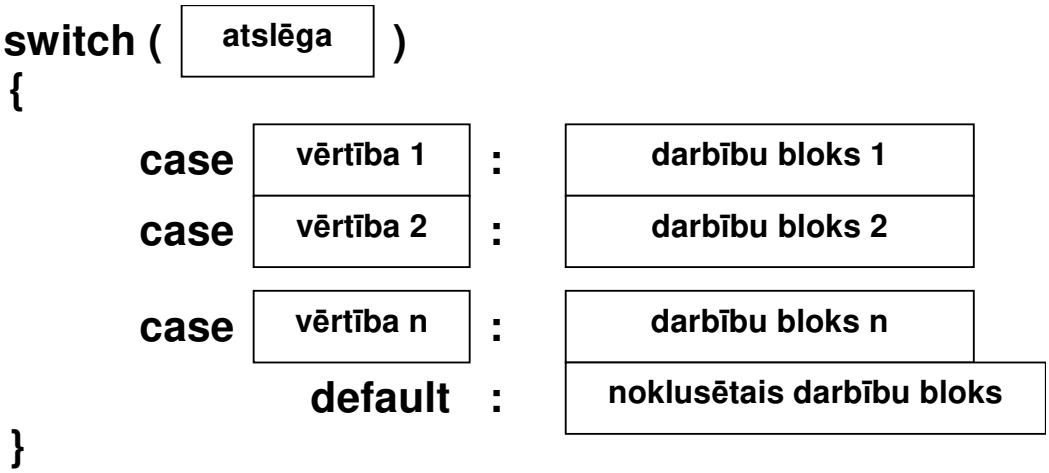
switch_statement:
switch (*expression*) { *case_list* }

case_list:
case_default$_{opt}$
case *value* : *statement_list*$_{opt}$

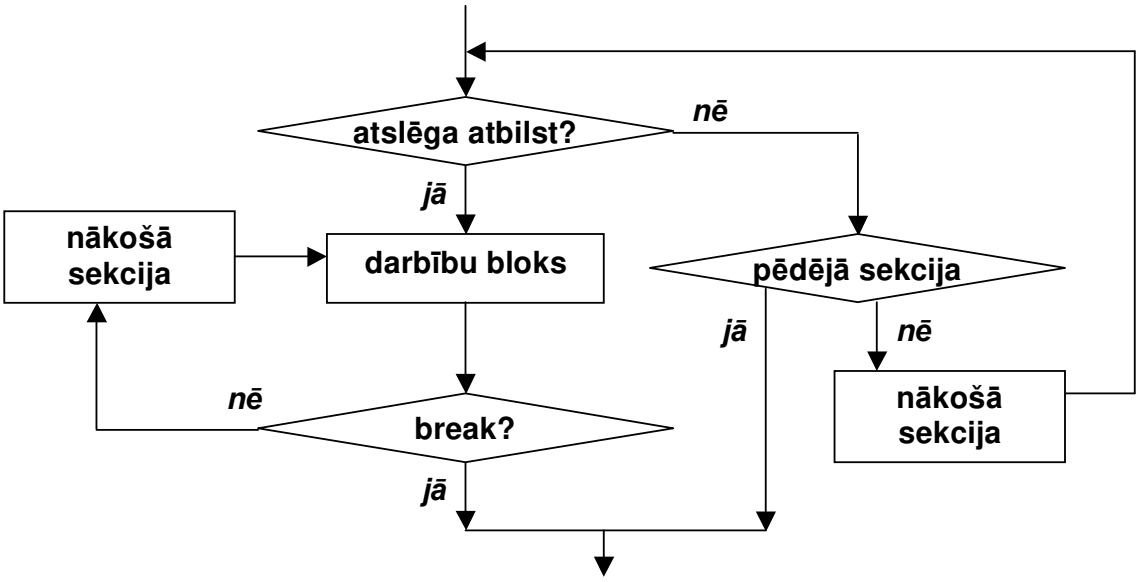
case_default:
default: *statement_list*$_{opt}$

statement_list:
statement
statement statement_list

Att. 6.5. Zarošanās priekšraksts *switch-case* (*switch_statement*)



Att. 6.6. Zarošanās priekšraksta *switch-case* loģiskā struktūra



Att. 6.7. Zarošanās priekšraksta *switch-case* darbības shēma

Switch-case vienkāršs piemērs brīvdienu noskaidrošanai (ievadā tiek dota nedēļas diena 1..7).

```
cin >> a;
switch (a)
{
    case 6:
        cout << "Saturday" << endl;
        break;
    case 7:
        cout << "Sunday" << endl;
        break;
    default:
        cout << "Working day" << endl;
};
```

```
6
Saturday
```

```
7
Sunday
```

```
5
Working day
```

Switch-case papildus piemērs brīvdienu noskaidrošanai, ja sestdiena un svētdiena atgriež vienādu rezultātu (ievērojiet, ka aiz ‘case 6’ nav break, tāpēc, ievadot 6, darbība switch priekšrakstā turpinās nākošajā apakšblokā).

```
cin >> a;
switch (a)
{
    case 6:
    case 7:
        cout << "Holiday" << endl;
        break;
    default:
        cout << "Working day" << endl;
};
```

```
6
Holiday
```

```
7
Holiday
```

```
5
Working day
```

Switch-case sarežģītāks darbības piemērs, kurš parāda arī operatora *break* nozīmi izejai no priekšraksta. (Ievērojiet, ka ar vērtību 1 nostrādā kods aiz ‘case 2’, jo aiz ‘case 1’ nav *break*. bez tam, tā kā aiz bloka ‘3’ nav break, ar vērtību 3 nostrādā gan bloks 3, gan bloks 4).

```
cin >> a;
switch (a)
{
    case 1:
```

```
        case 2:
            cout << "a is 1 or 2" << endl;
            break;
        case 3:
            cout << "a is 3" << endl;
        case 4:
            cout << "a is 4" << endl;
            break;
        default:
            cout << "a is bad" << endl;
    };
```

```
1
a is 1 or 2
```

```
2
a is 1 or 2
```

```
3
a is 3
a is 4
```

```
4
a is 4
```

```
5
a is bad
```

6.4. 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.

conditional_functor:
logical_expression ? *then_expression* : *else_expression*

then_expression:
expression

else_expression:
expression

Att. 6.8. Nosacījuma funktors *?:* (*conditional_functor*)

Funktora ?: priekšrocība ir ērta izmantošana, veidojot izteiksmes.

```
cin >> a;
cout << (a>0 ? "positive" : "not positive") << endl;
```

```
5
positive
```

```
-5
not positive
```

6.5. Izteiksmes vispārīgā nozīmē

Iepriekšējās sadaļās tika apskatītas aritmētiskas izteiksmes un loģiskas izteiksmes, uzsverot katras specifisko funkcionalitāti, tai pat laikā vienkāršojot to definēšanu. Tai pat laikā jau sadaļā 6.2.4. "Aritmētisku un loģisku izteiksmju saistība" tika parādīts, ka valodā C++ aritmētiskas un loģiskas izteiksmes netiek izšķirtas. Patiesībā ir vēl "sliktāk" – valodā C++ izteiksme ir vēl plašāks jēdziens nekā tikai aritmētisku un loģisku izteiksmju apvienojums.

> **Izteiksme** (*expression*) ir konstrukcija, kas sastāv no operatoriem un operandiem un nosaka skaitļošanas procesu.

Operatori izteiksmē var būt jebkādi un arī operandi (noteiktas vērtības reprezentējoši elementi) var būt dažādu tipu – ka tikai operatori tos spēj apstrādāt. Tāpēc vienīgais, kas neļauj izteiksmei būt tikpat plašam jēdzienam kā algoritms vai programma, ir tas, ka operandu sasaistei tiek lietoti **operatori** – un nekādas citas konstrukcijas.

Tālāk parādīts vienkāršots vispārīgas izteiksmes sintakses pieraksts. Tajā nav ievietoti vairāki specifiski elementi, piemēram, bitu līmeņa darbības (izņemot piešķiršanu), tipu pārveidošana u.c. Pilns izteiksmes sintakses apraksts būtu vismaz divas reizes garāks. Tai pat laikā jāpiezīmē, ka tik sarežģīts izteiksmes apraksts, no kura izriet arī izteiksmes kā konstrukcijas salīdzinoši lielās iespējas, ir C++ specifika – citās programmēšanas valodās izteiksmes jēdziens ir daudz vienkāršāks.

```
postfix_expression:
        literal
        variable
        function_call
        ( expression )
        postfix_expression [ expression ]
        postfix_expression member_operator variable
        postfix_expression member_operator function_call
        postfix_expression increment_decrement_operator

member_operator: one of
        . ->

increment_decrement_operator: one of
        ++ --

unary_expression:
        postfix_expression
        increment_decrement_operator unary_expression
        unary_operator unary_expression

unary_operator: one of
        * & + - ! ~

multiplicative_expression:
```

```
            unary_expression
            multiplicative_expression  multiplicative_operator
    unary_expression

    multiplicative_operator:  one of
            * / %

    additive_expression:
            multiplicative_expression
            additive_expression  additive_operator  multiplicative_expression

    additive_operator:  one of
            + -

    comparison_expression:
            additive_expression
            comparison_expression  comparison_operator  additive_expression

    comparison_operator:  one of
            < <= > >= == !=

    logical_and_expression:
            comparison_expression
            logical_and_expression  &&  comparison_expression

    logical_or_expression:
            logical_and_expression
            logical_or_expression  ||  logical_and_expression

    assignment_expression:
            logical_or_expression
            logical_or_expression  assignment_operator
    assignment_expression

    assignment_operator:  one of
            = *= /= %= += -= >>= <<= &= ^= |=

    expression:
            assignment_expression
            expression  ,  assignment_expression
```

Att. 6.9. Vienkāršota izteiksme (*expression*)

Nākošajā piemērā parādīta izteiksme, kura nav ne aritmētiska, ne loģiska – bet vienkārši izteiksme. Šī izteiksme izdrukā loģisku vērtību (1 vai 0), bet tās operandi ir skaitliskas un teksta vērtības.

Sekojošais piemērs satur izteiksmi

```
a > (s + "Hello, World!").size() * 2
```

Šī izteiksme atgriež loģisku vērtību, pasakot, vai taisnība, ka skaitlis a, kas ievadīts no klaviatūras, ir lielāks par teksta “Hello, World!” garumu (pieņemot, ka s ir “Hello, ”), kas pareizināts ar 2 (t.i. 26).

Kaut arī sekojošais piemērs tika rakstīts augstāk pieminētās izteiksmes dēļ, tomēr īstenībā par izteiksmi C++ nozīmē ir uzskatāma visa attiecīgā rindiņa:

```
cout << (a > (s + "Hello, World!").size() * 2) << endl
```

jo arī << ir operators (kurš gan netika ietverts Att. 6.9 izteiksmes aprakstā).

```
int a;
cin >> a;
string s = "Hello, ";
cout << (a > (s + "World!").size() * 2) << endl;
```

```
20
0
```

```
30
1
```

7. 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 valodā C++ (atšķirībā no daudzām citām valodām) 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**.

Cikla ķermenis ir programmas bloks cikla konstrukcijas ietvaros, kas nosaka darbību, kas ciklā tiks atkārtota. Viena cikla ķermeņa izpildīšanu vienu reizi sauc par **iterāciju**. Vispārīgā gadījumā cikla izpildē var būt neviena, viena vai vairākas iterācijas.

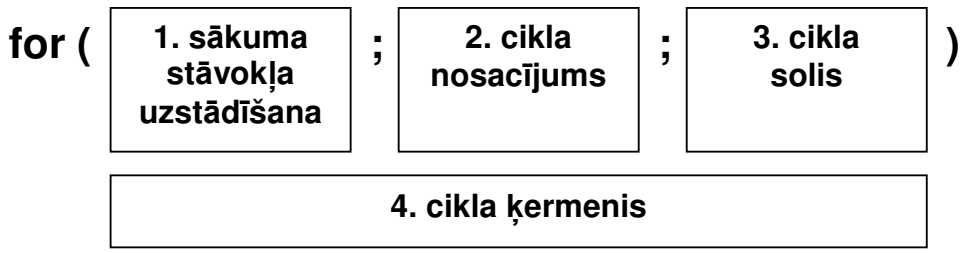
7.1. Cikls for

Sākot ar C++11 versiju, valodā C++ ir divu veidu for cikli:

- klasiskais – ar skaitītāju,
- "modernais" – datu virknes pārstaigāšanai.

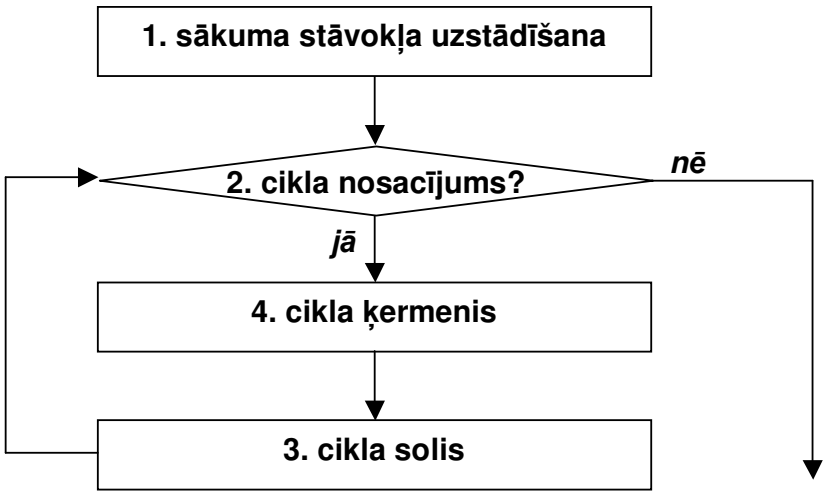
7.1.1. Cikls for – pamatvariants ar skaitītāju pār intervālu

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



Att. 7.1. Cikla *for* loģiskā struktūra

Cikls *for* sākas ar sākuma stāvokļa uzstādīšanu, kurā parasti tiek inicializēti (vai pat deklarēti un inicializēti) mainīgie. Pēc tam notiek cikla nosacījuma pārbaude. Ja pārbaude ir veiksmīga (*true*), tiek izpildīta kārtējā iterācija, kas sastāv no cikla ķermeņa izpildes, kam seko cikla soļa izpilde. Pēc katras iterācijas tiek pārbaudīts cikla nosacījums, un cikls tiek izpildīts tik ilgi, līdz cikla nosacījums vairs neizpildās (*false*):



Att. 7.2. Cikla *for* darbības shēma

Tālāk seko *for* cikla sintaktiskais apraksts. Ievērojiet, ka *for* cikla galvā visas 3 komponentes ir neobligātas – ja nav uzrādīta pirmā vai trešā komponente, tad tajā vietā vienkārši nekas nenotiek, bet ja nav uzrādīta otrā komponente (cikla nosacījums), tad tiek uzskatīts, ka cikla nosacījums ir *true*, un vienīgais veids, kā iziet no cikla, ir cikla ķermenī izsaukt operatoru *break*.

for_looping:
for (*for_heading*) $loop\_body_{opt}$

for_heading:
for_init_{opt} ; $loop\_condition_{opt}$; $for\_counter_{opt}$)

loop_body:
statement

for_init:
variable_declarator
expression

loop_condition:
logical_expression

for_counter:
expression

Att. 7.3. Cikls *for* (*for_looping*)

Nākošais for cikla piemērs parāda skaitļu 1..*n* summas izrēķināšanu:

```
int a, s=0;
cin >> a;
for (int i=1; i<=a; i++) s+=i;
cout << s << endl;
```

```
9
45
```

Šajā piemērā cikla ķermenis ir viens vienkāršs piešķiršanas priekšraksts *s+=i;*, taču parasti tas ir sarežģītāks, un to veido vesels instrukciju bloks, kas iekļauts figūriekavās:

```
for (...; ...; ...)
{
   ...
};
```

Ja *for* cikla inicializācijas komponentē tiek deklarēts mainīgais, tas darbojas tikai cikla ietvaros, bet pēc tā vairs nedarbojas:

```
for (int i=1; ...; ...)
{   ...  };
cout << i << endl; // KĻŪDA:  i vairs neeksistē!
```

7.1.2. Cikls for – datu virknes pārstaigāšanai

Sākot ar C++11 versiju, ir pieejams vēl viens cikla for veids – datu virknes pārstaigāšana, kur katru cikla iterāciju reprezentē nevis skaitītāja vērtība, bet gan elements datu virknē.

Att. 7.4. Cikla *for* datu virknes pārstaigāšanai struktūra

Nākošais *for* cikla piemērs parāda virknē norādīto skaitļu un to kvadrātu izdruku:

```
for (int m: {1,2,12})
    cout<<m<<" "<<m*m<<endl;
```

```
1 1
2 4
12 144
```

Skaitļu virkne var būt ne tikai cieti iekodēta, kā redzams iepriekšējā piemērā, bet arī kāda datu struktūra, piemēram vector (par vector, sk. arī sadaļu 20.2):

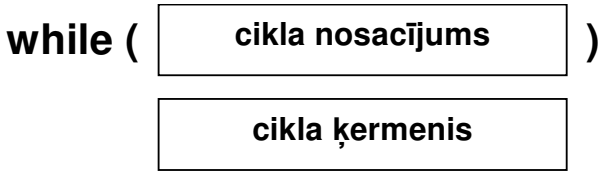
```
#include<iostream>
#include<vector>
using namespace std;
int main() {
    vector<string> seasons =
     {"winter","spring","summer","autumn"};
    int i=0;
    for (string s: seasons) {
```

```
            cout<<i+1<<" "<<s<<endl;
            ++i;
        }
    }
```

```
1 winter
2 spring
3 summer
4 autumn
```

7.2. 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.



Att. 7.5. Cikla *while* loģiskā struktūra

Cikla *while* sintakse parādīta nākošajā attēlā. Svarīgi ievērot, ka cikla nosacījumu aprakstošā loģiskā izteiksme obligāti jāliek iekavās (daudzās programmēšanas valodās tas nav obligāti):

Att. 7.6. Cikls *while* (*while_looping*).

while_looping:
while (*logical_expression*) $statement_{opt}$

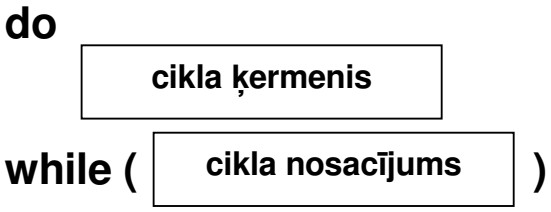
Nākošā programma ļauj no klaviatūras ievadīt veselus pozitīvus skaitļus un beigās izvada iepriekš ievadīto skaitļu summu. Kā ievadīšanas beigu pazīme kalpo nulles vai negatīva skaitļa ievadīšana:

```
int a, s=0;
cin >> a;
while (a > 0)
{
    s += a;
    cin >> a;
};
cout << s << endl;
```

```
6
2
7
0
15
```

7.3. 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.



Att. 7.7. Cikla *do-while* loģiskā struktūra

Atšķirībā no cikla *while*, cikla ķermenis ne tikai atrodas pirms nosacījuma, bet tas nedrīkst būt tukšs:

do-while_looping:
do *statement* while (*logical_expression*)

Att. 7.8. Cikls do-while (do-while_looping)

Nākošais programmas piemērs ir identisks iepriekšējam – programma ļauj no klaviatūras ievadīt veselus pozitīvus skaitļus un beigās izvada iepriekš ievadīto skaitļu summu, kā ievadīšanas beigu pazīme kalpo nulles vai negatīva skaitļa ievadīšana. Ieguvums ir tas, ka, salīdzinot ar *while* ciklu, nav 2 reizes jāatkārto skaitļa ievadīšanas kods *cin>>a*.

```
int a=0, s=0;
do
{
    s += a;
    cin >> a;
} while (a > 0);
cout << s << endl;
```

```
6
2
7
0
15
```

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, šī programma strādās pareizi tikai tad, ja pirms cikla mainīgais *a* būs inicializēts ar 0 (atšķirībā no attiecīgā piemēra ciklam *while*).

Tas, vai izmantot ciklu *while*, vai *do-while*, ir gaumes jautājums, tomēr ir uzskats, ka *while* cikls ir labāk lasāms, tāpēc ciklu *do-while* bez vajadzības nevajadzētu lietot.

7.4. 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.

Nākošā programma ir līdzīga kā iepriekšējā (*do-while*) – programma ļauj no klaviatūras ievadīt veselus pozitīvus skaitļus un beigās izvada iepriekš ievadīto skaitļu summu, kā ievadīšanas beigu pazīme kalpo nulles vai negatīva skaitļa ievadīšana. Tomēr tā ir papildināta ar divām lietām:

Rindiņa *if (a == b) break;* nodrošina, ka iziešana no cikla notiek arī tad, kad divreiz pēc kārtas ievada vienu un to pašu skaitli (rindiņa *b=a;* nodrošina, ka mainīgajā *b* glabājas iepriekšējā nolasītā vērtība) (pirmais izpildes piemērs aiz programmas).

Rindiņa *if (a%2 != 0) continue;* nodrošina, ka gadījumā, kad *a* ir nepāra skaitlis (dalījuma ar divi atlikums nav nulle), tiek pārlekts uz nākošo iterāciju, izlaižot pieskaitīšanu, kas nozīmē, ka šī programma nesummē nepāra skaitļus (otrais izpildes piemērs aiz programmas).

```
int a=0, b=0, s=0;
do
{
    b = a;
    cin >> a;
    if (a == b) break;
    if (a%2 != 0) continue;
    s += a;
} while (a > 0);
cout << s << endl;
```

```
4
4
4
```

```
1
2
3
4
0
6
```

Pirmajā izpildes piemērā otrais četrinieks kalpo par beigu pazīmi, tāpēc neieiet summā.

Operatori *break* un *continue* daudzos gadījumos ir ļoti ērti, tomēr jāsaprot, ka teorētiski tā ir neliela atkāpšanās no strukturētas programmēšanas pamatprincipiem (tiesa gan, nav salīdzināma ar *goto*), un bez vajadzības šos operatorus nevajadzētu izmantot.

Ikvienu *break* un *continue* operatoru var aizstāt ar citām konstrukcijām, papildus izmantojot *if* priekšrakstu vai papildinot cikla nosacījumu.

7.5. Cikla invariants algoritma pareizības novērtēšanā

7.5.1. Vispārīgs apraksts

Programmas vai algoritma pareizības novērtēšana notiek, dažādos programmas punktos definējot stāvokļus un izsakot apgalvojumus. Tādā veidā tiek iegūts apgalvojumu kopums, kurā katrs no apgalvojumiem ir sekas no programmas sākuma nosacījuma (*precondition*) un instrukciju virknes no programmas sākuma līdz tam punktam, kurā atrodas dotais apgalvojums. Ja šāds apgalvojums atrodas programmas beigās, tad, salīdzinot to ar programmas specifikāciju, iespējams spriest par programmas pareizību.

Viens no sarežģītākajiem uzdevumiem programmas vai algoritma pareizības noteikšanā, ir ciklu pareizības noteikšana (vieglāk pareizību noteikt ir konstrukcijām, kas nav saistītas ar atkārtošanu). Cikla pareizības noteikšanā noderīgs ir cikla invariants.

> **Cikla invariants** (*loop invariant*) ir apgalvojums, kas ir spēkā visā cikla darbības laikā.

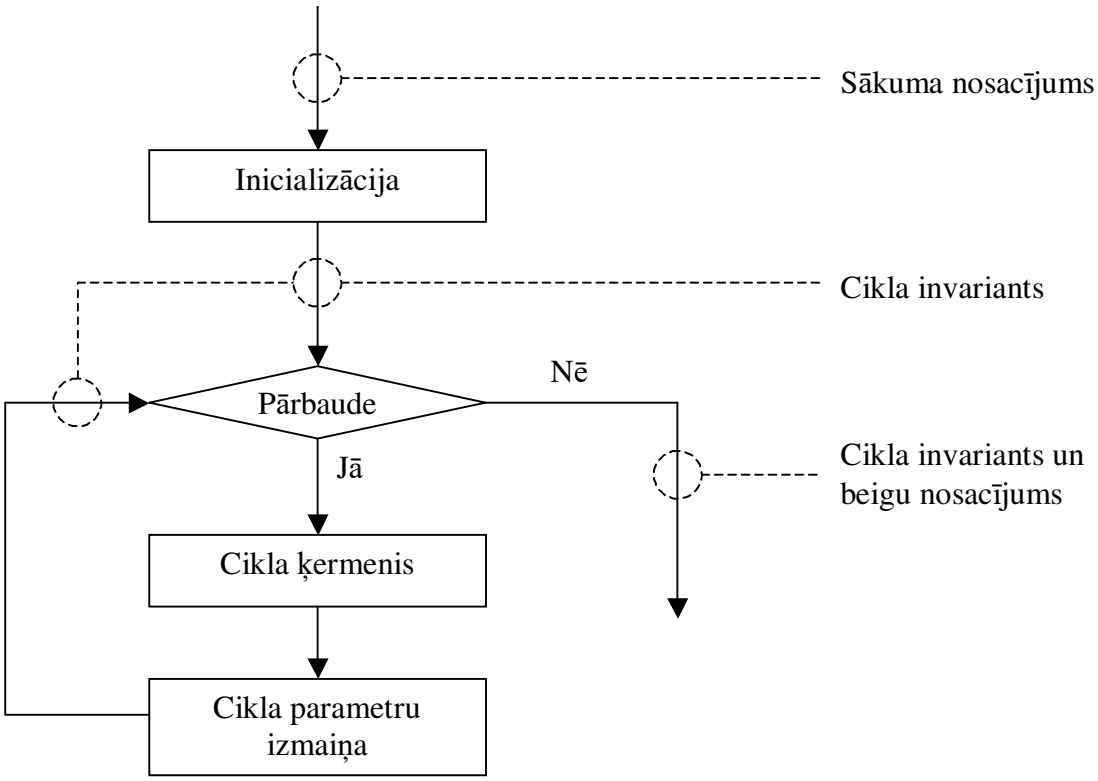
Nākošajā attēlā redzama tipiska cikla konstrukciju aprakstoša blokshēma un cikla invarianta un citu apgalvojumu izpildīšanās vieta tajā.

Kā redzams, cikla invariants ir apgalvojums, kas izpildās šādās vietās:

- tieši pirms cikla nosacījuma pārbaudes jebkurā cikla iterācijā,
- uzreiz pēc cikla (jo pirms pēdējā nosacījuma pārbaudes tas izpildījās, bet pati pārbaude neko nemaina).

Pēc cikla papildus cikla invariantam izpildās arī beigu nosacījums.

Tālāk tiks stāstīts, kā cikla invariantu var noskaidrot, pārbaudīt un kā pēc tam izmantot programmas pareizības noteikšanā.



Att. 7.9. *While* ciklam raksturīgie apgalvojumi par programmu

Cikla pareizības noteikšana ar cikla invariantu būtu veicama šādos soļos:

- Cikla invarianta izdomāšana.
- Pierādīšana, ka dotais apgalvojums ir cikla invariants, izmantojot matemātisko indukciju:
  - indukcijas bāze,
  - indukcijas solis.
- Cikla beigu nosacījuma un cikla invarianta apvienošana, kam būtu jānoved pie beigu nosacījuma (*postcondition*), kas arī ir kritērijs programmas pareizībai.

Ja pēdējie divi soļi uzskatāmi par zināma veida tehniku, tad pirmais solis (cikla invarianta izdomāšana) ir visai radoša darbība. Turklāt, programmas pārbaudē ir jēga tikai tādam cikla invariantam, kurā ir iekļauti tādi mainīgie, kas ir ietverti arī cikla nosacījumā, tādējādi ietekmējot cikla noslēgšanos.

7.5.2. Piemērs

Pierādīsim vienkāršas programmas pareizību, izmantojot cikla invariantu:

```
PRECONDITION: n≥0 i=0 s=m
SUM (m,n)
WHILE i ≠ n DO
     s ← s + 1
     i ← i + 1
RETURN s
POSTCONDITION: s = m+n
```

Att. 7.10. Algoritms, kura pareizība ir jāpierāda. Divu skaitļu summas iegūšana, izmantojot ciklu un skaitļa 1 pieskaitīšanu, bet neizmantojot pieskaitīšanu lielākām vērtībām

Uzdevums. Pierādīt, ka pēc cikla izpildes izpildīsies beigu nosacījums (ja pirms cikla ir spēkā sākuma nosacījums).

1. Cikla invarianta izdomāšana.

Cikla invariants var būt

$s = m + i$

2. Cikla invarianta pierādīšana.

Cikla invarianta pareizības pierādīšanā tiek lietota matemātiskā indukcija, kas ir tāda pierādīšanas tehnika, kas ir saistīta ar atkārtotu darbību izpildi.

2.1. Indukcijas bāze (inicializācija)

Indukcijas bāzes uzdevums – pierādīt, ka nosacījums ir spēkā sākumā.

Pirms pirmās iterācijas ir:

$s = m$
$i = 0$

tādējādi

$s = m + i$

2.2. Indukcijas solis (uzturēšana)

Indukcijas soļa uzdevums – pieņemot, ka iepriekšējā solī izpildās kāds nosacījums, pierādīt, ka šis nosacījums izpildās arī nākošajā solī.

Ar i' un s' apzīmēsim šo mainīgo vērtības iepriekšējā iterācijā.

Pieņemsim, ka

$s' = m + i'$

Jāpierāda, ka

$s = m + i$

Pierādījums. Ievērojiet, ka $s = s' + 1$ un $i = i' + 1$

$s' = m + i'$
$s' + 1 = m + i' + 1$ *Abās pusēs pieskaita 1*
$s = m + i$ *Atbilstoši darbībām cikla ķermenī $s = s' + 1$ un $i = i' + 1$*

Tātad, esam pierādījuši, ka tas ir cikla invariants.

3. Noslēgums.

Kad cikls tiek pabeigts, tad

$i = n$

Tas kopā ar cikla invariantu dod beigu nosacījumu (*postcondition*), t.i.

Ja

```
i = n & s = m + i
```

tad

```
s = m + n
```

Tātad, esam pierādījuši, ka, ja ir spēkā sākuma nosacījums (*precondition*), tad pēc cikla izpildes izpildās arī beigu nosacījums (*postcondition*).

Piezīme. Ir svarīgi, ka sākuma nosacījumā ir $n \geq 0$, jo savādāk cikla izpilde nekad neapstāsies.

8. Masīvi

8.1. Masīvs C++

Ciklu konstrukcijas dod iespējas ērti pierakstīt programmā atkārtotas darbības. Masīvi, savukārt, ir tie, kas nodrošina "atkārtotu datu" pierakstīšanu.

> **Masīvs** (*array*) ir vienāda tipa mainīgo virkne.

Valodā C++ masīvs ir zemāka līmeņa konstrukcija nekā vairumā programmēšanas valodu, tāpēc ar masīvu mēdz saistīt arī atmiņas apgabalu.

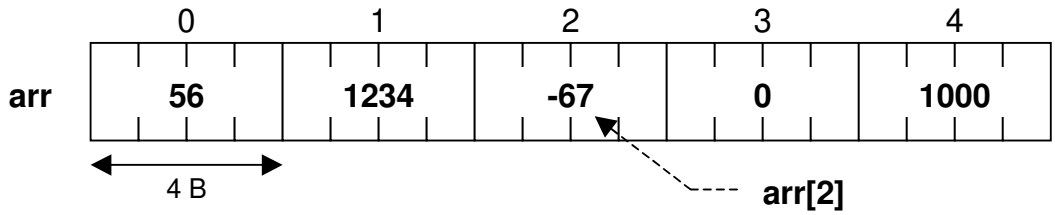
> **Masīvs** ir indeksēts atmiņas apgabals no fiksēta garuma elementiem.

Atsevišķu masīva mainīgo sauc par **elementu**. Katru elementu identificē **indekss** – skaitlis, kas norāda, pie kura no elementiem vēršas lietotājs.

Tādējādi, vienu masīva elementu identificē:

- masīva vārds,
- indekss.

Masīva indeksācija notiek pēc kārtas, bet valodā C++ – vienmēr, sākot ar skaitli 0. Tādējādi, masīvā ar garumu n tā elementi tiek indeksēti intervālā 0..n-1.



Att. 8.1. Masīva piemērs. Veselu skaitļu (*int*) masīvs *arr* garumā 5

Kaut arī masīva elementi operatīvajā atmiņā fiziski tiek glabāti kopā pēc kārtas, tomēr no datu apstrādes viedokļa, izmantojot tradicionālo masīva notāciju, tas būtu uzskatāms vien kā atsevišķu elementu kopums, nevis vienots veselums – vienīgā darbība, kuru var veikt ar masīvu kā ar vienotu veselumu, ir masīva elementu inicializācija pie deklarēšanas.

Masīva notācija nodrošina iespēju vērsties pie viena elementa pēc tā indeksa, izmantojot kvadrātiekavu operatoru:

array_element:
array_name [*element_index*]

element_index:
expression

Att. 8.2. Masīva elements (*array_element*)

Aprakstītā sintakse piekļūšanai pie elementa domāta gan vērtības nolasīšanai, gan ierakstīšanai.

Lūk, daži piemēri, kā piekļūt pie masīva *arr* elementiem (Att. 2.2):

```
arr[0]                  // 56
arr[2]                  // -67
int a = 3;
arr[a*2-2]              // 1000
arr[3] = 8              // elementam piešķir vērtību
arr[3]++                // elementa vērtību palielina par 1
arr[arr[4]%3] = 9       // elementam arr[1] piešķir vērtību 9
```

Lai strādātu ar masīvu, tas vispirms ir jāizveido, nosakot elementu skaitu masīvā. Atkarībā no tā, kādā atmiņā glabājas masīvs un kādā veidā tas ir izveidots, izšķir 2 veidu masīvus:

- statisks masīvs,
- dinamisks masīvs.

Kaut arī izveidošana abiem masīviem atšķiras, tomēr izmantošana (t.i. piekļuve elementiem, izmantojot masīva notāciju) ir identiska.

8.2. Statisks masīvs

Visvienkāršākais masīva veids ir **statisks masīvs** (*static array*) – tāds, kura elementu skaits ir zināms kompilēšanas (programmas rakstīšanas) brīdī.

Lai izveidotu statisku masīvu, pietiek to deklarēt, papildus uzrādot tā elementu skaitu (kā konstantu vērtību). Tiks apskatīti 3 veidi, kā var deklarēt statisku masīvu:

- vienkāršotais variants,
- tiešais variants ar vērtību inicializāciju,
- netiešais variants ar vērtību inicializāciju.

Pirmais variants ir otrā varianta apakšgadījums, un atsevišķi izdalīts tikai tādēļ, lai atvieglotu sākotnējo izpratni par masīva deklarēšanu.

static_array_declarator_simplified:
data_type array_name [*array_size*];

array_size:
constant positive integer value

Att. 8.3. Statiska masīva deklarēšana vienkāršoti (*static_array_declarator_simplified*)

Šādi deklarējot masīvu, pēc tam katrs elements, lai to izmantotu, jāaizpilda ar datiem atsevišķi – pa vienam.

```
int arr[5];           // deklarē masīvu garumā 5
arr[0] = 56;          // aizpilda masīva elementu
```

Masīva elementu skaitam nav obligāti jābūt literālim, tas var tikt uzdots arī ar konstantā mainīgā palīdzību (tas ir mainīgais ar papildus *const* modifikatoru, kas nozīmē, ka pēc inicializācijas to vairs nedrīkst mainīt).

```
const int arr_size = 5;
int arr[arr_size];    // deklarē masīvu garumā 5
```

Dažu kompilatoru realizācijas pieļauj statiska masīva izmēra uzdošanu ar mainīgā palīdzību (piemēram, *gcc*, *g++*), tomēr tas neatbilst C++ standartam.

Deklarējot masīvu, tiek izdalīta atmiņa masīva elementiem, tomēr masīvs automātiski netiek inicializēts. Masīvu pa vienam elementam var inicializēt vēlāk, tomēr to ir iespējams izdarīt arī visam masīvam uzreiz – deklarācijas brīdī. Tālāk dota statiska masīva deklarēšana ar inicializāciju:

static_array_declarator:
data_type array_name [*array_size*] $array\_init_{opt}$;

array_init:
= { *value_sequence* }

value_sequence:
value
value , *value_sequence*

Att. 8.4. Statiska masīva deklarēšana tiešajā variantā (*static_array_declarator*)

Inicializācijas vērtību virkne pēc izmēra **nedrīkst pārsniegt** masīva izmēru, tomēr drīkst būt mazāka:

```
int arr[5] = {56,1234,-67,0,1000};
int arr[5] = {56,1234,-67};
```

Trešais veids, kā deklarēt statisku masīvu, ir atstāt tukšas kvadrātiekavas un neuzrādīt masīva izmēru tiešā veidā, bet ļaut to netieši izdarīt inicializācijas vērtību virknei:

static_array_declarator2:
data_type array_name [ ] *array_init* ;

Att. 8.5. Statiska masīva deklarēšana netiešajā variantā (static_array_declarator2)

Nākošais piemērs netiešā veidā deklarē statisku masīvu garumā 7, to inicializējot:

```
int arr2[] = {11,22,33,44,55,66,0};
```

Nākošais piemērs parāda masīva izveidošanu, inicializēšanu, otra masīva izveidošanu, pārrakstīšanu otrā masīvā un otra masīva izdrukāšanu:

```
const int arr_size = 5;
int arr[arr_size] = {56,1234,-67,0,1000};
int arr2[arr_size];
for (int i=0; i<arr_size; i++) arr2[i] = arr[i];
for (int i=0; i<arr_size; i++) cout << arr2[i] << " ";
```

```
56 1234 -67 0 1000
```

Šis piemērs demonstrē tipisku masīva pārstaigāšanas tehniku – *for* ciklu no 0 līdz masīva beigām. Ievērojiet tajā vienu niansi: ņemot vērā, ka masīva beidzamā elementa numurs ir *arr_size-1*, tad *for* cikla nosacījums varētu izskatīties arī šādi, kas ir sintaktiski pareizi:

```
i<=arr_size-1
```

Tomēr praksē būtu vēlams izvēlēties variantu, kas parādīts piemērā, resp.,

```
i<arr_size
```

jo šis variants satur masīva izmēru (nevis par masīva izmēru par 1 mazāku vērtību) salīdzināšanas operācijas vienā pusē, un pieredze rāda, ka tas palīdz izvairīties no loģiskām divdomībām un tādējādi neuzmanības kļūdām.

8.3. Dinamisks masīvs

Dinamiska masīva (*dynamic array*) elementu skaits nav jāzina programmas rakstīšanas brīdī. Dinamisks masīvs ļauj ietaupīt programmā izmantojamo atmiņu. Tā ir liela priekšrocība, tomēr tādēļ ir jāpacieš zināmas "neērtības" – darbojoties ar dinamisko masīvu (salīdzinot ar statisko masīvu), ir jāveic divas papildus darbības:

- atmiņas izdalīšana dinamiskajam masīvam pirms darba ar to (izveidošana),
- atmiņas atbrīvošana darba beigās (iznīcināšana).

Statiskajiem masīviem šīs abas darbības notiek automātiski: izveidošana deklarēšanas brīdī, bet iznīcināšana – tā bloka beigās, kurā masīvs ticis deklarēts.

```
int arr_size;          // masīva izmērs
cin >> arr_size;       // var nebūt konstanta vērtība
int *arr;              // masīva mainīgā deklarēšana
arr = new int [arr_size];    // masīva izveidošana
    // masīva izmantošana
delete[] arr;          // masīva iznīcināšana
```

Atšķirībā no statiska masīva deklarēšanas, dinamiska masīva izveidošanā masīva izmēram (*array size*) nav jābūt konstantei, bet tas var būt mainīgais vai izteiksme – tur arī izpaužas masīva dinamisms.

Masīva mainīgā deklarēšanu un masīva izveidošanu:

```
int *arr;             // masīva mainīgā deklarēšana
arr = new int [size]; // masīva izveidošana
```

var apvienot vienā rindā:

```
int *arr = new int [size];
```

Dinamiska masīva mainīgā deklarēšanu un masīva izveidošanu raksturo sekojošā sintaktiskā shēma:

dynamic_array_declaration_and_creation:
dynamic_array_var_decl *dynamic_array_creation*$_{opt}$;

dynamic_array_var_decl:
data_type * *array_name*

dynamic_array_creation:
= new *data_type* [*expression*]

Att. 8.6. Dinamiska masīva mainīgā deklarēšana un masīva izveidošana (*dynamic_array_declaration_and_creation*)

Dinamiskā masīva izveidošana atsevišķi – ne tajā rindā, kur bijusi masīva mainīgā deklarēšana:

dynamic_array_creation_apart:
array_name *dynamic_array_creation* ;

Att. 8.7. Dinamiska masīva izveidošana atsevišķi (*dynamic_array_creation_apart*)

Tā kā dinamiskajam masīvam atmiņa tikusi izdalīta manuāli (nevis automātiski), arī iznīcināšana jāveic manuāli ar operatora *delete* palīdzību:

dynamic_array_deletion:
delete[] *array_name* ;

Att. 8.8. Dinamiska masīva iznīcināšana (*dynamic_array_deletion*)

Ja dinamiska masīva izmantošanai programmā neseko iznīcināšana, tas tiešā veidā neietekmē programmas darbību, tomēr nekontrolēti tērē sistēmas atmiņas resursus – rezervētie, bet neatbrīvotie atmiņas apgabali pēc programmas bloka darbības beigām paliek nepieejami programmai un tai pat laikā arī operētājsistēmai kopumā, jo tā tos uzskata par aizņemtiem.

Apkopojot iepriekš aplūkoto, darbojoties ar dinamisku masīvu, var izdalīt 4 šādus posmus:

- masīva mainīgā deklarēšana,
- masīva izveidošana, izmantojot operatoru *new*,
- masīva izmantošana – identiski statiskam masīvam,
- masīva iznīcināšana, izmantojot operatoru *delete*.

Atšķirībā no statiska masīva, dinamiskajam masīvam mainīgā deklarēšana un masīva izveidošana var notikt kā divas atsevišķas darbības. Tādējādi sanāk, ka kādu laiku programmā var eksistēt masīva mainīgais, kuram neatbilst nekāds masīvs. Tā tas tiešām ir, un, atgādinot, ka valodā C++ masīvs ir salīdzinoši zema līmeņa jēdziens, būtu jāuzsver, ka valodā C++ **masīva mainīgais un masīvs ir divas dažādas lietas**.

Ja parastā mainīgā gadījumā tas parasti reprezentē noteiktu vērtību, kas tajā glabājas, tad masīva mainīgais drīzāk reprezentē vietu, kur atmiņā masīvs sākas, nevis pašu masīvu.

8.4. Masīva izmantošanas īpatnības

8.4.1. Vienkāršākās darbības ar masīvu kā vienotu veselumu

Masīvs no izmantošanas viedokļa uzskatāms kā atsevišķu elementu kopums, nevis vienots veselums, tāpēc, darbojoties ar to, jāstrādā ar katru elementu atsevišķi. Tas jāņem vērā tādās darbībās kā masīvu aizpildīšana, salīdzināšana un garuma noteikšana. Tīri tehniski tālāk aprakstītās "grūtības" ir tādēļ, ka, kā jau iepriekš tika minēts: **masīva mainīgais un masīvs ir divas dažādas lietas**.

8.4.1.1. Masīva aizpildīšana

Vienu masīvu nevar piešķirt otram masīvam, piemērojot piešķiršanas operatoru masīva mainīgajiem:

```
arr2 = arr1;    // tā nedrīkst piešķirt vienu masīvu otram
```

Masīva aizpildīšana jāveic kādā no šiem diviem variantiem, kā tas iepriekš jau arī ir parādīts:

- aizpildot pa vienam elementam,
- veicot masīva inicializāciju pie deklarēšanas.

Augstāk parādītā piešķiršanas operācija daudzos gadījumos nostrādās, bet tā neveiks viena masīva satura pārrakstīšanu otrā.

8.4.1.2. Masīvu salīdzināšana

Divus masīvus nevar salīdzināt, piemērojot salīdzināšanas operatoru masīva mainīgajiem:

```
(arr2 == arr1)    // tā nedrīkst salīdzināt masīvu saturu
```

Salīdzināšana jāveic **pa vienam elementam**.

Augstāk parādītā salīdzināšanas operācija veiks divu masīva mainīgo salīdzināšanu, bet ne masīvu saturu salīdzināšanu, kurus reprezentē dotie mainīgie.

8.4.1.3. Masīva garuma noteikšana

Masīva garuma noteikšana, izmantojot pašu masīvu, vispārīgā gadījumā nav iespējama. Varētu teikt, ka masīvs "nezina savu garumu". Tas ir tāpēc, ka, kā jau ir ticis minēts, valodā C++ masīvs ir salīdzinoši zema līmeņa konstrukcija.

Vienīgais izņēmums ir statiskie masīvi, kuru garumu iespējams noskaidrot ar funkciju *sizeof*, rezultātu izdalot ar viena elementa garumu baitos:

```
int arr[5];
sizeof (arr) / sizeof (int)    // atgriež 5
```

Tomēr vispārīgā gadījumā programmā būtu jātur speciāls mainīgais vai konstante, kas glabā masīva garumu.

8.4.2. Masīvu apstrādes funkcijas

Valodā C++ masīvu kaut kādā nozīmē nevar uzskatīt par vienotu veselumu, un masīvā katrs elements jāapstrādā atsevišķi. Tomēr, ja mēs masīvu uztveram kā atmiņas apgabalu, t.i., kā baitu virkni, bet darbības ar masīvu kā darbības ar atmiņas apgabaliem, tad tam eksistē standarta zema līmeņa funkciju komplekts, kuru nosaukumi sākas ar "*mem...*".

Masīvu (atmiņas apgabalu) apstrādes funkcijas (*memcpy()*, *memcmp()*, *memchr()*) darbojas ar atmiņas apgabaliem pa baitam, turklāt pieprasa padot masīvu ar mainīgo, pārveidotu uz tipu *void**. Tehniski tas izskatās tā, ka pirms masīva mainīgā, kuru padod funkcijai caur parametru, papildus jāpieliek tipa pārveidotājs *(void*)*. Tā kā darbošanās ar masīvu notiek pa baitam, tad apstrādājamā masīva apjoms (sk. zemāk pie konkrētām funkcijām) jāuzrāda baitos. Piemēram, ja es vēlos apstrādāt *int* masīvā 5 elementus, man jāuzrāda attiecīgajai funkcijai, ka vēlos apstrādāt 20 baitus, jo *int* ietver 4 baitus.

8.4.2.1. Masīva satura kopēšana

memcpy

```
void* memcpy (void *to, const void *from, size_t count);
```

Funkcija *memcpy()* kopē *count* simbolus no masīva *from* uz masīvu *to*. Atgriež norādes *to* vērtību. Ja masīvi pārklājas, funkcijas darbība ir nedefinēta.

Nākošais piemērs parāda funkcijas *memcpy* lietojumu, kad masīva *arr1* saturs tiek pārkopēts uz masīvu *arr2*.

```
int arr1[3] = {11, 22, 33};
int arr2[3];
memcpy ((void*)arr2, (void*)arr1, 3*sizeof(int));
```

8.4.2.2. Divu masīvu satura salīdzināšana

memcmp

```
int memcmp (const void *buf1, const void *buf2, size_t count);
```

Funkcija *memcmp()* salīdzina pirmos *count* simbolus starp masīviem *buf1* un *buf2*, atgriežot veselu skaitli:

- 0, ja masīvi vienādi;

- <0, ja *buf1* mazāks par *buf2*;
- >0, ja *buf1* lielāks par *buf2*.

Piezīme. Ievērojiet, ka funkcijas *memcmp* atgrieztās vērtības būtība ir pretēja loģisku vērtību interpretācijai C++ (loģisku vērtību gadījumā 0 nozīmēja *false*).

Nākošajā piemērā *memcmp* atgriezīs 0, jo abi masīvi ir ar vienādu saturu.

```
int arr1[3] = {11, 22, 33};
int arr2[3] = {11, 22, 33};
memcmp ((void*)arr1, (void*)arr2, 3*sizeof(int));
```

8.4.2.3. Simbola meklēšana masīvā

memchr

```
void* memchr (void *buf, int c, size_t count);
```

Funkcija *memchr()* meklē pirmo sastapto simbolu *c* masīva *buf* pirmajos *count* simbolos. Funkcija atgriež norādi uz atrasto simbolu vai arī nulles norādi, ja simbols masīvā nav atrasts.

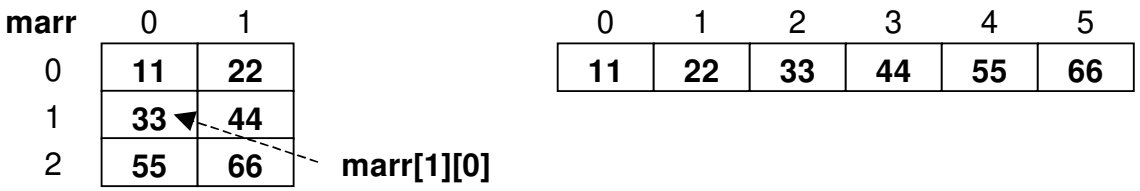
Nākošajā programmā funkcija *memchr* meklē burtu ‘s’ simbolu masīva *arr* pirmajos 8 baitos. Programma izdrukā 5, jo pirmais burts ‘s’ atrodas 5. pozīcijā (numerācija sākas ar 0). Lai saprastu, ko nozīmē jēdziens **norāde**, un, kā tieši tiek izrēķināta vajadzīgā vērtība 5, ir jāpārzina adresu aritmētika, kas tiks apskatīta nākošajās nodaļās.

```
char c;
char arr[8] = {'p','i','a','n','i','s','t','s'};
void *p = memchr (arr, 's', 8);
cout << (char*)p - arr << endl;
```

```
5
```

8.5. Vairākdimensiju masīvi

Vairākdimensiju masīvi ir tādi masīvi, kuros elementus identificē **divi vai vairāki indeksi**. Kaut arī jebkuru vairākdimensiju masīvu var izteikt kā viendimensijas masīvu, tomēr vairākdimensiju masīvu izmantošana var būtiski uzlabot programmas pārskatāmību.



Att. 8.9. Divu dimensiju masīvs (3x2) un tā reālais izvietojums atmiņā

Pievienojot jēdzienu par vairākdimensiju masīviem, vēršanās pie masīva elementa sintakse aprakstāma šādi:

array_element_multi:
array_name array_index_sequence

array_index_sequence:
[*element_index*]
[*element_index*] *array_index_sequence*

Att. 8.10. Masīva elements (vairākdimesiju) (*array_element_multi*)

Piemēram:

```
marr[1][0]            // 33 (sk. attēlu augstāk)
```

Arī vairākdimensiju masīvu iespējams inicializēt pie deklarēšanas, turklāt uzskatāmībai iespējams izmantot papildus strukturēšanu blokos, tomēr tas nav obligāti.

```
int marr[3][2] = {{11,22},{33,44},{55,66}};
```

vai

```
int marr[3][2] = {11,222,333,44,555,66};
```

Vairākdimensiju masīva deklarēšanas sintakse ir pietiekoši sarežģīta. Nākošā sintaktiskā shēma apraksta nedaudz vienkāršotu vairākdimensiju masīva deklarēšanas sintaksi.

array_declarator_multi:
data_type array_name array_dimensions array_init_multi$_{opt}$;

array_dimensions:
[*dimension_size*]
[*dimension_size*] *array_dimensions*

dimension_size:
constant positive integer value

array_init_multi:
= *enclosed_array_init_sequence*

enclosed_array_init_sequence:
{ *array_init_sequence* }

array_init_sequence:
array_init_value
array_init_sequence , *array_init_value*

array_init_value:
value
enclosed_array_init_sequence

Att. 8.11. Statiska vairākdimensiju masīva deklarēšana (*array_declarator_multi*)

8.6. Vairāku masīvu kopēja deklarēšana

Vairākus masīvus, kā arī attiecīgā tipa mainīgos iespējams deklarēt ar vienu kopēju deklaratoru. To parāda sekojošie piemēri.

Tā vietā, lai rakstītu vairākas rindas:

```
int a;
int b=0;
int arr1[3];
int arr2[] = {44,55};
```

var izmantot vienu vienīgu kopējo deklaratoru:

```
int a, b=0, arr1[3], arr2[] = {44,55};
```

9. Simboli un simbolu virknes

9.1. Simbols – teksta glabāšanas pamatvienība

9.1.1. Simbols un tā kods

Viena simbola (*character*) glabāšanu atmiņā nodrošina datu tips *char*. Tas aizņem vienu baitu un nodrošina 256 dažādu vērtību glabāšanu.

char vērtībai atbilst simbola literālis – simbols, kas ielikts vienkāršajās pēdiņās (apostrofos).

No informācijas apstrādes viedokļa tips *char* ir tas pats *int*, tikai vienā baitā, jo atmiņā simbolus reprezentē to kodi. atšķirība no *int* tipa tipam *char* izpaužas tikai ievadē un izvadē, kad vērtība tiek interpretēta vai formatēta par simbolu.

Tā kā *char* no informācijas apstrādes viedokļa ir tas pats *int*, tad atšķirībā no citām programmēšanas valodām, valodā C++, lai no simbola iegūtu tā kodu vai otrādi, nav jāpielieto nekādas funkcijas – sliktākajā gadījumā jāpiemēro tipu pārveidošana.

```
char c = 'A';
int i = c;
cout << c << endl;
cout << i << endl;
cout << (char)i << endl;
```

```
A
65
A
```

char tipa vērtībām piemērojama tā pati aritmētika, kas *int* vērtībām:

```
char c = 97;
cout << c << endl;
c++;
cout << c << endl;
```

```
a
b
```

9.1.2. ASCII kodu tabula simbolu kodēšanai

C++ standarts neparedz, vai *char* tipā glabātie simbolu kodi ir robežās 0..255 vai –128..+127, tomēr parasti kompilatoros ir realizēts otrs variants. Ja ir vēlēšanās, lai pilnīgi noteikti atmiņā tiktu glabātas vērtības kādā no šiem intervāliem, tad attiecīgi jālieto tipi *unsigned char* vai *signed char*. Tomēr, ja notiek darbs ar tekstu, kurš rakstīts latīņu alfabētā un ar papildus simboliem, kas pieejami uz klaviatūras tiešā veidā, tad par šo problēmu nav jāuztraucas, jo šie simboli ir kodēti robežās 0..127.

Pēc augšējiem piemēriem jau varēja redzēt, ka lielajam A atbilst kods 65, bet mazajiem a un b attiecīgi 97 un 98.

Tā ir ASCII kodu tabula, kas ir standartizēta visā pasaulē. Nākošajā tabulā parādīti svarīgāko ASCII simbolu vai to intervālu kodi.

Tab. 9.1.

Simbolu ASCII kodi

Simbols vai simbolu intervāls	Kods vai kodu intervāls
‘A’..‘Z’	65..90
‘a’..‘z’	97..122
‘0’..‘9’	48..57
‘ ’ (tukšums)	32
‘\n’ (jauna rindiņa)	10
‘\0’ (tukšais simbols)	0
!	33
"	34
#	35
$	36
%	37
&	38
'	39
(	40
)	41
*	42
+	43
,	44
-	45
.	46
/	47
:	58
;	59
<	60
=	61
>	62
?	63
@	64
{	123
|	124
}	125
~	126

Nedrīkst sajaukt simbolisko pierakstu ar skaitlisko – tas ir būtiski tieši attiecībā uz cipariem, piemēram, jāsaprot, ka simboliskais ‘9’ ir pavisam cita vērtība nekā skaitliskais 9.

Kā redzams pēc tabulas:

- visi lielie latīņu burti ir pēc kārtas,
- visi mazie latīņu burti ir pēc kārtas,
- starp lielo burtu un attiecīgo mazo burtu ir fiksēts attālums – 32,
- visi cipari ir pēc kārtas.

Piemēram, lai noskaidrotu, ka simbols mainīgajā *c* ir lielais latīņu alfabēta burts, der nosacījums:

```
c>='A' && c<='Z'
```

vai

```
c>=65 && c<=90
```

Lai noskaidrotu, ka simbols mainīgajā *c* ir cipars, der nosacījums:

```
c>='0' && c<='9'
```

vai

```
c>=48 && c<=57
```

Lai pārveidotu mazo burtu par lielo, pietiek atņemt 32:

```
if (c>=97 && c<=122) c-=32
```

vai

```
if (c>='a' && c<='z') c-=('a'-'A')
```

9.1.3. Standarta funkcijas simbolu apstrādei

Standarta bibliotēka <cctype> piedāvā funkciju klāstu simbolu apstrādei.

isalpha

```
int isalpha (int ch);
```

Funkcija *isalpha()* atgriež no nulles atšķirīgu vērtību, ja padotais arguments *ch* ir burts (latīņu lielais vai mazais).

```
isalpha('A')   // atgriež 1
isalpha('1')   // atgriež 0
```

isdigit

```
int isdigit (int ch);
```

Funkcija *isdigit()* atgriež no nulles atšķirīgu vērtību, ja padotais arguments *ch* ir cipars.

```
isdigit('A')   // atgriež 0
isdigit('1')   // atgriež 1
```

isalnum

```
int isalnum (int ch);
```

Funkcija *isalnum()* atgriež no nulles atšķirīgu vērtību, ja padotais arguments *ch* ir vai nu burts, vai cipars.

```
isalnum('A')   // atgriež 1
isalnum('1')   // atgriež 1
isalnum('#')   // atgriež 0
```

islower

```
int islower (int ch);
```

Funkcija *islower()* atgriež no nulles atšķirīgu vērtību, ja padotais arguments *ch* ir mazais burts ‘a’.. ‘z’.

```
islower('a')   // atgriež 1
islower('#')   // atgriež 0
```

isupper

```
int isupper (int ch);
```

Funkcija *isupper()* atgriež no nulles atšķirīgu vērtību, ja padotais arguments *ch* ir lielais burts ‘a’.. ‘z’.

```
isupper('A')   // atgriež 1
isupper('#')   // atgriež 0
```

tolower

```
int tolower (int ch);
```

Funkcija *tolower()* atgriež argumentu *ch*, pārveidotu par mazo burtu, ja tas ir lielais. Citādi atstāj neizmainītu.

```
tolower('A')   // atgriež 'a'
tolower('#')   // atgriež '#'
```

toupper

```
int toupper (int ch);
```

Funkcija *toupper()* atgriež argumentu *ch*, pārveidotu par lielo burtu, ja tas ir mazais. Citādi atstāj neizmainītu.

```
toupper('a')   // atgriež 'A'
toupper('#')   // atgriež '#'
```

9.2. Simbolu virkne – līdzeklis teksta informācijas glabāšanai

Simbolu virknes (*string*) ir ļoti svarīga katras programmēšanas valodas sastāvdaļa, jo nodrošina darbu ar teksta informāciju.

> **Simbolu virkne** (*string*) ir simboliskā tipa (piemēram, *char*) vērtību secība, kas reprezentē tekstuālu informāciju.

Lielākā daļa moderno universālo programmēšanas valodu nodrošina t.s. augsta līmeņa simbolu virknes, t.i., tādas, kas ietver sevī arī atmiņas vadību, līdz ar ko darbs ar simbolu virknēm ir ērts, drošs un intuitīvi saprotams. Valodā C++ ir pieejamas divu veidu simbolu virknes:

- **zema līmeņa** (C stila) simbolu virknes (*char**).
- **augsta līmeņa** (C++ stila) simbolu virknes (*string*).

Kaut arī valodā C++ augsta līmeņa simbolu virknes ir pieejamas – tieši tāpat kā citās programmēšanas valodās, tomēr galvenais un noklusētais simbolu virkņu veids ir tieši zema līmeņa simbolu virknes. Tam ir šādi iemesli:

- Valoda C++ ir valodas C paplašinājums, kurā bija pieejamas tikai zema līmeņa simbolu virknes, tādēļ, lai nodrošinātu savietojamību starp valodām daudzu standarta iespēju izmantošanai, arī C++ noklusētajā variantā izmanto zema līmeņa virknes.
- Valoda C++, paplašinot valodu C, neatteicās no maksimālas efektivitātes nodrošināšanas programmas – un zema līmeņa simbolu virknes dod šādu iespēju.
- augsta līmeņa simbolu virknes pašā valodā C++ standartizētas salīdzinoši vēlu (ilgi pēc tās rašanās), tāpēc kādu laiku C++ bija pieejamas tikai zema līmeņa simbolu virknes.

Vienošanās. Turpmāk, ja netiks precizēts, par kurām simbolu virknēm ir runa un tas nebūs skaidrs no konteksta, uzskatīt, ka tiek domātas zema līmeņa simbolu virknes.

9.2.1. Zema līmeņa simbolu virknes

9.2.1.1. Zema līmeņa simbolu virkņu uzbūves principi

Valodā C++ zema līmeņa simbolu virknes ir simbolu virkņu noklusētais variants. Ikviens teksta literālis, piemēram,

```
"Hello, World!"
```

tiek automātiski interpretēts kā zema līmeņa simbolu virkne.

> **Zema līmeņa (C stila) simbolu virkne** ir simboliskā tipa vērtību secība, kuru noslēdz speciāls simbolu virknes beigu simbols.

Zema līmeņa simbolu virknes datu tips ir

```
char*
```

Zvaigznīte * valodā C++ apzīmē virkni vai norādi informācijas bloka sākumu (par to tālākajās nodaļās).

Simbolu virknes beigu simbola (jeb **nulles simbola**, *'\0'* vai vienkārši vērtība 0) esamība ir **obligāts nosacījums**, lai simbolu secību varētu nosaukt par simbolu virkni.

Tā kā zema līmeņa simbolu virkne satur papildus nulles simbolu, tad simbolu virknes, kas satur tekstu garumā *n*, glabāšanai nepieciešamā atmiņa ir *n+1* vienības.

Zema līmeņa simbola virkni parasti glabā *char* masīvā, tāpēc zema līmeņa simbolu virkni var tehniski definēt kā masīva speciālgadījumu.

> **Zema līmeņa (C stila) simbolu virkne** ir tāds masīvs, kuru raksturo divas papildus īpašības:
>
> – datu tips, no kura sastāv masīvs, ir simbols (*char*),
>
> – vismaz viens no masīva simboliem ir t.s. simbolu virknes beigu simbols.

Pirmā no īpašībām nodrošina to, ka simbolu virknēs var glabāt teksta informāciju, bet otrā to, ka neatkarīgi no deklarētā (izveidotā) masīva garuma tajā var ērti glabāt dažādu garumu simbolu virknes (tiesa gan, nepārsniedzot masīva garumu - 1). Bez tam simbolu virknes beigu simbols palīdz uztvert simbolu virkni kā vienotu veselumu (atšķirībā no parasta masīva), ko nodrošina daudzas C++ standarta funkcijas un citas iespējas, kas spēj strādāt ar simbolu virknēm.

Šāda simbolu virknes definīcija ar masīvu ir praktiskāk saprotama, tomēr teorētiski neprecīza, jo:

- simbolu virknes sākumam nav obligāti jāsakrīt ar masīva sākumu,
- masīvā var glabāt vairāk nekā vienu simbolu virkni.

Tālākajos piemēros simbolu virknes un masīva, kurā tā glabājas, jēdzieni saplūdīs.

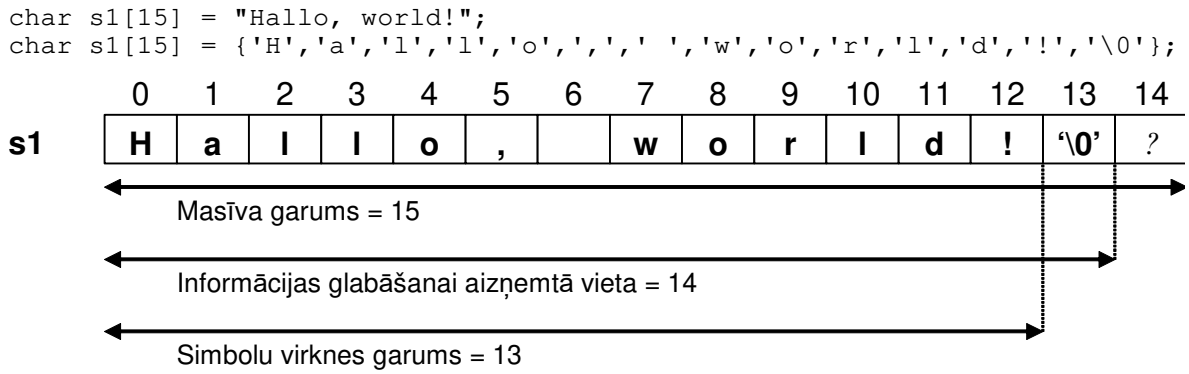
Neatkarīgi no masīva tipa, kas tiek lietots simbolu virknes glabāšanai (statisks, dinamisks), un no deklarēšanas veida (sintakses, sk. masīva deklarēšanu), simbolu virknes datu tips, kā jau tika pieminēts, ir *char**.

Simbolu virkņu (ja tās uztver par masīvu speciālgadījumu) deklarēšana un izmantošana ir veicama tāpat kā parastiem masīviem, jo simbolu virknes pašas savā ziņā ir masīvi.

Konstantu teksta informāciju, kas atbilst simbolu virknes datu tipam, var uzdot, ievietojot dubultpēdiņās, piemēram, "simboli".

Tukša simbolu virkne ("") ir tāda, kura sākas nulles simbolu (s[0] == '\0').

Nedrīkst sajaukt jēdzienus: simbols (piemēram, 'a') un simbolu virkne garumā 1 (piemēram, "a"), kaut arī tie izdrukājas identiski – simbolu virkne automātiski aiz sevis ietver arī beigu simbolu ("a" ir {'a', '\0'}).



Att. 9.1. Simbolu virkne "Hallo, world!", kas tiek glabāta masīvā ar garumu 15

Masīvā aiz simbolu virknes beigu simbola var atrasties informācija, tomēr no simbolu virknes viedokļa tai nav nozīmes (sk. elementu nr. 14).

9.2.1.2. Zema līmeņa simbolu virkņu apstrāde

Simbolu virkņu aizpildīšana.

Aizpildot simbolu virkni, programmētājs pats ir atbildīgs par 2 lietām:

- masīvā, kurā tiek ierakstīta simbolu virkne, ir jābūt pietiekoši daudz vietas,
- jānodrošina simbolu virknes beigu simbols (jāpieliek pašam vai jālieto konstrukcijas, kas to pieliek automātiski).

Simbolu virkni var uzdot, inicializējot masīvu pie deklarēšanas, turklāt to var darīt gan pa vienam simbolam, gan, izmantojot teksta literāli ar dubultpēdiņām:

```
char s1[10] = {'H','a','l','l','o','!','\0'};
char s2[10] = "Hallo!";
```

Abi piemēri ir identiski. Ievērojiet, ka simbolu virknes aizpildīšana ar literāli (dubultpēdiņās), automātiski nodrošina simbolu virknes beigu simbolu.

Simbolu virkni masīvā var ievadīt no klaviatūras, izmantojot operatoru >>. Tomēr šai metodei ir viens liels trūkums: operators >> ir formatētās ievades operators, kurš tukšumus un citus atdalītājsimbolus uzskata par informācijas beigu pazīmi:

```
char s[20];
cin >> s;
cout << s;
```

```
kartupelis
kartupelis
```

```
LU is OK.
LU
```

Tāpēc zema līmeņa simbolu virknes aizpildīšanai būtu lietojama funkcija *getline*.

getline

```
istream &getline (char* s, int num);
```

Funkcija *getline()* no faila (klaviatūras) lasa simbolus masīvā *buf*, kamēr vai nu nav nolasīti *num* simboli (šādā gadījumā *cin* objekts tiek bloķēts un ir jāatbloķē ar *clear()*), vai nu tiek sastapts simbols *'\n'* (ievadīts <ENTER>). Tā kā masīvā jārezervē viena vieta simbolu virknes beigu simbolam, tad no klaviatūras nevar nolasīt vairāk nekā *num-1* simbolu – pārējie tiek ignorēti. Svarīgi, ka funkcija *getline()* nav neatkarīga, bet ir ievades objekta (piemēram, *cin*) sastāvdaļa:

```
char s[20];
cin.getline (s, 20);
cout << s;
```

```
LU is OK.
LU is OK.
```

Citas standarta funkcijas simbolu virkņu apstrādei.

Uz simbolu virknēm attiecas tās pašas izmantošanas īpatnības, kas uz masīviem kopumā – tām nevar veikt piešķiršanu, salīdzināšanu un garuma noteikšanu, izmantojot parastos operatorus, tomēr simbolu virknēm ir pieejams standarta funkciju klāsts. Tālāk uzskaitītas dažas no tām. Standarta funkcijām, kas strādā ar zema līmeņa simbolu virknēm, raksturīgs tas, ka to nosaukums sākas ar "*str*". Tās ir ietvertas bibliotēkā *<string.h>*, tomēr netiešā veidā arī daudzās citās, piemēram *<iostream>*. Modifikators *const* pie parametra norāda, ka funkcija doto parametru noteikti neizmainīs, bet *char** ir datu tips, kas apzīmē simbolu virkni (precīzāk, norādi uz simbolu virkni).

strcpy

```
char* strcpy (char* s2, const char* s1);
```

Funkcija *strcpy()* kopē simbolu virknes *s1* saturu uz virkni *s2*. Jābūt nodrošinātam, ka virknē *s2* pietiks vietas.

```
char s[20]="Hello, World!";
char t[20]="Old value";
strcpy (t, s);
cout << t;
```

```
Hello, World!
```

strcat

```
char* strcat (char* s2, const char* s1);
```

Funkcija *strcat()* pievieno simbolu virknes *s1* kopiju virknes *s2* beigās. Jābūt nodrošinātam, ka virknē *s2* pietiks vietas.

```
char s[20]="Hello, ";
char t[20]="World!";
strcat (s, t);
cout << s;
```

```
Hello, World!
```

strcmp

```
int strcmp (const char* s1, const char* s2);
```

Funkcija *strcmp()* leksikogrāfiski salīdzina virknes *s1* un *s2*, atgriežot veselu skaitli:

- 0, ja virknes vienādas;
- <0, ja *s1* mazāka par *s2*;
- >0, ja *s1* lielāka par *s2*.

```
char s[10]="Hello";
char t[10]="Hello";
char u[10]="HELLO";
cout << strcmp (s, t) << endl;
cout << strcmp (s, u) << endl;
cout << strcmp (u, s) << endl;
```

```
0
1
-1
```

strlen

```
size_t strlen (const char* s);
```

Funkcija *strlen()* atgriež simbolu virknes s garumu.

```
char s[20]="LU is OK.";
char t[20]={'H','e','l','l','o',0};
cout << s << setw(3) << strlen(s) << endl;
cout << t << setw(3) << strlen(t) << endl;
```

```
LU is OK.  9
Hello  5
```

9.2.2. Augsta līmeņa simbolu virknes

Augsta līmeņa simbolu virknes valodā C++ ir objekti (lasi, mainīgie), kas nodrošina teksta glabāšanu un apstrādi, rūpējoties par nepieciešamo atmiņas vietu virknes saglabāšanai un nodrošinot ērtu sintaksi dažādām darbībām.

Augsta līmeņa simbolu virkne izmanto bibliotēku <*string*>, bet ir iekļauta arī <*iostream*> un tās tips ir *string*.

Darbs ar augsta līmeņa virknēm notiek ērti un konstrukcijas ir intuitīvi saprotamas.

Inicializācija un izdrukāšana notiek tāpat kā zema līmeņa virknēm:

```
string s = "Hello, World!";
cout << s;
```

```
Hello, World!
```

Ievade no klaviatūras, izmantojot formatētās ievades operatoru, noved pie tām pašām problēmām – tukšumi un citi atdalītājsimboli tiek uzskatīti par beigu pazīmi:

```
string s;
cin >> s;
```

```
    cout << s;
```

```
    kartupelis
    kartupelis
```

```
    LU is OK.
    LU
```

Tāpēc jāizmanto funkcija getline, kurai augsta līmeņa simbolu virknes gadījumā ir cita sintakse:

getline

```
istream &getline (istream &cin, string &s);
```

Funkcija *getline()* no faila (klaviatūras) lasa simbolus simbolu virknē *s*, kamēr vien nav sastapts simbols *'\n'* (ievadīts <ENTER>). Svarīgi, ka funkcija *getline()* ir nevis ievades objekta *cin* iekšēja funkcija, kā tas bija gadījumā ar zema līmeņa simbolu virknēm, bet gan neatkarīga funkcija, tādēļ objekts *cin* tai jāpadod kā parametrs:

```
    string s;
    getline (cin, s);
    cout << s;
```

```
    LU is OK.
    LU is OK.
```

Daudzas darbības, kas zema līmeņa simbolu virknēm bija neērtas un potenciāli bīstamas no kļūdu ielaišanas viedokļa, šeit ir pierakstāmas īsi un eleganti.

Informācijas kopēšana no vienas simbolu virknes uz otru, kā arī no zema līmeņa simbolu virknes, iespējama ar pierasto piešķiršanas operatoru, gan objekta deklarēšanas rindā, gan pēc tam:

```
    char a[20] = "Hello, World!";
    string s;
    s = a;
    string t = s;
    cout << s;
```

```
    Hello, World!
```

Nav jārūpējas ne par beigu simbolu, ne par to, vai pietiks vietas, kur virkni glabāt.

Simbolu virknes pievienošana vienu otrai galā notiek, vai nu izmantojot operatoru +, vai nu piešķiršanas operatoru +=, atkarībā no tā, kādi starprezultāti nepieciešami:

```
    string s = "Hello, ";
    string t = "World";
    s += t;
    cout << s + "!!!";
```

```
    Hello, World!!!
```

Augsta līmeņa simbolu virknes garumu var noskaidrot, izmantojot simbolu virknes objekta funkcijas *size()* un *length()* (strādā identiski).

```
    string s = "Hello, World!";
    cout << s.size() << endl;
    cout << s.length() << endl;
```

```
13
13
```

Piekļūt pie simbolu virknes elementiem var, izmantojot pierasto masīvu sintaksi:

```
string s = "Hello, World!";
cout << s[0] << s[7] << endl;
```

```
HW
```

Augsta līmeņa simbolu virknēs ir ērti glabāt un apstrādāt informāciju, tomēr daudzas standarta funkcijas pieprasa padot zema līmeņa funkcijas kā argumentus. Šim nolūkam der simbolu virknes objekta iekšēja funkcija *c_str()*, kas atgriež konstantu norādi uz augsta līmeņa simbolu virknei atbilstošo zema līmeņa virkni:

```
string s = "Hello, World!";
strcpy (a, s.c_str());
cout << a;
```

```
Hello, World!
```

Vēl papildus nosauktajām funkcijām augsta līmeņa simbolu virknes objektā ir pieejamas ļoti daudzas funkcijas, piemēram, apakšvirkņu atrašanai utt., kuras šī materiāla ietvaros netiks apskatītas.

10. Funkcijas

10.1. Funkcija kā programmas strukturēšanas līdzeklis

Viens no svarīgiem strukturētās programmēšanas pamatprincipiem ir t.s. procedurālā abstrakcija.

> **Procedurālā abstrakcija** ir programmas strukturēšanas veids, kad programma tiek strukturēta moduļos (procedūrās) pēc "melnās kastes" (*black box*) principa, t.i., nodalot redzamo vispārīgo (deklaratīvo) daļu no paslēptās detalizētās (realizācijas) daļas.

Ideālā gadījumā katrs modulis tiek noformēts tā, ka, lai ar to strādātu, nepieciešams zināt tikai to, kādi argumenti tam jāpadod un ka tas atgriezīs vajadzīgo rezultātu, bet nav nepieciešams zināt, kā tas uzbūvēts.

Parasti programmēšanas valodās to nodrošina procedūras un funkcijas. Valodā C++ formāli ir pieejamas **tikai funkcijas** (procedūru vietā kalpo funkcijas, kas atgriež tukšo (*void*) tipu, respektīvi, neatgriež neko, tātad, savā ziņā būtu uzskatāmas par procedūrām).

Funkciju veidošana pēc "melnās kastes" principa bieži tiek saukta par **informācijas slēpšanu** (*information hiding*). Informācijas slēpšana ir svarīgs mehānismu kopums jebkurā augsta līmeņa programmēšanas valodā, un tai ir ļoti svarīga loma arī objektorientētajā programmēšanā, kas tiks apskatīta vēlāk.

> **Funkcija** ir patstāvīgs programmas bloks, kas veic noteiktas darbības, un atgriež noteikta tipa vērtību. Funkciju raksturo vārds, atgriežamais tips un parametri.

Funkcijai

- var būt neviens vai vairāki parametri,
- ir vērtības atgriešanas (*return*) mehānisms, kuru raksturo atgriežamais tips.

Valodā C++

- ir vismaz viena funkcija: *main* – galvenā funkcija, jeb, pēc būtības – galvenā programma,
- funkcijas atgriežamais tips var būt tukšs (*void*).

Ar funkciju darbību cieši saistīti jēdzieni ir parametri un argumenti.

> **Parametri** ir mehānisms, ar kuru vērtības tiek nodotas funkcijai noteiktu darbību veikšanai.

Deklarējot funkciju, tiek noteikts parametru skaits un to tipi.

> **Argumenti** ir vērtības, kas, funkciju izsaucot, tai tiek padotas caur parametriem.

Alternatīva terminoloģija. Gan parametrus, gan argumentus reizēm mēdz saukt vienkārši par parametriem, bet reizēm, lai tos atšķirtu, parametrus sauc par **formālajiem parametriem**, bet argumentus par **faktiskajiem** jeb **aktuālajiem parametriem**.

Nākošajā piemērā ir parādīta funkcija ar diviem parametriem a un b, kas rēķina šo divu skaitļu summu, bet, šo funkciju izsaucot galvenajā funkcijā *main*, caur tiem tiek padoti argumenti x un y:

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

int add (int a, int b) // funkcijas realizācija
{
    return a + b;
};

int main ()
{
    int x=5, y=7, z;
    z = add (x, y);
    cout << z << endl;
    return 0;
}
```

Att. 10.1. Programma ar funkcija *add* divu veselu skaitļu saskaitīšanai

```
12
```

Kā redzams piemērā, funkcija sastāv no galvas un ķermeņa figūriekavu bloka veidā, kurā uzskaitītas darbības, kuras veic funkcija. Funkcijas galvā redzams atgriežamais tips (*int*), funkcijas vārds (*add*) un funkcijas parametru tipi un nosaukumi (*int a, int b*).

Parādītajā piemērā mēs redzam funkcijas *add* realizāciju pirms funkcijas *main* un pēc tam funkcijas *add* izsaukumu funkcijas *main* ietvaros.

10.2. Funkciju pielietošana

10.2.1. Vispārīgie principi

Valodā C++ no funkcijas realizācijas vēl var nodalīt funkcijas prototipu, kas veic funkcijas deklarēšanu vēl pirms funkcijas realizācijas.

Tādējādi funkcijas vispārīgā gadījumā C++ programmās ir sastopamas trīs dažādās formācijās:

- Funkcijas **prototips** jeb deklarācija (neobligāti).
- Funkcijas **realizācija** jeb definīcija.
- Funkcijas **izsaukums**.

Iepriekšējais funkcijas piemērs (Att. 2.7) saturēja tikai realizāciju un izsaukumu, bet prototips atsevišķi netika izdalīts, un tā lomu pildīja realizācija.

Valodā C++ svarīgs funkciju raksturojošs jēdziens ir funkcijas prototips.

> **Funkcijas prototips** jeb **deklarācija** ir tāda informācija par funkciju, kas ietver funkcijas vārdu, parametru skaitu un tipus, kā arī atgriežamo tipu, kas ir paziņojums par to, ka funkcija programmā tiks izmantota.

Funkcijas prototipam, ja tas vispār tiek lietots, jāatrodas gan pirms realizācijas, gan pirms izsaukuma.

Funkcijas deklarācija (prototips) **var** saturēt parametru vārdus un noklusētās vērtības.

Nākošais piemērs atkārto iepriekšējo ar vienu atšķirību – tiek atsevišķi izdalīts prototips *int add (int, int)*.

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

int add (int, int); // funkcijas deklarācija

int add (int a, int b) // funkcijas realizācija
{
    return a + b;
};

int main ()
{
    int x=5, y=7, z;
    z = add (x, y);
    cout << z << endl;
    return 0;
}
```

```
12
```

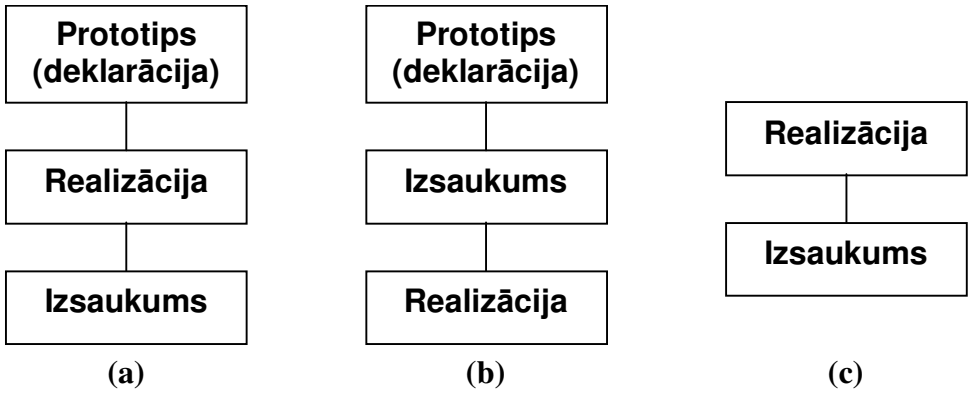
Att. 10.2. Programma ar funkciju *add* divu veselu skaitļu saskaitīšanai, izmantojot prototipu

Šajā piemērā var redzēt atšķirību starp prototipu un funkcijas galvu tās realizācijas daļā – prototipā drīkst neuzrādīt parametru vārdus, jo pietiek ar parametru tipiem.

Vispārīgā gadījumā trīs funkcijas formāciju secības variantus programmā nosaka šādi noteikumi:

- Funkcijas prototipam (deklarācijai) ir jābūt pirms realizācijas un izsaukuma;
- Realizācijas un izsaukuma savstarpējai secībai nav nozīmes, turklāt tie var atrasties arī dažādos failos;
- Funkcijas prototips (deklarācija) var tikt izlaists, tādā gadījumā tā lomu pilda realizācija.

Tādējādi ir iespējami šādi secības varianti:



Att. 10.3. Funkcijas formāciju secības varianti vienā failā

Funkciju prototipus parasti liek atsevišķos hedera failos (*.h*), kurus attiecīgajos C++ failos iekļauj ar direktīvas *#include* palīdzību. To, ka dažādi C++ faili veido vienotu programmu, realizē projektu mehānisms attiecīgajā izstrādes vidē.

Nākošajā piemērā atkal būs tā pati programma, tikai šoreiz programmas kods ir dalīts 3 dažādos failos:

add.h

```
int add (int, int); // funkcijas deklarācija
```

add.cpp

```
#include "add.h"

int add (int a, int b) // funkcijas realizācija
{
    return a + b;
};
```

addmain.cpp

```
#include "add.h"
#include <iostream>
using namespace std;

int main ()
{
    int x=5, y=7, z;
    z = add (x, y);
    cout << z << endl;
    return 0;
}
```

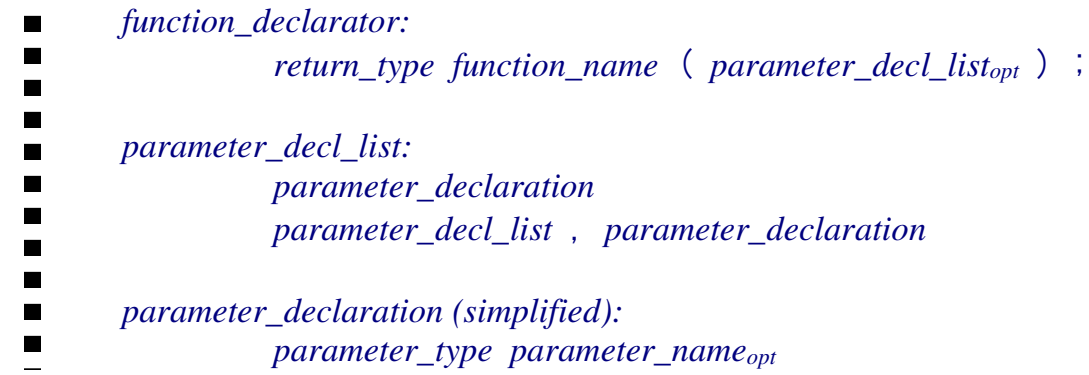
```
12
```

Att. 10.4. Programma ar funkciju *add* divu veselu skaitļu saskaitīšanai, izmantojot prototipu

Piemēros Att. 10.2 un Att. 10.4 funkcijas *add* deklarācija (prototips) ir:

```
int add (int, int);
```

Funkcijas deklarācijas vienkāršota sintakse parādīta nākošajā attēlā:



Att. 10.5. Funkcijas deklarācija (*function_declarator*)

Turpmāk, ja tas netiks īpaši uzsvērts, termini funkcijas prototips un funkcijas deklarācija tiks lietoti kā sinonīmi.

10.2.2. Funkcijas realizācija

> **Funkcijas realizācija** jeb **definīcija** ir pilns funkcijas apraksts, kas sastāv no funkcijas galvas (*head*), kas ir līdzīga prototipam, un funkcijas ķermeņa (*body*).

Funkcijas galvā atšķirībā no prototipa ir obligāti jāuzrāda parametru nosaukumi (salīdziniet neterminālo simbolu *parameter_declaration* un *parameter_definition* sintaksi!).

Programmā Att. 10.2 funkcijas realizācija ir:

```
int add (int a, int b)
{
    return a + b;
};
```

function_definition:
function_head *function_body*

function_head:
return_type *function_name* (*parameter_def_list*$_{opt}$)

parameter_def_list:
parameter_definition
parameter_def_list , *parameter_definition*

parameter_definition (simplified):
parameter_type *parameter_name*

function_body:
{ *statement_list*$_{opt}$ }

Att. 10.6. Funkcijas realizācija (*function_definition*)

Funkcijas ķermenim, ja tā atgriežamais tips nav *void*, obligāti jāsatur vismaz viens **atgriešanas priekšraksts** (*return statement*). Ja atgriežamais tips ir *void*, tad tas nav obligāti, tomēr funkcija drīkst saturēt vienu vai vairākus **tukšos atgriešanas operatorus**. Atgriešanas operators nodrošina funkcijas darbības pārtraukšanu, un, ja atgriežamais tips nav tukšs, tad arī vērtības atgriešanu.

return_statement:
`return` *expression* ;

return_statement_void:
`return` ;

Att. 10.7. Atgriešanas priekšraksts (*return_statement, return_statement_void*)

10.2.3. Funkcijas izsaukums

> **Funkcijas izsaukums** (*function call*) ir izpildāmā uzdevuma nodošana funkcijai, vajadzības gadījumā (un ja atgriežamais tips nav *void*) iegūstot funkcijas izrēķināto vērtību.

Programmā Att. 10.2 funkcijas izsaukums ir:

```
add (x, y)
```

function_call:
function_name (*argument_list*$_{opt}$)

argument_list:
argument
argument_list , *argument*

argument:
expression

Att. 10.8. Funkcijas izsaukums (*function_call*)

10.3. Funkcijas parametri

No izmantošanas viedokļa funkcijā ir divi parametru veidi:

- parametri-vērtības,
- paremetri-references.

10.3.1. Parametri-vērtības

Programmā Att. 10.2 abi funkcijas parametri ir parametri-vērtības, jo tie domāti tikai datu padošanai funkcijai vienā virzienā.

```
int add (int a, int b)
{
    return a + b;
};
    //... (funkcija main)
int x=5, y=7, z;
z = add (x, y);
```

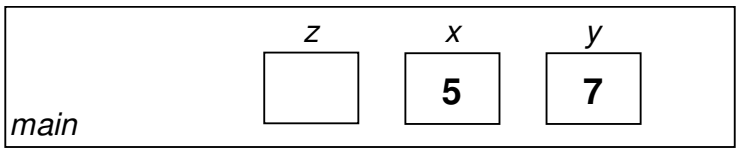
> **Parametrs-vērtība** ir tāds parametrs, caur kuru padotais arguments tiek dublēts funkcijas vajadzībām. Tā kā funkcija tādā gadījumā strādā ar padotās informācijas kopiju, tad izsaucošajā funkcijā attiecīgā informācija netiek izmainīta.
>
> **Parametrs-vērtība** ir tāds parametrs, caur kuru informācija tiek padota tikai vienā virzienā – funkcijā iekšā.

Kā var spriest no definīcijām, funkcijai padotais informācijas saturs tiek dublēts otrā eksemplārā funkcijas vajadzībām.

Parametru nodošanas mehānisms notiek šādi (sk. arī piemēru kopumā).

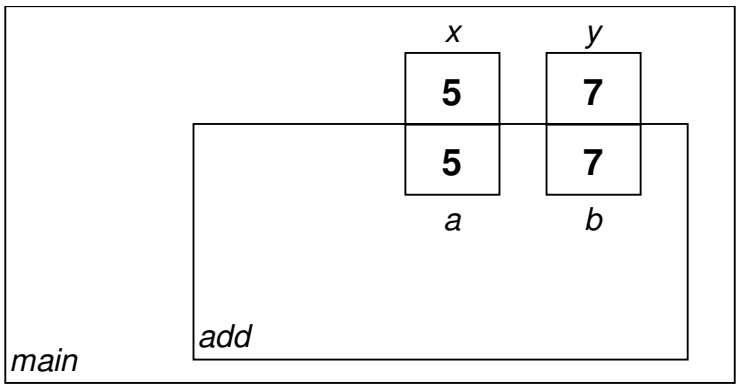
Sākotnējā programmas konfigurācija ir šāda:

```
int x=5, y=7, z;
```



Tiek izsaukta funkcija *add*:

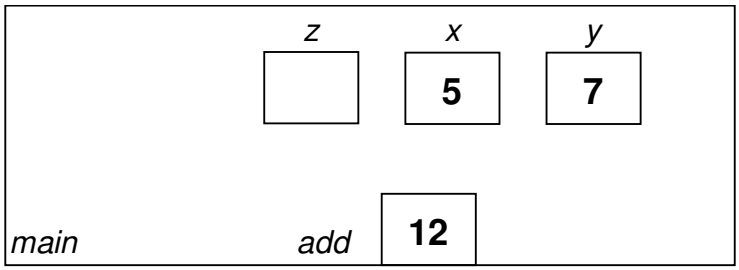
```
add (x, y)
```



Kā redzams, vērtības 5 un 7 tiek nodublētas funkcijas vajadzībām otrā eksemplārā. Pievērsiet uzmanību, ka attiecīgās vērtības izsaucošajā un izsauktajā modulī glabājas mainīgos ar atšķirīgiem vārdiem – mainīgo vārdiem nav nekādas nozīmes, parametru identifikācijas vienīgais nosacījums ir kārtas numurs, t.i., pirmo argumentu caur pirmo parametru utt.

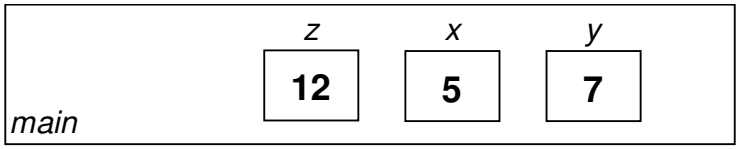
Funkcija izrēķina un atgriež vērtību (no funkcijas "palikusi pāri" tikai tās izrēķinātā vērtība):

```
return a + b;
```



Funkcijas izrēķinātā vērtība tiek piešķirta mainīgajam z:

```
z = add...
```



10.3.2. Parametri-references

Ja mūs būtu pieejami tikai parametri-vērtības, tad vienīgais veids, kā atgriezt no funkcijas kādu vērtību, būtu tās atgriešanas mehānisms, bet tas, kā zināms, spēj atgriezt tikai vienu vērtību. Ja no funkcijas jāatgriež vairāk nekā viena vērtība, tad ar šādu mehānismu nepietiek. Šim nolūkam valodā C++ ir pieejami parametri-references (pēc līdzības ar valodas PASCAL parametriem-mainīgajiem).

> **Parametrs-reference** ir tāds parametrs, caur kuru padotais arguments netiek dublēts, bet nonāk koplietošanas režīmā starp izsaucošo un izsaucamo funkciju.
>
> **Parametrs-reference** ir tāds parametrs, caur kuru informācija tiek padota abos virzienos – gan funkcijā iekšā, gan no funkcijas ārā.

Lai parametru apzīmētu kā references parametru, to deklarējot (definējot) aiz parametra tipa jāliek references simbols '&'. Pievienojot jēdzienu par parametru-referenci, parametra deklarācija un definīcija nu izskatās šādi:

parameter_definition (simplified #2):
parameter_type $\&_{opt}$ *parameter_name*

parameter_declaration (simplified #2):
parameter_type $\&_{opt}$ *parameter_name*$_{opt}$

Att. 10.9. References parametru deklarēšana/definēšana (*parameter_definition/parameter_declaration*)

References parametri nav pieejami valodā C. Tā vietā izmantojamas norādes (*pointers*).

Ja parametrs ir parametrs-reference, tad tas (ka parametrs ir reference) jāatspoguļo gan funkcijas deklarācijā, gan realizācijā, pievienojot references simbolu abās vietās.

Nākošais piemērs funkcionāli dara to pašu, ko iepriekšējie – rēķina divu skaitļu summu, tomēr funkcija, kas to dara, ir veidota tā, ka operandi tiek padoti pa otro un trešo parametru, bet rezultāts tiek atgriezts caur 1. parametru (salīdzināt ar Att. 10.2).

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

void add (int&, int, int); // funkcijas deklarācija

int main ()
{
    int x=3, y=4, z;
    add (z, x, y);
    cout << z << endl;
    return 0;
};

void add (int &r, int a, int b) // funkcijas realizācija
{
    r = a + b;
    return; // neobligāts
}
```

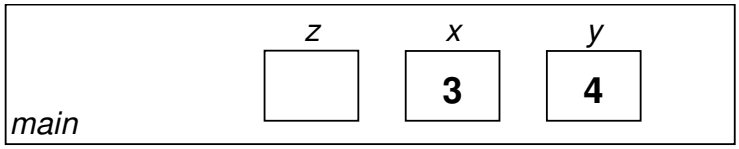
```
7
```

Att. 10.10. Programma ar funkciju *add* divu veselu skaitļu saskaitīšanai, izmantojot references parametru vērtības atgriešanai

Parametru nodošanas mehānisms notiek šādi.

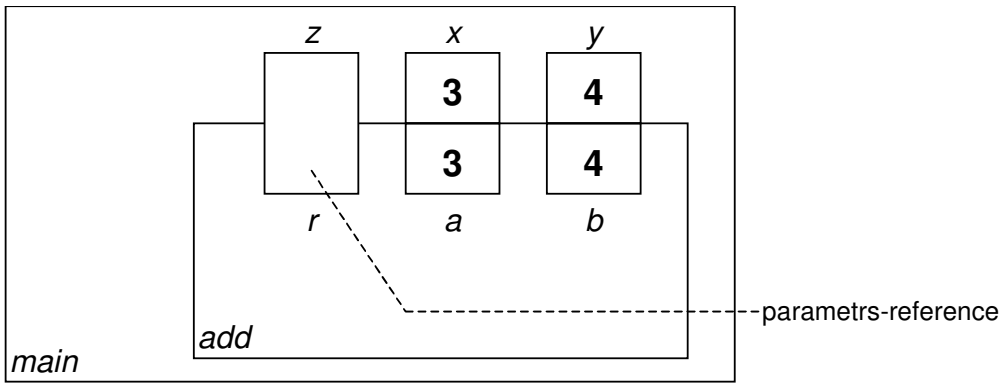
Sākotnējā programmas konfigurācija ir šāda:

```
int x=3, y=4, z;
```



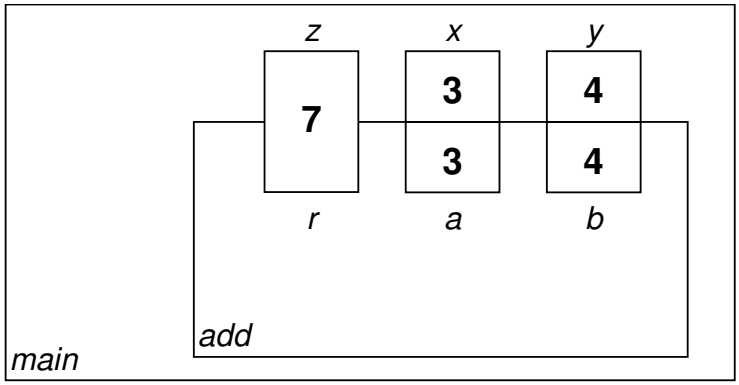
Funkcijas izsaukums:

```
add (z, x, y);
```

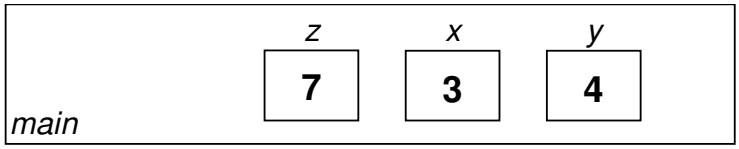


Kā redzams, otrais un trešais parametrs tiek dublēti funkcijas vajadzībām, bet pirmais (kurš ir paredzēts izrēķinātās vērtības atgriešanai) nonāk koplietošanas režīmā starp funkciju *main* un funkciju *add*.

Funkcijas *add* darbības beigās programmas konfigurācijas ir šāda:



Kad programmas vadība pēc funkcijas *add* izsaukuma atgriezusies galvenajā funkcijā:



11. Funkcijas: pārslogošana, redzamība, rekursija

11.1. Funkciju nosaukuma pārslogošana

> **Pārslogošana** (*overloading*) ir vienāda vārda izmantošana dažādu funkciju nosaukšanai.

Pārslogošana ir plašāka jēdziena – **polimorfisms** – apakšgadījums. Funkciju (nosaukuma) pārslogošana nav iespējama valodā C.

Funkciju pārslogošana valodā C++ iespējama tāpēc, ka funkcijas tiek identificētas nevis tikai ar vārdu (kā tas ir valodā C), bet papildus arī ar to parametru tipu virkni, resp., ar **signatūru**.

> **Funkcijas signatūra** (*signature*) valodā C++ ir funkcijas apraksts, kas ietver funkcijas vārdu un parametru tipu virkni, ieskaitot arī parametru modifikatorus *const*. Funkcijas signatūra identificē funkciju programmā.

- Pie (C++ funkcijas) sigantūras var pieskaitīt arī atgriežamo tipu. Funkcijas prototips ietver tikai vai gandrīz tikai funkcijas signatūru,

  BET

- nevar pastāvēt divas dažādas funkcijas, kuru prototips atšķiras tikai ar atgriežamo tipu.

Nākošajā piemērā parādītas divas funkcijas *add* (to prototipi un izsaukšana ar kontekstu) ar dažādām signatūrām (abi iepriekš apskatītie varianti ar un bez references izmantošanas), un šajā piemērā funkciju nosaukuma pārslogošana notikusi veiksmīgi:

```
void add (int& r, int a, int b); // funkcijas deklarācija
int add (int a, int b);          // funkcijas deklarācija
    ...
int x=3, y=4, z1, z2;
z1 = add (x, y);
add (z2, x, y);
    ...
```

Att. 11.1. Divas funkcijas *add*, kas izmanto funkciju nosaukuma pārslogošanu

Tomēr izrādās, ka, lai veiksmīgi izmantotu pārslogotas funkcijas, nepietiek ar korektu to definēšanu. Svarīgs ir arī izsaukšanas konteksts. Ja pēc funkcijas izsaukuma nav iespējams pateikt, kura no pārslogotajām funkcijām (kas varētu būt definētas korekti) ir domāta, iestājas kompilatora kļūda:

```
float add (float a, float b);     // funkcijas deklarācija
double add (double a, double b); // funkcijas deklarācija
     ...
int x=3, y=4;
cout << add (x, y) << endl;
     ...
```

Kompilatora kļūda:

```
fun7over.cpp: In function `int main()':
fun7over.cpp:18: call of overloaded
   `add(int&, int&)' is ambiguous
fun7over.cpp:5: candidates are:
   float add(float, float)
fun7over.cpp:10:
   double add(double, double)
```

Att. 11.2. Neviennozīmīga pārslogotas funkcijas izsaukšana

Funkcijas izsaukumā tiek padoti divi *int* parametri. Šāds komplekts der gan vienai, gan otrai pārslogotajai funkcijai, kas izraisa divdomīgu situāciju.

Lai šo kļūdu labotu, izsaukumā jāprecizē, kuru īsti funkciju jāizsauc. To dara, pielietojot argumentiem tipa pārveidošanu:

```
float add (float a, float b);     // funkcijas deklarācija
double add (double a, double b); // funkcijas deklarācija
     ...
int x=3, y=4;
cout << add ((double)x, (double)y) << endl;
     ...
```

```
7
```

Att. 11.3. Izvairīšanās no funkcijas divdomīgas izsaukšana

11.2. Noklusētie parametri

> **Noklusētie parametri** (*default parameters*) ir parametri, kurus funkcijas izsaukumā var izlaist, un kas izlaišanas gadījumā pieņem iepriekš noteiktas vērtības.

Parametrus, kas nav noklusētie, sauksim par **nestandarta parametriem**. Valodā C++ (atšķirībā, piemēram, no *Visual Basic*) noklusētie parametri nedrīkst atrasties pa kreisi no nestandarta parametriem. Noklusēto parametru raksturo tas, ka aiz tā deklarācijas vai definīcijas seko piešķiršanas operators un tiek uzrādīta vērtība:

```
int b=1
```

vai, ja tas ir deklarācijā un nav uzrādīti parametru vārdi:

```
int=1
```

```
int add (int a, int b=1) // funkcijas realizācija
{
    return a + b;
};
    ...
int x=5, y=7;
cout << add (x, y) << endl;
cout << add (x) << endl;
    ...
```

```
12
6
```

Att. 11.4. Noklusētie parametri

Noklusētie parametri vai nu funkcijas deklarācijā vai realizācijas galvā ietver noklusētās vērtības uzstādīšanu, tomēr parasti to dara tieši deklarācijā – abās vietās to darīt nedrīkst.

```
int add (int, int=1); // funkcijas deklarācija
    ...
int x=5, y=7;
cout << add (x, y) << endl;
cout << add (x) << endl;
    ...
int add (int a, int b) // funkcijas realizācija
{
    return a + b;
};
```

```
12
6
```

Att. 11.5. Noklusētie parametri ar to uzstādīšanu funkcijas deklarācijā

Pievienojot jēdzienu par noklusētajiem parametriem, parametra deklarācija un definīcija izskatās šādi:

parameter_definition (simplified #3):
parameter_type parameter_name $default\_value_{opt}$

parameter_declaration (simplified #3):
parameter_type $parameter\_name_{opt}$ $default\_value_{opt}$

default_value:
= value

Att. 11.6. Noklusēto parametru definēšana/deklarēšana (*parameter_definition/parameter_declaration*)

Parametri-references nevar būt noklusētie parametri.

11.3. Parametri-konstantes

> **Parametri-konstantes** jeb konstantie parametri ir parametri, caur kuriem padotos argumentus nedrīkst izmainīt funkcijas iekšienē. Parametra konstantums tiek norādīts ar modifikatora *const* palīdzību pirms parametra tipa.

Konstantuma īpašību parasti izmanto lielu vērtību nodošanai funkcijām, sintakses līmenī nodrošinot, kas šīs vērtības funkcijas iekšienē netiks izmainītas.

Mazu vērtību nodošanai (piemēram, *int*), lai nodrošinātu, ka tās netiks izmainītas funkcijas darbības laikā, pilnīgi pietiek ar parametriem-vērtībām, jo caur tiem nodotās vērtības tiek dublētas funkcijas vajadzībām, tāpēc oriģināli paliek neskarti.

Tomēr lielām vērtībām to nodošana, pilnībā dublējot to saturu, būtu nelietderīga resursu tērēšana, tāpēc tiek lietoti parametri-references vai parametri-norādes, kuriem savukārt, lai nodrošinātu neizmainīšanu, papildus pieliek konstantumu. Parametriem-konstantēm modifikators *const* ir jāpielieto gan funkcijas deklarācijā, gan realizācijā.

```
double add (const double &a, const double &b);
    ...
cout << add (x, y) << endl;
    ...
double add (const double &a, const double &b)
{
    return a + b;
}
```

Att. 11.7. Parametri-konstantes

11.4. Redzamības apgabals. Lokālie, statiskie un globālie mainīgie

Kā jau iepriekš tika minēts, lai nodrošinātu “melnās kastes” principu, jāveic informācijas slēpšana. Viens no mehānismiem, kā to izdarīt, ir lokālo mainīgo izmantošana funkcijās.

> **Lokālie mainīgie** (*local variables*) ir mainīgie, kas darbojas (ir redzami) tikai noteikta bloka (parasti – funkcijas) iekšienē.

Lokālais mainīgais *a* ir redzams tikai līdz bloka beigām, kurā tas deklarēts:

```
{
    ...
    int a;
    // sākot ar šo vietu, mainīgo a var izmantot
    ...
}
// te mainīgais a vairs nav redzams
```

No funkcijas darbības viedokļa par lokālajiem mainīgajiem ir uzskatāmi arī funkcijas parametri.

> Programmas daļu, kurā ir redzams noteikts mainīgais, sauc par šī mainīgā **redzamības apgabalu** (*scope*).

Balstoties uz redzamības apgabala definīciju, lokālā mainīgā redzamības apgabals ir bloks (parasti, funkcija).

> **Bloks** (*block*) ir (C++) programmas daļa, kas ietverta starp atverošo un aizverošo figūriekavu vienas funkcijas ietvaros.

Funkcijas ķermenis ir tipiskākais programmas bloka piemērs.

Mainīgajiem bez redzamības apgabala bieži svarīgs ir arī jēdziens eksistences periods.

> Programmas darbības intervālu, kurā eksistē noteikts mainīgais, sauc par šī mainīgā **eksistences periodu** vai eksistences laiku.

Lokālā mainīgā eksistences periods ir no tā deklarēšanas līdz tā bloka beigām, kurā tas deklarēts.

Lokālo mainīgo nozīme ir labāk saprotama, paralēli apskatot arī globālos mainīgos.

> **Globālie mainīgie** (*global variables*) ir mainīgie, kuru redzamības apgabals ir visa programma.

Globālos mainīgos parasti izmanto tikai specifiskos gadījumos, jo to izmantošana ir pretrunā ar procedurālo abstrakciju un parasti uzskatāma par sliktu stilu.

Globālais mainīgais tiek deklarēts ārpus jebkura bloka un tādēļ ir redzams jebkura bloka iekšienē. Kā jau tika minēts, tas ir slikti no programmas strukturēšanas viedokļa.

```
int g; // globālais mainīgais
int main ()
{
    // mainīgais g šeit ir redzams
}
void another_function (...)
{
    // mainīgais g šeit ir redzams
}
```

Arī standarta bibliotēkās tiek izmantoti globālie mainīgie. Tipiskākie piemēri ir *cin* un *cout*.

Specifiski lokālie mainīgie ir t.s. statiskie mainīgie.

> **Statiskie mainīgie** (*static variables*) ir tādi funkcijas lokālie mainīgie, kas saglabā to vērtības starp šīs funkcijas dažādiem izsaukumiem.

Statiskie mainīgie ir veids, kā funkciju nodrošināt ar "atmiņu". Statisko mainīgo deklarē, izmantojot modifikatoru *static* un obligāti inicializējot to deklarācijas ietvaros. Tiesa gan,

inicializācija, balstoties uz to, tiks veikta tikai, pirmo reizi izsaucot funkciju, citādi statiskums acīmredzami netiktu nodrošināts.

static_variable_declaration:
`static` *data_type variable_name* `=` *expression* `;`

Att. 11.8. Statiskā mainīgā deklarēšana (*static_varibale_declaration*)

Gan globālo, gan statisko mainīgo eksistences periods ir visa programma.

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

int counter ()
{
    static int cnt=0; // stat. mainīgā deklarācija
    cnt++; // atceras iepriekšējo vērtību un pieskaita 1
    return cnt;
};

int main ()
{
    for (int i=0; i<5; i++)
    {
        cout << counter() << endl;
    };
    return 0;
}
```

```
1
2
3
4
5
```

Att. 11.9. Statiska mainīgā izmantošana funkcijā

11.5. Atgriešanas mehānisms

Vērtības atgriešanas mehānisms (“return” mehānisms) nodrošina funkcijas darbības pārtraukšanu un programmas darbības nodošanu uz šo funkciju izsaukušo moduli, nepieciešamības gadījumā atgriežot funkcijas vērtību. Atgriešanu nodrošina atgriešanas operators *return*.

Parasti funkcijā atgriešanas operators ir viens, un tas novietots pašās funkcijas beigās (funkcijās ar tukšu atgriežamo tipu to var izlaist), tomēr atsevišķos gadījumos, lai vienkāršotu funkcijas pierakstu, ir ērti izmantot vairākus atgriešanas operatorus vienā funkcijā.

Nākošais programmas piemērs demonstrē funkciju *prime* pirmskaitļa atpazīšanai, kas izmanto vairāk nekā vienu *return* komandu.

```
bool prime (int num)
{
    for (int i=2; i<num; i++)
    {
        if (num % i == 0) return false;
    };
    return true;
};
```

Att. 11.10. Vairāki *return* operatori vienā funkcijā

11.6. Rekursīvas funkcijas

> **Rekursija** (*recursion*) ir process, kad process izsauc pats sevi. Funkcijas, kuras izsauc pašas sevi, sauc par **rekursīvām funkcijām** (*recursive functions*).

Daudzos gadījumos rekursīvās funkcijas ir ļoti ērti izmantojamas, tomēr tās jāpielieto ar piesardzību, jo daudzu funkciju rekursīvie varianti izpildās krietni lēnāk, turklāt liels daudzums rekursīvo izsaukumu var izraisīt steka pārpildīšanos.

Izstrādājot rekursīvu funkciju, jāatceras, ka tajā jāiekļauj nosacījuma operators, kurš nodrošina izeju no rekursīvā cikla. Ja tas netiks izpildīts, tad, vienreiz izsaucot funkciju, tā turpinās izsaukt sevi tik ilgi, kamēr notiks steka pārpildīšanās. Tas notiks tāpēc, ka funkcijas parametri un lokālie mainīgie glabājas stekā, un pie katra jauna izsaukuma stekā tiem tiek izdalīts atmiņas apgabals.

Pēc līdzības ar matemātisko indukciju, rekursīva funkcija sastāv vismaz no 2 šādām daļām:

- bāzes bloks;
- rekursīvais bloks;

Nākošajā piemērā parādīta rekursīva funkcija faktoriāla izrēķināšanai. Šī funkcija gan nekad nepārpildīs steku savas specifikas dēļ (faktoriāls pārāk lieliem skaitļiem nesatilps *int* tipa mainīgajā, tādējādi pie lielas argumenta vērtības visdrīzāk programma beigsies ar citu kļūdu), tomēr arī šeit rekursija nebūtu uzskatāma par labāko izvēli problēmas risināšanai.

```
int fact (int num)
{
    if (num <= 1) return 1;
    else return num * fact (num-1);
};
```

Att. 11.11. Rekursīva funkcija faktoriāla izrēķināšanai

Dotajā piemērā par rekursijas bāzes bloku kalpo:

```
if (num <= 1) return 1;
```

bet rekursiju nodrošina:

```
num * fact (num-1)
```

Bez tiešās rekursijas, kad funkcija izsauc pati sevi, ir iespējama arī netiešā rekursija – kad, piemēram, funkcija A izsauc funkciju B, bet funkcija B funkciju A.

12. Masīvi, norādes un references

12.1. Masīvs kā funkcijas parametrs

12.1.1. Viendimensijas masīvs kā funkcijas parametrs

Masīvu nevar nodot caur funkcijas parametru kā vērtību (tādējādi veicot tā satura dublēšanu), bet tā nodošana caur parametru iespējama tikai tādā veidā, kā tas notiek caur parametriem-referencēm. Šādiem noteikumiem valodā C++ varētu būt 2 iemesli: pirmkārt, masīvs ir relatīvi apjomīga datu struktūra un nebūtu saprātīgi dublēt tik lielu informācijas apjomu, otrkārt, valodā to nosaka valodas C++ īpatnības – masīva mainīgais reprezentē nevis masīvu kopumā, bet gan tikai norāda uz tā sākumu.

Lai definētu parametru, caur kuru iespējams nodot (viendimensijas) masīvu, iespējami divi dažādi deklarēšanas veidi, kas ir analoģiski:

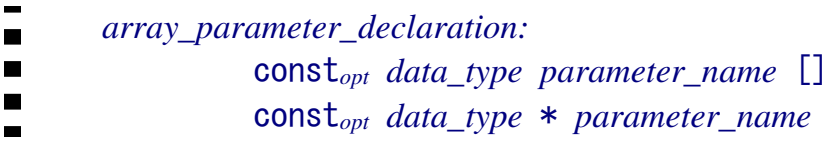
- **masīvu specifisks**, izmantojot masīvu notāciju – kvadrātiekavas [],
- **universāls**, izmantojot norāžu notāciju – zvaigznīti *.

Tāpat kā parastu parametru deklarēšanā, arī šajā gadījumā tā līdzinās mainīgo deklarēšanai. (Ievērojiet, ka parasta mainīgā, nevis parametra, deklarēšanas gadījumā abu sintakses variantu lietošanai ir atšķirības – tie nosaka attiecīgi statisku vai dinamisku masīvu.)

Sekojošie divi piemēri ir identiski, parādot divus alternatīvus veidus, kā deklarēt funkciju ar vienu parametru, caur kuru var nodot masīvu no simboliem *char*:

```
void function1 (char s[])
void function2 (char *s)
```

Kā citiem parametriem, pirms datu tipa var likt modifikatoru *const*, kas norādīs, ka masīvu funkcijā nedrīkst mainīt. Sintaktiski to apraksta nākošā shēma.



Att. 12.1. Masīva parametra deklarēšana (*array_parameter_declaration*)

Nākošajā piemērā tiks izveidoti divi masīvi – viens statiskais (*name*) un viens dinamiskais (*name2*) un abi izdrukāti, izmantojot vienu un to pašu funkciju, kurai tie tiek padoti caur parametru.

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

int printString (const char s[])
{
    cout << s << endl;
};

int main ()
{
    char name[10]; // statisks masīvs
    strcpy (name, "snow");
    printString (name); // masīva padošana funkcijai
    char *name2 = new char[10]; // dinamisks masīvs
    strcpy (name2, "snow2");
    printString (name2); // masīva padošana funkcijai
    delete[] name2;
    return 0;
}
```

```
snow
snow2
```

Att. 12.2. Viendimensijas masīvs kā funkcijas parametrs

Kā jau sākumā tika pieminēts,

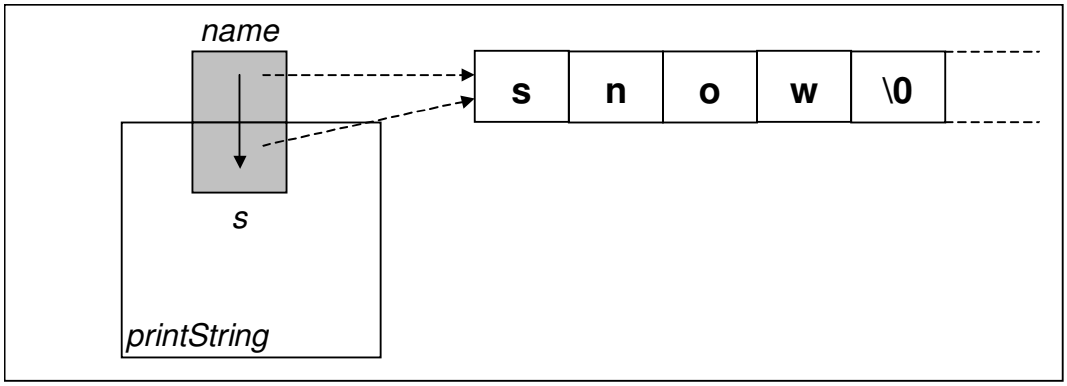
```
int printString (const char s[])
```

vietā varēja rakstīt

```
int printString (const char *s)
```

Var teikt, ka funkcijai netiek nodots pats masīvs, bet gan tikai norāde uz masīvu. Arī pats masīva mainīgais pēc būtības ir norāde. Viena no masīva (vai masīva parametra) definēšanas notācijām ietver zvaigznīti, kas nozīmē norādi. **Norāde** (*pointer*) ir ļoti svarīgs jēdziens valodā C++. Tas pēc būtības ir ļoti līdzīgs jēdzienam **reference** (parametri-references tika apskatīti nodaļā par funkcijām), un abi jēdzieni smalkāk tiks apskatīti tālāk šajā nodaļā.

Masīva padošanu caur funkcijas parametru (atbilstoši Att. 12.2) vizuāli parāda nākošais attēls:



Att. 12.3. Masīva padošana caur funkcijas parametru

12.1.2. Statisks vairākdimensiju masīvs kā funkcijas parametrs

Viendimensijas masīva un vairākdimensiju masīvu padošana caur parametru sintaktiski neatšķiras, jo masīva mainīgā būtība ir vienāda abos gadījumos – norāde uz masīva sākumu.

Līdzība ar viendimensijas (statisko) masīvu nodošanai caur parametru statiskam vairākdimensiju masīvam ir tāda, ka pirmajai dimensijai izmērs nav jāuzrāda, tomēr visām pārējām dimensijām ir. Tāpat kā viendimensijas masīvu gadījumā ir pieejamas divas notācijas – masīva un norāžu.

Tālāk ir redzami divi alternatīvi funkciju piemēri, caur kuriem padot divdimensiju masīvu, kura otrā dimensija ir ar lielumu 4.

```
void function1 (char s[][4])
void function2 (char (*s)[4])
```

Ievērojiet, ka norāžu notācijā ir jālieto papildus iekavas.

multi_array_parameter_declaration:

const$_{opt}$ *data_type parameter_name* [] *array_dimensions*

const$_{opt}$ *data_type* (* *parameter_name*) *array_dimensions*

array_dimensions:

[*dimension_size*]

[*dimension_size*] *array_dimensions*

Att. 12.4. Statiska vairākdimensiju masīva parametra deklarēšana (*multi_array_parameter_declaration*)

Ievērojiet, ka nākošajā piemērā lietotās norādes notācijas vietā

```
int printStrings (const char (*s)[4], int count)
```

var lietot arī masīvu notāciju:

```
int printStrings (const char s[][4], int count)
```

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

int printStrings (const char (*s)[4], int count)
{
    for (int i=0; i<count; i++) cout << s[i] << endl;
};

int main ()
{
    char arr[2][4] = {{'t', 'h', 'e', '\0'},
                      {'p', 'i', 'g', '\0'}};
    printStrings (arr, 2);
    return 0;
}
```

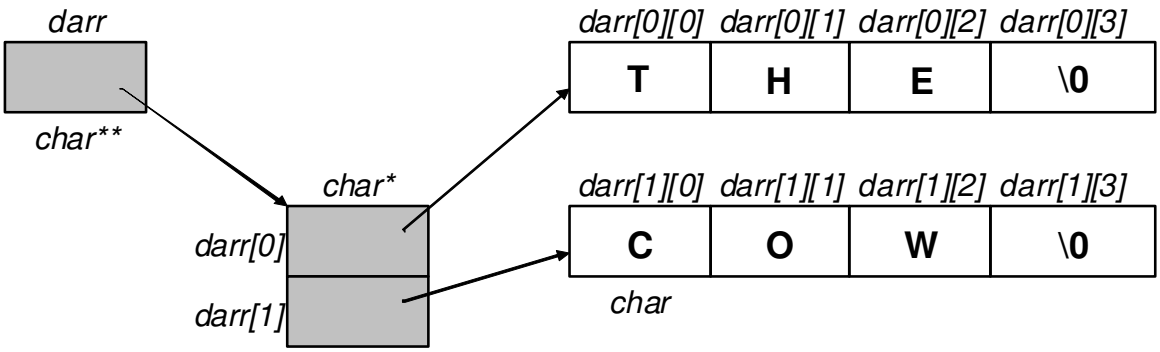
```
the
pig
```

Att. 12.5. Divdimensiju masīvs kā funkcijas parametrs (norādes notācija)

12.2. Dinamisks vairākdimensiju masīvs un tā nodošana caur parametru

Līdz šim ir tikuši apskatīti gan dinamiski masīvi, gan vairākdimensiju masīvi, tomēr masīvi, kas ir reizē dinamiski un daudzdimensiju – vēl nē.

Dinamiska masīva veidošana vairākdimensiju gadījumā ir sarežģītāka nekā viendimensijas masīva gadījumā, jo atmiņas izdalīšana jāveic katrai dimensijai atsevišķi, tādējādi var teikt, ka dinamiski vairākdimensiju masīvi nemaz nav masīvi tādā nozīmē, kā masīvi sākumā definēti: kaut arī to elementiem var piekļūt, izmantojot tradicionālo masīvu sintaksi, tomēr tie atmiņā atrodas dažādos apgabalos – vienā kopējā apgabalā atrodas tikai pirmā dimensija.



Att. 12.6. Dinamisks divdimensiju masīvs

Dinamisku vairākdimensiju masīva mainīgo deklarējot, jālieto tik daudz zvaigznīšu, cik masīvam ir dimensiju:

```
char **darr;  // divdimensiju din. masīva mainīgā deklarēšana
```

Tam atbilstošā sintakse:

dynamic_multiarray_declaration:
data_type dynamic_dimensions array_name ;

dynamic_dimensions:
*
* *dynamic_dimensions*

Att. 12.7. Dinamiska vairākdimensiju masīva deklarēšana (*dynamic_multiarray_declaration*)

Dinamisku vairākdimensiju masīvu veido pa dimensijām, un tā izmantošana notiek šādā secībā:

- atmiņas izdalīšana dimensijām, secībā no rupjākās uz smalkāko,
- dinamiska masīva izmantošana, izmantojot parasto masīvu sintaksi,
- atmiņas atbrīvošana pa dimensijām, sākot no smalkākās un beidzot ar rupjāko.

Deklarējot mainīgo, izmantojot zvaigznītes (norāžu) notāciju, jāatceras, ka zvaigznītes pieder pie datu tipa, un nosaucot vienu zvaigznīti, var izmantot **vai nu vārdu masīvs vai norāde**. Piemēram, divu dimensiju *int* masīva tips ir *int***, t.i. masīvs no masīviem no *int*, jeb norāde uz norādi no *int*. Katras nākošās dimensijas elementi ir ar tipu, kurā ir par vienu zvaigznīti mazāk, piemēram, masīvs ar tipu *int*** pirmajā līmenī sastāv no elementiem ar tipu *int**.

Dinamiska masīva parametrus deklarē tāpat kā dinamiska masīva mainīgo – lietojot tik daudz norāžu (zvaigznīšu), cik ir dimensiju (Att. 12.7):

```
int printStrings (char **s, int count)
```

Nākošais piemērs parāda divu dimensiju dinamiska masīva izmantošanu, nododot to kā parametru funkcijai.

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

int printStrings (char **s, int count)
{
    for (int i=0; i<count; i++) cout << s[i] << endl;
};

int main ()
{
    char **darr = new char*[2]; // pirmā dimensija
    for (int i=0; i<2; i++) darr[i] = new char[4]; // 2. dim.
    strcpy (darr[0], "THE");
    strcpy (darr[1], "COW");
    printStrings (darr, 2);
        // abu dimensiju likvidēšana, sākot ar 2.:
    for (int i=0; i<2; i++) delete[] darr[i];
    delete[] darr;
    return 0;
}
```

```
THE
COW
```

Att. 12.8. Dinamisks divu dimensiju masīvs

12.3. Masīvi un norādes

Valodā C++ masīviem un norādēm ir ļoti tieša saistība.

> **Norāde** (*pointer*) ir mainīgais (vai cits mehānisms), kas glabā (nosaka) atmiņas adresi.

Varētu teikt, ka norāde satur informāciju, kas "palīdz nonākt pie citas informācijas".

> **Adrese** (*address*) ir vērtība, kas apzīmē noteiktu vietu atmiņā.

Vizualizējot programmas darbību vai noteiktu atmiņas konfigurāciju, norāde parasti tiek attēlota kā bultiņa.

Masīvu un norāžu saistība ir tāda, ka **masīva mainīgais ir norāde**. Tādējādi var teikt, ka valodā C++ masīva mainīgais nevis reprezentē visu masīvu, bet gan tikai norāda uz tā sākumu. Kaut gan pieejamā masīva kvadrātiekavu notācija ļauj abstrahēties no šīs tehniskās detaļas, tomēr masīvu var apstrādāt arī, izmantojot norāžu notāciju.

Ieviešot jēdzienu par norādi, var izšķirt 2 veidu mainīgos:

- parastie mainīgie (glabā datus),
- norādes mainīgie (glabā adreses).

Norādes mainīgos parasti raksturo zvaigznīte (*) pie to deklarēšanas, vienīgais izņēmums ir statiski masīvi, kas tiek deklarēti, izmantojot kvadrātiekavu notāciju, tomēr savā būtībā arī ir norādes (kaut arī ar ierobežotām izmantošanas iespējām).

Adrese ir tehnisks jēdziens, kas domāts programmas iekšējām vajadzībām, tāpēc norāžu saturu tiešā veidā parasti nemēdz interpretēt.

Norāde savu vērtību, t.i., adresi, var iegūt šādos 3 veidos:

- statisks masīvs kā norāde – automātiski pie deklarēšanas,
  ```
  char arr[20]
  ```
- iegūstot to no citas norādes,
  ```
  char *arr2 = arr;
  ```
- noteiktam mainīgajam – izmantojot adreses ņemšanas operatoru &.
  ```
  int i;
  int *pointer = &i;
  ```

Deklarējot norādi, ir svarīgi norādīt, uz kāda tipa vērtību norāda mainīgais. Tam vislielākā nozīme ir nevis pašā adrešu apstrādē, bet gan, veicot apstrādi pēc adreses iegūtajiem datiem (piemēram, masīva no veseliem skaitļiem tips ir *int**, t.i., "norāde uz *int*", kas nozīmē, ka masīva elementi jāapstrādā kā *int* tipa vērtības).

Rindiņa
```
int *p;
```
deklarē mainīgo *p*, kas ir norāde uz *int* tipa vērtību, t.sk. var tikt lietots kā viendimensijas dinamiska masīva mainīgais.

Nākošais piemērs īpaši uzsver to, ka masīva mainīgais un pats masīvs ir divas dažādas lietas, un ka masīva mainīgais ir norāde: masīvs ir viens, bet mainīgie, kas uz viņu norāda, ir divi:

```
char arr[20] = "LU is OK";
char *arr2 = arr;
cout << arr2 << endl;
```

```
LU is OK
```

Att. 12.9. Vairākas norādes uz vienu un to pašu masīvu

Iepriekšējā piemērā uz vienu masīvu (jeb, vispārīgā gadījumā, uz vienu atmiņas apgabalu) norādīja divas norādes – un abas uz šī masīva sākumu.

Taču ir iespējams nolikt norādi arī citā masīva vietā. Norādes pārvietošana vai novietošana iespējama gan, izmantojot adreses ņemšanas operatoru &, gan adresu aritmētiku, pielietojot adreses mainīgajam aritmētiskas operācijas.

```
char arr[20] = "LU is OK";
char *arr2 = &arr[3]; // paņem adresi no elementa nr. 3
cout << arr2 << endl; // izdrukā simbolu virkni no tās vietas
arr2 += 3; // pārvieto norādi 3 vienības uz priekšu
cout << arr2 << endl; // izdrukā simbolu virkni no tās vietas
```

```
is OK
OK
```

Att. 12.10. Norāde uz masīva vidu un adresu aritmētika

Nākošais piemērs demonstrē adresu aritmētiku, parādot, cik vienību ir no masīvu sākumu līdz elementam nr. 3:

```
int arr[4];
cout << &arr[3] - arr << endl;
```

```
3
```

Att. 12.11. Adresu atņemšana

Adreses ņemšanas operatora & pieraksts ir tāds pats kā references operatora pieraksts. To, kurš operators tiek pielietots, nosaka pēc programmas konteksta.

12.4. Norādes, adreses un references

12.4.1. Adrese un vērtības iegūšana pēc adreses

Masīva mainīgais nav vienīgais norādes paveids. Norādes ir universāls līdzeklis datu pārvaldīšanai atmiņā. Saistībā ar norādēm izšķir 2 pretējas darbības:

- **adreses iegūšana** noteiktam atmiņas apgabalam pēc to reprezentējošā mainīgā, izmantojot adreses ņemšanas operatoru & (nesajaukt ar referenci!),
- **vērtības iegūšana pēc adreses** (nesajaukt ar norādes mainīgā deklarēšanu!).

Vērtības iegūšana pēc adreses notiek, pirms norādes liekot zvaigznītes operatoru *:

```
int arr[4] = {33,88,11,66};
int *p; // norādes mainīgā deklarēšana
p = &arr[2]; // arr[2] adreses piešķiršana mainīgajam p
cout << *p << endl; // satura (int) izdruka pēc adreses p
```

```
11
```

Att. 12.12. Adreses iegūšana un vērtības iegūšana pēc adreses

Lai strādātu ar adresu aritmētiku, teorētiski nebūtu jānorāda, norāde uz kādu tipu tā ir (vai *char**, vai *int**). Tomēr netiešā veidā tam ir nozīme šādos gadījumos:

iegūstot vērtību pēc adreses – iegūtā vērtība tiek interpretēta atbilstoši tipam (iepriekšējā piemērā **p* tiek interpretēts kā int, jo *p* tips ir *int**.

Izmantojot adresu aritmētiku, vienas vienības lielums ir nevis 1 baits, bet gan attiecīgā tipa lielums, piemēram, iepriekšējā piemērā *p++* nozīmētu nevis adreses pārvietošanos 1 baitu uz priekšu, bet gan 4 baitus, jo *int* izmērs ir 4 baiti.

Bez masīviem un simbolu virknēm norāžu mainīgos parasti izmanto dinamiskās datu struktūrās, kas tiks aprakstītas vēlāk.

12.4.2. References

Tehniski līdzīgs jēdziens norādei ir reference, un tas ir pieejams tikai valodā C++.

Referenci var definēt tehniski (no realizācijas viedokļa) vai no izmantošanas viedokļa.

Reference (*reference*) (tehniski) ir norādes paveids ar ierobežotām izmantošanas iespējām, bet ērtāku sintaksi.

Reference (*reference*) (no izmantošanas viedokļa) ir mainīgā sinonīms.

Tipiskākais referenču pielietojums ir funkciju parametros, tomēr vispārīgā gadījumā to pielietojums var būt plašāks – un parasti tikai, lai uzlabotu programmas lasāmību, salīdzinot ar alternatīvo variantu, lietojot norādes.

Referenci norāda references operators &. To nedrīkst sajaukt ar adreses ņemšanas operatoru – references operators piedalās mainīgā (vai parametra) deklarēšanā.

Vēl viena references atšķirība no norādes ir tāda, ka references mainīgo nevar deklarēt bez vērtības piešķiršanas tam deklarēšanas brīdī.

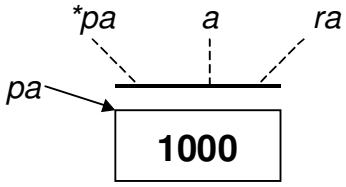
Referenci lieto arī funkcijas atgriežamajā vērtībā, tādējādi nodrošinot, ka funkcijas izsaukums var būt piešķiršanas operatora kreisajā pusē.

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

int main ()
{
    int a = 999;
    int &ra = a; // reference (sinonīms) uz a
    int *pa = &ra; // mainīgā a adrese
    (*pa)++; // a vērtība tiek palielināta caur adresi pa
    cout << a << endl; // izdrukā izmainīto a vērtību
    return 0;
}
```

```
1000
```

Att. 12.13. Norādes, adreses un references

Programmas konfigurācija izdrukas brīdī ir šāda:



13. Ievads objektorientētajā programmēšanā

13.1. Klase kā objektu īpašību apraksts

Programmēšanas valoda ir viens no algoritma pieraksta veidiem. Algoritms ir soļu, t.i., darbību virkne. Kaut arī programma sastāv gan no **darbībām**, gan **datiem**, tomēr klasiskais programmu strukturēšanas veids ir tieši pēc darbībām, un viens no galvenajiem C++ programmas strukturēšanas elementiem ir funkcijas. Tādējādi tradicionāli no strukturēšanas viedokļa darbības ir primāras, bet dati sekundāri. Galu galā arī procesors darbojas šādā veidā – pa soļiem veicot darbības.

Lielā daļā reālās pasaules problēmu tieši **datiem ir primāra nozīme**. Objektorientētā programmēšana nodrošina šo principu programmēšanas līmenī, panākot, ka programma vispirms tiek strukturēta pēc datiem un tikai tad pēc darbībām.

> **Objektorientētā programmēšana** (*object-oriented programming*) ir programmēšanas paradigma, kuru raksturo tas, ka dati un darbības tiek apvienoti kopējās struktūrās, kur dati ieņem primāro lomu.

Tehniski objektorientēto programmēšanu reprezentē klases un objekti.

> **Objekts** (*object*) programmā ir datu kopums kopā ar tām piesaistītajām darbībām. Datus objektā reprezentē **lauki** (*fields*) jeb **iekšējie mainīgie** (*member variables*), bet darbības reprezentē **metodes** (*methods*) jeb **iekšējās funkcijas** (*member functions*).

Turpmāk attiecīgi termini "lauki" un "iekšējie mainīgie", kā arī "metodes" un "iekšējās funkcijas" tiks lietoti kā sinonīmi.

Lai varētu izmantot objektus, vispirms ir jādefinē klase, kas ir viena tipa objektu īpašību apraksts.

> **Klase** (*class*) ir objekta tipa apraksts, kas ietver sevī iekšējo mainīgo deklarēšanu un iekšējo funkciju aprakstu, kā arī pieejas tiesību definēšanu mainīgajiem un funkcijām.

Nākošais piemērs satur vienkāršas klases definīciju:

```
class simple
{
public:
    int a;
    void print ()
    {
        cout << a << endl;
    }
}
```

Att. 13.1. Vienkārša klase ar vienu lauku un vienu metodi

Uz klases metodēm, tāpat kā uz parastajām funkcijām, attiecas **pārslogošana** (*overloading*), t.i., iespēja izmantot vienu nosaukumu vairāku metožu nosaukšanai.

Līdz ar jēdzienu klase, blakus jau zināmajiem – globālo un lokālo redzamības apgabalu (*scope*) parādās jauns redzamības apgabals – klases redzamības apgabals. Atrašanās klases redzamības apgabalā nozīmē to, ka iekšējais mainīgais vai funkcija ir redzami tikai dotā objekta ietvaros.

Klases redzamības apgabalā atrodošies mainīgie un funkcijas pēc **pieejamības** (*accessibility*) iedalās šādās 3 grupās:

- **publiskie** jeb atklātie (*public*), kas ir pieejami arī no ārējiem moduļiem,
- **privātie** jeb slēptie (*private*), kas ir pieejami tikai no klases iekšējām funkcijām, kā arī no funkcijām vai klasēm, kas dotajai klasei definētas kā **draudzīgās** (*friend*),
- **aizsargātie** (*protected*), kas pieejami arī no **mantotajām** (*inherited*) klasēm.

Specifikatori *public*, *private* un *protected* darbojas no pieminēšanas brīža vai nu līdz nākošajam specifikatoram, kas to atceļ, vai nu līdz klases galvas beigām, ja vairs neviena specifikatora nav. Turklāt no klases galvas sākuma, ja nav uzrādīts savādāk, pēc noklusēšanas darbojas *private*. Šie 3 specifikatori, lietoti klases galvā, sadala klases galvu sektoros, kur katrā sektorā ir noteikts vienots pieejamības līmenis.

Tādējādi nākošajā piemērā (Att. 13.2) lauks *a* ir ar pieejamību *private*, bet abas metodes – *public*.

Principu, ka objektā var eksistēt elementi, kas nav pieejami no ārējiem moduļiem (ne-publiski elementi), tātad, ir pieejami tikai caur citām – publiskām, metodēm, sauc par **iekapsulēšanu** (*encapsulation*), kas ir viens no galvenajiem objektorientētas programmēšanas principiem.

Nākošais piemērs parāda to pašu klasi *simple* (ieskaitot tās izmantošanu), bet kur lauks a ir slēptais, no ārējiem moduļiem nav pieejams, tāpēc bija jāievieš speciālā funkcija tā aizpildīšanai.

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

class simple
{
    int a;
public:
    void set (int x) { a = x; }
    void print () { cout << a << endl; }
};

int main ()
{
    simple s;
    s.set (99);
    s.print();
    return 0;
}
```

```
99
```

Att. 13.2. Vienkārša klase ar vienu slēpto lauku un divām metodēm

Vienkāršota klases definīcija, pieņemot, ka klases definīciju pilnībā satur klases galva, parādīta nākošajā shēmā.

class_definition_simple:
    class_keyword class_name { *class_section_list* }

class_keyword: one of
    `class struct union`

class_section_list:
    class_section
    class_section class_section_list

class_section:
    $accessibility\_specificator_{opt}$ *class_element_list*

accessibility_specificator:
    accessibility_keyword :

accessibility_keyword: one of
    `public protected private`

class_element_list:
    class_element
    class_element class_element_list

class_element:
    member_function
    member_variable

Att. 13.3. Vienkāršota klases galvas definīcija (*class_definition_simple*)

Kā ir redzamas no sintaktiskās shēmas, atslēgas vārda *class* vietā var lietot gan *struct*, gan *union*. *union* nozīme ir ļoti specifiska, taču *struct* darbojas pilnīgi identiski *class*, ar vienu izņēmumu: ja *class* noklusētā pieejamība ir *private*, tad *struct* noklusētā pieejamība ir *public*.

13.2. Darbs ar objektiem

Objekti ir klases pārstāvji un tos definē gandrīz tikpat vienkārši kā citus mainīgos parastajiem datu tipiem.

Objekti programmā var parādīties divos veidos, kas atšķiras **pēc sintakses** un **atmiņas izmantošanas:**

- **tiešā veidā** (izmantojot automātisko atmiņu),
- dinamiskā veidā.

13.2.1. Objekta izmantošana tiešā veidā

Tiešā veidā izmantot objektus ir ļoti ērti no sintakses viedokļa. Tai pat laikā citās programmēšanas valodās (piemēram, *Java*) tiešā objektu izmantošana vispār nav iespējama, tā ir valodas C++ specifika un neapšaubāmi – priekšrocība.

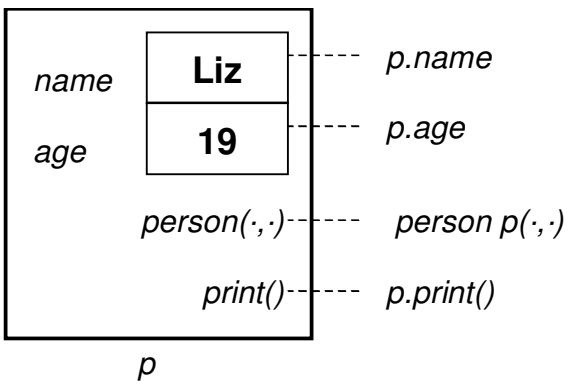
Objektu reprezentē mainīgais ar tipu "klase", un piekļuve pie objekta elementiem (laukiem un metodēm) notiek, izmantojot **punkta notāciju** (.). Objekts tiek automātiski izveidots pie mainīgā deklarēšanas un automātiski beidz pastāvēt programmas bloka beigās (kā tas pienākas "normāliem" lokālajiem mainīgajiem).

Piemērā ir definēta un izmantota arī speciāla klases metode ***person***, kas nosaukta klases vārdā. Tas ir **konstruktors**, kas paredzēts objekta aizpildīšanai pēc izveidošanas. Konstruktori tiks apskatīti nākamajās sadaļās.

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

class person
{
    char name[20];
    int age;
public:
    person (const char*, int);
    void print ()
    {
        cout << name << " " << age << endl;
    }
};

int main ()
{
    person p ("Liz", 19);
    p.print ();
};

person::person (const char *n, int a)
{
    strcpy (name, n);
    age = a;
}
```

```
Liz 19
```

Att. 13.4. Objekta izmantošana tiešā veidā



Att. 13.5. Objekta izmantošana tiešā veidā (atbilst programmai Att. 13.4)

13.2.2. Objekta izmantošana dinamiskā veidā

Jēga izmantot dinamiskus objektus ir tāda pati, kāda izmantot jau iepriekš apskatītos dinamiskos masīvus – lai dotu lietotājam (programmētājam) iespēju papildus kontrolēt resursu (šajā gadījumā, atmiņas) izmantošanu.

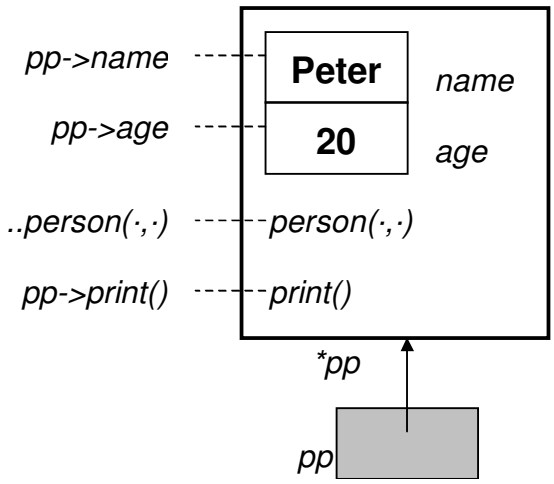
Objektu reprezentē mainīgais ar tipu "norāde uz klasi", un piekļuve pie objekta elementiem (laukiem un metodēm) notiek, izmantojot bultiņas notāciju (->). Objekts tiek izveidots, izmantojot operatoru *new* un likvidēts ar *delete*.

Piezīme. Daudzās citās programmēšanas valodās (*Java*, *Basic*) šis ir vienīgais iespējamais objektu izmantošanas veids (tiešā veida nemaz nav), tai pat laikā uzskatāmības dēļ šajās valodās bultiņas notācijas vietā tiek lietota punkta notācija, kas neapšaubāmi ir ērtāk lasāma un vieglāk uztverama.

```
...

class person
{
    ...
};

int main ()
{
    person *pp; // mainīgā deklarēšana
    pp = new person ("Peter", 20); // objekta izveidošana
    pp->print ();
    delete pp; // objekta likvidēšana
}
```

```
Peter 20
```

Att. 13.6. Objekta izmantošana dinamiskā veidā (klases pilnu definīciju sk. Att. 13.4)



Att. 13.7. Objekta izmantošana dinamiskā veidā (sk. programmu Att. 13.6)

13.3. Klases definēšana dalīti

13.3.1. Klases metožu definīciju iznešana ārpus klases galvas

Obligāta sastāvdaļa klases definēšanai (resp., uzprogrammēšanai) ir **hederis** jeb galvene (*header*) (Att. 13.1). Klases hederī tiek definēti visi lauki, tomēr klases metodes galvā var tikt uzrādītas divās detalizācijas pakāpēs:

- gan pilnībā (kā tas demonstrēts līdz šim Att. 13.2, Att. 13.4),
- gan var tikt norādīti tikai to prototipi (Att. 13.9), lai pēc tam metodi realizētu citur.

Otrais variants ir pēc līdzības ar iespēju C++ funkcijai izšķirt deklarāciju un realizāciju. Definējot klases metodi dalīti, klases **galvā paliek tikai metodes prototips, bet realizācija tiek iznesta ārpusē**.

Klases metodes realizācija ārpus klases hedera ir gandrīz tāda pati kā parastas funkcijas realizācija ar vienu izņēmumu – tā kā metode nav neatkarīga funkcija, tad, atrodoties ārpus galvas, metodei ir jāpieliek papildus informācija – pie kuras klases tā pieder. To dara, izmantojot redzamības atrisināšanas operatoru ::

```
method_realization_external:
        class_method_head  function_body

class_method_head:
        return_type_opt  class_method_id （ parameter_def_list_opt ）

class_method_id:
        class_name ：： function_name
```

Att. 13.8. Klases metodes realizācija ārēji (*method_realization_external*)

Klases metodes definīcijā atgriežamais tips ir ielikts kā izvēle, jo ir daži metožu veidi, kam tas netiek uzrādīts (konstruktors, destruktors, par kuriem skatīt tālāk).

```
// klases galva
class person
{
    char name[20];
    int age;
public:
    person (const char*, int); // metodes deklarācija
    void print (); // metodes deklarācija
};

...

person::person (const char *n, int a) // metodes realizācija
{
    strcpy (name, n);
    age = a;
};

void person::print () // metodes realizācija
{
    cout << name << " " << age << endl;
}
```

Att. 13.9. Klases metožu definēšana ārpus klases galvas

13.3.2. Klases dalīšana vairākos failos

Klasi var dalīt vairākos failos pēc līdzīga principa, kā tas darāms funkcijām. Klases galvu, katras atsevišķas metodes ārējo realizāciju un izsaukumus var likt atsevišķos failos. Klases hederis parasti tiek ievietots hedera (**.h*) failā, kuru pēc tam tieši vai netieši iekļauj visi moduļi, kuros klase tiek realizēta vai kur tiek izmantoti tās objekti.

Piemēram, iepriekšējā programma pa failiem būtu strukturējama šādi:

Att. 13.10. Klases metožu definēšana ārpus klases galvas.

person.h: klases galva

```
class person
{
    ...
};
```

person.cpp: klases metožu realizācija

```
#include "person.h"
person::person (...)
{
   ...
```

personmain.cpp: klases izmantošana

```
#include "person.h"
int main ()
{
    person p (...);
    ...
}
```

Ievērojiet, ka abos C++ failos tiek iekļauts “person.h” fails.

13.4. Konstruktori un destruktori

Konstruktors un destruktors ir svarīgas klases metodes, kuras piedalās attiecīgi objekta izveidošanā un likvidēšanā. Valodā C++ konstruktora nosaukums sakrīt ar klases nosaukumu, destruktora nosaukums gandrīz sakrīt ar klases nosaukumu.

13.4.1. Konstruktors

13.4.1.1. Konstruktora izveidošana un izmantošana

> **Konstruktors** (*constructor*) ir specifiska klases iekšējā funkcija, kas automātiski (netieši) tiek izsaukta uzreiz pēc objekta izveidošanas.

No vienas puses, konstruktors ir tāda pati vien metode kā citas, no otras puses, konstruktoram ir pietiekoši daudz atšķirību no citām metodēm:

- konstruktora izsaukšana notiek īpašā veidā,
- īpaša sintakse, konstruktoru definējot un izmantojot,
- īpaša shēma konstruktoru mantošanā (sk. mantošanu nākošajā nodaļā).

Valodā C++ konstruktors tiek saukts klases vārdā:

```
class person
{
    ...
    person (const char *n, int a)
    ...
};
```

Konstruktoram tiek padotas vērtības, ar kurām inicializēt jaunizveidoto objektu.

Klasē var būt vairāki konstruktori, un tie, tāpat kā citas funkcijas ar vienādu nosaukumu, atšķiras pēc parametru virknes.

Konstruktors objektam tiek izsaukts netieši, un konstruktora izsaukšana atšķiras atkarībā no objekta izmantošanas veida.

- Ja objekts tiek izmantots tiešā veidā, tad konstruktors tiek izsaukts pie objekta deklarēšanas.

```
class person
{
    ...
};
...
person p ("Liz", 19);
```

- Ja konstruktoram nav parametru, tad tukšās iekavas drīkst nelikt.

```
class simple
{
    ...
```

```
};
...
simple s; // tā vietā lai: simple s();
```

- Ja objekts ir dinamisks, tad konstruktors tiek netiešā veidā izsaukts, izsaucot operatoru *new*:

```
class person
{
    ...
};
...
person *pp = new person ("Peter", 20);
```

13.4.1.2. Konstruktoru veidi

Noklusētais konstruktors.

Katrā klasē ir vismaz viens konstruktors. Ja konstruktors nav definēts, tad kompilators automātiski izveido noklusēto konstruktoru (*default constructor*) bez parametriem ar tukšu ķermeni (resp., kurš neko nedara).

Ja klasē ir definēts (uzprogrammēts) **kaut viens** konstruktors, tad noklusētais konstruktors automātiski netiek veidots un, ja ir nepieciešamība pēc konstruktora bez parametriem, tad tas obligāti jādefinē pašam programmētājam:

```
class person
{
    ...
    person (const char *n, int a)
    ...
    person () {}; // tukšais konstruktors, izveidots tieši
    ...
};
```

Kopijas konstruktors.

Kopijas konstruktors ir viens no svarīgākajiem konstruktoru veidiem, un tam ir ļoti specifiska nozīme. Kopijas konstruktors ir konstruktors ar vienu parametru, kur parametrs ir ar tipu reference vai konstanta reference uz šīs pašas klases objektu.

Kopijas konstruktora būtība ir dot iespēju izveidot vienu objektu pēc cita objekta parauga.

Tomēr kopijas konstruktora lielākā specifika saistīta nevis ar tā definēšanu, bet gan ar tā izmantošanu, kā arī izmantošanas sintaksi (sk. zemāk abus gadījumus).

```
class person
{
    ....
    person (const person &p)
    {
        strcpy (name, p.name);
        age = p.age;
    };
    ...
};
```

Kopijas konstruktoru bez parastā izmantošanas veida izsauc vēl šādos gadījumos:

- **inicializējot objektu** ar citu objektu pie deklarēšanas,

```
person p ("Liz", 19);
person q = p; // tas pats, kas: person q (p);
```

- **nododot objektu** (bet nevis tā referenci) kādai funkcijai **kā parametru** – tādā gadījumā funkcijas vajadzībām tiek veidots jauns lokāls objekts, izsaucot kopijas konstruktoru.

```
void print (person p) // ārēja funkcija
{                     // izsauc kopijas konstruktoru
    p.print();        // lai izveidotu dublikātu
};
   ...
person p ("Liz", 19);
print (p); // objekta padošana funkcijai pēc vērtības
```

Citi konstruktori ar vienu parametru.

Arī citi konstruktori ar vienu parametru darbojas tāpat kā kopijas konstruktors – tos izsauc pie inicializēšanas ar piešķiršanu vai caur parametru, kura tips sakrīt ar šī konstruktora parametra tipu. Uz tiem attiecas tā pati izmantošanas specifika, kas uz kopijas konstruktoru.

```
class person
{
    ....
    person (const char *n)
    {
        strcpy (name, n);
        age = 18;
    };    ...
};
...
person p = "Ann"; // konstruktors ar vienu parametru
p.print (); // Izdrukā: Ann 18
```

Objekta lauku inicializēšanas speciālā sintakse konstruktorā.

Blakus tradicionālajai lauku inicializēšanas iespējai konstruktorā ir pieejama arī speciāla sintakse, kur lauki tiek inicializēti pirms konstruktora ķermeņa – konstruktora galvā. Šī sintakse ir alternatīva parastajai, un nav jāizmanto obligāti, varētu pat teikt, ka tās izmantošana ir tīri gaumes jautājums.

Tā vietā, lai definētu konstruktoru:

```
class numbers
{
    int num1;
    int num2;
    ...
    numbers (int n1, int n2)
    {
```

```
            num1 = n1;
            num2 = n2;
            ...
        };
```

var rakstīt tā:

```
    class numbers
    {
        int num1;
        int num2;
        ...
        numbers (int n1, int n2): num1(n1), num2(n2)
        {
            ...
        };
```

13.4.2. Destruktors

> **Destruktors** (*destructor*) ir specifiska klases iekšējā funkcija, kas automātiski tiek izsaukta tieši pirms objekta likvidēšanas.

Destruktora nosaukums valodā C++ ir vienāds ar klases nosaukumu, kam priekšā pielikts tildes simbols '~' (piemēram, *~person*). Destruktoram nav parametru, tādējādi klasei var būt tikai viens destruktors.

Ja par konstruktoru ir diezgan skaidrs, kāpēc tam atrasties katrā klasē, tad destruktoram ir jēga tikai tad, ja klase izmanto kaut kādus (dinamiskus) resursus, kuri, objektam likvidējoties, ir jāatbrīvo (piemēram, jāatbrīvo dinamiski izdalītā atmiņa vai jāaizver fails).

```
    class person
    {
        ....
        ~person () // destruktors
        {
            ...
        };
        ...
    };
```

Destruktora izsaukšana atšķiras atkarībā no objekta izmantošanas veida.

- Ja objekts tiek izmantots tiešā veidā, tad destruktors tiek automātiski izsaukts tā bloka beigās, kad objekts deklarēts.

```
    {
        person p (...);
        ...
    }  // destruktors automātiski nostrādā šeit
```

- Ja objekts ir dinamisks, tad destruktors tiek netiešā veidā izsaukts ar operatoru *delete*.

```
    {
```

```
        person *pp = new person (...);
        ...
        delete pp; // destruktors nostrādā šeit
    }
```

13.5. Klases statiskie elementi

Klases **statiskie elementi** (mainīgie un funkcijas) ir tādi klases elementi, kas ir kopīgi visiem klases objektiem neatkarīgi no to skaita. Vēl vairāk – tie eksistē arī tad, ja nav izveidots neviens klases objekts. Klases statiskos elementus parasti izmanto, lai izvairītos no globālajiem mainīgajiem (kas pēc būtības arī ir statiskie lauki) un ārpus klases funkcijām.

Pie statiskajiem klases elementiem var vērsties gan tradicionālajā veidā, kādam klases objektam piemērojot attiecīgi operatorus '.' vai '->', gan neatkarīgi, pirms attiecīgā mainīgā vai funkcijas pievienojot klases vārdu ar redzamības atrisināšanas operatoru ::.

```
        cout << months::get_month_name (1);
```

Klases statisko elementu nosaka modifikatora ***static*** pievienošana pirms elementa klases galvā.

```
    class months
    {
        static char monthnames[][10];
    public:
        static char* get_month_name (int num);
    };
```

Statiskā lauka definēšana.

Papildus tam, ka lauks klases galvā tiek deklarēts kā statisks, ārpus klases galvas tas vēl papildus ir jādefinē (uzreiz klases galvā to darīt nedrīkst):

```
    char months::monthnames[][10] = {"january", "february",
                      "march"};
```

Statiskas metodes definēšana.

Statisku metodi realizē tieši tāpat kā parastu ārpus klases funkciju, t.i., šajā funkcijā tiešā veidā nav pieejami klases elementi, kas nav statiski, jo pati metode nepieder nevienam objektam.

```
    char* months::get_month_name (int num)
    {
        return monthnames[num];
    }
```

13.6. Objekta norāde uz sevi 'this' un klases deklarēšana

13.6.1. Norāde uz sevi 'this'

Jebkura klases metode (kas nav statiska) var piekļūt objektam, kuram tā ir izsaukta. To var izdarīt ar speciālu norādes mainīgo *this*, kura tips ir *klases_vārds**. Parasti norāde *this* nav

nepieciešama, jo klases metodēm ir tieša piekļuve dotā objekta elementiem. Tomēr atsevišķos gadījumos bez tās nevar iztikt, piemēram, lai nodotu objektu kādai ārējai funkcijai.

Nākošais piemērs parāda ārēju funkciju *print_external*, kas prot izdrukāt personas objekta saturu. Šo ārējo funkciju izsauc klases iekšējā funkcija *print*, padodot sevi, tam izmantojot norādi *this* (tā kā arējā funkcija pieņem parametru, kas nav norāde, tad vērtība, kas jāpadod, ir **this*, nevis vienkārši *this*).

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

class person;

void print_external (const person&);

struct person
{
    string name;
    int age;
    void print () const { print_external (*this); };
};

int main ()
{
    person p;
    p.name = "Liz";
    p.age = 19;
    p.print ();
    return 0;
};

void print_external (const person &p)
{
    cout << p.name << " " << p.age << endl;
}
```

```
Liz 19
```

Att. 13.11. Norāde *this*

13.6.2. Klases deklarēšana

Klases deklarēšana ir paziņojums, ka klase ar doto vārdu tiks izmantota programmā (Att. 13.11):

```
class person;
```

Parasti klasi atsevišķi nedeklarē, bet deklarēšana automātiski notiek, definējot klases hederi. Tomēr atsevišķos gadījumos ir nepieciešams veidot struktūras vai funkcijas, kurās ietilpst dotā klase, vēl pirms klases hedera definēšanas (Att. 13.11):

```
void print_external (const person&);
```

Bez klases deklarēšanas nevar iztikt gadījumos, kad klase un kāds ārējs elements (piemēram, cita klase vai funkcija) izmanto viens otru, kā tas ir arī ir redzams Att. 13.11 – klase *person* izmanto ārējo funkciju *print_external*, bet šīs funkcijas savukārt izmanto šo klasi sava parametra tipa veidošanai.

14. Programmēšanas stils

14.1. Programmēšanas stils

14.1.1. Programmēšanas stils un tā nozīme

Rakstot programmu, ir svarīgi, lai tā strādātu pareizi un vēlams arī efektīvi, tomēr arī programmas pierakstīšanas stilam ir ļoti liela nozīme programmēšanā – programmētājs ir cilvēks, un tieši viņam ir jāskatās uz programmu, jāsaprot, kas programmā ir uzrakstīts, jāveic labojumi, un cilvēka psiholoģiskās īpašības nosaka to, ka informācijas pasniegšanas formai un veidam ir būtiska nozīme. Pareizā formā vai veiksmīgā stilā uzrakstīta programma ne tikai palīdz nākošajā dienā saprast, kas programmā ir bijis domāts, bet arī sekmē pašu domāšanas procesu, nodrošinot ne tikai kvalitatīvāku, bet arī ātrāku programmatūras izstrādes procesu. Tiesa gan, kamēr rakstītās programmas ir mazas, noteikta programmēšanas stila uzturēšana liekas tikai lieka laika izšķiešana. Varbūt atsevišķas stila detaļas varētu nenest gaidīto efektu, tomēr ir tādas, kuras būtu obligātas, rakstot jebkuru programmu.

> **Programmēšanas stils** ir programmas teksta veidošanas paņēmienu kopums, kas nodrošina tādas programmas iegūšanu, kas ir viegli lasāma, saprotama un modificējama.

Programmēšanas stils izpaužas gan kā noteiktu likumu ievērošana, gan kā attieksme pret programmas koda veidošanu.

Stils ir atkarīgs ne tikai no programmētāja, bet lielā mērā ietekmējas no programmēšanas valodas. Labai programmēšanas valodai no stila viedokļa piemīt šādas īpašības:

- **Skaidrība un ortogonalitāte** (*clarity and orthogonality*). Konstrukcijas ir skaidri saprotamas visās situācijās un nedod negaidītus efektus, tās kombinējot vienu ar otru.
- **Laba rakstāmība** (*writability*). Laba notācija palīdz rakstīt programmu.
- **Laba lasāmība** (*readability*). Programmas jēgu var viegli saprast bez papildus dokumentācijas.
- **Viennozīmīgums** (*no tricks*). Grūti izmantot konstrukcijas nepareizā nozīmē.

Valoda C++ pieder pie šādām valodām. Valodā C++ var uzrakstīt programmu, kas ir labi saprotama un viegli lasāma. Ja tas tā nav, tā jau ir programmētāja vaina:

- slikti izvēlētas datu struktūras un algoritms,
- slikts programmēšanas stils.

14.1.2. Programmēšanas stila noteikumi

14.1.2.1. Vispārzināmie programmēšanas stila noteikumi

- Izmantojiet atkāpes.
- Izmantojiet programmas prologu, lai dokumentētu svarīgākās īpašības:
  - nosaukums,
  - autors,
  - pirmās un pēdējās kompilācijas datums,

- attīstības vēsture (datums, autors, izmaiņas, mērķis, komentārs), versiju nozīme (versijas saglabāt!),
- parametru apraksts,
- veicamo funkciju apraksts,
- dažādu režīmu un stāvokļu raksturojums,
- izņēmuma situāciju raksturojums,
- blakus efekti,
- izsaucamās apakšprogrammas.

- Izmantojiet komentārus.
- Ierobežojiet atsevišķu programmas bloku (funkciju) izmērus (piemēram, 1 lappuse).
- Izmantojiet saturīgus identifikatorus mainīgajiem, funkcijām, klasēm, struktūrām.
- Izvairieties no *goto*.

14.1.2.2. Atkāpes un bloki

- Atkāpes ir jāizmanto obligāti.
- Atkāpes jāizmanto tā, lai tās norādītu uz programmas elementu hierarhisko struktūru.
- Iekļaut figūriekavās arī blokus no viena priekšraksta, lai izvairītos no potenciālām kļūdām, modificējot programmu.
- Figūriekavu blokus veidot tā, lai pēc bloka sākuma varētu viegli noteikt bloka beigas – vislabākais, lai aizverošā figūriekava būtu tieši zem attiecīgās atverošās.

Tālāk tiks uzskaitīti daži figūriekavu bloku veidošanas stili.

Stils nr 1.

Atverošā figūriekava tieši zem galvas, pakārtotās instrukcijas ar vienu atkāpi, aizverošā figūriekava tieši zem atverošās.

```
Konstrukcijas galva
{
    Instrukcija
    ....
}
```

Stils nr 2.

Atverošā figūriekava tieši aiz galvas, pakārtotās instrukcijas ar vienu atkāpi, aizverošā figūriekava tieši zem galvas.

```
Konstrukcijas galva {
    Instrukcija
    ....
}
```

Stils nr 3.

Atverošā figūriekava ar vienu atkāpi zem galvas, pakārtotās instrukcijas atverošās figūriekavas līmenī, aizverošā figūriekava tieši zem atverošās.

```
Konstrukcijas galva
    {
    Instrukcija
```

```
....
}
```

Stils nr 4.

Atverošā figūriekava ar vienu atkāpi zem galvas, pakārtotās instrukcijas vēl ar vienu atkāpi, aizverošā figūriekava tieši zem atverošās.

```
Konstrukcijas galva
    {
        Instrukcija
        ....
    }
```

14.1.2.3. Identifikatoru darināšana

- Mainīgo, funkciju, klašu un struktūru vārdus veidojiet saturīgus, lai pēc tiem būtu iespējams noteikt attiecīgā elementa nozīmi programmā, tai pat laikā identifikatori nedrīkst būt pārmērīgi gari.
- Nosakiet un konsekventi lietojiet noteiktu stilu dažādu elementu vārdu darināšanā.
- Izņēmums varētu būt vienīgi mainīgie, kas ir skaitītāji – tiem var atvēlēt arī pavisam īsus nosaukumus, piemēram, i, j, k.

Identifikatoru darināšanai ir dažādi stili.

Stils nr. 1.

Identifikatori sastāv no vairākiem vārdiem. Mainīgajiem katrs vārds, kas ietilpst identifikatorā, sākas ar lielo burtu, bet atlikušie burti vārdā ir mazie. Funkcijas darina tāpat, bet pirmais no vārdiem ir viss ar mazajiem burtiem. Klases un datu struktūras – tikai ar mazajiem burtiem. Piemērs:

```
VariableName
functionName
classname
```

Stils nr. 2.

Identifikatori sastāv no vairākiem vārdiem. Visi vārdi ir ar mazajiem burtiem, bet starp vārdiem ir pasvītrojuma zīme. Piemērs:

```
variable_name
function_name
class_name
```

Stils nr. 3.

Mainīgo nosaukumus sāk ar mainīgā datu tipa saīsinājumu:

```
intVariable
dblVariable
strVariable
chrVariable
```

14.1.2.4. Izteiksmju veidošana

- Rakstiet skaidri un vienkārši.
- Izmantojiet iebūvētās funkcijas.
- Efektivitātes dēļ neupurējiet skaidrību.

- Ļaujiet mašīnai darīt melno darbu.
- Nomainiet atkārtotas izteiksmes ar funkciju izsaukumiem.
- Izmantojiet iekavas, lai izvairītos no divdomīgām situācijām.

14.1.2.5. Programmas struktūra un vadības sistēma

- Priekšrakstu virknes iekļaujiet iekavās, novietojot aizverošo iekavu zemākā rindiņā zem atbilstošās atverošās iekavas.
- Lietojiet *if-else*, lai parādītu to, ka iespējamas tikai divas darbības.
- Lietojiet cikla priekšrakstu, lai veiktu atkārtojošās darbības.
- Rakstiet programmu tā, lai to varētu lasīt no augšas uz apakšu.
- Rakstiet vispirms viegli saprotamā pseidokodā (piemēram, grafiskā valodā), pēc tam pārveidojiet vajadzīgajā programmēšanas valodā.
- Izvairieties lietot iekļautus *if* bez *else*.
- Novietojiet darbības pēc iespējas tuvu to izpildes nosacījumiem.
- Izvēlieties atbilstošu datu tipu vadības nodrošināšanai.
- Lietojiet noturīgas konstrukcijas vienota stila saglabāšanai.
- Mēģiniet uzlabot programmas stilu.
- Lietojiet funkcijas.
- Katrai funkcijas jādara viens darbs, bet labi.
- Katrai funkcijai kaut kas ir "jānoslēpj".
- Atļaujiet datiem diktēt programmas struktūru.
- Nelabojiet sliktu programmu – pārrakstiet to.
- Lielu programmu rakstiet un pārbaudiet pa mazām daļām.
- Lietojiet rekursīvas funkcijas rekursīvi definētu datu apstrādei.

14.1.2.6. Ievade un izvade

- Pārbaudiet ieejas datus.
- Paziņojiet par ievades kļūdām.
- Plānojiet vienkāršu datu ievadi.
- Veidojiet vienveidīgu ieejas datu ievadi.
- Veidojiet brīva formāta ieejas datu ievadi, ja vien iespējams.
- Novietojiet ievadi/izvadi funkcijās.

14.1.2.7. Dažādi ieteikumi

- Inicializējiet mainīgos pirms to izmantošanas.

- Nelietojiet vairākas izejas no cikla.
- Pārbaudiet programmu uz robežvērtībām.
- Programmējiet, aizsargājoties no iecikošanās.
- Nepārbaudiet peldošus skaitļus uz vienādību, jo tie aptuveno darbību dēļ var tieši nesakrist.
- Rakstiet vispirms pareizi un skaidri, tikai pēc tam uzlabojiet.
- Saglabājiet pareizību, uzlabojot programmu.
- Neupurējiet skaidrību, ja varat iegūt tikai nelielu ieguvumu efektivitātē.
- Pārbaudiet, vai speciālie gadījumi tiešām ir jāapstrādā speciāli.

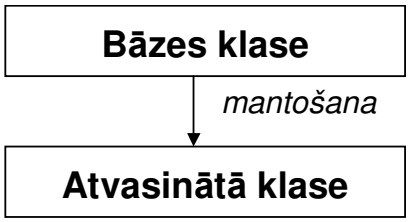
15. Mantošana un citi objektorientētās programmēšanas mehānismi

15.1. Mantošanas pamatprincipi

> **Mantošana** (*inheritance*) ir objektorientētās programmēšanas mehānisms, kad klase pārņem (manto) elementus (laukus un metodes) no jau eksistējošas klases vai pat vairākām klasēm.

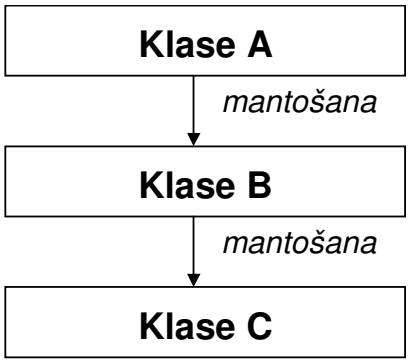
Mantošana ir vēl viens mehānisms pirmkoda vairākkārtējai izmantošanai, kas papildina koda strukturēšanas iespējas, kā arī nodrošina ērtāku uzturēšanu.

Klases, no kurām mantošanas rezultātā tiek iegūti elementi, sauc par **bāzes klasēm** (*base class*), bet klasi, kura tiek veidota, izmantojot mantošanu – par **atvasināto klasi** (*derived class*) vai **mantoto** (*inherited*) klasi.



Att. 15.1. Vienkārša mantošanas shēma

Mantošanas process var būt kaskādes veida, līdz ar to viena un tā pati klase var būt gan atvasinātā klase, gan bāzes klase citām klasēm.



Att. 15.2. Kaskādes veida mantošanas shēma

Klases mantošana notiek, pēc klases nosaukuma uzrādot klasi vai klases, no kurām tiks mantots (pēc līdzīgas sintakses, kā konstruktorā alternatīvajā variantā tiek inicializēti lauki).

```
class A
{
    ...
};
...
class B: public A
{
    ...
};
```

Att. 15.3. Mantošanas shēma: klase B manto īpašības no klases A

Mantošanas rezultātā visi A lauki un metodes automātiski ir arī klases B elementi, tomēr ir dažādi izņēmumi un sarežģījumi, par kuriem runāts tālāk.

class_definition_inheritance:
    class_keyword class_name inherit_list$_{opt}$ { *class_section_list* }

class_keyword: one of
    `class struct union`

inherit_list:
    : *base_class_list*

base_class_list:
    base_class
    base_class_list base_class

base_class:
    inheritance_degree class_name

inheritance_degree: one of
    `public private`

Att. 15.4. Vienkāršota klases galvas definīcija ar mantošanu (*class_definition_inheritance*)

Šajā shēmā nav precizēts jēdziens *class_section_list*, kas ir tāds pats, kā iepriekšējā nodaļā vienkāršotajā klases definēšanas shēmā.

Nākošajā programmas piemērā klase *twonumbers* tiek mantota no klases *number*, tāpēc satur gan lauku a, gan lauku b, turklāt, definējot klases *twonumbers* metodi print, ar noteiktu sintaksi ir izmantojama klases *number* metode *print*, lai arī tajā atrodošais nebūtu otrreiz jāprogrammē.

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

struct number // bāzes klase ar vienu lauku a
{
    int a;
    void print () const { cout << a << endl; };
};

struct twonumbers: public number // atvasinātā klase
{
    // mantošanas dēļ šeit ir arī lauks a, kā arī
    // metode number::print(), kas izdrukā lauku a
    int b;
    void print () const
    {
        number::print (); // bāzes klases print!
        cout << b << endl;
    };
};

int main ()
{
    twonumbers num;
    num.a = 5;
    num.b = 7;
    num.print ();
    return 0;
}
```

```
5
7
```

Att. 15.5. Vienkāršs klases mantošanas piemērs (*twonumbers* manto *number*)

15.2. Mantošanas procesu nodrošinošie mehānismi

Mantošanas mehānisma izmantošana izsauc nepieciešamību pēc vairākām papildus konstrukcijām un mehānismiem objektorientētajā programmēšanā:

- modifikators *protected* paplašina *private* redzamību arī uz mantotajām klasēm,
- konstruktoru kaskādes veida izsaukšana,
- *destruktoru* kaskādes veida izsaukšana,
- atklātā un slēptā mantošana,
- virtuālās funkcijas.

15.2.1. Konstruktoru kaskādes veida izsaukšana

Pirms izsaukt atvasinātas klases konstruktoru, obligāti jāveic arī bāzes klases konstruktora izsaukšana. Tādējādi tiek panākts, ka, izveidojot objektu, tiek izsaukti arī visu bāzes klašu konstruktori mantošanas secībā, un tikai tad pašas klases konstruktors. Šo kaskādes shēmu valoda C++ nodrošina sintakses līmenī.

Konstruktora galva ar bāzes klases konstruktora izsaukšanu:

```
twonumbers (int x, int y): number (x)
```

Tā kā klasē var būt vairāki konstruktori, tad, definējot atvasinātas klases konstruktoru, ar īpašu sintaksi tiek norādīts, kuru no bāzes klases konstruktoriem pirms tam izsaukt. Ja bāzes klases konstruktors netiek uzrādīts, tad bāzes klasē jāeksistē konstruktoram bez parametriem, kas tādā gadījumā tiek izsaukts.

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

class number
{
    int a; // tiešā veidā nav pieejams pat klasei twonumbers
public:
    number (int x) { a = x; };
    void print () const { cout << a << endl; };
};

class twonumbers: public number
{
    int b;
public:
    // tiek izsaukts bāzes klases konstruktors number(x)
    twonumbers (int x, int y): number (x)
    {
        b = y;
    };
    void print () const
    {
        number::print ();
        cout << b << endl;
    };
};

int main ()
{
    twonumbers num (5, 7);
    num.print ();
    return 0;
}
```

```
5
7
```

Att. 15.6. Konstruktoru kaskādes veida izsaukšana

Pievērsiet uzmanību atvasinātās klases konstruktoram:

```
twonumbers (int x, int y): number (x)
{
    b = y;
};
```

Ja konstruktors būtu realizēts nevis šādi, bet gan sekojoši (tiešā veidā aizpildot visus laukus):

```
twonumbers (int x, int y)
{
```

```
        a = x;
        b = y;
    };
```

tad dotajā kontekstā (sk. pilnu klases definīciju) iestātos divas kļūdas situācijas:

- tā kā aiz atvasinātās klases konstruktora nav uzrādīts bāzes klases konstruktors (bet tā izsaukšana ir obligāta), kompilators mēģina izsaukt noklusēto konstruktoru (bez parametriem), bet tāds bāzes klasē nav definēts (tur ir tikai konstruktors ar vienu parametru):
  `no matching function for call to `number::number()'`
- tiešā veidā tiek piekļūts mainīgajam *a*: *a* = *x*, kas ir definēts bāzes klasē kā *private*, tāpēc iestājas piekļūšanas kļūda:
  `` `int number::a' is private ``

15.2.2. Destruktoru kaskādes veida izsaukšana

Arī destruktori, tāpat kā konstruktori, likvidējot objektu, tiek izsaukti kaskādes veidā, tikai:

- izsaukšana notiek pretēji mantošanas secībai – vispirms pašas klases destruktors, tad bāzes klašu destruktori utt.,
- destruktoru kaskādes veida izsaukšana nav speciāli jādefinē, jo, sakarā ar to, ka katrā klasē ir tikai viens destruktors, izsaukšana notiek automātiski.

15.2.3. Atklātā un slēptā mantošana

Atklāto un slēpto mantošanu nosaka attiecīgi atslēgas vārdi *public* un *private* pirms bāzes klases nosaukuma, veicot mantošanas procesu (sk. jēdzienu *inheritance_degree* Att. 15.4).

- ja tiek mantots atklāti (*public*, kā piemērā parādīts), tad pieeja pie bāzes klases elementiem paliek tāda pati arī mantotajā klasē (attiecīgi *public* vai *protected*, jo *private* elementi tāpat nav redzami),
- ja tiek mantots slēpti (*private*, piemēros nav parādīts), tad visi bāzes klases elementi nonāk mantotajā klasē *private* statusā, tātad ir pieejami tikai no mantotās klases.

15.2.4. Virtuālās funkcijas

Parasti mantošanu ir jēga izmantot tad, ja bāzes klasei ir vairāk nekā viena atvasinātā klase vai arī ja arī pašai bāzes klasei tiek veidoti objekti, jo tikai šajā gadījumā tā īsti parādās pirmkoda koplietošanas princips.

Bieži rodas nepieciešamība no bāzes klases mantoto klašu un varbūt arī pašas bāzes klases objektus glabāt vienotā struktūrā, piemēram, kopējā masīvā. C++ dod tādu iespēju – pie mainīgā ar tipu "norāde uz bāzes klasi" drīkst veidot objektus ar tipu "atvasinātā klase" (klašu definīcijas sk. attēlā Att. 15.6):

```
    ...
struct number
{
    ...
class twonumbers: public number
{
```

```
    ...
int main ()
{
    number* nums[2];
    nums[0] = new number (99);
    nums[1] = new twonumbers (5, 7);
    nums[0]->print ();
    nums[1]->print ();
...
```

```
99
5
```

Kā redzams, komanda

```
nums[1]->print ();
```

acīmredzami ir nozīmējusi bāzes klases *print()* izsaukumu, jo nav izdrukāts skaitlis 7.

Mainīgo tipa (šeit, norāde uz *number*) un objekta tipa (šeit, *twonumbers*) neatbilstība var radīt problēmas gadījumos, ja mantotajā klasē **tiek pārdefinētas bāzes klases funkcijas** (šeit, funkcija *print()*) – rodas problēmas saprast, kuru funkciju izsaukt, vai bāzes klases vai atvasinātās klases attiecīgo funkciju. Standarta tipa noteikšanas mehānisms ir "pēc mainīgā tipa", tādējādi, pielietojot šo mehānismu, tiek izsaukta objektam neatbilstoša funkcija.

Lai novērstu šo mainīgā un objekta tipa neatbilstības problēmu, bāzes klasē attiecīgā funkcija jāatzīmē kā **virtuāla**, pirms funkcijas pielietojot atslēgas vārdu *virtual* (pilnas klašu definīcijas sk. attēlā Att. 15.6):

```
    ...
struct number
{
    ...
    virtual void print () ...
    ...
class twonumbers: public number
{
    ...
int main ()
{
    number* nums[2];
    nums[0] = new number (99);
    nums[1] = new twonumbers (5, 7);
    nums[0]->print ();
    nums[1]->print ();
...
```

```
99
5
7
```

Un 7 tiek izdrukāts!

Ievērojiet, ka atslēgas vārds *virtual* jāpielieto bāzes klasei, nevis atvasinātajai.

15.3. Citi objektorientētās programmēšanas mehānismi

15.3.1. Draugu funkcijas un klases

Elementu redzamību klasē pamatā nodrošina modifikatori *public*, *protected* un *private*. Šie ir vispārīgas nozīmes modifikatori, kas nav atkarīgi no moduļa, no kura varētu tikt veikta pieeja klases elementiem.

Valodā C++ ir iespēja klases iekšienē definēt klases draugus – funkcijas vai klases, kam ir atļauta pieeja arī tiem klases elementiem, kas nav publiski. Drauga statusa uzstādīšanu nodrošina atslēgas vārds *friend* potenciālās drauga klases deklarācijas vai drauga funkcijas prototipa priekšā dotās klases hederī.

```
friend void print (number &num);
friend class another_class;
```

Tas nodrošina visu klases elementu pieeju no dotajiem moduļiem (funkcijām vai klases metodēm).

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

class number
{
    int a;
public:
    number (int x) { a = x; };
    friend void print (number &num);
};

void print (number &num)
{
    cout << num.a << endl;
}

int main ()
{
    number num (99);
    print (num);
    return 0;
}
```

```
99
```

Att. 15.7. Drauga funkcija

Šajā piemērā rindiņa:

```
friend void print (number &num);
```

ļauj ārējai (no klases *number* viedokļa) funkcijai *print* piekļūt klases *number* slēptajam (*private*) elementam *a*:

```
num.a
```

15.3.2. Operatoru pārslogošana

Valodā C++ ļoti plaši pielieto operatorus, turklāt ir iespēja tos, līdzīgi funkcijām, pārslogot, tādējādi pielāgojot darbam ar dažādiem tipiem. Pārslogot var gandrīz jebkuru operatoru, izņemot šos:

```
.   .*   ::   ?:
```

Sintaktiski iespēju pārslogot nodrošina **operatoru funkcionālais pieraksts**, kurā operators tiek izsaukts vai definēts kā parasta funkcija un līdz ar to arī ir pārslogojams kā parasta funkcija (ar nelielām specifiskām niansēm).

Operatora funkcionālais nosaukums sastāv no atslēgas vārda *operator*, kam seko operatora simbols vai simboli, piemēram, *operator+ operator*= operator() operator[]*

Piemēram, rindiņa

```
c = a + b;
```

funkcionālajā pierakstā izskatās šādi:

```
c = operator+ (a, b);
```

Operatoru pārslogošanai ir vairāki ierobežojumi un noteikumi:

- operatoru prioritātes un asociatīvās īpašības saglabājas un nav maināmas,
- operatoru uzvedība nav pārdefinējama iebūvētajiem tipiem (piemēram, *int*),
- pārdefinētajam operatoram jābūt saistītam ar lietotāja izveidotu klasi (vai nu jābūt šīs klases iekšējam operatoram, vai arī klasei jābūt par kāda parametra tipu),
- operatoriem nevar būt noklusētie parametri,
- operatori, tāpat kā citas funkcijas, mantojas, izņemot operatoru =.

Operatoru pārslogošana ir samērā specifiska katram operatoram, tomēr ir izdalāmi 2 galvenie veidi:

- operators kā **neatkarīga funkcija** – tādā gadījumā operatoram funkcionālajā pierakstā būs tikpat parametru, cik ir operandu,
- operators kā **klases iekšējā funkcija** – tādā gadījumā operatoram funkcionālajā pierakstā būs par vienu parametru mazāk nekā ir operandu, jo par pirmo operandu formāli kalpos objekts, kam šis operators tiek izsaukts.

Abu pārslogošanas veidu atšķirības izpaužas tikai pašā pārdefinēšanas procesā – no izmantošanas viedokļa dažādi pārslogotie operatori neatšķiras.

Parasti operatorus pārslogo otrajā veidā – kā klases iekšēju funkciju, tas palīdz strukturēt programmu, operatoru piesaistot noteiktai klasei:

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

class number
{
    int a;
public:
    number (int x) { a = x; };
    void print () { cout << a << endl; };
    number& operator+= (int i)
    {
        a += i;
        return *this;
    };
};

int main ()
{
    number num (99);
    num += 6;
    num.print ();
    return 0;
}
```

```
105
```

Att. 15.8. Operatora kā klases iekšējās funkcijas pārslogošana

Šajā piemērā tiek pārslogots operators +=. Šim operatoram ir divi operandi, tomēr, tā kā operators tiek pārslogots klases iekšienē, par pirmo operandu uzstājas pats objekts (**this*), un parametrs apzīmē labās puses operandu. Dotajā kontekstā šī operatora pārslogošanu varētu vienkāršot uz:

```
void operator+= (int i)
{
    a += i;
};
```

Atgriežamais tips *number&* un attiecīgi atgriešanas komanda *return *this* nepieciešami tikai tādēļ, lai šo operatoru varētu izmantot kaskādes veidā (nodrošinot, ka tas atgriež t.s. *lvalue* – vērtību, ko var likt piešķiršanas operatora kreisajā pusē), piemēram,

```
(num += 6) += 7;  // šeit atgrieztu 112
```

Lai šo pašu operatoru pārslogotu ārēji, papildus tas jāpasludina par klases *number* draugu, jo operators tiešā veidā vēršas pie *private* lauka *a*, protams, jāveic atbilstošas izmaiņas funkcijā sakarā ar to, ka operatora pirmais operands ir funkcionālā pieraksta pirmais parametrs, nevis objekts.

```
class number
{
    ...
    friend number& operator+= (number&, int);
};

number& operator+= (number &num, int i)
{
    num.a += i;
    return num;
};

    ...
number num (99);
num += 6;
    ...
```

```
105
```

Att. 15.9. Operatora pārslogošana ārēji (pārējā koda pilno tekstu sk. Att. 15.8)

Lai klasei *number* pārslogotu operatoru +=, mēs varējām izvēlēties, vai to darīt iekšēji vai ārēji, tomēr, ja operatora pirmais operands ir kāda standarta klase, tad nav citu variantu, kā vien pārslogot operatoru kā neatkarīgu funkciju, citādi to nevar izdarīt bez pašas standarta klases izmaiņas.

Tipisks piemērs šajā gadījumā ir izdrukas operatora pārslogošana.

Neapšaubāmi

```
number num (99);
cout << num << endl;
```

automātiski nestrādās, jo operators << nav pārslogots tipam *number*.

Izdrukas objekts *cout* pieder klasei *ostream*, un izdrukas operatora << pārslogošana ir diezgan specifiska nepieciešamo zināšanu dēļ par C++ standarta bibliotēkas struktūru, tomēr tehniski viss notiek atbilstoši tam, kā tiek pārslogots operators kā neatkarīga funkcija.

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

class number
{
    int a;
public:
    number (int x) { a = x; };
    friend ostream& operator<< (ostream&, number&);
};

ostream& operator<< (ostream &os, number &num)
{
    os << num.a;
    return os;
};

int main ()
{
    number num (99);
    cout << num << endl; // pārslogotā << izmantošana
    return 0;
}
```

```
99
```

Att. 15.10. Izdrukas operatora << pārslogošana (salīdzināt ar += pārslogošanu Att. 15.9)

16. Teksta failu apstrāde

16.1. Fails C++

16.1.1. Fails kā informācijas glabāšanas mehānisms

Kaut arī no vispārīga algoritmu konstrukcijas viedokļa failiem nav specifiskas lomas, tomēr sakarā ar pietiekoši lielo praktisko nozīmi un specifiku, failu apstrāde ir stingri standartizēta visās programmēšanas valodās.

> **Fails** (*file*) ir datu kopums, kas izvietots sekundārajā atmiņā un kas operētājsistēmas līmenī tiek identificēts ar noteiktu faila vārdu.

Tāpat kā operatīvajai atmiņai, arī failam **mazākā adresējamā vienība ir 1 baits**, tādējādi no datu apstrādes viedokļa var uzskatīt, ka fails ir baitu virkne.

Kaut arī darbs ar failiem valodā C++ ir standartizēts, tomēr ir atsevišķas nianses, kas nosaka atšķirības failu apstrādē dažādās operētājsistēmās, jo no programmas viedokļa fails ir operētājsistēmas pakalpojums.

Svarīgs jēdziens darbā ar failu ir **faila beigu pazīme** (*EOF*, *end of file*), kam ir līdzīga nozīme kā simbolu virknes beigu simbolam. Daudzos gadījumos (sevišķi valodā C++) arī faila beigu pazīmi var uztvert kā "faila beigu simbolu" (un tā ir vieglāk uztvert programmas būtību), tomēr vispārīgā gadījumā faila beigu pazīme ir cita abstrakcijas līmeņa jēdziens nekā faila dati.

Valodā C++ failu apstrādei jāiekļauj standartbibliotēka *<fstream>*.

```
#include <fstream>
```

Failu C++ programmā identificē **faila objekts**, kuru reprezentē **faila mainīgais**, kurš var būt viens no tipiem:

- *fstream* – lasīšanai un rakstīšanai,
- *ofstream* – tikai rakstīšanai,
- *ifstream* – tikai lasīšanai.

```
fstream f; // faila objekta f deklarēšana
                lasīšanai+rakstīšanai
fstream fin; // faila objekta fin deklarēšana lasīšanai
fstream fout; // faila objekta fout deklarēšana rakstīšanai
```

Galvenās darbības ar failu ir

- lasīšana (*reading*),
- rakstīšana (*writing*),
- pārbaude uz faila beigu (*EOF*) iestāšanos (lasot failu).

Lasīšana no faila var notikt ne tālāk kā līdz faila beigām, turpretī rakstīšana, ja tā notiek faila galā, automātiski pārbīda faila beigas uz priekšu.

Gan lasīšana, gan rakstīšana failā standarta variantā notiek secīgi (pēc kārtas) (parasti, sākot ar faila sākumu), tomēr ir pieejamas arī tiešās pieejas metodes.

16.1.2. Faila atvēršana un aizvēršana

Ņemot vērā to, ka fails ir operētājsistēmas pakalpojums, turklāt failu sistēmas līmenī tas tiek identificēts ar faila vārdu, turpretī programmā – ar faila mainīgo, faila apstrādē ir nepieciešamas 2 šādas papildus darbības:

- **Faila atvēršana** (*opening*) pirms darba sākšanas (fiziskā faila piesaiste faila mainīgajam un noteikta darba režīma uzstādīšana).
- **Faila aizvēršana** (*closing*), darbu beidzot.

Faila atvēršana un aizvēršana ir saistīta ar noteiktu, pietiekoši apjomīgu darbību veikšanu operētājsistēmas līmenī, jo **fails ir resurss, par kuru atbild operētājsistēma** (piemēram, lai divi lietotāji reizē nevarētu rakstīt vienā failā).

Faila atvēršana notiek, izmantojot metodi *open*:

```
    f.open ("test.txt");
```

Funkcija *open* var pieņemt divus parametrus, no kuriem otrais nav obligāts.

open

```
void open (const char* filename, int mode);
```

Faila objekta funkcija *open()* atver failu *filename* režīmā *mode*. Režīmu nosaka viena vai vairākas vērtības (atdalītas ar '|'). Režīma parametru var izlaist – tādā gadījumā faila atvēršana notiek noklusētajā režīmā atkarībā no faila objekta tipa.

Tab. 16.1.

Svarīgākās faila atvēršanas režīma vērtības

Režīms	Apraksts
`ios::in`	Nosaka lasīšanas režīmu. Ja fails ar doto nosaukumu neeksistē, tad faila atvēršana beidzas ar neveiksmi.
`ios::out`	Nosaka rakstīšanas režīmu. Ja fails eksistē, tad izdzēš visu tā saturu. Ja fails neeksistē, tad izveido jaunu failu. Faila saturs netiek izdzēsts, ja failam papildus noteikts arī lasīšanas režīms. Faila atvēršana beidzas ar neveiksmi, ja papildus noteikts lasīšanas režīms un fails ar doto nosaukumu neeksistē.
`ios::binary`	Nosaka bināro režīmu lasīšanā vai rakstīšanā (sīkāk aprakstīts zemāk).
`ios::app`	(Tikai kopā ar *ios::out*) Nosaka, ka rakstīšanas režīmā atvērta faila saturs netiek izdzēsts (kā tas notiek noklusētajā variantā), bet gan rakstīšana turpinās, sākot ar faila beigām.

Tab. 16.2.

Noklusētais atvēršanas režīms dažādu tipu failu objektiem

Faila objekta tips	Noklusētais režīms
`fstream`	`ios::in | ios::out`
`ifstream`	`ios::in`
`ofstream`	`ios::out`

Uzstādot faila režīmu, ir obligāta vismaz viena no vērtībām *ios::in* vai *ios::out*, pārējās vērtības domātas precizēšanai.

Piemēri funkcijas *open* izmantošanai:

```
        // gan lasīšanai, gan rakstīšanai:
    file.open ("test.txt", ios::in | ios::out);
        // rakstīšanai, pēc atvēršanas turpinot faila beigās:
    file.open ("test.txt", ios::out | ios::app);
        // lasīšanai:
    file.open ("test.txt", ios::in);
        // lasīšanai un rakstīšanai binārā režīmā:
    file.open ("test.txt", ios::in | ios::out | ios::binary);
```

Pēc darba ar failu tas noteikti jāaizver, ko dara metode *close*.

```
    f.close ();
```

close

```
void close ();
```

Faila objekta funkcija *close()* aizver failu.

Faila objekta deklarēšanu un atvēršanu var apvienot vienā komandā un divu rindu vietā

```
    fstream f;
    f.open ("test.txt");
```

var rakstīt

```
    fstream f ("test.txt");
```

16.2. Teksta faila formatēta apstrāde

Viens no vienkāršākajiem failu veidiem ir teksta fails.

> **Teksta fails** (*text file*) ir fails, kas sastāv no simboliem (burtiem, cipariem, pieturzīmēm utt.).

Teksta fails ir atpazīstams pēc tā, ka tajā atrodamā informācija cilvēkam ir viegli uztverama, apskatot to ar vienkāršu teksta redaktoru (piemēram, *notepad* vai *vi*).

No programmēšanas viedokļa drīzāk runā nevis par teksta failu, bet gan par faila apstrādi teksta režīmā.

Faila apstrāde teksta režīmā parasti notiek **pa simbolam** (kas parasti gandrīz atbilst jēdzienam – pa baitam) vai **pa rindiņai**.

Teksta failu var apstrādāt (lasīt, rakstīt) formatēti (pārveidojot binārus (“neteksta”) datus par teksta un otrādi) vai neformatēti (lasot vai rakstot simbolus tiešā veidā).

No apstrādes viedokļa teksta failu var uztvert kā vienu no šādiem jēdzieniem:

- klaviatūras (ievade) vai displeja (izvade) analogs sekundārajā atmiņā,
- simbolu (baitu) virkne,
- teksta rindiņu virkne.

16.2.1. Formatēta izvade teksta failā

Formatēta izvade failā notiek, izmantojot izvades operatoru << un identiskus formatēšanas mehānismus (funkcijas un manipulatorus), kādi ir pieejami izdrukāšanai uz displeja un kas jau ir aprakstīti nodaļā "C++ pamati" (manipulatori ir pieejami bibliotēkā *<iomanip>*, bet *<iostream>* vietā stājas bibliotēka *<fstream>*)

Att. 16.1. Formatēta izvade teksta failā.

```
#include <fstream>
#include <iomanip>
    ...
fstream fout;
fout.open ("out.txt", ios::out);
fout << setw(5) << 1 << setw(5) << 999 << endl;
    ...
```

out.txt: ('·' tukšums, '¶' jaunas rindiņas simbols, '§' faila beigas)

```
····1··999¶
§
```

16.2.2. Formatēta ievade no teksta faila

16.2.2.1. Ievades no faila standarta shēma

Formatēta ievade no teksta faila notiek pēc līdzīgiem principiem, kādi ir pieejami ievadei no klaviatūras un kas jau ir aprakstīti iepriekšējās nodaļās. Formatēta ievade no faila notiek, izmantojot ievades operatoru *>>*.

Ievadei no faila, salīdzinājumā ar ievadi no klaviatūras, ir šādas atšķirības:

- jaunas rindiņas simbolam (atbilst ENTER) nav specifiskas nozīmes,
- pēc katras ievades darbības tiek pārbaudīts, vai nav sasniegtas faila beigas.

Faila beigu pazīmi uzrāda funkcija *eof*:

eof

```
bool eof () const;
```

Faila objekta funkcija *eof()* atgriež *true*, ja tiek konstatētas faila beigas, citādi atgriež *false*. (*const* norāda, ka par funkciju *eof* tiek garantēts, ka tā neizmaina faila objektu).

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

int main ()
{
    int i;
    fstream fin ("in.txt", ios::in);
    fin >> i;
    while (!fin.eof())
    {
        cout << i << endl;
        fin >> i;
    };
    fin.close ();
    return 0;
}
```

in.txt: (ievade) (‘·’ tukšums, ‘¶’ jaunas rindiņas simbols, ‘§’ faila beigas)

```
31·12·2006¶
§
```

konsole: (izvade)

```
31
12
2006
```

Att. 16.2. Formatēta ievade no teksta faila

Vēl viena tipiska vienošanās par teksta failiem ir tāda, ka parasti tie beidzas ar jaunas rindiņas simbolu (*newline*). Ja tas tā tiek pieņemts, tad vispārīgā gadījumā šādam failam apstrāde ir izveidojama vieglāk. Arī, piemēram, valodas C++ standarts pieprasa, ka C++ programmas faili (un tie ir teksta faili) beigtos ar jaunas rindiņas simbolu. Ja iepriekšējā piemērā ievades fails *in.txt* nebeigtos ar jaunas rindiņas simbolu, bet gan beigtos jau ar ‘6’, tad programma uz ekrāna izdrukātu tikai *31* un *12*, jo jau uzreiz pēc *2006* nolasīšanas tiktu nofiksētas faila beigas, un nākošā iterācija, kurā vajadzētu izdrukāt šo skaitli, vairs nenotiktu (tai pat laikā programmas beigās mainīgais *i* saturētu vērtību *2006*).

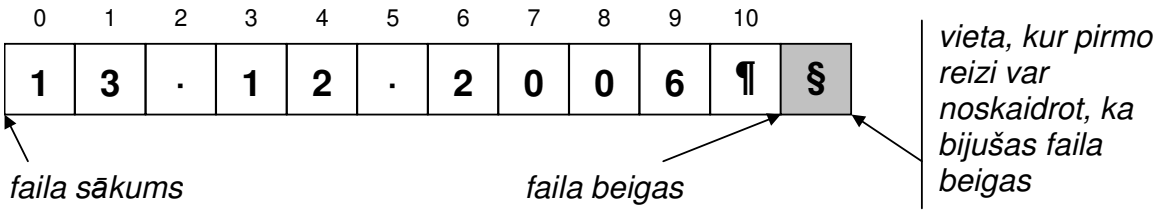
No iepriekšējās programmas izriet standarta shēma, kādā notiek secīga faila nolasīšana C++. Šī shēma raksturojas ar to, ka vienas cikla iterācijas laikā:

- tiek apstrādāta viena vērtība,
- bet nolasīta jau nākamā.

Tas saistīts ar C++ specifiku – C++ tiek uzstādīta faila beigu pazīme nevis tad, kad ir loģiskās faila beigas, bet gan tikai pēc tam, kad ir **mēģināts nolasīt aiz faila beigām**. Ar to C++ atšķiras no dažām citām programmēšanas valodām (piemēram, PASCAL, kur vienā iterācijā var nolasīt un apstrādāt to pašu vērtību).

```
FAILASECĪGANOLASĪŠANA (F)
    nolasa pirmo vērtību no faila
    WHILE (fails F nav beidzies)
        apstrādā nolasīto vērtību
        nolasa nākošo vērtību no faila
```

Att. 16.3. Faila secīgas nolasīšanas standarta shēma C++



Att. 16.4. Fails un tā beigu noteikšana C++

16.2.2.2. Kļūda ievaddatos un faila objekta bloķēšana

Faila apstrādes laikā var iestāties kļūdas situācija (piemēram, ievaddatu neatbilstība mainīgā tipam). Tādā gadījumā **faila objekts tiek bloķēts**, un visas turpmākās darbības ar šo failu tiek ignorētas. Tas notiek tieši tāpat kā ar standarta formatēto ievadu, kas jau tika aprakstīts iepriekšējās nodaļās.

Vispārīgā gadījumā var būt divi galvenie iemesli faila objekta bloķēšanai:

- faila beigas,
- ievades vai izvades kļūda.

Programmai ir jābūt gatavai, ka faila objekts varētu tikt bloķēts. Ideālā gadījumā pēc katras faila operācijas būtu jāpārbauda, vai nav iestājusies kļūda vai faila beigas.

Faila beigas var noskaidrot ar funkciju *eof()*, kas aprakstīta iepriekš, bet ievades vai izvades kļūdu var noskaidrot ar kādu no funkcijām *fail()* vai *bad()*.

fail

```
bool fail () const;
```

Faila objekta funkcija *fail()* atgriež *true*, ja tiek konstatēta ievades vai izvades kļūda, citādi atgriež *false*.

bad

```
bool bad () const;
```

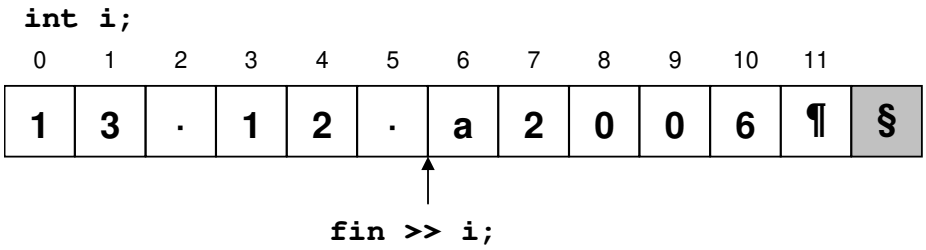
Faila objekta funkcija *bad()* atgriež *true*, ja tiek konstatēta fatāla ievades vai izvades kļūda, citādi atgriež *false*.

Funkciju *eof()*, *fail()*, *bad()* vietā bieži lieto (pēc nozīmes pretēju) funkciju *good()*, kas norāda, ka nav ne faila beigas, ne arī ir iestājusies kļūda.

good

```
bool good () const;
```

Faila objekta funkcija *good()* atgriež *true*, ja nav konstatēta ievades/izvades kļūda, kā arī (ievades gadījumā) nav konstatētas faila beigas.



Att. 16.5. Kļūdainu datu piemērs failā

Nākošajā piemērā, salīdzinot ar Att. 16.2, cikla nosacījuma *(fin.eof())*, t.i. "kamēr nav faila beigas", vietā izmantots *(fin.good())*, jeb "kamēr viss kārtībā". Tas nodrošina, ka, sastopot kļūdu, programma neieciklosies, bet beigs darbu:

```
...
int i;
fstream fin ("in.txt", ios::in);
fin >> i;
while (fin.good()) // vai vienkārši: while (fin)
{
    cout << i << endl;
    fin >> i;
};
...
```

in.txt: (ievade) ('·' tukšums, '¶' jaunas rindiņas simbols, '§' faila beigas)

```
31·12·a2006¶
§
```

konsole: (izvade)

```
31
12
```

Att. 16.6. Kļūdaina faila nolasīšana, apstājoties pie kļūdas (salīdzināt cikla nosacījumu ar piemēru Att. 16.2)

Kā redzams pēc programmas komentāra, izmantotās pārbaudes vietā

```
fin.good()
```

var lietot tā vienkāršoto sinonīmu – pašu faila objektu:

```
fin
```

16.2.2.3. Faila objekta atbloķēšana pēc kļūdas

Lai turpinātu darbu ar (kļūdas vai faila beigu iestāšanās dēļ) bloķētu faila objektu, jāveic šādas darbības (sk. arī objekta *cin* atbloķēšanu pēc kļūdas, kas aprakstīts iepriekš):

- objekta atbloķēšana ar metodi *clear()*,
- pasākumi kļūdas situācijas novēršanai (piemēram, cita faila atvēršana, ja kļūda bijusi pie atvēršanas vai noteikta datu daudzuma izlaišana (metode *ignore*), ja kļūda bijusi ievaddatos).

clear

```
void clear ();
```

Faila objekta funkcija *clear()* atbloķē faila objektu, uzstādot kļūdas un faila beigu bitus uz 0, bet *good* bitu uz 1.

ignore

```
istream &ignore (int count=1, char delim=EOF);
```

Faila objekta funkcija *ignore()* izlaiž noteiktu skaitu (*count*) simbolu ievades plūsmā, bet ne tālāk kā līdz pirmajam sastaptajam simbolam *delim*.

```
...
fstream fin ("in.txt", ios::in);
int i;
fin >> i;
while (fin)
{
    cout << i << endl;
    fin >> i; // mēģina lasīt kārtējo skaitli
    // cikls, kamēr kļūda netiek likvidēta,
    // bet tai pat laikā nav sasniegtas faila beigas
    while (!fin.good() && !fin.eof())
    {
        fin.clear ();          // atbloķē faila objektu
        fin.ignore (1, '\n'); // izlaiž vienu simbolu
        fin >> i;              // mēģina lasīt nākošo skaitli
    }
};
...
```

in.txt: (ievade) (‘·’ tukšums, ‘¶’ jaunas rindiņas simbols, ‘§’ faila beigas)

```
31*12·a2006xx#9999¶
§
```

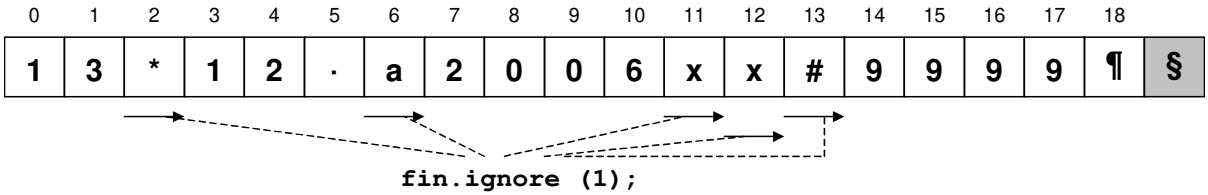
konsole: (izvade)

```
31
12
2006
9999
```

Att. 16.7. Kļūdaina veselu skaitļa faila nolasīšana ar kļūdaino simbolu izlaišanu (*ignore*)

Piezīme. Šī programma nemāk apstrādāt kļūdu, ja tā ir pašā faila sākumā. Šim nolūkam tādu pašu iekšējo ciklu *while (!fin.good() && !fin.eof())* vajadzētu ievietot arī pēc pirmreizējās nolasīšanas operācijas. Ja kļūda ir jau pašā faila sākumā (piemēram, tas sākas ar burtu), tad šī programma uzreiz apstājas.

Iepriekšējās programmas darbu kļūdaino simbolu izlaišanā parāda nākošais attēls.



Att. 16.8. Kļūdainu simbolu izlaišana, nolasot failu (programma Att. 16.7)

16.3. Teksta failu apstrāde bez formatēšanas

16.3.1. Teksta failu apstrāde pa vienam simbolam

Formatēšana nozīmē datu pārveidošanu pēc nolasīšanas vai pirms izdrukas (failā). Vienkāršākā teksta faila apstrāde bez formatēšanas ir **pa vienam simbolam**. Arī nolasīšana pa vienam simbolam notiek pēc faila nolasīšanas standarta shēmas, kas redzama Att. 1.8.

Teksta faila apstrāde pa vienam simbolam notiek, izmantojot funkcijas *get()* un *put()*, kas attiecīgi nolasa vai ieraksta vienu simbolu failā.

```
    fstream fin ("in.txt", ios::in);
    fstream fout ("out.txt", ios::out);
    char c;
    fin.get (c);
    while (fin)
    {
        fout.put (c);
        fin.get (c);
    };
    fin.close ();
    fout.close ();
```

in.txt: (ievade) (‘·’ tukšums, ‘¶’ jaunas rindiņas simbols, ‘§’ faila beigas)

```
    This¶
    is·an¶
    example.§
```

out.txt: (izvade) (‘·’ tukšums, ‘¶’ jaunas rindiņas simbols, ‘§’ faila beigas)

```
    This¶
    is·an¶
    example.§
```

Att. 16.9. Viena faila (*fin*) pārrakstīšana otrā (*fout*) pa vienam simbolam

get

```
istream &get (char &c);
```

Faila objekta metode *get()* ar vienu parametru nolasa vienu simbolu no ievades (faila) plūsmas un ievieto mainīgajā *c*. Ja fails atvērts binārā režīmā, tad tiek nolasīts nevis viens simbols, bet viens baits.

```
int get ();
```

Faila objekta funkcija *get()* bez parametriem darbojas tāpat, bet nolasīto simbolu atgriež kā atgriežamo vērtību.

put

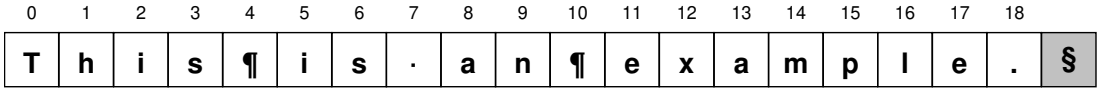
```
ostream &put (char c);
```

Faila objekta funkcija *put()* ieraksta padoto simbolu *c* izvades (faila) plūsmā. Ja fails atvērts binārā režīmā, tad mainīgajā *c* saglabātais baits tiek precīzi pārrakstīts failā.

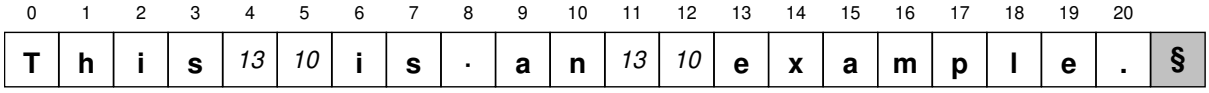
Metodes *put* un *get* ir vienas no retajām faila objektā, kuras spēj strādāt divējādi:

- teksta režīmā,
- binārā režīmā.

Apstrādājot failu binārā režīmā, informācija tiek apstrādāta precīzi pa baitam, bet teksta režīmā – pa simbolam. Teksta failā šie jēdzieni parasti sakrīt, tomēr ir daži izņēmumi. Labākais piemērs tam ir fakts, ka *Windows* sistēmā jaunas rindiņas simbolu ¶ kodē nevis ar vienu baitu (jaunas rindiņas simbola kods: attiecīgi 13 vai 10), kā tas ir *Unix/Linux* un *Apple*, bet gan ar divu baitu secību: <13,10>. Tādējādi *Windows* sistēmā kodēts teksta fails ir garāks uz jaunas rindiņas simbolu rēķina.



Att. 16.10. Iepriekšējās programmas ievades faila *in.txt* interpretācija teksta režīmā (Att. 16.9)



Att. 16.11. Iepriekšējās programmas ievades faila *in.txt* interpretācija binārā režīmā (*Windows* sistēmā kodētam failam) (Att. 16.9)

Teksta režīms dod iespēju rakstīt universālas vairāku platformu programmas, platformu atšķirību atrisināšanu uzticot pašam C++.

Tādu duālas darbības funkciju kā *get* un *put* darbības veidu nosaka režīms, kādā atvērts fails. **Pēc noklusēšanas faila atvēršanas režīms ir teksta**. Ja, atverot failu, ir uzrādīta bināra režīma vērtība *ios::binary*, tad apstrādes režīms ir binārs:

```
fstream fin ("in.txt", ios::in | ios::binary);
fstream fout ("out.txt", ios::out | ios::binary);
```

Teksta režīma priekšrocība ir tāda, ka C++ pats uzņemas interpretēt teksta failu atkarībā no dotās platformas kodējuma, tādējādi viens un tas pats C++ programmas kods ir derīgs dažādām platformām. Binārs režīms ļauj paskatīties uz failu precīzi pa baitam – kāds tas precīzi ir, neveicot nekādu papildus apstrādi (interpretēšanu).

Līdzīgā veidā teksta un binārais režīms sastopams failu pārsūtīšanā, izmantojot *ftp* protokolu.

Faila objektā ir pieejamas arī tādas metodes, kuras spēj strādāt tikai binārā režīmā, un faila atvēršanas režīma uzstādīšana (teksta vai bināra) to darbību no šāda viedokļa neietekmē. Šādas metodes tiks apskatītas nākošajās nodaļās, runājot par bināru failu apstrādi.

16.3.2. Teksta failu nolasīšana pa rindai

Bieži lietota teksta failu nolasīšana ir **pa rindai**. Dalīšana rindās ir teksta strukturēšanas veids, kas atbilst tam, kā to dara cilvēki, pierakstot dabisko valodu tekstus. Tas varētu būt izskaidrojums šāda veida teksta failu apstrādes biežajam lietojumam. Nolasīšana pa rindai notiek pēc faila nolasīšanas standarta shēmas.

Faila nolasīšana pa vienai rindai notiek, izmantojot funkciju *getline()*, kam ir 2 modifikācijas – zema un augsta līmeņa simbolu virknēm. Abas modifikācijas jau apskatītas pie standarta ievades iepriekšējās nodaļās.

getline (*ar zema līmeņa simbolu virkni buferim*)
```
istream &getline (char *buf, int num, char delim='\n');
```

Faila objekta funkcija *getline()* lasa simbolus masīvā *buf*, kamēr nav nolasīti *num-1* simboli (viena vieta automātiski tiek rezervēta simbolu virknes beigu simbolam) vai kamēr netiek sastapts atdalītājsimbols (pēc noklusēšanas – jaunas rindiņas simbols) vai faila beigas. Nolasot *num-1* simbolus pirms sastapts atdalītājsimbols, iestājas ievades kļūda, un faila objekts tiek nobloķēts. Nolasītais atdalītājsimbols netiek ievietots masīvā. Nolasītās virknes galā automātiski tiek ievietots simbolu virknes beigu simbols.

getline (*ar augsta līmeņa simbolu virkni buferim*)
```
istream &getline (istream &file, string &buf, char delim='\n');
```

Funkcija *getline()* lasa simbolus augsta līmeņa simbolu virknē *buf*, kamēr netiek sastapts atdalītājsimbols (pēc noklusēšanas – jaunas rindiņas simbols) vai faila beigas. Funkcija *getline()* augsta līmeņa simbolu virknēm nav faila objekta funkcija, tāpēc faila (vai cits ievades) objekts tai tiek padots kā parametrs.

```
#include <fstream>
#include <iostream>
    ...
string s;
fstream fin ("in.txt", ios::in);
getline (fin, s);
while (fin)
{
    cout << s << endl;
    getline (fin, s);
};
fin.close ();
    ...
```

in.txt: (ievade) (‘·’ tukšums, ‘¶’ jaunas rindiņas simbols, ‘§’ faila beigas)
```
This¶
is·an¶
example.§
```

konsole: (izvade)
```
This
is an
example.
```

Att. 16.12. Teksta faila nolasīšana pa rindām un izvade uz ekrāna (par buferi izmantota augsta līmeņa simbolu virkne)

17. Bināru failu apstrāde

17.1. Vienkāršākās faila binārās apstrādes operācijas

17.1.1. Binārās failu apstrādes atšķirības

Ar **bināru failu** tiek saprasts fails, kurš nav teksta fails, respektīvi, tāds, kurš satur arī neteksta datus. Tāds fails būtu jāapstrādā bināri, un no programmētāja viedokļa svarīgāks jēdziens par bināru failu ir faila **bināra apstrāde**.

> **Faila bināra apstrāde** ir darbības ar failu baitu līmenī, neinteresējoties par datu interpretāciju no datu apstrādes viedokļa programmā.

Tādējādi no faila binārās apstrādes viedokļa fails ir "neko neizsakošu" baitu virkne.

Valodā C++ baitu apzīmē datu tips

```
char
```

bet baitu virkni:

```
char*
```

kas arī ir svarīgākie datu tipi failu (un ne tikai failu) binārajā apstrādē.

Bināru failu izmantošanas lielākais mīnuss ir tas, ka šādu failu atverot teksta redaktorā, parasti nekas nav saprotams – faila saturu interpretē ar šo failu saistītā programma.

Tomēr bināru failu izmantošanas priekšrocības ir pietiekoši svarīgas, lai nopietnās programmās datu saglabāšanai bieži izmantotu tieši tos:

- binārā formā glabāti dati aizņem mazāk vietas (sekundārās atmiņas resursu ietaupījums),
- lai rakstītu/lasītu uz/no bināra faila, parasti nav nepieciešama nekāda datu pārveidošana (atšķirībā no formatēta izvada/ievada) (procesora resursu ietaupījums),
- apstrādājot failu binārā režīmā, ir pieejamas vairākas efektīvas failu apstrādes metodes (piemēram, meklēšana pēc pozīcijas), kas teksta režīmā nav izmantojamas.

Par bināriem failiem daļēji tika runāts jau nodaļā par teksta failiem – faila objekta funkcijas *get()* un *put()* spēja darboties arī binārajā režīmā. Lai piespiestu šīs funkcijas darboties binārā režīmā, bija nepieciešams noteikt faila atvēršanas papildus režīmu *ios::binary*.

Šajā nodaļā tiks apskatītas failu apstrādes metodes, kas darbojas tikai binārā režīmā, līdz ar to binārā režīma uzstādīšana pie faila atvēršanas nav obligāti nepieciešama.

17.1.2. Darba sākums ar bināru failu

Faila atvēršana, aizvēršana un visi galvenie principi darbam ar bināru failu ir tādi paši kā darbam ar teksta failu, kas aprakstīts nodaļā par teksta failu apstrādi. Vienīgi bināra faila gadījumā parasti daudz precīzāk jāapzinās apstrādājamā faila struktūra, jo datu interpretācija būs jādefinē pašam programmētājam, nevis jāatstāj standarta faila apstrādes metožu ziņā.

17.1.3. Izvade failā binārā režīmā

Izvadi binārā režīmā nodrošina faila objekta funkcija *write*.

write

```
ostream &write (const char* buf, int num);
```

Faila objekta funkcija *write()* ieraksta *num* baitus izvades plūsmā (failā) no bufera (atmiņas apgabala), kura sākuma adrese glabājas mainīgajā *buf*.

Ir svarīgi, ka failā ierakstāmie dati tiek padoti kā baitu virkne (*char**) neatkarīgi no tā, kā norādītie dati tiek interpretēti programmā. Šim nolūkam parasti jāveic datu tipa pārveidošana uz *char**.

Ja mums ir programmas mainīgais, kurš reprezentē noteiktu informāciju atmiņā, tad funkcijai *write* padodamā bufera (parametrs nr. 1) sagatavošana parasti notiek šādos 2 soļos:

- adreses paņemšana no mainīgā (ja mainīgais jau nav norāde),
- norādes pārveidošana uz tipu *char** (ja tai jau nav tāds tips), izmantojot klasisko tipu pārveidošanas operatoru (*((char*)pointer*) vai (*reinterpret_cast<char*>(pointer)*).

Otrs funkcijas *write()* arguments ir baitu skaits, cik jāieraksta failā. Šeit ērti izmantot funkciju *sizeof()*, kam kā argumentu var ņemt gan datu tipu, gan mainīgo, kam vēlas noskaidrot lielumu. Tomēr, lai nodrošinātu informācijas apmaiņu starp dažādām platformām, vai starp programmām, kas kompilētas ar dažādiem kompilatoriem, dažreiz drošāk ir caur parametru #2 padot konstantu vērtību.

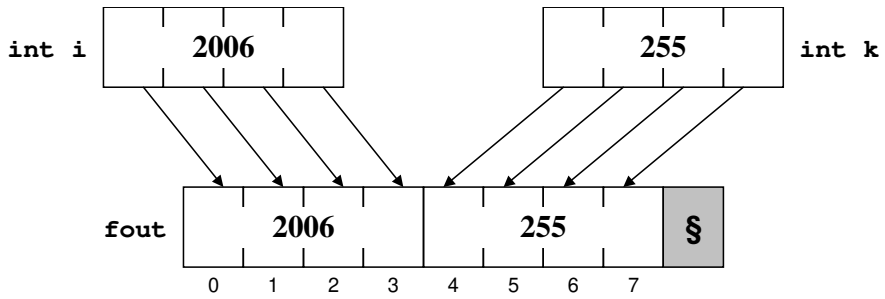
Funkcija *write()*, kam caur parametru #2 tiek padots 1, lielā mērā atbilst binārā režīmā izsauktai funkcijai *put()*.

```
    fstream fout ("integers.bin", ios::out);
    int i=2006, k=255;
        // ierakstu vienu skaitli:
    fout.write ((char*)&i, sizeof(int));
        // ierakstu otru skaitli:
    fout.write (reinterpret_cast<char*>(&k), sizeof(int));
    fout.close ();
```

integers.bin: (izvade) (*hex* redaktorā)

```
00000000: D6 07 00 00 FF 00 00 00|                    | Ö■■■■■■■
```

Att. 17.1. Binära faila izvade



Att. 17.2. Iepriekšējās programmas (Att. 16.7) darbība pēc būtības

Ja ar metodi *write* failā raksta no mainīgā ar tipu *char*, tad tipa pārveidošana nav nepieciešama, jo *&c* jau ir ar tipu *char**:

```
    char c;
        ...
    fout.write (&c, 1); // identisks fout.put(c) binārā režīmā
```

Ja ar metodi *write* failā raksta no *char* masīva (piemēram, simbolu virknes), tad arī adreses ņemšana (&) nav nepieciešama, jo *s* jau ir ar tipu *char**:

```
char s[20];
    ...
fout.write (s, 20);
```

17.1.4. Nolasīšana no faila binārā režīmā

Nolasīšanu binārā režīmā nodrošina faila objekta funkcija *read*.

read

```
istream &read (char* buf, int num);
```

Faila objekta funkcija *read()* nolasa *num* baitus no ievades plūsmas (faila) un ieraksta buferī (atmiņas apgabalā), kura sākuma adrese glabājas mainīgajā *buf*.

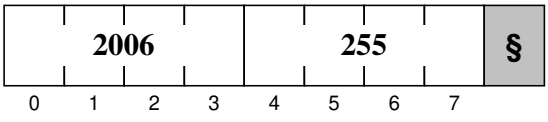
Abu funkcijai padodamo argumentu noformēšana notiek identiski funkcijai *write()*.

Funkcija *read()*, kam caur parametru #2 tiek padots 1, lielā mērā atbilst binārā režīmā izsauktai funkcijai *get()*.

Nākošā programma demonstrē bināra *int* vērtību faila (piemēram, tāda, kas tika izveidots ar programmu Att. 16.7) nolasīšanu un vērtību izdrukāšanu uz ekrāna.

```
int i;
fstream fin ("integers.bin", ios::in);
fin.read ((char*)&i, sizeof(int));
while (fin)
{
    cout << i << endl;
    fin.read ((char*)&i, sizeof(int));
};
fin.close ();
```

integers.bin: (ievade) (pēc būtības)



konsole: (izvade)

```
2006
255
```

Att. 17.3. Bināra nolasīšana no faila

Kaut arī funkcija *read()* darbojas līdzīgi funkcijai *write()*, tikai pretējā virzienā, tai ir spēkā viena papildus nianse – failā varētu būt atlicis mazāk simbolu nekā tiek padots caur parametru #2. Šajā gadījumā tiek nolasīti tikai tik baiti, cik failā ir, bet nolasīto baitu skaitu var noskaidrot, izmantojot funkciju *gcount()*.

gcount

```
int gcount () const;
```

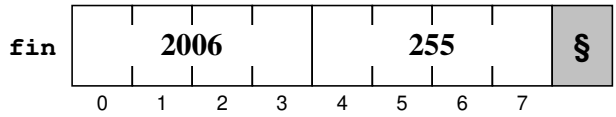
Faila ievades objekta funkcija *gcount()* atgriež simbolu skaitu, kas nolasīti pēdējā ievades operācijā.

Nākošā programma pārraksta vienu failu otrā binārā režīmā. Līdzīga programma tika demonstrēta nodaļā par teksta failiem, taču šeit pārrakstīšana notiek nevis pa vienam baitam,

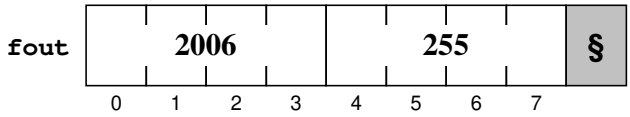
bet pa vairākiem, tā nodrošinot ātrāku apstrādi. Informācijas noglabāšanai starp nolasīšanu un ierakstīšanu tiek lietots buferis (šeit garumā 3). Metode *gcount* tiek izmantota, lai zinātu, cik kārtējā reizē nolasīti baiti, lai zinātu, cik savukārt no bufera jāieraksta otrā failā. Problēmas ar to, ka otrā failā nesanāks ierakstīt visu buferi, var rasties tikai pēdējā nolasīšanas porcijā no faila. Šī programma der viena faila pārrakstīšanai otrā neatkarīgi no tā, cik liels bufera izmērs (*BUFFER_SIZE*) ir uzlikts. Nolasīto baitu skaits katrā solī tiek parādīts izdrukā uz ekrāna.

```
const unsigned int BUFFER_SIZE = 3;
char buffer[BUFFER_SIZE];
fstream fin ("integers.bin", ios::in);
fstream fout ("integers2.bin", ios::out);
while (fin)
{
    fin.read (buffer, BUFFER_SIZE);
    cout << fin.gcount() << endl;
    fout.write (buffer, fin.gcount());
};
fin.close ();
fout.close ();
```

integers.bin: (ievade) (pēc būtības)



integers2.bin: (izvade) (pēc būtības)



konsole: (izvade)

```
3
3
2
```

Att. 17.4. Viena faila (*fin*) pārrakstīšana otrā (*fout*) binārā režīmā, izmantojot buferi

17.2. Tiešās pieejas metodes bināro failu apstrādē

Noklusētā metode faila apstrādei ir sekvenciālā pēc kārtas, taču pateicoties sekundārās atmiņas tehniskajām iespējām, ir iespēja apstrādāt faila datus, piekļūstot tiem tiešā veidā – kā caur indeksu masīva elementam.

Strādājot ar failu tiešās pieejas režīmā, jāatceras, ka

- faila informācijas pamatvienība ir **baits**,
- baitu numerācija failā sākas ar 0.

Strādājot ar failu tiešās pieejas režīmā, īpaši svarīgs kļūst jēdziens **faila norāde** – tā ir aktuālā vieta failā, sākot no kuras uz priekšu tiks veikta kārtēja nolasīšana, ierakstīšana vai cita darbība. Tiešās pieejas darbības ir tās, kuras darbojas ar faila norādi.

Šajā sadaļā tiks aprakstītas šādas failu apstrādes metodes:

- *seek*, *seekp* – faila norādes uzstādīšana noteiktā pozīcijā,
- *tellg*, *tellp* – atrašanās vietas noskaidrošana failā,
- *peek* – "paskatīšanās vienu simbolu uz priekšu".

17.2.1. Faila norādes uzstādīšana (pozīcijas meklēšana failā)

Faila norādes uzstādīšanu noteiktā pozīcijā veic funkcijas *seek/seekp*.

seekg un **seekp**

```
istream &seekg (int offset, ios::seek_dir origin = ios::beg);
ostream &seekp (int offset, ios::seek_dir origin = ios::beg);
```

seekg() un *seekp()* ir funkcijas, kas uzstāda faila norādi noteiktā pozīcijā – baitu skaitu *offset* relatīvi pret noteiktu izejas pozīciju *origin*. Ja otrais arguments tiek izlaists, faila norāde tiek uzstādīta relatīvi pret faila sākumu. *seekg()* ir pieejama failiem, kas atvērti lasīšanas režīmā, bet *seekp()* – failiem, kas atvērti rakstīšanas režīmā. Ja fails ir atvērts abos režīmos, ir pieejamas abas funkcijas, un tās dara vienu un to pašu.

Parametrs *offset* var būt arī negatīvs.

Parametrs *origin* var saturēt vienu no šādām vērtībām, norādot izejas pozīciju, relatīvi pret kuru tiks veikta nobīde (faila norādes uzstādīšana):

- *ios::beg* – faila sākums,
- *ios::cur* – faila norādes pašreizējā pozīcija,
- *ios::end* – faila beigas (pirms beigu norādes).

```
    fstream f ("seek.txt", ios::in | ios::out | ios::binary);
    char c;
    f.seekg (-1, ios::end); // aiziet uz pozīciju pirms pēdejā
    f.get (c); // nolasa pēdējo
    f.seekp (0, ios::beg); // aiziet uz faila sākumu
    f.put (c); // ieraksta sākumā vienu simbolu
    f.close ();
```

seek.txt: (ievade) (‘§’ – faila beigas)

```
    ABCDEFGHIJ§
```

seek.txt: (izvade) (‘§’ – faila beigas)

```
    JBCDEFGHIJ§
```

Att. 17.5. Metožu *seekg/seekp* demonstrācija. Faila beidzamo simbolu ieraksta pirmajā pozīcijā

17.2.2. Faila norādes atrašanās vietas noskaidrošana

Faila norādes atrašanās vietas noskaidro ar funkcijām *tellg/tellp*.

tellg un tellp

```
int tellg ();
int tellp ();
```

tellg() un *tellp()* ir funkcijas, kas nosaka faila norādes atrašanās pozīciju (baitu skaitu no faila sākuma) (faila norāde – vieta failā, kurā tiks izpildīta nākamā ievades vai izvades operācija). *tellg()* ir pieejama failiem, kas atvērti lasīšanas režīmā, bet *tellp()* – failiem, kas atvērti rakstīšanas režīmā. Ja fails ir atvērts abos režīmos, ir pieejamas abas funkcijas.

```
    fstream f ("tell.txt", ios::in);
    f.seekg (0, ios::end); // aiziet uz beigām
    cout << f.tellg() << endl; // izdrukā pozīciju (=faila garums)
    f.seekg (-3, ios::cur); // paiet trīs soļus atpakaļ
    cout << f.tellg() << endl; // izdrukā pozīciju
    f.close ();
```

tell.txt: (ievade) ('§' – faila beigas)

```
    ABCDEFGHIJ§
```

konsole: (izvade)

```
    10
    7
```

Att. 17.6. Metodes *tellg* demonstrācija (ietver metodi, kā noteikt faila garumu)

17.2.3. Paskatīšanās vienu simbolu uz priekšu

Paskatīšanos vienu soli uz priekšu, nepārbīdot faila norādi, nodrošina funkcija *peek*.

peek

```
int peek ();
```

Funkcija *peek()* atgriež kārtējo simbolu failā, bet, atšķirībā no *get()*, nepārbīda faila norādi ("paskatās vienu simbolu uz priekšu").

peek() ir pieejama failiem, kas atvērti lasīšanas režīmā.

```
    char c;
    fstream f ("peek.txt", ios::in | ios::binary);
    f.seekg (5); // aiziet uz pozīciju 5 (pirms 'F')
    c = f.peek(); // nolasa, bet nepārbīda norādi
    cout << c << endl;
    c = f.get(); // nolasa un pārbīda norādi
    cout << c << endl;
    c = f.get(); // vēlreiz nolasa un pārbīda norādi
    cout << c << endl;
    f.close ();
```

peek.txt: (ievade) ('§' – faila beigas)

```
    ABCDEFGHIJ§
```

konsole: (izvade)

```
    F
    F
    G
```

Att. 17.7. Metodes *peek* demonstrācija

17.3. Datu struktūru glabāšana binārā failā

Sarežģītāku datu struktūru objekti teorētiski arī var tikt ierakstīti/nolasīti no faila, padodot funkcijai *write()*/*read()* visu objektu uzreiz, tomēr vispārīgā gadījumā šādos gadījumos ierakstīšana/nolasīšana notiek pa vienam laukam, jo

- ne visi faila lauki vienmēr satur datus – lauks var būt arī norāde,
- ne vienmēr failā būtu jāieraksta visi dati, kas glabājas objektā,

- dažreiz, pārrakstot datus failā, ir nepieciešama optimizācija no datu glabāšanai nepieciešamās vietas viedokļa.

> **Ieraksts** (*record*) (jeb **komponente**) ir (no datu apstrādes viedokļa) vienots informācijas kopums failā.

Nākošajā piemērā failā tiks glabāti dati par personu. Datu glabāšana notiek 2 failos – pirmajā, izmantojot fiksēta garuma ierakstus, tādējādi, neizmantojot glabāšanai izmantojamās vietas optimizāciju, otrajā, izmantojot mainīga garuma ierakstus un tādējādi samazinot kopējo faila izmēru.

Šajā gadījumā faila ieraksts jeb komponente ir informācija par vienu personu.

Pirmajā failā (*pers1.bin*) viena ieraksta struktūra ir šāda:

- *name* – personas vārds – 20B,
- *age* – personas vecums – 4 B.

Otrajā failā (*pers2.bin*) viena ieraksta struktūra ir šāda:

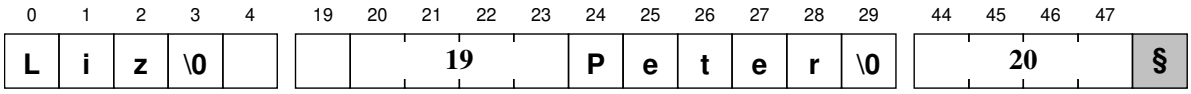
- *name_length* – personas vārda garums – 1B,
- *name* – personas vārds – *name_length* B,
- *age* – personas vecums – 1B.

Glabāšanas vietas ietaupījums otrajā failā ir galvenokārt uz personas vārda glabāšanas rēķina, ieviešot papildus lauku – personas vārda garums – un neglabājot lieku informāciju, bez tam personas vecums tiek glabāts vienā baitā, ar ko pilnīgi pietiek.

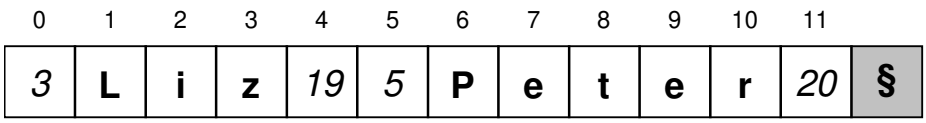
Kompaktās glabāšanas mīnuss ir faila tiešās pieejas apstrādes izslēgšana no apstrādes iespējām – nonākt pie ieraksta ar noteiktu indeksu var tikai ar secīgu faila pārstaigāšanu, jo ieraksti ir mainīga garuma.

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

class person
{
    char name[buffer_size];
    int age;
public:
    person (const char *n, int a)
    {
        strcpy (name, n);
        age = a;
    };
    void write1 (ostream &fout)
    {
        fout.write (name, buffer_size); // pilna garuma masīvs
        fout.write ((char*)&age, sizeof(age)); // visa int vērtība
    };
    void write2 (ostream &fout)
    {
        int slen = strlen(name);
        fout.write ((char*)&slen, 1); // vārda garums vienā baitā
        fout.write (name, slen);      // vārds
        fout.write ((char*)&age, 1);  // vecums vienā baitā
    }
};

int main ()
{
    person p ("Liz", 19);
    person q ("Peter", 20);
    ofstream fout1 ("pers1.bin");
    p.write1 (fout1);
    q.write1 (fout1);
    fout1.close ();
    ofstream fout2 ("pers2.bin");
    p.write2 (fout2);
    q.write2 (fout2);
    fout2.close ();
    return 0;
}
```

pers1.bin: (izvade) (pēc būtības)



pers2.bin: (izvade) (pēc būtības)



pers1.bin: (izvade) (hex redaktorā)

```
00000000: 4C 69 7A 00 8D 45 D8 50|2C 00 00 00 02 00 00 00 | Liz■■EŲP,■■■■■■
00000010: 60 FF 22 00 13 00 00 00|50 65 74 65 72 00 03 78 | `■"■■■■■Peter■■
00000020: FF FF FF FF 58 FF 22 00|76 88 00 78 14 00 00 00 | ■■■■X■"■v■■x■■■
```

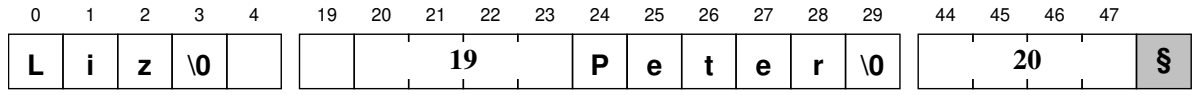
pers2.bin: (izvade) (hex redaktorā)

```
00000000: 03 4C 69 7A 13 05 50 65|74 65 72 14             | ■Liz■■Peter■
```

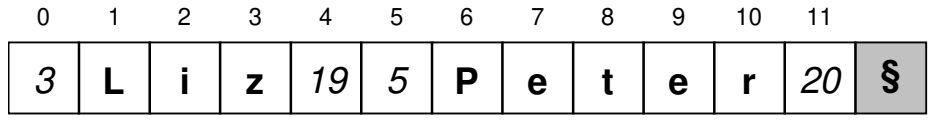
Att. 17.8. Personas informācijas izvade binārā failā

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

class person
{
    char name[buffer_size];
    int age;
public:
    bool read1 (istream &fin)
    {
        fin.read (name, buffer_size); // nolasa vārdu
        fin.read ((char*)&age, sizeof(age)); // nolasa vecumu
        return fin.good ();
    };
    bool read2 (istream &fin)
    {
        int slen = 0; // svarīgi aizpildīt visus baitus ar 0
        age = 0; // svarīgi aizpildīt visus baitus ar 0
        fin.read ((char*)&slen, 1); // nolasa vārda gar. no 1 B
        fin.read (name, slen); // nolasa vārdu uzrādītajā garumā
        name[slen] = '\0'; // pieliek beigās beigu simbolu
        fin.read ((char*)&age, 1); // nolasa vecumu no 1 baita
        return fin.good ();
    };
    void print ()
    {
        cout << name << " " << age << endl;
    }
};

int main ()
{
    person p;
    ifstream fin1 ("pers1.bin");
    while (p.read1 (fin1)) p.print ();
    fin1.close ();
    ifstream fin2 ("pers2.bin");
    while (p.read2 (fin2)) p.print ();
    fin2.close ();
    system ("pause");
    return 0;
}
```

pers1.bin: (ievade) (pēc būtības)



pers2.bin: (ievade) (pēc būtības)



konsole: (izvade)

```
Liz 19
Peter 20
Liz 19
Peter 20
```

Att. 17.9. Personas informācijas izvade binārā failā

18. Datu struktūras

Datu struktūru veidošanai valodā C++ pieejamas divas gandrīz identiskas konstrukcijas *struct* un *class*, par kurām jau ir stāstīts nodaļā par objektorientēto programmēšanu. Konstrukcija *struct* ir mantojums no valodas C, kur tā bija vairāku mainīgo apvienojums (neietverot metodes). Vēl viena specifiska konstrukcija, kas nāk no valodas C, ir *union*, kas tiks aprakstīta tālāk šajā nodaļā.

18.1. Masīvs no struktūrām

Viens no veidiem, kā apvienot struktūru objektus, ir masīvi. Ņemot vērā to, ka struktūru objekti C++ programmā var parādīties 2 veidos, apkopošana masīvos iespējama divos veidos:

- masīvs no struktūras objektiem,
- masīvs no norādēm uz struktūras objektiem.

Nākošais piemērs parāda masīva *parr* izveidošanu no struktūras objektiem tiešā veidā.

```
    ...
const int arr_size = 3;
struct person
{
    char name[20];
    int age;
};
    ...
person parr[arr_size]; // masīva deklarēšana
for (int i=0; i<arr_size; i++)
{
    cin >> parr[i].name;
    cin >> parr[i].age;
};
for (int i=0; i<arr_size; i++)
{
    cout << parr[i].name << " " << parr[i].age << endl;
};
```

```
Liz
19
Ann
22
Peter
20
Liz 19
Ann 22
Peter 20
```

Att. 18.1. Masīvs no struktūrām

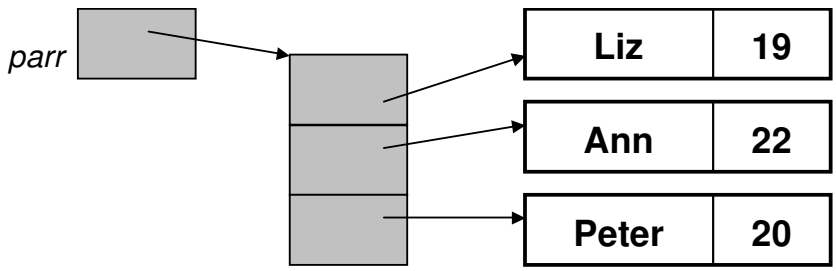
Att. 18.2. Programmā Att. 18.1 izveidotais masīvs no struktūrām

Nākošais piemērs demonstrē funkcionāli identisku programmu, kura izmanto dinamisku masīvu no norādēm uz struktūras objektiem.

```
    ...
const int arr_size = 3;
struct person
{
    char name[20];
    int age;
};
    ...
person **parr;
parr = new person*[arr_size]; // norāžu masīva izveide
for (int i=0; i<arr_size; i++)
{
    parr[i] = new person;
    cin >> parr[i]->name;
    cin >> parr[i]->age;
};
for (int i=0; i<arr_size; i++)
{
    cout << parr[i]->name << " " << parr[i]->age << endl;
    delete parr[i]; // masīva likvidēšana
};
```

```
Liz
19
Ann
22
Peter
20
Liz 19
Ann 22
Peter 20
```

Att. 18.3. Dinamisks masīvs no norādēm uz struktūrām



Att. 18.4. Programmā Att. 18.3 izveidotais masīvs no struktūrām

18.2. Saistītais saraksts – dinamiskas datu struktūras

18.2.1. Dinamiskas datu struktūras

> **Dinamiska datu struktūra** (*dynamic data structure*) ir datu struktūra, kas programmas darbības laikā ļauj elastīgi mainīt tās izmēru.

Dinamisks masīvs ir viena no ērtākajām zema līmeņa dinamiskajām konstrukcijām, tomēr viens no lielākajiem tā mīnusiem ir relatīvi lielais nepieciešamais atmiņas pārrakstīšanas apjoms masīva izmēra izmaiņas gadījumā (masīvu nevar pagarināt, bet jāveido jauns ar vajadzīgo izmēru un jāpārkopē visi dati uz jauno vietu).

"Īsti" dinamiskas datu struktūras strādā pēc cita – ķēdēšanas principa, nodrošinot to, ka katra jauna elementa pievienošana vai izmešana neprasa veikt daudz izmaiņu atmiņā.

Dinamiska datu struktūra sastāv no elementiem, kur katrs elements vispārīgā gadījumā nodrošina divas funkcijas un tādējādi sastāv no divām daļām:

- dati,
- saites uz citiem elementiem.

18.2.2. Saistītais saraksts

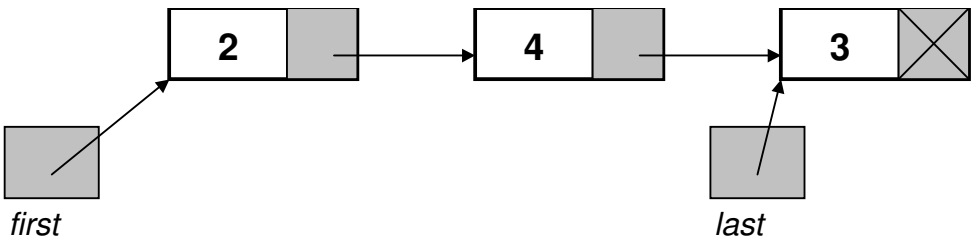
Vienkāršākajā gadījumā saišu daļa sastāv no vienas saites – norādes uz nākošo elementu, tādējādi veidojot lineāru struktūru – **saistīto sarakstu** (*linked list*). Katrs elements saistītajā sarakstā vai nu norāda uz nākošo elementu vai satur norādi ar tukšo adresi *NULL*.

Bez elementiem, kas veido struktūras pamatmasu, saistītais saraksts ietver šādas komponentes:

- norāde uz pirmo elementu,
- norāde uz pēdējo elementu (neobligāta, paredzēta, lai tehniski atvieglotu saraksta papildināšanu galā.

Saraksts var būt tukšs (bez elementiem), to norāda tukša norāde uz pirmo elementu (sākot darbu ar sarakstu, šai norādei obligāti jābūt inicializētai).

Standarta variantā katrs jaunais elements tiek likts **saistītā saraksta galā**, līdz ar to saraksta elementu secība atbilst tā aizpildīšanas secībai.



Att. 18.5. Saistītais saraksts no 3 elementiem (atbilst programmai Att. 18.6)

Saistīto sarakstu (*linked list*), ja nav norādīts savādāk, veido tā, ka katru jauno elementu kabina klāt beidzamajam, tādējādi elementi tajā glabājas ievadīšanas secībā.

```
    ...
struct elem
{
    int num;
    elem *next;
};
    ...
// first-norāde uz pirmo elementu
// last-norāde uz pēdējo elementu
// p-palīgnorāde
elem *first=NULL, *last=NULL, *p;
int i;
cin >> i;
// ievadīšana, kamēr nav sastapta 0
while (i != 0)
{
    p = new elem; // izveido objektu
    p->num = i;   // aizpilda ar nolasīto vērtību
    p->next = NULL; // norāde uz nākošo - tukša!
    if (first == NULL)
    {
        // ja saraksts tukšs
        // gan pirmais, gan pēdējais norāda uz jauno elementu
        first = last = p;
    }
    else
    {
        // ja saraksts nav tukšs
        // pieliek galā beidzamajam
        last->next = p;
        // un jaunais kļūst par beidzamo
        last = last->next;
    };
    cin >> i;
};
// saraksta izdruka
for (p = first; p!=NULL; p=p->next)
{
    cout << p->num << endl;
};
// saraksta iznīcināšana
p = first;
while (p!=NULL)
{
    first = first->next;
    delete p;
    p = first;
};
```

```
2
4
3
0
2
4
3
```

Att. 18.6. Saistītā saraksta (*linked list*) izveidošana un izdrukāšana

Atšķirībā no masīva, kur katram elementam var piekļūt uzreiz, uzrādot indeksu, pie saistīta saraksta elementiem var piekļūt tikai, secīgi pārstaigājot sarakstu, sākot ar pirmo elementu.

Tipiska darbība ar daudzām datu struktūrām ir to pārstaigāšana pa vienam elementam (*traversal*), veicot noteiktu darbību ar katru elementu.

```
STRUKTŪRASSECĪGAPĀRSTAIGĀŠANA (S)
    aiziet uz struktūras sākumu
    WHILE (struktūra S nav beigusies)
        apstrādā kārtējo elementu
        pāriet uz nākošo elementu
```

Att. 18.7. Datu struktūru (t.sk. saistīta saraksta) secīgas pārstaigāšanas shēma

Šī shēma ir analoģiska tai, kas nosaka, kādā veidā tiek organizēta datu nolasīšana no faila.

Strādājot ar dinamiskām datu struktūrām, ir ļoti svarīgi kontrolēt nulles norādes (*NULL* jeb 0):

- ja elements ir pēdējais virknē, tad nedrīkst aizmirst uzstādīt *NULL* tā norādei uz nākošo elementu,
- ja kāda norāde ir *NULL*, nedrīkst mēģināt pēc šis norādes nokļūt pie kāda objekta un to apstrādāt (tipiska kļūda darbā ar dinamiskajām datu struktūrām).

Parasti datu struktūru apstrādes darbības tiek ievietotas funkcijās. Tādā gadījumā funkcijai padod norādi uz pirmo un dažreiz arī pēdējo saraksta elementu (piemēram, *elem**):

```
void print_list (elem *first)
{
    ...
};
```

Ja ir sagaidāms, ka norāde funkcijā varētu izmainīties, tad parametrs jādefinē kā reference. Piemēram, elementa pievienošanas funkcijā norāde uz pirmo elementu mainās, ja pievienošana notiek tukšam sarakstam, bet norāde uz beidzamo elementu mainās katru reizi, tāpēc abiem attiecīgajiem parametriem jābūt referencēm (*elem*&*):

```
void add_element (elem *&first, elem *&last, int i)
{
    ...
};
```

18.2.3. Elementa iespraušana saistītā sarakstā

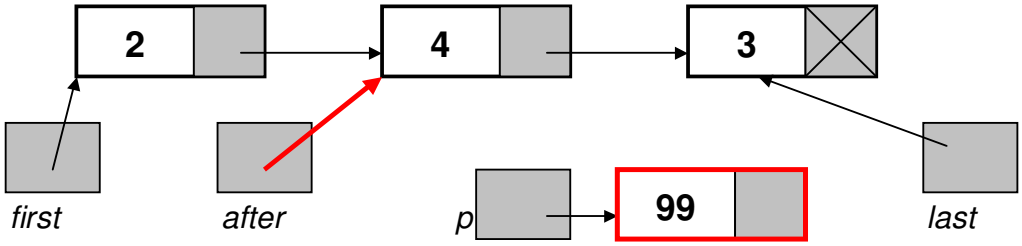
Noklusētais variants saistītā saraksta papildināšanai ir elementu pievienošana saraksta beigās, tomēr reizēm (piemēram, veidojot sakārtotu sarakstu) ir nepieciešams ievietot elementu saraksta vidū.

Ja ir tāda nepieciešamība, tad vispirms ir jāiegūst norāde uz elementu, aiz kura tiks iesprausts jaunais elements.

Nākošais piemērs sadalīts 3 daļās un par pamatu izmanto sarakstu, kāds tika iegūts iepriekšējā programmā (Att. 18.6).

- Sekojošais programmas fragments parāda, ka, lai aiz elementa 4 iespraustu 99, jāiegūst norāde *after* un jāizveido jaunais elements (piemērā netiek parādīts, kā norāde uz šo elementu (*after*) tiek iegūta).

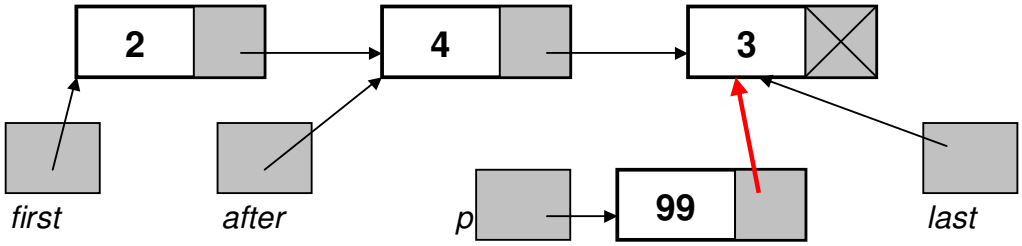
```
elem *after;
    ...
elem *p = new elem;;
p->num = i;   // aizpilda ar nolasīto vērtību
```



Att. 18.8. Elementa iespraušana sarakstā – norāde uz elementu, aiz kura iespraust – *after* un ievietojamais elements *p*

- Tad jaunajam elementam pieķēdē galā saraksta beigas.

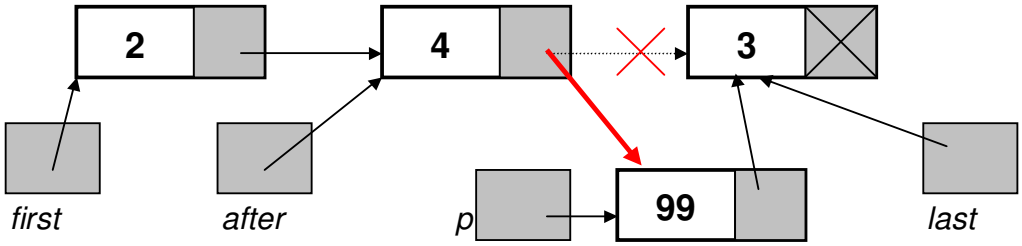
```
p->next = after->next;
```



Att. 18.9. Elementa iespraušana sarakstā – saraksta beigu piekabināšana jaunajam elementam

- Visbeidzot no iepriekšējā elementa (*after*) tiek novilkta saite uz jauno elementu.

```
after->next = p;
```



Att. 18.10. Elementa iespraušana sarakstā – iepriekšējā elementa saites novilkšana uz jauno elementu

18.3. Apvienojuma konstrukcija union

Apvienojums *union* ir īpašs struktūras veids, kurā visi mainīgie izmanto **vienu un to pašu atmiņas apgabalu**. Apvienojuma izmērs ir vienāds ar lielākā lauka izmēru.

Sintaktiski apvienojuma konstrukcija ir līdzīga *class* vai *struct* konstrukcijām, tikai to raksturo atslēgas vārds *union*.

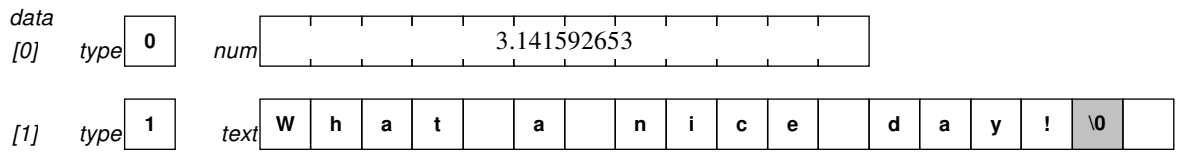
Apvienojumu parasti izmanto, lai noteiktu atmiņas apgabalu pēc vajadzības izmantotu dažādiem nolūkiem, tomēr teorētiski var izmantot arī apvienojuma doto iespēju apskatīt vienu un to pašu atmiņas apgabalu no dažādu tipu aspektiem.

Nākošais piemērs parāda veidu, kā vienā masīvā var glabāt gan skaitļus, gan tekstu. Tas parāda tipisku tehniku, kad konstrukcija *union* tiek izmantota komplektā ar *struct*.

```
...
union dbltxt
{
    long double num;
    char text[18];
};
// struktūra, kas var uzvesties divējādi
struct dual
{
    char type; // 0-skaitlis, 1-teksts
    dbltxt field; // mainīga tipa lauks
};
    ...
dual data[2];
data[0].type = 0;
data[0].field.num = 3.141592653;
data[1].type = 1;
strcpy (data[1].field.text, "What a nice day!");
for (int i=0; i<2; i++)
{
    if (data[i].type == 0)
        cout << data[i].field.num << endl;
    else
        cout << data[i].field.text << endl;
};
```

```
3.14159
What a nice day!
```

Att. 18.11. Konstrukcijas *union* izmantošana – masīvs no dažādu tipu vērtībām



Att. 18.12. Iepriekšējās programmas datu struktūra

19. Ģeneriskā programmēšana un šabloni

> **Ģeneriskā programmēšana** ir programmēšanas paradigma, kas ļauj pierakstīt algoritmus, izmantojot ģeneriskus (vispārinātus) datu tipus, kas tiek specificēti vēlāk.

Daudzās augsta līmeņa valodās šo datu tipu ģeneriskumu nodrošina t.s. dinamiskā tipu sistēma, bet valodās ar statisku tipu sistēmu, kāda ir C++, šo programmēšanas pieeju arsenālu daļēji nodrošina ģeneriskā programmēšana, kas strādā efektīvāk nekā dinamiskās tipu sistēmas pieeja.

19.1. Šablons kā mehānisms

Blakus daudzkāršai mantošanai un operatoru pārslogošanai **šablons** jeb **veidne** (*template*) ir mehānisms, kas nav sastopams daudzās citās objektorientētās programmēšanas valodās. Šabloni nodrošina pirmkoda vairākkārtēju izmantošana pēc nedaudz līdzīga principa kā mantošana, tomēr dod iespēju izveidot daudz efektīvākas konstrukcijas nekā izmantojot citas metodes.

Šablonu mehānisms ir ļoti spēcīgs, bet tajā pašā laikā arī ļoti sarežģīts. Piemērs šablonu izmantošanai standarta bibliotēkās ir t.s. **standarta šablonu bibliotēka** (*standard template library*, ***STL***), kas ietver sevī daudzu datu struktūru un algoritmu realizācijas. Tajā pašā laikā jaunajās realizācijās arī ievades un izvades bibliotēkas (tādas kā *iostream*, *fstream*) ir būvētas uz šablonu mehānisma bāzes.

Praktiski veidojot programmas, programmētājiem visbiežāk nāksies izmantot dažādas standarta bibliotēkas uz šablonu bāzes, nevis pašam veidot šablonus, tomēr paša šablonu mehānisma iepazīšana noteikti palīdz efektīvāk izmantot šīs bibliotēkas.

> **Šablons** ir mehānisms, kas reprezentē shēmu, pēc kuras var tikt veidotas klases vai funkcijas.

No viena šablona veidotās konstrukcijas savā starpā atšķiras ar atšķirīgu datu tipu komplektu – līdzīgi kā atšķiras divas funkcijas ar vienādu nosaukumu un parametru skaitu, bet atšķirīgiem parametru tipiem.

Klases šablona piemērs varētu būt "kastīte kaut kam", bet klases, kas no šī šablona varētu tikt veidotas, būtu, piemēram, "kastīte gurķiem" vai "kastīte tomātiem".

Šablonu mehānisms ir pielietojams divās formās:

- funkciju šabloni,
- klašu šabloni.

No programmēšanas viedokļa šablons ir tāda kā universāla funkcija vai klase, tomēr no statiski tipizētas valodas viedokļa, kādās izmanto šo paradigmu, šablons ir cita līmeņa konstrukcija, kas padara tās lietošanu sarežģītāku.

No viena šablona veidotās klases atšķiras pēc pielietotā datu tipu komplekta, tādējādi datu tipi ir šablona parametru vērtības, un klašu šablonus sauc arī par **parametrizētajiem tipiem**. Plašs klašu šablonu pielietojums ir dažādu datu struktūru realizācijā, kuru elementi var būt dažādu tipu vērtības, tāpēc klašu šabloni tiek saukti arī par **konteineru klasēm**.

Šablons programmas tekstā no parastas funkcijas vai klases atšķiras pēc divām pazīmēm:

- virs funkcijas/klases ir papildus rindiņa – šablona galva, kas deklarē, ka dotā funkcija/klase ir šablons, un satur formālo tipu sarakstu,
- funkcija/klase blakus parastiem tipiem (kā parametru, lauku vai lokālo mainīgo tipiem) satur arī formālos tipus, respektīvi, identifikatorus, kas vēlāk būs jāprecizē par konkrētiem tipiem.

```
template_definition:
        template < formal_type_list > template_body

formal_type_list:
        formal_type_declaration
        formal_type_declaration , formal_type_list

formal_type_declaration:
        class formal_type_name
        typename formal_type_name

template_body:
        class_definition
        class_mehod_definition
        function_declaration
        function_definition
```

Att. 19.1. Šablona definēšana (*template_definition*)

Pirms katra formālā tipa ir vai nu atslēgas vārds *class* vai *typename*. Šie abi atslēgas vārdi dotajā kontekstā ir sinonīmi – abi ir pielietojami gan funkciju, gan klašu šablonu veidošanai.

19.2. Funkciju šabloni

Funkciju šablons ir shēma funkciju veidošanai. Funkciju šablons ir vienkāršākais piemērs šablonu mehānisma demonstrēšanai.

Funkciju šablonu izveidošanai un izmantošanai ir šādas īpašības:

- funkciju šablona izveidošana ir līdzīga parastas funkcijas izveidošanai ar nekonkretizētiem datu tipiem, kam pievienota šablona galva,
- funkciju šablona izmantošana pilnīgi neatšķiras no parastas funkcijas izmantošanas.

Jāatceras, ka funkciju šablons nav universāla funkcija (kaut arī no izmantošanas viedokļa tā varētu likties), bet shēma funkciju izveidošanā, t.i. uz tā bāzes tiek izveidots tik daudz "īstu" funkciju, cik nepieciešams.

No tā izriet, ka

- funkciju šablona (arī klašu šablona) izmantošana nodrošina efektīva izpildāmā koda ģenerēšanu, jo katram gadījumam tiek izveidota sava funkcija (vai klase) (nevis viena universāla),
- tajā pašā laikā šabloni šī paša iemesla dēļ nenodrošina kompakta izpildāmā koda ģenerēšanu.

Lai šablonu drīkstētu izmantot noteiktam tipu komplektam, dotajos tipos ir jābūt definētām visām darbībām, kas pielietotas šablona definēšanā (nākamajā piemērā datu tipiem, ar kuriem tiks izmantots šablons, ir jābūt definētai saskaitīšanas operācijai (+) – tipiem *int* un *string* tāda ir definēta).

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

template <typename T>
T add (T a, T b)
{
    return a + b;
};

int main ()
{
    int x=2, y=3, z;
    string s="Hello, ", t="World!", u;
    z = add (x, y);
    u = add (s, t);
    cout << z << endl;
    cout << u << endl;
    return 0;
}
```

```
5
Hello, World!
```

Att. 19.2. Funkciju šablons

Šajā programmā, balstoties uz šablonu, kompilators patiesībā izveido divus šādu funkciju izpildāmos kodus:

```
int add (int a, int b)
{
    return a + b;
};
string add (string a, string b)
{
    return a + b;
};
```

19.3. Klašu šabloni

Klašu šablona izveidošana notiek pēc līdzīgiem principiem kā funkciju šablona izveidošana – šablonu klasi ievada identiska šablona galva, un klases definēšanā tiek izmantoti arī nekonkretizētie tipi, tomēr klašu šablona izmantošana objektu veidošanai ir sarežģītāka nekā funkciju šabloniem – kompilators pēc konteksta nevar noteikt, ar kādu aktuālo tipu komplektu jāveido objekts, tāpēc tas jāuzrāda tiešā veidā:

```
template_class_type:
        template_name < type_list >

type_list:
        data_type
        data_type , type_list
```

Att. 19.3. Ar šablonu definētas klases tips (*template_class_type*)

Aktuālo tipu komplekta uzrādīšana pie klases objekta deklarēšanas var pasliktināt programmas lasāmību, bet izeja ir konstrukcijas *typedef* lietošana, kas ļauj no klases šablona izveidoto klasi nosaukt ar konkrētu vārdu un tādējādi padarīt tālāko šīs klases izmantošanu ērtāku.

```
data_type_synonym_definition:
        typedef data_type alternative_name ;
```

Att. 19.4. Datu tipa sinonīma veidošana (data_type_synonym_definition)

Nākošajā piemērā šablons apraksta klases, kuras sastāv no diviem laukiem, kur katrs var būt ar savu tipu.

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

template <class T1, class T2>
struct two
{
    T1 first;
    T2 second;
    two (const T1 &f, const T2 &s)
    {
        first = f;
        second = s;
    };
    void print ()
    {
        cout << first << " " << second << endl;
    }
};
typedef two<string,int> person;

int main ()
{
    two<int,int> t (3,4);
    t.print ();
    two<string,string> s ("Hello,", "World!");
    s.print ();
    person p ("Liz", 19);
    p.print ();
    return 0;
}
```

```
3 4
Hello, World!
Liz 19
```

Att. 19.5. Klašu šablons

Uz klašu šablona bāzes tika izveidotas 3 dažādas klases:

```
two<int,int>
two<string,string>
two<string,int>
```

Lai atvieglotu klases *two<string,int>* izmantošanu, tika izveidots šī tipa sinonīms *person*:

```
typedef two<string,int> person;
```

Tas tālākajā kodā ļāva "aizmirst" par šablonu stūra iekavu notāciju.

Nākošais piemērs parāda dinamiska masīva definēšanu ar šablonu palīdzību (ietverot atmiņas vadību), kurš tiek darbināts ar *string* tipa vērtībām.

Tā kā klase satur (pašas klases nodrošinātu) dinamisku datu struktūru, tad korektai darbībai ir kritiski, ka klasē tiek definēts gan kopijas konstruktors, gan piešķiršanas operators.

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

template <class T>
class dynamicarray
{
    int Size;
    T *Array;
public:
    dynamicarray (int s=0)
    {
        Size = s;
        if (Size > 0) Array = new T[Size];
    };
    dynamicarray (const dynamicarray &d)
    {
        Size = d.Size;
        if (Size > 0) Array = new T[Size];
        for (int i=0; i<Size; i++) Array[i] = d.Array[i];
    };
    dynamicarray& operator= (const dynamicarray &d)
    {
        if (Size > 0) delete[] Array;
        Size = d.Size;
        if (Size > 0) Array = new T[Size];
        for (int i=0; i<Size; i++) Array[i] = d.Array[i];
        return *this;
    };
    ~dynamicarray ()
    {
        if (Size > 0) delete[] Array;
    };
    void append (const T &value)
    {
        T *tmparray = new T[Size+1];
        for (int i=0; i<Size; i++) tmparray[i] = Array[i];
        tmparray[Size] = value;
        if (Size > 0) delete[] Array;
        Size++;
        Array = tmparray;
    };
    void print ()
    {
        for (int i=0; i<Size; i++) cout << Array[i] << endl;
    };
    T& operator[] (int i) { return Array[i]; };
};
typedef dynamicarray<string> stringarray;

int main ()
{
    stringarray arrs;
    arrs.append ("Hello,");
    arrs.append ("Word!");
    arrs.print ();
    stringarray arrs2 = arrs;
    arrs2.print ();
    stringarray arrs3;
    arrs3 = arrs;
    arrs3.print ();
}
```

```
Hello,
Word!
Hello,
Word!
Hello,
Word!
```

Att. 19.6. Ar šablonu definēts dinamisks masīvs

20. Standarta šablonu bibliotēka (STL)

20.1. Kopsavilkums par STL

20.1.1. STL vispārīgs apraksts

Standarta šablonu bibliotēka (*standard template library*, ***STL***) ir C++ standarta bibliotēka, kas sevī ietver ģeneriskas (dažādiem datu tipiem paredzētas) datu struktūras un algoritmus.

STL izveidošana bija veiksmīgs solis C++ bibliotēku standartizēšanā, jo ilgu laiku praktiski vienīgās C++ standarta bibliotēkas bija tās, kas bija mantotas no C (ja neskaita ievades un izvades bibliotēkas).

STL sastāvā ietilpst 3 komponentu kategorijas:

- **Konteineri** – tās ir datu struktūras, kas veido STL pamatu (piemēram, *vector*, *stack*, *map*), un pārējo divu kategoriju komponentes nodarbojas ar to apkalpošanu.
- **Iteratori** – specializētas norādes piekļūšanai konteineru klašu vērtībām.
- **Algoritmi** – funkciju šabloni, kas apkalpo konteineru datus (piemēram, *sort()*).

Konteineru var uztvert kā datu struktūru, kas veidota no blokiem (“lego” klucīšiem), kur ir svarīgi, kā bloki ir savstarpēji izvietoti, bet nav svarīgi, kāda tipa vērtības blokos tiek glabātas. Savukārt iteratori ir norāžu (*pointers*) mehānisma adaptācija konteineriem, kas kalpo kā mehānisms manipulācijai ar konteinera klases blokiem (un tajos glabātajām vērtībām).

20.1.2. STL konteineru klases

STL satur daudzas konteineru klases, kas reprezentē vispārzināmas datu struktūras. Tas ir ļoti spēcīgs līdzeklis, tomēr prasa pietiekoši plašas zināšanas algoritmos un datu struktūrās, lai tās izmantotu “pēc pilnas programmas”. Nākošajā tabulā parādītas STL pieejamās konteineru klases, dažas no kurām tiks nedaudz apskatītas vēlāk.

Tab. 20.1.

STL konteineru klases

Iekļaujamā bibliotēka	Konteineru klases	Apraksts
`<bitset>`	`bitset`	bitu kopas
`<vector>`	`vector`	dinamiskie masīvi
`<list>`	`list`	saraksti
`<map>`	`map` `multimap`	kartes vai vārdnīcas – līdzīgi masīviem, bet indeksa lomu šeit var spēlēt arī citu tipu vērtības
`<stack>`	`stack`	steki – lineāras struktūras, kas organizētas pēc principa “pēdējais ienācis, pirmais izgājis” (LIFO)
`<queue>`	`queue` `priority_queue`	rindas – lineāras struktūras, kas organizētas pēc principa “pirmais ienācis, pirmais izgājis” (FIFO)
`<deque>`	`deque`	indeksētas divvirzienu rindas, vislīdzīgākie tipam *vector*, bet nenodrošina datu glabāšanu pēc kārtas

`<set>`	`set` `multiset`	kopas

20.1.3. STL iteratori un algoritmi

Iteratori ir mehānismi, kas paredzēti piekļūšanai konteineru elementiem, kas lielā mērā (gan idejiski, gan sintaktiski) atgādina norādes (*pointers*). Iteratorus deklarē, izmantojot tipu *iterator*, kas definēts konteineru klasēs (nevis, pielietojot papildus * simbolu, kā tas pierasts pie norādēm). Divi svarīgi daudzu konteineru klašu iebūvētie iteratori ir *begin()* un *end()*, kas norāda attiecīgi uz konteinera pirmo elementu un elementu aiz pēdējā (pēc līdzības ar beigu simbolu zema līmeņa simbolu virknēs). Iteratori var būt vairāku tipu – patvaļīgas pieejas (*random access*), divvirzienu (*bidirectional*) u.c., un to, kāds iterators tiek izmantots, nosaka konteineru klases tips un konteksts. Iteratoru darbības princips tiks demonstrēts, apskatot konkrētas konteineru klases.

STL piedāvā vairāku algoritmu realizācijas, kas apkalpo konteineru klases, piemēram, meklēšana, kārtošana, samainīšana vietām.

20.2. STL klase 'vector'

STL klase *vector* (vektors) realizē dinamisku masīvu ar daudzām papildus īpašībām, kas atgādina klasi *dynamicarray*, kas definēta piemērā Att. 19.6.

Klase *vector* ir viena no populārākajām STL konteineru klasēm. Tās izmantošanas ērtums balstās uz piekļuves iespēju elementu vērtībām, izmantojot indeksu. Tajā pašā laikā jāņem vērā, ka, lai nodrošinātu piekļuvi pēc indeksa, šajā struktūrā nepieciešami mehānismi, kas prasa papildus skaitļošanas un atmiņas resursus šī servisa nodrošināšanai.

20.2.1. Vektora sākotnēja aizpildīšana

Klases *vector* objekta izveidošana un aizpildīšana iespējama šādos veidos:

- tukša vektora izveidošana,
- vairāku elementu vektora ar fiksētu vērtību izveidošana,
- inicializācijas virknes izmantošana aizpildīšanai (sākot ar C++11),
- cita vektora vai masīva pilna vai daļēja kopēšana.

```
#include <vector>
#include <iostream>
using namespace std;
int main ()
{
    vector<int> aa; // tukšs vektors
    for (auto &a: aa) { cout<<a<<" "; }; cout<<"//"<<endl;
    vector<int> bb (5,99); // fiksētas vērtības pavairošana
    for (auto &a: bb) { cout<<a<<" "; }; cout<<"//"<<endl;
    vector<int> cc = {11,22,33}; // inicializācijas virkne
    for (auto &a: cc) { cout<<a<<" "; }; cout<<"//"<<endl;
    vector<int> dd = cc; // pilna kopija
    for (auto &a: dd) { cout<<a<<" "; }; cout<<"//"<<endl;
    auto prelast = cc.end();
    --prelast;
    vector<int> ee (cc.begin(), prelast); // daļēja kopija
    for (auto &a: ee) { cout<<a<<" "; }; cout<<"//"<<endl;
}
```

```
//
99 99 99 99 99 //
11 22 33 //
11 22 33 //
11 22 //
```

Att. 20.1. 'vector' objekta izveidošana un aizpildīšana

Paskaidrojumi (Att. 20.1):

- Speciālais **for** cikls. Vektora izdrukāšana šeit notiek, izmantojot spciālu for ciklu *for (vērtība: struktūra)*.
- **auto**. Mehānisms *auto* šeit reprezentē elementa vērtības datu tipu (*int*). auto nozīmē nevis "universālu" datu tipu, bet gan mehānismu, kas konkrētu datu tipu noskaidro pēc konteksta (šeit vektorā no *int* viens elements ir ar datu tipu *int*). Programmā šis konkrētais datu tips neparādās, tomēr kompilēšanas brīdī tas ir noskaidrojams, un izpildāmajā modulī tas jau tiek ievietots (nevis *auto*).
- *begin()* – iterators uz vektora pirmo elementu.
- *end()* – iterators uz pēdējo (fiktīvo) elementu (sk. ilustrāciju Att. 20.3)
- *--prelast* – iteratora prelast pārvietošana vienu pozīciju atpakaļ.

20.2.2. Piekļūšana vektora elementiem un vektora pārstaigāšana

Saraksta *list* elementiem nevar piekļūt, izmantojot indeksu, tomēr ir vairāki citi veidi, lai piekļūtu vienam vai vairākiem elementiem:

- pārstaigāšana ar indeksu,
- *for* cikls (intervālam),
- ar iteratoru (piemēram, pārstaigājot intervālu),
- intervāla pārstaigāšana ar *for_each*, izmantojot papildus funkciju, kura apstrādā katru elementu,
- pirmā elementa vērtība – *front()*,
- pēdējā elementa vērtība – *back()*.

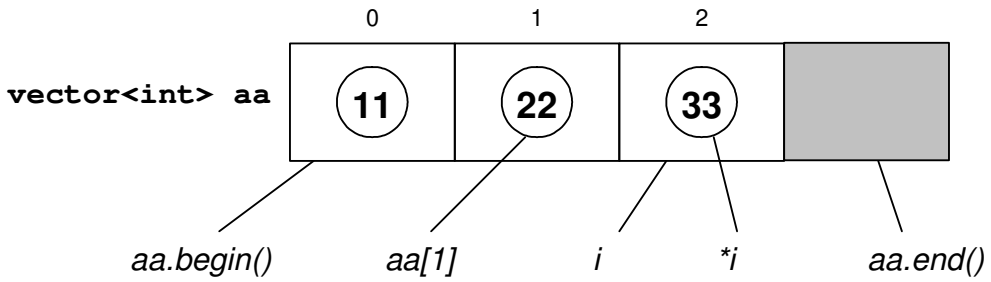
```
#include <algorithm>
#include <vector>
#include <iostream>
using namespace std;
void psqr(int i) { // šādi apstrādā katru elementu
    cout<<i*i<<" ";
}
int main ()
{
    vector<int> aa = {11,22,33};
    // pārstaigāšana ar indeksu
    for (int i=0; i<aa.size(); i++)
        { cout<<aa[i]<<" "; };
    cout<<"//"<<endl;
    // for cikls intervālam
    for (auto &a: aa)
        { cout<<a<<" "; };
    cout<<"//"<<endl;
    // intervāla pārstaigāšana ar iteratoru
    for (auto i=aa.begin();i!=aa.end();++i)
        { cout<<*i<<" "; };
    cout<<"//"<<endl;
    // intervāla pārstaigāšana ar foreach
    for_each (aa.begin(),aa.end(),psqr);
    cout<<"//"<<endl;
    cout<<"Pirmais "<<aa.front()<<endl;
    cout<<"Beidzamais "<<aa.back()<<endl;
}
```

```
11 22 33 //
11 22 33 //
11 22 33 //
121 484 1089 //
Pirmais 11
Beidzamais 33
```

Att. 20.2. Piekļuve ‘vector’ elementiem

Paskaidrojumi (Att. 20.2):

- *++i* – iteratora i pārvietošana par vienu pozīciju uz priekšu,
- **i* – elementa vērtības iegūšana pēc iteratora,
- *size()* – vektora izmērs.



Att. 20.3. Programmā Att. 20.2 izveidotais vektors

20.2.3. Elementu pievienošana vektoram un elementu izdzēšana no vektora

Elementa pievienošana vektoram iespējama šādi:

- palielinot vektora izmēru (*resize*),
- beigās (*push_back*),
- vidū, pirms elementa, uz kuru norāda iterators (*insert*).

Sākumā vektoram pievienot elementu ar speciālu metodi nevar, tādējādi tiek demonstrēts, ka to darīt vektorā ir neefektīvi, un, ja tas jādara, tad būtu vēlams izmantot citu datu struktūru (piemēram, *deque*). Tai pašā laikā to tomēr var izdarīt, izmantojot *insert* pirms pirmā elementa (*begin*).

```
#include <algorithm>
#include <vector>
#include <iostream>
using namespace std;
int main ()
{
    vector<int> aa = {11,22,33,44,55};
    for (auto &a: aa) { cout<<a<<" "; };
    cout<<"//"<<endl;
    aa.resize(7,-1); // palielināšana ar noklusēto vērtību -1
    for (auto &a: aa) { cout<<a<<" "; };
    cout<<"//"<<endl;
    aa.insert(aa.begin(),77); // sākumā ar insert
    for (auto &a: aa) { cout<<a<<" "; };
    cout<<"//"<<endl;
    aa.push_back(99); // beigās
    for (auto &a: aa) { cout<<a<<" "; };
    cout<<"//"<<endl;
    auto a=aa.begin();
    ++a; ++a; ++a;
    aa.insert(a,888); // pirms 33
    for (auto &a: aa) { cout<<a<<" "; };
    cout<<"//"<<endl;
    vector<int> bb = {111,222,333};
    aa.insert(a,bb.begin(),bb.end()); // pirms 33 intervālu
    for (auto &a: aa) { cout<<a<<" "; };
    cout<<"//"<<endl;
}
```

```
11 22 33 44 55 //
11 22 33 44 55 -1 -1 //
77 11 22 33 44 55 -1 -1 //
77 11 22 33 44 55 -1 -1 99 //
77 11 22 888 33 44 55 -1 -1 99 //
77 11 22 111 222 333 888 33 44 55 -1 -1 99 //
```

Att. 20.4. Elementu pievienošana vektoram

Elementu dzēšana no vektora iespējama šādi:

- no beigām (*pop_back*),
- pēc iteratora (*erase*) – ja vajag no sākuma, izmanto šo,
- samazina izmēru (*resize*).

```
#include <algorithm>
#include <vector>
#include <iostream>
using namespace std;
int main ()
{
    vector<int> aa = {11,22,33,44,55,66,77};
    for (auto &a: aa) { cout<<a<<" "; };
    cout<<"//"<<endl;
    aa.erase(aa.begin()); // izdzēš pirmo, izmantojot erase
    for (auto &a: aa) { cout<<a<<" "; };
    cout<<"//"<<endl;
    aa.pop_back(); // izdzēš pēdējo
    for (auto &a: aa) { cout<<a<<" "; };
    cout<<"//"<<endl;
    auto i=aa.begin();
    ++i; ++i;
    i = aa.erase(i); // izdēš pēc iteratora
    // erase atgriež iteratoru uz nākošo pēc izdzēstā
    for (auto &a: aa) { cout<<a<<" "; };
    cout<<"//"<<endl;
    aa.resize(2); // samazina līdz 2 vērtībām
    for (auto &a: aa) { cout<<a<<" "; };
    cout<<"//"<<endl;
}
```

```
11 22 33 44 55 66 77 //
22 33 44 55 66 77 //
22 33 44 55 66 //
22 33 55 66 //
22 33 //
```

Att. 20.5. Elementu izdzēšana no vektora

20.3. STL klase 'list'

STL klase *list* (saraksts) ir līdzīga klasei *vector* ar to atšķirību, ka tajā nav pieejama piekļuve elementu vērtībām, izmantojot indeksu. Tomēr, ja tas nav nepieciešams, tad labāk vektora vietā lietot tieši sarakstu, jo šī konstrukcija ir mazāk prasīga pēc resursiem struktūras izmēra maiņas gadījumā (piemēram, pievienojot jaunu elementu). Tai pat laikā klasē *list* ir pieejamas daudzas ļoti vērtīgas metodes, piemēram, elementa iespraušana.

20.3.1. Saraksta sākotnēja aizpildīšana

Klases *list* objekta izveidošana un aizpildīšana iespējama šādos veidos (Att. 20.6):

- tukša saraksta izveidošana,
- vairāku elementu saraksta ar fiksētu vērtību izveidošana,
- inicializācijas virknes izmantošana aizpildīšanai (sākot ar C++11),
- cita saraksta vai masīva pilna vai daļēja kopēšana.

```
#include <list>
#include <iostream>
using namespace std;
int main ()
{
    list<int> aa; // tukšs saraksts
    for (auto &a: aa) { cout<<a<<" "; }; cout<<"//"<<endl;
    list<int> bb (5,99); // fiksētas vērtības pavairošana
    for (auto &a: bb) { cout<<a<<" "; }; cout<<"//"<<endl;
    list<int> cc = {11,22,33}; // inicializācijas virkne
    for (auto &a: cc) { cout<<a<<" "; }; cout<<"//"<<endl;
    list<int> dd = cc; // pilna kopija
    for (auto &a: dd) { cout<<a<<" "; }; cout<<"//"<<endl;
    auto prelast = cc.end();
    --prelast;
    list<int> ee (cc.begin(), prelast); // daļēja kopija
    for (auto &a: ee) { cout<<a<<" "; }; cout<<"//"<<endl;
}
```

```
//
99 99 99 99 99 //
11 22 33 //
11 22 33 //
11 22 //
```

Att. 20.6. ‘list’ objekta izveidošana un aizpildīšana

20.3.2. Piekļūšana saraksta elementiem un saraksta pārstaigāšana

Saraksta *list* elementiem nevar piekļūt, izmantojot indeksu, tomēr ir vairāki citi veidi, lai piekļūtu vienam vai vairākiem elementiem (Att. 20.7):

- *for* cikls (intervālam),
- ar iteratoru (piemēram, pārstaigājot intervālu),
- intervāla pārstaigāšana ar *for_each*, izmantojot papildus funkciju, kura apstrādā katru elementu,
- pirmā elementa vērtība – *front()*,
- pēdējā elementa vērtība – *back()*.

```
#include <algorithm>
#include <list>
#include <iostream>
using namespace std;
void psqr(int i) { // šādi apstrādā katru elementu
    cout<<i*i<<" ";
}
int main ()
{
    list<int> aa = {11,22,33,44,55};
    // for cikls intervālam
    for (auto &a: aa)
        { cout<<a<<" "; };
    cout<<"//"<<endl;
    // intervāla pārstaigāšana ar iteratoru
    for (auto i=aa.begin();i!=aa.end();++i)
        { cout<<*i<<" "; };
    cout<<"//"<<endl;
    // intervāla pārstaigāšana ar foreach
    for_each (aa.begin(),aa.end(),psqr);
    cout<<"//"<<endl;
    cout<<"Pirmais "<<aa.front()<<endl;
    cout<<"Beidzamais "<<aa.back()<<endl;
}
```

```
11 22 33 44 55 //
11 22 33 44 55 //
121 484 1089 1936 3025 //
Pirmais 11
Beidzamais 55
```

Att. 20.7. Piekļuve ‘list’ objekta elementiem

20.3.3. Elementu pievienošana sarakstam un elementu izdzēšana no saraksta

Elementa pievienošana sarakstam iespējama šādi:

- sākumā (*push_front*),
- beigās (*push_back*),
- vidū, pirms elementa, uz kuru norāda iterators (*insert*).

```
#include <algorithm>
#include <list>
#include <iostream>
using namespace std;
int main ()
{
    list<int> aa = {11,22,33,44,55};
    for (auto &a: aa) { cout<<a<<" "; };
    cout<<"//"<<endl;
    aa.push_front(77); // sākumā
    for (auto &a: aa) { cout<<a<<" "; };
    cout<<"//"<<endl;
    aa.push_back(99); // beigās
    for (auto &a: aa) { cout<<a<<" "; };
    cout<<"//"<<endl;
    auto a=aa.begin();
    ++a; ++a; ++a;
    aa.insert(a,888); // pirms 33
    for (auto &a: aa) { cout<<a<<" "; };
    cout<<"//"<<endl;
    list<int> bb = {111,222,333};
    aa.insert(a,bb.begin(),bb.end()); // pirms 33 intervālu
    for (auto &a: aa) { cout<<a<<" "; };
    cout<<"//"<<endl;
}
```

```
11 22 33 44 55 //
77 11 22 33 44 55 //
77 11 22 33 44 55 99 //
77 11 22 888 33 44 55 99 //
77 11 22 888 111 222 333 33 44 55 99 //
```

Att. 20.8. Elementu pievienošana sarakstam

Elementu dzēšana no saraksta iespējama šādi:

- no sākuma (*pop_front*),
- no beigām (*pop_back*),
- pēc iteratora (*erase*),
- pēc vērtības (*remove*).

```
#include <algorithm>
#include <list>
#include <iostream>
using namespace std;
int main ()
{
    list<int> aa = {11,22,33,44,55,66,77};
    for (auto &a: aa) { cout<<a<<" "; };
    cout<<"//"<<endl;
    aa.pop_front(); // izdzēš pirmo
    for (auto &a: aa) { cout<<a<<" "; };
    cout<<"//"<<endl;
    aa.pop_back(); // izdzēš pēdējo
    for (auto &a: aa) { cout<<a<<" "; };
    cout<<"//"<<endl;
    auto i=aa.begin();
    ++i; ++i;
    i = aa.erase(i); // izdzēš iteratora norādīto
    // erase atgriež iteratoru uz nākošo pēc izdzēstā
    for (auto &a: aa) { cout<<a<<" "; };
    cout<<"//"<<endl;
    aa.remove(55); // izdzēš pēc vērtības
    for (auto &a: aa) { cout<<a<<" "; };
    cout<<"//"<<endl;
}
```

```
11 22 33 44 55 66 77 //
22 33 44 55 66 77 //
22 33 44 55 66 //
22 33 55 66 //
22 33 66 //
```

Att. 20.9. Elementu izdzēšana no saraksta

20.4. STL algoritmi

20.4.1. STL algoritmi ‘find’, ‘find_if’, ‘binary_search’

Vērtību atrašana virknē var notikt šādos veidos:

- norādot konkrētu vērtību (*find*),
- norādot derīguma nosacījumu funkcijas formā (*find_if*),
- meklēšana sakārtotā virknē (*binary_search*).

Ja attiecīgā metode atrod elementu pēc nosacījuma, tā atgriež iteratoru uz šo elementu, citādi atgriež iteratoru *end()*.

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
int main () {
  vector<int> aa={12,25,40,57};
  int val;
  cin>>val;
  auto i = find (aa.begin(), aa.end(), val);
  if (i != aa.end())
     cout<<"Exists: "<<*i<<endl;
  else
     cout<<"NOT FOUND"<<endl;
}
```

```
40
Exists: 40
```

```
22
NOT FOUND
```

Att. 20.10. Elementa atrašana pēc vērtības ar *find*

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
bool IsOdd (int val) {
  return val%2 != 0;
}
int main () {
  vector<int> aa={12,25,40,57};
  // izmantojot parasto funkciju nepāra skaitļiem
  auto i = find_if (aa.begin(), aa.end(), IsOdd);
  cout << "The first odd value is " << *i << '\n';
  // izmantojot anonīmo funkciju pāra skaitļiem
  auto k = find_if (aa.begin(), aa.end(), [](int val) { return
                 val%2==0; });
  cout << "The first even value is " << *k << '\n';
}
```

```
The first odd value is 25
The first even value is 12
```

Att. 20.11. Elementa atrašana pēc nosacījuma ar *find_if*

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
int main () {
  vector<int> aa={12,25,40,57};
  int val;
  cin>>val;
  binary_search (aa.begin(), aa.end(), val);
  if (binary_search (aa.begin(), aa.end(), val))
     cout<<"Exists: "<<val<<endl;
  else
     cout<<"NOT FOUND: "<<val<<endl;
}
```

```
40
Exists: 40
```

```
22
NOT FOUND: 22
```

Att. 20.12. Binārā meklēšana sakārtotā virknē ar *binary_search*

20.4.2. STL algoritmi ‘count’ un ‘count_if’

Vērtību atrašana virknē var notikt šādos veidos:

- norādot konkrētu vērtību (*find*),
- norādot derīguma nosacījumu funkcijas formā (*find_if*).

Ja attiecīgā metode atrod elementu pēc nosacījuma, tā atgriež iteratoru uz šo elementu, citādi atgriež iteratoru *end()*.

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
int main () {
  vector<int> aa={10,25,40,55,40,13};
  cout<<count (aa.begin(), aa.end(), 40)<<endl;
}
```

```
2
```

Att. 20.13. Elementu saskaitīšana pēc vērtības ar *count*

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
bool IsOdd (int val) {
  return val%2 != 0;
}
bool IsEven (int val) {
  return val%2 == 0;
}
int main () {
  vector<int> aa={10,25,47,55,40,13};
  int cnt = count_if (aa.begin(), aa.end(), IsOdd);
  cout << "Amount of odd values is " << cnt << '\n';
  cnt = count_if (aa.begin(), aa.end(), IsEven);
  cout << "Amount of even values is " << cnt << '\n';
}
```

```
Amount of odd values is 4
Amount of even values is 2
```

Att. 20.14. Elementu saskaitīšana pēc nosacījuma ar *count_if*

20.4.3. STL algoritmi ‘max’, ‘min’, ‘max_element’, ‘min_element’, ‘swap’

STL bibliotēkā pieejami arī ‘vienkāršie algoritmi’:

- maksimālā vērtība no divām (*max*),
- maksimālā vērtība virknē, iteratoru uz to (*max_element*),
- minimālā vērtība no divām (*min*),
- minimālā vērtība virknē, iteratoru uz to (*min_element*),
- vērtību samainīšana vietām (*swap*).

Ja attiecīgā metode atrod elementu pēc nosacījuma, tā atgriež iteratoru uz šo elementu, citādi atgriež iteratoru *end()*.

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
int main () {
  vector<int> aa={35,10,25,40,55,40,13};
  for (auto &a: aa) { cout<<a<<" "; };
  cout<<"//"<<endl;
  cout<<"max: "<<*max_element (aa.begin(), aa.end())<<endl;
  cout<<"min: "<<*min_element (aa.begin(), aa.end())<<endl;
  int a=5, b=7;
  cout <<a<<" "<<b<<endl;
  cout<<"max: "<<max(a,b)<<endl;
  cout<<"min: "<<min(a,b)<<endl;
  swap(a,b);
  cout <<a<<" "<<b<<endl;
}
```

```
35 10 25 40 55 40 13 //
max: 55
min: 10
5 7
max: 7
min: 5
7 5
```

Att. 20.15. STL vienkāršās funkcijas

20.4.4. STL algoritms ‘sort’

Viens no redzamākajiem STL algoritmiem ir kārtošanas algoritms *sort()*. Ar to var kārtot ne tikai *vector*, bet arī *deque* elementus (konteineram *list* ir realizēta iekšēja *sort* metode).

Kārtošanas algoritms izmantojams divos veidos:

- **Standarta variants**, izmantojot standarta salīdzināšanas (“mazāk”) kritēriju, kāds pieejams struktūrā glabājamiem objektiem.
- **Specializētais variants**, izmantojot speciāli izveidotu salīdzināšanas kritēriju.

Kārtošanas algoritma izmantošana standarta variantā. Tiek izsaukta funkcija *sort()* ar diviem parametriem – (1)sākuma iterators un (2)beigu iterators (jāatceras, ka beigu iteratoram jānorāda nevis uz beidzamo kārtojamo elementu, bet nākošo aiz tā).

Kārtošanas algoritma izmantošana specializētajā variantā. Tiek izsaukta funkcija *sort()* ar trīs parametriem – (1) sākuma iterators, (2) beigu iterators un (3) salīdzināšanas kritērijs. Salīdzināšanas kritērijs ir funkcija, kas par diviem objektiem, kādus var glabāt dotā struktūra, pasaka – vai pirmais ir mazāks par otro.

Funkcija, kas definē salīdzināšanas kritēriju, ir jānoformē noteiktā formātā:

- kā speciāli izveidotas klases operatora *operator()* pārdefinēšana (pārslogošana) vai
- speciāla salīdzināšanas funkcija.

Kad šāda klase (vai funkcija) izveidota, tad, izsaucot kārtošanas funkciju *sort()*, trešajā parametrā tiek padots klases nosaukums ar tukšām iekavām (vai funkcijas nosaukums).

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

typedef vector<int> intarray; // sinonīms datu tipam
                 vector<int>

class lastdigit_compare
{
public:
    bool operator() (const int &v1, const int &v2) const
    {
        return v1%10 < v2%10;
    }
};

int main ()
{
    intarray aa={52,28,47};
    // sakārto pēc skaitļa vērtības (noklusēti):
    sort (aa.begin(), aa.end());
    for (auto &a: aa) { cout<<a<<" "; };
    cout<<"//"<<endl;
    // sakārto pēc pēdējā cipara vērtības:
    sort (aa.begin(), aa.end(), lastdigit_compare());
    for (auto &a: aa) { cout<<a<<" "; };
    cout<<"//"<<endl;
}
```

```
28 47 52 //
52 47 28 //
```

Att. 20.16. STL algoritma ‘sort’ izmantošana, izmantojot speciālo klasi

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

typedef vector<int> intarray; // sinonīms datu tipam
                  vector<int>

bool lastdigit_compare (const int &v1, const int &v2)
{
    return v1%10 < v2%10;
};

int main ()
{
    intarray aa={52,28,47};
    // sakārto pēc skaitļa vērtības:
    sort (aa.begin(), aa.end());
    for (auto &a: aa) { cout<<a<<" "; };
    cout<<"//"<<endl;
    // sakārto pēc pēdējā cipara vērtības:
    sort (aa.begin(), aa.end(), lastdigit_compare);
    for (auto &a: aa) { cout<<a<<" "; };
    cout<<"//"<<endl;
}
```

```
28 47 52 //
52 47 28 //
```

Att. 20.17. STL algoritma ‘sort’ izmantošana, izmantojot speciālo salīdzināšanas funkciju

20.5. STL klase ‘map’

STL klase *map* (karte) nodrošina vērtību pāru glabāšanu, kā arī piekļūšanu elementiem pēc pirmās vērtības. No izmantošanas viedokļa karte līdzinās vārdnīcai. Ja elementa pirmās vērtības tips ir *int*, tad *map* praktiski nodrošina vektora funkcionalitāti. Pievienojot jaunu elementu kartei, tiek automātiski nodrošināta *map* elementu sakārtošana pēc elementu pirmajām vērtībām. Tā kā klase *map* nodrošina vērtību pāru glabāšanu, tā izmanto STL klasi *pair* viena elementa vērtību glabāšanai.

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

typedef map<string,string> dictionary;
typedef dictionary::iterator dictit;

int main ()
{
    dictionary d;
    // aizpildīšana ar masīva sintaksi
    d["day"] = "giorno";
    d["street"] = "strada";
    d["italian"] = "italiano";
    d["red"] = "rosso";
    string s;
    cin >> s; // meklējamais vārds
    dictit it = d.find (s);
    if (it != d.end())
        cout << it->first << " " << it->second << endl;
    else cout << "NOT FOUND";
    return 0;
}
```

```
red
red rosso
```

```
blue
NOT FOUND
```

Att. 20.18. STL klases ‘map’ izmantošana

Piemērā izmantoto klases *map* īpašību apraksts:

- *begin()* – iterators (norāde) uz pirmo elementu,
- *end()* – iterators (norāde) uz virtuālu elementu aiz pēdējā,
- *[]* – piekļuve elementa vērtībai (struktūrai *pair*) pēc pirmās vērtības; vai elementa pievienošana struktūrai, automātiski nodrošinot tās sakārtošanu,
- *map<T1,T2>::iterator* – kartes iteratora tips,
- *it++* – iteratora pāreja uz nākošo elementu,
- *it->first, *it.first* – piekļuve elementa pirmajai vērtībai, izmantojot iteratoru,
- *it->second, *it.second* – piekļuve elementa otrajai vērtībai, izmantojot iteratoru,
- *find()* – funkcija, kas atgriež iteratoru uz elementu ar doto pirmo vērtību (ja tāds elements neeksistē, tad atgriež iteratoru *end()*).

20.6. STL klase ‘pair’

STL klase *pair* nodrošina divu dažādu vērtību glabāšanu vienā struktūrā, kas atgādina klasi *two*, kas definēta piemērā Att. 19.5.

Klase *pair* ietver divus laukus: *first* un *second*, un to izmanto arī citi konteineri, piemēram, *map*, kas ir apskatīts šajā materiālā.

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

typedef pair<string,int> person;

int main ()
{
    person p ("Liz", 19);
    cout << p.first << " " << p.second << endl;
    return 0;
}
```

```
Liz 19
```

Att. 20.19. STL klases ‘pair’ izmantošana

Vairāk nekā divu dažādu vērtību glabāšanai tiek izmantots konteiners *tuple*.

21. Izņēmumu apstrāde, nosaukumu telpas, norādes uz funkcijām, komandrindas argumenti

21.1. Izņēmumu apstrāde

Izņēmumu apstrāde (*exception handling*) ir mehānisms, kas ļauj noteiktā veidā strukturēt programmu, lai nošķirtu kļūdu apstrādes daļu no pārējās programmas daļas.

Konceptuāli izņēmumu apstrāde ietver divas galvenās darbību grupas, kuras saistītas ar 3 speciāliem atslēgas vārdiem (*try*, *throw*, *catch*) un kas notiek pēc shēmas, kas parādīta Att. 18.6:

- **Izņēmuma izmešana** (*throw*) – pirmkoda daļas izpildes pārtraukšana kļūdas dēļ un paziņošana par kļūdu. Izņēmumu izmešana notiek speciālā *try* blokā.
- **Izņēmuma pārtveršana** (*catch*), nodrošināt izņēmuma gadījuma apstrādi atbilstoši izņēmuma tipam. Izņēmumu uztveršana notiek uzreiz aiz *try* bloka, kurā varētu notikt izņēmumu izmešana.

```
try
{
    ...
    if (kļūdas veids nr. 1) throw some_exception1;
    ...
    if (kļūdas veids nr. 1) throw some_exception2;
    ...
}
catch (T1 exc)
    { /* izņēmumu apstrāde ar tipu T1 */ }
catch (T2 exc)
    { /* izņēmumu apstrāde ar tipu T2 */ }
catch (...)
    { /* citu izņēmumu apstrāde */ }
```

Att. 21.1. Izņēmumu apstrādes pieraksta shēma C++

Praktiski kļūdu apstrādi var realizēt arī, neizmantojot izņēmumu apstrādes mehānismu, bet tādā gadījumā pašam programmētājam ir jāveido konstrukcijas kļūdas paziņojumu nosūtīšanai uz kļūdu apstrādes vietu, kas varētu izpausties kā speciālas nozīmes funkcijas atgriežamās vērtības vai pat papildus parametri funkcijām, kas būtu paredzēti kļūdu paziņojumu pārsūtīšanai starp moduļiem.

Īpaši ērta izņēmumu apstrāde varētu būt gadījumos, kad kļūda tiek konstatēta vienā, bet apstrādāta citā funkciju izsaukumu hierarhijas līmenī. Tajā pašā laikā jāsaprot, ka standarta izņēmumu apstrādes mehānisma izmantošana saistīta ar vērā ņemamu papildus resursu izmantošanu.

‘try’ bloks.

‘try’ bloks ir programmas daļa, kas aiz atslēgas vārda *try* iekļauta figūriekavu blokā, uz kuru (ieskaitot funkcijas visos līmeņos, kas tiek izsauktas no šī bloka) attiecas aiz šī bloka realizētā izņēmuma situāciju apstrāde, piemēram,

```
try
{
    // izņēmumu situāciju noskaidrošanas zona
}
```

Kļūdas situācijas iestāšanos noskaidro pats programmētājs – izņēmumu apstrādes infrastruktūra tikai palīdz kļūdas paziņojumu nosūtīt uz apstrādes vietu.

Izņēmuma izmešana ar 'throw'.

Izmantojot operatoru *throw*, noteikts kļūdas paziņojums (skaitlis, teksts, objekts) tiek nosūtīts pārtveršanai uz attiecīgo *catch* bloku. Izmetot izņēmumu, programma attiecīgajā vietā tiek pārtraukta un tiek nodrošināta visos līmeņos izsaukto funkciju pabeigšana (un attiecīgi lokālo mainīgo likvidēšana), lai nonāktu līdz kļūdu apstrādes blokam.

Izmetot izņēmumu, ir svarīga ne tikai vērtība, bet arī tips, jo tieši atbilstoši izmestajam tipam tiks izvēlēta izņēmuma apstrādes funkcija.

```
throw "ERROR: Division by zero"
```

Izņēmumu apstrāde ar 'catch'.

Aiz *try* bloka atrodas *catch* bloku kopums, kas atgādina funkciju *catch* bez atgriežamā tipa un vienu parametru kopumu, tādējādi nodrošinot izņēmuma apstrādi atbilstoši tā tipam.

catch_section:
catch_block_list$_{opt}$ *default_catch_block*

catch_block_list:
catch_block
catch_block_list catch_block

catch_block:
`catch` `(` *parameter_definition* `)` *catch_body*

default_catch_block:
`catch` `(` `...` `)` *catch_body*

Att. 21.2. Izņēmumu apstrādes sadaļa (*catch_section*)

- Ja *catch* bloks ar attiecīgo tipu neeksistē, tad izņēmuma apstrāde notiek *catch(...)* blokā.
- Ja *catch(...)* bloks neeksistē, tad kļūda tiek pārsūtīta tuvākajam ārējam kļūdu apstrādes blokam, bet, ja tāds neeksistē, programma beidzas ar kļūdu.

Nākošais piemērs demonstrē izņēmumu apstrādes mehānismu funkcijai *process*, kas rēķina izteiksmes $x + y / z$ vērtību. Rēķinot funkcijas vērtību, nedrīkst tikt pārkāpti šādi nosacījumi – z nav 0, un saskaitāmie ir robežās 0..100.

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

double x_over_y (double x, double y)
{
    if (y == 0) throw "ERROR: Division by zero";
    return x / y;
};

double x_plus_y_over_z (double x, double y, double z)
{
    double div = x_over_y (y, z);
    if (x > 100 || div > 100) throw 100;
    else if (x < 0 || div < 0) throw 0.0;
    return x + div;
};

void process (double x, double y, double z)
{
    try
    {
        cout << x_plus_y_over_z (x, y, z) << endl;
    }
    catch (const char *error_text)
        { cout << error_text << endl; }
    catch (int error_num)
        { cout << "ERROR: Addend too big" << endl; }
    catch (...)
        { cout << "ERROR: Unknown error" << endl; };
};

int main ()
{
    /* Meklētās kļūdas:
        - nedrīkst dalīt ar 0
        - saskaitīšanas operācija apstrādā skaitļus 0..100 */
    process (2, 4, 0);     // 2 + 4 / 0
    process (999, 4, 0.5); // 999 + 4 / 0.5
    process (-2, 4, 0.5);  // -2 + 4 / 0.5
    process (2, 4, 0.5);   // 2 + 4 / 0.5
    return 0;
}
```

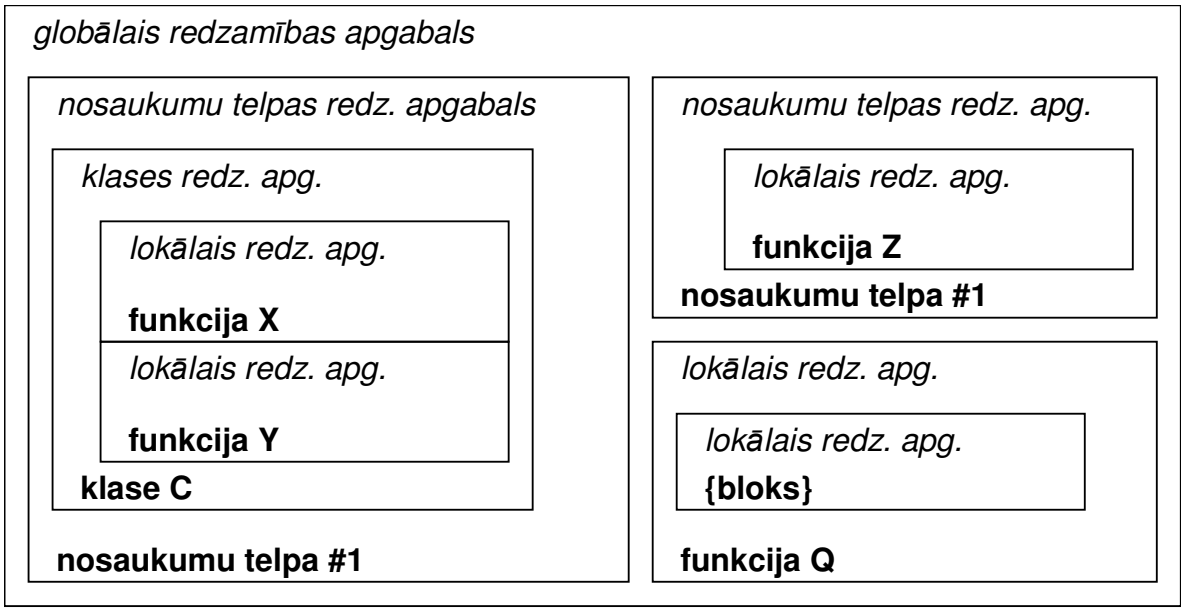
```
ERROR: Division by zero
ERROR: Addend too big
ERROR: Unknown error
10
```

Att. 21.3. Izņēmumu apstrāde

Piezīme. Izņēmumu apstrādes mehānisms nav domāts kļūdu noskaidrošanai programmā, bet gan noskaidroto kļūdu paziņojumu nosūtīšanai uz speciālu apstrādes vietu.

21.2. Nosaukumu telpas

Iepriekš tika apskatīti tādi mainīgo, funkciju u.c. programmas elementu redzamības apgabalu (*scope*) veidi kā lokālais, globālais, klases. Tomēr atsevišķos gadījumos ar to nepietiek, piemēram, ja divi dažādi programmētāji izveidojuši klases vai funkcijas ar vienādiem nosaukumiem, un tas viss jāapvieno kopējā projektā, var rasties problēmas, un bez viena programmētāja "piekāpšanās", nomainot nosaukumus uz citu neiztikt (turklāt arī tas ne vienmēr ir iespējams). Tāpēc ir izveidots speciāls redzamības tips ar nolūku sašķelt globālo redzamības apgabalu – nosaukumu telpas (*namespaces*). Arī standarta bibliotēkas vairs netiek liktas globālajā redzamības apgabalā, bet gan nosaukumu telpā *std*.



Att. 21.4. Dažādi redzamības apgabali

Nosaukumu telpu izmantošana sastāv no divām daļām:

- Programmas elementa piesaiste noteiktai nosaukumu telpai.
- Piekļūšana noteiktā nosaukumu telpā atrodošam programmas elementam.

Programmas elementu piesaiste nosaukumu telpai notiek, ievietojot to nosaukumu telpas blokā, kas sastāv no nosaukumu telpas galvas ar atslēgas vārdu ***namespace*** un tam sekojošā figūriekavu bloka, kurā ir ievietota programmas daļa, kura ir piesaistīta šai nosaukumu telpai.

```
namespace my_std
{
    void function1...
    class class1...
    ...
}
```

namespace_definition:
namespace *namespace_name* { *program_elements* }

Att. 21.5. Nosaukumu telpas definēšana (*namespace_definition*)

Programmas bloku piesaistei nosaukumu telpai ir šādas īpašības:

- Piesaisti var veikt dalīti – vairākos blokos, kas nozīmēs piesaisti vienai un tai pašai nosaukumu telpai, ja to nosaukumi sakritīs.
- Nosaukumu telpas ir atklātas – jebkurš var brīvi piesaistīt savas programmas elementus jebkādai nosaukumu telpai (t.sk. standarta nosaukumu telpai *std*) bez jebkādiem ierobežojumiem.

Piekļūšana noteiktai nosaukumu telpai piesaistītam elementam iespējama divos veidos:

- Tiešā veidā, izmantojot nosaukumu telpas vārdu un redzamības atrisināšanas operatoru :: tieši pie izmantojamā programmas elementa – pēc līdzības ar klases statisko elementu izsaukšanu.

```
my_std::function1...
```

- Pievienojot nosaukumu telpu aktuālajam redzamības gabalam, izmantojot atslēgas frāzi *using namespace* un nosaukumu telpas vārdu.

```
using namespace my_std;
...
function1...
```

Nosaukumu telpu izmantošanu demonstrē nākošais piemērs, kurā esošā programma saskaita divus skaitļus.

```
#include <iostream>
namespace my_std
{
    int cout (int a, int b)
    {
        return a + b;
    }
};

int main ()
{
    int a=7, b=5;
    {
        // līdz bloka beigām būs spēkā std;
        using namespace std;
        // cout pēc noklusēšanas ir izdrukas objekts
        cout << a << " + " << b << " = ";
    };
    // līdz funkcijas beigām būs spēkā my_std;
    using namespace my_std;
    // cout pēc noklusēšanas ir saskaitīšanas funkcija
    std::cout << cout (a, b) << std::endl;
    return 0;
}
```

```
7 + 5 = 12
```

Att. 21.6. Nosaukumu telpas

21.3. Norādes uz funkcijām

Valodā C++ mainīgie ir lietojami ne tikai, lai glabātu datus vai adreses uz datiem, bet tie var kalpot arī kā norādes uz funkcijām.

Vismaz divi svarīgi gadījumi, kad būtu noderīga funkciju izsaukšana caur norādi:

- Iespēja izsaukt vienu no vairākām funkcijām, izmantojot indeksu, ja funkcijas tiek glabātas masīvā. Neizmantojot norādes uz funkcijām, būtu jāizmanto vairāku līmeņu *if-then-else* priekšraksts, kas ne tikai palielinātu pirmkoda izmēru, bet arī palielinātu procesoram veicamo darbību skaitu.
- Iespēja izmantot universālas (konteineru) funkcijas, kas kā parametru spēj pieņemt citu funkciju, lai izsauktu pie sevis iekšienē.

Norādes uz funkciju deklarēšana.

Norādes (vai norādes tipa) definēšana funkcijai sintaktiski ir ļoti līdzīga funkcijas prototipam, vienīgā atšķirība ir tā, ka funkcijas (vai funkcijas tipa) vārds tiek likts iekavās un pirms funkcijas (vai funkcijas tipa) vārda ir norādes operators *.

```
int (*int_function_type) (int, int)
```

To var pēc tam izmantot 2 veidos:

- tiešā veidā:

```
int (*fp) (int, int);
```

- vai caur norādes tipu:

```
typedef int (*int_function_type) (int, int);
...
int_function_type fp;
```

Abos programmas fragmentos tiek deklarēta funkcijas norāde *fp*, kurai par piešķirt funkciju ar diviem *int* parametriem un atgriežamo tipu *int*.

Tieši iekavas ap funkcijas norādes vai funkcijas tipa nosaukumu ir tās, kas atšķir to no parasta datu tipa vai parasta mainīgā:

namespace_definition:
*return_type (* identifier) param_declaration_list;*

Att. 21.7. Norādes un funkciju deklarēšana (*function_pointer_declaration*)

Funkcijas piešķiršana norādei.

Funkciju piešķir norādei tieši tāpat kā vērtību piešķir mainīgajam, pirms funkcijas vārda var likt un var nelikt adreses operatoru. Tieši tāpat kā datu mainīgo, funkciju var nodot kā parametru citai funkcijai.

```
// ja add ir funkcija ar diviem int parametriem un atgriež int
fp = add;
```

assign_function_to_pointer:
pointer = &opt function_name

Att. 21.8. Funkcijas piešķiršana norādei (*assign_function_to_pointer*)

Funkcijas izsaukums caur norādi.

Funkcijas izsaukums caur norādi standarta variantā atšķiras no parasta funkciju izsaukuma ar to pašu, ar ko funkcijas norādes deklarēšana atšķiras no funkcijas prototipa – izsaucot funkcijas norādei liek priekšā norādes operatoru * un to visu liek iekavās, tomēr jaunais standarts ļauj funkciju izsaukt caur norādi tieši tā, kā izsauc funkciju – t.i., neliekot norādei priekšā norādes operatoru * un neliekot to iekavās.

```
// izsauc parastu funkciju:
res = add (7, 5);
// izsauc caur norādi:
res = (*fp) (7, 5);
// tomēr pa jaunam var arī tā:
res = fp (7, 5);
```

Nākošais piemērs parāda funkciju norāžu izmantošanu.

Tajā tiek definētas divas funkcijas *add* un *multiply*. Tām atbilstošais funkcijas tips ir *int (*int_function_type) (int, int);*. Bez tam ir izveidota universāla funkcija *compute*, kas "māk" gan saskaitīt, gan reizināt, jo tai kā parametru var padot funkciju.

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

int add (int a, int b)
{
    return a + b;
};

int multiply (int a, int b)
{
    return a * b;
};

typedef int (*int_function_type) (int, int);

int compute (int (*fp)(int, int), int a, int b)
{
    return (*fp) (a, b);
};

int main ()
{
    int a=7, b=5;
    int (*fp)(int, int);
    fp = add;
    // izsauc add caur norādi
    cout << (*fp) (a, b) << endl;
    int_function_type fp2 = add;
    // izsauc add caur norādi
    cout << fp2 (a, b) << endl;
    // izsauc multiply caur compute
    cout << compute (multiply, a, b) << endl;
    int (*funcs[2])(int, int);
    // iepriekšējo varēja arī tā: int_function_type funcs[2];
    funcs[0] = add;
    funcs[1] = multiply;
    // izsauc add un multiply caur norādēm, kas glabājas
                masīvā
    for (int i=0; i<2; i++)
    {
        cout << compute (funcs[i], a, b) << endl;
    };
    return 0;
}
```

```
12
12
35
12
35
```

Att. 21.9. Norādes uz funkcijām

21.4. Komandrindas argumenti

Programmas tekstā funkciju *main()* bieži vien raksta bez parametriem, tomēr tos var lietot, lai apstrādātu programmas izsaukšanā no komandrindas padotos argumentus.

Lai kontrolētu programmai padotās argumentu vērtības, funkcijai *main()* ir divi parametri:

- Parametrs #1. Vesels skaitlis. Programmai padoto argumentu skaits.
- Parametrs #2. Masīvs no simbolu virknēm. Programmai padoto argumentu vērtības.

Jāievēro viena īpatnība, ka masīva izmērs ir par vienu lielāks nekā argumentu skaits, jo elementā #0 glabājas programmu izsaucošā moduļa vārds, līdz ar to programmas pirmā argumenta vērtība vienmēr būs lielāka par 0.

Komandrindas argumentu apstrādi ar 3 dažādiem programmas izsaukšanas piemēriem demonstrē nākamais piemērs.

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

int main (int argc, char **argv)
{
    if (argc == 1)
    {
        cout << "\nUsage: ";
        cout << argv[0] << " ARG1, ARG2 [, ...]" << endl;
    }
    else if (argc == 2)
    {
        cout << "\nYou must specify at least 2 arguments\n";
        cout << "\nUsage: ";
        cout << argv[0] << " ARG1, ARG2 [, ...]" << endl;
    }
    else
    {
        cout << "The values of the arguments are:\n";
        for (int i=1; i<argc; i++)
        {
            cout << argv[i] << endl;
        }
    };
    return 0;
}
```

```
myfolder>cmdarg

Usage: msc2cmdline ARG1, ARG2 [, ...]
```

```
myfolder>cmdarg ONE

You must specify at least 2 arguments

Usage: msc2cmdline ARG1, ARG2 [, ...]
```

```
myfolder>cmdarg ONE TWO
The values of the arguments are:
ONE
TWO
```

Att. 21.10. Komandrindas argumenti (tiek pieņemts, ka programma saucas *cmdarg* un to izsauc no direktorijas *myfolder*)