2. C++ pamati

Nodaļas saturs:

2.1. 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) vienkārši ir datu veids.

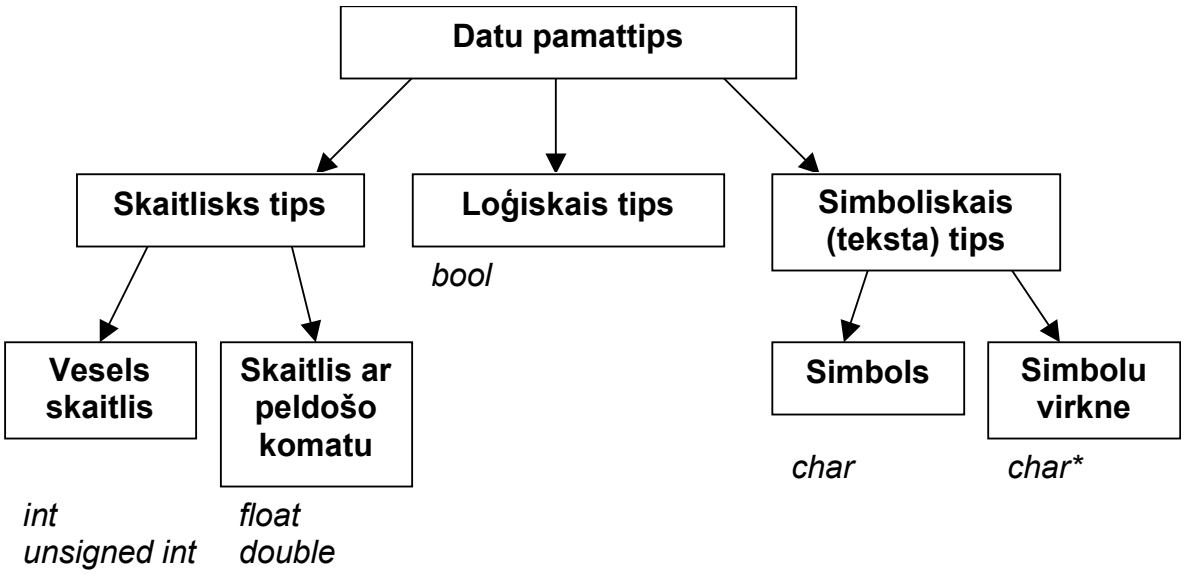
Valodā C++ nav iespējams strādāt ar vērtību, ja nav zināms (nav noteikts) tās datu tips. Tam ir praktiski iemesli, ko lielā mērā izskaidro nākošā datu tipa definīcija.

Datu tips (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, 16/5=3.1) 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 attēlā 2.1.



Attēls 2.1. Valodas C++ datu pamattipu klasifikācija un datu tipu piemēri

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 paradīti galvenie C++ pamattipi un dots īss to raksturojums.

Tabula 2.1. C++ valodas 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	$\sim 10^{-38}..10^{38}$ 7 zīmīgie cipari	`1.286 -1.2e+9`
double	8B	$\sim 10^{-308}..10^{308}$ 15 zīmīgie cipari	
long double	10B (vismaz)	$\sim 10^{-4932}..10^{4932}$ 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 2 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 piekārtots mainīgā datu tips;
- ja vērtība netiek glabāta mainīgajā (tā ar konstanta vērtība), tad tai piekārtots datu tips atkarībā no šīs vērtības pieraksta;
- ja vērtība atrodas dinamiskās atmiņas apgabalā, tās datu tips tiek noteikts, to uzrādot pie operatora *new*, rezervējot atmiņas apgabalu.

Tabula 2.2. Datu tipa noteikšana pēc konstantas vērtības

Konstantas vērtības 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" ""`

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 (sk. nodaļu 2.3).

2.2. Mainīgo deklarēšana

Pirms izmantot mainīgo programmā, tas ir jādeklarē.

Deklarācija (*declaration*) ir paziņojums, ka noteikts programmas elements (visbiežāk mainīgais) tiks izmantots programmā.

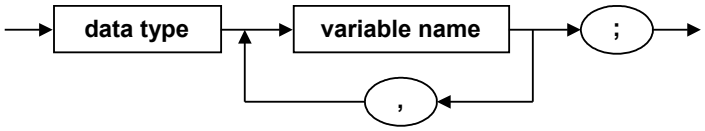
Mainīgā deklarācija (piemēram, pirmkods 1.1, rindas 6-8) ietver šādas informācijas uzrādīšanu par mainīgo:

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

Mainīgā deklarēšanu veic mainīgo deklarācijas konstrukcija. 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.

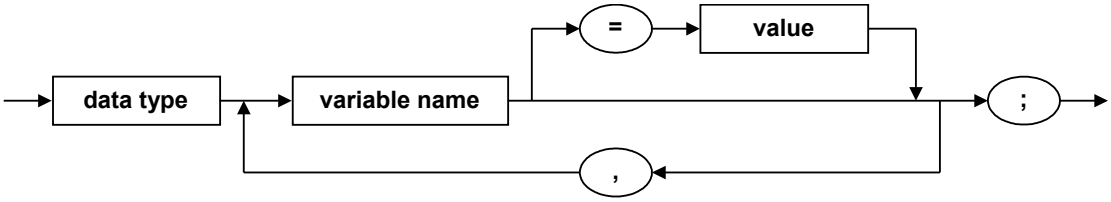
Vienkāršākais veids, kā deklarēt mainīgo, aprakstīts sintaktiskajā diagrammā 1.1 (atbilst pirmkoda 1.1 rindai 8).

Sintakse 1.1. *variable declaration* (mainīgo deklarācija)



Mainīgo deklarēšana ar inicializāciju parādīta sintaktiskajā diagrammā 1.2 (atbilst pirmkoda 1.1 rindām 6,7).

Sintakse 1.2. ***variable declaration with initialization*** **(mainīgo deklarācija ar inicializāciju)**

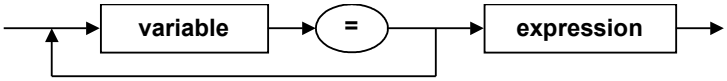


2.3. Piešķiršanas priekšraksts un tipu pārveidošana

Piešķiršana (*assignment*) ir vērtības ierakstīšana mainīgā.

Piešķiršanu veic **piešķiršanas priekšraksts** (*assignment statement*), kas ietver sevī piešķiršanas operatoru (vienlīdzības zīmi). Kreisajā pusē no tā obligāti jābūt mainīgajam. Labajā var būt jebkura konstrukcija, kas reprezentē vērtību (t.sk. konstanta vērtība vai mainīgais), ko vispārīgā gadījumā sauc par izteiksmi (*expression*) (sk. nodaļu 2.4).

Sintakse 1.3. ***assignment statement*** **(piešķiršanas priekšraksts)**



Atgriezeniskā saite diagramma nozīmē, ka piešķiršanas operatori ir izmantojami viens pēc otra kaskādes veidā, piemēram, (b=c; a=b) vietā var rakstīt a=b=c.

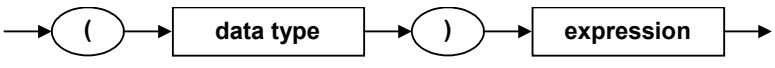
Parasti mainīgajam ir tāds pats tips kā tam piešķiramajai vērtībai. Tomēr dažreiz mainīgā tips atšķiras no tam piešķiramās vērtības tipa (piemēram, pirmkods 1.1, rinda 12), tādā gadījumā runā par tipu pārveidošanu.

Tipu pārveidošana (*type casting*) ir process ir datu pārveidošana 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 tipu 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 tipu pārveidošanas rezultātā iespējama potenciāla informācijas zaudēšana, kompilators par to brīdina, pieprasot tipu pārveidošanas operatora pielietošanu (sintakse 1.4; pirmkods 1.1, rinda 14). 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*).

Sintakse 1.4. ***type casting operator*** **(tipa pārveidošanas operators)**



Piešķiršana nav vienīgais gadījums, kas var izsaukt tipu pārveidošanu, to var nodrošināt jebkurš programmas konteksts, piemēram, aritmētisku operāciju pielietošana.

Pirmkods 1.1. Piešķiršana un tipa pārveidošana (*cpp1assign.cpp*)

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

```
03
04 int main ()
05 {
06     int i, j=0, k;
07     char c='A';
08     char d;
09     i = 70;
10     k = i + 1;
11     cout << c << endl;
12     d = i;
13     cout << i << " " << d << endl;
14     cout << k << " " << (char)k << endl;
15     return 0;
16 }
```

Programmas darbības piemērs:

```
A
70 F
71 G
```

Komentāri pie pirmkoda piemēra 1.1.

- Rindās 6-8 tiek deklarēti 5 mainīgie, 2 no tiem tiek inicializēti pie deklarēšanas.
- Rindā 12 mainīgajam d (ar tipu *char*) tiek piešķirta mainīga i (ar tipu *int*) vērtība 70.
- Rindā 13 d tiek izdrukāts kā 'F', kas norāda, ka burta 'F' kods ir 70.
- Rindā 14 mainīgā k vērtība (71) tiek pārveidota uz tipu *char* bez papildus mainīgā izmantošanas. "(char)k" tiek izdrukāts kā 'G', kas norāda, ka burta 'G' kods ir 71.

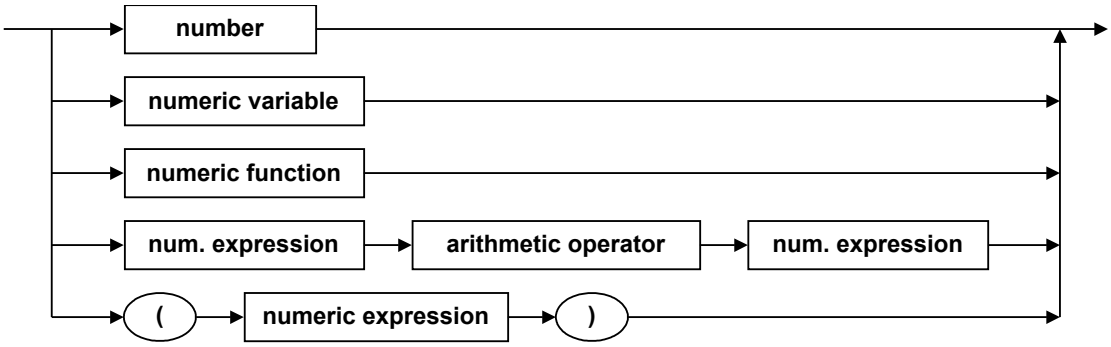
2.4. Aritmētiskas izteiksmes un piešķiršana ar izrēķināšanu

2.4.1. Aritmētiskas izteiksmes

Aritmētiska izteiksme (*numeric expression*) ir konstrukcija, kas sastāv no skaitliskām operācijām un skaitliskām vērtībām, un kuru pēc šo operāciju izpildes reprezentē skaitliska vērtība.

Triviālie izteiksmju piemēri ir skaitliskas konstantes, mainīgie un funkcijas, bet vispārīgi aritmētiskas izteiksmes tiek definētas rekursīvi, un to sintakse ir aprakstāma šādi:

Sintakse 1.5. *numeric expression* (aritmētiska izteiksme)



Sintakse 1.6. *arithmetic operator* (aritmētiska operācija)

<arithmetic operator> ::= + | - | * | / | %

Augstāk dotais apraksts attiecas uz aritmētiskām izteiksmēm kopumā, tomēr valodā C++ aritmētiskas izteiksmes jēdziens ir plašāks nekā citās valodās, piemēram, pie aritmētiskām operācijām var pieskaitīt bitu līmeņa (*bitwise*) operācijas, bez tam arī loģiskās operācijas un izteiksmes, kas tiks apskatītas nedaudz vēlāk, ir savā ziņā uzskatāmas par skaitliskām.

Valodā C++ specifiska ir operācija '/', kas atkarībā no konteksta nozīmē vai nu parasto vai veselo skaitļu dalīšanu.

Pirmkoda piemērā 1.2 parādīta šādas aritmētiskas izteiksmes izrēķināšana:

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

Pirmkods 1.2. Aritmētiska izteiksme (*cpp2expr.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 int main ()
05 {
06     double x, y, z;
07     cout << "Input 2 numeric values:" << endl;
08     cin >> x >> y;
09     z = (sin(x) + 1) / y + y / (x + 1.5);
10     cout << "Result is " << z << endl;
11     return 0;
12 }
```

Programmas darbības piemērs:

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

2.4.2. Skaitliskas operācijas un funkcijas

Galvenās C++ skaitliskās operācijas ir +,-,*,/,% (sk. sintaksi 1.6).

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 operācijām pieejami arī atbilstošie speciālie piešķiršanas operatori (sk. sadaļu 2.4.3). Bitu līmeņa operāciju darbības princips ir labi saprotams, ja iztēlojas skaitļa attēlojumu atmiņā pa bitam. Bitu līmeņa operācijas šajā materiālā netiks aprakstītas.

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 exp);
```

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

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 (sk. sadaļu 2.3), piemēram, *a = (int)ceil (1.85)*.

floor

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

Funkcija *ceil()* 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.

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

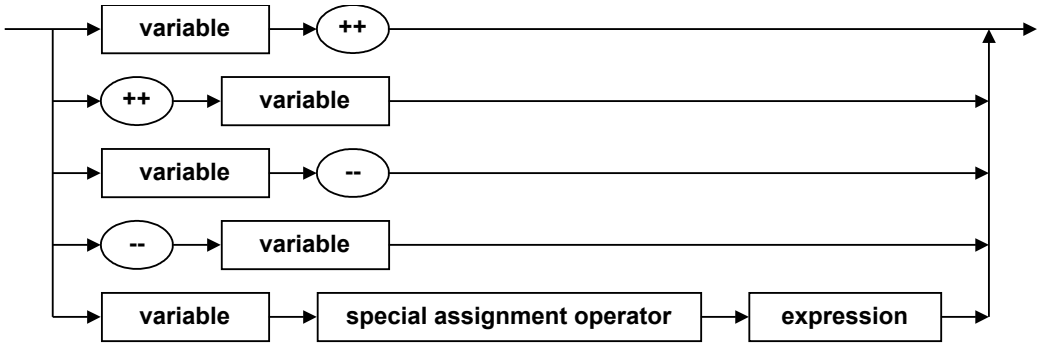
```
float fabs (float arg);
```

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

2.4.3. Piešķiršana 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++*. Arī pašas valodas C++ vārds cēlies no viena šāda operatora.

Sintakse 1.7. ***assignment statement special*** **(speciālais piešķiršanas priekšraksts)**



Sintakse 1.8. ***special assignment operator*** **(speciālais piešķiršanas operators)**

<special assignment operator> ::= += | -= | *= | /= | %= | <<= | >>= | ^= | &= | '|='

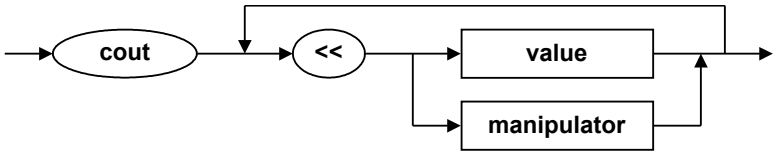
2.5. Standarta ievade un izvade

Standarta ievade (no klaviatūras) un izvade (uz ekrāna) notiek, izmantojot 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 atmiņa. Tādējādi formatēšana (*formatting*) nozīmē datu pārveidošana starp teksta formātu (klaviatūra vai ekrāns) un datu glabāšanas formātu atmiņā.

2.5.1. Formatēta izvade

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

Sintakse 1.9. ***formatted output*** **(formatēta izvade)**



"Vērtība" (*value*) var būt jebkura konstrukcija, kas reprezentē kādu vērtību – aritmētiska izteiksme, teksts utt. Manipulators (*manipulator*) ir līdzeklis izvades formatēšanai.

Formatēšana var tikt veikta ar divu veidu līdzekļiem – manipulatoriem (piemēram, *setw* un *fixed* pirmkoda 1.3 rindā 18) vai objekta *cout* fukcijām (piemēram, *width* un *precision* pirmkoda 1.3 rindās 12,13). Lielākā daļa formatēšanas operāciju ir veicamas abos variantos un parasti *cout* funkcija un attiecīgas manipulators ir no funkcionālā viedokļa identiski (piemēram, *width* un *setw* pirmkoda 1.3 rindās 12,18). Daži no manipulatoriem atrodas bibliotēkā <*iostream*>, bet daži atrodas <*iomanip*>. Lielākā daļa formatēšanas operāciju darbojas no uzstādīšanas brīža līdz programmas beigām vai operācijas atsaukšanai (piemēram, precizitātes uzstādīšana ar *precision*/*setprecision*), tomēr ir operācijas, kas attiecas tikai uz nākošās vērtības izvadi, pēc kā automātiski tiek atceltas (piemēram, *width*/*setw*).

Svarīgākās operācijas formatētajā izvadē (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).

- *width/setw* – uzstāda izvades apgabala platumu simbolos (darbojas tikai uz nākošo izvadāmo vērtību, pēc noklusēšanas izslēgts, t.i., izmanto tik daudz vietas, cik nepieciešams).
- *setf(ios::fixed)/fixed* – uzstādītās precizitātes noteikšana, ka tā attiecas uz cipariem aiz komata (nevis zīmīgajiem).
- *fill/setfill* – uzstāda aizpildīšanas simbolu (ir nozīme gadījumā, ja ir uzstādīts arī platums, pēc noklusēšanas – tukšums).
- *scientific* – manipulators, kas nosaka zinātnisko skaitļa izvades formātu.
- *setf(ios::scientific)* – komanda, kas nosaka, ka uzstādītā precizitāte attiecas uz zīmīgajiem cipariem (nevis cipariem aiz komata).
- *left,right* – manipulatori pielīdzināšanai pie kreisās vai labās malas (ir nozīme gadījumā, ja ir uzstādīts platums).
- *endl* – jaunas rindiņas izdruka, identiska '\n' izmantošanai.

Pirmkods 1.3. Formatēta izvade (*cpp3out.cpp*)

```
01 #include <iostream>
02 #include <iomanip>
03 using namespace std;
04
05 int main ()
06 {
07     int i = 5;
08     double d = 271.84753;
09     char c = 'A';
10     cout << 5 << endl;
11     cout << d << " " << c << endl;
12     cout.width (11);
13     cout.precision (5);
14     cout.setf (ios::fixed);
15     cout << d << '*' << endl;
16     cout << scientific << d << '*' << endl;
17     cout.fill ('_');
18     cout << setw(12) << setprecision(4) << fixed << d << '*' <<
                endl;
19     cout.setf (ios::scientific);
20     cout << left << setfill('^') << setw(12) << d << '*' << endl;
21     return 0;
22 }
```

Programmas darbības piemērs:

```
5
271.848 A
  271.84753*
2.71848e+002*
____271.8475*
271.8^^^^^^^*
```

Komentāri pie pirmkoda piemēra 1.3.

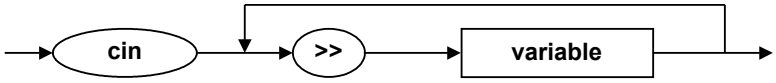
- ‘5’ izdruku nodrošina rinda 10.
- ‘271.848 A’ izdruku nodrošina rinda 11 (pēc noklusēšanas skaitli ar peldošo komatu izdrukā ar 6 zīmīgajiem cipariem).

- ‘ 271.84753*’ – rindas 12-15, precizitāte 5 cipari aiz komata, ko nodrošina *precision* komplektā ar *fixed*, izvades platums 11, pielīdzināšana pie labās malas (pēc noklusēšanas)
- ‘2.71848e+002*’ – rinda 16, *scientific* nosaka zinātnisko formatēšanu, iepriekš uzstādītais platums vairs nedarbojas.
- ‘____271.8475*’ – rindas 17-18, izdrukas platums 12, precizitāte 4 cipari aiz komata, ko nodrošina *setprecision* komplektā ar *fixed*, bez tam aizpildīšanas simbols ir ‘_’, ko nosaka *fill*.
- ‘271.8^^^^^^^*’ – rindas 19-20, izdrukas platums 12, aizpildīšanas simbols ‘^’, izdrukas precizitāte 4 zīmīgie cipari, ko nodrošina *ios::scientific* komplektā ar *setprecision* (rindā 18), bez tam noteikts kreisais pielīdzinājums (*left*).

2.5.2. Formatēta ievade

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 1.10. *formatted intput* (formatēta ievade)



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*).

Ja nolasīšana bijusi neveiksmīga, objekts *cin* tiek nobloķēts, un, ja to neatbloķē, visas turpmākās darbības ar to tiek ignorētas. Lai turpinātu darbu pēc neveiksmīgas ievades, ir jāveic divas operācijas (sk. pirmkodu 1.5):

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

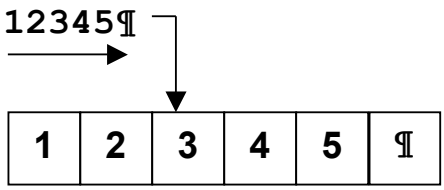
Lai labāk saprastu ievades un izvades principus, ir jāzina, ka tās nenotiek tiešā veidā, bet gan, izmantojot starpbuferi informācijas uzkrāšanai.

Ievades process tādējādi notiek 2 posmos (attēls 2.2):

- 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.

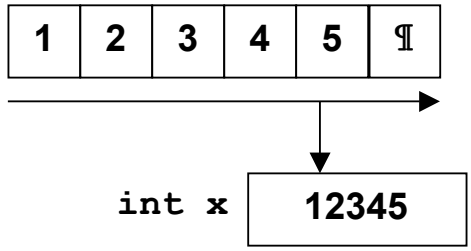
```
int x;
cin >> x;
```

1. solis.
Pēc ENTER nospiešanas informācija tiek ierakstīta buferī.



---

2. solis.
No bufera tiek mēģināts nolasīt noteikta tipa vērtību un ierakstīt mainīgajā.



Attēls 2.2. Vienkāršota ievades darbības shēma

Pirmkods 1.4. Formatēta ievade (*cpp4in.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 int main ()
05 {
06     int i;
07     cin >> i;
08     if (cin.good()) cout << i << endl;
09     else cout << "Input error" << endl;
10     return 0;
11 }
```

Programmas darbības piemērs #1:

```
abc
Input error
```

Programmas darbības piemērs #2:

```
123
123
```

Pirmkods 1.5. Formatēta ievade ar labošanas pieprasījumu (*cpp5in.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 int main ()
05 {
06     int i;
07     cout << "Input integer value: ";
08     cin >> i;
09     while (!cin.good())
10     {
11         cin.clear ();
12         cin.ignore (256, '\n');
13         cout << "Try again: ";
14         cin >> i;
15     };
```

```
16      cout << "Success: " << i << endl;
17      return 0;
18  }
```

Programmas darbības piemērs:

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

2.6. Daži specifiski operatori

2.6.1. Operators (,) (komats)

Operators **komats** (,) līdzīgi semikolam (;) nodrošina priekšrakstu virknes pierakstu, bet beigās atgriež pēdējā priekšraksta atgriezto vērtību. Pirmkoda piemēra 1.6 rindā 7: mainīgajam *c* tiek piešķirta vērtība *a*b*, pirms tam gan *a*, gan *b* tika palielināti par 1.

Ar komatu atdalītus priekšrakstus izmanto *for* cikla konstrukcijas galvā, un komata operatoru šeit izmanto tādēļ, ka semikols tiek izmantots komponenšu atdalīšanai *for* cikla galvā, tādēļ nav izmantojams elementu atdalīšanai vienas komponentes ietvaros.

Pirmkods 1.6. Specifiski operatori (*cpp6comma.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 int main ()
05 {
06      int a=3, b=5, c, d;
07      c = (a++, b++, a*b);
08      cout << a << " " << b << " " << c << endl;
09      a = b = 1;
10      c = ++a;
11      d = b++;
12      cout << a << " " << b << " " << c << " " << d << endl;
13      return 0;
14 }
```

Programmas darbības piemērs:

```
4 6 24
2 2 2 1
```

2.6.2. Operatoru ++ un -- prefiksais un postfiksais variants

Operatori ++ un – attiecīgi palielina vai samazina mainīgā vērtību par vienu vienību. Taču abiem šiem operatoriem eksistē gan prefiksais, gan postfiksais variants. Abi varianti veic mainīgā vērtības izmaiņu, bet atšķirība ir vērtībā, ko atgriež – prefiksais variants atgriež jauno vērtību, bet postfiksais iepriekšējo (sk. rindas 9-12 pirmkoda piemērā 1.6).

Parasti šie operatori tiek izmantoti tikai vērtības izmaiņai, un ir vienalga, kuru variantu lietot.