9. Objektorientētā programmēšana

Nodaļas saturs:

- 9.1. Klases un objekti
- 9.2 Konstruktors un destruktors
- 9.3 Klases statiskie elementi
- 9.4 Objekta norāde uz sevi 'this' un klases deklarēšana
- 9.5 Mantošana
  - 9.5.1 Mantošanas pamatprincipi
  - 9.5.2 Virtuālās funkcijas
  - 9.5.3 Daudzkāršā mantošana
- 9.6 Citi objektorientētās programmēšanas mehānismi
  - 9.6.1 Draugu funkcijas un klases
  - 9.6.2 Operatoru pārslogošana

9.1. Klases un objekti

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 turpretī 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** (*methodes*) 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 (līdzīgi kā konstrukcijā *struct*) un iekšējo funkciju aprakstu, kā arī pieejas tiesību definēšanu mainīgajiem un funkcijām.

Sintaktiski klase tiek definēta pēc līdzības ar konstrukciju *struct*.

Uz klases metodēm, tāpat kā uz parastajām funkcijām attiecas **pārslogošana** (*overloading*), t.i., iespēja izmantot vienu nosaukumu vairāku metožu nosaukšanai.

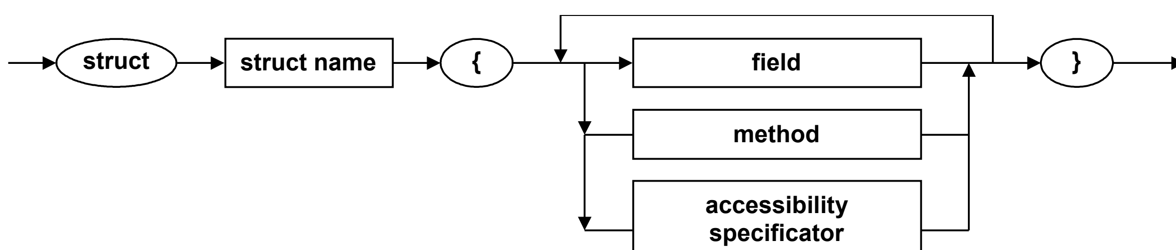
Līdz ar jēdzienu klase, blakus jau zināmajiem – globālo un lokālo redzamības apgabalu (*scope*) parādās jauns redzamības apgabals – klases redzamības apgabals. Atrašanās klases redzamības gabalā nozīmē to, ka iekšējais mainīgais vai funkcija ir redzami tikai dotā objekta ietvaros.

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** (*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.

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 **inkapsulāciju** (*encapsulation*), kas ir viens no galvenajiem objektorientētas programmēšanas principiem.

Sintakse 9.1. *class definition* (klases definēšana)



Objekti programmā var parādīties divos veidos, kas atšķiras pēc sintakses un atmiņas izmantošanas (tāpat kā struktūras *struct* objektiem):

- tiešā veidā (izmantojot automātisko atmiņu),
- dinamiskā veidā.

Objekta izmantošana tiešā veidā.

Mainīgā tips ir klase (pirmkoda piemērs 9.1, rinda 18) un piekļuve pie objekta elementiem (laukiem un metodēm) notiek, izmantojot **punkta notāciju** (rinda 19). Objekts tiek automātiski izveidots pie mainīgā deklarēšanas un automātiski beidz pastāvēt programmas bloka beigās.

Objekta izmantošana dinamiskā veidā.

Mainīgā tips ir norāde uz klasi (pirmkoda piemērs 9.1, rinda 20) un piekļuve pie objekta elementiem notiek, izmantojot **bultiņas notāciju** (rinda 21). Objekts tiek izveidots, izmantojot operatoru *new* (rinda 20) un likvidēts ar *delete* (rinda 22).

Pirmkoda piemērs 9.1 parāda objektu izmantošanu gan tiešā, gan dinamiskā veidā. Piemēru ilustrē attēls 9.1.

Ja runā par piekļuvi pie objekta elementiem (ar punkta vai bultiņas notāciju), tad pareizāk būtu teikt, ka tā ir piekļuve tiešā veidā vai caur norādi, jo objekta izveidošana tiešā vai dinamiskā veidā neierobežo iespējas piekļūt tā elementiem ar dažādu sintaksi (protams, izmantojot citus mainīgos vai funkcijas).

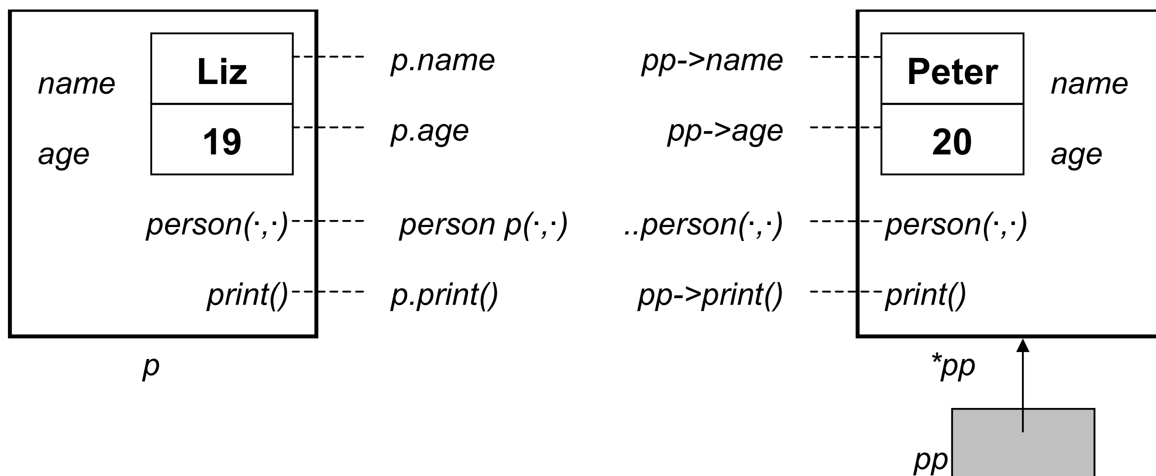
Pirmkoda piemēra 9.1 rindās 9 un 26-30 ir definēta, bet rindās 18 un 20 izmantota speciāla klases metode – **konstruktors**, kas paredzēta objektu veidošanai. Konstruktori tiks apskatīti nākamajās nodaļās.

Pirmkods 9.1. Konstrukcija *class* (*cls1class.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 class person
05 {
06      char name[20];
07      int age;
08 public:
09      person (const char*, int);
10      void print ()
11      {
12          cout << name << " " << age << endl;
13      }
14 };
15
16 int main ()
17 {
18      person p ("Liz", 19);
19      p.print ();
20      person *pp = new person ("Peter", 20);
21      pp->print ();
22      delete pp;
23      return 0;
24 };
25
26 person::person (const char *n, int a)
27 {
28      strcpy (name, n);
29      age = a;
30 }
```

Programmas darbības piemērs:

```
Liz 19
Peter 20
```



Attēls 9.1. Klases objekta izmantošana (atbilst pirmkoda piemēram 9.1)

Klases definēšana dalīti.

Obligāta sastāvdaļa klases definēšanai ir **hederis** jeb galvene (*header*) (pirmkoda piemērs 9.1, rindas 4-14). Klases hederī tiek definēti visi lauki, tomēr klases metodes galvā var tikt definētas gan pilnībā (pirmkoda piemērs 9.1, rindas 10-13), gan var tikt norādīti tikai to prototipi (pirmkoda piemērs 9.1, rinda 9), lai pēc tam metodi realizētu citur. Ja klases metodi definē ārpus klases hedera, tad pirms attiecīgās metodes jāuzrāda, uz kuru klasi tas attiecas ("person::", pirmkoda piemērs 9.1, rinda 26), kas ir visai loģiska prasība, jo dažādām klasēm drīkst būt vienāda nosaukuma metodes un cita starpā arī tāpēc, ka klases metožu realizācija var būt izkaisīta vairākos failos. Operators :: ir redzamības atrisināšanas operators, kas tiek izmantots arī citos kontekstos.

Klases dalīšana vairākos failos.

Klasi var dalīt vairākos failos pēc līdzīga principa kā tas darāms funkcijām (pirmkods 9.2). Klases galvu, katras atsevišķas metodes ārējo realizāciju un izsaukumus var likt atsevišķos failos. Klases hederis tiek ievietots hedera failā, kuru pēc tam tieši vai netieši iekļauj visi moduļi, kuros klase tiek definēta vai izmantota (pirmkods 9.2, rindas 19, 26).

Pirmkods 9.2. Klases dalīšana vairākos failos

class2class.h:

```
01 #ifndef CLS2CLASS
02 #define CLS2CLASS
03 #include <iostream>
04 using namespace std;
05
06 class person
07 {
08     char name[20];
09     int age;
10 public:
11     person (const char*, int);
12     void print ()
13     {
14         cout << name << " " << age << endl;
15     }
16 };
17
18 #endif
```

class2class.cpp:

```
19 #include "cls2class.h"
20
21 person::person (const char *n, int a)
22 {
23     strcpy (name, n);
24     age = a;
25 }
```

class2classmain.cpp:

```
26 #include "cls2class.h"
27
28 int main ()
29 {
```

```
30      person p ("Liz", 19);
31      p.print ();
32      person *pp = new person ("Peter", 20);
33      pp->print ();
34      delete pp;
35      return 0;
36 }
```

Programmas darbības piemērs:

```
Liz 19
Peter 20
```

9.2. Konstruktors un destruktors

Konstruktors (*constructor*) ir specifiska klases iekšējā funkcija, kas automātiski (netieši) tiek izsaukta uzreiz pēc objekta izveidošanas.

Valodā C++ konstruktors tiek saukts klases vārdā (pirmkods 9.3, rinda 10). Klasē var būt vairāki konstruktori, un tie, tāpat kā citas funkcijas ar vienādu nosaukumu, atšķiras pēc parametru virknes.

Konstruktors 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 (pirmkods 9.3, rinda 23). Ja konstruktoram nav parametru, tad tukšās iekavas drīkst nelikt (pirmkods 9.4, rinda 29).

Ja objekts ir dinamisks, tad konstruktors tiek netiešā veidā izsaukts, izsaucot operatoru *new* (pirmkods 9.1, rinda 20).

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 '~' (pirmkods 9.3, rinda 15). Destruktoram nav parametru, tādējādi klasei var būt tikai viens destruktors.

Ja konstruktoram ir jēga 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) (pirmkods 9.3, klases *person* lauks *name*).

Destruktora izsaukšana atšķiras atkarībā no objekta izmantošanas veida.

Ja objekts tiek izmantots tiešā veidā, tad destruktors tiek izsaukts tā bloka beigās, kad objekts deklarēts (pirmkods 9.1, objekts *p* tiek likvidēts rindā 24).

Ja objekts ir dinamisks, tad destruktors tiek netiešā veidā izsaukts ar operatoru *delete* (pirmkods 9.1, objekts *pp* tiek likvidēts rindā 22).

Pirmkoda piemērā 9.3 klases *person* dinamiskās atmiņas vadību nodrošina metodes *init* un *destroy*, kuras tiek izsauktas arī no konstruktora un destruktora. Šajā piemērā tiek izmantots tikai viens objekts, kurš rindā 25 nomaina savu saturu.

Pirmkods 9.3. Konstruktors un destruktors (*cls3consdest.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 class person
05 {
06 private:
07     char *name;
08     int age;
09 public:
10     person (const char *n, int a)
11     {
12         name = NULL;
13         init (n, a);
14     };
15     ~person () { destroy (); };
16     void init (const char*, int);
17     void destroy ();
18     void print () { cout << name << " " << age << endl; }
19 };
20
21 int main ()
22 {
23     person p ("Liz", 19);
24     p.print ();
25     p.init ("Peter", 20);
26     p.print ();
27     return 0;
28 };
29
30 void person::destroy ()
31 {
32     cout << "Deleting: " << name << endl;
33     delete[] name;
34 };
35
36 void person::init (const char *n, int a)
37 {
38     if (name != NULL) destroy ();
39     name = new char[strlen(n)+1];
40     strcpy (name, n);
41     age = a;
42 }
```

Programmas darbības piemērs:

```
Liz 19
Deleting: Liz
Peter 20
Deleting: Peter
```

Ir vairāku veidu konstruktori, turklāt konstruktoru izmantošanā ir pielietojamas papildus sintakses iespējas.

Noklusētais konstruktors.

Katrā klasē ir vismaz viens konstruktors. Ja konstruktors nav definēts, tad kompilators automātiski izveido **noklusēto konstruktotu** (*default constructor*) bez parametriem (tādu kā pirmkoda piemērā 9.4 rindā 19).

Ja klasē ir definēts kaut viens konstruktors, tad noklusētais konstruktors automātiski netiek veidots un, ja ir nepieciešamība pēc konstruktora bez parametriem (pirmkods 9.4, rinda 29), tad konstruktors bez parametriem obligāti jādefinē (rinda 19).

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 (pirmkods 9.4, rindas 13-17). Kopijas konstruktoru bez parastā izmantošanas veida, izsauc vēl šādos gadījumos:

- inicializējot objektu ar citu objektu pie deklarēšanas (pirmkods 9.4, rinda 27 (bet nav rindā 30!)),
- nododot objektu (bet nevis tā referenci) kādai funkcijai kā parametru – tādā gadījumā funkcijas vajadzībām tiek veidots jauns lokāls objekts, izsaucot kopijas konstruktoru.

Citi konstruktori ar vienu parametru.

Arī citi konstruktori ar vienu parametru darbojas tāpat kā kopijas konstruktors – tos izsauc pie inicializēšanas ar piešķiršanu (pirmkods 9.4, rinda 32) vai caur parametru, kuras tips ir dotā klase.

Objekta lauku inicializēšanas speciālā sintakse konstruktorā.

Blakus tradicionālajai lauku inicializēšanas iespējai (pirmkods 9.4, rindas 10,11) konstruktorā ir pieejama arī speciālā sintakse, kur lauki tiek inicializēti pirms konstruktora ķermeņa – konstruktora galvā (pirmkods 9.4, rinda 18). Šī sintakse ir alternatīva parastajai, un nav jāizmanto obligāti.

Pirmkods 9.4. Konstruktoru veidi (*cls4construct.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 class timesimple
05 {
06     int hours, minutes;
07 public:
08     timesimple (int h, int m)
09     {
10         hours = h;
11         minutes = m;
12     };
13     timesimple (const timesimple &t)
14     {
15         hours = t.hours + 1;
16         minutes = t.minutes + 1;
17     };
18     timesimple (int h): hours(h), minutes(0) {};
19     timesimple () {};
20     void print () const;
21 };
```

```
22
23 int main ()
24 {
25     timesimple t (8, 20);
26     t.print ();
27     timesimple u = t;
28     u.print ();
29     timesimple v;
30     v = t;
31     v.print ();
32     timesimple w = 23;
33     w.print ();
34     return 0;
35 };
36
37 void timesimple::print () const
38 {
39     cout << hours << ":" << minutes << endl;
40 }
```

Programmas darbības piemērs:

```
8:20
9:21
8:20
23:0
```

Komentāri pie pirmkoda piemēra 9.4.

- Rindā 27 tiek izsaukts kopijas konstruktors, par ko liecina tas, ka rindā 28 izdrukājas '9:21', nevis '8:20'.
- Rindā 29 tiek izsaukts konstruktors bez parametriem (rinda 19).
- Rindā 30 netiek izsaukts kopijas konstruktors, bet tiek veikta parastā piešķiršana.
- Rindā 32 tiek izsaukts konstruktors ar vienu parametru (rinda 18).
- Modifikators *const* funkcijas *print* galvas beigās (rindas 20,37) sintaktiskā līmenī garantē, ka šī funkcija ne tieši, ne netieši neizmainīs pašu objektu. Ja šādā funkcijā tiek mēģināts tiešā veidā izmainīt kādu lauku vai izsaukt citu iekšējo funkciju, kas nav atzīmētā ar *const*, iestājas sintakses kļūda.

9.3. 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 (pirmkods 9.5, rinda 13).

Klases statisko elementu nosaka modifikatora ***static*** pievienošana pirms elementa klases galvā (pirmkods 9.5, rindas 6,8).

Statiskā lauka definēšana.

Papildus tam, ka lauks klases galvā tiek deklarēts kā statisks, ārpus klases galvas tas vēl papildus ir jādefinē (pirmkods 9.5, rinda 27).

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.

Pirmkods 9.5. Klases statiskie elementi (*cls5static.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 class months
05 {
06     static char monthnames[][10];
07 public:
08     static char* get_month_name (int num);
09 };
10
11 int main ()
12 {
13     cout << months::get_month_name (1);
14     return 0;
15 };
16
17 char months::monthnames[][10] = {"january", "february", "march"};
18
19 char* months::get_month_name (int num)
20 {
21     return monthnames[num];
22 }
```

Programmas darbības piemērs:

```
february
```

9.4. Objekta norāde uz sevi ‘this’ un klases deklarēšana

Norāde *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.

Pirmkoda piemērā 9.6 parādīts, kā klases *person* izdrukāšana tiek nodota divām ārējām funkcijām ar dažādiem parametriem – attiecīgi references un norādes.

Pirmkods 9.6. Objekta norāde uz sevi ‘this’ (*cls6this.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
```

```
04 class person;
05
06 void print_1 (const person&);
07 void print_2 (const person*);
08
09 struct person
10 {
11     string name;
12     int age;
13     void print1 () const { print_1 (*this); };
14     void print2 () const { print_2 (this); };
15 };
16
17 int main ()
18 {
19     person p;
20     p.name = "Liz";
21     p.age = 19;
22     p.print1 ();
23     p.print2 ();
24     return 0;
25 };
26
27 void print_1 (const person &p)
28 {
29     cout << p.name << " " << p.age << endl;
30 };
31
32 void print_2 (const person *p)
33 {
34     cout << p->name << " " << p->age << endl;
35 }
```

Programmas darbības piemērs:

```
Liz 19
Liz 19
```

Klases deklarēšana.

Klases deklarēšana ir paziņojums, ka klase ar doto vārdu tiks izmantota programmā (pirmkods 9.6, rinda 4). Parasti klasi atsevišķi nedeklarē, bet deklarēšanu veic, definējot klases hederi. Tomēr atsevišķos gadījumos ir nepieciešams veidot struktūras vai funkcijas, kurās ietilpst dotā klase, vēl pirms klases hedera definēšanas (pirmkods 9.6, rindas 6,7). Bez klases deklarēšanas nevar iztikt gadījumos, kad klase un kāds ārējs elements (piemēram, cita klase vai funkcija) izmanto viens otru, kā tas ir arī pirmkoda piemērā 9.6 – klase *person* izmanto funkcijas *print_1* un *print_2*, bet šīs funkcijas savukārt izmanto šo klasi sava parametra tipa veidošanai.

9.5. Mantošana

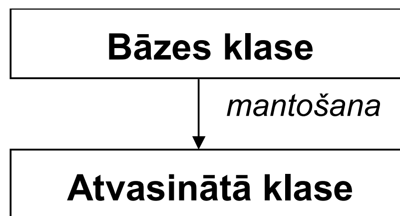
9.5.1. Mantošanas pamatprincipi

Mantošana (*inheritance*) ir objektorientētās programmēšanas mehānisms, kad klase pārņem

(manto) elementus (laukus un metodes) no jau eksistējošas klases vai pat vairākām klasēm.

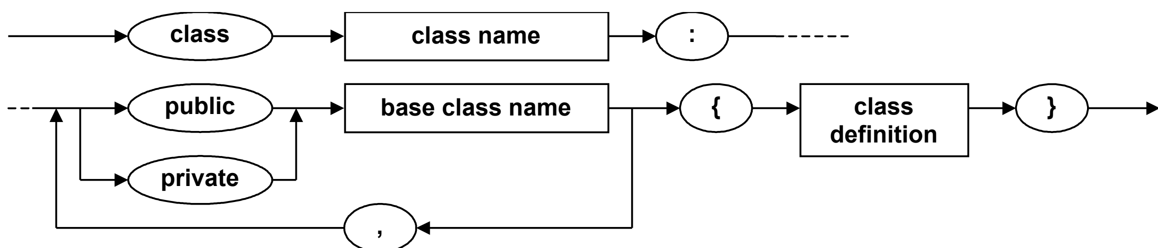
Mantošana (sk. sintaksi 9.2 un attēlu 9.2) ir vēl viens mehānisms pirmkoda vairākkārtējai izmantošanai, kas papildina koda strukturēšanas iespējas, kā arī nodrošina ērtāku uzturēšanu.

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.

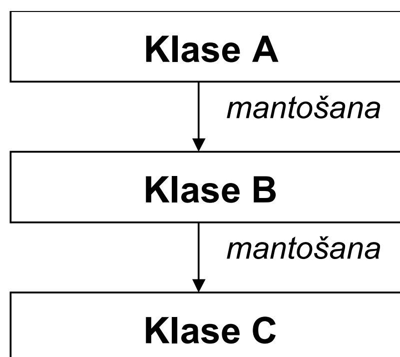


Attēls 9.2. Mantošana (atbilst pirmkoda piemēram 9.7)

Sintakse 9.2. *inheritance* (mantošana)



Mantošanas process var būt kaskādes veida, līdz ar to viena un tā pati klase var būt gan atvasinātā klase, gan bāzes klase citām klasēm.



Attēls 9.3. Kaskādes veida mantošana

Mantošanas mehānisma izmantošana izsauc nepieciešamību pēc vairākām papildus konstrukcijām un mehānismiem objektorientētajā programmēšanā:

- modifikators **protected** (pirmkods 9.7, rinda 6; jau aprakstīts nodaļā 9) paplašina **private** redzamību arī uz mantotajām klasēm,
- konstruktoru kaskādes veida izsaukšana,
- destruktoru kaskādes veida izsaukšana,
- atklātā un slēptā mantošana,
- virtuālās funkcijas.

Konstruktoru kaskādes veida izsaukšana

Pirms izsaukt atvasinātas klases konstruktoru, obligāti jānodrošina arī bāzes klases konstruktora izsaukšana. Tādējādi tiek nodrošināts, ka, izveidojot objektu, tiek izsaukti arī **visu bāzes klašu konstruktori mantošanas secībā**, un tikai tad pašas klases konstruktors.

Tā kā klasē var būt vairāki konstruktori, tad, definējot atvasinātas klases konstruktoru, ar īpašu sintaksi (sk. sintakses diagrammu 9.3) tiek norādīts, kuru no bāzes klases konstruktoriem pirms tam izsaukt (pirmkods 9.7, rinda 49). Ja bāzes klases konstruktors netiek uzrādīts, tad bāzes klasē jāeksistē konstruktoram bez parametriem, kas tādā gadījumā tiek izsaukts. Atvasinātas klases konstruktora definēšana lokālā veidā atgādina pašas klases mantošanu (salīdzināt ar sintakses diagrammu 9.2).

Destruktoru kaskādes veida izsaukšana.

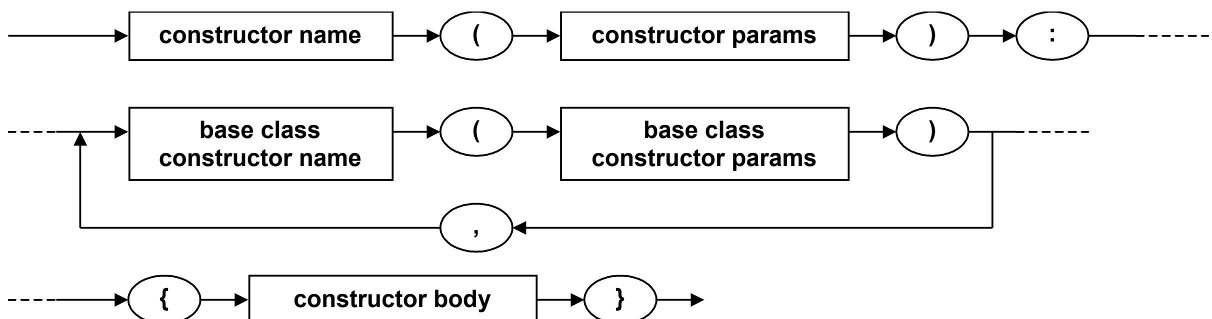
Arī destruktori, tāpat kā konstruktori, likvidējot objektu, tiek izsaukti kaskādes veidā, tikai:

- izsaukšana notiek pretēji mantošanas secībai – vispirms pašas klases konstruktors, tad bāzes klašu konstruktori utt.,
- destruktoru kaskādes veida izsaukšana nav jādefinē, jo katrā klasē ir tikai viens destruktors, tāpēc izsaukšana notiek automātiski.

Pirmkoda piemērs 9.7 demonstrē destruktora kaskādes veida izsaukšanu – likvidējot objektu *s* rindiņā 31:

- vispirms tiek izsaukts klases *student* destruktors, izdrukājot "STUDENT Peter DELETED",
- pēc tam tiek automātiski izsaukts klases *person* destruktors, izdrukājot "PERSON Peter DELETED".

Sintakse 9.3. ***class constructor definition*** **(konstruktora definēšana)**



Atklātā un slēptā mantošana.

Atklāto un slēpto mantošanu nosaka attiecīgi atslēgas vārdi *public* un *private* pirms bāzes klases nosaukuma, veicot mantošanas procesu (sk. sintaksi 9.2).

- Ja tiek mantots atklāti (*public*, pirmkods 9.7, rinda 15), 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*), tad visi bāzes klases elementi nonāk mantotajā klasē *private* statusā, tātad, ir pieejami tikai no mantotās klases.

Pirmkods 9.7. Mantošana (*cls7inherit.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 class person
```

```
05 {
06 protected:
07     string name;
08     int age;
09 public:
10     person (const string&, int);
11     ~person ();
12     void print ();
13 };
14
15 class student:public person
16 {
17     int semester;
18 public:
19     student (const string&, int, int);
20     ~student ();
21     void print ();
22 };
23
24 int main ()
25 {
26     person p ("Liz", 19);
27     p.print ();
28     student s ("Peter", 20, 4);
29     s.print ();
30     return 0;
31 };
32
33 person::person (const string &n, int a)
34 {
35     name = n;
36     age = a;
37 };
38
39 void person::print ()
40 {
41     cout << name << " " << age << endl;
42 };
43
44 person::~person ()
45 {
46     cout << "PERSON " << name << " DELETED" << endl;
47 };
48
49 student::student (const string &n, int a, int s):person (n, a)
50 {
51     semester = s;
52 };
53
54 void student::print ()
55 {
56     cout << name << " " << age << " " << semester << endl;
57 };
58
59 student::~student ()
60 {
```

```
61     cout << "STUDENT " << name << " DELETED" << endl;
62 }
```

Programmas darbības piemērs:

```
Liz 19
Peter 20 4
STUDENT Peter DELETED
PERSON Peter DELETED
PERSON Liz DELETED
```

9.5.2. 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 bāze klasi" drīkst veidot objektus ar tipu "atvasinātā klase" (sk. pirmkoda piemēru 9.8 rindu 28, kad pie mainīgā *p[1]* ar tipu *person** tiek veidots objekts ar tipu *student*, nevis *person*).

Šāda mainīgo tipa un objekta tipa neatbilstība var radīt problēmas gadījumos, ja mantotajā klasē **tiek pārdefinētas bāzes klases funkcijas** (piemēram, pirmkoda piemērā 9.8 – gan klasē *person*, gan klasē *student* ir funkcija *print()* bez parametriem) – 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, tiktu izsaukta objektam neatbilstoša funkcija (šajā gadījumā – objektam ar tipu *student* tiktu izsaukta klases *person* attiecīgā funkcija).

Lai novērtu šo mainīgā un objekta tipa neatbilstības problēmu, bāzes klasē attiecīgā funkcija jāatzīmē kā **virtuāla**, pirms funkcijas pielietojot atslēgas vārdu *virtual* (pirmkods 9.8, rindas 11,12). Kā redzams no piemēra, šī problēma attiecas arī uz destruktoriem. Šajā piemērā *virtual* statuss nodrošina to, ka objektam *p[1]* rindās 29 un 30 tiek izsaukt pareizās (klases *student*) funkcijas – attiecīgi *print()* un destruktors.

Pirmkods 9.8. Virtualās funkcijas (*cls8virtual.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 class person
05 {
06 protected:
07     string name;
08     int age;
09 public:
10     person (const string&, int);
11     virtual ~person ();
12     virtual void print ();
13 };
14
15 class student:public person
16 {
17     int semester;
```

```
18 public:
19     student (const string&, int, int);
20     ~student ();
21     void print ();
22 };
23
24 int main ()
25 {
26     person* pp[2];
27     pp[0] = new person ("Liz", 19);
28     pp[1] = new student ("Peter", 20, 4);
29     for (int i=0; i<2; i++) pp[i]->print ();
30     for (int i=0; i<2; i++) delete pp[i];
31     return 0;
32 };
33
34 person::person (const string &n, int a)
35 {
36     name = n;
37     age = a;
38 };
39
40 void person::print ()
41 {
42     cout << name << " " << age << endl;
43 };
44
45 person::~person ()
46 {
47     cout << "PERSON " << name << " DELETED" << endl;
48 };
49
50 student::student (const string &n, int a, int s):person (n, a)
51 {
52     semester = s;
53 };
54
55 void student::print ()
56 {
57     cout << name << " " << age << " " << semester << endl;
58 };
59
60 student::~student ()
61 {
62     cout << "STUDENT " << name << " DELETED" << endl;
63 }
```

Programmas darbības piemērs:

```
Liz 19
Peter 20 4
PERSON Liz DELETED
STUDENT Peter DELETED
PERSON Peter DELETED
```

9.5.3. Daudzkāršā mantošana

Daudzkāršā mantošana ir mantošana, izmantojot vairāk nekā vienu bāze klasi. C++ atšķirībā no daudzām citām programmēšanas valodām nodrošina šādu iespēju.

Pirmkoda piemērs 9.9 demonstrē daudzkāršo mantošanu, apvienojot divas klases: *inputinteger*, kas māk nolasīt skaitli un saglabāt laukā *value*, un *outputinteger*, kas māk no lauka *value* izdrukāt skaitli.

Veicot daudzkāršo mantošanu tiek izveidota klase *iointeger*, kas māk, gan nolasīt un saglabāt skaitli, gan saglabātu skaitli izdrukāt.

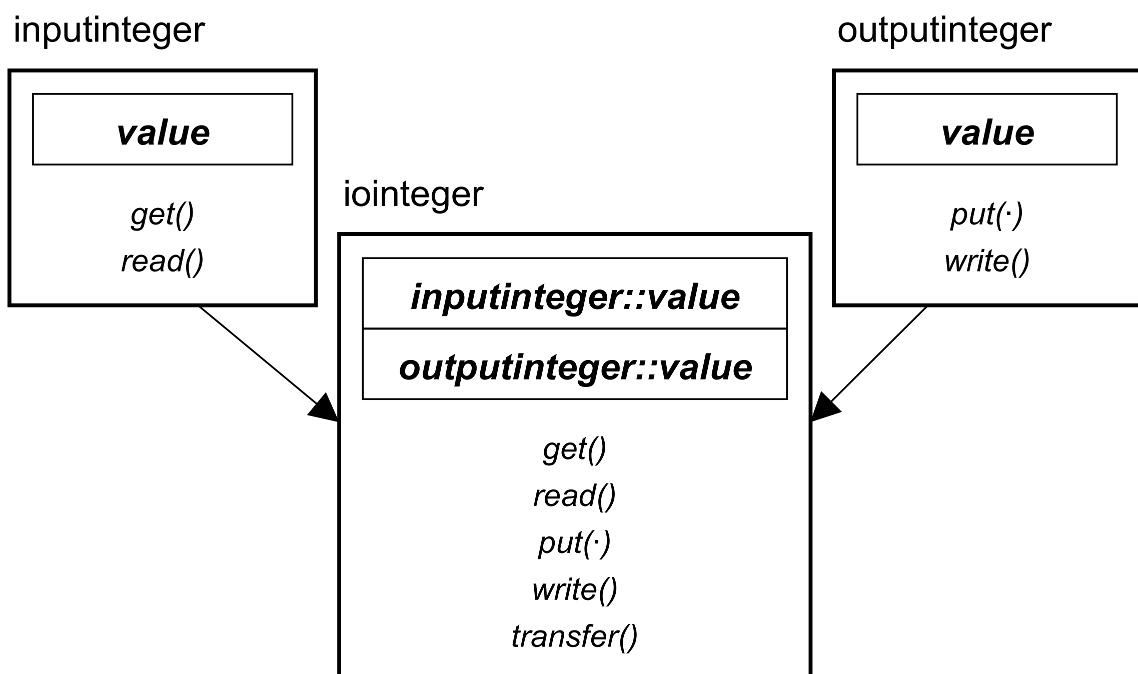
Pirmkoda piemērā 9.9 rindas 33-39 funkcionāli veic to pašu darbu, izmantojot divas atsevišķas klases, ko rindas 40-43, izmantojot vienu, daudzkāršās mantošanas ceļā iegūtu klasi.

Pirmkoda piemērs 9.9 un tam atbilstošais attēls 9.4 demonstrē šādu daudzkāršās mantošanas īpašību:

- mantojot vienāda nosaukuma elementus no dažādām klasēm, tie kļūst par dažādiem elementiem, pie kuriem var piekļūt, norādot bāzes klasi.

Šīs īpašības dēļ klasē *iointeger* ir nepieciešama funkcija *transfer()* – lai pārrakstītu vērtību no viena *value* lauka uz otru.

Pirmkoda piemērā 9.10 vienāda nosaukuma elementu neērtība, kas rodas daudzkāršās mantošanas gadījumā, tiek atrisināta.



Attēls 9.4. Daudzkāršā mantošana (atbilst pirmkoda piemēram 9.9)

Pirmkods 9.9. Daudzkāršā mantošana (*cls9multiple.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 class inputinteger
05 {
06 protected:
```

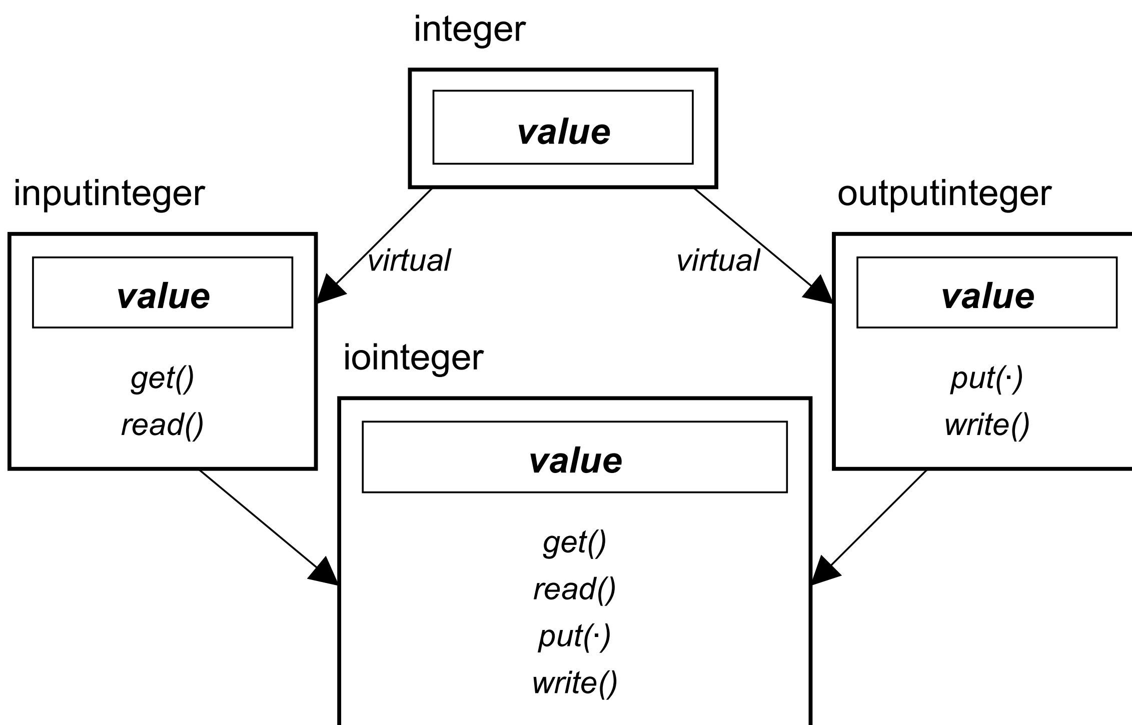
```
07     int value;
08 public:
09     int get () { return value; };
10     void read () { cin >> value; };
11 };
12
13 class outputinteger
14 {
15 protected:
16     int value;
17 public:
18     bool put (int i) { value = i; };
19     void write () { cout << value << endl; };
20 };
21
22 class iointeger: public inputinteger, public outputinteger
23 {
24 public:
25     void transfer ()
26     {
27         outputinteger::value = inputinteger::value;
28     }
29 };
30
31 int main ()
32 {
33     inputinteger inp;
34     outputinteger outp;
35     int i;
36     inp.read ();
37     i = inp.get ();
38     outp.put (i);
39     outp.write ();
40     iointeger ioint;
41     ioint.read ();
42     ioint.transfer ();
43     ioint.write ();
44     return 0;
45 }
```

Programmas darbības piemērs:

```
5
5
7
7
```

Vienāda nosaukuma elementu sapludināšana daudzkāršajā mantošanā.

Ja nepieciešams, ka no divām bāzes klasēm nākuši vienāda nosaukuma elementi mantotajā klasē nedublētos, bet tiktu uztverti kā viens, šīm abām bāzes klasēm savukārt šis elements, izmantojot virtuālo mantošanu, jāmanto no kādas kopējas bāzes klases. Tas demonstrēts pirmkoda piemērā 9.10 un attēlā 9.5, izmantojot papildus bāzes klasi *integer*. Virtuālā mantošana tiek veikta, pie bāzes klases lietojot papildus atslēgas vārdu *virtual* (pirmkoda piemērs 9.10, rindas 10,17)



Attēls 9.5. Daudzkāršā mantošana (atbilst pirmkoda piemēram 9.10)

Pirmkods 9.10. Daudzkāršā mantošana (*cls10multiple.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 class integer
05 {
06 protected:
07     int value;
08 };
09
10 class inputinteger: public virtual integer
11 {
12 public:
13     int get () { return value; };
14     void read () { cin >> value; };
15 };
16
17 class outputinteger: public virtual integer
18 {
19 public:
20     bool put (int i) { value = i; };
21     void write () { cout << value << endl; };
22 };
23
24 class iointeger: public inputinteger, public outputinteger
25 {
26 };
27
28 int main ()
29 {
```

```
30     iointeger ioint;
31     ioint.read ();
32     ioint.write ();
33     return 0;
34 }
```

Programmas darbības piemērs:

```
5
5
```

9.6. Citi objektorientētās programmēšanas mehānismi

9.6.1. 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 (pirmkods 9.11, rindas 10,11). Tas nodrošina visu klases elementu pieeju no dotajiem moduļiem (pirmkods 9.11, rindas 16,24).

Pirmkods 9.11. Draugu funkcijas un klases (*cls11friend.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 class person
05 {
06     string name;
07     int age;
08 public:
09     person (const string&, int);
10     friend void print (const person&);
11     friend class personprint;
12 };
13
14 void print (const person &p)
15 {
16     cout << p.name << " " << p.age << endl;
17 };
18
19 class personprint
20 {
21 public:
22     void print (const person &p)
23     {
24         cout << p.name << " " << p.age << endl;
25     };
26 };
27
28 int main ()
29 {
```

```
30     person p ("Liz", 19);
31     print (p);
32     personprint pp;
33     pp.print (p);
34     return 0;
35 };
36
37 person::person (const string &n, int a)
38 {
39     name = n;
40     age = a;
41 }
```

Programmas darbības piemērs:

```
Liz 19
Liz 19
```

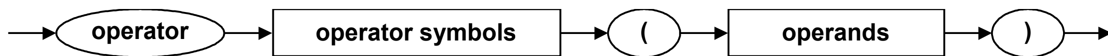
9.6.2. Operatoru pārslogošana

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 (sintakse 9.4).

Sintakse 9.4. ***operator as a function*** **(operators kā funkcija)**



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 (piemēram, operatori += un << pirmkoda piemērā 9.12).
- 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 (piemēram, operators + pirmkoda piemērā 9.12).

Abu pārslogošanas veidu atšķirības izpaužas tikai pašā pārdefinēšanas procesā – no izmantošanas viedokļa dažādi pārslogotie operatori neatšķiras.

Parasti operatorus pārslogo veidā nr. 2 – kā klases iekšēju funkciju, tas palīdz strukturēt programmu, operatoru piesaistot noteiktai klasei.

Tomēr, ja operatora pirmais operands ir kāda standarta klase (kā, piemēram, *ostream* pirmkoda piemērā 9.12), tad nav citu variantu kā vien pārslogot operatoru kā neatkarīgu funkciju, citādi to nevar izdarīt bez pašas standarta klases izmaiņas.

Pirmkoda piemērs 9.12 demonstrē operatoru pārslogošanu klasei *integer*, kas satur vienu veselu skaitli. Tiek pārslogoti 3 operatori – saskaitīšanas operators (+), saskaitīšanas ar piešķiršanu operators (+=) un standarta formatētās izdrukas operators klasei *ostream* (<<).

Jāuzsver, ka, pārslogojot operatoru << klasei *ostream*, pārslogošana attiecas ne tikai uz objektu *cout*, bet arī uz failiem (klases *ofstream*, *fstream*) un klasi *stringstream*, jo nosauktās klases tiek mantotas no klases *ostream*.

Pirmkods 9.12. Operatoru pārslogošana (*cls12operator.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 class integer
05 {
06     int value;
07 public:
08     integer (int i) { value = i; };
09     integer () {};
10     integer operator+ (const integer&);
11     friend integer& operator+= (integer&, const integer&);
12     friend ostream &operator<< (ostream&, const integer&);
13 };
14
15 int main ()
16 {
17     integer i(5);
18     integer j(2);
19     j += i;
20     integer k;
21     k = i + j;
22     cout << i << endl;
23     cout << j << endl;
24     cout << k << endl;
25     return 0;
26 };
27
28 integer &operator+= (integer &i, const integer &k)
29 {
30     i.value += k.value;
31     return i;
32 };
33
34 ostream &operator<< (ostream &out, const integer &i)
35 {
36     out << i.value;
37 };
38
```

```
39 integer integer::operator+ (const integer &i)
40 {
41     integer ret(value);
42     ret.value += i.value;
43     return ret;
44 }
```

Programmas darbības piemērs:

```
5
7
12
```

Komentāri pie pirmkoda piemēra 9.12.

- Operators =+ (rindas 11,28-32) tiek pārslogots kā neatkarīga funkcija, bet šajā gadījumā to drīkstēja pārslogot arī kā klases iekšējo funkciju.
- Operators << (rindas 12,34-37) tiek pārslogots kā neatkarīga funkcija, un tas ir vienīgais variants, jo *ostream* ir standarta klase.
- Izdrukas operatora pārslogošana << ir ļoti specifiska un to labāk veikt pēc parauga, nevis vispirms mēģināt izprast tā darbības principus visos sīkumos.
- Operators + (rindas 10,39-44) tiek pārslogots, kā klases iekšējā funkcijā, tāpēc par pirmo operandu pārdefinēšanas laikā formāli uzstājas pats objekts (sk. lauku *value* rindā 41).