5. C++ funkcijas

Nodaļas saturs:

5.1. Kopsavilkums
5.2. Funkciju pielietošana
5.3. Funkcijas deklarācija un prototips
5.4. Funkcijas realizācija
5.5. Funkcijas izsaukums
5.6. Parametri-vērtības un parametri-references
5.7. Funkcijas pārslogošana
5.8. Noklusētie parametri
5.9. Parametri-konstantes
5.10. Lokālie un statiskie mainīgie
5.11. Vērtības atgriešanas mehānisms
5.12. Rekursīvas funkcijas

5.1. Kopsavilkums

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.

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++ 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*).

Funkcijas 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.

Piemēram (sk. pirmkodu 5.2), funkcijai *add* ir divi parametri – a un b (rinda 4), bet, šo funkciju izsaucot galvenajā funkcijā *main*, caur tiem tiek padoti argumenti x un y (rinda 13).

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**.

Nākošie pieci pirmkoda piemēri 5.1-5.5 visi apraksta no izmantošanas viedokļa pilnīgi vienādas programmas – tās pieņem no klaviatūras 2 veselus skaitļus un pēc tam izdrukā to summu.

Pirmkods 5.1. Programma divu skaitļu saskaitīšanai bez speciālas saskaitīšanas funkcijas

```
01 #include <iostream>
02 using namespace std;
03
04 int main ()
05 {
06     int x, y, z;
07     cin >> x >> y;
08     z = x + y;
09     cout << z << endl;
10     return 0;
11 }
```

Programmas darbības piemērs:

```
3
4
7
```

Pirmkods 5.2. Programma divu skaitļu saskaitīšanai, izmantojot speciālu saskaitīšanas funkciju *add* (attēls 5.1(c))

```
01 #include <iostream>
02 using namespace std;
03
04 int add (int a, int b)
05 {
06     return a + b;
07 };
08
09 int main ()
10 {
11     int x, y, z;
12     cin >> x >> y;
13     z = add (x, y);
14     cout << z << endl;
15     return 0;
16 }
```

Programmas darbības piemērs:

```
3
4
7
```

Pirmkods 5.3. Programma divu skaitļu saskaitīšanai, izmantojot speciālu saskaitīšanas funkciju *add*, atsevišķi izdalot prototipu (attēls 5.1(b))

```
01 #include <iostream>
02 using namespace std;
03
04 int add (int, int);
05
06 int main ()
07 {
08     int x, y, z;
09     cin >> x >> y;
10     z = add (x, y);
11     cout << z << endl;
12     return 0;
13 };
14
15 int add (int a, int b)
16 {
17     return a + b;
18 }
```

Programmas darbības piemērs:

```
3
4
7
```

5.2. Funkciju pielietošana

Funkcijas ir sastopamas C++ programmās trīs dažādās formācijās:

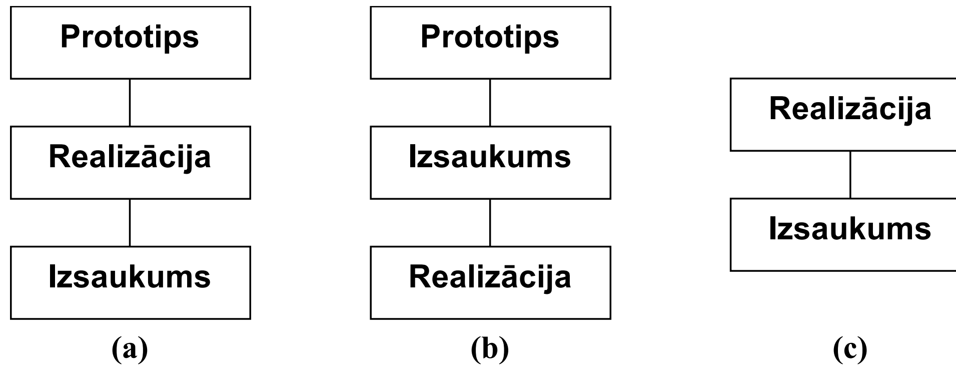
1. Funkcijas prototips jeb deklarācija (pirmkods 5.3, rinda 4);
2. Funkcijas realizācija jeb definīcija (pirmkods 5.3, rindas 15-18);
3. Funkcijas izsaukums (pirmkods 5.3, rinda 10).

Funkcijas **prototips** ir tāda informācija par funkciju, kas ietver funkcijas vārdu, parametru skaitu un tipus, kā arī atgriežamo tipu.

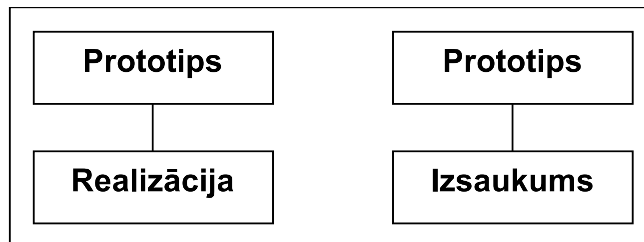
Trīs funkcijas formāciju secību programmā nosaka šādi noteikumi:

1. Funkcijas prototipam ir jābūt pirms realizācijas un izsaukuma;
2. Realizācijas un izsaukuma savstarpējai secībai nav nozīmes, tie var atrasties arī dažādos failos;
3. Funkcijas prototips var tikt izlaists, tādā gadījumā tā lomu pilda realizācija (pirmkods 5.2 un attēls 5.1(c)).

Tādējādi ir iespējami šādi secības varianti (attēli 5.1 un 5.2):



Attēls 5.1. Funkcijas formāciju secības varianti vienā failā



Attēls 5.2. Funkcijas formāciju izvietojums divos failos

Pirmkods 5.4. Programma divu skaitļu saskaitīšanai, izvietojot pirmkodu vairākos (trīs) failos (atbilst attēlam 5.2)

add.h

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

fun4fun.cpp

```
02 #include "add.h"
03
04 int add (int a, int b)
05 {
06     return a + b;
07 }
```

fun4main.cpp

```
08 #include "add.h"
09 #include <iostream>
10 using namespace std;
11
12 int main ()
13 {
14     int x, y, z;
15     cin >> x >> y;
16     z = add (x, y);
17     cout << z << endl;
18     return 0;
19 }
```

Programmas darbības piemērs:

```
3
4
7
```

Pirmkoda piemērs 5.4 pēc būtības atbilst pirmkoda izvietojumam, kas redzams attēlā 5.2. Atšķirība ir tikai tehniska: lai prototips nebūtu jāatkārto divas reizes, tas tiek ievietots atsevišķā **hedera** (**.h*) failā, un pēc tam iekļauts katrā no *cpp* failiem, izmantojot direktīvu

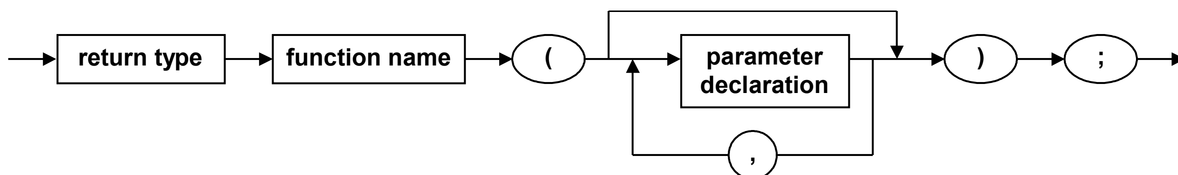
#include. Svarīgi, ka šajā gadījumā faila nosaukums jāliek dubultajās pēdiņās (atšķirībā no standarta bibliotēku iekļāvumiem, kurus liek stūra iekavās). Kaut arī visiem 3 failiem doti to nosaukumi, tomēr būtiski ir tikai, lai hedera fails "add.h" sauktos kā piemērā, jo tā nosaukums parādās abu pārējo programmas failu tekstā. Bez tam ir svarīgi, ka hedera fails atrodas tajā pašā direktorijā, kur abi *cpp* faili, citādi direktorija būtu jāatspoguļo faila iekļāvumā. Šajā piemērā var redzēt atšķirību starp prototipu un funkcijas galvu tās realizācijas daļā – prototipā drīkst neuzrādīt parametru vārdus, jo pietiek ar parametru tipiem.

5.3. Funkcijas deklarācija un prototips

Funkcijas deklarācija ir paziņojums par to, ka funkcijas programmā tiks izmantota. Funkcijas deklarācija obligāti satur **funkcijas prototipu** (funkcijas vārds, atgriežamais tips un parametru tipi), bet var saturēt arī parametru nosaukumus. Funkcijas prototipa uzdevums ir noteikt funkcijas interfeisu.

Funkcijas deklarācijai (piemēram, pirmkods 5.3, rinda 4) ir šāda sintakse:

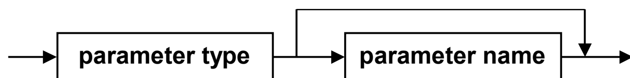
Sintakse 5.1. ***function declaration*** **(funkcijas deklarācija)**



kur

return type – atgriežamais tips, kas var būt arī tukšs (*void*)

Sintakse 5.2. ***parameter declaration*** **(parametra deklarācija)**



Parametra deklarācija šeit dota vienkāršotā variantā. Paplašināto variantu sk. zemāk.

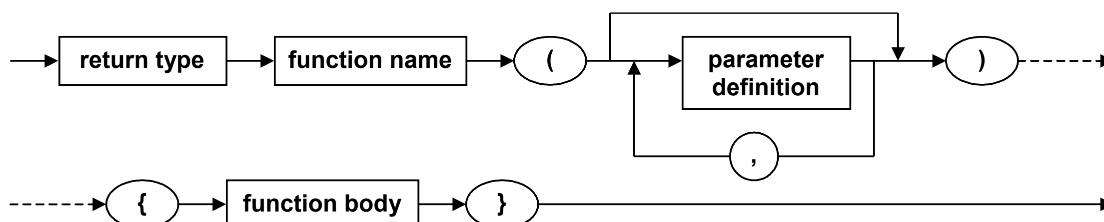
5.4. Funkcijas realizācija

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

Funkcijas galvā (piemēram, pirmkods 5.3, rinda 15), atšķirībā no prototipa, ir obligāti jāuzrāda parametru nosaukumi.

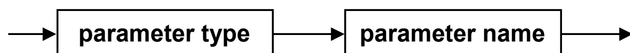
Funkcijas realizācijai (piemēram, pirmkods 5.3, rindas 15-18) ir šāda sintakse:

Sintakse 5.3. ***function realization*** **(funkcijas realizācija)**



Atšķirībā no parametra deklarācijas, parametra definīcijā ir obligāti uzrādāms parametra vārds:

Sintakse 5.4. ***parameter definition*** **(parametra definīcija)**



Parametra definīcija šeit dota vienkāršotā variantā. Paplašināto variantu sk. zemāk.

Funkcijas ķermenim, ja tā atgriežamais tips nav *void*, obligāti jāsatur vismaz viens **atgriešanas operators** (*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.

Atgriešanas operatoram (piemēram, pirmkods 5.3, rinda 17) ir šāda sintakse:

Sintakse 5.5. ***return statement*** **(atgriešanas operators)**

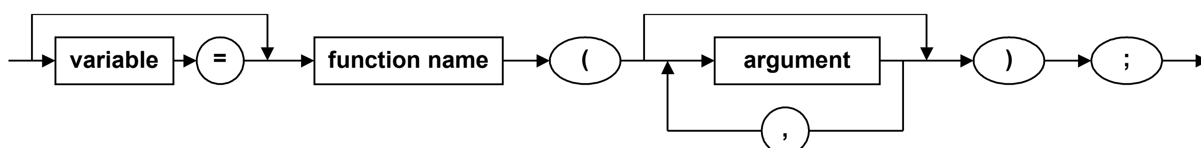
Sintakse 5.6. ***void return statement*** **(tukšais atgriešanas operators)**

5.5. Funkcijas izsaukums

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

Funkcijas izsaukumam (piemēram, pirmkods 5.3, rinda 10) ir šāda sintakse:

Sintakse 5.7. ***function call*** **(funkcijas izsaukums)**



5.6. Parametri-vērtības un parametri-references

Pirmkoda piemēros 5.1-5.4 abi funkcijas *add* parametri bija paredzēti informācijas padošanai funkcijai, bet iegūtā summa atgriezta, izmantojot atgriešanas mehānismu. Tā kā abi parametri ir paredzēti tikai informācijas padošanai uz funkcijas iekšieni, tad tos varētu nosaukt par parametriem-vērtībām.

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.

Tipisks gadījums, kad ar šādiem parametriem nepietiek, ir nepieciešamība funkcijai atgriezt vairāk nekā vienu vērtību (kā zināms ar atgriešanas mehānismu var atgriezt tikai vienu

vērtību). Šim nolūkam valodā C++ ir pieejami parametri-references (pēc līdzības ar valodas PASCAL parametriem-mainīgajiem). Lai parametru apzīmētu kā references parametru, to deklarējot (definējot) aiz parametra tipa jāliek references simbols '&'.

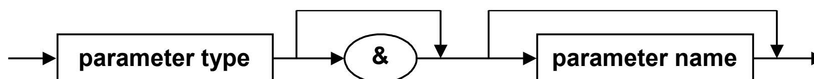
Parametrs-reference ir tāds parametrs, caur kuru padotais arguments netiek dublēts, bet nonāk koplietošanas režīmā starp izsaucošo un izsaucamo funkciju.

Parametrs-reference ir tāds parametrs, caur kuru informācija tiek padota abos virzienos – gan funkcijā iekšā, gan no funkcijas ārā.

References parametri nav pieejami valodā C. Tā vietā izmantojamas norādes (*pointers*).

Pievienojot jēdzienu par parametru-referenci, parametra deklarācija nu izskatās šādi:

Sintakse 5.8. ***parameter declaration 2*** **(parametra deklarācija)**



kur

 & – references simbols.

Ja parametrs ir parametrs-reference, tas jāatspoguļo gan funkcijas deklarācija, gan realizācijā, pievienojot references simbolu abās vietās. Funkcijas piemērs, kas izmanto references parametru, parādīts piemērā 5.5.

Pirmkods 5.5. Saskaitīšanas funkcija, kurā pirmais parametrs kalpo rezultāta atgriešanai (salīdzināt ar pirmkoda piemēru 5.3)

```
01 #include <iostream>
02 using namespace std;
03
04 void add (int&, int, int);
05
06 int main ()
07 {
08     int x, y, z;
09     cin >> x >> y;
10     add (z, x, y);
11     cout << z << endl;
12     return 0;
13 };
14
15 void add (int &r, int a, int b)
16 {
17     r = a + b;
18     return; // rinda nav obligāta
19 }
```

Programmas darbības piemērs:

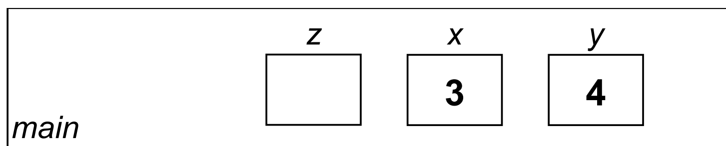
```
3
4
7
```

Pirmkoda piemērā 5.5 tiek izmantota funkcija ar tukšo atgriežamo tipu *void*, tāpēc *return* komanda šīs funkcijas ķermenī nav obligāta.

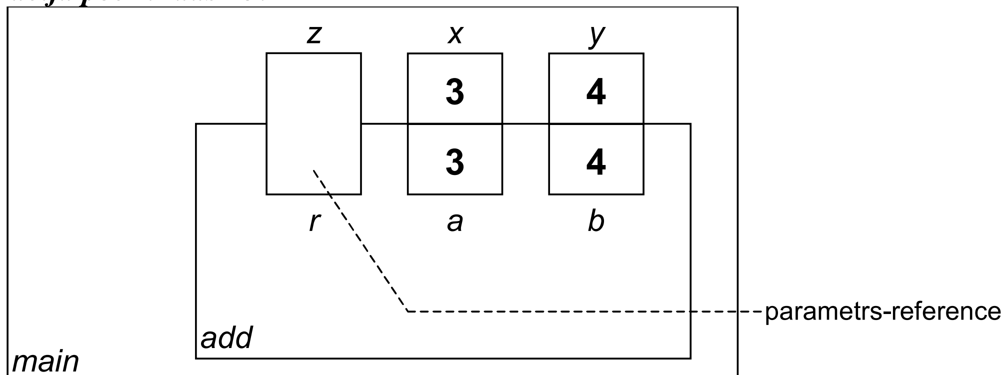
<u>Programmas darbības demonstrācija (pirmkoda piemērs 5.5).</u>

Tiek pieņemts, ka lietotājs uz pieprasījumu ievada attiecīgi skaitļus 3 un 4.

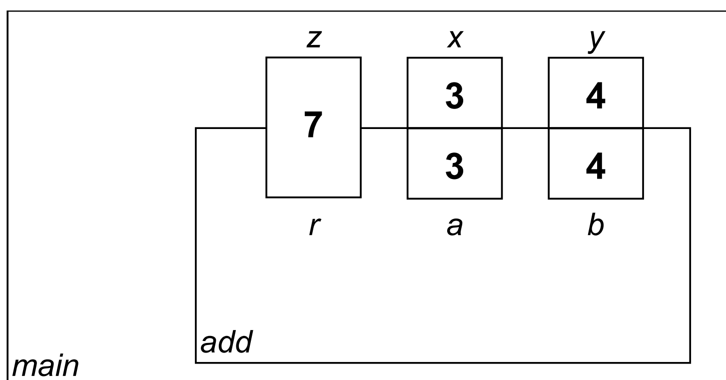
Konfigurācija pēc rindas 09:



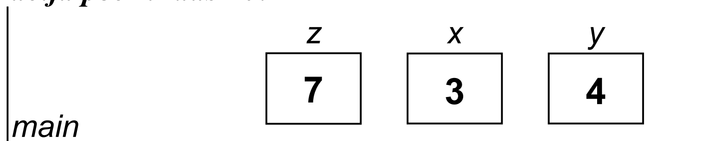
Konfigurācija pēc rindas 15:



Konfigurācija pēc rindas 17:



Konfigurācija pēc rindas 10:



5.7. Funkcijas pārslogošana

Pārslogošana (*overloading*) ir vienāda vārda izmantošana dažādu funkciju nosaukšanai.

Pārslogošana ir plašāka jēdziena – **polimorfisms** – apakšgadījums. Funkciju 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*) ir funkcijas apraksts, kas ietver funkcijas vārdu un parametru tipu virkni, ieskaitot arī parametru modifikatorus *const*.

- Atšķirībā no prototipa, signatūra neietver atgriežamo tipu.
- Nevar pastāvēt divas dažādas funkcijas, kuru prototips atšķiras tikai ar atgriežamo tipu, jo to signatūras vienalga sakrīt.

Pirmkods 5.6. Funkciju pārslogošana

<table>
<tr>
<td>

```
01 #include <iostream>
02 using namespace std;
03
04 void add (int& r, int a, int b);
05 int add (int a, int b);
06
07 int main ()
08 {
09     int x, y, z1, z2;
10     cin >> x >> y;
11     z1 = add (x, y);
12     add (z2, x, y);
13     cout << z1 << ", " << z2 << endl;
14     return 0;
15 };
16
17 void add (int &r, int a, int b)
18 {
19     r = a + b;
20 };
21
22 int add (int a, int b)
23 {
24     return a + b;
25 }
```

</td>
<td>

Programmas darbības piemērs:

```
3
4
7, 7
```

</td>
</tr>
</table>

Pirmkods 5.7. Funkciju pārslogošana. Kļūda: divdomīgs izsaukums

<table>
<tr>
<td>

```
01 #include <iostream>
02 using namespace std;
03
04 float add (float a, float b)
05 {
06     return a + b;
07 };
08
09 double add (double a, double b)
10 {
11     return a + b;
12 };
13
14 int main ()
15 {
16     int x, y;
17     cin >> x >> y;
18     cout << add (x, y) << endl;
19     return 0;
20 }
```

</td>
<td>

Kompilatora kļūdas paziņojums:

Divdomīgs pārslogotas funkcijas izsaukums (precīzāku aprakstu skatīt zemāk)

</td>
</tr>
</table>

Pirmkoda piemērā 5.7 parādītajai programmai kompilators atgriež apmēram šādu kļūdas paziņojumu:

```
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)
```

Programma tiktu nokompilēta bez kļūdām, ja 16. rindā mainīgie x un y tiktu deklarēti kā *float* vai *double*, tādējādi tieši norādot izmantojamo funkciju.

5.8. 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.

Noklusētie parametri vai nu funkcijas deklarācijā vai realizācijas galvā ietver noklusētās vērtības uzstādīšanu, tomēr parasti to dara tieši deklarācijā – abās vietās to darīt nedrīkst.

Pievienojot jēdzienu par noklusētajiem parametriem, parametra deklarācija izskatās šādi:

Sintakse 5.9. *parameter declaration 3* (parametra deklarācija)

Piezīme. Šī diagramma nenorāda, piemēram, ka parametri-referencēm nevar būt noklusētie.

Pirmkods 5.8. Noklusētie parametri

```
01 #include <iostream>
02 using namespace std;
03
04 int add (int a, int b=1);
05
06 int main ()
07 {
08      int x, y;
09      cin >> x >> y;
10      cout << add (x, y) << endl;
11      cout << add (x) << endl;
12      return 0;
13 };
14
15 int add (int a, int b)
16 {
17      return a + b;
18 }
```

Programmas darbības piemērs:

```
3
4
7
4
```

Komentāri pie pirmkoda piemēra 5.8.

- Funkcijas prototipā (rinda 4) drīkst izlaist parametru vārdus:
  `int add (int, int=1);`
- Noklusēto parametru izmantošana ir cieši saistīta ar funkciju pārslogošanu, piemēram, ja nebūtu pieejams noklusēto parametru mehānisms, tad varētu nodefinēt divas funkcijas *add*, tomēr, izmantojot noklusētos parametrus, tas izdarāms īsāk.

5.9. 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.

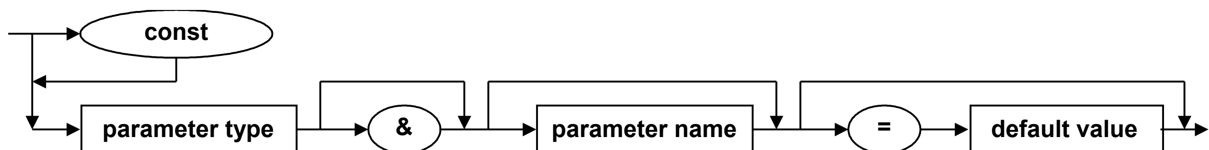
Konstantuma īpašību parasti izmanto lielu vērtību nodošanai funkcijām, sintakses līmenī nodrošinot, kas šīs vērtības funkcijas iekšienē netiks izmainītas. Mazu vērtību nodošanai (piemēram, *int*) pilnīgi pietiek ar parametriem-vērtībām, tomēr lielām vērtībām, lai taupītu resursus un tās argumentu nodošanas gaitā netiktu dublētas, izmanto parametrus-references vai parametrus norādes, papildus pieliekot konstantumu, ja nepieciešams nodrošināt, ka funkcijas iekšienē vērtību nedrīkst izmainīt. Parametriem-konstantēm modifikators *const* ir jāpielieto gan funkcijas deklarācijā, gan realizācijā.

Pirmkods 5.9. Parametri-konstantes

```
01 #include <iostream>
02 using namespace std;
03
04 double add (const double &a, const double &b);
05
06 int main ()
07 {
08     double x, y;
09     cin >> x >> y;
10     cout << add (x, y) << endl;
11     return 0;
12 };
13
14 double add (const double &a, const double &b)
15 {
16     return a + b;
17 }
```

Pievienojot konstantuma īpašību, parametra deklarācija izskatās šādi:

Sintakse 5.10. ***parameter declaration 4*** **(parametra deklarācija)**



5.10.Lokālie un statiskie mainīgie

Kā jau iepriekš tika minēts, lai nodrošinātu “melnās kastes” principu, jāveic informācijas slēpšana. Viens no mehānismiem, kā to izdarīt, lokālo mainīgo izmantošana funkcijās.

Lokālie mainīgie (*local variables*) ir mainīgie, kas darbojas (ir redzami) tikai noteikta bloka (parasti – funkcijas) iekšienē.

No funkcijas 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 deklar

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. 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.

Gan globālo, gan statisko mainīgo eksistences periods ir visa programma.

Sintakse 5.11. *static variable declaration* (statiskā mainīgā deklarācija)

Pirmkods 5.10. Lokālie, statiskie un globālie mainīgie

```
01 #include <iostream>
02 using namespace std;
03
04 int total_sum = 0;
05
06 int add (int, int);
07 int getSerialNumber (bool inc=true);
08
09 int main ()
10 {
11     int x, y, z;
12     cin >> x >> y;
13     z = add (x, y);
14     cout << "Sum:   " << z << endl;
15     cin >> x >> y;
16     z = add (x, y);
17     cout << "Sum:   " << z << endl;
```

```
18     cout << "Total: " << total_sum << endl;
19     cout << "Calls: " << getSerialNumber(false) << endl;
20     return 0;
21 };
22
23 int getSerialNumber (bool inc)
24 {
25     static int serial = 0;
26     if (inc) serial++;
27     return serial;
28 };
29
30 int add (int a, int b)
31 {
32     int r = a;
33     if (b > 0)
34     {
35         int i = 0;
36         while (i < b)
37         {
38             r++;
39             i++;
40         }
41     }
42     else if (b < 0)
43     {
44         for (int i=0; i>b; i--) r--;
45     };
46     getSerialNumber ();
47     total_sum += r;
48     return r;
49 }
```

Programmas darbības piemērs:

```
1
-2
Sum:   -1
3
4
Sum:   7
Total: 6
Calls: 2
```

Šī programma no lietotāja prasa ievadīt četrus veselus skaitļus. Pēc pirmo divu skaitļu ievadīšanas izdrukā to summu, bet pēc otru divu skaitļu ievadīšanas izdrukā vispirms otru divu skaitļu summu, tad visu 4 skaitļu summu, bet beigās summēšanas reižu skaitu (resp., 2).

Komentāri pie pirmkoda piemēra 5.10.

- Mainīgais *total_sum* (rinda 04) ir globālais mainīgais.
- Funkcija *getSerialNumber*, izsaucot ar true vai bez parametriem, veic pieskaitīšanu iekšējam skaitītājam, bet, izsaucot ar *false*, atgriež iekšējā skaitītāja vērtību.
- Funkcija add ir saskaitīšanas funkcija, kam salīdzinot ar parastu saskaitīšanas funkciju ir 3 īpatnības: (1) saskaitīšana/atņemšana realizēta īpatnējā veidā, lai demonstrētu atsevišķas valodas konstrukcijas, (2) izsaucot funkciju *getSerialNumber* rindā 46, tiek

pieskaitīta katra funkcijas izsaukšanas reize, (3) globālajā mainīgajā *total_sum* tiek krāta kopējā skaitļu summa.

- Mainīgais *serial* (rinda 25) ir statiskais mainīgais.
- Mainīgais *r* (rinda 32) ir lokālais mainīgais ar redzamības apgabalu rindās 32-48.
- Mainīgais *i* (rinda 35) ir lokālais mainīgais ar redzamības apgabalu rindās 35-39.
- Mainīgais *i* (rinda 44) ir lokālais mainīgais ar redzamības apgabalu rindā 44 (*for* operatora ietvaros).
- Funkcijas parametri *a* un *b* (rinda 30) no funkcijas *add* viedokļa ir lokālie mainīgie ar redzamības apgabalu rindās 30-48.
- Globālā mainīgā izmantošanas dēļ funkcija *add* nav universāla, bet piesaistīta šī globālā mainīgā nosaukumam.

5.11.Vērtības atgriešanas mehānisms

Vērtības atgriešanas mehānisms nodrošina funkcijas darbības pārtraukšanu un programmas darbības nodošanu uz šo funkciju izsaukušo moduli, nepieciešamības gadījumā atgriežot funkcijas vērtību. Atgriešanu nodrošina atgriešanas operators *return* (sk. sadaļu 5.4).

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.

Pirmkods 5.11. Funkcijas atgriešanas mehānisma demonstrācija ar funkcijām pirmskaitļu atpazīšanai

```
01 #include <iostream>
02 using namespace std;
03
04 bool prime1 (int num)
05 {
06     bool ret = true;
07     for (int i=2; i<num; i++)
08     {
09         if (num % i == 0) { ret = false;
               break; };
10     };
11     return ret;
12 };
13
14 bool prime2 (int num)
15 {
16     for (int i=2; i<num; i++)
17     {
18         if (num % i == 0) return false;
19     };
20     return true;
21 };
22
23 int main ()
24 {
25     int x;
26     cin >> x;
```

Programmas darbības piemērs nr. 1:

```
3
1
1
```

Programmas darbības piemērs nr. 2:

```
4
0
0
```

```
27     cout << prime1(x) << endl;
28     cout << prime2(x) << endl;
29     return 0;
30 }
```

Šī programma prasa ievadīt veselu (pozitīvu) skaitli un pasaka, vai tas ir pirmskaitlis, vai nav (2 reizes, ko nodrošina divas funkcijas). Funkcija *prime1* izmanto vienu atgriešanas operatoru pašās beigās, bet funkcija *prime2* – divus, kas nedaudz vienkāršo funkcijas pierakstu.

5.12.Rekursīvas funkcijas

Rekursija (*recursion*) ir process, kad funkcija izsauc pati 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 funkcijas bez rekursīvā izsaukuma (pirmkoda piemērs 5.12, rinda 6). 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 (pirmkoda piemērs 5.12, rinda 6);
- rekursīvais bloks (pirmkoda piemērs 5.12, rinda 7);

Pirmkoda piemērā 5.12. 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.

Pirmkods 5.12. Rekursīva funkcija faktoriāla izrēķināšanai

```
01  #include <iostream>
02  using namespace std;
03
04  int fact (int num)
05  {
06      if (num == 1) return 1;
07      else return num * fact (num-1);
08  };
09
10  int main ()
11  {
12      int x;
13      cin >> x;
14      cout << fact(x) << endl;
15      return 0;
16  }
```