12.Citas C++ konstrukcijas

Nodaļas saturs:

12.1.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.

Divi vismaz svarīgi gadījumi, kad būtu noderīga funkciju izsaukšana caur norādi:

- Iespēja izsaukt vienu no vairākām funkcijām, izmantojot indeksu, ja funkcijas tiek glabātas masīvā (pirmkods 12.1, rindas 33-38). Neizmantojot norādes uz funkcijām, būtu jāizmanto vairāku līmeņu *if-then-else* operators, 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 (pirmkods 12.1, rindas 16-19,29,37).

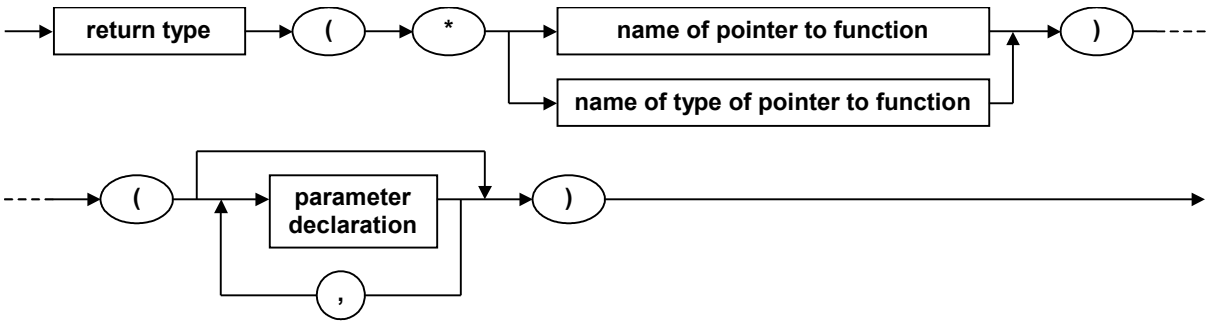
Norāžu uz funkcijām izmantošanu ilustrē pirmkoda piemērs 12.1.

Norādes uz funkciju deklarēšana.

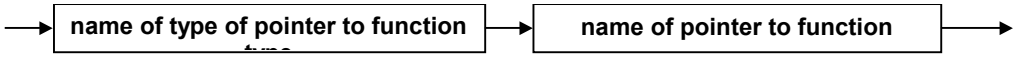
Norādi uz funkciju var deklarēt tiešā veidā (pirmkods 12.1, rinda 24; sintakse 12.1) vai netiešā veidā, sākumā ar operatoru *typedef* definējot funkcijas tipu (pirmkods 12.1, rindas 14+27; sintakse 12.1+12.2).

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 *.

Sintakse 12.1. ***declaration of pointer/pointer type to function*** **(norādes/norādes tipa uz funkciju deklarēšana)**



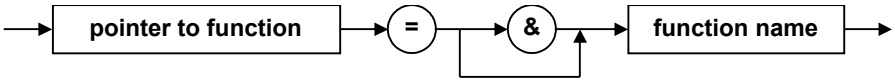
Sintakse 12.2. ***declaration of pointer to function using pointer type*** **(norādes uz funkciju deklarēšana, izmantojot norādes tipu)**



Funkcijas piešķiršana norādei.

Funkciju piešķir norādei tieši tāpat kā vērtību piešķir mainīgajam (pirmkods 12.1, rindas 25,25,33,34; sintakse 12.3), pirms funkcijas vārda var likt un var nelikt adreses operatoru. Tieši tāpat kā datu mainīgo, funkciju var nodot kā parametru citai funkcijai (pirmkods 12.1, rindas 29,37).

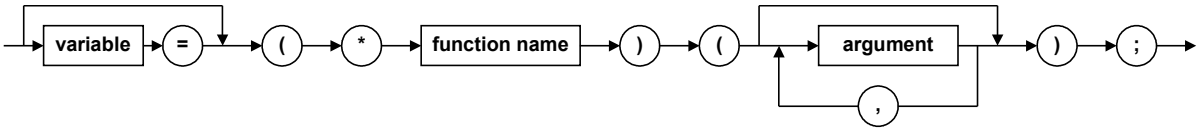
Sintakse 12.3. ***assignment of function to pointer*** **(funkcijas piešķiršana norādei)**



Funkcijas izsaukums caur norādi.

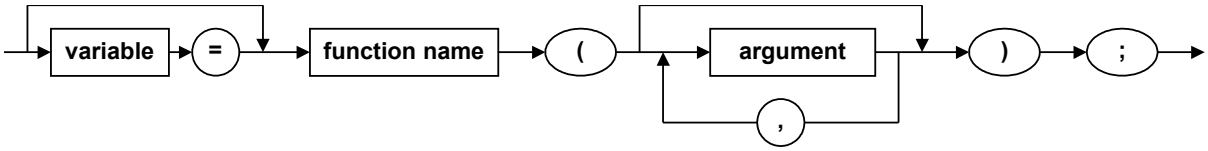
Funkcijas izsaukums caur norādi standarta variantā atšķiras no parasta funkciju izsaukuma ar to pašu, ar ko funkcijas norādes deklarēšana atšķiras no funkcijas prototipa – izsaucot funkcijas norādei liek priekšā norādes operatoru * un to visu liek iekavās (pirmkods 12.1, rindas 18,26; sintakse 12.4).

Sintakse 12.4. ***function call by pointer*** **(funkcijas izsaukums caur norādi)**



Tomēr jaunais standarts ļauj funkciju izsaukt caur norādi tieši tā, ka izsauc funkciju – t.i., neliekot norādei priekšā norādes operatoru * un neliekot to iekavās (pirmkods 12.1, rinda 28; sintakse 12.5).

Sintakse 12.5. ***function call by pointer simplified*** **(funkcijas izsaukums caur norādi vienkāršotais)**



Pirmkoda piemērs 12.1 demonstrē divu funkciju – *add* un *multiply* pielietošanu, izmantojot norādes uz funkcijām.

Pirmkods 12.1. Norāde uz funkciju (*msc1fun.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 int add (int a, int b)
05 {
06     return a + b;
07 };
08
09 int multiply (int a, int b)
10 {
11     return a * b;
12 };
13
14 typedef int (*int_function_type) (int, int);
15
16 int compute (int (*fp)(int, int), int a, int b)
17 {
18     return (*fp) (a, b);
```

```
19 };
20
21 int main ()
22 {
23     int a=7, b=5;
24     int (*fp)(int, int);
25     fp = add;
26     cout << (*fp) (a, b) << endl;
27     int_function_type fp2 = add;
28     cout << fp2 (a, b) << endl;
29     cout << compute (multiply, a, b) << endl;
30     a++;
31     b++;
32     int (*funcs[2])(int, int);
33     funcs[0] = add;
34     funcs[1] = multiply;
35     for (int i=0; i<2; i++)
36     {
37         cout << compute (funcs[i], a, b) << endl;
38     }
39     return 0;
40 }
```

Programmas darbības piemērs:

```
12
12
35
14
48
```

Komentāri pie pirmkoda piemēra 12.1.

- Rinda 14. Tiek definēts norādes uz funkcijām tips *int_function_type*, kas apzīmē tādu funkciju tipu, kas pieņem divus *int* parametrus un atgriež *int* tipa vērtību.
- Rinda 16. Tiek deklarēta funkcija *compute()*, kas kā pirmo parametru pieņem funkciju, kas pieņem divus *int* parametrus un atgriež *int* tipa vērtību.
- Rinda 18. Tiek izsaukta funkcija, kas tikusi padota caur pirmo parametru un uz kuru norāda *fp*.
- Rinda 24. Tiek deklarēta norāde funkciju norāde *fp*, kas var glabāt adresi tādai funkcijai, kas pieņem divus *int* parametrus un atgriež *int* tipa vērtību.
- Rinda 25. Funkcijas norādei *fp* tiek piešķirta funkcijas *add()* adrese (ievērojiet, ka aiz *add()* neseko parametru iekavas!).
- Rinda 26. Caur norādi *fp* tiek izsaukta funkcija *add()*.
- Rinda 27. Tiek deklarēta funkcijas norāde *fp2* un tai tiek piešķirta adrese funkcijai *add()*.
- Rinda 28. Caur norādi *fp2* tiek izsaukta funkcija *add()*.
- Rinda 29. Tiek izsaukta funkcija *compute()*, kam caur parametru tiek padota funkcija *multiply()*.
- Rinda 32. Tiek deklarēts masīvs *funcs[]* no diviem elementiem, kas var būt norādes uz funkcijām, kas pieņem divus *int* parametrus un atgriež *int* tipa vērtību.
- Rindas 33,34. Masīvs *funcs[]* tiek aizpildīts attiecīgi ar norādēm uz funkcijām *add()* un *multiply()*.

- Rinda 37. Tiek izsaukta funkcija pēc attiecīgās masīvā *funcs[]* glabātās norādes.

12.2.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 pirmkoda piemērs 12.2.

Pirmkods 12.2. Komandrindas argumenti (*msc2cmdline.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 int main (int argc, char **argv)
05 {
06     if (argc == 1)
07     {
08         cout << "\nUsage: ";
09         cout << argv[0] << " ARG1, ARG2 [, ...]" << endl;
10     }
11     else if (argc == 2)
12     {
13         cout << "\nYou must specify at least 2 arguments\n";
14         cout << "\nUsage: ";
15         cout << argv[0] << " ARG1, ARG2 [, ...]" << endl;
16     }
17     else
18     {
19         cout << "The values of the arguments are:\n";
20         for (int i=1; i<argc; i++)
21         {
22             cout << argv[i] << endl;
23         }
24     };
25     return 0;
26 }
```

Programmas darbības piemērs #1 (ietverot programmas izsaukumu komandrindā):

```
c:\ecpp\12msc>msc2cmdline

Usage: msc2cmdline ARG1, ARG2 [, ...]
```

Programmas darbības piemērs #2 (ietverot programmas izsaukumu komandrindā):

```
c:\ecpp\12msc>msc2cmdline ONE
```

```
You must specify at least 2 arguments

Usage: msc2cmdline ARG1, ARG2 [, ...]
```

Programmas darbības piemērs #2 (ietverot programmas izsaukumu komandrindā):

```
c:\ecpp\12msc>msc2cmdline ONE TWO
The values of the arguments are:
ONE
TWO
```

12.3.Masīvu apstrāde

Nodaļā par masīviem tika teikts, ka valodā C++ masīvu nevar uzskatīt par vienotu veselumu, ka masīvā katrs elements jāapstrādā atsevišķi (piemēram, pārrakstot vienu masīvu otrā). Tomēr, ja mēs masīvu uztveram, 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..." – no vienas puses, pēc līdzības ar atbilstošām funkcijām zema līmeņa simbolu virkņu apstrādei, no otras puses pēc līdzības ar funkcijām *read()* un *write()* failu apstrādē.

Masīvu (atmiņas apgabalu) apstrādes funkcijas (*memcpy()*, *memcmp()*, *memchr()*) darbojas ar atmiņas apgabaliem pa baitam, tomēr atšķirībā no zema līmeņa simbolu virkņu funkcijām un failu apstrādes funkcijām *read()* un *write()*, atmiņas apgabals šeit tiek padots caur norādēm ar tipu *void** (nevis *char**). No programmēšanas viedokļa gan šai atšķirībai ir visai simboliska nozīme.

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. Sk. pirmkodu 12.3, rindas 8,12.

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 vienāda1;
- <0, ja *buf1* mazāks par *buf2*;
- >0, ja *buf1* lielāks par *buf2*.

Sk. pirmkodu 12.3, rindas 17,18.

memchr

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

Funkcija *memchr()* meklē pirmo sastapto simbolu *c* masīvā *buf* pirmajos *count* simbolos. Funkcija atgriež norādi uz atrasto simbolu vai arī nulles norādi, ja simbols masīvā nav atrasts. Sk. pirmkodu 12.3, rinda 20.

Pirmkoda piemērs 12.3 demonstrē aprakstīto zema līmeņa masīva apstrādes funkciju izmantošanu.

Pirmkods 12.3. Masīvu apstrāde (*msc3mem.cpp*)

```
01 #include <iostream>
02 using namespace std;
03 const int arr_size = 3;
04
05 int main ()
06 {
07     int a=7, b=5;
08     memcpy ((void*)&b, (void*)&a, sizeof(int));
09     cout << a << " " << b << endl;
10     int arr1[arr_size] = {11, 22, 33};
11     int arr2[arr_size] = {55, 66, 77};
12     memcpy ((void*)arr2, (void*)&arr1, (arr_size-1)*sizeof(int));
13     for (int i=0; i<arr_size; i++)
14     {
15         cout << arr2[i] << endl;
16     };
17     cout << memcmp (arr1, arr2, (arr_size-1)*sizeof(int)) <<
                endl;
18     cout << memcmp (arr1, arr2, arr_size*sizeof(int)) << endl;
19     arr2[2] = 65;
20     void *p = memchr (arr2, 65, arr_size*sizeof(int));
21     cout << (char*)p - (char*)arr2 << endl;
22     return 0;
23 }
```

Programmas darbības piemērs:

```
7 7
11
22
77
0
-1
8
```

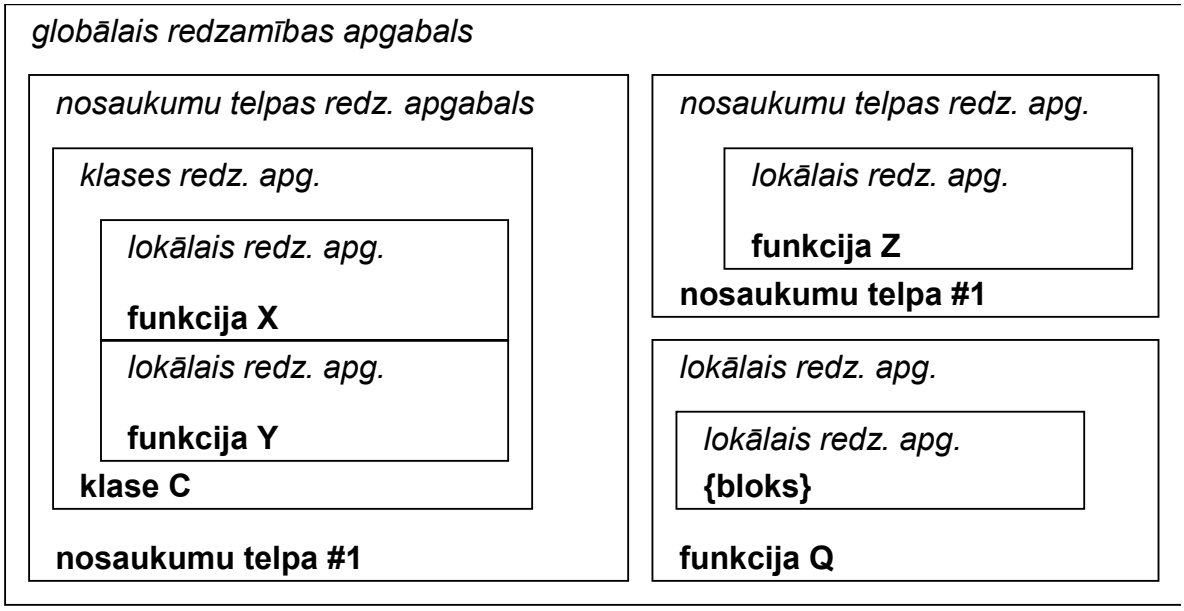
Komentāri pie pirmkoda piemēra 12.3.

- Rinda 8. Pārkopējot informāciju pa baitam, mainīgajam *b* tiek piešķirta mainīgā *a* vērtība. Ievērojiet, ka funkcijai *memcpy()* padodamās masīvu adreses ir jāpārveido uz tipu *void**.
- Rinda 9 pierāda, ka mainīgais *b* tagad satur to pašu vērtību, ko *a*.
- Rinda 12. No masīva *arr1* uz masīvu *arr2* pa baitam tiek pārkopēti pirmie divi skaitļi (8 baiti, pieņemot, ka *int* lielums ir 4 baiti).
- Rindas 13-16, izdrukājot "11 22 77", pierāda, ka masīva *arr2* pirmās divas vērtības ir nomainījušās.
- Rinda 17, izdrukājot 0, parāda, ka masīvu *arr1* un *arr2* pirmie 2 elementi (8 baiti) ir vienādi.
- Rinda 18, izdrukājot -1, parāda, ka masīvi *arr1* un *arr2*, salīdzinot tos pilnā garumā, nav vienādi.
- Rinda 19. Masīva *arr2* elementam nr. 2 tiek piešķirta vērtība 65, kas, pieņemot, ka *int* lielums ir 4 baiti un tā kodēšanā lietots *little-endian* princips, visticamāk nozīmē to, ka šī masīva elementa saturs pa baitam ir {65,0,0,0}.

- Rinda 20 meklē masīvā *arr2* baitu ar vērtību 65. Atrastā baita adrese tiek ievietota mainīgajā *p*.
- Rinda 21, atņemot no atrastā baita adreses masīva *arr2* sākuma adresi, parāda, ka baits ar vērtību 65 masīvā *arr2* atrodas pozīcijā 8.

12.4.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*.



Attēls 12.1. Dažādi redzamības apgabali

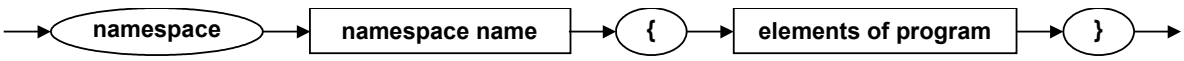
Nosaukumu telpu izmantošana sastāv no divām daļām:

- Programmas bloka piesaiste noteiktai nosaukumu telpai.
- Piekļūšana noteiktā nosaukumu telpā atrodošam programmas elementam.

Programmas bloka piesaiste nosaukumu telpai.

Programmas bloka 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 (sintakse 12.6; pirmkods 12.5, rindas 1,2,7).

Sintakse 12.6. ***assigning code to a namespace*** **(programmas daļas piesaiste nosaukumu telpai)**



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 nosaukumu telpas elementam

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. Piemēram, pirmkods 12.4, rinda 11.
- Pievienojot nosaukumu telpu aktuālajam redzamības gabalam, izmantojot atslēgas frāzi *using namespace* un nosaukumu telpas vārdu. Piemēram, pirmkods 12.5, rindas 10+15, vai pirmkods 12.6, rindas 15+16,18+19.

Pirmkoda piemērs 12.4 parāda iespēju izveidot funkciju ar nosaukumu *cout()* (rindas 1-4) un ievietojot to globālajā redzamības telpā, tajā pašā laikā nekonfliktējot ar izdrukas objektu *cout*, kam nosaukumu telpā *std* tiek piekļūts tiešā veidā (rinda 11).

Pirmkods 12.4. Nosaukumu telpas #1

msc4namespace.h:

```
01 int cout (int a, int b)
02 {
03     return a + b;
04 };
```

msc4namespace.cpp:

```
05 #include <iostream>
06 #include "msc4namespace.h"
07
08 int main ()
09 {
10     int a=7, b=5;
11     std::cout << cout (a, b) << std::endl;
12     return 0;
13 }
```

Programmas darbības piemērs:

```
12
```

Pirmkoda piemērs 12.5 parāda nosaukumu telpas *my_std* definēšanu, tajā ievietojot funkciju *cout()* (rindas 1-7). Pēc tam nosaukumu telpa *my_std* tiek iekļauta globālajā redzamības apgabalā (rinda 10), bet funkcija nekonfliktē ar izdrukas objektu *cout*, jo tam nosaukumu telpā *std* tiek piekļūts tiešā veidā (rinda 15).

Pirmkods 12.5. Nosaukumu telpas #2

msc5namespace.h:

```
01 namespace my_std
02 {
03     int cout (int a, int b)
```

```
04      {
05          return a + b;
06      }
07  };
```

msc5namespace.cpp:

```
08  #include <iostream>
09  #include "msc5namespace.h"
10  using namespace my_std;
11
12  int main ()
13  {
14      int a=7, b=5;
15      std::cout << cout (a, b) << std::endl;
16      return 0;
17  }
```

Programmas darbības piemērs:

```
12
```

Pirmkoda piemērs 12.6 demonstrē nosaukumu telpas iekļaušanu dažādos redzamības apgabalos:

- Rinda 15 nodrošina nosaukumu telpas *my_std* pievienošanu redzamības apgabalam rindās 16,17.
- Rinda 18 nodrošina nosaukumu telpas *std* pievienošanu redzamības apgabalam rindās 19-21.

Pirmkods 12.6. Nosaukumu telpas #3

msc6namespace.h:

```
01  namespace my_std
02  {
03      int cout (int a, int b)
04      {
05          return a + b;
06      }
07  };
```

msc6namespace.cpp:

```
08  #include <iostream>
09  #include "msc6namespace.h"
10
11  int main ()
12  {
13      int a=7, b=5, result;
14      {
15          using namespace my_std;
16          result = cout (a, b);
17      };
18      using namespace std;
19      cout << result << endl;
20      return 0;
21  }
```

Programmas darbības piemērs:

```
12
```

12.5.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 attēlā 12.2:

- 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
{
    kaut kāds pirmkods
    if (error1) throw some_exception1;
    kaut kāds pirmkods
    if (error2) throw some_exception2;
    kaut kāds pirmkods
}
catch (T1 exc)
    { izņēmumu apstrāde ar tipu T1}
catch (T2 exc)
    { izņēmumu apstrāde ar tipu T2}
catch (...)
    { citu izņēmumu apstrāde }
```

Attēls 12.2. Izņēmumu apstrādes 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 citā, bet apstrādāta citā funkciju izsaukumu hierarhijas līmenī (kas arī ir demonstrēts pirmkoda piemērā 12.7). 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, pirmkoda piemērs 12.7, rindas 20-23).

Šajā blokā programmētājs pats (nevis to nodrošina kompilators) apzina visas iespējamās kļūdas situācijas un nosaka izņēmuma izmešanu.

Izņēmuma izmešana ar ‘throw’

Izmantojot operatoru *throw* (pirmkoda piemērs 12.7, rindas 6,13,14) 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 vērtībai, bet arī tips, jo tieši atbilstoši izmestajam tipam tiks izvēlēta izņēmuma apstrādes funkcija.

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.

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.

Pirmkoda piemērs 12.7 demonstrē izņēmumu apstrādes mehānismu funkcijai process, kas rēķina izteiksmes $x + y / z$ vērtību, ja netiek pārkāpti nosacījumi – z nav 0, un saskaitāmie ir robežās 0..100.

Pirmkods 12.7. Izņēmumu apstrāde (*msc7exception.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 double x_over_y (double x, double y)
05 {
06     if (y == 0) throw "ERROR: Division by zero";
07     return x / y;
08 };
09
10 double x_plus_y_over_z (double x, double y, double z)
11 {
12     double div = x_over_y (y, z);
13     if (x > 100 || div > 100) throw 100;
14     else if (x < 0 || div < 0) throw 0.0;
15     return x + div;
16 };
17
18 void process (double x, double y, double z)
19 {
20     try
21     {
22         cout << x_plus_y_over_z (x, y, z) << endl;
23     }
```

```
24      catch (const char *error_text)
                  { cout << error_text << endl; }
25      catch (int error_num)
                  { cout << "ERROR: Addend too big" << endl; }
26      catch (...) { cout << "ERROR: Unknown error" << endl; };
27  };
28
29  int main ()
30  {
31      process (2, 4, 0);
32      process (999, 4, 0.5);
33      process (-2, 4, 0.5);
34      process (2, 4, 0.5);
35      return 0;
36  }
```

Programmas darbības piemērs:

```
ERROR: Division by zero
ERROR: Addend too big
ERROR: Unknown error
10
```

Komentāri pie pirmkoda piemēra 12.7.

- Funkcijas *process()* izsaukums rindā 31 (nulles dēļ argumentā #3) izsauc izņēmuma izmešanu rindā 6, bet tiek apstrādāts atbilstoši tā tipam blokā *catch (const char*)* pirmkoda rindā 24.
- Funkcijas *process()* izsaukums rindā 32 (pārāk lielā skaitļa dēļ argumentā #1) izsauc izņēmuma izmešanu rindā 13, bet tiek apstrādāts atbilstoši tā tipam blokā *catch (int)* pirmkoda rindā 25.
- Funkcijas *process()* izsaukums rindā 33 (negatīva skaitļa dēļ argumentā #1) izsauc izņēmuma izmešanu rindā 14, bet tiek apstrādāts noklusētajā apstrādes blokā *catch (...)* pirmkoda rindā 26, jo *catch* bloks, kas apstrādātu *double* vērtības, nav definēts.
- Funkcijas *process()* izsaukums rindā 34 nostrādā bez kļūdām un rindā 22 izdrukā vērtību 10 (10=2+4/0.5).