4. Masīvi un zema līmeņa simbolu virknes

Nodaļas saturs:

4.1. Masīvs

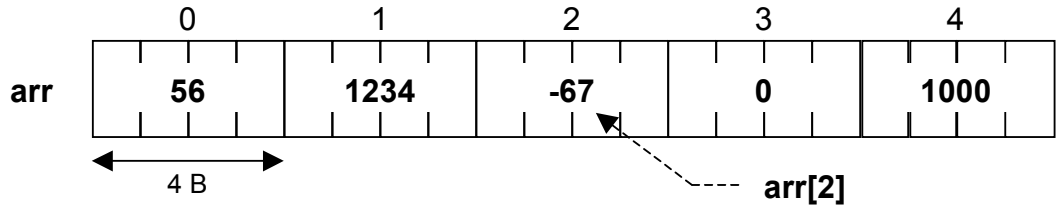
Masīvs (*array*) ir vienādu tipu mainīgo virkne.

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

Tādējādi, kopumā vienu masīva elementu identificē:

- masīva vārds,
- indekss.

Masīva indeksācija notiek pēc kārtas, bet valodā C++ – vienmēr, sākot ar 0. Tādējādi masīvā ar garumu n tā elementi tiek indeksēti robežās 0..n-1 (sk. attēlu 4.1).

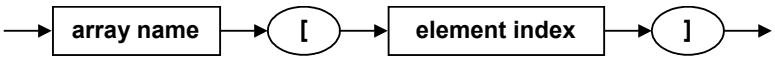


Attēls 4.1. Veselu skaitļu (int) masīvs* arr *garumā 5

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

Vēršanās pie viena elementa notiek izmantojot kvadrātiekavu operatoru [] (piemēram, pirmkods 4.1, rinda 9):

Sintakse 4.1. ***array element*** **(masīva elements)**



Piemēram, lai vērstos pie masīva elementa -67 (sk. attēlu 4.1), raksta *arr[2]*.

4.2. Statisks masīvs

Visvienkāršākais masīva veids ir statisks masīvs (*static array*) – tāds, kura elementu skaits ir zināms kompilācijas (programmas rakstīšanas) brīdī. Deklarējot statisku masīvu, ir jāuzrāda tā elementu skaits kā konstanta vērtība (pirmkods 4.1, rindas 7, 8).

Pirmkods 4.1. Statiska masīva izveidošana un izmantošana (*arr1static.cpp*)

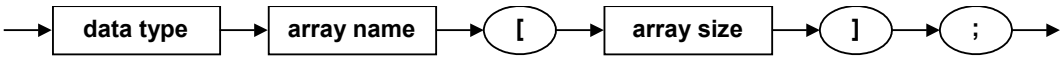
```
01 #include <iostream>
02 using namespace std;
03 const int arr_size = 5;
04
05 int main ()
06 {
07     int arr[arr_size] = {56,1234,-67,0,1000};
08     int arr2[5];
09     for (int i=0; i<arr_size; i++) arr2[i] = arr[i];
10     for (int i=0; i<arr_size; i++) cout << arr2[i] << " ";
11     cout << endl;
12     int arr3[] = {11,22,33,44,55,66,0};
13     for (int i=0; arr3[i]!=0; i++) cout << arr3[i] << " ";
14     cout << endl;
15     return 0;
16 }
```

Programmas darbības piemērs:

```
56 1234 -67 0 1000
11 22 33 44 55 66
```

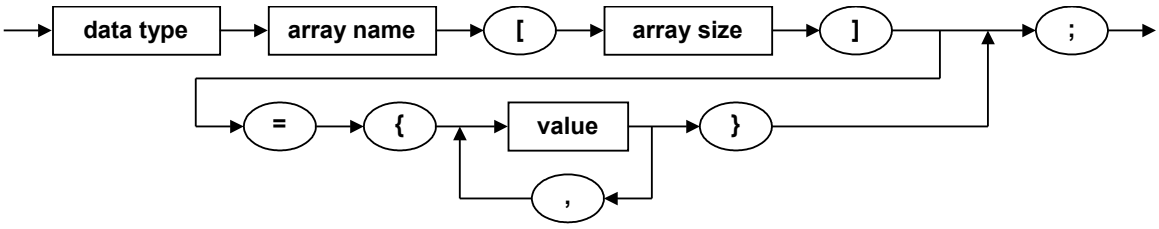
Vienkāršākais veids, kā deklarēt masīvu parādīts nākošajā sintaktiskajā diagrammā (skat. atbilstošo piemēru pirmkodā 4.1, rindā 8).

Sintakse 4.2. *static array declaration* (statiska masīva deklarēšana)



Deklarējot masīvu, tiek izdalīta atmiņa masīva elementiem, tomēr masīvs automātiski netiek inicializēts. Masīvu pa vienam elementam var inicializēt vēlāk, tomēr to ir iespējams izdarīt arī visam masīvam uzreiz – deklarācijas brīdī. Tālāk dota statiska masīva deklarēšana ar inicializāciju (skat. atbilstošo piemēru pirmkodā 4.1, rindā 7).

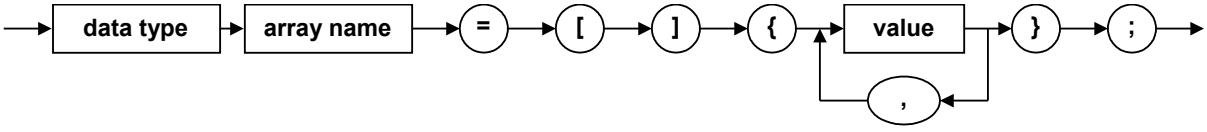
Sintakse 4.3. *static array declaration with initialization* (statiska masīva deklarēšana ar inicializāciju)



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

Trešais veids kā deklarēt statisku masīvu, ir atstāt tukšas kvadrātiekavas un neuzrādīt masīva izmēru tiešā veidā, bet ļaut to netieši izdarīt inicializācijas vērtību virknei, kā tas parādīts nākošajā sintaktiskajā diagrammā (skat. atbilstošo piemēru pirmkodā 4.1, rindā 12).

Sintakse 4.4. *static array declaration with initialization 2* (statiska masīva deklarēšana ar inicializāciju 2)



4.3. Dinamisks masīvs

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

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

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

Pirmkods 4.2. Dinamiska masīva izveidošana un izmantošana (*arr2dynamic.cpp*)

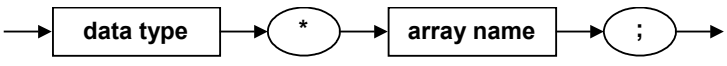
```
01 #include <iostream>
02 using namespace std;
03
04 int main ()
05 {
06     int *arr;
07     int size;
08     cout << "Input array size: ";
09     cin >> size;
10     arr = new int [size];
11     cout << "Input " << size << " integer values: " << endl;
12     for (int i=0; i<size; i++) cin >> arr[i];
13     cout << "Array:" << endl;
14     for (int i=0; i<size; i++) cout << arr[i] << " ";
15     cout << endl;
16     delete[] arr;
17     return 0;
18 }
```

Programmas darbības piemērs:

```
Input array size: 4
Input 4 integer values:
11
22
33
44
Array:
11 22 33 44
```

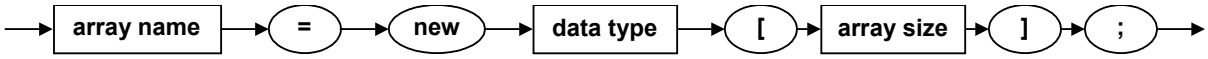
Dinamiskā masīva deklarēšana (piemēram, pirmkods 4.2, rinda 6) parādīta sintaktiskajā diagrammā 4.5.

Sintakse 4.5. *dynamic array declaration* (dinamiska masīva deklarēšana)



Deklarējot dinamisku masīvu, nav iespējama tā tūlītēja inicializācija, jo, atšķirībā no statiska masīva, dinamiskais masīvs pie deklarēšanas netiek līdz galam izveidots, tātad, nav uzreiz gatavs darbam. Pirms strādāt ar dinamisko masīvu, tam ir jāizdala atmiņa (pirmkods 4.2, rinda 10; sintaktiskā diagramma 4.6).

Sintakse 4.6. ***dynamic array creation*** **(dinamiska masīva izveidošana)**



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

Līdzīgi kā inicializējot statisku masīvu deklarēšanas brīdī, arī dinamiska masīva deklarēšanu un izveidošanu var apvienot vienā rindiņā. Piemēram, pirmkoda piemērā 4.2 rindas 6 un 10 var apvienot kā:

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

Kad dinamiskais masīvs ir izveidots, tad tā izmantošana ir identiska statiska masīva izmantošanai (pirmkods 4.2, rindas 12 un 14; sintaktiskā diagramma 4.1).

Tā kā dinamiskajam masīvam atmiņa tikusi izdalīta manuāli (nevis automātiski), arī iznīcināšana jāveic manuāli ar komandas *delete* palīdzību (pirmkods 4.2, rinda 16; sintaktiskā diagramma 4.7).

Sintakse 4.7. ***dynamic array deletion*** **(dinamiska masīva iznīcināšana)**

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

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

- masīva deklarēšana (sintaktiskā diagramma 4.5),
- masīva izveidošana ar komandu *new* (sintaktiskā diagramma 4.6),
- masīva izmantošana – tāpat kā statiskam masīvam (sintaktiskā diagramma 4.1),
- masīva iznīcināšana ar komandu *delete* (sintaktiskā diagramma 4.7).

4.4. Masīva izmantošanas īpatnības

Masīvs no izmantošanas viedokļa uzskatāms kā atsevišķu elementu kopums, nevis vienots veselums, tāpēc, darbojoties ar to, jāstrādā ar katru elementu atsevišķi. Tas jāņem vērā tādās darbībās kā masīvu aizpildīšana, salīdzināšana un garuma noteikšana.

Masīva aizpildīšana.

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

```
arr2 = arr1;
```

Masīva aizpildīšana jāveic kādā no šiem diviem variantiem:

- aizpildot pa vienam elementam (piemēram, pirmkods 4.1, rinda 9),
- veicot masīva inicializāciju pie deklarēšanas (piemēram, pirmkods 4.1, rinda 7).

Masīvu salīdzināšana.

Divus masīvus **nevar** salīdzināt, piemērojot salīdzināšanas operatoru masīva mainīgajiem:

```
(arr2 == arr1)
```

Salīdzināšana jāveic pa vienam elementam.

Masīva garuma noteikšana.

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

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

```
sizeof (static_array) / sizeof (array_elem_type)
```

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

4.5. Zema līmeņa simbolu virknes

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

Lielākā daļa moderno universālo programmēšanas valodu nodrošina t.s. augsta līmeņa simbolu virknes, t.i., tādas, kas ietver sevī arī atmiņas vadību. Arī valodā C++ ir pieejamas augsta līmeņa simbolu virknes (tips *string*), tomēr valodā C++

- zema līmeņa simbolu virknes mantojumā no valodas C ir pamata (noklusētais) variants,
- augsta līmeņa simbolu virknes tikušas standartizētas salīdzinoši vēlu, tāpēc diezgan ilgi pēc C++ rašanās praktiski vienīgās bija tieši zema līmeņa simbolu virknes.

Zema un augsta līmeņa simbolu virknes mēdz saukt arī attiecīgi par C un C++ stila simbolu virknēm.

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

Zema līmeņa (C stila) simbolu virknes valodā C++ ir tādi masīvi, kurus raksturo divas papildus īpašības:

- datu tips, no kura sastāv masīvs, ir **simbols** (*char*),
- vismaz viens no masīva simboliem ir t.s. **simbolu virknes beigu simbols** jeb nulles simbols ('\0' jeb 0).

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

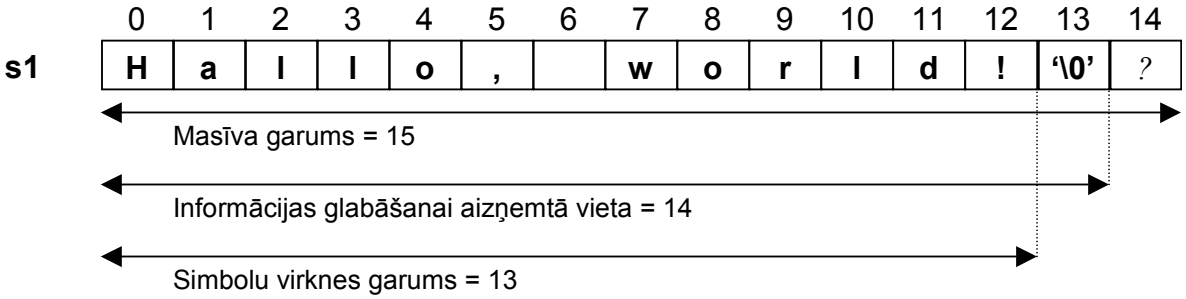
Neatkarīgi no simbolu virknes tipa (statisks, dinamisks) un no deklarēšanas veida (sintakses, sk. masīva deklarēšanu), simbolu virknes datu tips ir *char**, kas nozīmē arī "norāde uz *char*". Par masīvu, t.sk. simbolu virkņu saistību ar jēdzienu "norāde" (*pointer*) skatīt tālākās nodaļās.

Simbolu virkņu deklarēšana un izmantošana ir veicama tāpat kā parastiem masīviem, jo simbolu virknes pašas ir masīvi. Konstantu teksta informāciju, kas atbilst simbolu virknes datu tipam, var uzdot, ievietojot dubultpēdiņās, piemēram, "simboli".

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

Nedrīkst sajaukt jēdzienus: **simbols** (piemēram, *'a'*) no **simbolu virkne garumā 1** (piemēram, *"a"*), kaut arī tie izdrukājas identiski – simbolu virkne automātiski aiz sevis ietver arī beigu simbolu (*"a"* ir *{'a','\0'}*).

```
char s1[15] = "Hallo, world!";
char s1[15] = 'H','a','l','l','o',',',' ','w','o','r','l','d','!','\0';
```



Attēls 4.2. Simbolu virkne "Hallo, world!", kas tiek glabāta masīvā ar garumu 15

Masīvā aiz simbolu virknes beigu simbola var atrasties informācija, tomēr no simbolu virknes viedokļa tai nav nozīmes.

Standarta funkcijas simbolu virkņu apstrādei.

Uz simbolu virknēm attiecas tās pašas izmantošanas īpatnības, kas uz masīviem kopumā – tiem nevar veikt piešķiršanu, salīdzināšanu un garuma noteikšanu, izmantojot parastos operatorus, tomēr simbolu virknēm ir pieejams standarta funkciju klāsts. Tālāk uzskaitītas dažas no tām (sk. arī nodaļu 4.4). Standarta funkcijām, kas strādā ar zema līmeņa simbolu virknēm, raksturīgs tas, ka to nosaukums sākas ar "str". Tās ir ietvertas bibliotēkā <*string.h*>, tomēr netiešā veidā arī daudzās citās, piemēram <*iostream*> (sk. pirmkodu 4.3). Modifikators *const* pie parametra norāda, ka funkcija doto parametru noteikti neizmainīs, bet *char** ir datu tips, kas apzīmē simbolu virkni (precīzāk, norādi uz simbolu virkni).

strcpy

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

Funkcija *strcpy()* kopē simbolu virknes *s1* saturu uz virkni *s2*. Jābūt nodrošinātam, ka virknē *s2* pietiks vietas. Sk. pirmkodu 4.3, rindu 18.

strcat

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

Funkcija *strcat()* pievieno simbolu virknes *s1* kopiju virknes *s2* beigās. Jābūt nodrošinātam, ka virknē *s2* pietiks vietas. Sk. pirmkodu 4.3, rindu 21.

strcmp

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

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

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

Sk. arī pirmkodu 4.3, rindas 29, 31.

strlen

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

Funkcija *strlen()* atgriež simbolu virknes *s* garumu. Sk. pirmkodu 4.3, rindas 24-28.

Simbolu virknes aizpildīšanai no klaviatūras vai testa faila der funkcija *getline()* (standarta bibliotēka <*iostream.h*>). Tālāk parādīts šīs funkcijas vienkāršots variants.

getline

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

Funkcija *getline()* no faila (klaviatūras) lasa simbolus masīvā *buf*, kamēr vai nu nav nolasīti *num* simboli, vai nu tiek sastapts simbols *'\n'* (ievadīts <ENTER>). Tā kā masīvā jārezervē viena vieta simbolu virknes beigu simbolam, tad no klaviatūras nevar nolasīt vairāk nekā *num-1* simbolu – pārējie tiek ignorēti. Svarīgi, ka funkcija *getline()* nav neatkarīga, bet ir ievades objekta (piemēram, *cin*) sastāvdaļa. Sk. pirmkodu 4.3, rindas 23.

Bez nosauktajām 5 funkcijām ir pieejamas arī daudzas citas.

Pirmkods 4.3. Zema līmeņa simbolu virknes (*arr2string.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 int main ()
05 {
06     char s1[15] = "Hallo, world!";
07     char s2[20];
08     char *s3 = new char[18];
09     char s4[] = {'H','A','L','L','O',',',' ',
                    'W','O','R','L','D','!','\0'};
10     char s5[20];
11     int i=0;
12     while (s1[i]!='\0')
13     {
14         s2[i] = s1[i];
15         i++;
16     };
17     s2[i] = '\0'; // s2[i] = 0;
18     strcpy (s3, s2);
19     s3[1] = 'e';
20     s3[7] = 'W';
21     strcat (s3, "!!");
22     cout << "Input text: ";
23     cin.getline (s5, 20);
24     cout << "s1=" << s1 << " (" << strlen(s1) << ")" << endl;
25     cout << "s2=" << s2 << " (" << strlen(s2) << ")" << endl;
26     cout << "s3=" << s3 << " (" << strlen(s3) << ")" << endl;
27     cout << "s4=" << s4 << " (" << strlen(s4) << ")" << endl;
28     cout << "s5=" << s5 << " (" << strlen(s5) << ")" << endl;
29     if (strcmp(s1,s2)==0) cout << "s1 == s2" << endl;
30     else cout << "s1 <> s2" << endl;
31     if (strcmp(s2,s3)==0) cout << "s2 == s3" << endl;
32     else cout << "s2 <> s3" << endl;
33     delete[] s3;
34     return 0;
35 }
```

Programmas darbības piemērs:

```
Input text: this is a string.
s1=Hallo, world! (13)
s2=Hallo, world! (13)
```

```
s3=Hello, World!!! (15)
s4=HALLO, WORLD! (13)
s5=this is a string. (17)
s1 == s2
s2 <> s3
```

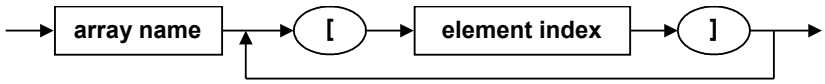
Komentāri pie pirmkoda piemēra 4.3.

- Simbolu virknes inicializēšanai pie deklarēšanas pieejami divi varianti – pa simbolam (rinda 9) un ar teksta konstanti (rinda 6).
- Rindās 11-17 parādīta informācijas pārkopēšana no virknes *s1* uz *s2*, kas atbilst tam, ko veic funkcija *strcpy()* (rinda 18).
- Simbolu virknes beigu simbols programmas tekstā uzrādāms vai nu kā simbols *'\0'*, vai kā skaitliska vērtība *0* (rinda 17).
- Rindā 21 simbolu virknei *s3* galā tiek pieliktas divas izsaukuma zīmes.
- Rindās 29-32 tiek salīdzināt attiecīgi simbolu virknes *s1* un *s2*, un *s2* un *s3*, izvadot uz ekrāna attiecīgo rezultātu.
- Atšķirībā no parastiem masīviem, simbolu virkņu attēlošanai uz ekrāna drīkst pielietot izdrukas operatoru << (rindas 24-28).

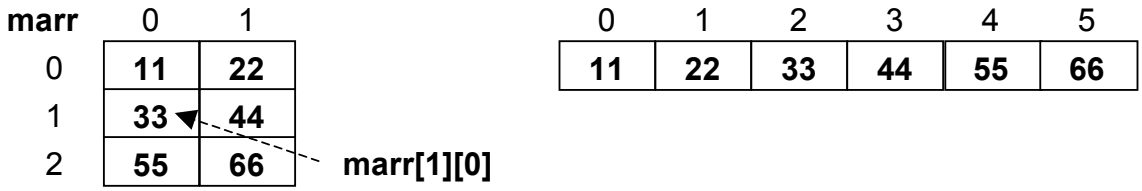
4.6. Daudzdimensiju masīvi

Daudzdimensiju masīvi ir tādi masīvi, kuros elementus identificē **divi vai vairāki indeksi**. Kaut arī jebkuru daudzdimensiju masīvu var izteikt kā viendimensijas masīvu, tomēr daudzdimensiju masīvu izmantošana var būtiski uzlabot programmas pārskatāmību. Pievienojot jēdzienu par daudzdimensiju masīviem, vēršanās pie masīva elementa sintakse aprakstāma šādi:

Sintakse 4.8. ***array element 2*** **(masīva elements 2)**



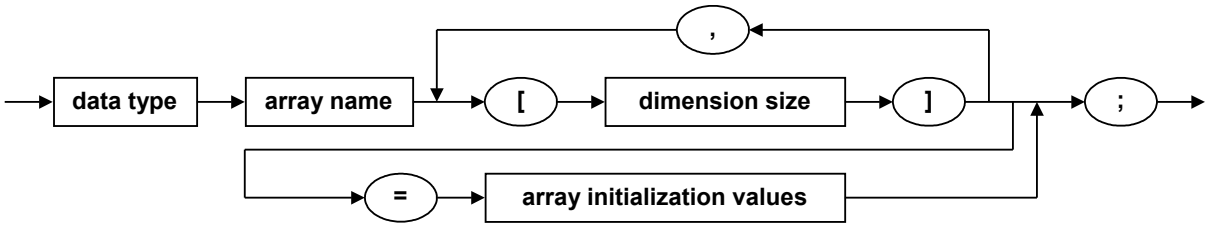
Statisks daudzdimensiju masīvs atmiņā attēlojas tieši tāpat kā viendimensijas masīvs (sk. attēlu 4.3). Ar dinamisku daudzdimensiju masīvu ir savādāk – lai arī izmantošana ir tāda pati kā statiskajam, tomēr būtiski atšķiras tā izveidošana un iznīcināšana, bez tam arī izvietojums atmiņā.



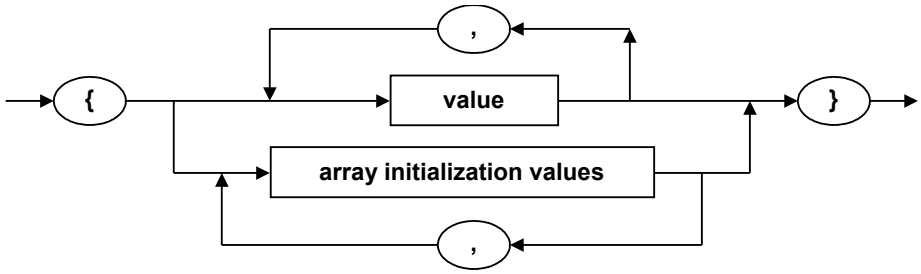
Attēls 4.3. Divdimensiju masīvs (3x2) un tā reālais izvietojums atmiņā

Arī daudzdimensiju masīvu iespējams inicializēt pie deklarēšanas (sintaktiskās diagrammas 4.9, 4.10), turklāt uzskatāmībai iespējams izmantot papildus strukturēšanu blokos (pirmkods 4.4, rinda 7), tomēr tas nav obligāti (pirmkods 4.4, rinda 8).

Sintakse 4.9. ***static array declaration with initialization 3*** **(statiska masīva deklarēšana ar inicializāciju 3)**



Sintakse 4.10. ***array initialization values*** **(masīva inicializācijas vērtības)**



Pirmkods 4.4. Divdimensiju masīvi (*arr3multi.cpp*)

```
01 #include <iostream>
02 using namespace std;
03 const int dim1 = 3, dim2 = 2;
04
05 int main ()
06 {
07     int marr1[dim1][dim2] = {{11,22},{33,44},{55,66}};
08     int marr2[dim1][dim2] = {11,222,333,44,555,66};
09     for (int i=0; i<dim1; i++)
10         for (int k=0; k<dim2; k++)
11         {
12             cout << i << ',' << k << ": ";
13             cout << marr1[i][k];
14             if (marr1[i][k] == marr2[i][k]) cout << " == ";
15             else cout << " <> ";
16             cout << marr2[i][k] << endl;
17         }
18     return 0;
19 }
```

Programmas darbības piemērs:

```
0,0: 11 == 11
0,1: 22 <> 222
1,0: 33 <> 333
1,1: 44 == 44
2,0: 55 <> 555
2,1: 66 == 66
```

Pirmkoda piemērs 4.4 demonstrē programmu, kurā tiek inicializēti un pēc tam izdrukāti divi divdimensiju masīvi, pie reizes veicot to salīdzināšanu pa elementam. Atbilst attēlam 4.3.