8. Datu struktūras

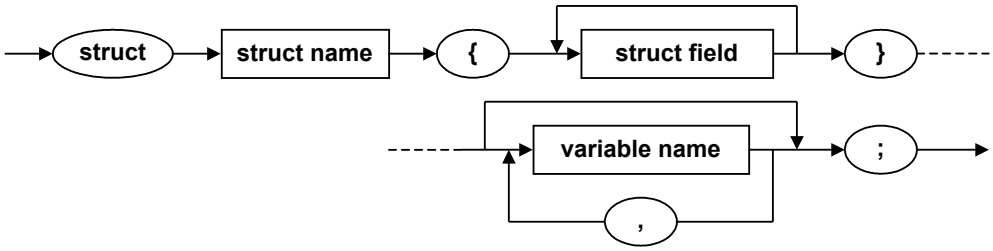
Nodaļas saturs:

8.1. Konstrukcija struct

No saliktiem datu tipiem līdz šim ticis apskatīts tikai masīvs, kas ir vienāda tipa elementu kopums. Šajā nodaļā aprakstītā konstrukcija *struct* nodrošina dažādu tipu mainīgo apvienošanu kopējā datu tipā – kopējā struktūrā.

Struktūras definēšana ar *struct* ir lauku uzskaitījums figūriekavu blokā, kam dots kopējs vārds (sintakse 8.1; pirmkods 8.1, rindas 4-8). Katrs lauks sintaktiski tiek definēts līdzīgi kā tiek deklarēts mainīgais. Struktūras lauki var būt gan ar vienkāršu datu tipu, gan saliktu, piemēram, masīvi vai citas struktūras.

Sintakse 8.1. ***structure definition*** **(struktūras definēšana)**



Struktūru objekti (mainīgie) var programmā parādīties divos veidos, kas atšķiras pēc sintakses un atmiņas izmantošanas:

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

Struktūras izmantošana tiešā veidā.

Mainīgā tips ir struktūras tips (pirmkoda piemērs 8.1, rinda 12) un piekļuve pie struktūras laukiem notiek, izmantojot **punkta notāciju** (rindas 13,15). Objekts tiek automātiski izveidots pie mainīgā deklarēšanas un automātiski beidz pastāvēt programmas bloka beigās.

Struktūras izmantošana dinamiskā veidā.

Mainīgā tips ir norāde uz struktūras tipu (pirmkoda piemērs 8.1, rinda 16) un piekļuve pie struktūras laukiem notiek, izmantojot **bultiņas notāciju** (rinda 17). Objekts tiek izveidots, izmantojot operatoru *new* (rinda 16) un likvidēts ar *delete* (rinda 20).

Pirmkoda piemērs 8.1 parāda ar *struct* konstrukciju veidotu struktūru izmantošanu gan tiešā, gan dinamiskā veidā. Piemēru ilustrē attēls 8.1.

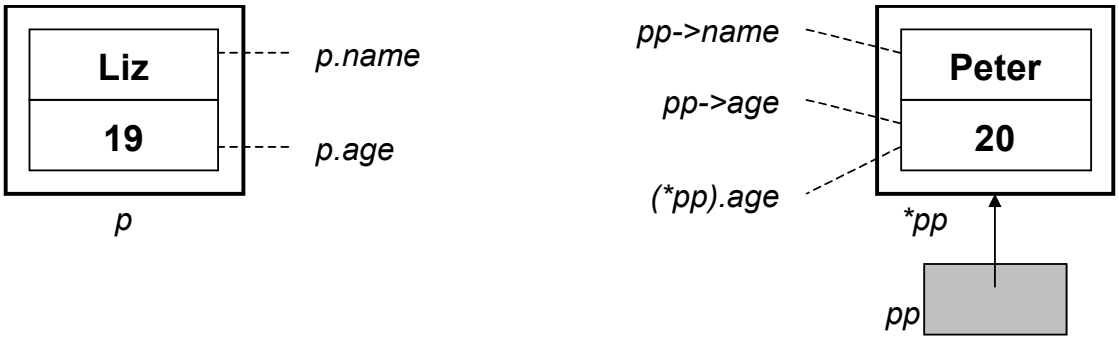
Pirmkods 8.1. Konstrukcija *struct* (*dst1struct.cpp*)

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

```
05 {
06     char name[20];
07     int age;
08 };
09
10 int main ()
11 {
12     person p;
13     strcpy (p.name, "Liz");
14     p.age = 19;
15     cout << p.name << " " << p.age << endl;
16     person *pp = new person;
17     strcpy (pp->name, "Peter");
18     pp->age = 20;
19     cout << pp->name << " " << (*pp).age << endl;
20     delete pp;
21     return 0;
22 }
```

Programmas darbības piemērs:

```
Liz 19
Peter 20
```



Attēls 8.1. Ar konstrukciju struct veidotas struktūras izmantošana (atbilst pirmkoda piemēram 8.1)

Struktūras objektus var izmantot tiešā veidā vai dinamiskā veidā, tieši tāpat struktūru objektus var nodot funkcijai caur parametru – kā norādes vai tiešā veidā (t.sk. caur referenci), turklāt objekta nodošanas veidu var mainīt atkarībā no funkcijas parametra tipa, caur kuru objekts jānodod. Šo principu demonstrē pirmkoda piemērs 8.2 (sk. arī attēlu 8.2).

Pirmkods 8.2. Ar *struct* veidotas struktūras nodošana caur parametru (*dst2struct.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 struct person
05 {
06     char name[20];
07     int age;
08 };
09
10 void print1 (person *x)
11 {
12     cout << x->name << " " << x->age << endl;
13 };
```

```
14
15 void print2 (person &y)
16 {
17     cout << y.name << " " << y.age << endl;
18 };
19
20 int main ()
21 {
22     person p;
23     strcpy (p.name, "Liz");
24     p.age = 19;
25     print1 (&p);
26     print2 (p);
27     person *pp = new person;
28     strcpy (pp->name, "Peter");
29     pp->age = 20;
30     print1 (pp);
31     print2 (*pp);
32     delete pp;
33     return 0;
34 }
```

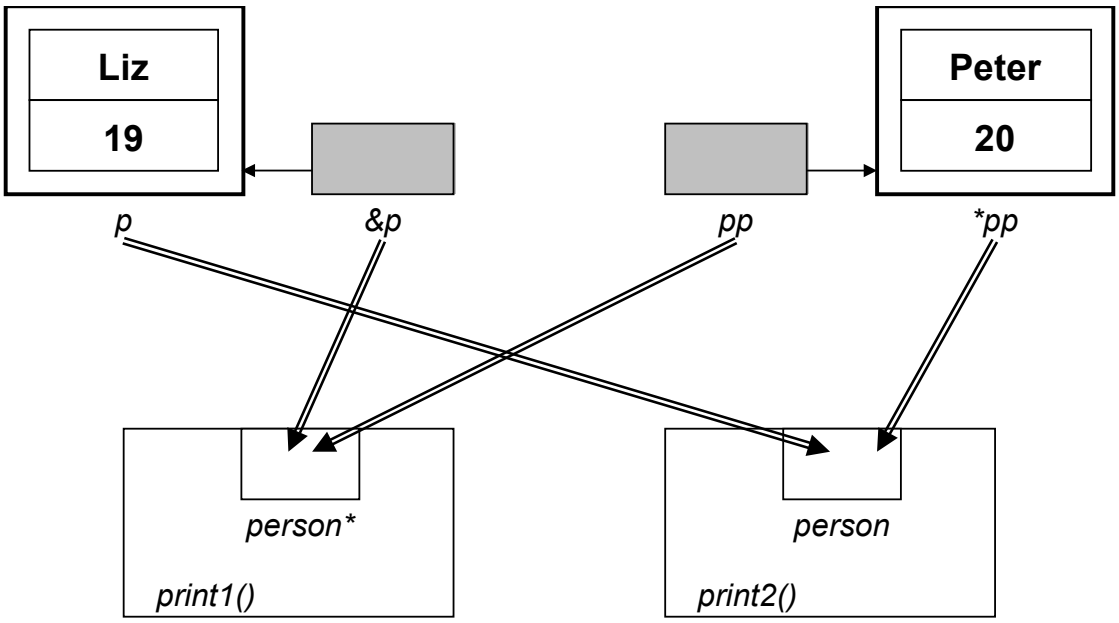
Programmas darbības piemērs:

```
Liz 19
Liz 19
Peter 20
Peter 20
```

Komentāri pie pirmkoda piemēra 8.2.

- Piemērā demonstrēti divi struktūras *person* objekti – p (tiešā veidā, rinda 22), pp (dinamiskais, rinda 27).
- Definētas divas izdrukāšanas funkcijas – *print1* (pieņem norādi caur parametru, rindas 10-13) un *print2* (pieņem objektu tiešā veidā, rindas 15-18).
- 25. rindā tiešā veidā definēts objekts tiek padots caur norādes parametru, paņemot objekta adresi.
- 26. rindā tiešā veidā definēts objekts tiek padots caur parametru, kas pieņem objektu tiešā veidā.
- 30. rindā objekta norāde tiek padota caur norādes parametru.
- 31. rindā norādes mainīgais tiek lietots, lai padotu objektu tiešā veidā.
- 15. rindā parametrs varētu būt parametrs bez references (&), bet tādā gadījumā, padodot objektu funkcijai, tā saturs tiktu dublēts funkcijas vajadzībām, tomēr funkcionalitāte nemainītos:

```
void print2 (person y)
```



Attēls 8.2. Struktūras objekta padošana caur parametru (atbilst pirmkoda piemēram 8.2)

8.2. Masīvs no struktūrām

Viens no veidiem, ka apvienot struktūru objektus, ir masīvi. Ņemot vērā to, ka struktūru objekti C++ programmā var parādīties 2 veidos, tā arī apkopošana masīvos iespējama divos veidos:

- masīvs no struktūras objektiem (pirmkods 8.3),
- masīvs no norādēm uz struktūras objektiem (pirmkods 8.4).

Pirmkoda piemēri 8.3 un 8.4 parāda objektu izvietošanu masīvos atbilstoši diviem objekta parādīšanās veidiem, turklāt piemērā 8.3 tiek lietots statisks masīvs, bet piemērā 8.4 – dinamisks masīvs. Abi piemēri no funkcionālā viedokļa ir identiski.

Pirmkods 8.3. Masīvs no struktūrām (*dst3structarr.cpp*)

```
01 #include <iostream>
02 using namespace std;
03 const int arr_size = 3;
04
05 char person_names[arr_size][20]={"Liz","Ann","Peter"};
06 int person_ages[arr_size]={19,22,20};
07
08 struct person
09 {
10     char name[20];
11     int age;
12 };
13
14 int main ()
15 {
16     person parr[arr_size];
17     for (int i=0; i<arr_size; i++)
18     {
19         strcpy (parr[i].name, person_names[i]);
```

```
20          parr[i].age = person_ages[i];
21      };
22      for (int i=0; i<arr_size; i++)
23      {
24          cout << parr[i].name << " " << parr[i].age << endl;
25      };
26      return 0;
27  }
```

Programmas darbības piemērs:

```
Liz 19
Ann 22
Peter 20
```

Attēls 8.3. Struktūras objektu ievietošana statiskā masīvā tiešā veidā (atbilst pirmkoda piemēram 8.3)

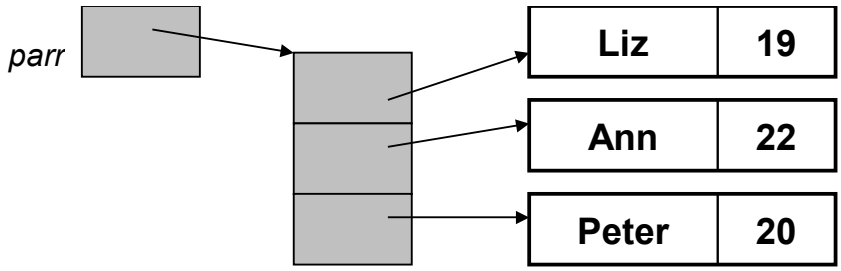
Pirmkods 8.4. Dinamisks masīvs no struktūrām (*dst4structdarr.cpp*)

```
01 #include <iostream>
02 using namespace std;
03 const int arr_size = 3;
04
05 char person_names[arr_size][20]={"Liz","Ann","Peter"};
06 int person_ages[arr_size]={19,22,20};
07
08 struct person
09 {
10      char name[20];
11      int age;
12 };
13
14 int main ()
15 {
16      person **parr;
17      parr = new person*[arr_size];
18      for (int i=0; i<arr_size; i++)
19      {
20          parr[i] = new person;
21          strcpy (parr[i]->name, person_names[i]);
22          parr[i]->age = person_ages[i];
23      };
24      for (int i=0; i<arr_size; i++)
25      {
26          cout << parr[i]->name << " " << parr[i]->age << endl;
27          delete parr[i];
28      };
29      delete[] parr;
30      return 0;
31  }
```

Programmas darbības piemērs:

```
Liz 19
```

```
Ann 22
Peter 20
```



Attēls 8.4. Struktūras objektu ievietošana dinamiskā masīvā no norādēm (atbilst pirmkoda piemēram 8.4)

8.3. Dinamiskas datu struktūras

Dinamiska datu struktūra (*dynamic data structure*) ir datu struktūra, kas programmas darbības laikā ļauj elastīgi mainīt tās izmēru.

Dinamisks masīvs ir viena no ērtākajām zema līmeņa dinamiskajām konstrukcijām, tomēr viens no lielākajiem tā mīnusiem ir relatīvi lielais nepieciešamais atmiņas pārrakstīšanas apjoms masīva izmēra izmaiņas gadījumā (masīvu nevar pagarināt, bet jāveido jauns ar vajadzīgo izmēru un jāpārkopē visi dati uz jauno vietu).

"Īsti" dinamiskas datu struktūras strādā pēc cita – ķēdēšanas principa, nodrošinot to, ka katra jauna elementa pievienošana vai izmešana neprasa veikt daudz izmaiņu atmiņā.

Dinamiska datu struktūra sastāv no elementiem, kur katrs elements vispārīgā gadījumā nodrošina divas funkcijas un tādējādi sastāv no divām daļām:

- dati (piemēram, pirmkods 8.5, rinda 6),
- saites uz citiem elementiem (piemēram, pirmkods 8.5, rinda 7).

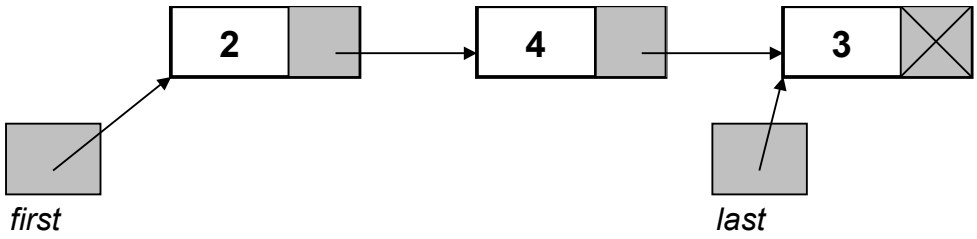
Vienkāršākajā gadījumā saišu daļa no vienas saites – norādes uz nākošo elementu, tādējādi veidojot lineāru struktūru – **saistīto sarakstu** (*linked list*). Katrs elements saistītajā sarakstā vai nu norāda uz nākošo elementu vai satur norādi ar tukšo adresi *NULL* (ja ir pēdējais) (attēls 8.5).

Bez elementiem, kas veido struktūras pamatmasu, saistītais saraksts ietver šādas komponentes:

- norāde uz pirmo elementu,
- norāde uz pēdējo elementu (neobligāta, paredzēta, lai tehniski atvieglotu saraksta papildināšanu galā.

Saraksts var būt tukšs (bez elementiem), to norāda tukša norāde uz pirmo elementu (sākot darbu ar sarakstu, šai norādei obligāti jābūt inicializētai).

Standarta variantā katrs jaunais elements tiek likts **saistītā saraksta galā**, līdz ar to saraksta elementu secība atbilst tā izpildīšanas secībai.



Attēls 8.5. Saistīta saraksta vispārīgā shēma (atbilst pirmkoda piemēriem 8.5 un 8.6)

Viena elementa pievienošana saistītajam sarakstam (pirmkods 8.5, rindas 17-28) sastāv no šādiem darbību blokiem:

- elementa izveidošana un aizpildīšana ar datiem (rindas 17-19),
- izveidotā elementa pievienošana sarakstam (rindas 20-28).

Izveidotā elementa pievienošana atšķiras tukšam sarakstam (rindas 21-23) un netukšam sarakstam (rindas 25-28):

- tukšam sarakstam pievienots elements kļūst reizē par pirmo un arī par pēdējo elementu,
- pievienojot elementu netukšam sarakstam, pirmais elements netiek mainīts.

Pirmkoda piemērā 8.5 parādītā programma ļauj ievadīt veselus skaitļus no klaviatūras, veidojot saistīto sarakstu ar šādām vērtībām. Pēc tam saraksts tiek izdrukāts pa vienam elementam (rindas 31-34) un iznīcināts (rindas 35-41).

Pirmkods 8.5. Dinamiska datu struktūra – saistītais saraksts (*dst5linkedlist.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 struct elem
05 {
06     int num;
07     elem *next;
08 };
09
10 int main ()
11 {
12     elem *first=NULL, *last=NULL, *p;
13     int i;
14     cin >> i;
15     while (i != 0)
16     {
17         p = new elem;
18         p->num = i;
19         p->next = NULL;
20         if (first == NULL)
21         {
22             first = last = p;
23         }
24         else
25         {
26             last->next = p;
27             last = last->next;
28         };
29         cin >> i;
```

```
30      };
31      for (p = first; p!=NULL; p=p->next)
32      {
33          cout << p->num << endl;
34      };
35      p = first;
36      while (p!=NULL)
37      {
38          first = first->next;
39          delete p;
40          p = first;
41      };
42      return 0;
43  }
```

Programmas darbības piemērs:

```
2
4
3
0
2
4
3
```

Atšķirībā no masīva, kur katram elementam var piekļūt uzreiz, uzrādot indeksu, pie saistīta saraksta elementiem var piekļūt tikai, secīgi pārstaigājot sarakstu, sākot ar pirmo elementu.

Tipiska darbība ar daudzām datu struktūrām ir to pārstaigāšana pa vienam elementam, veicot noteiktu darbību ar katru elementu (attēls 8.6).

```
aiziet uz saraksta sākumu
WHILE (nav saraksta beigas)
    apstrādā kārtējo elementu
    pāriet uz nākošo elementu
END WHILE
```

Attēls 8.6. Datu struktūru (t.sk. saistīta saraksta) secīgas pārstaigāšanas shēma

Saistīta saraksta standarta pārstaigāšanas secīgas shēmai atbilst saistītā saraksta izdrukāšana, kā arī daļēji atbilst saistītā saraksta likvidēšana pirmkoda piemērā 8.5.

Pirmkoda piemērs 8.6 ir funkcionāli identisks piemēram 8.5. Vienīgā atšķirība ir tā, ka šeit galvenās darbības ar sarakstu ir iznestas atsevišķās funkcijās.

Pirmkods 8.6. Saistītais saraksts ar funkciju izmantošanu (*dst6linkedlist.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 struct elem
05 {
06     int num;
07     elem *next;
```

```
08 };
09
10 void add_element (elem *&first, elem *&last, int i)
11 {
12     elem *p = new elem;
13     p->num = i;
14     p->next = NULL;
15     if (first == NULL)
16     {
17         first = last = p;
18     }
19     else
20     {
21         last->next = p;
22         last = last->next;
23     }
24 };
25
26 void print_list (elem *first)
27 {
28     elem *p = first;
29     while (p!=NULL)
30     {
31         cout << p->num << endl;
32         p = p->next;
33     };
34 };
35
36 void delete_list (elem *&first)
37 {
38     elem *p = first;
39     while (p!=NULL)
40     {
41         first = first->next;
42         delete p;
43         p = first;
44     };
45     first = NULL;
46 };
47
48 int main ()
49 {
50     elem *first=NULL, *last;
51     int i;
52     cin >> i;
53     while (i != 0)
54     {
55         add_element (first, last, i);
56         cin >> i;
57     };
58     print_list (first);
59     delete_list (first);
60     return 0;
61 }
```

Programmas darbības piemērs:

```
2
4
3
0
2
4
3
```

Komentāri pie pirmkoda piemēra 8.6.

- Funkcija *add_element()* pievieno jaunu elementu saraksta galā. Funkcijai tiek padota norāde uz pirmo elementu, norāde uz pēdējo elementu un veidojamā elementa vērtība.
- Funkcija *print_list()* izdrukā saraksta saturu uz ekrāna.
- Funkcija *delete_list()* izdzēš sarakstu (atbrīvo atmiņu).
- Dinamisku datu struktūru padod funkcijai caur parametru kā norādi uz tās pirmo elementu (pirmais parametrs funkcijās *add_element()*, *print_list()*, *delete_list()*). Tas tehniski atbilst masīva nodošanai caur parametru.
- Parametra tips saraksta nodošanai ir *elem**, kas nozīmē "norāde uz *elem*".
- Funkcija *add_element()* kā parametru pieņem arī norādi uz saraksta pēdējo elementu, kas atvieglo elementa pievienošanas procedūru saraksta beigās. Šī parametra tips arī ir *elem**.
- Vairāki no norādes parametriem (uz saraksta sākumu vai beigām) ir parametri references (*elem*&*), kas nozīmē, ka funkcijas darbības laikā dotais parametrs (norāde) var mainīties, un šīs izmaiņas jānodod atpakaļ izsaucošajam modulim.
- Funkcijas *add_element()* darbības laikā vienmēr mainīsies norāde uz pēdējo elementu (*last*), bet uz pirmo elementu (*first*) tikai tad, ja pievienošana notiks tukšam sarakstam.
- Funkcijas *delete_list()* darbības laikā norāde uz pirmo elementu mainīsies gandrīz vienmēr, izņemot gadījumu, kad saraksts jau ir tukšs.
- Funkcijas *print_list()* darbības laikā norāde uz saraksta pirmo elementu nemainās, tāpēc parametrs nav reference (*&*), bet ir vienkārši norāde uz saraksta sākumu (*elem**).
- Ja tiktu veidota tāda funkcija, kas **izmaina** elementu **vērtības**, tomēr **neizmaina norādi** uz pirmo elementu (piemēram, funkcija, kas visu elementu vērtības izmaina par 1), arī tad nebūtu nepieciešama norādes nodošana caur referenci.
- Funkcija *delete_list()* darbības beigās saraksta sākuma norādei piešķir vērtību *NULL*, atgriežot to caur referenci. Ja darbs ar saraksta norādi (*first*) pēc šīs funkcijas darbināšanas neturpinās, kā tas ir šajā gadījumā (rinda 59), tad NULL vērtību teorētiski varētu nepiešķirt (rinda 45) un parametru caur referenci nenodot (36).

Pirmkoda piemērā 8.7 papildus realizēta funkcija *insert_element()*, kas tāpat kā *add_element()* pievieno elementu sarakstam, tomēr, to veic noteiktā pozīcijā un tikai tad, ja tas ir iespējams (piemēram, nav iespējams tukšam sarakstam pievienot elementu tā, lai tas būtu 2. pozīcijā).

Funkcija *insert_element()* sastāv no 2 daļām – (1) pos = 1, jaunais elements tiek likts saraksta sākumā (rindas 35-40); (2) pos <> 1, ja tāda pozīcija jaunajam elementam ir iespējama, tad tas tur tiek ievietots, citādi jauns elements vispār netiek izveidots (rindas 41-55). Tehniski atšķirība ir tieši tajā apstāklī, vai jaunais elements tiek likts sākumā vai nē. Ja jauno elementu neliek sākumā, tad pirms ievietošanas ir jāatrod elements, kas atrodas **pirms** tā paredzētās atrašanās vietas (rindas 45-48), tad notiek pati ievietošana (rindas 49-54, uz iepriekšējo elementu norāda *q*). Elements tiek pievienots tikai tad, ja ir atrasta pareizā pozīcija (to nosaka

nosacījums rindā 49). Elements netiek pievienots, ja *pos<1* vai *pos>n+1*, kur *n* ir esošā saraksta garums.

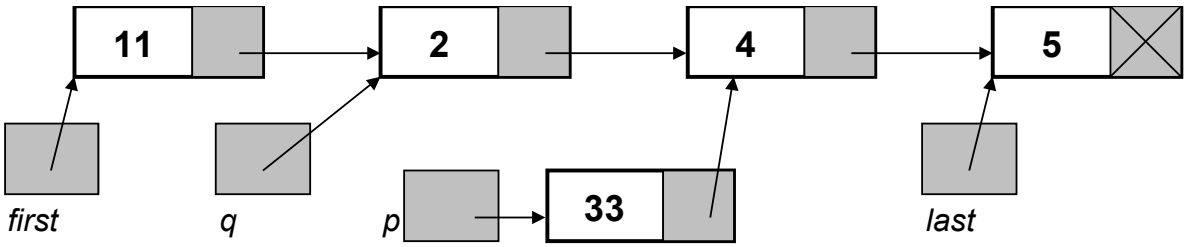
Pirmkods 8.7. Elementa iespraušana saistītajā sarakstā (*dst7linkedlist.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 struct elem
05 {
06     int num;
07     elem *next;
08 };
09
10 elem* create_element (int i)
11 {
12     elem *p = new elem;
13     p->num = i;
14     p->next = NULL;
15     return p;
16 };
17
18 void add_element (elem *&first, elem *&last, int i)
19 {
20     elem *p = create_element (i);
21     if (first == NULL)
22     {
23         first = last = p;
24     }
25     else
26     {
27         last->next = p;
28         last = last->next;
29     }
30 };
31
32 void insert_element (elem *&first, elem *&last, int i, int pos)
33 {
34     elem *p;
35     if (pos == 1)
36     {
37         p = create_element (i);
38         p -> next = first;
39         first = p;
40     }
41     else if (pos > 1 && first != NULL)
42     {
43         elem *q = first;
44         int curr_pos;
45         for (curr_pos = 2; curr_pos < pos && q->next != NULL; q =
                q->next)
46         {
47             curr_pos++;
48         };
49         if (curr_pos == pos)
```

```
50          {
51              p = create_element (i);
52              p->next = q->next;
53              q->next = p;
54          }
55      }
56  };
57
58  void print_list (elem *first)
59  {
60      elem *p = first;
61      while (p!=NULL)
62      {
63          cout << p->num << endl;
64          p = p->next;
65      };
66  };
67
68  void delete_list (elem *&first)
69  {
70      elem *p = first;
71      while (p!=NULL)
72      {
73          first = first->next;
74          delete p;
75          p = first;
76      };
77      first = NULL;
78  };
79
80  int main ()
81  {
82      elem *first=NULL, *last;
83      add_element (first, last, 2);
84      add_element (first, last, 4);
85      add_element (first, last, 5);
86      insert_element (first, last, 11, 1);
87      insert_element (first, last, 33, 3);
88      insert_element (first, last, 66, 6);
89      insert_element (first, last, 88, 8);
90      print_list (first);
91      delete_list (first);
92      return 0;
93  }
```

Programmas darbības piemērs:

```
11
2
33
4
5
66
```

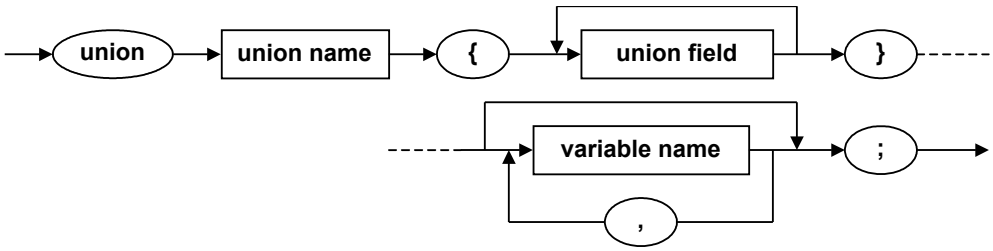


Attēls 8.7. Elementa iespraušana sarakstā (atbilst pirmkoda piemēram 8.7, konfigurācija pēc rindas 52, izsaucot funkciju rindā 87)

8.4. Konstrukcija union

Apvienojums *union* ir īpašs struktūras veids, kurā visi mainīgie izmanto vienu un to pašu atmiņas apgabalu. Apvienojuma izmērs ir vienāds ar lielākā lauka izmēru.

Sintakse 8.2. *union definition* (apvienojuma definēšana)



Apvienojumu parasti izmanto, lai noteiktu atmiņas apgabalu pēc vajadzības izmantotu dažādiem nolūkiem (pirmkods 8.8, rindas 13-16), tomēr teorētiski var izmantot arī apvienojuma doto iespēju apskatīt vienu un to pašu atmiņas apgabalu no dažādu tipu aspektiem (rindas 17-21).

Pirmkods 8.8. Konstrukcija *union* (*dst8union.cpp*)

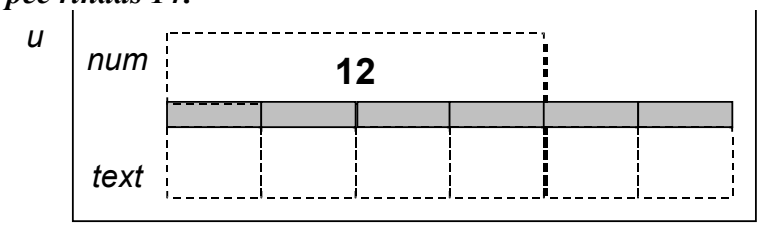
```
01 #include <iostream>
02 using namespace std;
03
04 union unelem
05 {
06     int num;
07     char text[6];
08 };
09
10 int main ()
11 {
12     unelem u;
13     u.num = 12;
14     cout << u.num << endl;
15     strcpy (u.text, "Hello");
16     cout << u.text << endl;
17     u.text[0] = 'A';
18     u.text[1] = '\0';
19     u.text[2] = '\0';
20     u.text[3] = '\0';
21     cout << u.num << endl;
22     return 0;
23 }
```

Programmas darbības piemērs:

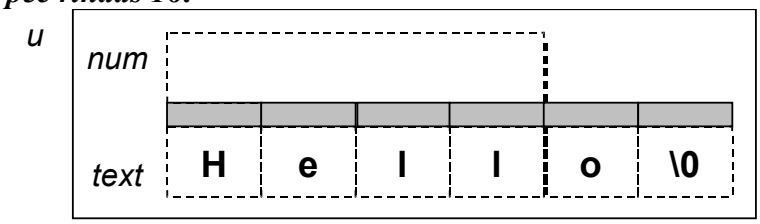
```
12
Hello
65
```

Programmas darbības demonstrācija (pirmkoda piemērs 8.8).

Konfigurācija pēc rindas 14:



Konfigurācija pēc rindas 16:



Konfigurācija pēc rindas 21: