10.Bināru failu apstrāde

Nodaļas saturs:

10.1.Vienkāršākās faila binārās apstrādes operācijas

Ar **bināru failu** tiek saprasts fails, kurš nav teksta fails, respektīvi, tāds, kurš satur arī neteksta datus. Tāds fails būtu jāapstrādā bināri, un no programmētāja viedokļa svarīgāks jēdziens par bināru failu ir faila **binära apstrāde**.

Faila binära apstrāde ir darbības ar failu baitu līmenī, neinteresējoties par datu interpretāciju no datu apstrādes viedokļa programmā.

Tādējādi no faila binārās apstrādes viedokļa, fails ir neko neizsakošu baitu virkne.

Valodā C++ baitu apzīmē datu tips
`char`

bet baitu virkni:
`char*`

kas arī svarīgākie datu tipi failu (un ne tikai failu) binārajā apstrādē.

Bināru failu izmantošanas lielākais mīnuss ir tas, ka šādu failu atverot teksta redaktorā, parasti nekas nav saprotams – faila saturu interpretē ar šo failu saistītā programma.

Tomēr bināru failu izmantošanas priekšrocības ir pietiekoši svarīgas, lai nopietnās programmās datu saglabāšanai bieži izmantotu tieši tos:

- binārā formā glabāti dati aizņem mazāk vietas (sekundārās atmiņas resursu ietaupījums),
- lai rakstītu/lasītu uz/no binära faila, parasti nav nepieciešama nekāda datu pārveidošana (atšķirībā no formatēta izvada/ievada) (procesora resursu ietaupījums),
- apstrādājot failu binārā režīmā, ir pieejamas vairākas efektīvas failu apstrādes metodes (piemēram, meklēšana pēc pozīcijas), kas teksta režīmā nav izmantojamas.

Par bināriem failiem daļēji tika runāts jau nodaļā par teksta failiem – faila objekta funkcijas *get()* un *put()* spēja darboties arī binārajā režīmā. Lai piespiestu šīs funkcijas darboties binārā režīmā, bija nepieciešams noteikt faila atvēršanas papildus režīmu *ios::binary*.

Šajā nodaļā tiks apskatītas failu apstrādes metodes, kas darbojas tikai binārā režīmā, līdz ar to binārā režīma uzstādīšana pie faila atvēršanas nav obligāti nepieciešama.

Darba sākums ar bināru failu.

Faila atvēršana, aizvēršana un visi galvenie principi darbam ar bināru failu ir tādi paši kā darbam ar teksta failu, kas aprakstīts nodaļā par teksta failu apstrādi. Vienīgi binära faila gadījumā parasti daudz precīzāk jāapzinās apstrādājamā faila struktūra, jo datu interpretācija būs jādefinē pašam programmētājam, nevis jāatstāj standarta faila apstrādes metožu ziņā.

Izvade binārā režīmā.

Izvadi binārā režīmā nodrošina faila objekta funkcija *write*.

write

```
ostream &write (const char* buf, int num);
```

Faila objekta funkcija *write()* ieraksta *num* baitus izvades plūsmā (failā), no bufera (atmiņas apgabala), kura sākuma adrese glabājas mainīgajā *buf*.

Ir svarīgi, ka failā ierakstāmie dati tiek padoti kā baitu virkne (*char**) neatkarīgi no tā, kā norādītie dati tiek interpretēti programmā. Šim nolūkam parasti jāveic datu tipa pārveidošana uz *char**.

Ja mums ir programmas mainīgais, kurš reprezentē noteiktu informāciju atmiņā, tad funkcijai *write* padodamā bufera (parametrs nr. 1) sagatavošana parasti notiek šādos 2 soļos:

- adrese paņemšana no mainīgā (ja mainīgais jau nav norāde) (piemēram, pirmkoda piemērs 10.1, rinda 8: &i),
- norādes pārveidošana uz tipu (*char**) (ja tai jau nav tāds tips), izmantojot klasisko tipu pārveidošanas operatoru *((char*)pointer)* vai *(reinterpret_cast<char*>(pointer))* (pirmkoda piemērs 10.1, rindas 8,9).

Otrs funkcijas *write()* arguments ir baitu skaits, cik jāieraksta failā. Šeit ērti izmantot funkciju sizeof(), kam kā argumentu var ņemt gan datu tipu, gan mainīgo, kam vēlas noskaidrot lielumu. Tomēr, lai nodrošinātu informācijas apmaiņu starp dažādām platformām, vai starp programmām, kas kompilētas ar dažādiem kompilatoriem, dažreiz drošāk ir caur parametru #2 padot konstantu vērtību.

Funkcija *write()*, kam caur parametru #2 tiek padots 1, lielā mērā atbilst binārā režīmā izsauktai funkcijai *put()*.

Pirmkoda piemērs 10.1 parāda 2 *int* vērtību ierakstīšanu failā binārā veidā.

Pirmkods 10.1. Bināra faila izvade (*bin1out.cpp*)

```
01 #include <fstream>
02 using namespace std;
03
04 int main ()
05 {
06     fstream fout ("bin_out1.int", ios::out);
07     int i=2006, k=255;
08     fout.write ((char*)&i, sizeof(int));
09     fout.write (reinterpret_cast<char*>(&k), sizeof(int));
10     fout.close ();
11     return 0;
12 }
```

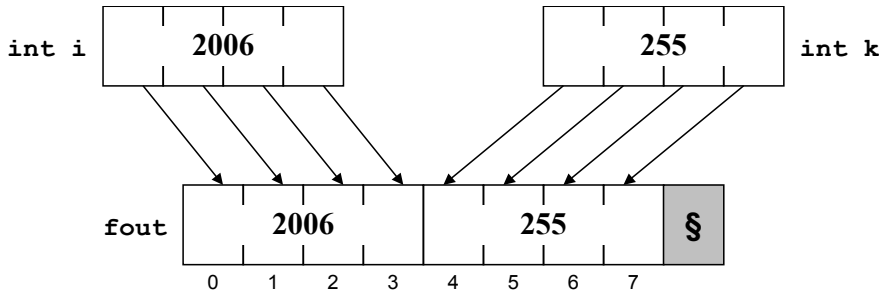
Programmas darbības piemērs, pieņemot, ka *int* aizņem 4 baitus

bin_out1.int (izvade, apskatot ar teksta redaktoru; '¤' ne-teksta simbols)

```
Ö¤¤¤¤¤¤¤
```

bin_out1.int (izvade, pēc būtības)

(sk. failu *fout* attēlā 10.1)



Attēls 10.1. int vērtību izvade binārā failā (atbilst pirmkoda piemēram 10.1)

Nolasīšana binārā režīmā.

Nolasīšanu binārā režīmā nodrošina faila objekta funkcija *read*.

read

```
istream &read (char* buf, int num);
```

Faila objekta funkcija *read()* nolasa *num* baitus no ievades plūsmas (faila) un ieraksta buferī (atmiņas apgabalā), kura sākuma adrese glabājas mainīgajā *buf*.

Abu funkcijai padodamo argumentu noformēšana notiek identiski funkcijai *write()*.

Funkcija *read()*, kam caur parametru #2 tiek padots 1, lielā mērā atbilst binārā režīmā izsauktai funkcijai *get()*.

Pirmkoda piemērs 10.2 demonstrē pirmkoda piemērā 10.1 izveidotā faila nolasīšanu binārā režīmā un tā satura (divu veselu skaitļu) izvadi uz ekrāna.

Kaut arī funkcija *read()* darbojas līdzīgi funkcijai *write()*, tikai pretējā virzienā, tai ir spēkā viena papildus nianse – failā varētu būt atlicis mazāk simbolu nekā tiek padots caur parametru #2. Šajā gadījumā tiek nolasīti tikai tik baiti, cik failā ir, bet nolasīto baitu skaitu var noskaidrot, izmantojot funkciju *gcount()*.

gcount

```
int gcount () const;
```

Faila ievades objekta funkcija *gcount()* atgriež simbolu skaitu, kas nolasīti pēdējā ievades operācijā (sk. pirmkodu 10.3, rindu 16).

Pirmkods 10.2. Bināra faila nolasīšana (*bin2in.cpp*)

```
01 #include <fstream>
02 #include <iostream>
03 using namespace std;
04
05 int main ()
06 {
07     fstream fin ("bin_out1.int", ios::in);
08     int i;
09     fin.read ((char*)&i, sizeof(int));
10     while (fin)
11     {
12         cout << i << endl;
13         fin.read ((char*)&i, sizeof(int));
14     };
```

```
15      fin.close ();
16      return 0;
17  }
```

Programmas darbības piemērs, pieņemot, ka *int* aizņem 4 baitus

bin_out1.int (ievade; tas pats, kas pirmkoda piemēra 10.1 programmas izvade)

(sk. failu *fout* attēlā 10.1)

konsole (izvade)

```
2006
255
```

Programma pirmkoda piemērā 10.3 veic līdzīgu darbu kā programma piemērā 10.2, demonstrējot 2 papildus īpašības (sk. arī attēlu 10.2):

- vienai no faila nolasītajai (arī failā ierakstītajai) datu porcijai nav jāatbilst programmas līmenī interpretējamam datu apgabalam – ierakstīta vai nolasīta tiek "bezpersoniska" baitu virkne,
- funkcija gcount parāda nolasīto baitu skaitu katrā porcijā.

Pirmkods 10.3. Binära faila nolasīšana pa patvaļīgiem blokiem (*bin3in.cpp*)

```
01 #include <fstream>
02 #include <iostream>
03 using namespace std;
04
05 int main ()
06 {
07      fstream fin ("bin_out1.int", ios::in);
08      int arr[2];
09      char *p = (char*)arr;
10      fin.read (p, 3);
11      cout << fin.gcount() << endl;
12      while (fin)
13      {
14          p += 3;
15          fin.read (p, 3);
16          cout << fin.gcount() << endl;
17      };
18      for (int i=0; i<2; i++) cout << arr[i] << endl;
19      fin.close ();
20      return 0;
21 }
```

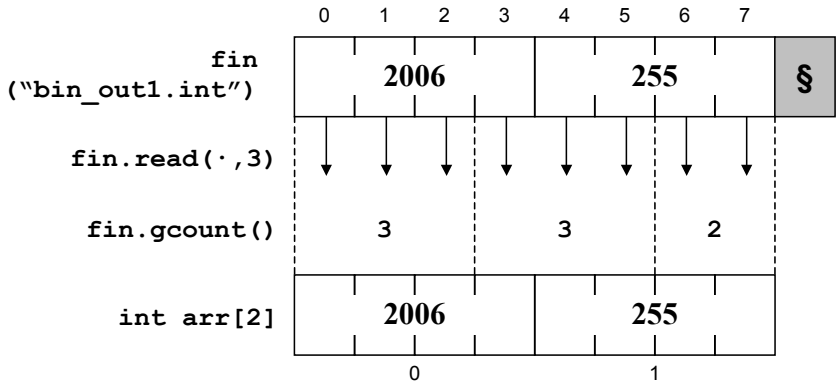
Programmas darbības piemērs, pieņemot, ka *int* aizņem 4 baitus

bin_out1.int (ievade; tas pats, kas pirmkoda piemēra 10.1 programmas izvade)

(sk. failu *fout* attēlā 10.1)

konsole (izvade)

```
3
3
2
2006
255
```



Attēls 10.2. Faila bin_out1.int nolasīšanas shēma patvaļīgos (no nolasāmās informācijas datu tipiem neatkarīgos) blokos pa 3 baitiem pirmkoda piemērā 10.3

Pirmkoda piemērā 10.4 demonstrētā *double* vērtību masīva binära ierakstīšana failā, bet piemērā 10.5 – šī paša ieveidotā faila nolasīšana un satura izdrukāšana uz ekrāna.

Pirmkods 10.4. Binära faila izvade (*bin4out.cpp*)

```
01 #include <fstream>
02 using namespace std;
03 const int arr_size = 5;
04
05 int main ()
06 {
07     fstream fout ("bin_out4.dbl", ios::out);
08     double arr[arr_size] = {3.14, 99.99, 2.7, 0, 1.4};
09     for (int i=0; i<arr_size; i++)
10     {
11         fout.write ((char*)&arr[i], sizeof(double));
12     };
13     fout.close ();
14     return 0;
15 }
```

Programmas darbības piemērs, pieņemot, ka *double* aizņem 8 baitus

bin_out4.dbl (izvade)

{3.14, 99.99, 2.7, 0, 1.4} (katram skaitlim 8 baiti, faila kopējais garums 40 baiti)

Pirmkods 10.5. Binära faila nolasīšana (*bin5in.cpp*)

```
01 #include <fstream>
02 #include <iostream>
03 using namespace std;
04
05 int main ()
06 {
07     fstream fin ("bin_out4.dbl", ios::in);
08     double d;
09     fin.read ((char*)&d, sizeof(double));
10     while (fin)
11     {
```

```
12          cout << d << endl;
13          fin.read ((char*)&d, sizeof(double));
14      };
15      fin.close ();
16      return 0;
17  }
```

Programmas darbības piemērs, pieņemot, ka *double* aizņem 8 baitus

bin_out4.dbl (ievade; tas pats, kas pirmkoda piemēra 10.4 programmas izvade)

{3.14, 99.99, 2.7, 0, 1.4} (katram skaitlim 8 baiti, faila kopējais garums 40 baiti)

konsole (izvade)

```
3.14
99.99
2.7
0
1.4
```

Pirmkoda piemērs 10.6 demonstrē faila apstrādi binārā režīmā – reizē gan lasīšanas, gan rakstīšanas režīmā. Šī programma samaina vietām (ar pirmkoda piemērā 10.4 aprakstīto programmu izveidotā) faila minimālo un maksimālo vērtību un izdrukā uz ekrāna faila saturu pirms un pēc izmaiņas.

Palaižot programmu otro reizes pēc kārtas, fails atgūst sākotnējo saturu (minimālā un maksimālā vērtība tiek samainītas atpakaļ).

Pirmkods 10.6. Minimālā un maksimālā skaitļa samainīšana vietām (*bin6minmax.cpp*)

```
01 #include <fstream>
02 #include <iostream>
03 using namespace std;
04
05 void print (fstream &fin)
06 {
07     double d;
08     fin.clear ();
09     fin.seekg (0);
10     fin.read ((char*)&d, sizeof(double));
11     while (fin)
12     {
13         cout << d << endl;
14         fin.read ((char*)&d, sizeof(double));
15     };
16     cout << endl;
17 };
18
19 int main ()
20 {
21     fstream fin ("bin_out4.dbl", ios::in | ios::out);
22     double d, min, max;
23     int pos=0, minpos=0, maxpos=0;
24     fin.read ((char*)&d, sizeof(double));
25     min = max = d;
26     while (fin)
27     {
```

```
28          if (d < min)
29          {
30              min = d;
31              minpos = pos;
32          }
33          else if (d > max)
34          {
35              max = d;
36              maxpos = pos;
37          };
38          fin.read ((char*)&d, sizeof(double));
39          pos++;
40      };
41      print (fin);
42      if (minpos != maxpos)
43      {
44          fin.clear ();
45          fin.seekg (minpos*sizeof(double));
46          fin.write ((char*)&max, sizeof(double));
47          fin.seekp (maxpos*sizeof(double));
48          fin.write ((char*)&min, sizeof(double));
49      };
50      print (fin);
51      fin.close ();
52      return 0;
53  }
```

Programmas darbības piemērs, pieņemot, ka *double* aizņem 8 baitus

bin_out4.dbl (ievade; tas pats, kas pirmkoda piemēra 10.4 programmas izvade)

{3.14, 99.99, 2.7, 0, 1.4} (katram skaitlim 8 baiti, faila kopējais garums 40 baiti)

bin_out4.dbl (izvade)

{3.14, 0, 2.7, 99.99, 1.4} (skaitļi 0 un 99.99 ir samainīti vietām)

konsole (izvade) (faila saturs pirms un pēc samainīšanas)

```
3.14
99.99
2.7
0
1.4

3.14
0
2.7
99.99
1.4
```

10.2.Datu struktūru glabāšana binārā failā

Sarežģītāku datu struktūru objekti teorētiski arī var tikt ierakstīti/nolasīti no faila, padodot funkcijai *write()*/*read()* visu objektu uzreiz, tomēr vispārīgā gadījumā šādos gadījumos ierakstīšana/nolasīšana notiek pa vienam laukam, jo

- ne visi faila lauki vienmēr satur datus – lauks var būt arī norāde,

- ne vienmēr failā būtu jāieraksta visi dati, kas glabājas objektā,
- dažreiz, pārrakstot datus failā, ir nepieciešama optimizācija no datu glabāšanai nepieciešamās vietas viedokļa.

Pirmkoda piemēros 10.7 un 10.8 parādītās programmas veic tieši šādu objekta informācijas ierakstīšanu/nolasīšanu no faila.

Datu glabāšana notiek 2 failos – pirmajā, izmantojot fiksēta garuma ierakstus, tādējādi, neizmantojot glabāšanai izmantojamās vietas optimizāciju, otrajā, izmantojot mainīga garuma ierakstus un tādējādi samazinot kopējo faila izmēru.

Par **ierakstu** (*record*) jeb **komponenti** sauksim vienu vienotu informācijas kopumu (no programmas viedokļa) failā (šajā gadījumā informāciju par vienu personu).

Pirmajā failā (*bin_out7a.prs*) viena ieraksta struktūra ir šāda:

- *name* – personas vārds – 20B,
- *age* – personas vecums – 4 B.

Otrajā failā (*bin_out7b.prs*) viena ieraksta struktūra ir šāda:

- *name_length* – personas vārda garums – 1B,
- *name* – personas vārds – (name_length)B,
- *age* – personas vecums – 1B.

Glabāšanas vietas ietaupījums otrajā failā ir galvenokārt uz personas vārda glabāšanas rēķina, ieviešot papildus lauku – personas vārda garums – un neglabājot lieku informāciju, bez tam personas vecums tiek glabāts vienā baitā, ar ko pilnīgi pietiek.

Kompaktās glabāšanas mīnuss ir faila tiešās pieejas apstrādes izslēgšana no apstrādes iespējām – nonākt pie ieraksta ar noteiktu indeksu var tikai ar secīgu faila pārstaigāšanu, jo ieraksti ir mainīga garuma.

Pirmkods 10.7. Personas informācijas izvade binārā failā (*bin7persout.cpp*)

```
01 #include <fstream>
02 #include <iostream>
03 using namespace std;
04 const int buffer_size = 20;
05
06 class person
07 {
08     char name[buffer_size];
09     int age;
10 public:
11     person (const char *n, int a)
12     {
13         strcpy (name, n);
14         age = a;
15     };
16     void write1 (ostream &fout)
17     {
18         fout.write (name, buffer_size);
19         fout.write ((char*)&age, sizeof(age));
20     };
21     void write2 (ostream &fout)
22     {
23         int slen = strlen(name);
24         fout.write ((char*)&slen, 1);
```

```
25          fout.write (name, slen);
26          fout.write ((char*)&age, 1);
27      }
28  };
29
30  int main ()
31  {
32      person p ("Liz", 19);
33      person q ("Peter", 20);
34      ofstream fout1 ("bin_out7a.prs");
35      p.write1 (fout1);
36      q.write1 (fout1);
37      fout1.close ();
38      ofstream fout2 ("bin_out7b.prs");
39      p.write2 (fout2);
40      q.write2 (fout2);
41      fout2.close ();
42      return 0;
43  }
```

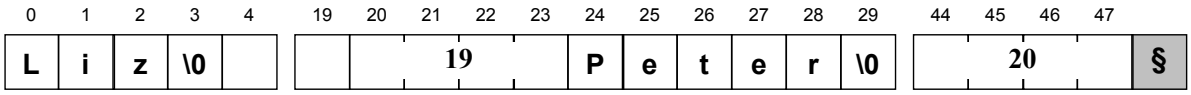
Programmas darbības piemērs, pieņemot, ka *int* aizņem 4 baitus

bin_out7a.prs (izvade; fiksēta garuma ieraksti)

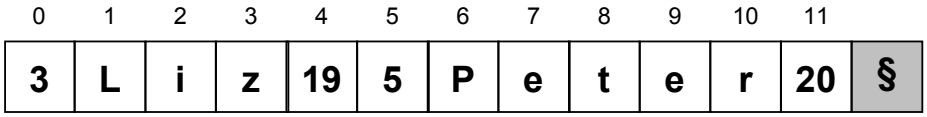
(sk. attēlu 10.3)

bin_out7b.prs (izvade; mainīga garuma ieraksti)

(sk. attēlu 10.4)



Attēls 10.3. Izvades fails bin_out7a.prs, kas glabā informāciju par personām fiksēta garuma ierakstos (atbilstoši pirmkoda piemēram 10.7)



Attēls 10.4. Izvades fails bin_out7b.prs, kas glabā informāciju par personām mainīga garuma ierakstos (atbilstoši pirmkoda piemēram 10.7)

Pirmkods 10.8. Personas informācijas nolasīšana no bināra failā (*bin8persint.cpp*)

```
01 #include <fstream>
02 #include <iostream>
03 using namespace std;
04 const int buffer_size = 20;
05
06 class person
07 {
08     char name[buffer_size];
09     int age;
10 public:
11     bool read1 (istream &fin)
12     {
```

```
13          fin.read (name, buffer_size);
14          fin.read ((char*)&age, sizeof(age));
15          return fin.good ();
16      };
17      bool read2 (istream &fin)
18      {
19          int slen;
20          fin.read ((char*)&slen, 1);
21          fin.read (name, slen);
22          name[slen] = '\0';
23          fin.read ((char*)&age, 1);
24          return fin.good ();
25      };
26      void print ()
27      {
28          cout << name << " " << age << endl;
29      }
30  };
31
32  int main ()
33  {
34      person p;
35      ifstream fin1 ("bin_out7a.prs");
36      while (p.read1 (fin1)) p.print ();
37      fin1.close ();
38      ifstream fin2 ("bin_out7b.prs");
39      while (p.read2 (fin2)) p.print ();
40      fin2.close ();
41      return 0;
42  }
```

Programmas darbības piemērs, pieņemot, ka *int* aizņem 4 baitus

bin_out7a.prs (ievade; fiksēta garuma ieraksti)

(sk. attēlu 10.3)

bin_out7b.prs (ievade; mainīga garuma ieraksti)

(sk. attēlu 10.4)

konsole (izvade)

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

10.3.Tiešās pieejas metodes binära faila apstrādē

Pirmkoda piemērs 10.9 demonstrē tiešās pieejas apstrādes metodes failam *bin_out7a.prs*, kas parādīts attēlā 10.3. Tajā papildus izmantotas vairākas funkcijas: *tellg()*, *seekg()*, *peek()*.

tellg un **tellp**

```
int tellg ();
int tellp ();
```

tellg() un *tellp()* ir funkcijas, kas nosaka faila norādes atrašanās pozīciju (baitu skaitu no faila sākuma) (faila norāde – vieta failā, kurā tiks izpildīta nākamā ievades vai izvades operācija).

tellg() ir pieejama failiem, kas atvērti lasīšanas režīmā, bet *tellp()* – failiem, kas atvērti rakstīšanas režīmā. Ja fails ir atvērts abos režīmos, ir pieejamas abas funkcijas.

seekg un **seekp**

```
istream &seekg (int offset, ios::seek_dir origin = ios::beg);
ostream &seekp (int offset, ios::seek_dir origin = ios::beg);
```

seekg() un *seekp()* ir funkcijas, kas uzstāda faila norādi noteiktā pozīcijā – baitu skaitu *offset* relatīvi pret noteiktu izejas pozīciju *origin*. Ja otrais arguments tiek izlaists, faila norāde tiek uzstādīta relatīvi pret faila sākumu. *seekg()* ir pieejama failiem, kas atvērti lasīšanas režīmā, bet *seekp()* – failiem, kas atvērti rakstīšanas režīmā. Ja fails ir atvērts abos režīmos, ir pieejamas abas funkcijas.

Parametrs *offset* var būt arī negatīvs.

Parametrs origin var saturēt vienu no šādām vērtībām, norādot izejas pozīciju, relatīvi pret kuru tiks veikta nobīde (faila norādes uzstādīšana):

- *ios::beg* – faila sākums,
- *ios::cur* – faila norādes pašreizējā pozīcija,
- *ios::end* – faila beigas (pirms beigu norādes).

peek

```
int peek ();
```

Funkcija *peek()* atgriež kārtējo simbolu failā, bet, atšķirībā no *get()*, nepārbīda faila norādi (“paskatās vienu simbolu uz priekšu”).

peek() ir pieejama failiem, kas atvērti lasīšanas režīmā.

Pirmkods 10.9. Meklēšana un pozīcijas noskaidrošana binārā failā (*bin9seektell.cpp*)

```
01 #include <fstream>
02 #include <iostream>
03 using namespace std;
04 const int buffer_size = 20;
05
06 int main ()
07 {
08     char name[buffer_size], c, d, e, f;
09     ifstream fin ("bin_out7a.prs", ios::binary);
10     fin.seekg (0, ios::end);
11     cout << "Filesize: " << fin.tellg() << endl;
12     fin.seekg (-sizeof(int)-buffer_size, ios::cur);
13     fin.read (name, buffer_size);
14     cout << name << endl;
15     fin.seekg (0, ios::beg);
16     fin.read (&c, 1);
17     e = fin.peek ();
18     fin.get (d);
19     cout << c << d << e;
20     fin.close ();
21     return 0;
22 }
```

Programmas darbības piemērs, pieņemot, ka *int* aizņem 4 baitus

bin_out7a.prs (ievade; fiksēta garuma ieraksti)

(sk. attēlu 10.3)

konsole (izvade)

```
Filesize: 48
Peter
Lii
```

Komentāri pie pirmkoda piemēra 10.9 (sk. arī attēlu 10.3).

- Rinda 10. Faila norāde tiek nostādīta uz faila beigām (pozīcija 0 relatīvi pret faila beigām).
- Rinda 11. Tiek izdrukāta pašreizējā faila pozīcija – 48.
- Rinda 12. Faila norāde tiek pārbīdīta 24 pozīcijas atpakaļ (24 = *int* lielums + 20). Tagad faila norāde atrodas pozīcijā 24.
- Rinda 13. Tiek nolasīta baitu virkne garumā 20 no pozīcijas 24. Buferis *name* satur vērtību "Peter", faila norāde atrodas pozīcijā 44.
- Rinda 15. Faila norāde tiek nolikta uz faila sākumu – pozīcijā 0.
- Rinda 16. Mainīgajā c tiek nolasīta vērtība 'L'. Faila norāde pāriet uz pozīciju 1.
- Rinda 17. Mainīgajā e tiek nolasīta vērtība 'i'. Faila norāde paliek pozīcijā 1.
- Rinda 18. Mainīgajā e tiek nolasīta vērtība 'i'. Faila norāde pāriet uz pozīciju 2.