7. Teksta failu apstrāde

Nodaļas saturs:

7.1. Fails

Kaut arī no vispārīga algoritmu konstrukcijas viedokļa failiem nav specifiskas lomas, tomēr sakarā ar pietiekoši lielo praktisko nozīmi, failu apstrāde ir stingri standartizēta praktiski visās programmēšanas valodās.

Fails (*file*) ir datu kopums, kas izvietots sekundārajā atmiņā un kas operētājsistēmas līmenī tiek identificēts ar noteiktu faila vārdu.

Tāpat kā operatīvajai atmiņai, arī failam **mazākā adresējamā vienība ir 1 baits**, tādējādi no datu apstrādes viedokļa var uzskatīt, ka fails ir baitu virkne.

Kaut arī darbs ar failiem valodā C++ ir standartizēts, tomēr ir atsevišķas nianses, kas nosaka atšķirības failu apstrādē dažādās operētājsistēmās, jo no programmas viedokļa fails ir operētājsistēmas pakalpojums.

Svarīgs jēdziens darbā ar failu ir **faila beigu pazīme** (*EOF*, *end of file*), kam ir līdzīga nozīme kā simbolu virknes beigu simbolam. Daudzos gadījumos (sevišķi valodā C++) arī faila beigu pazīmi var uztvert kā "faila beigu simbolu" (un tā ir vieglāk uztvert programmas būtību), tomēr vispārīgā gadījumā faila beigu pazīme ir cita abstrakcijas līmeņa jēdziens nekā faila dati.

Failu C++ programmā identificē **faila objekts**, kuru reprezentē **faila mainīgais** (piemēram, *fout* pirmkoda piemērā 7.1).

Galvenās darbības ar failu ir

- lasīšana (*reading*),
- rakstīšana (*writing*),
- pārbaude uz faila beigu (*EOF*) iestāšanos (lasot failu).

Lasīšana no faila var notikt ne tālāk, ka līdz faila beigām, taču rakstīšana, ja tā notiek faila galā, automātiska pārbīda faila beigas uz priekšu.

Gan lasīšana, gan rakstīšana failā standarta variantā notiek secīgi (pēc kārtas) (parasti, sākot ar faila sākumu), tomēr ir pieejamas arī tiešās pieejas metodes.

Valodā C++ failu apstrādei jāiekļauj standartbibliotēka <*fstream*> (pirmkods 7.1, rinda 1), un var lietot 3 tipu failu objektus: *ifstream* (tikai lasīšanai), *ofstream* (tikai rakstīšanai), *fstream* (gan lasīšanai, gan rakstīšanai; pirmkods 7.1, rinda 7).

Ņemot vērā to, ka fails ir operētājsistēmas pakalpojums, turklāt failu sistēmas līmenī tas tiek identificēts ar faila vārdu, turpretī programmā – ar faila mainīgo, faila apstrādē ir nepieciešamas 2 šādas papildus darbības:

- **Faila atvēršana** (*opening*) pirms darba sākšanas (fiziskā faila piesaiste faila mainīgajam un noteikta darba režīma uzstādīšana).
- **Faila aizvēršana** (*closing*), darbu beidzot (pirmkods 7.1, rinda 10).

Faila atvēršana un aizvēršana ir saistīta ar noteiktu, pietiekoši apjomīgu darbību veikšanu operētājsistēmas līmenī, jo **fails ir resurss, par kuru atbild operētājsistēma** (piemēram, lai divi lietotāji reizē nevarētu rakstīt vienā failā).

Faila atvēršanu var veikt 2 veidos:

- izmantojot (faila objekta) funkciju *open* ar parametriem (faila vārds un režīms),
- norādot (tādus pašus) atvēršanas parametrus jau faila mainīgā deklarēšanas laikā.

Tādējādi rindu 7 un 8 (pirmkods 7.1):
```
fstream fout;
fout.open ("txt_out1.txt", ios::out);
```
vietā var rakstīt:
```
fstream fout ("txt_out1.txt", ios::out);
```

Faila atvēršanas un aizvēršanas funkciju apraksts.

open
```
void open (const char* filename, int mode);
```

Faila objekta funkcija *open()* atver failu *filename* režīmā *mode* (pirmkods 7.1, rinda 8). Režīmu nosaka viena vai vairākas vērtības (atdalītas ar '|'). Režīma parametru var izlaist – tādā gadījumā faila atvēršana notiek noklusētajā režīmā atkarībā no faila objekta tipa (tabula 7.2).

Piemēri funkcijas *open* izmantošanai:
```
file.open ("test.txt", ios::in | ios::out);
file.open ("test.txt", ios::out | ios::app);
file.open ("test.txt", ios::in);
file.open ("test.txt", ios::in | ios::out | ios::binary);
```

Uzstādot faila režīmu, ir obligāta vismaz viena no vērtībām *ios::in* vai *ios::out*, pārējās vērtības domātas precizēšanai.

Tabula 7.1. Svarīgākās faila atvēršanas režīma vērtības

`ios::in`	Nosaka lasīšanas režīmu. Ja fails ar doto nosaukumu neeksistē, tad faila atvēršana beidzas ar neveiksmi.
`ios::out`	Nosaka rakstīšanas režīmu. Ja fails eksistē, tad izdzēš visu tā saturu. Ja fails neeksistē, tad izveido jaunu failu. Faila saturs netiek izdzēsts, ja failam papildus noteikts arī lasīšanas režīms. Faila atvēršana beidzas ar neveiksmi, ja papildus noteikts lasīšanas režīms un fails ar doto nosaukumu neeksistē.
`ios::binary`	Nosaka bināro režīmu lasīšanā vai rakstīšanā (sīkāk aprakstīts zemāk).
`ios::app`	(Tikai kopā ar *ios::out*) Nosaka, ka rakstīšanas režīmā atvērta faila saturs netiek izdzēsts (kā tas notiek noklusētajā variantā), bet gan rakstīšana turpinās, sākot ar faila beigām.

Tabula 7.2. Noklusētais atvēršanas režīms dažādu tipu failu objektiem

`fstream`	`ios::in | ios::out`
`ifstream`	`ios::in`
`ofstream`	`ios::out`

close

```
void close ();
```

Faila objekta funkcija *close()* aizver failu (pirmkods 7.1, rinda 10).

7.2. Teksta fails un formatēta izvade

Viens no vienkāršākajiem failu veidiem ir teksta fails.

Teksta fails (*text file*) ir fails, kas sastāv no simboliem (burtiem, cipariem, pieturzīmēm utt.).

Praktiski teksta fails ir atpazīstams pēc tā, ka tajā atrodošā informācija cilvēkam ir uztverama, apskatot to ar vienkāršu teksta redaktoru (piemēram, *notepad* vai *vi*).

No programmēšanas viedokļa vairāk runā nevis par teksta failu, bet gan par faila apstrādi teksta režīmā.

Faila apstrāde teksta režīmā parasti notiek **pa simbolam** (kas parasti gandrīz atbilst jēdzienam – pa baitam) vai **pa rindiņai**.

Teksta failu var apstrādāt (lasīt, rakstīt) formatēti (pārveidojot binārus ("neteksta") datus par teksta un otrādi) vai neformatēti (lasot vai rakstot simbolus tiešā veidā).

No apstrādes viedokļa teksta failu var uztvert no 3 aspektiem:

- klaviatūras (ievade) vai displeja (izvade) analogs sekundārajā atmiņā,
- simbolu (baitu) virkne,
- teksta rindiņu virkne.

Formatēta izvade failā (pirmkoda piemērs 7.1) notiek, izmantojot izvades operatoru << un identiskus formatēšanas mehānismus (funkcijas un manipulatorus), kādi ir pieejami izdrukāšanai uz displeja (pirmkods 7.1, rinda 9) un kas jau ir aprakstīti nodaļā "C++ pamati" (manipulatori ir pieejami bibliotēkā <*iomanip*>, bet <*iostream*> vietā stājas bibliotēka <*fstream*>)

Pirmkods 7.1. Formatēta faila izvade (*txt1fmtout.cpp*)

```
01 #include <fstream>
02 #include <iomanip>
03 using namespace std;
04
05 int main ()
06 {
07     fstream fout;
08     fout.open ("txt_out1.txt", ios::out);
09     fout << setw(5) << 1 << setw(5) << 999 << endl;
10     fout.close ();
11     return 0;
12 }
```

Programmas darbības piemērs (tagad un turpmāk lietoti šādi neredzamo simbolu apzīmējumi: '·' tukšums, '¶' jaunas rindiņas simbols, '§' faila beigas

txt_out1.txt (izvade)

```
····1··999¶
§
```

7.3. Formatēta ievade no teksta faila

7.3.1. Ievades no faila standarta shēma

Formatēta ievade no teksta faila (pirmkoda piemērs 7.2) notiek pēc līdzīga principa, kādi ir pieejami ievadei no klaviatūras un kas jau ir aprakstīti nodaļā "C++ pamati". Formatēta ievade no faila notiek, izmantojot ievades operatoru >>.

Ievadei no faila, salīdzinājumā ar ievadi no klaviatūras, ir šādas atšķirības:

- jaunas rindiņas simbolam (atbilst ENTER) nav specifiskas nozīmes,
- pēc katras ievades darbības tiek pārbaudīts, vai nav sasniegtas faila beigas.

Programma, kas parādīta pirmkoda piemērā lasa veselus skaitļus no teksta faila un izdrukā tos uz ekrāna, kamēr nav sasniegtas faila beigas.

Faila beigu pazīmi norāda funkcija *eof*:

eof

```
bool eof () const;
```

Faila objekta funkcija *eof()* atgriež *true*, ja tiek konstatētas faila beigas, citādi atgriež *false*. (*const* norāda, ka par funkciju *eof* tiek garantēts, ka tā neizmaina faila objektu; sīkāk par to aprakstīts nodaļā par objektorientēto programmēšanu).

Pirmkods 7.2. Formatēta faila ievades standarta shēma (*txt2fmtin.cpp*)

```
01 #include <fstream>
02 #include <iostream>
03 using namespace std;
04
05 int main ()
06 {
07     fstream fin;
08     int i;
09     fin.open ("txt_in2.txt", ios::in);
10     fin >> i;
11     while (!fin.eof())
12     {
13         cout << i << endl;
14         fin >> i;
15     };
16     fin.close ();
17     return 0;
18 }
```

Programmas darbības piemērs ('·' tukšums, '¶' jaunas rindiņas simbols, '§' faila beigas)

txt_in2.txt (ievade)

```
31·12·2006¶
§
```

konsole (izvade)

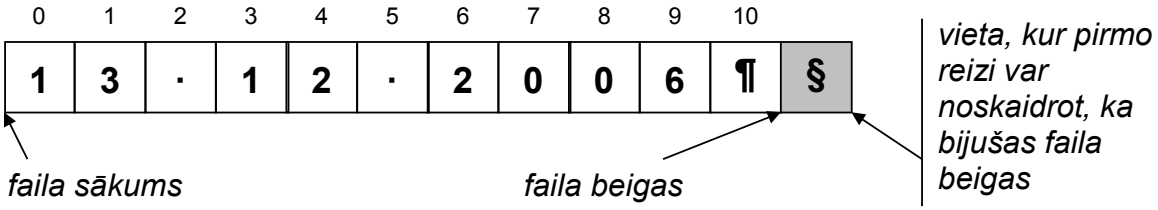
```
31
12
2006
```

Vēl viena tipiska vienošanās par teksta failiem ir tāda, ka parasti tie beidzas ar jaunas rindiņas simbolu (*newline*). Ja tas tā tiek pieņemts, tad vispārīgā gadījumā šādam failam apstrāde ir izveidojama vieglāk. Arī, piemēram, valodas C++ standarts pieprasa, ka C++ programmas faili (un tie ir teksta faili) beigtos ar jaunas rindiņas simbolu. Ja pirmkoda piemērā 7.2 ievades fails *txt_in2.txt* nebeigtos ar jaunas rindiņas simbolu, bet gan beigtos jau ar '6', tad programma uz ekrāna izdrukātu tikai *31* un *12*, jo jau uzreiz pēc *2006* nolasīšanas tiktu nofiksētas faila beigas, un nākošā iterācija, kurā vajadzētu izdrukāt šo skaitli, vairs nenotiktu (tai pat laikā programmas beigās mainīgais *i* saturētu vērtību *2006*).

No pirmkoda piemērā 7.2 parādītās programmas izriet standarta shēma, kādā notiek secīga faila nolasīšana C++ (attēls 7.1). Šī shēma raksturojas ar to, ka vienas cikla iterācijas laikā tiek apstrādāta viena vērtība, bet nolasīta jau nākamā. Tas saistīts ar C++ specifiku – C++ tiek uzstādīta faila beigu pazīme nevis tad, kad ir loģiskās faila beigas, bet gan tikai pēc tam, kad ir **mēģināts nolasīt aiz faila beigām** (attēls 7.2). Ar to C++ atšķiras no dažām citām programmēšanas valodām (piemēram, PASCAL, kur vienā iterācijā var nolasīt un apstrādāt to pašu vērtību).

```
nolasa pirmo vērtību no faila
WHILE (nav faila beigas)
    apstrādā nolasīto vērtību
    nolasa nākošo vērtību no faila
END WHILE
```

Attēls 7.1. Faila secīgas nolasīšanas standarta shēma C++



Attēls 7.2. Fails un faila beigas C++ (atbilst failam txt_in2.txt pirmkoda piemērā 7.2)

7.3.2. Faila objekta bloķēšana pēc kļūdas

Faila apstrādes laikā var iestāties kļūdas situācija (piemēram, ievaddatu neatbilstība mainīgā tipam). Tādā gadījumā **faila objekts tiek bloķēts**, un visas turpmākās darbības ar šo failu tiek ignorētas.

Vispārīgā gadījumā var būt divi galvenie iemesli faila objekta bloķēšanai:

- faila beigas,
- ievades vai izvades kļūda.

Programmai ir jābūt gatavai, ka faila objekts varētu tikt bloķēts. Ideālā gadījumā pēc katras faila operācijas būtu jāpārbauda, vai nav iestājusies kļūda vai faila beigas.

Faila beigas var noskaidrot ar funkciju *eof()*, kas aprakstīta, iepriekš, bet ievades vai izvades kļūdu var noskaidrot ar kādu no funkcijām *fail()* vai *bad()*.

fail

```
bool fail () const;
```

Faila objekta funkcija *fail()* atgriež *true*, ja tiek konstatēta ievades vai izvades kļūda, citādi atgriež *false*.

bad

```
bool bad () const;
```

Faila objekta funkcija *bad()* atgriež *true*, ja tiek konstatēta fatāla ievades vai izvades kļūda, citādi atgriež *false*.

Funkciju *eof()*, *fail()*, *bad()* vietā bieži lieto (pēc nozīmes pretēju) funkciju *good()*, kas norāda, ka nav ne faila beigas, ne arī ir iestājusies kļūda (pirmkods 7.4, rinda 11).

good

```
bool good () const;
```

Faila objekta funkcija *good()* atgriež *true*, ja nav konstatēta ievades/izvades kļūda, kā arī (ievades gadījumā) nav konstatētas faila beigas.

Faila objekta bloķēšanu ilustrē pirmkoda piemērs 7.3 (atšķiras no 7.2 tikai ar rindu 9 – paņem citu failu, kurā ir kļūda datos). Šī programma nekorekti apstrādā ievades kļūdu, tāpēc ieciklojas: kļūda notiek rindiņā 13, mēģinot nolasīt skaitli, bet failā sastopot burtu ‘a’, (attēls 7.3), faila objekts tiek bloķēts, bet cikls nebeidzas, jo cikla beigu nosacījums ir faila beigu iestāšanās (nevis kļūda).

Lai vienkāršākajā gadījumā risinātu šo problēmu, būtu nepieciešams cikla nosacījumu nomainīt uz vispārīgāku (pirmkods 7.4 – atšķiras no pirmkoda 7.3 tikai ar rindu 11).

Pirmkods 7.3. Formatēta faila ievades mēģinājums ar kļūdu ievaddatos (*txt3fmtin.cpp*)

```
01 #include <fstream>
02 #include <iostream>
03 using namespace std;
04
05 int main ()
06 {
07     fstream fin;
08     int i;
09     fin.open ("txt_in3.txt", ios::in);
10     fin >> i;
11     while (!fin.eof())
12     {
13         cout << i << endl;
14         fin >> i;
15     };
16     fin.close ();
17     return 0;
18 }
```

Programmas darbības piemērs (‘·’ tukšums, ‘¶’ jaunas rindiņas simbols, ‘§’ faila beigas)

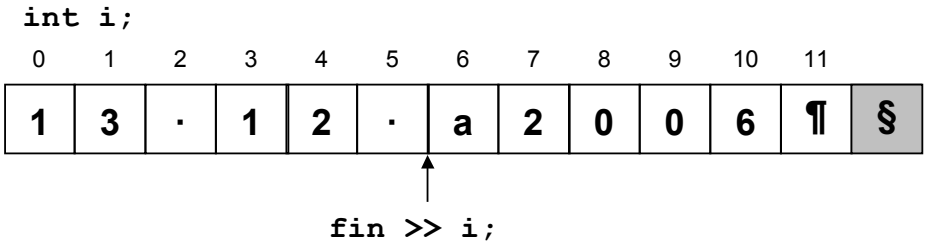
txt_in3.txt (ievade)

```
31·12·a2006¶
§
```

konsole (izvade)

```
31
12
12
12
```

```
12
12
(utt., jo programma ieciklojas)
```



Attēls 7.3. Faila objekta bloķēšana ievaddatu neatbilstības dēļ

Pirmkoda piemērā 7.4 parādītā programma ir programmas 7.3 uzlabojums (rinda 11), kas nodrošina faila nolasīšanas apstāšanos ne tikai, sastopot faila beigas, bet arī neveiksmīga datu nolasīšanas mēģinājuma gadījumā (arī atbilst attēlam 7.3).

Pirmkods 7.4. Formatēta faila ievades pārtraukums pēc kļūdas ievaddatos (*txt4fmtin.cpp*)

```
01 #include <fstream>
02 #include <iostream>
03 using namespace std;
04
05 int main ()
06 {
07     fstream fin;
08     int i;
09     fin.open ("txt_in3.txt", ios::in);
10     fin >> i;
11     while (fin.good())
12     {
13         cout << i << endl;
14         fin >> i;
15     };
16     fin.close ();
17     return 0;
18 }
```

Programmas darbības piemērs (‘·’ tukšums, ‘¶’ jaunas rindiņas simbols, ‘§’ faila beigas)

txt_in3.txt (ievade)

```
31·12·a2006¶
§
```

konsole (izvade)

```
31
12
```

7.4. Faila objekta atbloķēšana pēc kļūdas

Lai turpinātu darbu ar (kļūdas vai faila beigu iestāšanās dēļ) bloķētu faila objektu, jāveic šādas darbības:

- objekta atbloķēšana ar *clear()*,
- pasākumi kļūdas situācijas novēršanai (piemēram, cita faila atvēršana, ja kļūda bijusi pie atvēršanas vai noteikta datu daudzuma izlaišana, ja kļūda bijusi ievaddatos).

Pirmkoda piemērā 7.5 to demonstrē rindas 17 un 18. Rindas 15-20 kopumā veic ievaddatu ignorēšanu pa vienam baitam (bet ne tālāk kā līdz faila beigām) līdz tiek veiksmīgi nolasīts skaitlis.

clear

```
void clear ();
```

Faila objekta funkcija *clear()* atbloķē faila objektu, uzstādot kļūdas un faila beigu bitus uz 0, bet *good* bitu uz 1.

ignore

```
istream &ignore (int count=1, char delim=EOF);
```

Faila objekta funkcija *ignore()* izlaiž noteiktu skaitu (*count*) simbolu ievades plūsmā, bet ne tālāk kā līdz pirmajam sastaptajam simbolam *delim*.

Pirmkods 7.5. Faila objekta atbloķēšana pēc kļūdas ievaddatos (*txt5fmtin.cpp*)

```
01 #include <fstream>
02 #include <iostream>
03 using namespace std;
04
05 int main ()
06 {
07     fstream fin;
08     int i;
09     fin.open ("txt_in5.txt", ios::in);
10     fin >> i;
11     while (fin)
12     {
13         cout << i << endl;
14         fin >> i;
15         while (!fin.good() && !fin.eof())
16         {
17             fin.clear ();
18             fin.ignore (1, '\n');
19             fin >> i;
20         }
21     };
22     fin.close ();
23     return 0;
24 }
```

Programmas darbības piemērs (‘·’ tukšums, ‘¶’ jaunas rindiņas simbols, ‘§’ faila beigas)

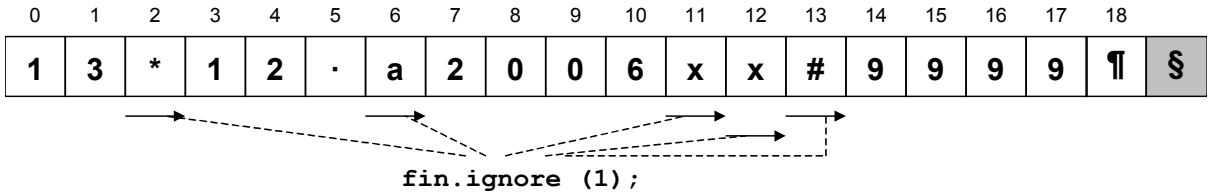
txt_in5.txt (ievade)

```
31*12·a2006xx#9999¶
§
```

konsole (izvade)

```
31
12
2006
9999
```

Pirmkoda piemēra 7.5 rinda 11 parāda, ka nosacījuma *fin.good()* vietā var lietot vienkārši pašu faila objektu *fin*.



Attēls 7.4. Funkcijas ignore darbības demonstrācija atbilstoši pirmkoda piemēram 7.5

7.5. Teksta failu apstrāde pa vienam simbolam

Formatēšana nozīmē datu pārveidošanu pēc nolasīšanas vai pirms izdrukas (failā). Vienkāršākā teksta faila apstrāde bez formatēšanas ir **pa vienam simbolam**. Arī nolasīšana pa vienam simbolam notiek pēc faila nolasīšanas standarta shēmas, kas redzama attēlā 7.1.

Teksta faila apstrāde pa vienam simbolam notiek, izmantojot funkcijas *get()* un *put()*, kas attiecīgi nolasa vai ieraksta vienu simbolu failā (abu funkciju aprakstu sk. zemāk).

Pirmkoda piemērā 7.6 parādītā programma (pa vienam simbolam) izdrukā teksta faila saturu uz ekrāna un beigās arī faila garumu simbolos. Attēlā 7.5 redzams faila saturs pa simbolam (kopā 19 simboli).

Pirmkods 7.6. Teksta faila nolasīšana pa vienam simbolam (*txt6get.cpp*)

```
01 #include <fstream>
02 #include <iostream>
03 using namespace std;
04
05 int main ()
06 {
07     fstream fin;
08     char c;
09     int filesize = 0;
10     fin.open ("txt_in6.txt", ios::in);
11     fin.get (c);
12     while (fin)
13     {
14         filesize++;
15         cout << c;
16         fin.get (c);
17     };
18     fin.close ();
19     cout << filesize << endl;
20     return 0;
21 }
```

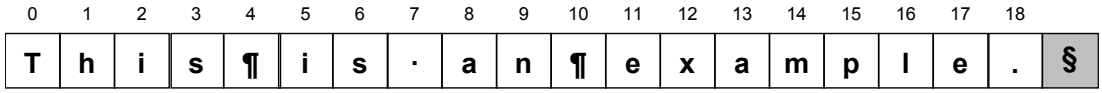
Programmas darbības piemērs (‘·’ tukšums, ‘¶’ jaunas rindiņas simbols, ‘§’ faila beigas)

txt_in6.txt (ievade)

```
This¶
is·an¶
example.§
```

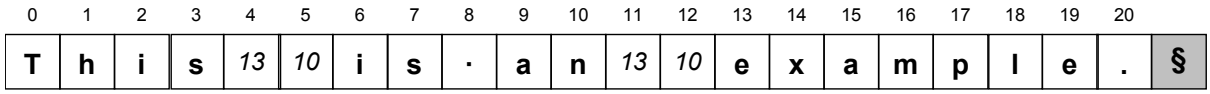
konsole (izvade)

```
This
is an
example.19
```



Attēls 7.5. Ievades fails txt_in6.txt, nolasot teksta režīmā (atbilstoši pirmkoda piemēram 7.6)

Tomēr arī teksta failu var nolasīt gan teksta, gan binārā režīmā. Labākais piemērs tam ir fakts, ka *Windows* sistēmā jaunas rindiņas simbolu ¶ kodē nevis ar vienu baitu (simbola kods 13 vai 10), ka tas ir *Unix/Linux* un *Macintosh* sistēmās, bet gan ar divu baitu secību: <13,10>. Tādējādi *Windows* sistēmā kodēts teksta fails ir garāks uz jaunas rindiņas simbolu rēķina (sk. attēlu 7.6, kā arī pārliecinieties, ka faila *txt_in6.txt* garums *Windows* kodējumā ir 21B).



Attēls 7.6. Ievades fails txt_in6.txt, nolasot binārā režīmā (Windows sistēmā kodētam failam, atbilstoši pirmkoda piemēram 7.6)

Valoda C++ dod iespēju gan ņemt vērā dažādu platformu kodēšanas īpašības, gan tās ignorēt. To nodrošina attiecīgi ar bināra vai teksta režīma failu apstrādi, režīma papildus vērtību norādot pie atvēršanas. Ja ir uzrādīta bināra režīma vērtība *ios::binary*, tad apstrādes režīms ir binārs, ja nav uzrādīts – tad teksta (speciālas teksta režīma vērtības nav).

Programma 7.6 lieto teksta režīmu faila nolasīšanā (jo rindiņā 10 nav uzrādīts *ios::binary* atvēršanas režīms), tādējādi fails tiek interpretēts atbilstoši attēlam 7.5 un faila izmērs tiek noteikts – 19 simboli.

Tomēr, ja piemēra 7.6 rindiņu 10 nomainītu uz

```
fin.open ("txt_in6.txt", ios::in | ios::binary);
```

– tad fails *txt_in6.txt* (ja tas kodēts *Windows* sistēmā) tiktu interpretēts atbilstoši attēlam 7.6 un faila izmērs tiktu noteikts – 21 baits un uz ekrāna būtu šāda izdruka (salīdzināt ar pirmkoda piemēra 7.6 izdruku uz ekrāna):

konsole (izvade)

```
This
is an
example.21
```

Teksta režīma priekšrocība ir tāda, ka C++ pats uzņemas interpretēt teksta failu atkarībā no dotās platformas kodējuma, tādējādi viens un tas pats C++ programmas kods ir derīgs dažādām platformām. Binārs režīms ļauj paskatīties uz failu precīzi pa baitam – kāds tas precīzi ir, neveicot nekādu papildus apstrādi (interpretēšanu).

Līdzīgā veidā teksta un binārais režīms sastopams failu pārsūtīšanā, izmantojot *FTP* protokolu.

Binārais un teksta režīms attiecas arī uz izvadi failā (faila objekts *fout* piemērā 7.7). Viena simbola nolasīšanai vai ierakstīšanai izmantotās funkcijas *get()* un *put()* ir izmantojamas vai nu teksta vai binārā režīmā, kas ir atkarīgs no uzrādītā režīma pie faila atvēršanas, tomēr faila

objektam ir pieejamas arī tādas funkcijas, kuras spēj strādāt tikai binārā režīmā, un faila atvēršanas režīma uzstādīšana (teksta vai bināra) to darbību no šāda viedokļa neietekmē.

get

```
istream &get (char &c);
```

Faila objekta funkcija *get()* ar vienu parametru nolasa vienu simbolu no ievades (faila) plūsmas un ievieto mainīgajā *c*. Ja fails atvērts binārā režīmā, tad tiek nolasīts nevis viens simbols, bet viens baits.

```
int get ();
```

Faila objekta funkcija *get()* bez parametriem darbojas tāpat, bet nolasīto simbolu atgriež kā atgriežamo vērtību.

put

```
ostream &put (char c);
```

Faila objekta funkcija *put()* ieraksta padoto simbolu *c* izvades (faila) plūsmā. Ja fails atvērts binārā režīmā, tad mainīgajā *c* saglabātais baits tiek precīzi pārrakstīts failā.

Pirmkoda piemērā 7.7 parādītā programma pārraksta vienu teksta failu otrā pa vienam simbolam (atšķirība no piemēra 7.6 ir tikai tā, ka ekrāna vietā ir otrs fails).

Pirmkods 7.7. Teksta faila pārrakstīšana citā failā pa vienam simbolam (*txt7getput.cpp*)

```
01 #include <fstream>
02 #include <iostream>
03 using namespace std;
04
05 int main ()
06 {
07     char c;
08     fstream fin ("txt_in6.txt", ios::in);
09     fstream fout ("txt_out7.txt", ios::out);
10     fin.get(c);
11     while (fin)
12     {
13         fout.put(c);
14         fin.get(c);
15     };
16     fin.close ();
17     fout.close ();
18     return 0;
19 }
```

Programmas darbības piemērs (‘·’ tukšums, ‘¶’ jaunas rindiņas simbols, ‘§’ faila beigas)

txt_in6.txt (ievade)

```
This¶
is·an¶
example.§
```

txt_out7.txt (izvade)

```
This¶
is·an¶
example.§
```

konsole (izvade)

(nekas netiek izvadīts)

7.6. Teksta failu nolasīšana pa rindai

Bieži lietota teksta failu nolasīšana ir **pa rindai**. Dalīšana rindās ir teksta strukturēšanas veids, kas atbilst tam, kā to dara cilvēki, pierakstot dabisko valodu tekstus. Tas varētu būt izskaidrojums šāda veida teksta failu apstrādes biežajam lietojumam. Nolasīšana pa rindai notiek pēc faila nolasīšanas standarta shēmas, kas redzama attēlā 7.1.

Faila nolasīšana pa vienai rindai notiek, izmantojot funkciju *getline()*, kam ir 2 modifikācijas – zema un augsta līmeņa simbolu virknēm.

getline (*ar zema līmeņa simbola virkni buferim*)
```
istream &getline (char *buf, int num, char delim='\n');
```

Faila objekta funkcija *getline()* lasa simbolus masīvā *buf*, kamēr nav nolasīti *num-1* simboli (viena vieta automātiski tiek rezervēta simbolu virknes beigu simbolam) vai kamēr netiek sastapts atdalītājsimbols (pēc noklusēšanas – jaunas rindiņas simbols) vai faila beigas. Nolasot *num-1* simbolus pirms sastapts atdalītājsimbols, iestājas ievades kļūda, un faila objekts tiek nobloķēts. Nolasītais atdalītājsimbols netiek ievietots masīvā. Nolasītās virknes galā automātiski tiek ievietots simbolu virknes beigu simbols.

Funkcijai *getline()* ir vēl arī citas modifikācijas, bez tam *getline()*, tāpat kā *get()* un *put()*, ir izmantojama arī standarta ievadei (*cin*) no klaviatūras.

Pirmkoda piemērā 7.8 parādītā programma nolasa teksta failu pa vienai rindai un pārraksta otrā failā. Ievērojiet, ka otrā failā beigās tiek pielikts papildus jaunas rindiņas simbols. Šajā piemērā visas ievades faila rindiņas bija pietiekoši īsas, lai ietilptu masīvā *s*.

Pirmkods 7.8. Teksta faila pārrakstīšana citā failā pa rindai (*txt8getline.cpp*)

```
01 #include <fstream>
02 using namespace std;
03 const unsigned size = 20;
04
05 int main ()
06 {
07     char s[size];
08     fstream fin ("txt_in6.txt", ios::in);
09     fstream fout ("txt_out8.txt", ios::out);
10     fin.getline (s, size, '\n');
11     while (fin)
12     {
13         fout << s << endl;
14         fin.getline (s, size, '\n');
15     };
16     fin.close ();
17     fout.close ();
18     return 0;
19 }
```

Programmas darbības piemērs (‘·’ tukšums, ‘¶’ jaunas rindiņas simbols, ‘§’ faila beigas)

txt_in6.txt (ievade)

```
This¶
is·an¶
example.§
```

txt_out8.txt (izvade)

```
This¶
is·an¶
example.¶
§
```

konsole (izvade)

(nekas netiek izvadīts)

Pirmkoda piemērs 7.9 demonstrē funkcijas *getline()* darbību gadījumā, kad dažu rindiņu garums failā pārsniedz masīva garumu, kurā tās tiek ielasītas.

Pirmkoda piemērā 7.9, salīdzinot ar 7.8, ir šādas izmaiņas:

- Masīva izmērs (rinda 3) ir mazāks nekā dažu rindiņu izmērs, tādēļ nepieciešama faila objekta atbloķēšana, ja lasīšana beigusies vietas trūkuma dēļ masīvā (rinda 14).
- Ievades failā *txt_in9.txt*, salīdzinot ar *txt_in6.txt*, ir papildus jaunas rindiņas simbols beigās.

Pirmkods 7.9. Teksta faila pārrakstīšana citā failā pa rindai, izmantojot ievades masīvu ar ierobežotu izmēru (*txt9getline.cpp*)

```
01 #include <fstream>
02 using namespace std;
03 const unsigned size = 5;
04
05 int main ()
06 {
07     char s[size];
08     fstream fin ("txt_in9.txt", ios::in);
09     fstream fout ("txt_out9.txt", ios::out);
10     fin.getline (s, size, '\n');
11     while (!fin.eof())
12     {
13         fout << s << endl;
14         if (!fin.good()) fin.clear ();
15         fin.getline (s, size, '\n');
16     };
17     fin.close ();
18     fout.close ();
19     return 0;
20 }
```

Programmas darbības piemērs (‘·’ tukšums, ‘¶’ jaunas rindiņas simbols, ‘§’ faila beigas)

txt_in9.txt (ievade)

```
This¶
is·an¶
example.¶
§
```

txt_out9.txt (izvade)

```
This¶
is·a¶
n¶
exam¶
ple.¶
```

```
§
```

konsole (izvade)

(nekas netiek izvadīts)

Pirmkoda piemēri 7.8 un 7.9 parāda, ka funkcija *getline()* ir veiksmīgi izmantojama gadījumos, kad rindiņu izmērs nepārsniedz noteiktu fiksētu (masīva jeb bufera) garumu. Lasīšanas pārtraukšanu vietas trūkuma dēļ buferī var uzskatīt par speciālu vai pat ārkārtas gadījumu, un tās rezultātā izceļas ievades kļūda, bloķējot faila objektu.

Lietojot *getline()* augsta līmeņa simbolu virknēm (*string*), vairs nav jāuztraucas par iespējamo vietas trūkumu, nolasot teksta faila daļu (pirmkods 7.10).

getline (*ar augsta līmeņa simbola virkni buferim*)

```
istream &getline (istream &file, string &buf, char delim='\n');
```

Funkcija *getline()* lasa simbolus augsta līmeņa simbolu virknē *buf*, kamēr netiek sastapts atdalītājsimbols (pēc noklusēšanas – jaunas rindiņas simbols) vai faila beigas. Funkcija *getline()* augsta līmeņa simbolu virknēm nav faila objekta funkcija, tāpēc faila (vai cits izvades) objekts tai tiek padots kā parametrs.

Pirmkods 7.10. Teksta faila pārrakstīšana citā failā pa rindai (*txt10getline.cpp*)

```
01 #include <fstream>
02 using namespace std;
03
04 int main ()
05 {
06     string s;
07     fstream fin ("txt_in9.txt", ios::in);
08     fstream fout ("txt_out10.txt", ios::out);
09     getline (fin, s);
10     while (fin)
11     {
12         fout << s << endl;
13         getline (fin, s);
14     };
15     fin.close ();
16     fout.close ();
17     return 0;
18 }
```

Programmas darbības piemērs (‘·’ tukšums, ‘¶’ jaunas rindiņas simbols, ‘§’ faila beigas)

txt_in9.txt (ievade)

```
This¶
is·an¶
example.¶
§
```

txt_out10.txt (izvade)

```
This¶
is·an¶
example.¶
§
```

konsole (izvade)

(nekas netiek izvadīts)