

2. C++ programmas uzbūves pamatprincipi

2.1. C++ izcelšanās

Valoda C++ ir izcēlusies no valodas C, kuru 1970. gadā izstrādājis AT&A Bell Laboratories līdzstrādnieks **Deniss Ričijs** (*Dennis Ritchie*). Valodas C sākotnējais uzdevums bija operētājsistēmas UNIX veidošanai un uzturēšanai. Tomēr šī valoda kļuva ļoti populāra un to sāka izmantot lietojumprogrammu izstrādei, kā arī citās operētājsistēmās. Valoda C uzskatāma par kaut ko, kas ir pa vidu starp ļoti augsta un zema līmeņa programmēšanas valodām. Kaut arī sistēmprogrammu rakstīšanai C uzskatāms par ļoti piemērotu, tomēr lietojumprogrammatūras izstrādē valodai C piemīt vairāki trūkumi – tajā rakstītās programmas nav tik viegli uztveramas un saprotamas kā citu augsta līmeņa programmēšanas valodu programmas, turklāt tajā nav daudzu iebūvētu automātiskās pārbaudes iespēju (resp., augsta līmeņa konstrukciju) kā pierasts daudzās citās valodās.

Lai kompensētu valodas C nepilnības, 20. gs. 70. gadu beigās – 80. gadu sākumā AT&A Bell Laboratories līdzstrādnieks **Bjerns Stroustrups** (*Bjarne Stroustrup*) izstrādāja programmēšanas valodu C++, kas daudzos gadījumos ir izrādījusies labāka par savu priekšteci. Tajā ieviestas daudzas augsta līmeņa konstrukcijas, kas atvieglo programmētāja darbu. Bez tam C++ ir saglabājis savietojamību ar valodu C (lielākā daļa C programmu ir arī C++ programmas), kas savulaik visticamāk bija izšķiroši, lai C++ iegūtu savu plašo pielietojumu un popularitāti.

Uz valodu C++ vēl lielākā mērā attiecas tas, ko savulaik attiecināja uz valodu C – tajā ir kopā apvienotas gan zema, gan ļoti augsta līmeņa konstrukcijas, kas vēl lielākā mērā, kā tas bija valodai C, ir valodas C++ spēks un reizē arī tās vājums. Tās vājums galvenokārt izpaužas valodas C++ sarežģītībā un plašajā apjomā, kam ir divas galvenās sekas: ir salīdzinoši grūti izveidot kompilatoru (programmu, kas no pirmkoda teksta izveido izpildāmu kodu) un tā prasa augstāku profesionalitāti no programmētāja. Vairāk kā divus gadu desmitus pēc C++ izcelšanās var uzskatīt, ka pirmā problēma ir atrisināta – ir uzbūvēti pietiekoši kvalitatīvi un efektīvi kompilatori, un ērtas izstrādes vides, atliek tikai programmētāja profesionalitāte. Tāpēc, iegūstot nepieciešamās zināšanas un iemaņas, valoda C++ programmētāja rokās var kļūt par tik spēcīgu rīku, ar kuru var cerēt mēroties spēkiem tikai retā programmēšanas valoda.

2.2. C++ programmas failu struktūra

Programma valodā C++ sastāv no viena vai vairākiem **C++ failiem** (faila paplašinājums parasti `.cpp` vai `.cc`), kā arī iespējami no citiem failiem, no kuriem tipiskākie ir t.s. **hedera** (jeb galvas) faili (faila paplašinājums `.h`). Tātad, triviālā C++ programma sastāv no viena C++ faila. Triviāla C++ programma parādīta Att. 2-1. Šī programma, kā jau redzams tās darbības piemērā, uz ekrāna izdrukā tekstu “Hello, world!”.

Sākot ar šo programmu un turpmāk materiālā programmas vai to fragmenti tiks marķēti ar biezu svītru programmas koda kreisajā pusē, pēc tam var sekot **programmas darbības piemērs**, kurš marķēts ar trīskāršo svītru kreisajā pusē. Ja programmas darbības piemērā parādās lietotāja ievads no klaviatūras, tas tiek atzīmēts treknināti (*bold*).

Att. 2-1. Triviāla C++ programma.

```
#include <iostream>
using namespace std;
```

```

int main ()
{
    cout << "Hello, world!" << endl;
    return 0;
}

Hello, world!

```

Ko pēc šīs vienkāršās programmas var spriest vai sākt stāstīt par C++ programmas struktūru:

- Programmas galveno daļu (to, kas veic darbības) veido instrukcijas, kuru pieraksts beidzas ar semikolu.
- Programmas instrukcijas var tikt apvienotas blokos, ko ietver figūriekavas { }.
- Parasti figūriekavu bloks ir kādas funkcijas daļa, kuras nosaukums ir rindā pirms figūriekavu bloka – šajā gadījumā tā ir funkcija *main*.
- Programma var sastāvēt arī no vairākām funkcijām, bet vismaz viena funkcija – funkcija *main* noteikti atrodas programmā, un tieši no šīs funkcijas sākas programmas izpilde.
- Jebkurā funkcijā parasti atrodas vismaz viena (un parasti tieši viena) rindiņa, kas satur atslēgas vārdu *return*, un kas saistīta ar funkcijas izpildes beigšanu (t.i., izeju no funkcijas).
- C++ programma praktiski nav iedomājama bez kādu bibliotēku iekļaušanas (tas tādēļ, ka, atšķirībā no daudzām citām programmēšanas valodām, pat relatīvi ikdienišķas lietas, kā, piemēram, ievads, izvads un failu apstrāde, nav iekļautas valodas kodolā, no programmēšanas viedokļa gan tas ir tikai sintakses jautājums). Vienas bibliotēkas iekļaušanu nodrošina komanda *#include*, pēc kā seko bibliotēkas vārds stūra iekavās <>:

šajā gadījumā tiek iekļauta bibliotēka <iostream>, kas atbild par standarta ievadi un izvadi (no klaviatūras, uz ekrāna),

ir divu veidu standarta bibliotēkas – vecā un jaunā tipa, vecās beidzas ar “.h”, bet jaunās nē,

mūsu programmā iekļautā bibliotēka <iostream> ir jaunā tipa bibliotēka, viņai atbilstošā vecā tipa bibliotēka būtu <iostream.h>,

abu tipu bibliotēkas darbojas līdzvērtīgi, bet sintaktiski ir viena atšķirība – ja programmā ir **vismaz viena jaunā** bibliotēka (un mūsu programmā tāda ir), tad aiz bibliotēku iekļaušanas papildus jālieto rinda *using namespace std;* (tas ir ļoti vienkāršots šīs rindiņas nozīmes izskaidrojums),

programmā bez standarta bibliotēkām var tikt izmantotas arī citas bibliotēkas, tādā gadījumā to failu nosaukumus liek nevis stūra iekavās, bet gan dubultpēdiņās, piemēram *#include "test.h"*.

Lūk, divi piemēri ar alternatīvu bibliotēkas *iostream* iekļāvumu vecajā un jaunajā variantā.

```

#include <iostream.h>

#include <iostream>
using namespace std;

```

Tādējādi mūsu programmā vienīgā rindiņa, kas reprezentē algoritmu (teksta izdrukāšanu uz ekrāna) ir

```

cout << "Hello, world!" << endl;

```

Viss pārējais ir valodas C++ specifisks teksts, no algoritma viedokļa varētu teikt – “formalitāte”. Turpmāk, aprakstot dažādas programmas konstrukcijas, “formalitātes” parasti tiks izlaistas, ja nu vienīgi iespējams tiks nosauktas nepieciešamās standarta bibliotēkas noteiktas konstrukcijas izmantošanai.

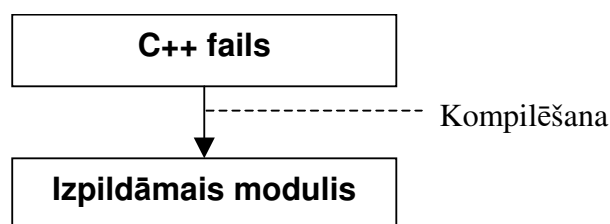
2.3. C++ programmas kompilēšana un palaišana

2.3.1. Kompilēšanas pamatprincipi

Programmas rakstīšanas mērķis parasti ir iegūt izpildāmu moduli (piemēram .exe failu), kas veic noteiktas darbības. Lai no programmas iegūtu izpildāmo moduli, nepieciešams kompilators.

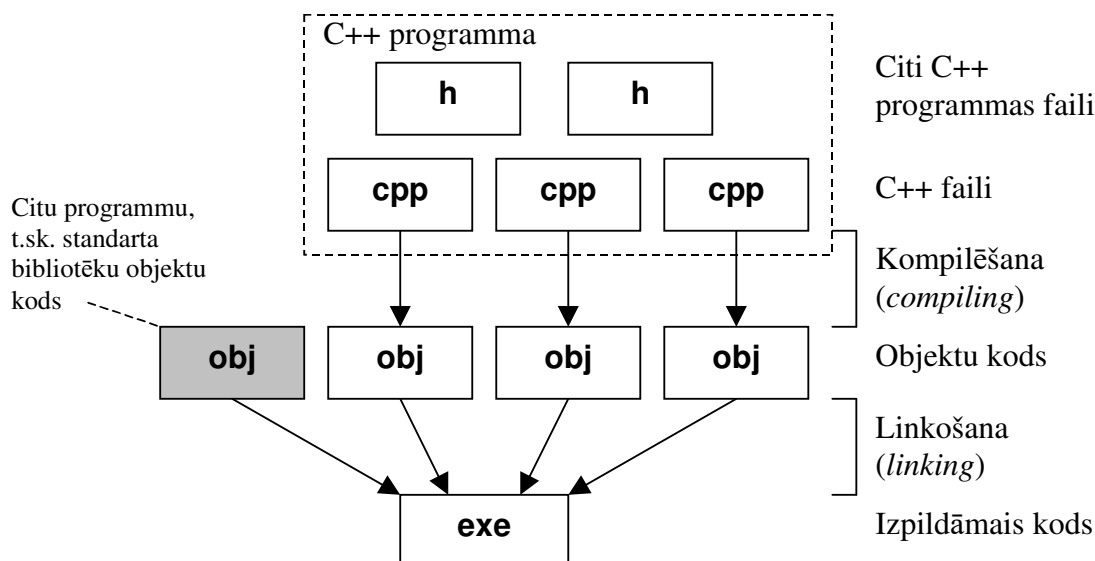
Kompilators (*compiler*) ir datorprogramma, kas pārveido noteiktā programmēšanas valodā uzrakstītu programmu par izpildāmu moduli (vispārīgā gadījumā – par programmu mašīnkodā, tādu, kuru dators var tiešā veidā izpildīt).

Att. 2-2. Izpildāma moduļa triviāla iegūšana no C++ programmas.



Lielākā daļa no C++ izstrādes vidēm dod iespēju veikt triviālas C++ programmas kompilēšanu (izpildāmā moduļa iegūšanu) tādā veidā kā parādīts Att. 2-2. Tomēr dažas izstrādes vides pieprasa veidot t.s. projektu, kas apskatīts laboratorijas darbu daļā. Kā jau tika minēts, vispārīgā gadījumā C++ programma sastāv no vairākiem failiem, turklāt īpaša vieta tajā ir C++ failiem (sk. Att. 2-3).

Att. 2-3. Izpildāma moduļa iegūšana no C++ programmas vispārīgā gadījumā.



Kaut arī sarunu valodā visu izpildāmā moduļa iegūšanas procesu sauc par kompilēšanu, tomēr šis process sastāv no divām fāzēm, no kurām par kompilēšanu sauc tikai pirmo:

1. fāze. Kompilēšana. Katram C++ failam tiek uzbūvēts atbilstošs **objektu fails** (faila paplašinājums parasti *.obj* vai *.o*), kopumā veidojot **objektu kodu**;

2. fāze. Linkošana. No iegūtā objektu koda, kā arī citu programmu, t.sk. standarta bibliotēku objektu koda tiek izveidots gala produkts – izpildāmais kods (izpildāmais modulis vai moduļi).

Pilnu izpildāmā koda iegūšanas procesu mēdz saukt arī par **uzbūvēšanu** (*build*), tomēr sarunu valodā, kā jau tika minēts, tiek lietots termins kompilēšana.

Abas izpildāmā koda uzbūvēšanas fāzes parasti tomēr veic viens un tas pats kompilators. Programmas failu organizāciju, kā arī pareizu kompilatora un citu palīgprogrammu izsaukšanas secību izpildāmā koda iegūšanai parasti nodrošina izstrādes vide (piemēram, *Dev-C++*, *Microsoft Visual C++*, *Borland C++*, *Anjuta*), izmantojot **projektu** (*project*) mehānismu, tādējādi programmētājs var abstrahēties no šīs shēmas un “ar vienas pogas spiedienu” nonākt no programmas līdz rezultātam. Otrs bieži lietots variants programmas izpildāmā koda uzbūvēšanai, kas īpaši izplatīts *Unix/Linux* platformās, ir t.s. **make** mehānisms.

Turpmāk, apgūstot dažādas C++ konstrukcijas, parasti pietiks ar C++ programmām ar viena faila struktūru.

2.3.2. Programmas kompilēšana

Programmas kompilēšana, izmantojot noteiktu izstrādes vidi, parasti iespējama vismaz divos variantos: **triviālajā** (bez projekta veidošanas) un **parastajā** (ar projekta veidošanu). Jebkura izstrādes vide, kompilējot programmu, vienu vai vairākas reizes izsauc kompilatoru (piemēram, izstrādes vide *Dev-C++* izmanto kompilatorus *gcc* un *g++*).

2.3.2.1. Triviālais variants programmas kompilēšanā

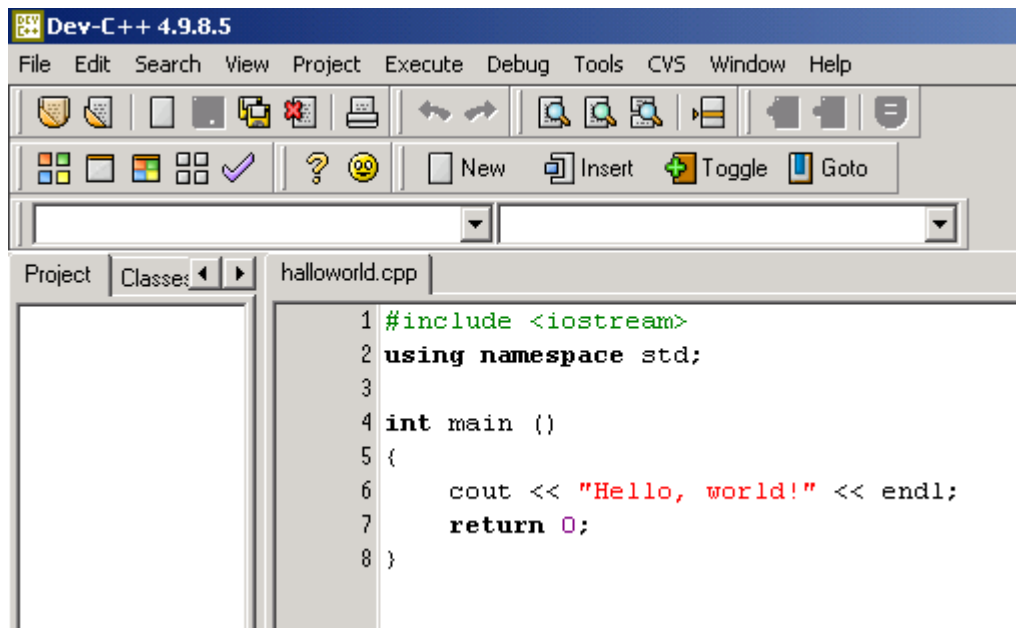
Triviālais variants pieejams tikai programmām, kuras sastāv no viena *.cpp* faila (citi iekļautie faili, piemēram, *.h*, papildus drīkst būt). Bez tam, lai programmu kompilētu šajā variantā, iegūstamajam izpildāmajam modulim ir jābūt teksta režīmā jeb ar konsoles tipu (triviālais variants var nederēt, ja būs nepieciešama aplikācija ar logu izmantošanu). Triviālais variants ir ļoti ērts izmantošanā, un to atbalsta gandrīz visas izstrādes vides.

Programmas kompilēšana triviālajā variantā (Att. 2-2, Att. 2-4):

atvērt *.cpp* failu ar izstrādes vides teksta redaktoru,

ar komandu *compile* vai *build* veikt kompilēšanu, iegūstot izpildāmo moduli.

Att. 2-4. Faila atvēršana izstrādes vidē Dev-C++.



Triviālajā variantā kompilētas programmas rezultāts parasti ir tikai izpildāmais modulis (piemēram, .exe fails).

2.3.2.2. Projekta veidošana un kompilēšana izstrādes vidē

Ja programmu veido vairāki .cpp faili, tad izstrādes vides ietvaros jāveido t.s. projekts. Projekts ir konkrētās izstrādes vides specifisks jēdziens un nav C++ jautājums, tāpēc katrā izstrādes vidē darbs ar projektu varētu nedaudz atšķirties.

C++ programmas kompilēšana, veidojot projektu no jauna:

- izveidot jaunu projektu,

- piesaistīt visus programmas .cpp failus, kā arī neobligāti citus failus, projektam – tas jāveic ar izstrādes vides specifiskiem līdzekļiem – failus projektā prasti ir iespējams grupēt folderos,

- ar komandu *compile* vai *build* veikt kompilēšanu, iegūstot izpildāmo moduli.

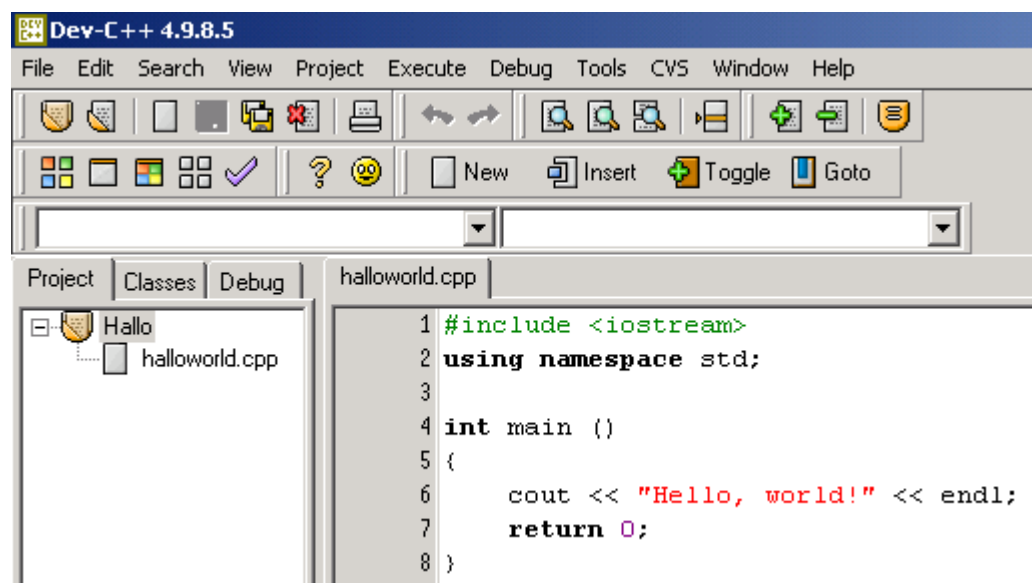
Pēc projekta izveidošanas to parasti reprezentē speciāls projekta fails, kura paplašinājums ir atkarīgs no izstrādes vides. Šajā failā (parasti teksta režīmā) ir uzskaitīti visi projektā ietilpstošie faili, kā arī dažādi projekta konfigurācijas parametri.

C++ programmas kompilēšana izveidotam projektam:

- atvērt projekta failu ar izstrādes vides (teksta) redaktoru,

- ar komandu *compile* vai *build* veikt kompilēšanu, iegūstot izpildāmo moduli.

Att. 2-5. Projekts “Hallo” izstrādes vidē Dev-C++.



Projekta kompilēšanas rezultātā izveidojas vismaz šādi faili:

katram C++ failam viens objekta fails (.obj vai .o)

izpildāmais modulis (piemēram .exe).

2.3.2.3. Programmas kompilēšana komandrindas režīmā

Programmu iespējams nokompilēt arī, neizmantojot izstrādes vidi. Lai to izdarītu, jāveic ar kompilatoru tiešā veidā jāveic šādas darbības:

kompilators kompilēšanas režīmā vienreiz jāizsauc katram C++ failam, lai izveidotu atbilstošo objekta failu,

kompilators jāizsauc linkošanas režīmā, uzskaitot visus iepriekšējā solī izveidotos objekta failus un izveidojot izpildāmo moduli.

Ja programma sastāv no viena faila, tad kompilēšana, lai iegūtu izpildāmo moduli, var izdarīt arī vienā solī.

Izplatīti kompilatori dažādās platformās ir *gcc/g++*. Tos izmanto arī izstrādes vide *Dev-C++*.

2.3.3. Programmas palaišana

Programmu var palaist tikai tad, ja programma ir līdz galam nokompilēta un ir izveidots izpildāmais modulis.

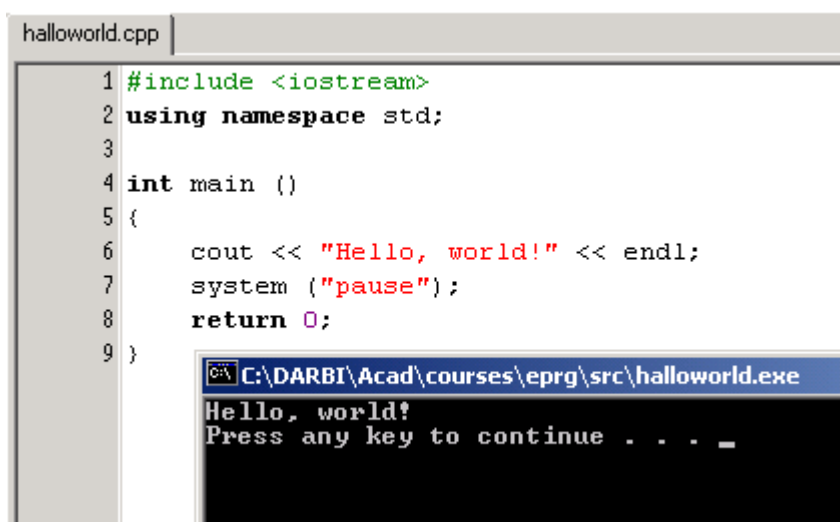
Programmas palaišana no izstrādes vides parasti notiek ar programmu “Run”, tomēr, ja izstrādes vide ir grafiska, bet programma darbojas teksta (konsoles) režīmā, var rasties problēma (un tāda problēma ir arī vidē Dev-C++): spiežot komandu “Run”, kaut kas nozibsnī, bet programmas darbības rezultātu ieraudzīt nevar. Tas tādēļ, ka programma speciāli savai izpildei atver jaunu konsoli (melnu lodziņu), bet tūlīt pēc darbības (šajā gadījumā “Hello, World!” izdrukāšanas), konsole tiek aizvērta (šāda problēma visticamāk nebūs *Linux/Unix* bāzētās izstrādes vidēs, kur konsoles izsaukšana būs iekļauta izstrādes vidē).

Lai šo problēmu atrisinātu, jāveic viena no divām darbībām:

Pirms programmas beigām programmā jāieraksta viena papildus rindiņa, kas “pietur” programmu, lai tā nebeidzas (un tādējādi neizver konsoles logu). Tam der jebkura ievades operācija, bet Windows vidē ir pieejama komanda *system ("pause")*;; kas pietur programmu, prasot nospiegt jebkuru taustiņu lai turpinātu.

Vispirms jāatver konsole pašam (piemēram, Windows vidē – *START-Run + cmd*), tad konsolē komandrindas režīmā jāaiziet uz izpildāmā koda direktoriju un tiešā veidā jāpalaiž programma.

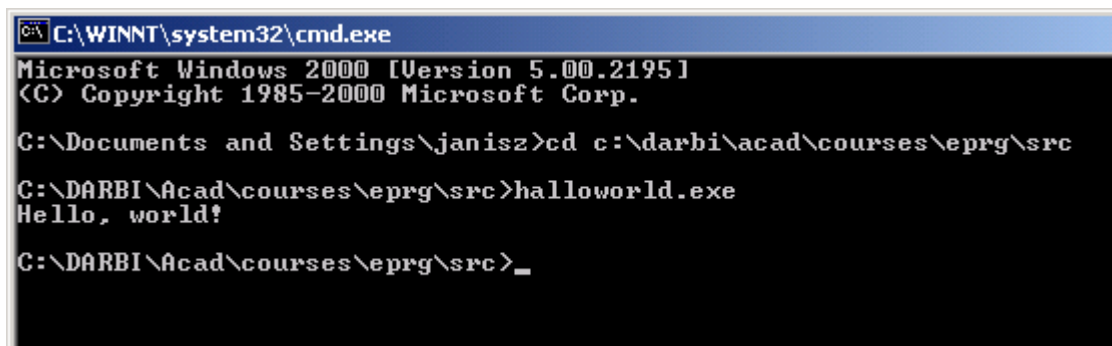
Att. 2-6. Programmas palaišana no izstrādes vides ar RUN, programmā izmantojot papildus rindiņu programmas “pieturēšanai” pirms beigām.



```
helloworld.cpp
1 #include <iostream>
2 using namespace std;
3
4 int main ()
5 {
6     cout << "Hello, world!" << endl;
7     system ("pause");
8     return 0;
9 }
```

C:\DARBI\Acad\courses\eprg\src\helloworld.exe
Hello, world!
Press any key to continue . . . _

Att. 2-7. Programmas palaišana no konsoles komandrindas režīmā.



```
C:\WINNT\system32\cmd.exe
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\Documents and Settings\janisz>cd c:\darbi\acad\courses\eprg\src
C:\DARBI\Acad\courses\eprg\src>helloworld.exe
Hello, world!
C:\DARBI\Acad\courses\eprg\src>_
```

2.4. C++ programmas vispārējā struktūra

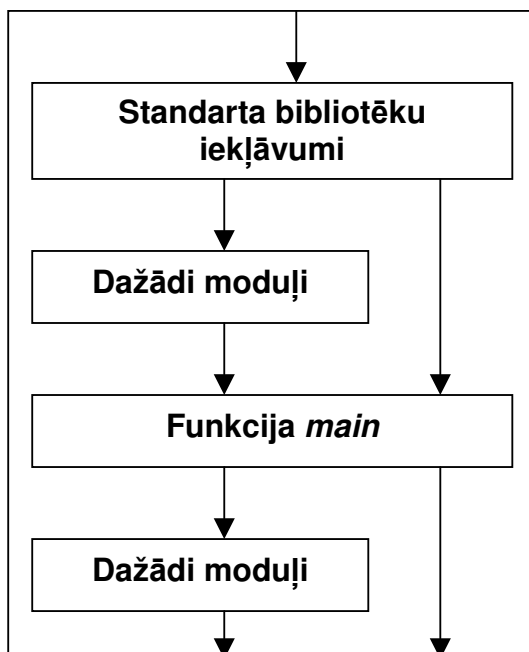
2.4.1. Programmas struktūras galvenā shēma

Valodas C++ programma (kā jebkurā citā programmēšanas valodā rakstīta programma) ir konstrukciju kopums, kas apraksta noteiktu algoritmu.

C++ programma no valodas konstrukciju viedokļa ir galvenokārt dažādu C++ moduļu kopums, starp kuriem viens ir galvenais modulis jeb t.s. galvenā funkcija *main*. Tādējādi triviāla C++ programma sastāv tikai no funkcijas *main*, kas atbilst jēdzienam “galvenā programma” dažās citās programmēšanas valodās (sk. Att. 2-1). Bez tam, atšķirībā no daudzām citām programmēšanas valodām, ikviena C++ programma (arī pati vienkāršākā) ir

praktiski neiedomājama bez vismaz kādas standarta bibliotēkas iekļāvuma. Tas tādēļ, ka daudzas tipiskas funkcijas (piemēram, ievade, izvade, failu apstrāde, simbolu virkņu apstrāde, matemātiskās operācijas) nav iekļautas valodas kodolā.

Att. 2-8. C++ programmas vienkāršota struktūra.



No leksiskā viedokļa C++ programmas struktūru veido **leksēmas** (*lexemes*) jeb **marķieri** (*tokens*):

- identifikatori (*identifiers*),
- atslēgas vārdi (*keywords*),
- literāļi jeb konstantās vērtības (*literals*),
- operatoru marķieri (*operator tokens*) un
- dienesta simboli (*punctuators*).

Sintaktiskā līmenī programmu veido:

- mainīgie (*variables*),
- konstantes (*constants*), t.sk., literāļi,
- datu tipi (*data types*),
- funkcijas (*functions*),
- izteiksmes (*expressions*).

Konstrukciju līmenī varētu teikt, ka C++ programmas sastāvā ietilpst:

- priekšraksti (*statements*),
- deklaratori (*declarators*),
- bloki (*blocks*),
- (konstrukciju) galvas.

Bez tam ikviena C++ programma nav iedomājama bez **preprocesēšanas** (priekšapstrādes) **komandām**, no kurām tipiskākās ir bibliotēku iekļaušanas komandas *#include*.

2.4.2. Programmas piemērs

Vienkāršākajā gadījumā C++ programma ir funkciju kopums, kurā tieši viena saucas *main*. Tieši no funkcijas *main* sākās C++ programmas izpilde. Izņēmums ir vienīgi *Windows* specifiskas programmas, kur programmu ievadošā funkcija tiek saukta *WinMain*. Nezinot, kas ir C++ funkcija, sākumā pietiek uztvert funkciju kā figūriekavu blokā ietvertu instrukciju virkni, pirms kura ir funkcijas galva, kas cita starpā ietver arī funkcijas nosaukumu. Programma Att. 2-9 demonstrē vienkāršu C++ programmu, kas veic skaitļa ievadi, tā sareizināšanu ar citu skaitli un rezultāta izvadi.

Att. 2-9. Programmas piemērs valodā C++, kas izrēķina ievadītā skaitļa reizinājumu ar 1.2.

```
#include <iostream>
using namespace std;

int main ()
{
    double x, y, z;
    cout << "Input a numeric value:" << endl;
    cin >> x;
    y = 1.2;
    z = x * y;
    cout << x << '*' << y << '=' << z << endl;
    return 0;
}

Input a numeric value:
2.5
2.5*1.2=3
```

No leksiskā viedokļa šī programma ietver šādus elementus:

identifikatori (ieskaitot rezervētos vārdus): std, main, x, y, z, cout, endl, cin;

atslēgas vārdi: using, namespace, int, double, return

literāļi: [1.2] ["Input a numeric value:"] ['*'] ['='] [0]

operatori un dienesta simboli: () {} , ; << >> = *

No sintaktiskā viedokļa programma ietver šādus elementus:

mainīgie: x, y, z, cout, cin

konstantes: visi literāļi un [endl]

datu tipi: int double

funkcijas: main

izteiksmes: (piemēram) [x * y]

No konstrukciju viedokļa programmā ir izdalāmas šādas daļas:

priekšraksti: (piemēram) [cin >> x;] [z = x * y;]

[cout << "Input a numeric value:" << endl;]

deklaratori: [using namespace std;] [double x, y, z;]

bloki: [{ ... }]

galvas [int main ()]

Bez tam programmā ir viena preprocesora komanda: bibliotēkas *iostream* iekļāvums.

2.4.3. Komentāri

Komentāri ir tādi programmas fragmenti, kas neietekmē programmas darbību (resp., izpildāmā koda veidošanu), jo kompilators tos ignorē. Komentārus programmētājs lieto, lai pierakstītu papildus informāciju pie programmas, kas, piemēram, vēlāk ļautu vieglāk lasīt programmu.

Valodā C++ ir divu veidu komentāri:

- Vienas rindas komentāri – sākas ar simbolu pāri `//` un beidzas līdz ar rindas beigām.
- Vairāku rindu komentāri – sākas ar simbolu pāri `/*` un beidzas ar simbolu pāri `*/`.

Nākošais piemērs demonstrē abu veidu komentārus:

Att. 2-10. Komentāri C++ programmā.

```
/* *****  
**  
** Programma "Hello, World"  
** Izveidota 2007.04.05.  
** Labota 2007.04.09.  
** Autors Mr. X  
**  
***** */  
  
#include <iostream>  
using namespace std; /*  
tā bija bibliotēku iekļaušana */  
  
// galvenā funkcija  
int main ()  
{  
    cout << "Hello, world!" << endl;  
    return 0; // beidz funkcijas darbu  
}
```

Šajā programmā ir četri komentāri – pa divi no katra veida:

```
/* *****  
**  
** Programma "Hello, World"  
** Izveidota 2007.04.05.  
** Labota 2007.04.09.  
** Autors Mr. X  
**  
***** */  
  
/*  
tā bija bibliotēku iekļaušana */  
  
// galvenā funkcija
```

15. Mantošana un citi objektorientētās programmēšanas mehānismi

15.1. Mantošanas pamatprincipi

Mantošana (*inheritance*) ir objektorientētās programmēšanas mehānisms, kad klase pārņem (manto) elementus (laukus un metodes) no jau eksistējošas klases vai pat vairākām klasēm.

Mantošana ir vēl viens mehānisms pirmkoda vairākkārtējai izmantošanai, kas papildina koda strukturēšanas iespējas, kā arī nodrošina ērtāku uzturēšanu.

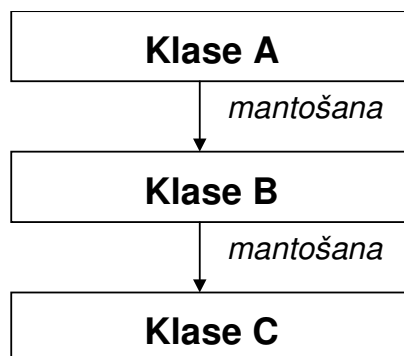
Klases, no kurām mantošanas rezultātā tiek iegūti elementi, sauc par **bāzes klasēm** (*base class*), bet klasi, kura tiek veidota, izmantojot mantošanu – par **atvasināto klasi** (*derived class*) vai **mantoto** (*inherited*) klasi.

Att. 15-1. Vienkārša mantošanas shēma.



Mantošanas process var būt kaskādes veida, līdz ar to viena un tā pati klase var būt gan atvasinātā klase, gan bāzes klase citām klasēm.

Att. 15-2. Kaskādes veida mantošanas shēma.



Klases mantošana notiek, pēc klases nosaukuma uzrādot klasi vai klases, no kurā tiks mantots (pēc līdzīgas sintakses, kā konstruktorā alternatīvajā variantā tiek inicializēti lauki).

Att. 15-3. Mantošanas shēma: klase B manto īpašības no klases A.

```
class A
{
    ...
};
...
class B: public A
{
```

```
}; ...
```

Mantošanas rezultātā visi A lauki un metodes automātiski ir arī klases B elementi, tomēr ir dažādi izņēmumi un sarežģījumi, par kuriem runāts tālāk.

Att. 15-4. Vienkāršota klases galvas definīcija ar mantošanu (*class_definition_inheritance*).

```
class_definition_inheritance:
    class_keyword class_name inherit_list_opt { class_section_list }

class_keyword: one of
    class struct union

inherit_list:
    : base_class_list

base_class_list:
    base_class
    base_class_list base_class

base_class:
    inheritance_degree class_name

inheritance_degree: one of
    public private
```

Šajā shēmā nav precizēts jēdziens *class_section_list*, kas ir tāds pats, kā iepriekšējā nodaļā vienkāršotajā klases definēšanas shēmā.

Nākošajā programmas piemērā klase *twonumbers* tiek mantota no klases *number*, tāpēc saturs gan lauku *a*, gan lauku *b*, turklāt, definējot klases *twonumbers* metodi *print*, ar noteiktu sintaksi ir izmantojama klases *number* metode *print*, lai arī tajā atrodošais nebūtu otrreiz jāprogrammē.

Att. 15-5. Vienkāršs klases mantošanas piemērs (*twonumbers* manto *number*).

```
#include <iostream>
using namespace std;

struct number // bāzes klase ar vienu lauku a
{
    int a;
    void print () const { cout << a << endl; };
};

struct twonumbers: public number // atvasinātā klase
{
    // mantošanas dēļ šeit ir arī lauks a, kā arī
    // metode number::print(), kas izdrukā lauku a
    int b;
    void print () const
    {
```

```

        number::print (); // bāzes klases print!
        cout << b << endl;
    };
};

int main ()
{
    twonumbers num;
    num.a = 5;
    num.b = 7;
    num.print ();
    return 0;
}

```

15.2. Mantošanas procesu nodrošinošie mehānismi

Mantošanas mehānisma izmantošana izsauc nepieciešamību pēc vairākām papildus konstrukcijām un mehānismiem objektorientētajā programmēšanā:

- modifikators *protected* paplašina *private* redzamību arī uz mantotajām klasēm,
- konstruktoru kaskādes veida izsaukšana,
- destruktoru kaskādes veida izsaukšana,
- atklātā un slēptā mantošana,
- virtuālās funkcijas.

15.2.1. Konstruktoru kaskādes veida izsaukšana

Pirms izsaukt atvasinātas klases konstruktoru, obligāti jāveic arī bāzes klases konstruktoru izsaukšana. Tādējādi tiek panākts, ka, izveidojot objektu, tiek izsaukti arī visu bāzes klašu konstruktori mantošanas secībā, un tikai tad pašas klases konstruktors. Šo kaskādes shēmu valoda C++ nodrošina sintakses līmenī.

Konstruktoru galva ar bāzes klases konstruktoru izsaukšanu:

```
twonumbers (int x, int y): number (x)
```

Tā kā klasē var būt vairāki konstruktori, tad, definējot atvasinātas klases konstruktoru, ar īpašu sintaksi tiek norādīts, kuru no bāzes klases konstruktoriem pirms tam izsaukt. Ja bāzes klases konstruktors netiek uzrādīts, tad bāzes klasē jāeksistē konstruktoram bez parametriem, kas tādā gadījumā tiek izsaukts.

Att. 15-6. Konstruktoru kaskādes veida izsaukšana.

```

#include <iostream>
using namespace std;

class number
{
    int a; // tiešā veidā nav pieejams pat klasei twonumbers
public:

```

```

        number (int x) { a = x; };
        void print () const { cout << a << endl; };
};

class twonumbers: public number
{
    int b;
public:
    // tiek izsaukts bāzes klases konstruktors number(x)
    twonumbers (int x, int y): number (x)
    {
        b = y;
    };
    void print () const
    {
        number::print ();
        cout << b << endl;
    };
};

int main ()
{
    twonumbers num (5, 7);
    num.print ();
    return 0;
}

```

5
7

Pievērsiet uzmanību atvasinātās klases konstruktoram:

```

    twonumbers (int x, int y): number (x)
    {
        b = y;
    };

```

Ja konstruktors būtu realizēts nevis šādi, bet gan sekojoši (tiešā veidā aizpildot visus laukus):

```

    twonumbers (int x, int y)
    {
        a = x;
        b = y;
    };

```

tad dotajā kontekstā (sk. pilnu klases definīciju) iestātos divas kļūdas situācijas:

tā kā aiz atvasinātās klases konstruktora nav uzrādīts bāzes klases konstruktors (bet tā izsaukšana ir obligāti), kompilators mēģina izsaukt noklusēto konstrukturu (bez parametriem), bet tāds bāzes klasē nav definēts (tur ir tikai konstruktors ar vienu parametru):

```
no matching function for call to `number::number()'
```

tiešā veidā tiek piekļūts mainīgajam a : $a = x$, kas ir definēts bāzes klasē kā *private*, tāpēc iestājas piekļūšanas kļūda:

```
`int number::a' is private
```

15.2.2. Destruktoru kaskādes veida izsaukšana

Arī destruktori, tāpat kā konstruktori, likvidējot objektu, tiek izsaukti kaskādes veidā, tikai:

izsaukšana notiek pretēji mantošanas secībai – vispirms pašas klases destruktors, tad bāzes klašu destruktori utt.,

destruktoru kaskādes veida izsaukšana nav speciāli jādefinē, jo, sakarā ar to, ka katrā klasē ir tikai viens destruktors, izsaukšana notiek automātiski.

15.2.3. Atklātā un slēptā mantošana

Atklāto un slēpto mantošanu nosaka attiecīgi atslēgas vārdi *public* un *private* pirms bāzes klases nosaukuma, veicot mantošanas procesu (sk. jēdzienu *inheritance_degree* Att. 15-4).

ja tiek mantots atklāti (*public*, kā piemērā parādīts), tad pieeja pie bāzes klases elementiem paliek tāda pati arī mantotajā klasē (attiecīgi *public* vai *protected*, jo *private* elementi tāpat nav redzami),

ja tiek mantots slēpti (*private*, piemēros nav parādīts), tad visi bāzes klases elementi nonāk mantotajā klasē *private* statusā, tātad, ir pieejami tikai no mantotās klases.

15.2.4. Virtuālās funkcijas

Parasti mantošanu ir jēga izmantot tad, ja bāzes klasei ir vairāk nekā viena atvasinātā klase vai arī ja arī pašai bāzes klasei tiek veidoti objekti, jo tikai šajā gadījumā tā īsti parādās pirmkoda koplietošanas princips.

Bieži rodas nepieciešamība no bāzes klases mantoto klašu un varbūt arī pašas bāzes klases objektus glabāt vienotā struktūrā, piemēram, kopējā masīvā. C++ dod tādu iespēju – pie mainīgā ar tipu “norāde uz bāze klasi” drīkst veidot objektus ar tipu “atvasinātā klase” (klašu definīcijas sk. attēlā Att. 15-6):

```
...
struct number
{
    ...
    class twonumbers: public number
    {
        ...
    }
}

int main ()
{
    number* nums[2];
    nums[0] = new number (99);
    nums[1] = new twonumbers (5, 7);
    nums[0]->print ();
    nums[1]->print ();
    ...
}

99
5
```

Kā redzams, komanda

```
nums[1]->print ();
```

acīmredzami ir nozīmējusi bāzes klases *print()* izsaukumu, jo nav izdrukāts skaitlis 7.

Mainīgo tipa (šeit, norāde uz *number*) un objekta tipa (šeit, *twonumbers*) neatbilstība var radīt problēmas gadījumos, ja mantotajā klasē **tiek pārdefinētas bāzes klases funkcijas** (šeit, funkcija *print()*) – rodas problēmas saprast, kuru funkciju izsaukt, vai bāzes klases, vai atvasinātās klases attiecīgo funkciju. Standarta tipa noteikšanas mehānisms ir “pēc mainīgā tipa”, tādējādi, pielietojot šo mehānismu, tiek izsaukta objektam neatbilstoša funkcija.

Lai novērstu šo mainīgā un objekta tipa neatbilstības problēmu, bāzes klasē attiecīgā funkcija jāatzīmē kā **virtuāla**, pirms funkcijas pielietojot atslēgas vārdu *virtual* (pilnas klašu definīcijas sk. attēlā Att. 15-6):

```
...
struct number
{
    ...
    virtual void print () ...
    ...
class twonumbers: public number
{
    ...
int main ()
{
    number* nums[2];
    nums[0] = new number (99);
    nums[1] = new twonumbers (5, 7);
    nums[0]->print ();
    nums[1]->print ();
    ...
99
5
7
```

Un 7 tiek izdrukāts!

Ievērojiet, ka atslēgas vārdu *virtual* jāpielieto bāzes klasei, nevis atvasinātajai.

15.3. Citi objektorientētās programmēšanas mehānismi

15.3.1. Draugu funkcijas un klases

Elementu redzamību klasē pamatā nodrošina modifikatori *public*, *protected* un *private*. Šie ir vispārīgas nozīmes modifikatori, kas nav atkarīgi no moduļa, no kura varētu tikt veikta pieeja klases elementiem.

Valodā C++ ir iespēja klases iekšienē definēt klases draugus – funkcijas vai klases, kam ir atļauta pieeja arī tiem klases elementiem, kas nav publiski. Drauga statusa uzstādīšanu nodrošina atslēgas vārds *friend* potenciālās drauga klases deklarācijas vai drauga funkcijas prototipa priekšā dotās klases hederī.

```
friend void print (number &num);
friend class another_class;
```


Tas nodrošina visu klases elementu pieeju no dotajiem moduļiem (funkcijām vai klases metodēm).

Att. 15-7. Drauga funkcija.

```
#include <iostream>
using namespace std;

class number
{
    int a;
public:
    number (int x) { a = x; };
    friend void print (number &num);
};

void print (number &num)
{
    cout << num.a << endl;
}

int main ()
{
    number num (99);
    print (num);
    return 0;
}
```

|||

99

Šajā piemērā rindiņa:

```
friend void print (number &num);
```

ļauj ārējai (no klases *number* viedokļa) funkcijai *print* piekļaut klases *number* slēptajam (*private*) elementam *a*:

```
num.a
```

15.3.2. Operatoru pārslogošana

Valodā C++ ļoti plaši pielieto operatorus, turklāt ir iespēja tos, līdzīgi funkcijām, pārslogot, tādējādi pielāgojot darbam ar dažādiem tiem. Pārslogot var gandrīz jebkuru operatoru, izņemot šos:

```
.  .*  ::  ?:
```

Sintaktiski iespēju pārslogot nodrošina **operatoru funkcionālais pieraksts**, kurā operators tiek izsaukts vai definēts kā parasta funkcija un līdz ar to arī ir pārslogojams kā parasta funkcija (ar nelielām specifiskām niansēm).

Operatora funkcionālais nosaukums sastāv no atslēgas vārda *operator*, kam seko operatora simbols vai simboli, piemēram, *operator+ operator*= operator() operator[]*

Piemēram, rindiņa

```
c = a + b;
```

funkcionālajā pierakstā izskatās šādi:

```
c = operator+ (a, b);
```

Operatoru pārslogošanai ir vairāki ierobežojumi un noteikumi:

- operatoru prioritātes un asociatīvās īpašības saglabājas un nav maināmas,
- operatoru uzvedība nav pārdefinējama iebūvētajiem tiptiem (piemēram, *int*),
- pārdefinētajam operatoram jābūt saistītam ar lietotāja izveidotu klasi (vai nu jābūt šīs klases iekšējam operatoram, vai arī klasei jābūt par kāda parametra tipu),
- operatoriem nevar būt noklusētie parametri,
- operatori, tāpat kā citas funkcijas, mantojas, izņemot operatoru `=`.

Operatoru pārslogošana ir samērā specifiska katram operatoram, tomēr ir izdalāmi 2 galvenie veidi:

operators kā **neatkarīga funkcija** – tādā gadījumā operatoram funkcionālajā pierakstā būs tikpat parametru, cik ir operandu,

operators kā **klases iekšējā funkcija** – tādā gadījumā operatoram funkcionālajā pierakstā būs par vienu parametru mazāk nekā ir operandu, jo par pirmo operandu formāli kalpos objekts, kam šis operators tiek izsaukts.

Abu pārslogošanas veidu atšķirības izpaužas tikai pašā pārdefinēšanas procesā – no izmantošanas viedokļa dažādi pārslogotie operatori neatšķiras.

Parasti operatorus pārslogo otrajā veidā – kā klases iekšēju funkciju, tas palīdz strukturēt programmu, operatoru piesaistot noteiktai klasei:

Att. 15-8. Operatora kā klases iekšējās funkcijas pārslogošana.

```
#include <iostream>
using namespace std;

class number
{
    int a;
public:
    number (int x) { a = x; };
    void print () { cout << a << endl; };
    number& operator+= (int i)
    {
        a += i;
        return *this;
    };
};

int main ()
{
    number num (99);
    num += 6;
    num.print ();
    return 0;
}
```

105

Šajā piemērā tiek pārslogots operators `+=`. Šim operatoram ir divi operandi, tomēr, tā kā operators tiek pārslogots klases iekšienē, par pirmo operandu uzstājas pats objekts (**this*), un

parametrs apzīmē labās puses operandu. Dotajā kontekstā šī operatora pārslogošanu varētu vienkāršot uz:

```
void operator+= (int i)
{
    a += i;
};
```

Atgriežamais tips *number&* un attiecīgi atgriešanas komanda *return *this* nepieciešami tikai tādēļ, lai šo operatoru varētu izmantot kaskādes veidā (nodrošinot, ka tas atgriež t.s. *lvalue* – vērtību, ko var likt piešķiršanas operatora kreisajā pusē), piemēram,

```
(num += 6) += 7; // šeit atgrieztu 112
```

Lai šo pašu operatoru pārslogotu ārēji, papildus tas jāpasludina par klases *number* draugu, jo operators tiešā veidā vērsas pie *private* lauka *a*, protams, atbilstošas izmaiņas funkcijā sakarā ar to, ka operatora pirmais operands ir funkcionālā pieraksta pirmais parametrs, nevis objekts.

Att. 15-9. Operatora pārslogošana ārēji (pārējā koda pilno tekstu sk. Att. 15-8).

```
class number
{
    ...
    friend number& operator+= (number&, int);
};

number& operator+= (number &num, int i)
{
    num.a += i;
    return num;
};

...
number num (99);
num += 6;
...

105
```

Lai klasei *number* pārslogotu operatoru *+=*, mēs varējām izvēlēties, vai to darīt iekšēji, vai ārēji, tomēr, ja operatora pirmais operands ir kāda standarta klase, tad nav citu variantu kā vien pārslogot operatoru kā neatkarīgu funkciju, citādi to nevar izdarīt bez pašas standarta klases izmaiņas.

Tipiskais piemērs šajā gadījumā ir izdrukšanas operatora pārslogošana.

Neapšaubāmi

```
number num (99);
cout << num << endl;
```

automātiski nestrādās, jo operators *<<* nav pārslogots tipam *number*.

Izdrukšanas objekts *cout* pieder klasei *ostream*, un izdrukšanas operatora *<<* pārslogošana ir diezgan specifiska nepieciešamo zināšanu dēļ par C++ standarta bibliotēkas struktūru, tomēr tehniski viss notiek atbilstoši tam, kā tiek pārslogots operators kā neatkarīga funkcija.

Att. 15-10. Izdrukšanas operatora *<<* pārslogošana (salīdzināt ar *+=* pārslogošanu Att. 15-9).

```
#include <iostream>
using namespace std;
```

```

class number
{
    int a;
public:
    number (int x) { a = x; };
    friend ostream& operator<< (ostream&, number&);
};

ostream& operator<< (ostream &os, number &num)
{
    os << num.a;
    return os;
};

int main ()
{
    number num (99);
    cout << num << endl; // pārslogotā << izmantošana
    return 0;
}

```



99

16. Teksta failu apstrāde

16.1. Fails C++

16.1.1. Fails kā informācijas glabāšanas mehānisms

Kaut arī no vispārīgā algoritmu konstrukcijas viedokļa failiem nav specifiskas lomas, tomēr sakarā ar pietiekoši lielo praktisko nozīmi un specifiku, failu apstrāde ir stingri standartizēta 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 nianšes, 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.

Valodā C++ failu apstrādei jāiekļauj standartbibliotēka `<fstream>`.

```
#include <fstream>
```

Failu C++ programmā identificē **faila objekts**, kuru reprezentē **faila mainīgais**, kurš var būt vienu no tipiem:

fstream – lasīšanai un rakstīšanai,

ofstream – tikai rakstīšanai,

ifstream – tikai lasīšanai.

```
fstream f; // faila objekta f deklarēšana lasīšanai+rakstīšanai
fstream fin; // faila objekta fin deklarēšana lasīšanai
fstream fout; // faila objekta fout deklarēšana rakstīšanai
```

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, automātiski 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.

16.1.2. Faila atvēršana un aizvēršana

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.

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 atvēršana notiek, izmantojot metodi *open*:

```
f.open ("test.txt");
```

Funkcija *open* var pieņemt divus parametrus, no kuriem otrais nav obligāts.

open

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

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

Tab. 16-1. Svarīgākās faila atvēršanas režīma vērtības.

Režīms	Apraksts
<code>ios::in</code>	Nosaka lasīšanas režīmu. Ja fails ar doto nosaukumu neeksistē, tad faila atvēršana beidzas ar neveiksmi.
<code>ios::out</code>	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.
<code>ios::binary</code> <code>ios::app</code>	Nosaka bināro režīmu lasīšanā vai rakstīšanā (sīkāk aprakstīts zemāk). (Tikai kopā ar <code>ios::out</code>) Nosaka, ka rakstīšanas režīmā atvērta faila saturs netiek izdzēsts (kā tas notiek noklusētajā variantā), bet gan rakstīšana turpinās, sākot ar faila beigām.

Tab. 16-2. Noklusētais atvēršanas režīms dažādu tipu failu objektiem.

Faila objekta tips	Noklusētais režīms
<code>fstream</code>	<code>ios::in</code> <code>ios::out</code>
<code>ifstream</code>	<code>ios::in</code>
<code>ofstream</code>	<code>ios::out</code>

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

Piemēri funkcijas *open* izmantošanai:

```
// gan lasīšanai, gan rakstīšanai:
```

```

file.open ("test.txt", ios::in | ios::out);
    // rakstīšanai, pēc atvēršanas turpinot faila beigās:
file.open ("test.txt", ios::out | ios::app);
    // lasīšanai:
file.open ("test.txt", ios::in);
    // lasīšanai un rakstīšanai binārā režīmā:
file.open ("test.txt", ios::in | ios::out | ios::binary);

```

Pēc darba ar failu, tas noteikti jāaizver, ko dara metode *close*.

```
f.close ();
```

close

```
void close ();
```

Faila objekta funkcija *close()* aizver failu.

Faila objekta deklarēšanu un atvēršanu var apvienot vienā komandā un divu rindu vietā

```

fstream f;
f.open ("test.txt");

```

var rakstīt

```
fstream f ("test.txt");
```

16.2. Teksta faila formatēta apstrāde

Viens no vienkāršākajiem failu veidiem ir teksta fails.

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

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

No programmēšanas viedokļa drīzāk 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 kā vienu no šādiem jēdzieniem:

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

16.2.1. Formatēta izvade teksta failā

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

Att. 16-1. Formatēta izvade teksta failā.

```
#include <fstream>
#include <iomanip>

...
fstream fout;
fout.open ("out.txt", ios::out);
fout << setw(5) << 1 << setw(5) << 999 << endl;
...
```

out.txt: (‘.’ tukšums, ‘¶’ jaunas rindiņas simbols, ‘\$’ faila beigas)

```
....1...999¶
$
```

16.2.2. Formatēta ievade no teksta faila

16.2.2.1. Ievades no faila standarta shēma

Formatēta ievade no teksta faila notiek pēc līdzīga principa, kādi ir pieejami ievadei no klaviatūras un kas jau ir aprakstīti iepriekšējās nodaļās. 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.

Faila beigu pazīmi uzrāda 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).

Att. 16-2. Formatēta ievade no teksta faila.

```
#include <fstream>
#include <iostream>
using namespace std;

int main ()
{
    int i;
    fstream fin ("in.txt", ios::in);
    fin >> i;
    while (!fin.eof())
    {
        cout << i << endl;
        fin >> i;
    }
}
```



```

    };
    fin.close ();
    return 0;
}
in.txt: (ievade) ('.' tukšums, '¶' jaunas rindiņas simbols, '$' faila beigas)
31·12·2006¶
$
console: (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 iepriekšējā piemērā ievades fails *in.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 iepriekšējās programmas izriet standarta shēma, kādā notiek secīga faila nolasīšana C++. 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**. Ar to C++ atšķiras no dažām citām programmēšanas valodām (piemēram, PASCAL, kur vienā iterācijā var nolasīt un apstrādāt to pašu vērtību).

Att. 16-3. Faila secīgas nolasīšanas standarta shēma C++.

```

FAILASECĪGANOLASĪŠANA (F)
    nolasa pirmo vērtību no faila
    WHILE (fails F nav beidzies)
        apstrādā nolasīto vērtību
        nolasa nākošo vērtību no faila

```

Att. 16-4. Fails un tā beigu noteikšana C++.



16.2.2.2. Kļūda ievaddatos un faila objekta bloķēšana

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. Tas no tiek tieši tāpat, kā ar standarta formatēto ievadu, kas jau tika aprakstīts iepriekšējās nodaļās.

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*.

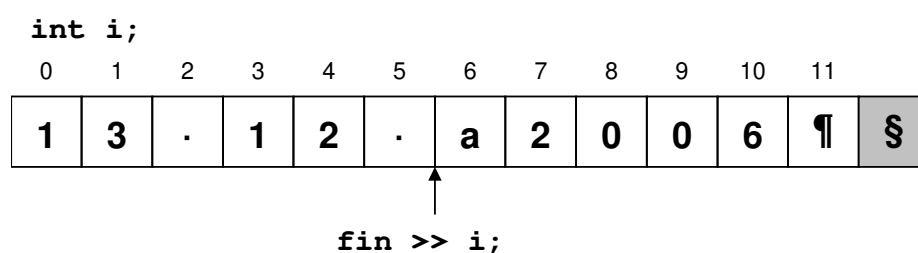
Funkciju *eof()*, *fail()*, *bad()* vietā bieži lieto (pēc nozīmes pretēju) funkciju *good()*, kas norāda, ka nav ne faila beigas, ne arī ir iestājusies kļūda.

good

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

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

Att. 16-5. Kļūdainu datu piemērs failā.



Nākošajā piemērā, salīdzinot ar Att. 16-2, cikla nosacījuma (*fin.eof()*), t.i. “kamēr nav faila beigas”, vietā izmantots (*fin.good()*), jeb, “kamēr viss kārtībā”. Tas nodrošina, ka, sastopot kļūdu, programma neieciklosies, bet beigs darbu:

Att. 16-6. Kļūdaina faila nolasīšana, apstājoties pie kļūdas (salīdzināt cikla nosacījumu ar piemēru Att. 16-2).

```
...
int i;
fstream fin ("in.txt", ios::in);
fin >> i;
while (fin.good()) // vai vienkārši: while (fin)
{
    cout << i << endl;
```

```

        fin >> i;
    };
    ...
in.txt: (ievade) ('.' tukšums, '¶' jaunas rindiņas simbols, '$' faila beigas)
31.12.a2006¶
$
konsole: (izvade)
31
12

```

Kā redzams pēc programmas komentāra, izmantotās pārbaudes vietā

```
fin.good()
```

var lietot tā vienkāršoto sinonīmu – pašu faila objektu:

```
fin
```

16.2.2.3. Faila objekta atbloķēšana pēc kļūdas

Lai turpinātu darbu ar (kļūdas vai faila beigu iestāšanās dēļ) bloķētu faila objektu, jāveic šādas darbības (sk. arī objekta *cin* atbloķēšanu pēc kļūdas, kas aprakstīts iepriekš):

objekta atbloķēšana ar metodi *clear()*,

pasākumi kļūdas situācijas novēršanai (piemēram, cita faila atvēršana, ja kļūda bijusi pie atvēršanas vai noteikta datu daudzuma izlaišana (metode *ignore*), ja kļūda bijusi ievaddatos).

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*.

Att. 16-7. Kļūdaina veselu skaitļa faila nolasīšana ar kļūdaino simbolu izlaišanu (*ignore*).

```

...
fstream fin ("in.txt", ios::in);
int i;
fin >> i;
while (fin)
{
    cout << i << endl;
    fin >> i; // mēģina lasīt kārtējo skaitli
    // cikls, kamēr kļūda netiek likvidēta,
    // bet tai pat laikā nav sasniegtas faila beigas
    while (!fin.good() && !fin.eof())
    {
        fin.clear ();           // atbloķē faila objektu
        fin.ignore (1, '\n');   // izlaiž vienu simbolu
        fin >> i;               // mēģina lasīt nākošo skaitli
    }
}

```

```

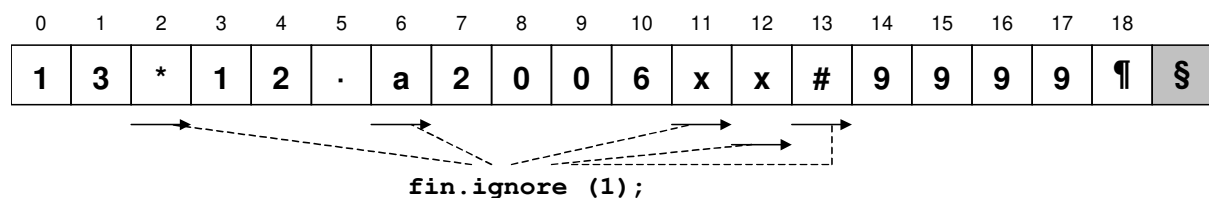
};
...
in.txt: (ievade) ('.' tukšums, '¶' jaunas rindiņas simbols, '$' faila beigas)
31*12.a2006xx#9999¶
$
konsole: (izvade)
31
12
2006
9999

```

Piezīme. Šī programma nemāk apstrādāt kļūdu, ja tā ir pašā faila sākumā. Šim nolūkam tādu pašu iekšējo ciklu `while (!fin.good() && !fin.eof())` vajadzētu ievietot arī pēc pirmreizējās nolasīšanas operācijas. Ja kļūda ir jau pašā faila sākumā (piemēram, tas sākas ar burtu), tad šī programma uzreiz apstājas.

Iepriekšējās programmas darbu kļūdaino simbolu izlaišanā parādā nākošais attēls.

Att. 16-8. Kļūdainu simbolu izlaišana, nolasot failu (programma Att. 16-7).



16.3. Teksta failu apstrāde bez formatēšanas

16.3.1. Teksta failu apstrāde pa vienam simbolam

Formatēšana nozīmē datu pārveidošanu pēc nolasīšanas vai pirms izdrukāšanas (failā).

Vienkāršākā teksta faila apstrāde bez formatēšanas ir **pa vienam simbolam**. Arī nolasīšana pa vienam simbolam notiek pēc faila nolasīšanas standarta shēmas, kas redzama Att. 1-8.

Teksta faila apstrāde pa vienam simbolam notiek, izmantojot funkcijas `get()` un `put()`, kas attiecīgi nolasā vai ieraksta vienu simbolu failā.

Att. 16-9. Viena faila (*fin*) pārrakstīšana otrā (*fout*) pa vienam simbolam.

```

fstream fin ("in.txt", ios::in);
fstream fout ("out.txt", ios::out);
char c;
fin.get (c);
while (fin)
{
    fout.put (c);
    fin.get (c);
};
fin.close ();
fout.close ();
in.txt: (ievade) ('.' tukšums, '¶' jaunas rindiņas simbols, '$' faila beigas)
This¶
is.an¶

```

```

||| example.$
out.txt: (izvade) ('.' tukšums, '\n' jaunas rindiņas simbols, '$' faila beigas)
||| This\n
||| is an\n
||| example.$

```

get

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

Faila objekta metode *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.

Metodes *put* un *get* ir vienas no retajām faila objektā, kuras spēj strādāt divējādi:

teksta režīmā,

binārā režīmā.

Apstrādājot failu binārā režīmā, informācija tiek apstrādāta precīzi pa baitam, bet teksta režīmā – pa simbolam. Teksta failā šie jēdzieni parasti sakrīt, tomēr ir daži izņēmumi. Labākais piemērs tam ir fakts, ka *Windows* sistēmā jaunas rindiņas simbolu `\n` kodē nevis ar vienu baitu (jaunas rindiņas 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.

Att. 16-10. Iepriekšējās programmas ievades faila *in.txt* interpretācija teksta režīmā (Att. 16-9).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
T	h	i	s	\n	i	s	.	a	n	\n	e	x	a	m	p	l	e	.\$

Att. 16-11. Iepriekšējās programmas ievades faila *in.txt* interpretācija binārā režīmā (*Windows* sistēmā kodētam failam) (Att. 16-9).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T	h	i	s	13	10	i	s	.	a	n	13	10	e	x	a	m	p	l	e	.\$

Teksta režīms dod iespēju rakstīt universālas vairāku platformu programmas, platformu atšķirību atrisināšanu uzticot pašam C++.

Tādu duālas darbības funkciju, kā *get* un *put*, darbības veidu nosaka režīms, kādā atvērts fails.

Pēc noklusēšanas faila atvēršanas režīms ir teksta. Ja, atverot failu, ir uzrādīta bināra režīma vērtība *ios::binary*, tad apstrādes režīms ir binārs:

```

| fstream fin ("in.txt", ios::in | ios::binary);
| fstream fout ("out.txt", ios::out | ios::binary);

```

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.

Faila objektā ir pieejamas arī tādas metodes, kuras spēj strādāt tikai binārā režīmā, un faila atvēršanas režīma uzstādīšana (teksta vai bināra) to darbību no šāda viedokļa neietekmē. Šādas metodes tiks apskatītas nākošajās nodaļās, runājot par bināru failu apstrādi.

16.3.2. 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.

Faila nolasīšana pa vienai rindai notiek, izmantojot funkciju *getline()*, kam ir 2 modifikācijas – zema un augsta līmeņa simbolu virknēm. Abas modifikācijas jau apskatītas pie standarta ievades iepriekšējās nodaļās.

getline (ar zema līmeņa simbolu 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.

getline (ar augsta līmeņa simbolu 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.

Att. 16-12. Teksta faila nolasīšana pa rindām un izvade uz ekrāna (par buferi izmantota augsta līmeņa simbolu virkne).

```
#include <fstream>
#include <iostream>
...
string s;
fstream fin ("in.txt", ios::in);
getline (fin, s);
while (fin)
{
    cout << s << endl;
    getline (fin, s);
};
fin.close ();
...
```

in.txt: (ievade) (' ' tukšums, '¶' jaunas rindiņas simbols, '\$' faila beigas)

```
||| This
||| is an
||| example.$
konsole: (izvade)
||| This
||| is an
||| example.
```

17. Bināru failu apstrāde

17.1. Vienkāršākās faila binārās apstrādes operācijas

17.1.1. Binārās failu apstrādes atšķirības

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

Faila bināra 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 bināra 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.

17.1.2. 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 bināra faila gadījumā parasti daudz precīzāk jāapzinās apstrādājamā faila struktūra, jo datu interpretācija būs jādefinē pašam programmētājam, nevis jāatstāj standarta faila apstrādes metožu ziņā.

17.1.3. Izvade failā 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),

norādes pārveidošana uz tipu `char*` (ja tai jau nav tāds tips), izmantojot klasisko tipu pārveidošanas operatoru (`(char*)pointer`) vai (`reinterpret_cast<char*>(pointer)`).

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()`.

Att. 17-1. Bināra faila izvade.

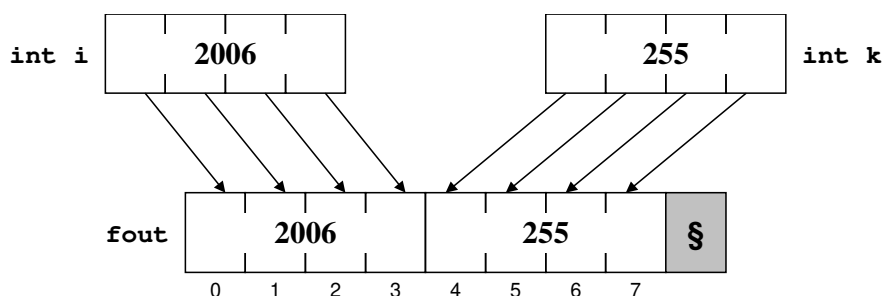
```
fstream fout ("integers.bin", ios::out);
int i=2006, k=255;
    // ierakstu vienu skaitli:
fout.write ((char*)&i, sizeof(int));
    // ierakstu otru skaitli:
fout.write (reinterpret_cast<char*>(&k), sizeof(int));
fout.close ();
```

integers.bin: (izvade) (hex redaktorā)

||| 00000000: D6 07 00 00 FF 00 00 00|

| 00000000

Att. 17-2. Iepriekšējās programmas (Att. 16-7) darbība pēc būtības.



Ja ar metodi `write` failā raksta no mainīgā ar tipu `char`, tad tipa pārveidošana nav nepieciešama, jo `&c` jau ir ar tipu `char*`:

```
char c;
...
fout.write (&c, 1); // identisks fout.put(c) binārā režīmā
```

Ja ar metodi *write* failā raksta no *char* masīva (piemēram, simbolu virknes), tad arī adreses ņemšana (&) nav nepieciešama, jo *s* jau ir ar tipu *char**:

```
char s[20];
...
fout.write (s, 20);
```

17.1.4. Nolasīšana no faila 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()*.

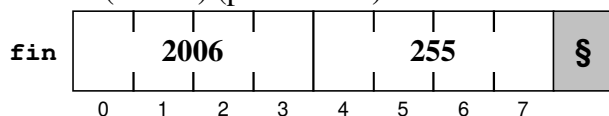
Funkcija *read()*, kam caur parametru #2 tiek padots 1, lielā mērā atbilst binārā režīmā izsauktai funkcijai *get()*.

Nākošā programma demonstrē bināra *int* vērtību faila (piemēram, tāda, kas tika izveidots ar programmu Att. 16-7) nolasīšana un vērtību izdrukāšana uz ekrāna.

Att. 17-3. Bināra nolasīšana no faila.

```
int i;
fstream fin ("integers.bin", ios::in);
fin.read ((char*)&i, sizeof(int));
while (fin)
{
    cout << i << endl;
    fin.read ((char*)&i, sizeof(int));
};
fin.close ();
```

integers.bin: (ievade) (pēc būtības)



konsole: (izvade)

```
2006
255
```

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.

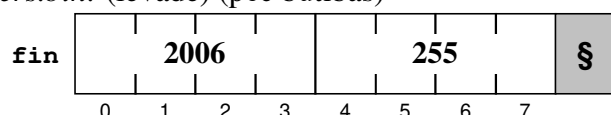
Nākošā programma pārraksta vienu failu otrā binārā režīmā. Līdzīga programma tika demonstrēta nodaļā par teksta failiem, taču šeit pārrakstīšana notiek nevis pa vienam baitam,

bet pa vairākiem, tā nodrošinot ātrāku apstrādi. Informācijas noglabāšanai starp nolasīšanu un ierakstīšanu tiek lietots buferis (šeit, garumā 3). Metode *gcount* tiek izmantota, lai zinātu, cik kārtējā reizē nolasīti baiti, lai zinātu, cik savukārt no bufera jāieraksta otrā failā. Problēmas ar to, ka otrā failā nesanāks ierakstīt visu buferi var rasties tikai pēdējā nolasīšanas porcijā no faila. Šī programma der viena faila pārrakstīšanai otrā neatkarīgi no tā, cik liels bufera izmērs (*BUFFER_SIZE*) ir uzlikts. Nolasīto baitu skaits katrā solī tiek parādīts izdrukā uz ekrāna.

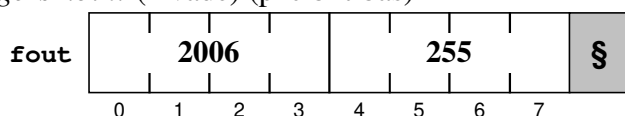
Att. 17-4. Viena faila (*fin*) pārrakstīšana otrā (*fout*) binārā režīmā, izmantojot buferi.

```
const unsigned int BUFFER_SIZE = 3;
char buffer[BUFFER_SIZE];
fstream fin ("integers.bin", ios::in);
fstream fout ("integers2.bin", ios::out);
while (fin)
{
    fin.read (buffer, BUFFER_SIZE);
    cout << fin.gcount() << endl;
    fout.write (buffer, fin.gcount());
};
fin.close ();
fout.close ();
```

integers.bin: (ievade) (pēc būtības)



integers2.bin: (izvade) (pēc būtības)



konsole: (izvade)

```
3
3
2
```

17.2. Tiešās pieejas metodes bināro failu apstrādē

Noklusētā metode faila apstrādei ir sekvenciālā pēc kārtas, taču pateicoties sekundārās atmiņas tehniskajām iespējām, ir iespējā apstrādāt faila datus, piekļūstot tiem tiešā veidā – kā caur indeksu masīva elementam.

Strādājot ar failu tiešās pieejas režīmā, jāatceras, ka

faila informācijas pamatvienība ir **baits**,

baitu numerācija failā sākas ar 0.

Strādājot ar failu tiešās pieejas režīmā, īpaši svarīgs kļūst jēdziens **faila norāde** – tā ir aktuālā vieta failā, sākot no kuras uz priekšu tiks veikta kārtēja nolasīšana, ierakstīšana vai cita darbība. Tiešās pieejas darbības ir tās, kuras darbojas ar faila norādi.

Šajā sadaļā tiks aprakstītas šādas failu apstrādes metodes:

seek, *seekp* – faila norādes uzstādīšana noteiktā pozīcijā,

tellg, *tellp* – atrašanās vietas noskaidrošana failā,

peek – “paskatīšanās vienu simbolu uz priekšu”.

17.2.1. Faila norādes uzstādīšana (pozīcijas meklēšana failā)

Faila norādes uzstādīšana noteiktā pozīcijā veic funkcijas *seek/seekp*.

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, un tās dara vienu un to pašu.

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).

Att. 17-5. Metožu *seekg/seekp* demonstrācija. Faila beidzamo simbolu ieraksta pirmajā pozīcijā.

```
fstream f ("seek.txt", ios::in | ios::out | ios::binary);  
char c;  
f.seekg (-1, ios::end); // aiziet uz pozīciju pirms pēdējā  
f.get (c); // nolasa pēdējo  
f.seekp (0, ios::beg); // aiziet uz faila sākumu  
f.put (c); // ieraksta sākumā vienu simbolu  
f.close ();  
seek.txt: (ievade) ('$' – faila beigas)  
||| ABCDEFGHIJS  
seek.txt: (izvade) ('$' – faila beigas)  
||| JBCDEFGHIJS
```

17.2.2. Faila norādes atrašanās vietas noskaidrošana

Faila norādes atrašanās vietas noskaidro ar funkcijām *tellg/tellp*.

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.

Att. 17-6. Metodes *tellg* demonstrācija (ietver metodi, kā noteikt faila garumu).

```
fstream f ("tell.txt", ios::in);
```

```

f.seekg (0, ios::end); // aiziet uz beigām
cout << f.tellg() << endl; // izdrukā pozīciju (=faila garums)
f.seekg (-3, ios::cur); // paiet trīs soļus atpakaļ
cout << f.tellg() << endl; // izdrukā pozīciju
f.close ();

```

tell.txt: (ievade) ('\$' – faila beigas)

```

ABCDEFGHIJ$

```

konsole: (izvade)

```

10

```

```

7

```

17.2.3. Paskatīšanās vienu simbolu uz priekšu

Paskatīšanos vienu soli uz priekšu, nepārbīdot faila norādi nodrošina funkcija *peek*.

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.

Att. 17-7. Metodes *peek* demonstrācija.

```

char c;
fstream f ("peek.txt", ios::in | ios::binary);
f.seekg (5); // aiziet uz pozīciju 5 (pirms 'F')
c = f.peek(); // nolasa, bet nepārbīda norādi
cout << c << endl;
c = f.get(); // nolasa un pārbīda norādi
cout << c << endl;
c = f.get(); // vēlreiz nolasa un pārbīda norādi
cout << c << endl;
f.close ();

```

peek.txt: (ievade) ('\$' – faila beigas)

```

ABCDEFGHIJ$

```

konsole: (izvade)

```

F

```

```

F

```

```

G

```

17.3. 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.

Ieraksts (*record*) (jeb **komponente**) ir (no datu apstrādes viedokļa) vienots informācijas kopums failā.

Nākošajā piemērā failā tiks glabāti dati par personu. 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.

Šajā gadījumā faila ieraksts jeb komponente ir informācija par vienu personu.

Pirmajā failā (*pers1.bin*) viena ieraksta struktūra ir šāda:

name – personas vārds – 20B,

age – personas vecums – 4 B.

Otrajā failā (*pers2.bin*) 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.

Att. 17-8. Personas informācijas izvade binārā failā.

```
#include <fstream>
#include <iostream>
using namespace std;
const int buffer_size = 20;

class person
{
    char name[buffer_size];
    int age;
public:
    person (const char *n, int a)
    {
        strcpy (name, n);
        age = a;
    };
    void writel (ostream &fout)
    {
        fout.write (name, buffer_size); // pilna garuma masīvs
        fout.write ((char*)&age, sizeof(age)); // visa int
            vērtība
    };
    void write2 (ostream &fout)
    {
        int slen = strlen(name);
```



```

};
bool read2 (istream &fin)
{
    int slen = 0; // svarīgi aizpildīt visus baitus ar 0
    age = 0; // svarīgi aizpildīt visus baitus ar 0
    fin.read ((char*)&slen, 1); // nolasa vārda gar. no 1 B
    fin.read (name, slen); // nolasa vārdu uzrādītajā garumā
    name[slen] = '\0'; // pieliek beigās beigu simbolu
    fin.read ((char*)&age, 1); // nolasa vecumu no 1 baita
    return fin.good ();
};
void print ()
{
    cout << name << " " << age << endl;
}
};

int main ()
{
    person p;
    ifstream fin1 ("pers1.bin");
    while (p.read1 (fin1)) p.print ();
    fin1.close ();
    ifstream fin2 ("pers2.bin");
    while (p.read2 (fin2)) p.print ();
    fin2.close ();
    system ("pause");
    return 0;
}

```

pers1.bin: (ievade) (pēc būtības)

0	1	2	3	4	19	20	21	22	23	24	25	26	27	28	29	44	45	46	47
L	i	z	\0				19			P	e	t	e	r	\0			20	\$

pers2.bin: (ievade) (pēc būtības)

0	1	2	3	4	5	6	7	8	9	10	11	
3	L	i	z	19	5	P	e	t	e	r	20	§

konsole: (izvade)

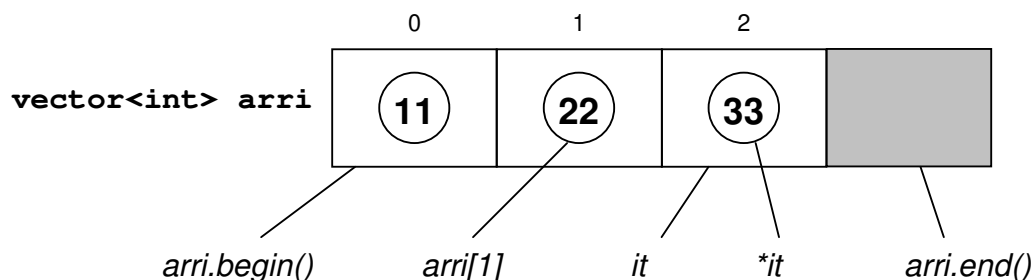
```

Liz 19
Peter 20
Liz 19
Peter 20

```


push_back() – funkcija, kas pievieno vektoram elementu tā beigās,
vector<T>::iterator – vektora iteratora tips,
it++ – iteratora pāreja uz nākošo elementu,
[] – piekļuve vektora elementa vērtībai (nevis elementam!) pēc indeksa,
**it* – piekļuve vektora elementa vērtībai, izmantojot iteratoru.

Att. 19-8. Programmā Att. 19-7 izveidotais vektors.



19.2.3. STL klase 'list'

STL klase *list* (saraksts) ir līdzīga klasei *vector* ar to atšķirību, ka tajā nav pieejama piekļuve elementu vērtībām, izmantojot indeksu. Tomēr, ja tas nav nepieciešams, tad labāk vektora vietā lietot tieši sarakstu, jo šī konstrukcija ir mazāk prasīga pēc resursiem struktūras izmēra maiņas gadījumā (piemēram, pievienojot jaunu elementu). Tai pat laikā klasē *list* ir pieejamas daudzas ļoti vērtīgas metodes, piemēram, elementa iespraušana.

Att. 19-9. STL klases 'list' izmantošana.

```
#include <list>
#include <iostream>
using namespace std;
typedef list<int> intlist;
typedef intlist::iterator intit;

int main ()
{
    intlist arri;
    // aizpildīšana:
    arri.push_back (11);
    arri.push_back (22);
    arri.push_back (33);
    // pārstaigāšana:
    intit iti = arri.begin ();
    while (iti != arri.end ())
    {
        cout << *iti << endl;
        iti++;
    };
    return 0;
}
```

Piemērā izmantoto klases *list* īpašību apraksts:

- begin()* – iterators (norāde) uz pirmo elementu,
- end()* – iterators (norāde) uz virtuālu elementu aiz pēdējā,
- push_back()* – funkcija, kas pievieno sarakstam elementu tā beigās,
- list<T>::iterator* – saraksta iteratora tips,
- iti++* – iteratora pāreja uz nākošo elementu,
- *iti* – piekļuve vektora elementa vērtībai, izmantojot iteratoru.

19.2.4. STL algoritms ‘sort’

Viens no redzamākajiem STL algoritmiem ir kārtšanas algoritms *sort()*. Ar to var kārtot ne tikai *vector*, bet arī *deque* elementus (kontaineram *list* ir realizēta iekšēja *sort* metode).

Kārtšanas algoritms izmantojams divos veidos:

- **Standarta variants**, izmantojot standarta salīdzināšanas (“mazāk”) kritēriju, kāds pieejams struktūrā glabājamiem objektiem.
- **Specializētais variants**, izmantojot speciāli izveidotu salīdzināšanas kritēriju.

Kārtšanas algoritma izmantošana standarta variantā. Tiek izsaukta funkcija *sort()* ar diviem parametriem – (1)sākuma iterators un (2)beigu iterators (jāatceras, ka beigu iteratoram jānorāda nevis uz beidzamo kārtojamo elementu, bet nākošo aiz tā).

Kārtšanas algoritma izmantošana specializētajā variantā. Tiek izsaukta funkcija *sort()* ar trīs parametriem – (1)sākuma iterators, (2)beigu iterators un (3) salīdzināšanas kritērijs. Salīdzināšanas kritērijs ir funkcija, kas par diviem objektiem, kādus var glabāt dotā struktūrā, pasaka – vai pirmais ir mazāks par otro.

Funkcija, kas definē salīdzināšanas kritēriju, ir jānoformē noteiktā formātā:

- kā speciāli izveidotas klases operatora *operator()* pārdefinēšana (pārslogošana) vai
- speciāla salīdzināšanas funkcija.

Kad šāda klase (vai funkcija) izveidota, tad, izsaucot kārtšanas funkciju *sort()*, trešajā parametrā tiek padots klases nosaukums ar tukšām iekavām (vai funkcijas nosaukums).

Att. 19-10. STL algoritma ‘sort’ izmantošana, izmantojot speciālo klasi.

```
#include <algorithm>
#include <vector>
#include <iostream>
using namespace std;

typedef vector<int> intarray;

class lastdigit_compare
{
public:
    bool operator() (const int &v1, const int &v2) const
    {
```

```

        return v1%10 < v2%10;
    }
};

int main ()
{
    intarray arri(3);
    arri[0] = 52;
    arri[1] = 28;
    arri[2] = 47;
    // sakārto pēc skaitļa vērtības:
    sort (arri.begin(), arri.end());
    for (int i=0; i<arri.size(); i++) cout << arri[i] << endl;
    // sakārto pēc pēdējā cipara vērtības:
    sort (arri.begin(), arri.end(), lastdigit_compare());
    cout << endl;
    for (int i=0; i<arri.size(); i++) cout << arri[i] << endl;
    return 0;
}

```

28
47
52

52
47
28

Att. 19-11. STL algoritma ‘sort’ izmantošana, izmantojot speciālo salīdzināšanas funkciju (tikai fragmenti, kas atšķirīgi no iepriekšējā piemēra).

```

// ***** sk. iepriekšējo piemēru *****
bool lastdigit_compare (const int &v1, const int &v2)
{
    return v1%10 < v2%10;
};
// *****
    sort (arri.begin(), arri.end(), lastdigit_compare);
// *****

```

19.2.5. STL klase ‘map’

STL klase *map* (karte) nodrošina vērtību pāru glabāšanu, kā arī piekļūšanu elementiem pēc pirmās vērtības. No izmantošanas viedokļa karte līdzinās vārdnīcai. Ja elementa pirmās vērtības tips ir *int*, tad *map* praktiski nodrošina vektora funkcionalitāti. Pievienojot jaunu elementu kartei, tiek automātiski nodrošināta *map* elementu sakārtošana pēc elementu pirmajām vērtībām. Tā kā klase *map* nodrošina vērtību pāru glabāšanu, tā izmanto STL klasi *pair* viena elementa vērtību glabāšanai.

Att. 19-12. STL klases ‘map’ izmantošana.

```

#include <map>
#include <iostream>

```

```

using namespace std;

typedef map<string,string> dictionary;
typedef dictionary::iterator dictit;

int main ()
{
    dictionary d;
    // aizpildīšana ar masīva sintaksi
    d["day"] = "giorno";
    d["street"] = "strada";
    d["italian"] = "italiano";
    d["red"] = "rosso";
    string s;
    cin >> s; // meklējamais vārds
    dictit it = d.find (s);
    if (it != d.end())
        cout << it->first << " " << it->second << endl;
    else cout << "NOT FOUND";
    return 0;
}

```

red
red rosso

blue
NOT FOUND

Piemērā izmantoto klases *map* īpašību apraksts:

begin() – iterators (norāde) uz pirmo elementu,

end() – iterators (norāde) uz virtuālu elementu aiz pēdējā,

[] – piekļuve elementa vērtībai (strukturai *pair*) pēc pirmās vērtības; vai elementa pievienošana strukturai, automātiski nodrošinot tās sakārtošanu,

map<T1,T2>::iterator – kartes iteratora tips,

it++ – iteratora pāreja uz nākošo elementu,

it->first, **it.first* – piekļuve elementa pirmajai vērtībai, izmantojot iteratoru,

it->second, **it.second* – piekļuve elementa otrajai vērtībai, izmantojot iteratoru,

find() – funkcija, kas atgriež iteratoru uz elementu ar doto pirmo vērtību (ja tāds elements neeksistē, tad atgriež iteratoru *end()*).

19.2.6. STL klase ‘pair’

STL klase *pair* nodrošina divu dažādu vērtību glabāšanu vienā strukturā, kas atgādina klasi *two*, kas definēta piemērā Att. 19-5.

Klase *pair* ietver divus laukus: *first* un *second*, un to izmanto arī citi konteineri, piemēram, *map*, kas tiks apskatīts šajā materiālā.

Att. 19-13. STL klases ‘pair’ izmantošana.

```
#include <iostream>
```

```
using namespace std;

typedef pair<string,int> person;

int main ()
{
    person p ("Liz", 19);
    cout << p.first << " " << p.second << endl;
    return 0;
}

||| Liz 19
```

Vairāk nekā divu dažādu vērtību glabāšanai tiek izmantots kontainers *tuple*.

20. Izņēmumu apstrāde, nosaukumu telpas, norādes uz funkcijām, komandrindas argumenti

20.1. 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 Att. 18-6:

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.

Att. 20-1. Izņēmumu apstrādes pieraksta shēma C++.

```
try
{
    ...
    if (kļūdas veids nr. 1) throw some_exception1;
    ...
    if (kļūdas veids nr. 1) throw some_exception2;
    ...
}
catch (T1 exc)
    { /* izņēmumu apstrāde ar tipu T1 */ }
catch (T2 exc)
    { /* izņēmumu apstrāde ar tipu T2 */ }
catch (...)
    { /* citu izņēmumu apstrāde */ }
```

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 vienā, bet apstrādāta citā funkciju izsaukumu hierarhijas līmenī. 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,

```
try
{
    // izņēmumu situāciju noskaidrošanas zona
```

```
}
```

Kļūdas situācijas iestāšanos noskaidro pats programmētājs – izņēmumu apstrādes infrastruktūra tikai palīdz kļūdas paziņojumu nosūtīt uz apstrādes vietu.

Izņēmuma izmešana ar ‘throw’.

Izmantojot operatoru *throw* 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.

```
throw "ERROR: Division by zero"
```

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.

Att. 20-2. Izņēmumu apstrādes sadaļa (*catch_section*).

```
catch_section:
    catch_block_listopt default_catch_block

catch_block_list:
    catch_block
    catch_block_list catch_block

catch_block:
    catch ( parameter_definition ) catch_body

default_catch_block:
    catch ( ... ) catch_body
```

- 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.

Nākošais piemērs demonstrē izņēmumu apstrādes mehānismu funkcijai *process*, kas rēķina izteiksmes $x + y/z$ vērtību. Rēķinot funkcijas vērtību, nedrīkst tikt pārkāpti šādi nosacījumi – z nav 0, un saskaitāmie ir robežās 0..100.

Att. 20-3. Izņēmumu apstrāde.

```
#include <iostream>
using namespace std;

double x_over_y (double x, double y)
{
    if (y == 0) throw "ERROR: Division by zero";
    return x / y;
};
```

```

double x_plus_y_over_z (double x, double y, double z)
{
    double div = x_over_y (y, z);
    if (x > 100 || div > 100) throw 100;
    else if (x < 0 || div < 0) throw 0.0;
    return x + div;
};

void process (double x, double y, double z)
{
    try
    {
        cout << x_plus_y_over_z (x, y, z) << endl;
    }
    catch (const char *error_text)
        { cout << error_text << endl; }
    catch (int error_num)
        { cout << "ERROR: Addend too big" << endl; }
    catch (...)
        { cout << "ERROR: Unknown error" << endl; };
};

int main ()
{
    /* Meklētās kļūdas:
       - nedrīkst dalīt ar 0
       - saskaitīšanas operācija apstrādā skaitļus 0..100 */
    process (2, 4, 0);      // 2 + 4 / 0
    process (999, 4, 0.5); // 999 + 4 / 0.5
    process (-2, 4, 0.5);  // -2 + 4 / 0.5
    process (2, 4, 0.5);   // 2 + 4 / 0.5
    return 0;
}

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

```

Piezīme. Izņēmumu apstrādes mehānisms nav domāts kļūdu noskaidrošanai programmā, bet gan noskaidroto kļūdu paziņojumu nosūtīšanai uz speciālu apstrādes vietu.

20.2. 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*.

18.2. Saistītais saraksts – dinamiskas datu struktūras

18.2.1. 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 mīnusi 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,

saites uz citiem elementiem.

18.2.2. Saistītais saraksts

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*.

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

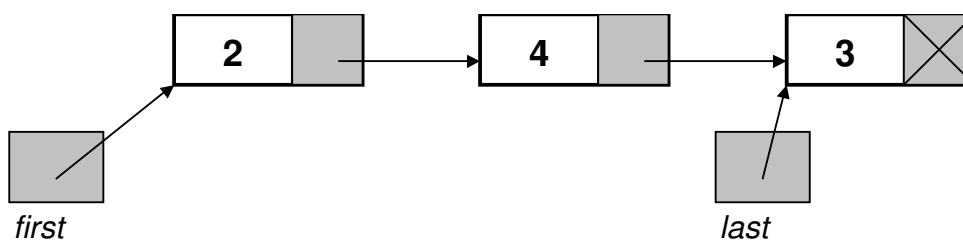
norāde uz pirmo elementu,

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.

Att. 18-5. Saistītais saraksts no 3 elementiem (atbilst programmai Att. 18-6).



Saistīto sarakstu (*linked list*), ja nav norādīts savādāk, veido tā, ka katru jauno elementu kabina klāt beidzamajam, tādējādi elementi tajā glabājas ievadīšanas secībā.

Att. 18-6. Saistītā saraksta (*linked list*) izveidošana un izdrukāšana.

```
...
struct elem
{
    int num;
    elem *next;
};

...
// first-norāde uz pirmo elementu
// last-norāde uz pēdējo elementu
// p-palīgnorāde
elem *first=NULL, *last=NULL, *p;
int i;
cin >> i;
// ievadišana, kamēr nav sastapta 0
while (i != 0)
{
    p = new elem; // izveido objektu
    p->num = i;    // aizpilda ar nolasīto vērtību
    p->next = NULL; // norāde uz nākošo - tukša!
    if (first == NULL)
    {
        // ja saraksts tukšs
        // gan pirmais, gan pēdējais norāda uz jauno elementu
        first = last = p;
    }
    else
    {
        // ja saraksts nav tukšs
        // pieliek galā beidzamajam
        last->next = p;
        // un jaunais kļūst par beidzamo
        last = last->next;
    }
    cin >> i;
};

// saraksta izdruka
for (p = first; p!=NULL; p=p->next)
{
    cout << p->num << endl;
};

// saraksta iznīcināšana
p = first;
while (p!=NULL)
{
    first = first->next;
    delete p;
    p = first;
};
```

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 (*traversal*), veicot noteiktu darbību ar katru elementu.

Att. 18-7. Datu struktūru (t.sk. saistīta saraksta) secīgas pārstaigāšanas shēma.

```
STRUKTŪRAS SECĪGĀ PĀRSTAIGĀŠANA (S)  
  aiziet uz struktūras sākumu  
  WHILE (struktūra S nav beigusies)  
    apstrādā kārtējo elementu  
    pāriet uz nākošo elementu
```

Šī shēma ir analogiska tai, kas nosaka, kādā veidā tiek organizēta datu nolasīšana no faila.

Strādājot ar dinamiskām datu struktūrām, ir ļoti svarīgi kontrolēt nullēs norādes (*NULL* jeb 0):

ja elements ir pēdējais virknē, tad nedrīkst aizmirst uzstādīt *NULL* tā norādei uz nākošo elementu,

ja kāda norāde ir *NULL*, nedrīkst mēģināt pēc šīs norādes nokļūt pie kāda objekta un to apstrādāt (tipiska kļūda darbā ar dinamiskajām datu struktūrām).

Parasti datu struktūru apstrādes darbības tiek ievietotas funkcijās. Tādā gadījumā funkcijai padod norādi uz pirmo un dažreiz arī pēdējo saraksta elementu (piemēram, *elem**):

```
void print_list (elem *first)  
{  
    ...  
};
```

Ja ir sagaidāms, ka norāde funkcijā varētu izmainīties, tad parametrs jādefinē kā reference. Piemēram, elementa pievienošanas funkcijā norāde uz pirmo elementu mainās, ja pievienošana notiek tukšam sarakstam, bet norāde uz beidzamo elementu mainās katru reizi, tāpēc abiem attiecīgajiem parametriem jābūt referencēm (*elem*&*):

```
void add_element (elem *&first, elem *&last, int i)  
{  
    ...  
};
```

18.2.3. Elementa iespraušana saistītā sarakstā

Noklusētais variants saistītā saraksta papildināšanai ir elementu pievienošana saraksta beigās, tomēr reizēm (piemēram, veidojot sakārtotu sarakstu) ir nepieciešams ievietot elementu saraksta vidū.

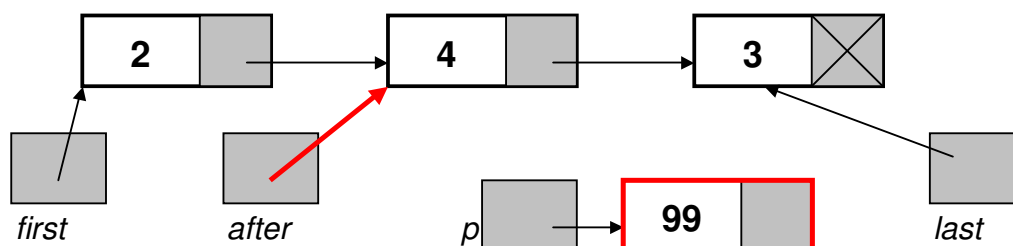
Jā ir tāda nepieciešamība, tad vispirms ir jāiegūst norāde uz elementu, aiz kura tiks iesprausts jaunais elements.

Nākošais piemērs sadalīts 3 daļās un par pamatu izmanto sarakstu, kāds tika iegūts iepriekšējā programmā (Att. 18-6).

- Sekojošais programmas fragments parāda, ka, lai aiz elementa 4 iespraustu 99, jāiegūst norāde *after* un jāizveido jaunais elements (piemērā netiek parādīts, kā norāde uz šo elementu (*after*) tiek iegūta).

```
elem *after;
...
elem *p = new elem;;
p->num = i;    // aizpilda ar nolasīto vērtību
```

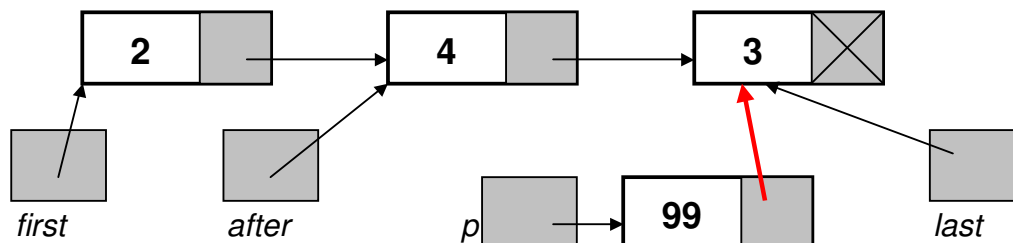
Att. 18-8. Elementa iespraušana sarakstā – norāde uz elementu, aiz kura iespraust – *after* un ievietojamais elements *p*.



- Tad jaunajam elementam piekādē galā saraksta beigas.

```
p->next = after->next;
```

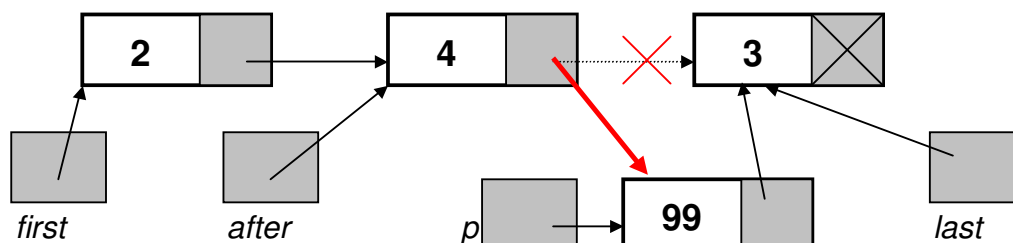
Att. 18-9. Elementa iespraušana sarakstā – saraksta beigu piekabināšana jaunajam elementam.



- Visbeidzot no iepriekšējā elementa (*after*) tiek novilkta saite uz jauno elementu.

```
after->next = p;
```

Att. 18-10. Elementa iespraušana sarakstā – iepriekšējā elementa saites novilkšana uz jauno elementu.



13. Sistēmprogrammēšanas elementi

Nodaļas saturs:

- 13.1. Pavedienprocesi
 - 13.1.1 Pavedienprocesi Windows standartā
 - 13.1.2. Pavedienprocesi POSIX standartā
- 13.2 Logu programmēšana Windows

13.1. Pavedienprocesi

Ar sistēmprogrammēšanu šeit tiek saprastas tādas programmēšanas aktivitātes, kurās izmantojamās konstrukcijas nav C++ standarts, bet ir platformas (respektīvi, sistēmas) atkarīgas. Viena no šādām tēmām, ar ko saistās sistēmprogrammēšana, ir **paralēlie procesi**. Šajā nodaļā tiks aplūkoti t.s. viegla svara (*lightweighed*) procesi jeb **pavedienperocesi** (*threads*), kas raksturojas ar to, ka tiem netiek izdalīti neatkarīgi resursi un kam neatkarīga no citiem (paralēla) ir tikai to izpilde. Vienas programmas ietvaros ar pavedienprocesi parasti pietiek, lai realizētu (šķietami) paralēlu darbību virkņu izpildi.

Kad kāda programma tiek palaista, tad priekš tās automātiski tiek ievēidots viens process (galvenais process), un programmas pašas ziņā ir – vai palaist vēl kādus papildus procesus paralēli galvenajam.

Kaut arī sintakse starp dažādām platformām atšķiras, tomēr galvenie principi ir līdzīgi. Šeit aplūkotās tikai pašas vienkāršākās konstrukcijas, un tās ir:

- procesa izveidošana (palaišana);
- procesu sinhronizēšana un pabeigšana.

Tālāk parādītie piemēri Windows un Linux vidēs palaiž divus paralēlos procesus (sauksim tos par A un B), un katrs no šiem procesiem izdrukā uz ekrāna vērtības no 0 līdz 3 (pievienojot priekšā savu identifikatoru, attiecīgi A vai B), tad katrs procesa beigās izdrukā, ka ir beidzies, bet kad beigušies abi procesi, tiek izdrukāts arī šāds paziņojums (sk. pirmkoda piemērus 13.1 un 13.2).

13.1.1. Pavedienprocesi Windows standartā

Lai Windows platformā strādātu ar procesiem, jāielādē bibliotēka `<windows.h>`.

Windows platformā (sk. pirmkoda piemēru 13.1) viens no būtiskiem jēdzieniem programmēšanai ir t.s. *handle* ('rokturis'). Tā ir sava veida **norāde** (*pointer*) uz Windows specifiskiem objektiem, un tā to sauksim arī turpmāk materiālā. Lai Windows sistēmā strādātu ar pavedienprocesi, vispirms ir jādefinē norādes uz tiem (sk. 10 rindu pirmkodā).

Rindās 13 un 14 tiek attiecīgi palaisti 2 paralēlie procesi. To veic ar funkcijas *CreateThread* palīdzību. Svarīgākie principi funkcijas *CreateThread* izsaukumā:

- Paralēlā procesa veidošana pamatā nozīmē funkcijas izsaukumu, Windows vidē ir noteikts, ka tā vienmēr ir funkcija ar atgriežamo tipu (*DWORD WINAPI*) un tieši vienu parametru ar tipu (*void **).
- Mūsu piemērā šī funkcija ir process, kas definēta rindās 25-33.

- Izsaukamo funkciju padod *CreateThread* parametrā #3, bet tai nododamo parametru parametrā #4 (kā redzams mūsu piemērā, funkcijai tiek padota norāde uz mūsu doto nosaukumu procesam (A vai B)).
- Bez tam Windows sistēma pieprasa padot papildus procesa identifikatora informāciju caur parametru #6), kas, manuprāt, ir liekvārdība, jo pēc procesa palaišanas procesu identificē attiecīgā norāde (*handle*).
- Funkcija atgriež norādi uz izveidoto procesu.

Funkcija *WaitForMultipleObjects* nodrošina, ka programma neiet tālāk, kamēr visi pakārtotie procesi, kas tiek padoti masīvā formā caur parametru #2, nav beigušies – šādi tiek nodrošināta sinhronizācija.

Pēc procesa darbības beigām jāpaziņo sistēmai par resursa atbrīvošanu (funkcija *CloseHandle*), kas pēc būtības ir kaut kas līdzīgs faila aizvēršanai.

Funkcijā *process* tiek papildus izmantota sistēmas funkcija *Sleep*, kas nodrošina gaidīšanu attiecīgo skaitu milisekunžu.

Pirmkods 13.1. Paralēlie procesi Windows vidē (*sys1threadswin.cpp*)

```
01 #include <windows.h>
02 #include <iostream>
03 using namespace std;
04
05 const int ITERATION_COUNT = 4;
06 DWORD WINAPI process (void *ptr);
07
08 int main()
09 {
10     HANDLE thread_a, thread_b;
11     char id_a={'A'}, id_b={'B'};
12     DWORD id2_a, id2_b;
13     thread_a = CreateThread (NULL, 0, process, (void*)&id_a, 0,
14                             &id2_a);
15     thread_b = CreateThread (NULL, 0, process, (void*)&id_b, 0,
16                             &id2_b);
17     HANDLE tt[2];
18     tt[0] = thread_a;
19     tt[1] = thread_b;
20     WaitForMultipleObjects (2, tt, TRUE, INFINITE);
21     CloseHandle (thread_a);
22     CloseHandle (thread_b);
23     cout << "END OF " << id_a << " " << id_b << endl;
24     return 0;
25 }
26
27 DWORD WINAPI process (void *ptr)
28 {
29     for (int i=0; i<ITERATION_COUNT; i++)
30     {
31         cout << *((char*)ptr) << i << endl;
32         Sleep (rand()%2*100);
33     }
34     cout << "End of the thread " << *((char*)ptr) << endl;
35 }
```

Programmas darbības piemērs:

```
A0
B0
A1
B1
A2
A3
B2
B3
End of the thread End of the thread B
A
END OF A B
```

Pamēģiniet laist programmu no komandrindas vairākkārtīgi, iegūstot dažādus rezultātus (funkcijā `process` notiek gadījuma skaitļu izmantošana), bez tam dažādu rezultātu iegūšana parāda, ka, pirms tas netiek speciāli pateikts (šeit ar `WaitForMultipleObjects`), tikmēr nekāda sinhronizācija starp procesiem nenotiek.

13.1.2. Pavedienprocesi POSIX standartā

Atšķirībā no Windows standarta, POSIX nav kādas konkrētas operētājsistēmas standarta, bet gan standarts, kuru pēc tradīcijas atbalsta *Unix/Linux* sistēmas.

Pirmkoda piemērs 13.2 parāda to pašu programmu, kas iepriekš, tikai POSIX variantā (attiecīgi to var nokompilēt tikai *Unix/Linux* sistēmā).

Lai POSIX standartā strādātu ar procesiem, jāielādē bibliotēka `<pthread.h>`.

Pēc līdzības ar Windows *handle*, vispirms ir jāizveido procesa objekti ar tipu `pthread_t` (rinda 9).

Procesu palaiž ar funkciju `pthread_create` (rindas 12 un 13), kas izsauc funkciju, kam ir viens parametrs (`void *`) ar atgriežamo tipu (`void *`). Funkcija un tās parametrs tiek padots caur parametriem #3 un #4. Funkcija `pthread_create` caur parametru #1 inicializē procesa objektu.

Sinhronizāciju POSIX standartā veic ar funkciju `pthread_join` (rindas 14 un 15), un tas tiek veikts katram sinhronizējamam procesam atsevišķi (nevis ar vienu kopēju komandu, kā ir *Windows*).

Pirmkods 13.2. POSIX paralēlie procesi (`sys2threadsposix.cc`)

```
01 #include <iostream>
02 #include <pthread.h>
03 using namespace std;
04
05 void *process (void *ptr);
06
07 int main()
08 {
09     pthread_t thread_a, thread_b;
10     char id_a='A', id_b='B';
11     char *res_a, *res_b;
12     pthread_create (&thread_a, NULL, process, (void*)&id_a);
13     pthread_create (&thread_b, NULL, process, (void*)&id_b);
14     pthread_join (thread_a, (void**)&res_a);
15     pthread_join (thread_b, (void**)&res_b);
```

```
16     cout << "END OF " << *res_a << ' ' << *res_b << endl;
17     pthread_exit (NULL);
18     return (0);
19 }
20
21 void *process (void *ptr)
22 {
23     for (int i=0; i<4; i++)
24     {
25         cout << "" << *((char*)ptr) << i << endl;
26         sleep (rand()%2);
27     };
28     cout << "End of the thread " << *((char*)ptr) << endl;
29     return ptr;
30 }
```

Programmas darbības piemērs:

```
A0
B0
B1
A1
B2
A2
B3
End of the thread B
A3
End of the thread A
END OF A B
```

13.2. Logu programmēšana Windows

Nākošais piemērs demonstrē vienkāršu logu programmu Windows vidē, kas dara sekojošas darbības vai spēj veikt sekojošus uzdevumus:

- Izveido logu ar nosaukumu „The Hello World program”;
- Loga pirmajā rindiņā izveda tekstu „Hallo World”, kuru pārzīmē arī mainot loga izmērus vai aktivizējot logu;
- Loga otrajā rindiņā skaita sekundes kopš programmas palaišanas;
- Loga trešajā rindiņā izveda nospiešamā taustiņa kodu;
- Var tikt aizvērts arī, nospiežot taustiņu *Esc*.

Lai palaistu programmu, projekta tipam Dev-C++ vidē jābūt *Win32 GUI* (pierastā *WIN32 Console* vietā, sk. *sys3window.dev*), tāpēc tas nav iespējams ārpus Dev-C++ projekta (pa taisno no *cpp* faila).

Programma sastāv no šādām galvenajām daļām (pirmkoda piemērs 13.3):

- Funkcija *WinMain* (rindas 10-59) Windows sistēmā aizstāj pierasto *main*. Tādējādi Windows sistēmā C++ vairs nav „tīrs”, bet ir ticis inkorporēts sistēmā.
- Tiek inicializēta struktūra loga izveidošanai (rindas 17-32).
- Tiek izveidots logs (rindas 34-48).
- Visa programmas darbība ir „ķert” notikumus un tos apstrādāt, ko reprezentē t.s. notikumu cikls (rindas 50-55).

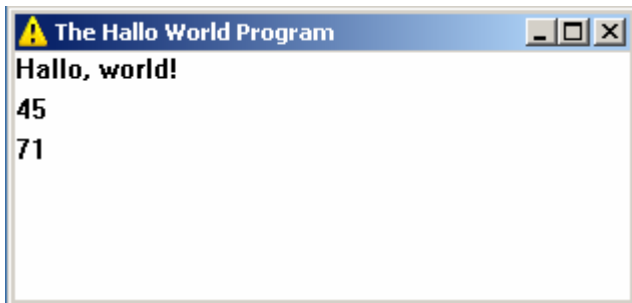
- Katra atsevišķa notikuma apstrādi veic funkcija *WndProc* ar fiksētu interfeisu (rindas 7, 20, 61-125). Šo funkciju (atšķirībā no *WinMain*) drīkst nosaukt citādi, tikai tas attiecīgi jāpiereģistrē (rinda 20).

Funkcijā *WndProc* redzami galvenie *Windows* notikumi:

- WM_CREATE – loga izveidošana;
- WM_PAINT – loga pārzīmēšana (piemēram, mainot izmēru vai aktivizējot), ;
- WM_TIMER – taimera notikums;
- WM_DESTROY – loga likvidēšana;
- WM_KEYDOWN – taustiņa nospiešana, funkcijas parametrs #3 (šeit *wParam*) nosaka nospiebtā taustiņa kodu.

Teksta izvadei uz ekrāna tiek lietota funkcija *TextOut*.

Lai veiktu izvadi logā, *Windows* sistēmā tiek noteiktā veidā izmantots t.s. aparatūras konteksts (*device context, DC*) un zīmēšanas parametru konfigurēšanas struktūra (*paint structure*), rindas 66, 77-80, 85, 90, 93, 112-115.



Attēls 13.1. Programmas (13.3) darbības piemērs

Pirmkods 13.3. Vienkāršs logs *Windows* sistēmā (*sys3window.cpp*)

```
01 #include <windows.h>
02 #include <string>
03 #include <stdio.h>
04
05 using namespace std;
06
07 LRESULT CALLBACK WndProc(HWND hWnd, UINT nMsg, WPARAM wParam,
08                               LPARAM lParam);
09
10 int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPreInst,
11                     LPSTR lpszCmdLine, int nCmdShow)
12 {
13     HWND          hWnd;
14     MSG           msg;
15     WNDCLASSEX    wc;
16
17     //fill the WNDCLASSEX structure with the appropriate values
18     wc.cbSize = sizeof(WNDCLASSEX);
19     wc.style = CS_HREDRAW | CS_VREDRAW;
20     wc.lpfnWndProc = WndProc;
21     wc.cbClsExtra = 0;
22     wc.cbWndExtra = 0;
23     wc.hInstance = hInst;
24     wc.hIcon = LoadIcon(NULL, IDI_EXCLAMATION);
```

```
25     wc.hCursor = LoadCursor(NULL, IDC_ARROW);
26     wc.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
27     wc.lpszMenuName = NULL;
28     wc.lpszClassName = "Hallo World";
29     wc.hIconSm = LoadIcon(NULL, IDI_EXCLAMATION);
30
31     //register the new class
32     RegisterClassEx(&wc);
33
34     //create a window
35     hWnd = CreateWindowEx(
36         WS_EX_APPWINDOW,
37         "Hallo World",
38         "The Hallo World Program",
39         WS_OVERLAPPEDWINDOW | WS_VISIBLE,
40         CW_USEDEFAULT,
41         CW_USEDEFAULT,
42         CW_USEDEFAULT,
43         CW_USEDEFAULT,
44         NULL,
45         NULL,
46         hInst,
47         NULL
48     );
49
50     //event loop - handle all messages
51     while(GetMessage(&msg, NULL, 0, 0))
52     {
53         TranslateMessage(&msg);
54         DispatchMessage(&msg);
55     }
56
57     //standard return value
58     return (msg.wParam);
59 }
60
61 LRESULT CALLBACK WndProc(HWND hWnd, UINT nMsg, WPARAM wParam,
62                             LPARAM lParam)
63 {
64     static int Ticks = 0;
65     //device context used for drawing
66     HDC hDC;
67
68     //find out which message is being sent
69     switch(nMsg)
70     {
71         case WM_CREATE:
72             //create the timer (0.1 seconds)
73             SetTimer(hWnd, 1, 100, NULL);
74             break;
75
76         case WM_PAINT:
77             PAINTSTRUCT ps;
78             hDC = BeginPaint (hWnd, &ps);
79             TextOut (hDC, 0, 0, "Hallo, world!", 13);
80             EndPaint (hWnd, &ps);
```

```
81         break;
82
83     case WM_TIMER: //when the timer goes off (only one)
84         //get the dc for drawing
85         hDC = GetDC(hWnd);
86         if (Ticks % 10 == 0)
87         {
88             char text[20];
89             sprintf (text, "%i", Ticks/10);
90             TextOut (hDC, 0, 20, text, strlen(text));
91         };
92         Ticks = (Ticks+1) % 1000000;
93         ReleaseDC(hWnd, hDC);
94         break;
95
96     case WM_DESTROY:
97         //destroy the timer
98         KillTimer(hWnd, 1);
99         //end the program
100        PostQuitMessage(0);
101        break;
102
103     case WM_KEYDOWN:
104         if (wParam == VK_ESCAPE)
105         {
106             SendMessage(hWnd, WM_DESTROY, 0, 0);
107         }
108         else
109         {
110             char text[20];
111             sprintf (text, "%i", wParam);
112             hDC = GetDC(hWnd);
113             TextOut (hDC, 0, 40, "      ", 7);
114             TextOut (hDC, 0, 40, text, strlen(text));
115             ReleaseDC(hWnd, hDC);
116         };
117         break;
118
119     default:
120         //let Windows handle every other message
121         return(DefWindowProc(hWnd, nMsg, wParam,
122                                lParam));
123
124     return 0;
125 }
```