Vilniaus universitetas

Matematikos ir informatikos fakultetas

Programų sistemų katedra

Valdas Undzėnas

http://www.mif.vu.lt/~valund/

I N F O R M A T I K A

Programavimas C++ kalba

Mokymo priemonė

Dalykas skaitomas VU Fizikos fakulteto 2 kurso studentams

Vilnius - 2009

Turinys

ĮVADAS 5
i. Bendrieji informatikos klausimai 5
ii. Trumpai apie kompiuterių programinę įrangą. C++ programų apdorojimo etapai 6
iii. Apie mokymo priemonę 7
1. C++ programos struktūra 8
2. Kintamieji. Duomenų tipai 11
2.1. Vardai (identifikatoriai) 11
2.2. Pagrindiniai duomenų tipai 11
2.3. Kintamųjų skelbimas 12
2.4. Kintamųjų galiojimo sritis 12
2.5. Kintamųjų inicializacija 13
2.6. Pažintis su simbolių eilutėmis 13
3. Konstantos 15
3.1. Konstantų tipai 15
3.2. Apibrėžtosios konstantos (*#define*) 17
3.3. Paskelbtosios konstantos (*const*) 18
4. Operatoriai 19
4.1. Priskyrimo operatorius (=) 19
4.2. Aritmetiniai operatoriai (+ , - , * , / , %) 20
4.3. Sudėtiniai priskyrimo operatoriai (+= , -= , *= , /= , %= , >>= , <<= , &= , ^= , ||=) 20
4.4. Reikšmės didinimo ir mažinimo operatoriai (++ , --) 21
4.5. Palyginimo operatoriai (== , != , > , < , >= , <=) 22
4.6. Loginiai operatoriai (! , &&, ||) 23
4.7. Sąlygos operatorius (?) 24
4.8. Kablelio operatorius (,) 24
4.9. Bitų operatoriai (& , | , ^ , ~ , << , >>) 25
4.10. Aiškiai nurodomi duomens tipo keitimo operatoriai 25
4.11. Operatorius *sizeof()* 25
4.12. Kiti operatoriai 26
4.13. Operatorių pirmumo lygiai 26
5. Pagrindinis duomenų įvedimas/išvedimas 28
5.1. Standartinis išvedimas (*cout*) 28
5.2. Standartinis įvedimas (*cin*) 30
5.3. *cin* ir simbolių eilutės 31
5.4. *stringstream* funkcija 32

5.5. Duomenų įvedimo/išvedimo funkcijos *printf* ir *scanf* 33
5.6. Darbas su duomenų failais 36

6. Valdantieji sakiniai 40
6.1. Sąlygos sakinys: *if* ir *else* 40
6.2. Ciklai (*while, do-while, for*) 41
6.3. Peršokimo sakiniai (*break, continue, goto, exit()*) 44
6.4. Variantinis sakinys (*switch*) 46
6.5. Paprastų programų pavyzdžiai 47

7. Funkcijos 53
7.1. *void* tipo funkcijos 55
7.2. Argumentų reikšmių ir adresų perdavimas parametrams 56
7.3. Numatytųjų reikšmių parametrai 58
7.4. Funkcijos vienodais vardais 59
7.5. Funkcijų šablonai (*template*) 59
7.6. *inline* funkcijos 62
7.7. Rekursijos 62
7.8. Funkcijų skelbimas 63

8. Masyvai 66
8.1. Masyvų inicializacija 66
8.2. Kreipimasis į masyvo elementus 67
8.3. Daugiamačiai masyvai 68
8.4. Masyvai kaip funkcijų parametrai 69
8.5. Programų su masyvais pavyzdžiai 71

9. Simbolių sekos 75
9.1. Simbolių sekų inicializacija 75
9.2. Simbolių sekų naudojimas 76

10. Rodyklės (pointers) 78
10.1. Adreso operatorius (&) 78
10.2. Duomens nurodytu adresu operatorius (*) 79
10.3. Rodyklės tipo kintamųjų paskelbimas 80
10.4. Rodyklės ir masyvai 81
10.5. Rodyklių inicializacija 82
10.6. Rodyklių aritmetika 83
10.7. Rodyklių rodyklės 84
10.8. *void* rodyklės 85
10.9. *null* rodyklė 86
10.10. Funkcijų rodyklės 86

11. Dinaminė atmintis 88
11.1. Operatoriai *new* ir *new[ ]* 88

11.2. Operatoriai *delete* ir *delete[ ]* ... 89
11.3. Dinaminė atmintis ANSI-C++ standarte ... 91
12. Duomenų struktūros ... 92
12.1. Duomenų struktūros ... 92
12.2. Struktūrų rodyklės ... 95
12.3. Struktūros struktūrose ... 97
12.4. Dinaminės duomenų struktūros: sąrašai, medžiai ir kita ... 97
13. Kiti duomenų tipai ... 102
13.1. Apibrėžtieji duomenų tipai (*typedef*) ... 102
13.2. Bendrosios atminties sritys (*union*) ... 102
13.3. Bevardės bendrosios atminties sritys *union* ... 104
13.4. Išvardijimai (*enum*) ... 105
14. Klasės ir objektai ... 106
14.1. Konstruktoriai ir destruktoriai ... 109
14.2. Klasės su keliais konstruktoriais ... 111
14.3. Numatytasis (*default*) konstruktorius ... 112
14.4. Klasių rodyklės ... 113
14.5. Klasių skelbimas naudojant *struct* ir *union* ... 114
14.6. *static* nariai klasėse ... 115
14.7. Klasių šablonai (template) ... 116
15. Draugiškumas ir paveldimumas ... 118
15.1. Draugiškosios funkcijos (*friend*) ... 118
15.2. Draugiškosios klasės ... 119
15.3. Klasių paveldimumas ... 120
15.4. Kas nepaveldima iš bazinės klasės? ... 123
15.5. Daugeriopas paveldimumas ... 123
16. Polimorfizmas ... 125
16.1. Bazinių klasių rodyklės ... 125
16.2. Virtualūs klasių nariai ... 126
16.3. Abstrakčiosios bazinės klasės ... 128
17. Dar apie duomenų failų įvedimą/išvedimą ... 133
17.1. Įvedimo/išvedimo funkcijos *read()* ir *write()* ... 133
17.2. Įvedimo/išvedimo funkcijos *fscanf(), fprintf()* ir kitos ... 134
Šaltiniai ... 139
Priedas. C++ standartinės bibliotekos. ... 140

ĮVADAS

i. Bendrieji informatikos klausimai

Kas yra informacija? Skaičiai, tekstai, garsai, vaizdai, kt. Informacijos perdavimo būdai: knygos, telefonas, radijas, televizija, kompiuterių tinklai (einama link viskam bendro duomenų perdavimo tinklo). Problema - kryptingas informacijos apdorojimas: ją reikia dėlioti (kaupti) taip, kad galėtume greičiausiai susirasti mums reikalingą informaciją (rikiavimas pagal autoriaus pavardę, mokslo sritis, bazinius žodžius, kt.); kompaktiškas saugojimas, greitas perdavimas.

Informatika - mokslas apie informacijos apdorojimo metodus, informacijos technologijas, kompiuterių aparatinės ir programinės įrangos kūrimą, kt.

Kompiuterių istorija: jų kartos, universalūs ir specializuoti, skaitmeniniai ir analoginiai.

Kompiuterio struktūra: įrenginių paskirtis, ryšiai tarp jų, kompiuterių architektūra. IBM PC sudėtis, įrenginių charakteristikos.

Duomenų vaizdavimas kompiuteriuose: 2 (dvejetainė) skaičiavimo sistema ir jos vartojimo priežastys. Ryšys tarp 2 skaič. sist. ir 10 skaič. sist. Apie 8 skaič. sist. ir 16 skaič. sist., kodėl jų prireikė. Skaičių vaizdavimas fiksuotu ir slankiu kableliu. Simboliai ir jų kodavimas.

Programinis kompiuterių valdymo principas. Programa - tai mašininių komandų seka. Mašininės komandos struktūra.

Algoritmas: kokie veiksmai ir kokia eilės tvarka jie turi būti atlikti, norint išspręsti kažkokį uždavinį. Nuosekli veiksmų seka.

Programavimo automatizavimas: operacinės sistemos, programavimo kalbos, programų paketai ir t.t.

Komandinė ir grafinė sąveikos: dialogo tarp žmogaus ir kompiuterio palaikymo būdai. Pvz., MS DOS aplinka, NORTON COMMANDER, WINDOWS, LINUX. Dažnai sutinkama santrumpa GUI – *Graphic User Interface* (grafinė vartotojo sąveika).

Informatikos standartai: susitarimai laikytis vienodos tvarkos, taisyklių. Pvz., OSI (*Open System Interconnection*) modelis - tai standartų rinkinys, kurio reikia laikytis, kad skirtingų rūšių kompiuteriai galėtų palaikyti ryšį; protokolai - standartų rinkiniai, kurių laikomasi perduodant duomenis kompiuterių tinklais. ISO - tarptautinė standartizacijos organizacija; yra ir kitos: CEN, ETSI, t.t.

ii. Trumpai apie kompiuterių programinę įrangą. C++ programų apdorojimo etapai

Operacinė sistema (OS) - tai gatavų programų rinkinys, padedantis vartotojui prižiūrėti bei eksploatuoti kompiuterį. Kompiuterių gamintojai, programinės įrangos kūrėjai pateikia vartotojams OS užrašytą diske, pastoviojoje atmintyje, kt. Pvz., MS DOS, jo komponentės BIOS, COMMAND.COM, AUTOEXEC.BAT, tvarkyklės ("draiveriai"), NORTON COMMANDER; WINDOWS; UNIX; LINUX ir t.t. Įvairiems darbams palengvinti yra kuriama pagalbinė programinė įranga (programavimo kalbų kompiliatoriai, duomenų bazių valdymo sistemos, tekstų rengimo, elektroninio pašto, naršymo po internetą programos, kt.). Čia akcentuosime tik tuos komponentus, kurie reikalingi programuojant C++ kalba.

C++ kalba parašyta programa - tai tekstas iš įvairių simbolių. Pirmas uždavinys žmogui yra perkelti tokios programos tekstą į kompiuterio atmintį. Šiam tikslui gali būti panaudota pagalbinė programa - tekstų rengyklė (EDIT, KEDIT, NOTEPAD ar kt.; plačiai žinomas WORD šiems tikslams netinka). Programos tekstas turi būti užrašytas į failą, kurio vardo plėtinys yra `.c` arba `.cpp` (pvz., `abc.c` arba `abc.cpp`). Tekstų rengyklių teikiamos paslaugos yra labai plačios: įterpti ar ištrinti simbolį, eilutę, perkelti teksto dalį iš vienos vietos į kitą, kopijuoti, kt.

Kadangi kompiuteris gali veikti tik pagal mašininėmis komandomis išreikštą programą, todėl būtinas antras etapas. Jo metu įvesta į kompiuterio atmintį teksto pavidalo C++ programa, kuri vadinama programos pradiniu moduliu (`.cpp`), pertvarkoma į mašininių komandų pavidalą. Tą atlieka kita pagalbinė programa − C++ kompiliatorius. Jei kompiliatorius aptiko klaidų C++ programos pradiniame modulyje, reikia grįžti į pirmąjį etapą, tekstų rengykle ištaisyti programą ir vėl kompiliuoti ją.

Kompiliavimo ir redagavimo metu gaunama vykdymui parengta vartotojo programa mašinine kalba - vykdomasis modulis (.exe).

Pertvarkius C++ programą į mašininį pavidalą (suformavus programos kodą), belieka ją vykdyti. Netgi kompiuteriui dirbant pagal vartotojo sudarytą programą, į šį procesą gali įsiterpti OS.

Yra sukurtos integruotosios programavimo sistemos arba aplinkos (IDE − *Integrated Development Environment*), kuriose galima atlikti visų aukščiau minėtų pagalbinių programų veiksmus.

iii. Apie mokymo priemonę

Programavimo kalba C ir jos modifikacija C++ pasižymi programų rašymo glaustumu, elementų įvairumu. Ši mokymo priemonė skirta skaitytojams, kurie nori išmokti programuoti C++ kalba ir nėra mokęsi kitų kalbų.

1-13 skyriuose aiškinami C kalbos elementai, kurie yra C++ kalbos dalis. Tačiau netgi skaitytojams, kurie jau yra susipažinę su C kalba, rekomenduojama peržiūrėti šiuos skyrius, nes yra nežymių C++ ir C kalbos sintaksės skirtumų.

14-16 skyriuose aprašomos klasės, supažindinama su objektiniu programavimu.

Daugelyje skyrių yra pavyzdžiai, iliustruojantys C++ kalbos elementus. Patartina išnagrinėti juos, suprasti kiekvieną programos eilutę ir tik po to eiti prie naujos temos.

C++ kalbos tarptautinis standartas ANSI-C++ buvo priimtas palyginus neseniai. Jis buvo paskelbtas 1997 metais ir peržiūrėtas 2003 metais, nors pati C++ kalba egzistuoja nuo 1980 metų. Todėl yra daug ankstesnės laidos kompiliatorių, kurie nepalaiko visų minėto standarto C++ kalbos galimybių. Šioje mokymo priemonėje dėstoma medžiaga reikalauja ANSI-C++ standarto reikalavimus atitinkančių kompiliatorių. Yra nemažai komercinių, o taip pat laisvai platinamų C++ programavimo sistemų (programavimo sistema apima tekstų rengyklę, kompiliatorių, įvairias standartines bibliotekas, kt.), pvz., *Dev-C++ 5.0 beta 9.2 (4.9.9.2)*. Ją galima rasti adresu
http://www.bloodshed.net/dev/devcpp.html .

Šioje mokymo priemonėje pateikiami pavyzdžiai yra **konsolės** tipo programos. Tai tokios programos, kurios naudoja teksto pavidalo duomenis ryšiui su vartotoju palaikyti ir savo darbo rezultatams rodyti. Konsolė - tai kompiuterio valdymo pultas, turintis klaviatūrą ir ekraną. Visi C++ kompiliatoriai gali apdoroti konsolės tipo programas. Vartotojams tereikia pasiskaityti darbo su kompiliatoriumi vadovą (*user's manual*).

Be C++ yra ir kitos programavimo kalbos bei sistemos (aplinkos): Java, Pascal, Perl, PHP, Visual Basic ir daugelis kitų. Išmokti programavimo pagrindų naudojant C++ ir rašyti nesudėtingas kompiuterių programas yra šios mokomosios disciplinos uždavinys.

Mokymo priemonė parengta naudojant šaltinius [Sou06, Vid02, BBK+01].

1. C++ programos struktūra

Mokytis programavimo kalbos geriausia yra rašant programas ir vykdant jas kompiuteriu. Žemiau pateikiame pirmąją programą C++ kalba. Kairėje pusėje yra rodomas programos pradinis modulis (*source code*), o dešinėje – programos rezultatai, kuriuos ji išveda į ekraną. Kelias, kaip reikėtų įvesti, koreguoti ir kompiliuoti programą, priklauso nuo programavimo sistemos, kurią jūs naudojate.

```
// pirmoji programa C++ kalba

#include <iostream>
using namespace std;

int main ()
{ cout << "Sveiki gyvi!";
  return 0;
}
```

```
Sveiki gyvi!
```

Tai paprasčiausia C++ programa, tačiau joje jau yra pagrindiniai elementai, kurie sutinkami kiekvienoje programoje. Išnagrinėkime kiekvieną šios programos eilutę.

// pirmoji programa C++ kalba

Visa eilutė arba tik jos dalis, prasidedanti dviem pasvirusiais brūkšniais (//), yra komentaras ir įtakos programos darbui neturi. Kad būtų lengviau atskirti komentarus nuo programos teksto, komentarus rašysime kiek kitokiu šriftu. Programuotojai naudoja juos savo pastaboms ar paaiškinimams į programos tekstą įterpti. Duotoje programoje tai tik pastaba, kas tai per programa.

`#include <iostream>`

Simboliu # prasidedanti eilutė yra nuoroda preprocesoriui. Preprocesorius – tai kompiliatoriaus dalis. `#include <iostream>` nurodoma, kad preprocesorius įtrauktų į mūsų programą priemones iš standartinio `<iostream>` failo. Tai gatavų C++ programų biblioteka duomenims įvesti/išvesti atitinkamu būdu. Jų reikia mūsų programoje.

Preprocesorius programoje sutikęs nuorodą `#include` (ji rašoma atskiroje eilutėje), į programos tekstą įstato visą nurodyto failo turinį:

```
#include "failo_vardas"
#include <failo_vardas>
```

Šių dviejų nuorodų skirtumas yra toks, kad kompiliatorius ieško nurodyto failo skirtinguose disko kataloguose. Kai `failo_vardas` yra apkabintas dvigubomis kabutėmis,

kompiliatorius visų pirma failo ieško tame kataloge, kuriame yra #include nuorodą turinti programa. Šiame kataloge neradus nurodyto failo, paieška tęsiama numatytajame (*default*) kataloge, kuris konfigūruojant kompiliatorių buvo paskirtas standartinių (gatavų) bibliotekų failams. Jei failo_vardas yra apskliaustas ženklais < ir >, jo ieškoma tik kataloge, kuris paskirtas standartinių bibliotekų failams.

Šalia standartinių bibliotekų failų programuotojai gali susikurti asmeninių bibliotekų failus ir #include nuorodomis įtraukti juos į įvairias savo programas kai tik to prireikia.

```
using namespace std;
```

Visi standartinėse bibliotekose naudojami elementai yra aprašyti vardų sąraše (*namespace*) std. Šitokia programos eilute nurodoma, kad norime naudotis to vardų sąrašo elementais. Tai daugeliui C++ programų, naudojančių standartines bibliotekas, reikalingas sakinys.

```
int main ()
```

Šitokia eilute nurodoma programos pagrindinės *(main)* funkcijos pradžia. Kiekvienoje C++ programoje turi būti pagrindinė funkcija. Nuo jos pradedamas visos programos darbas, nesvarbu kurioje programos pradinio modulio vietoje ji būtų parašyta. Po main rašomi skliaustai (), kurie C++ kalboje nurodo, kad prieš juos einantis vardas yra funkcijos, o ne kitokio tipo elemento vardas. Tarp skliaustų gali būti nurodomi parametrai. Toliau rašomas pagrindinės funkcijos programos tekstas turi būti tarp riestinių skliaustų { }.

```
cout << "Sveiki gyvi!";
```

Ši eilutė yra C++ kalbos sakinys (angl. *statement*). Sakinys − tai paprasta arba sudėtinga išraiška, kuria nurodomi konkretūs reikiami atlikti veiksmai. cout nurodoma, kad į ekraną reikia išvesti simbolių seką, t.y. tekstą Sveiki gyvi!

cout programa imama iš standartinio failo <iostream>, kurios elementai yra aprašyti vardų sąraše std. Todėl šioje programoje ir prireikė eilučių #include <iostream> ir using namespace std;

Pastebėkime, kad C++ kalboje sakiniai baigiami kabliataškiu.

```
return 0;
```

return sakiniu baigiamas pagrindinės funkcijos darbas. Po jo nurodomas programos baigimo kodas. 0 reiškia, kad pagrindinė funkcija veikė normaliai ir baigė darbą be klaidų. Paprastai C++ programos baigiamos tokiu sakiniu. Kadangi pagrindinės funkcijos (*main*) pradžią nurodančioje eilutėje yra int, tai baigimo kodas šioje programoje yra nurodytas sveikuoju skaičiumi (*integer*). Pagrindinės funkcijos baigimo kodas gali būti panaudotas organizuoti tolesniam kompiuterio darbui. Kai rašoma void main() arba tik main(), tai pagrindinė funkcija jokio baigimo kodo negrąžina ir return sakinys nereikalingas.

Atkreipkime dėmesį į tai, kad ne visos programos eilutės iššaukia konkrečius veiksmus vykdant sukompiliuotą programą. Dalis iš jų yra tik paaiškinimai (pvz., *// pirmoji programa C++ kalba*) arba nuorodos (pvz., `#include <iostream>`).

Programos tekstą skaidyti į eilutes galima įvairiai. Svarbu, kad jis žmogui būtų lengvai skaitomas. Pvz., vietoje

```
int main ()
{ cout << "Sveiki gyvi!";
  return 0;
}
```

galima rašyti

```
int main () { cout << "Sveiki gyvi!"; return 0; }
```

arba

```
int main () {
  cout << "Sveiki gyvi!"; return 0; }
```

Preprocesoriaus nuorodoms, t.y. toms eilutėms, kurios pradedamos simboliu #, ši bendroji sakinių taisyklė netaikoma. Jos rašomos atskirose eilutėse, gale nededant kabliataškio.

Komentarai programoje gali būti rašomi dviem būdais:

```
// komentaras eilutės pabaigoje
/* komentaras bet kurioje eilutės vietoje
    arba apimantis kelias eilutes */
```

Komentarai, prasidedantys simbolių pora `/*` ir pasibaigiantys `*/` , gali užimti kelias eilutes. Pailiustruokime tai modifikuodami mūsų pirmąją programą.

```
    /* pirmoji programa C++ kalba,
       kuri yra daugiau pakomentuota */

#include <iostream>          // nuoroda preprocesoriui
using namespace std;

int main ()                  // pagrindinės funkcijos pradžia
{ cout << "Sveiki gyvi!";    // išves į ekraną Sveiki gyvi!
  return 0;
}
```

```
Sveiki gyvi!
```

2. Kintamieji. Duomenų tipai

Kintamasis programoje nurodomas vardu (identifikatoriumi), ir jis programos bėgyje gali keisti savo reikšmę. Kintamasis suprantamas kaip kompiuterio operatyviosios atminties tam tikro dydžio sritis, kurioje galima įsiminti nustatyto tipo reikšmę (sveikąjį skaičių, realųjį skaičių, tekstą, kt.). Atminties srities dydis priklauso nuo kintamojo tipo. Programoje visi kintamieji turi būti aprašyti, nurodytas jų tipas.

2.1. Vardai (identifikatoriai)

Vardas − tai raidžių, skaitmenų ir pabraukimo simbolių (_) seka, prasidedanti raide. Varduose vartoti kitokių simbolių negalima. Pabraukimo simboliu prasidedantys vardai naudojami tik kompiliatoriuje, kaip specialią paskirtį turintys vardai.

Kintamųjų vardai negali sutapti su C++ kalbos baziniais žodžiais

asm, auto, bool, break, case, catch, char, class, const, const_cast, continue, default, delete, do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, operator, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while,

o taip pat žodžiais

and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, xor_eq.

Kai kurie kompiliatoriai gali turėti ir kitų rezervuotų žodžių, kurių negalima vartoti kaip kintamųjų vardų.

Pastaba: C++ kalboje didžiosios ir mažosios raidės varduose traktuojamos skirtingai. Todėl, pvz., vardai ABC, Abc, aBc, abc yra skirtingi.

2.2. Pagrindiniai duomenų tipai

Pagrindiniai duomenų tipai nurodomi baziniais žodžiais. Faktiškai tipas atspindi galimą duomens reikšmių diapazoną ir kiek tokiai reikšmei saugoti atmintyje skiriama baitų.

Tipas	Reikšmių diapazonas	Užima baitų
char	signed: nuo -128 iki 127 , simbolio kodas unsigned: nuo 0 iki 255 , simbolio kodas	1
short int (short)	signed: nuo -32768 iki 32767 unsigned: nuo 0 iki 65535	2

int	signed: nuo -2147483648 iki 2147483647 unsigned: nuo 0 iki 4294967295	4
long int (long)	signed: nuo -2147483648 iki 2147483647 unsigned: nuo 0 iki 4294967295	4
bool	true arba false	1
float	3.4e +/- 38 (mantisėje - 7 skaitmenys)	4
double	1.7e +/- 308 (mantisėje - 15 skaitmenų)	8
long double	1.7e +/- 308 (mantisėje - 15 skaitmenų)	8
wchar_t	platusis simbolis	2

Pastaba: skirtinguose kompiliatoriuose duomenų tipai gali būti realizuoti skirtingai, t.y. gali skirtis šioje lentelėje nurodyti reikšmių diapazonai, užimamos atminties dydis.

2.3. Kintamųjų skelbimas

Kintamasis gali būti naudojamas tik paskelbus jį ir nurodžius, kokio tipo duomenims jis yra skirtas. Skelbimo sintaksė yra tokia: pirma nurodomas duomenų tipas, o po jo - vieno arba kelių kintamųjų vardai.

```
int  a;
int  b, c, d;
float  x, y;
```

Sveikojo tipo kintamieji gali turėti ženklą (gali įgyti teigiamas ir neigiamas reikšmes) ir gali būti be jo (gali įgyti tik teigiamas reikšmes). Tai nurodoma žodžiais signed ir unsigned. Pavyzdžiui:

```
unsigned short int  kiekis;
signed int  temperatura;
```

Nurodant sveikųjų kintamųjų ilgį, vietoje short int galima rašyti tik short, o vietoje long int tik long.

Taip pat sveikiesiems kintamiesiems vietoje signed int galima rašyti tik signed, o vietoje unsigned int - tik unsigned.

2.4. Kintamųjų galiojimo sritis

Kintamieji gali būti globalieji ir lokalieji. Globalieji yra tie, kurie programos tekste yra aprašyti dar prieš pagrindinę ir kitas funkcijas. Lokalieji kintamieji yra tokie, kurie paskelbti funkcijos arba programos bloko viduje, t.y. riestiniais skliaustais {} apribotame programos tekste.

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

int a;                          //
char r;                         //  globalieji kintamieji
unsigned int c;                 //

int main ()
{ int metai;                    //
  float ada;                    //  lokalieji kintamieji
  cout << "Kiek jums metu?";
  cin >> metai;
  …
  return 0;
}
```

Į globaliuosius kintamuosius galima kreiptis iš bet kurios programos vietos, einančios po šių kintamųjų paskelbimo. Lokaliųjų kintamųjų galiojimo sritis yra apribota riestiniais skliaustais {} , ir į juos galima kreiptis nuo paskelbimo vietos iki uždarančio riestinio skliausto.

2.5. Kintamųjų inicializacija

Paskelbto kintamojo reikšmė paprastai būna neapibrėžta (nežinoma). Tačiau paskelbimo metu kintamajam galima suteikti pradinę reikšmę, t.y. inicializuoti kintamąjį. C++ kalboje tai galima atlikti dviem būdais:

1) po kintamojo vardo rašant lygybės ženklą ir nurodant pradinę reikšmę;

2) po kintamojo vardo skliaustuose nurodant pradinę reikšmę.

Pvz.,

```
int  a = 5;
int  b (2);
```

2.6. Pažintis su simbolių eilutėmis

Kintamieji, kurie gali įsiminti daugiau kaip vieną simbolį, yra vadinami simbolių eilutėmis arba tiesiog eilutėmis. C++ kalbos standartinėje bibliotekoje `<string>` yra priemonės, palengvinančios jų naudojimą. Eilutės nepriklauso pagrindinių duomenų tipui, tačiau jos funkcionuoja panašiai, kaip daugelis pagrindinio tipo duomenų. Aprašant eilutes, jų tipas nurodomas baziniu žodžiu string.

Pirmas esminis eilučių skirtumas nuo pagrindinio tipo duomenų yra tas, kad, prieš skelbiant jas, į programą turi būti įtraukta standartinė biblioteka `<string>` ir naudojama vardų sritis (*namespace*) std.

```
        // pirmas simbolių eilutės pavyzdys
#include <iostream>
#include <string>
using namespace std;

int  main ()
{ string  se = "Tai eilute";   // skelbiama ir inicializuojama eilute
   cout << se;
   return 0;
}
```

```
Tai eilute
```

Skelbimo metu eilučių tipo kintamiesiems, kaip ir skaitiniams kintamiesiems, pradinės reikšmės gali būti suteiktos dviem būdais. Pvz.,

```
string  se = "Tai eilute";
string  se ("Tai eilute");
```

Jeigu skelbiant eilutę pradinė reikšmė jai nesuteikiama, tai, kaip ir kitokiems kintamiesiems, reikšmė gali būti suteikiama programos bėgyje priskyrimo operatoriumi. Pvz.,

```
          // programos su eilute pavyzdys
#include <iostream>
#include <string>
using namespace std;

int  main ()
{ string  se;
  se = "Tai pirma eilutes reiksme";             // priskyrimo sakinys
  cout << se << endl;
  se = "Cia jau kita eilutei suteikiama reiksme ";
  cout << se << endl;
  return 0;
}
```

3. Konstantos

Konstanta – tai išraiška, kurios reikšmė išlieka tokia pati visą programos darbo laiką.

3.1. Konstantų tipai

Sveikieji skaičiai.

```
1746    +508    -273
```

Čia parodytos trys dešimtainių skaičių konstantos.

Sveikųjų skaičių konstantos gali būti rašomos ir aštuntaine arba šešioliktaine skaičiavimo sistema. Aštuntainio skaičiaus pradžioje rašomas 0 (nulis), o šešioliktainio – 0x (nulis ir x).

```
75      // dešimtainis skaičius
0113    // aštuntainis
0x4b    // šešioliktainis
```

Šiame pavyzdyje visų trijų konstantų reikšmės yra vienodos, tik jos užrašytos skirtingomis skaičiavimo sistemomis.

Kaip ir int tipo kintamieji, taip ir sveikųjų skaičių konstantos gali turėti požymius unsigned ir long. Jie nurodomi raidėmis u ir l skaičiaus gale. u ir l gali būti mažosios arba didžiosios.

```
75      // int tipo konstanta
75u     // unsigned int
75L     // long
75ul    // unsigned long
```

Slankaus kablelio skaičiai.

Tai dešimtainiai skaičiai, turintys ir trupmeninę dalį. Jie gali būti rašomi dvejopai: naudojant tašką trupmeninei daliai atskirti nuo sveikosios arba naudojant eksponentinį pavidalą, t.y. su raide e.

```
3.14159    // 3.14159
6.02e23    // 6.02 padauginta iš 10, pakeltos 23 laipsniu
1.6e-19    // 1.6 padauginta iš 10, pakeltos -19 laipsniu
3.0        // 3.0
```

Nesant papildomų nurodymų slankaus kablelio konstantos yra double tipo. Norint turėti float arba long double tipo konstantą, jos gale reikia rašyti raidę f arba l. Šiose konstantose raidės e, f ir l gali būti mažosios arba didžiosios.

```
3.14159L    // long double tipo konstanta
6.02E23f    // float
```

Simbolis ir eilutė.

```
'z'                  // simbolio konstanta, char tipo
"Sveiki gyvi"        // simbolių eilutės konstanta, string tipo
```

Simbolio tipo (char) konstantos rašomos tarp viengubų kabučių, o eilučių tipo (string) konstantos - tarp dvigubų kabučių.

Atkreipkime dėmesį į šiuos du užrašus:

```
x    'x'
```

Užrašas x reiškia kintamojo vardą, o užrašas 'x'– simbolio konstantą.

Kai kurie simboliai turi tam tikrų ypatumų, kaip, pvz., valdantieji simboliai (kai kuriuos atitinka klavišai, pvz., *Enter*, *Tab*, kt.). Tai specialūs simboliai, kuriuos sunku arba neįmanoma pavaizduoti programos tekste. Todėl valdantieji simboliai vaizduojami jų pradžioje rašant atgal pasvirusį brūkšnį (\ ) ir po to vieną kokį nors kitą simbolį. Štai valdančiųjų simbolių sąrašas:

Valdantysis simbolis	Reikšmė
\n	pereiti į kitos eilutės pradžią (pvz., *Enter* klavišas)
\r	grįžti į tos pačios eilutės pradžią (*Home* klavišas)
\t	tabuliacija (*Tab* klavišas)
\v	vertikali tabuliacija (spausdintuvams valdyti)
\b	grįžti per vieną poziciją (*Backspace* klavišas)
\f	*form feed* (spausdintuvams valdyti)
\a	perspėjimo signalas (pyptelėjimas)
\'	vienguba kabutė (')
\"	dviguba kabutė (")
\?	klaustukas (?)
\\	atgal pasviręs brūkšnys (\ )

Konstantų su valdančiaisiais simboliais pavyzdžiai:

```
'\n'
'\t'
"Kaire \t Desine"
"vienas\ndu\ntrys"
```

Taip pat bet kurį simbolį galima nurodyti jo skaitiniu ASCII kodu, prieš jį rašant atgal pasvirusį brūkšnį. Kodas gali būti užrašytas tik aštuntainiu arba šešioliktainiu skaičiumi. Aštuntainis kodas rašomas tuoj po brūkšnio, pvz., \25 , o šešioliktainis kodas dar turi x , pvz., \x4A .

Simbolių eilutės konstanta gali būti rašoma keliose iš eilės einančiose puslapio eilutėse. Pereinant į kitą puslapio eilutę rašomas atgal pasviręs brūkšnys. Pvz.,

```
    "simboliu eilute uzrasyta dviejose \
     puslapio eilutese"
```

Keletas eilučių, tarp kurių yra tarpai, tabuliacija (buvo paspaustas klavišas *tab*) ar perėjimas į naują puslapio eilutę (*enter*), yra sukabinamos ir reiškia vieną ilgą eilutę. Pvz.,

```
"Tai yra"     "viena ilga"     "simboliu eilute."
```

Jei norime turėti eilutę iš plačiųjų simbolių wchar_t (t.y. simbolių, kuriems atmintyje skiriama po 2 baitus), jos priekyje rašoma raidė L. Pvz.,

```
L"Tai placiuju simboliu eilute"
```

Loginės konstantos.

Tai bool tipo reikšmės. Jos užrašomos žodžiais true ir false .

3.2. Apibrėžtosios konstantos (*#define*)

Jeigu ta pati konstanta vartojama keliuose programos sakiniuose ir yra ilgas jos užrašas, patogiau ją pasivadinti vardu (vardinė konstanta), jam suteikti norimą reikšmę ir vartoti šį vardą, kur bus reikalinga tokia reikšmė. Be to, sumanius pakeisti konstantos reikšmę, tereikės taisyti vienoje programos vietoje, o ne visuose sakiniuose, kur tokia konstanta pavartota. C++ kalboje apibrėžtosios konstantos skelbiamos naudojant preprocesoriaus nuorodą #define. Šios nuorodos bendrasis pavidalas yra

```
#define vardas reikšmė
```

Pvz.,

```
#define  PI  3.14159265
#define  bruksnys  “\n------------\n“
```

Taip gavome dvi naujas apibrėžtąsias konstantas: PI ir bruksnys. Šiuos vardus galima naudoti programoje kaip ir bet kurią įprastą konstantą. Pvz.,

```
    //  apibrėžtosios konstantos programoje
    //  apskritimo ilgiui skaičiuoti
#include <iostream>
using namespace std;

#define  PI  3.14159
#define  bruksnys  “\n------------\n“

int main ()
{ double r = 5.0;              // spindulys
  double apskrilgis;

  apskrilgis = 2 * PI * r;
  cout << apskrilgis;
  cout << bruksnys;

  return 0;
}
```

```
31.4159
----------
```

#define yra preprocesoriaus nuoroda, o ne C++ kalbos sakinys. Todėl jos gale kabliataškis nededamas.

3.3. Paskelbtosios konstantos (*const*)

Bazinis žodis const įgalina paskelbti konstantas nurodant ir jų tipą, kaip tai daroma kintamųjų atveju. Pvz.,

```
const int  plotis = 100;
const char  tabul = '\t';
const  kuku = 12440;
```

Kai po const tipas nenurodomas (kaip paskutiniame pavyzdyje), laikoma, kad yra int.

4. Operatoriai

Kadangi jau susipažinome su kintamaisiais ir konstantomis, išnagrinėkime galimus veiksmus su jais. Veiksmai C++ kalboje nurodomi operatoriais. Operatorius – tai kažkoks simbolis arba jų derinys, kuriuo nurodoma atlikti kažkokį veiksmą.

4.1. Priskyrimo operatorius (=)

Priskyrimo operatoriumi reikšmė priskiriama kintamajam.

```
a = 5;
```

Šiuo sakiniu sveikoji reikšmė 5 priskiriama kintamajam a.

Pagrindinė priskyrimo sakinio taisyklė yra ta, kad jis vykdomas *iš dešinės į kairę:*

```
a = b;
```

Šiuo sakiniu nurodoma, kad reikia paimti kintamojo b reikšmę ir priskirti ją kintamajam a. Reikšmė, kurią iki šiol turėjo kintamasis a, čia nėra naudojama ir ji prarandama. Ateityje, keičiant kintamojo b reikšmę, įtakos kintamajam a tai neturės.

Panagrinėkime tokią C++ programą:

```
    // priskyrimo operatoriaus iliustracija
#include <iostream>
using namespace std;

int  main ()
{ int  a, b;          //  a=?,   b=?
  a = 10;             //  a=10, b=?
  b = 4;              //  a=10, b=4
  a = b;              //  a=4,   b=4
  b = 7;              //  a=4,   b=7

  cout << "a=";
  cout << a;
  cout << " b=";
  cout << b;

  return 0;
}
```

```
a=4 b=7
```

Šios programos rezultatas yra toks: kintamojo a reikšmė yra 4, o kintamojo b – 7. Pastebėkime, kad keičiant b reikšmę, kintamojo a reikšmė nekinta.

C++ kalbos ypatybė, lyginant ją su kitomis programavimo kalbomis, yra ta, kad priskyrimo operacija gali būti atliekama ir priskyrimo operatoriaus dešinėje pusėje. Pvz.,

```
a = 2 + (b = 5);
```

Tai ekvivalentiška tokiems dviems priskyrimo sakiniams:

```
b = 5;
a = 2 + b;
```

Tai reiškia, kad visų pirma kintamajam b priskiriama reikšmė 5, o po to, prie šios reikšmės pridėjus 2, gauta suma 7 priskiriama kintamajam a.

C++ kalboje taip pat yra teisinga tokia išraiška:

```
a = b = c = 5;
```

Šiuo sakiniu reikšmė 5 priskiriama visiems trims kintamiesiems a, b ir c.

4.2. Aritmetiniai operatoriai (+ , - , * , / , %)

C++ kalboje yra leistinos penkios aritmetinės operacijos:

+	sudėtis
-	atimtis
*	daugyba
/	dalyba
%	dalybos liekana

Sudėties, atimties, daugybos ir dalybos operacijos visiškai atitinka matematikoje naudojamas operacijas. Tik operatorius dviejų sveikųjų skaičių dalybos liekanai gauti yra žymimas procento ženklu. Pvz., jeigu užrašysime:

```
a = 11 % 3;
```

tai kintamasis a įgis reikšmę 2, kadangi tai yra skaičiaus 11 padalinto iš 3 dalybos liekana.

4.3. Sudėtiniai priskyrimo operatoriai (+= , -= , *= , /= , %= , >>= , <<= , &= , ^= , ||=)

Kintamojo reikšmei pakeisti, kai reikia atlikti veiksmą su jo ankstesne reikšme, galima naudoti sudėtinį priskyrimo operatorių. Pvz.:

Išraiška	Išraiškos ekvivalentas
dydis += prieaugis;	dydis = dydis + prieaugis;
a -= 5;	a = a - 5;
a /= b;	a = a / b;
kaina *= kiekis + 1;	kaina = kaina * (kiekis + 1);

Tas pats galioja ir visiems kitiems (% , >> , << , & , ^ , |) operatoriams.

Programos pavyzdys:

```
        // sudėtinis priskyrimas
#include <iostream>
using namespace std;

int  main ()
{ int  a, b=3;
  a = b;
  a += 2;            // ekvivalentiška a=a+2
  cout << a;
  return 0;
}
```

```
5
```

4.4. Reikšmės didinimo ir mažinimo operatoriai (++ , --)

Išraiškų, kuriomis norima vienetu padidinti arba sumažinti kintamojo reikšmę, trumpesnis užrašas gaunamas naudojant didinimo (++) arba mažinimo (--) operatorių. Jie yra ekvivalentiški atitinkamai operatoriams +=1 ir -=1 . Todėl sakinių

```
c++;    c += 1;    c = c+1;
```

atliekamos funkcijos yra ekvivalentiškos. Visi jie kintamojo c reikšmę padidina vienetu.

Šie operatoriai gali būti rašomi prieš kintamojo vardą arba po jo. Paprastos išraiškos, kaip a++ arba ++a , duoda tą patį rezultatą. Tačiau sudėtingesnėse išraiškose, ++ ar -- rašymas prieš ir po kintamojo vardo turi svarbų skirtumą. Pvz., rašant ++a , kintamojo a reikšmė visų pirma padidinama vienetu ir po to naudojama sudėtingoje išraiškoje, o rašant a++ , sudėtingoje išraiškoje naudojama turima a reikšmė ir tik po to ji didinama vienetu. Atkreipkime dėmesį į skirtumą šiuose pavyzdžiuose:

1 pavyzdys	2 pavyzdys
B=3; A=++B; // *A reikšmė - 4, B reikšmė - 4*	B=3; A=B++; // *A reikšmė - 3, B reikšmė - 4*

1 pavyzdyje B reikšmė visų pirma padidinama vienetu ir po to priskiriama kintamajam A, o 2 pavyzdyje B turima reikšmė visų pirma priskiriama kintamajam A ir tik po to padidinama vienetu.

4.5. Palyginimo operatoriai (== , != , > , < , >= , <=)

Dviejų išraiškų rezultatams palyginti naudojami palyginimo operatoriai. Jų darbo rezultatas yra loginio tipo (bool) reikšmė: true arba false . Žemiau pateikiamas C++ kalboje naudojamų palyginimo operatorių sąrašas:

==	lygu
!=	nelygu
>	daugiau
<	mažiau
>=	daugiau arba lygu
<=	mažiau arba lygu

Štai keletas palyginimo pavyzdžių:

```
(7 == 5)     // rezultatas false
(5 > 4)      //            true
(3 != 2)     //            true
(6 >= 6)     //            true
(5 < 5)      //            false
```

Vietoje čia palyginamų skaitinių konstantų, žinoma, gali būti bet kokios leistinos išraiškos arba kintamieji. Tarkime, kad a=2, b=3 ir c=6. Todėl:

```
(a == 5)        // rezultatas false, nes 2 nelygu 5
(a*b >= c)      //            true,    ( 2*3 >= 6 )
(b+4 > a*c)     //            false,   ( 3+4 > 2*6 )
((b=2) == a)    //            true
```

Būkime atidūs! Operatorius = (vienas lygybės ženklas) nėra tas pat, kaip operatorius == (du lygybės ženklai). Pirmasis yra priskyrimo operatorius, o antrasis − palyginimo. Todėl paskutinė išraiška ((b=2) == a) skaičiuojama taip: visų pirma 2 priskiriamas kintamajam b , o po to ši reikšmė palyginama su kintamojo a reikšme. Taigi, rezultatas yra true.

4.6. Loginiai operatoriai (! , &&, ||)

C++ kalboje šauktuku (!) nurodoma loginė operacija NE (angliškai NOT). Ji turi tik vieną operandą, kuris rašomas ženklo dešinėje, ir invertuoja jo loginę reikšmę, t.y. true keičia į false ir atvirkščiai. Pvz.:

```
!(5 == 5)   // rezultatas false, nors palyginimas (5 == 5) duoda true
!(6 <= 4)   // rezultatas true, nors palyginimas (6 <= 4) duoda false
!true       // rezultatas false
!false      // rezultatas true
```

Operatoriumi && (du simboliai &) nurodoma loginė operacija IR (angliškai AND). Jos rezultatas yra true tik tada, kai abiejų operandų reikšmės yra true.

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

Operatoriumi || (du vertikalūs brūkšniai) nurodoma loginė operacija ARBA (angliškai OR). Jos rezultatas yra true, kai bent vieno operando reikšmė yra true.

a	b	a || b
true	true	true
true	false	true
false	true	true
false	false	false

Pavyzdys:

```
((5 == 5) && (3 > 6))    // rezultatas false   (true && false)
((5 == 5) || (3 > 6))    //            true    (true || false)
```

4.7. Sąlygos operatorius (?)

Sąlygos operatoriaus ? (klaustukas) pavidalas yra toks:

išraiška ? rezultatas1 : rezultatas2

Kairėje operatoriaus pusėje rašoma loginė išraiška, o dešinėje - dvitaškiu atskirtos dvi reikšmės. Jei išraiškos reikšmė yra true, tai sąlygos operatoriaus rezultatas yra rezultatas1, o priešingu atveju – rezultatas2 . Pvz.:

```
7==5 ? 4 : 3        // gausime 3, nes 7 nelygu 5
7==5+2 ? 4 : 3      // gausime 4, nes 7 lygu 5+2
5>3 ? a : b         // gausime kintamojo a reikšmę, nes 5 daugiau už 3
a>b ? a : b         // gausime kintamojo a arba b reikšmę, kuri didesnė
```

```
        // Programos su sąlygos operatoriumi pavyzdys
#include <iostream>
using namespace std;

int main ()
{ int  a,b,c;
  a=2;
  b=7;
  c = (a>b) ? a : b;        // c įgis reikšmę  7
  cout << c;
  return 0;
}
```

```
7
```

4.8. Kablelio operatorius (,)

Kablelio operatoriumi (,) atskiriamos dvi arba daugiau išraiškų, kur įprastai rašoma tik viena. Kai reikšmei apskaičiuoti naudojamas išraiškų rinkinys, tai, apskaičiuojant jas iš kairės į dešinę, galutinę reikšmę duoda toliausiai dešinėje stovinti išraiška. Pvz.:

```
a = (b=3, b+2);
```

Vykdant šį sakinį, visų pirma kintamajam b priskiriama reikšmė 3, o po to išraiškos b+2 rezultatas priskiriamas kintamajam a. Taigi, įvykdžius šį sakinį a reikšmė bus lygi 5, o b reikšmė bus lygi 3.

4.9. Bitų operatoriai (& , | , ^ , ~ , << , >>)

Bitų operatoriais nurodomi veiksmai, kuriuos reikia atlikti su kintamaisiais, imant dvejetainį jų reikšmių pavidalą, kuriuo jie yra saugomi atmintyje.

<table>
<tr><th>Operatorius</th><th>Ekvivalentas kitose kalbose</th><th>Paaiškinimai</th></tr>
<tr><td>&</td><td>AND</td><td>Loginė operacija IR su atitinkamais operandų bitais.</td></tr>
<tr><td>|</td><td>OR</td><td>Loginė operacija ARBA su atitinkamais operandų bitais.</td></tr>
<tr><td>^</td><td>XOR</td><td>Loginė operacija griežtasis-ARBA su atitinkamais operandų bitais.</td></tr>
<tr><td>~</td><td>NOT</td><td>Loginė operacija NE (inversija) su kiekvienu operando bitu. Šis operatorius turi tik vieną operandą.</td></tr>
<tr><td><<</td><td>SHL</td><td>Visų operando bitų postūmis į kairę (Shift Left).</td></tr>
<tr><td>>></td><td>SHR</td><td>Visų operando bitų postūmis į dešinę (Shift Right).</td></tr>
</table>

4.10. Aiškiai nurodomi duomens tipo keitimo operatoriai

Tipo keitimo operatoriai įgalina keisti duomens esamą tipą kitokiu. C++ kalboje tam yra keletas būdų. Paprasčiausias iš jų yra paveldėtas iš C kalbos, prieš išraišką skliaustuose nurodant tipą, į kokį turi būti pertvarkytas išraiškos rezultatas:

```
int  i;
float  f = 3.14;
i = (int) f;
```

Šiame pavyzdyje float tipo kintamojo f reikšmė 3.14 pertvarkoma į int tipo dydį 3 ir priskiriama sveikajam kintamajam i. Kitas būdas padaryti tą patį C++ kalboje yra naudojant funkcijas. Po tipą atitinkančios funkcijos vardo skliaustuose nurodoma išraiška, kurios rezultatą norime pertvarkyti į reikiamą tipą:

```
i = int ( f );
```

4.11. Operatorius *sizeof()*

Šis operatorius turi vieną parametrą, kuris gali būti tipą nurodantis bazinis žodis arba kintamasis, ir gražina skaičių, kiek baitų atmintyje užima nurodyto tipo reikšmė:

```
a = sizeof (char);
```

Šiuo sakiniu kintamajam `a` bus priskirtas `1`, nes `char` tipo kintamajam atmintyje skiriamas tik vienas baitas. `sizeof` gražinamas dydis yra konstanta, kuri visada būna apibrėžta prieš vykdant programą.

4.12. Kiti operatoriai

Vėliau šioje mokymo priemonėje sutiksime daugiau operatorių, susijusių su rodyklėmis (*pointer*) arba objektinio programavimo specifika. Jie nagrinėjami atitinkamuose skyriuose.

4.13. Operatorių pirmumo lygiai

Rašant sudėtingas, daug operacijų turinčias išraiškas, gali iškilti abejonių, kurios operacijos atliekamos pirma, o kurios vėliau. Pavyzdžiui, išraiška

```
a = 5 + 7 % 2;
```

gali kelti abejones, ką gi iš tikro gausime:

```
a = 5 + (7 % 2);      // a reikšmė bus 6
a = (5 + 7) % 2;      // a reikšmė bus 0
```

Teisingas atsakymas yra 6, t.y. pirmos iš šių dviejų išraiškų rezultatas.

Yra nustatyti ne tik aritmetinių, bet ir kiekvieno C++ kalboje sutinkamo operatoriaus pirmumo lygiai (prioritetai). Operatorių pirmumo lygiai, pradedant aukščiausiu ir baigiant žemiausiu, yra šie:

Pirmumo lygis	Operatorius	Paaiškinimai	Atlikimo išraiškoje kryptis
1	:: (du dvitaškiai)	Priklausymo sričiai (*scope*) operatorius	Iš kairės į dešinę
2	() [] . -> ++ -- dynamic_cast static_cast reinterpret_cast const_cast typeid	Rašomi po vardo (*postfix*)	Iš kairės į dešinę
3	++ -- ~ ! sizeof new delete	Vieno operando, rašomi prieš vardą (*prefix*)	Iš dešinės į kairę
	* &	Netiesioginiai ir nuorodos (*pointers*)	
	+ -	Operando ženklo operatoriai	

<table>
<tr><td>4</td><td>(type)</td><td>Aiškus tipo keitimo</td><td>Iš dešinės į kairę</td></tr>
<tr><td>5</td><td>.* ->*</td><td>Rodyklės į narį</td><td>Iš kairės į dešinę</td></tr>
<tr><td>6</td><td>* / %</td><td>Daugybos, dalybos</td><td>Iš kairės į dešinę</td></tr>
<tr><td>7</td><td>+ -</td><td>Sudėties, atimties</td><td>Iš kairės į dešinę</td></tr>
<tr><td>8</td><td><< >></td><td>Postūmio</td><td>Iš kairės į dešinę</td></tr>
<tr><td>9</td><td>< > <= >=</td><td>Palyginimo</td><td>Iš kairės į dešinę</td></tr>
<tr><td>10</td><td>== !=</td><td>Palyginimo</td><td>Iš kairės į dešinę</td></tr>
<tr><td>11</td><td>&</td><td>Loginio IR su bitais</td><td>Iš kairės į dešinę</td></tr>
<tr><td>12</td><td>^</td><td>Loginio griežtojo-ARBA su bitais</td><td>Iš kairės į dešinę</td></tr>
<tr><td>13</td><td>|</td><td>Loginio ARBA su bitais</td><td>Iš kairės į dešinę</td></tr>
<tr><td>14</td><td>&&</td><td>Loginio IR</td><td>Iš kairės į dešinę</td></tr>
<tr><td>15</td><td>||</td><td>Loginio ARBA</td><td>Iš kairės į dešinę</td></tr>
<tr><td>16</td><td>?:</td><td>Sąlygos</td><td>Iš dešinės į kairę</td></tr>
<tr><td>17</td><td>= *= /= %= += -=
>>= <<= &= ^= !=</td><td>Priskyrimo</td><td>Iš dešinės į kairę</td></tr>
<tr><td>18</td><td>,</td><td>Kablelio</td><td>Iš kairės į dešinę</td></tr>
</table>

Šios lentelės stulpelyje “Atlikimo išraiškoje kryptis” nurodoma tokio paties pirmumo lygio operatorių atlikimo tvarka, jei išraiškoje paeiliui jų yra keletas.

Taigi, rašydami sudėtingas išraiškas ir nebūdami tikri dėl operatorių pirmumo lygio, dėkime skliaustus. Tokias išraiškas yra lengviau skaityti.

5. Pagrindinis duomenų įvedimas/išvedimas

Ankstesniuose skyriuose programų pavyzdžiuose naudojome labai nedaug žmogaus ir programos sąveikos priemonių. Priemonės rezultatams išvesti į monitoriaus ekraną ir duomenims įvesti klaviatūra yra laikomos standartinėse bibliotekose.

C++ kalboje vartojama srauto (*stream*) sąvoka, kalbant apie nuoseklios kreipties įrenginius, kaip monitorius ar klaviatūra. Srautas yra simbolių seka, prie kurios programa gali prijungti arba iš kurios gali paimti simbolius. Todėl mums nebūtina žinoti dalykų, susijusių su įrenginių fiziniais ypatumais; mums tereikia žinoti, kad jais simboliai priimami arba perduodami nuosekliai.

Standartinėje C++ bibliotekoje yra failas <iostream>, kuriame laikomos duomenų srauto įvedimo ir išvedimo (*input output stream*) priemonės.

5.1. Standartinis išvedimas (*cout*)

Įprastu atveju programa išveda į monitoriaus ekraną standartinį simbolių srautą, kuris C++ kalboje vadinamas cout.

cout yra naudojamas drauge su siuntimo operatoriumi << (du ženklai „mažiau negu“).

```
cout << "Isveda teksta";    // ekrane bus: Isveda teksta
cout << 120;                //             120
cout << x;                  //             kintamojo x reikšmė
```

Operatorius << prijungia prie srauto nurodytus duomenis. Aukščiau pateiktame pavyzdyje prie standartinio išvedimo srauto cout bus prijungta eilutė "Isveda teksta", skaičius 120 ir kintamojo x reikšmė.

Išvedant simbolių eilutes, nepamirškime jų apkabinti dvigubomis kabutėmis.
Pvz., šie du sakiniai duos skirtingą rezultatą:

```
cout << "Labas";      // ekrane bus žodis Labas
cout << Labas;        // ekrane bus kintamojo Labas reikšmė
```

Viename išvedimo sakinyje operatorius << gali būti naudojamas keletą kartų:

```
cout << "Tai yra " << "C++ kalbos" << " isvedimo sakinys.";
```

Šiuo sakiniu į monitoriaus ekraną bus išvestas tekstas: Tai yra C++ kalbos isvedimo sakinys.

Imkime dar vieną pavyzdį, kuriuo į ekraną išvedama eilučių ir kintamųjų reikšmių kombinacija arba kelių kintamųjų reikšmės:

```
cout << "Labas. Man " << amzius << " metai, "
     << "o mano ugis yra " << ugis << " cm.";
```

Tarkime, kad kintamojo amzius reikšmė yra 24, o kintamojo ugis − 195. Tuomet šiuo sakiniu bus išvesta: Labas. Man 24 metai, o mano ugis yra 195 cm.

Atkreipkime dėmesį, kad cout sakiniu išvedant duomenis nepereinama į naują eilutę tol, kol aiškiai to nenurodysime. Todėl šiais dviem sakiniais

```
cout << "Tai yra pirmas sakinys.";
cout << "Tai jau kitas sakinys.";
```

duomenys bus išvedami į vieną eilutę ir atrodys taip:

Tai yra pirmas sakinys. Tai jau kitas sakinys.

Rezultatai išvedami į kelias eilutes tada, kai cout sakinyje yra aiškūs nurodymai. Šiam tikslui gali būti panaudotas specialus perėjimo į naują eilutę simbolis \n. Pvz., sakiniais

```
cout << "Pirma teksto eilute.\n ";
cout << "Antra eilute.\nTrecia eilute.";
```

gausime tokį rezultatą ekrane:

Pirma teksto eilute.
Antra eilute.
Trecia eilute.

Pereiti į naują eilutę galima ir panaudojus manipuliatorių endl. Pvz.:

```
cout << "Pirmas sakinys." << endl;
cout << "Antras sakinys." << endl;
```

Šie du cout sakiniai rezultatus išves į dvi atskiras eilutes:

Pirmas sakinys.
Antras sakinys.

endl manipuliatoriaus ir specialaus perėjimo į naują eilutę simbolio \n skirtumas pasireiškia srautuose, naudojančiuose išvedimo buferius: endl išvalo buferį. Nežiūrint to, \n ir endl gali būti naudojami nedarant tarp jų skirtumo.

Išvedamų rezultatų formatui nustatyti naudojami standartiniame <iomanip> faile saugomi manipuliatoriai. Štai du iš jų:

setw(m),   m – lauko plotis reikšmei išvesti;
setprecision(n), n – skaitmenų skaičius išvedamoje reikšmėje.

Jei setw nurodyto lauko pločio skaičiui išvesti nepakanka, šis manipuliatorius ignoruojamas. Manipuliatorius setw galioja tik artimiausiai išvedamai reikšmei, o setprecision − iki naujo nurodymo.

Žemiau pateiktoje programoje iliustruojamas manipuliatorių setw ir setprecision veikimas:

Programa	Rezultatai
// manipuliatorių setw *ir* setprecision *naudojimas*	
#include <iostream> *// iš čia ima* cout	
#include <iomanip> *// iš čia ima* setw *ir* setprecision	
using namespace std;	
int main()	
{ float A = 378.236;	
cout << "0. A=" << A << endl;	0. A=378.236
cout << "1. A=" << setw(9) << A << endl;	1. A= 378.236
cout << "2. A=" << setprecision(2)<< A << endl;	2. A=3.8e+02
cout << "3. A=" << setw(10) << setprecision(4) << A << endl;	3. A= 378.2
A = 12.345678;	
cout << "4. A=" << A << endl;	4. A=12.35
return 0;	
}	

5.2. Standartinis įvedimas (*cin*)

Klaviatūra yra dažniausiai naudojamas įrenginys simboliams įvesti. Tokiam įvedimui C++ kalboje yra skirtas standartinis srautas cin.

cin yra naudojamas drauge su priėmimo operatoriumi >> (du ženklai „daugiau negu“). Jis rašomas prieš kintamąjį, kuris įsimena iš standartinio įvedimo srauto paimtus duomenis. Pvz.:

```
int  amzius;
cin >> amzius;
```

Čia pirmuoju sakiniu skelbiamas sveikojo tipo int kintamasis amzius, o sutikusi cin sakinį programa sustos ir lauks, kol klaviatūra bus įvesta kintamojo amzius reikšmė.

Klaviatūra įvesti duomenys priimami tik po to, kai paspaudžiamas klavišas *enter* (kitur jis vadinamas *return* klavišu). Netgi tada, kai reikia įvesti tik vieną simbolį, dar būtina paspausti *enter* klavišą.

Visada būtina atsižvelgti į kintamojo tipą, kuriam cin sakiniu įvedama reikšmė. Jei yra sveikojo tipo kintamasis, būtina įvesti sveikąjį skaičių, jei yra simbolio kintamasis - įvedamas simbolis, jei simbolių eilutės – įvedama simbolių eilutė.

```
// įvedimo/išvedimo pavyzdys
#include <iostream>
using namespace std;

int main ()
{int  i;
  cout << "Iveskite sveikaji skaiciu: ";
  cin >> i;
  cout << "Ivestas skaicius yra " << i;
  cout << ", dvigubai didesnis yra " << i*2 << ".\n";
  return 0;
}
```

```
Iveskite sveikaji
skaiciu: 702
Ivestas skaicius yra
702, dvigubai didesnis
yra 1404.
```

Vienu cin sakiniu galima įvesti reikšmes daugiau kaip vienam kintamajam.

```
cin >> a >> b;
```

Tas pats būtų jei rašytume

```
cin >> a;
cin >> b;
```

Šiais abiem rašymo atvejais reikia įvesti dvi reikšmes: pirmą - kintamajam a, antrą - kintamajam b. Įvedamos reikšmės turi būti atskirtos vienu iš būdų: tarpo simboliu, tabuliacijos simboliu (klavišas *tab*) arba perėjimu į naują eilutę (klavišas *enter*).

5.3. *cin* ir simbolių eilutės

Eilučių tipo (string) kintamųjų, kaip ir pagrindinių tipų kintamųjų, reikšmėms įvesti galima naudoti standartinį įvedimo srautą cin ir priėmimo operatorių >>.

```
string  eilkint;
cin >> eilkint;
```

Tačiau būtina atkreipti dėmesį į tai, kad šiuo atveju įvedimas nutrūksta sutikus tarpo simbolį. Taigi, naudojant cin, galima įvesti tik vieną žodį. Iš keleto žodžių susidedančio sakinio įvesti negalėsime.

Ištisas teksto eilutes įvesti galima funkcija getline, kuri naudoja standartinį įvedimo srautą cin:

// *cin naudojimas simbolių eilutėms įvesti*	
#include <iostream>	
#include <string>	
using namespace std;	
int main ()	
{ string eilut;	
cout << "Koks jusu vardas ir pavarde? ";	Koks jusu vardas ir pavarde?
getline (cin, eilut);	Jonas Pavardenis
cout << "Labas " << eilut << ".\n";	Labas Jonas Pavardenis.
cout << "Kokiu menu domites? ";	Kokiu menu domites?
getline (cin, eilut);	Muzika
cout << "As irgi domiuosi " << eilut << "!\n";	As irgi domiuosi Muzika!
return 0;	
}	

Pastebėkime, kad abiejuose kreipiniuose į funkciją getline naudojamas tas pats eilutės vardas eilut. Antro kreipinio metu anksčiau įvesta eilut reikšmė keičiama naujai įvedama.

5.4. *stringstream* funkcija

C++ kalboje yra standartinės bibliotekos failas <sstream>, kuriame laikoma standartinė funkcija stringstream. Ši funkcija įgalina string tipo kintamąjį traktuoti kaip simbolių srautą. Tokiu būdu mes galime naudoti eilutės kintamąjį įvestiems arba išvedamiems simboliams pasidėti. Tai ypač naudinga, kai eilutes turinį reikia transformuoti į skaitinę reikšmę ir atvirkščiai. Pvz., norint iš eilutės imti simbolius ir iš jų gauti sveikojo tipo reikšmę, reikėtų rašyti:

```
string  eilut ("1204");
int  a;
stringstream(eilut) >> a;
```

Čia turime paskelbtą string tipo kintamąjį eilut, kuriam suteikta pradinė reikšmė "1204", ir int tipo kintamąjį a. Toliau, stringstream funkcija, traktuojanti eilut reikšmę kaip simbolių srautą, ir priėmimo operatorius >> transformuoja simbolių eilutę į int tipo pavidalą ir perduoda kintamajam a. Taip kintamasis a įgyja skaitinę reikšmę 1024.

Programos su srtingstream pavyzdys:

```
    // stringstream naudojimo pavyzdys
#include <iostream>
#include <string>
#include <sstream>          // iš čia ima stringstream
 using namespace std;

int main ()
 { string  eilut
   float  kaina=0;
   int  kiekis=0;
   cout << "Iveskite kaina: ";
   getline (cin,eilut);
   stringstream(eilut) >> kaina;
   cout << "Iveskite kieki: ";
   getline (cin,eilut);
   stringstream(eilut) >> kiekis;
   cout << "Kaina is viso: " << kaina*kiekis << endl;
   return 0;
 }
```

```
Iveskite kaina: 22.25

Iveskite kieki: 7

Kaina is viso: 155.75
```

Šiame pavyzdyje standartiniu įvedimo būdu skaitinės kintamųjų kaina ir kiekis reikšmės gaunamos netiesiogiai. getline funkcija, naudodama standartinį įvedimo srautą cin, įvestą eilutę patalpina kintamajame eilut. Po to stringstream funkcija ir operatorius >> transformuoja įvestus simbolius į int tipo reikšmes (kaina, kiekis).

Toks netiesioginis sveikųjų skaičių reikšmių įvedimas int tipo kintamiesiems suteikia didesnes kontrolės galimybes. Įvedus eilutę, galima patikrinti, ar joje yra skaitmenys, ar nėra neleistinų simbolių.

5.5. Duomenų įvedimo/išvedimo funkcijos *printf* ir *scanf*

C++ kalboje yra įvairios duomenų įvedimo/išvedimo standartinės priemonės. Jos laikomos ne vien jau minėtose bibliotekose <iostream>, <iomanip>, <sstream>, bet ir bibliotekose <cstdio> (C kalboje ji vadinosi <stdio.h>), <fstream>, <istream>, <ostream>, kt.

Šiame skyriuje susipažinkime su bibliotekos <cstdio> priemonėmis.

Standartiniams simbolių srautams (išvedamiems į ekraną ir įvedamiems klaviatūra) valdyti dar gali būti vartojamos funkcijos printf ir scanf. Kreipinių į jas pavidalas yra:

```
printf ( šablonas , kintamųjų_sąrašas );
scanf  ( šablonas, kintamųjų_adresų_sąrašas );
```

Pastebėkime, kad duomenų išvedimo funkcijoje printf po šablono nurodomi kintamųjų vardai, o įvedimo funkcijoje scanf - kintamųjų adresai. Pavyzdžiui, jei turime kintamąjį vardu a, tai jo adresas atmintyje gaunamas užrašius &a.

Šablonas – tai simbolių eilutės konstanta (ji rašoma tarp dvigubų kabučių) su joje įterptais duomenų formatais ir valdančiaisiais simboliais.

Įvedamiems/išvedamiems duomenims skiriamų laukų struktūra arba jų interpretavimo būdas nurodomas **duomenų formatais**. Tipinė formato struktūra yra tokia:

% [požymis] [lauko_dydis] [.tikslumas] duomenų_tipas

Čia stačiakampiais skliaustais tik nurodoma, kad to formato elemento gali ir nebūti.

Formate dažniausiai naudojami požymiai yra tokie:

– (minusas) - lygiuoti išvedamus duomenis pagal kairįjį lauko kraštą;

+ (pliusas) - nurodyti išvedamo skaičiaus ženklą, net jei jis teigiamas.

Jei požymis nenurodomas, išvedama reikšmė lygiuojama pagal dešinįjį lauko kraštą, o prieš išvedamą teigiamą skaičių pliuso ženklas nededamas.

Formate duomenų_tipai žymimi raidėmis, parodytomis šioje lentelėje:

Duomenų tipo raidė	Duomenų tipas
d arba i	sveikasis skaičius (int);
o	aštuntainis sveikasis skaičius;
u	sveikasis skaičius be ženklo;
x	šešioliktainis sveikasis skaičius;
c	simbolis (char);
f	slankaus kablelio skaičius (float);
s	simbolių eilutė (string);
e	rodyklinės formos skaičius (float);
p	rodyklė (*pointer*)

Išvedimo funkcijoje printf valdantieji simboliai gali būti įterpiami bet kurioje šablono vietoje. Primename, kad valdančiojo simbolio užrašymo sintaksė yra \simbolis, pvz., \n , \t ir kt. Jeigu norime šablone turėti dvigubą kabutę (") arba atbulai pasvirusį brūkšnį (\), jie turi būti rašomi atitinkamai \" ir \\. To priežastis yra ta, kad kabute (") nurodoma eilutės pradžia ir pabaiga, o simboliu \ - valdantysis simbolis.

Programa, iliustruojanti funkcijos printf veikimą:

```
 #include <cstdio>
using namespace std;
 int  main()
 { int  a = 1023;
   long int  b = 2222;
   short int  c = 123;
```

```
    unsigned int  d = 1234;
    char e = 'X';
    float  f = 3.14159;
    double  g = 3.1415926535898;
   printf("1. a=%d\n", a);     // dešimtainis sveikasis          1. a=1023
   printf("2. a=%o\n", a);     // aštuntainis sveikasis          2. a=1777
   printf("3. a=%x\n", a);     // šešioliktainis sveikasis       3. a=3ff
   printf("4. b=%ld\n", b);        // dešimtainis ilgas          4. b=2222
   printf("5. d=%u\n", d);         // sveikasis be ženklo        5. d=1234
   printf("6. e=%c\n", e);         // simbolis                   6. e=X
   printf("7. f=%f\n", f);         // realusis skaičius          7. f=3.141590
   printf("8. f=%e\n", c);         // realusis su eile           8. f=3.141590e+000
   printf("\n");                   // tik pereiti į kitą eilutę
   printf("9. a=%7d\n", a);            // 7 pozicijų laukas      9. a=   1023
   printf("10. a=%–7d\n", a);        // - veda nuo kaires        10. a=1023
    c = 5;  d = 8;
   printf("11. a=%*d\n",c,a);          // lauko plotis 5 poz.    11. a= 1023
   printf("12. a=%*d\n",d,a);          // lauko plotis 8 poz.    12. a=    1023
   printf("13. f=%f\n", f);            // realusis skaičius      13. f=3.141590
   printf("14. f=%10f\n", f);          // 10 poz. realiam        14. f=  3.141590
   printf("15. f=%10.3f\n", f);        // trupmenai 3 p.         15. f=     3.142
   printf("16. g=%11.8f\n", g);        // trupmenai 8 p.         16. g= 3.14159265
   printf("17. g=%+11.8f\n", g);   // su +                       17. g=+3.14159265
   printf("18. a=%d g=%5.2f\n", a, g);                           18. a=1023 g= 3.14
   return 0;
 }
```

Įvedimo funkcijos scanf šablone dažniausiai nurodomi tik įvedamų duomenų formatai. Pačiuose formatuose nurodomas tik duomenų tipas. Pvz.:

```
int a;
float b;
scanf("%d %f", &a, &b);
```

Šiuo atveju klaviatūra įvedami nurodyto tipo skaičiai turi būti atskirti tarpu, tabuliacijos simboliu (‘\t‘) arba perėjimu į kitą eilutę (‘\n‘). Pvz., įvedus 45 8.56, kintamasis a įgis reikšmę 45, o kintamasis b – 8.56.

Jei funkcijos scanf šablone prieš duomenų formatus bus nurodyti kažkokie simboliai, tai juos taip pat reikės įvesti. Pvz.:

```
scanf("a= %d f= %f", &a, &b);
```

Čia klaviatūra įvedami duomenys turės atrodyti taip: a= 45 f= 8.56 .

5.6. Darbas su duomenų failais

Failai (bylos, rinkmenos) - tai išoriniais kompiuterio įrenginiais įvedami/išvedami duomenų rinkiniai (tekstai, skaičiai, vaizdai, garsai). Pvz., duomenys į kompiuterio atmintį gali būti įvedami klaviatūra, iš disko, kompiuterių tinklo ryšio kanalu, mikrofonu, vaizdo kamera, kt. Iš kompiuterio atminties duomenys (skaičiavimo rezultatai) dažniausiai išspausdinami, išvedami į ekraną, diską, garsiakalbį, perduodami kompiuterių tinklo ryšio kanalu.

Failai egzistuoja atskirai nuo programų. Sukurti failai saugomi įvairiose laikmenose (diskuose, *flash* atminties kištukuose, kt.). Norint skaityti anksčiau sukurtą failą, kompiuterio programoje turi būti nurodyta, kokiu įrenginiu tai norima daryti. Kuomet failas yra kuriamas (rašomas), kompiuterio programoje turi būti nurodyta, kuriuo įrenginiu (atitinkamai - laikmenoje) tai norima daryti. Taigi, kompiuterio programose turi būti kintamieji, kuriais būtų pažymėti (atstovaujami) failai. C++ kalboje tokių kintamųjų tipas dažniausiai nurodomas baziniu žodžiu ifstream, ofstream. Kintamieji-failai dažniausiai aprašomi taip:

```
ifstream a, b;         // kai norima skaityti failą (input);
ofstream x, y, z;      // kai norima rašyti į failą (output).
```

Prieš naudojant šitokio tipo kintamuosius, programoje turi būti nuoroda preprocesoriui

```
#include <fstream>,
```

kad jis į programą įtrauktų standartinę biblioteką <fstream>, kurioje laikomos duomenų failuose įvedimo/išvedimo priemonės.

Programoje aprašytas kintamasis-failas turi būti susiejamas su konkrečiu fiziniu failu, t.y. jis turi nurodyti įrenginį, kur tas failas yra ar bus kuriamas, ir failo vardą. Šie duomenys kintamajam-failui suteikiami inicializuojant tokį kintamąjį

```
#include <fstream>
ofstream failas1 ("c:\\rezultatai.txt");
```

arba atidarant failą specialia open funkcija

```
#include <fstream>
ofstream failas1;
failas1.open ("c:\\rezultatai.txt");
```

Failo ieškojimo kelio nuorodoje reikia rašyti du atbulai pasvirusius brūkšnius (\\), o ne vieną, kaip kitur yra įprasta, nes specialieji simboliai pradedami \, pvz., \n, \t, \? .

Baigus darbą su failu, jis uždaromas close funkcija:

```
failas1.close();
```

Pastebėkime, kad tarp kintamojo-failo vardo ir funkcijos vardo dedamas taškas. Taip nurodoma, kurį gi failą norime atidaryti ar uždaryti, nes programa gali dirbti iš karto su keletu failų. Vėliau, kai susipažinsime su objektinio programavimo pagrindais, matysime, kad ifstream, ofstream atitinka klasių vardus, open(), close() − minėtų klasių narių-funkcijų vardus, o failas1 − mūsų paskelbto objekto vardą.

Pateikiame duomenų išvedimo į failą ištisą programą:

```
    // pagrindinės išvedimo į failą operacijos
#include <iostream>
#include <fstream>
using namespace std;

int main ()
{ ofstream  isifa;
  isifa.open ("c:\\rezult.txt");
  isifa << "Sis tekstas rasomas i faila.\n";
  isifa.close();
  return 0;
}
```

```
[failas c:\rezult.txt]
Sis tekstas rasomas i faila.
```

Čia mūsų paskelbtas isifa atitinka mums jau žinomą standartinį cout, išvedantį duomenis į ekraną.

Analogiškai yra ir su duomenų įvedimu iš failo:

```
    // pagrindinės įvedimo iš failo operacijos
#include <iostream>
#include <fstream>
using namespace std;

int main ()
{ ifstream  ivisfa;
  int  a, b;
  ivisfa.open ("c:\\duom.txt");
  ivisfa >> a >> b;                    // įveda iš failo
  cout << a << "+" << b << "=" << a+b << endl;
  ivisfa.close();
  return 0;
}
```

```
[failas c:\duom.txt]
27 482

[ekrane]
27+482=509
```

Šioje programoje ivisfa atitinka standartinį cin, įvedantį duomenis klaviatūra. Čia iš failo įvestų skaičių suma išvedama į ekraną standartiniu cout.

Matome, kad prieš duomenų įvedimą ar išvedimą failai turi būti atidaromi, o baigus darbą - uždaromi. Failo atidarymo prasmė yra ta, kad, pvz., išvedant duomenis

į diską, diske visų pirma turi būti parengta sritis kuriamam failui (kataloge įrašomas failo vardas, paskiriama jam vieta diske), programai atmintyje papildomai paskiriami laukai, kurie vadinami įvedimo/išvedimo buferiais. Uždarant failą, pvz., diske užrašomas failo pabaigos požymis (jei failas buvo kuriamas), išlaisvinama įvedimo/išvedimo buferiams skirta atminties sritis.

Panagrinėkime dar vieną programą, kuri prašo nurodyti, iš kurio failo norėsime vesti duomenis ir į kurį failą norėsime rašyti rezultatus:

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

int main ()
{ ifstream  aa;
  ofstream  bb;
  char  s [50];
  int  a, b;
  cout <<"Is kurio failo vesite duomenis?\n";
  cin >> s;
  aa.open (s);
  cout <<"I kuri faila isvesite rezultatus?\n";
  cin >> s;
  bb.open(s);
  aa >> a >> b;
  bb << a << "+" << b << "=" << a+b << endl;
  aa.close();
  bb.close();
  return 0;
}
```

```
Is kurio failo vesite
duomenis?
c:\duom.txt
I kuri faila isvesite
rezultatus?
C:\rez.txt

            [c:\duom.txt]
27 482
            [c:\rez.txt]
27+482=509
```

Dirbant su failais gali atsitikti, kad norimo failo nepavyksta atidaryti, o įvedant duomenis iš failų pasiekiama failo pabaiga. Šioms situacijoms kontroliuoti yra atitinkamai funkcijos is_open() ir eof(). Funkcijos is_open() reikšmė yra true, jei failas atidarytas, o eof() duoda false, jei dar nepasiekta failo pabaiga. Šių funkcijų naudojimo pavyzdys:

```
        // tekstinio failo skaitymas
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
```

``
int main ()
{ string eilut;
  ifstream manofa;
  manofa.open ("pavyz.txt");
  if (!manofa.is_open())
   { cout << "Failo atidaryti nepavyko"; return 1;
   }
  while (!manofa.eof())
   { getline (manofa, eilut);
     cout << eilut << endl;
   }
  manofa.close();
  return 0;
}
``
	Viena teksto faile eilute.
	Kita teksto faile eilute.

Čia simbolių eilutėms įvesti naudojome geltine() funkciją, tik jos argumentas yra mūsų paskelbtas kintamasis manofa, o ne cin, kaip tą darėme „5.3. *cin* ir simbolių eilutės“ skyriuje.

C++ kalboje yra ir daugiau priemonių duomenims įvesti/išvesti. Pvz., funkcijomis read() ir write() perduodami dvejetainio (binarinio) pavidalo duomenys. Įvedimo/išvedimo metu jokia duomenų transformacija iš dvejetainio pavidalo į simbolinį pavidalą ir atvirkščiai neatliekama. Minėtas funkcijas ir eilę kitų nagrinėsime vėliau, kai susipažinsime su rodyklėmis.

6. Valdantieji sakiniai

Programose sakiniai dažnai nėra vykdomi tokia eilės tvarka, kaip jie yra užrašyti. Jose galimi išsišakojimai, sakinių kartojimai, sprendimų priėmimai. Šiam tikslui yra valdantieji sakiniai, kuriais nurodoma, kada ir ką programa turi daryti esant tam tikroms aplinkybėms.

Prieš pradėdami nagrinėti valdančiuosius sakinius, visų pirma susipažinkime su sudėtinio sakinio arba bloko sąvoka. Blokas – tai sakinių grupė, apkabinta riestiniais skliaustais {}. Jame sakiniai, kaip priimta C++ kalboje, vienas nuo kito skiriami kabliataškiais. Pvz.:

{ sakinys1; sakinys2; sakinys3; }

Daugelyje valdymo sakinių, kaip jų sudėtinė dalis, yra kitas sakinys. Tas kitas sakinys gali būti paprastas (jis baigiamas kabliataškiu) arba sudėtinis (keletas bloką sudarančių sakinių). Tais atvejais, kai norime turėti paprastą sakinį, apkabinti jo riestiniais skliaustais { } nereikia. Tačiau sudėtinio sakinio atveju riestiniai skliaustai { }, kuriais formuojamas blokas, būtini.

6.1. Sąlygos sakinys: *if* ir *else*

Sąlygos sakinio, prasidedančio baziniu žodeliu if, bendrasis pavidalas yra toks:

if (sąlygos_išraiška) sakinys

Jis vykdomas taip. Visų pirma apskaičiuojama sąlygos_išraiška. Jei jos reikšmė yra true, tuomet yra vykdomas nurodytas sakinys. Jei jos reikšmė yra false, tuomet nurodytas sakinys nevykdomas ir programa tęsiama sakiniu, einančiu po sąlygos sakinio. Pvz., programos fragmentas

```
if (x == 100)
   cout << "x yra lygus 100";
```

atspausdins x yra lygus 100 , jei kintamojo x reikšmė iš tikro bus lygi 100.

Taip pat yra galimybė nurodyti, ką reikėtų daryti, kai sąlygos_išraiškos rezultatas yra false. Tam gali būti naudojamas bazinis žodis else, kuriuo išplečiamas if sakinys:

if (sąlygos_išraiška) sakinys1 else sakinys2

Pvz., sakiniais

```
if (x == 100)
   cout << "x yra lygus 100";
else
   cout << "x nera lygus 100";
```

į ekraną bus išvesta x yra lygus 100 , jei x reikšmė yra 100. Kitais atvejais bus išvesta x nera lygus 100.

Po else gali būti rašomas naujas if sakinys. Toliau einantis pavyzdys rodo, kaip bus elgiamasi, kai x yra daugiau, lygus ir mažiau už 0:

```
if (x > 0)
   cout << "x yra teigiamas";
else  if (x < 0)
         cout << "x yra neigiamas";
      else
         cout << "x yra lygus 0";
```

Prisiminkime, kad šakoje norint atlikti kelis sakinius, juos būtina apskliausti riestiniais skliaustais {}.

6.2. Ciklai (*while, do-while, for*)

Ciklų paskirtis yra kartoti sakinį (paprastą arba sudėtinį) tam tikrą kiekį kartų arba kol nebus įvykdyta tam tikra sąlyga.

while ciklas.

Šio ciklo bendrasis pavidalas yra

while (sąlygos_išraiška) sakinys

Jame nurodytas sakinys turi būti kartojamas tol, kol prieš tai apskaičiuota sąlygos_išraiška duoda rezultatą true.

Žemiau pateikiama programa while ciklui pailiustruoti:

```
   // programa su while ciklu
#include <iostream>
using namespace std;

int main ()
{ int  n;
  cout << "Iveskite pradini skaiciu: ";
  cin >> n;

  while (n>0)
    { cout << n << ", ";
      --n;          // n  reikšmė mažinama vienetu
    }
  cout << "\n Pirmyn!";
  return 0;
}
```

```
Iveskite pradini skaiciu: 8

8, 7, 6, 5, 4, 3, 2, 1,

 Pirmyn!
```

Šios programos vykdymo pradžioje vartotojui į ekraną išvedamas priminimas Iveskite pradini skaiciu. Įvedus skaičių, while ciklo blokas (sudėtinis sakinys) kartojamas tol, kol n>0. Kai tik n pasidaro lygus 0, ciklas nutraukiamas ir į ekraną išvedama Pirmyn!

Atkreipkime dėmesį į tai, kad while ciklo bloke turi būti sakinys, turintis įtaką tam, kad ciklo sąlygos_išraiškos rezultatas taptų false. Kitokiu atveju turėtume nesibaigiantį ciklą. Todėl mūsų turėtoje programoje yra sakinys --n;

Pastebėkime, kad while ciklas gali būti neatliktas nė karto, pvz., jei įvestume neigiamą n reikšmę.

do-while ciklas.

Šio ciklo bendrasis pavidalas yra toks:

do sakinys while (sąlygos_išraiška);

Jis vydomas taip pat, kaip ir while ciklas. Skirtumas yra tik toks, kad čia sąlygos_išraiška apskaičiuojama po to, kai įvykdomas sakinys. Taip garantuojama, kad sakinys bus įvykdytas bent vieną kartą. Pvz., žemiau pateikiama programa prašys vesti skaičius tol, kol neįvesime 0.

```
// programa su do-while ciklu
#include <iostream>
using namespace std;

int main ()
{ unsigned long  n;
  cout << "Darbui baigti iveskite 0. \n";
  do
    { cout << "Iveskite skaiciu: ";
      cin >> n;
      cout << "Jus ivedete: " << n << "\n";
    }
  while (n != 0);
  return 0;
}
```

```
Darbui baigti iveskite 0.

Iveskite skaiciu: 12345
Jus ivedete: 12345
Iveskite skaiciu: 160277
Jus ivedete: 160277
Iveskite skaiciu: 0
Jus ivedete: 0
```

do-while ciklų nutraukimo sąlyga suformuojama ciklo bloke. Mūsų pavyzdyje tai įvestas skaičius 0.

for ciklas.

Šio ciklo bendrasis pavidalas yra:

for (inicializacija; sąlyga; reikšmės_keitimas) sakinys

Jame sakinys kartojamas tol, kol sąlyga duoda reikšmę true. Kaip matome, jame dar yra inicializacijos sakinys ir reikšmės_keitimo sakinys. Šitoks ciklas specialiai yra sukurtas pakartotinam veiksmų vykdymui vadovaujantis skaitikliu. Skaitiklis visų

pirma yra inicializuojamas (jam suteikiama pradinė reikšmė), o vėliau jo reikšmė keičiama nurodytu dydžiu kiekvienos iteracijos metu.

for ciklo veiksmų sekos punktai yra tokie:

1. Atliekama inicializacija. Paprastai tai pradinės reikšmės suteikimas skaitiklio kintamajam. Šis veiksmas atliekamas tik vieną kartą.

2. Tikrinama sąlyga, ar skaitiklio kintamojo reikšmė nėra didesnė (mažesnė) už nurodytą dydį. Jei gaunama reikšmė true, ciklo sakinys yra vykdomas. Priešingu atveju ciklas baigiamas ir sakinys nebevykdomas.

3. Vykdomas ciklo sakinys. Tai gali būti paprastas arba sudėtinis sakinys (blokas).

4. Atlikus sakinį, vykdomas reikšmės_keitimo sakinys. Dažniausia keičiama skaitiklio kintamojo reikšmė (didinama arba mažinama) ir grįžtama į ciklo veiksmų sekos 2 punktą.

Žemiau pateikiama programa su for ciklu, kuriame kintamojo-skaitiklio n reikšmė yra mažinama:

```
  // programa su for ciklu
#include <iostream>
using namespace std;
int main ()
{ for ( int n=10; n>0; n-- )
      cout << n << ", ";

  cout << "\n Pirmyn!";
  return 0;
}
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
 Pirmyn!
```

for cikle inicializacijos ir reikšmės_keitimo sakinių gali ir nebūti. Jie gali būti tušti, tačiau laukus skiriantys kabliataškiai (;) turi būti rašomi. Pvz., gali būti rašoma for (; n<10 ;), jei nereikia inicializacijos ir reikšmės_keitimo sakinių, arba for (; n<10; n++), jei nereikia tik inicializacijos sakinio. Galbūt inicializacija buvo atlikta dar iki for ciklo.

for cikle inicializacija ir reikšmės_keitimas gali susidėti iš keleto išraiškų. Išraiškos skiriamos kableliu (,). Pvz., tarkime, cikle norime inicializuoti daugiau kaip vieną kintamąjį:

```
for ( n=0, i=100; n!=i; n++, i-- )
  {   // bloko sakiniai …
  }
```

Šis ciklas bus kartojamas 50 kartų, jei n ir i reikšmės nebus keičiamos bloko sakiniais.

6.3. Peršokimo sakiniai (*break, continue, goto, exit()*)

break sakinys.

Cikle sutikus break sakinį, ciklas nutraukiamas, nors jo tęsimo sąlyga ir yra įvykdyta. break gali būti naudojamas begaliniams ciklams nutraukti arba nutraukti ciklams dar jiems normaliai nepasibaigus.

Pavyzdys:

```
// break sakinio naudojimo pavyzdys
#include <iostream>
using namespace std;

int main ()
{ int  n;
  for (n=10; n>0; n--)
    { cout << n << ", ";
      if (n==3)                       // sąlyga ciklui nutraukti
        { cout << "\n Ciklas nutrauktas!";
          break;
        }
    }
  return 0;
}
```

```
10, 9, 8, 7, 6, 5, 4, 3,
 Ciklas nutrauktas!
```

Šioje programoje for ciklas nutraukiamas, kai kintamojo-skaitiklio n reikšmė lygi 3.

continue sakinys.

continue sakinys įgalina nutraukti tik ciklo iteraciją jos nebaigus, peršokti į kitos įteracijos pradžią ir tęsti ciklą, kol jis nesibaigs normaliai.

Pavyzdys:

```
// continue  sakinio naudojimo pavyzdys
#include <iostream>
using namespace std;

int main ()
{ for ( int n=10; n>0; n-- )
    { if (n==5)  continue;
      cout << n << ", ";
    }
  cout << "\n Pirmyn!";
  return 0;
}
```

```
10, 9, 8, 7, 6, 4, 3, 2, 1,
 Pirmyn!
```

Čia for ciklo iteracija, kai n lygu 5, nutraukiama, bet ciklas vykdomas toliau.

goto sakinys.

goto sakiniu besąlygiškai peršokama į kitą programos vietą. Peršokimo vieta nurodoma žyme, kuri kaip argumentas nurodoma goto sakinyje.

Žymės rašomos tik prieš tuos programos sakinius, į kuriuos norėsime peršokti goto sakiniais. Tarp žymės ir sakinio dedamas dvitaškis. goto sakinyje po nurodytos žymės dvitaškis nededamas.

Apskritai, struktūriniame ir objektiniame programavime goto sakiniai nevartotini. Juos naudoja mažiau išprusę programuotojai.

Pavyzdys:

Programa	Rezultatas

// *goto sakinio vartojimo pavyzdys*
#include <iostream>
using namespace std;
 	
int main ()
{ int n=10;
  zym1:
  cout << n << ", ";
  n--;
  if (n>0) goto zym1;
  cout << "\n Pirmyn!";
  return 0;
}
	10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
	Pirmyn!

Šioje programoje ciklas yra suorganizuotas naudojant if ir goto sakinius.

exit() funkcija.

Atkreipkime dėmesį, kad exit() yra gatava funkcija ir laikoma bibliotekoje <cstdlib>.

exit() funkcijos paskirtis yra stabdyti programos darbą, nurodant specifinį stabdymo kodą. Šios funkcijos prototipas (C++ kalboje „prototipas“ – tai funkcijos pirmasis sakinys- antraštė) yra:

```
void exit (int isejimokodas);
```

isejimokodas reikšmę naudoja kai kurios operacinės sistemos, ir ji gali būti naudojama kitose kviečiamose programose. Pagal susitarimą isejimokodas lygus 0 (sakinys exit(0);) reiškia, kad programa baigė darbą normaliai. Kitokios isejimokodas reikšmės rodo, kad programoje buvo klaidų arba gauti nenumatyti rezultatai.

6.4. Variantinis sakinys (*switch*)

Variantinio sakinio switch struktūra yra kažkiek savita. Priklausomai nuo jame esančios išraiškos reikšmės, vykdoma viena iš sakinių grupių, kurių gali būti norimas kiekis. Tai šiek tiek panašu į if ir else if sakinius. switch sakinio pavidalas yra toks:

```
switch (išraiška)
{ case konstanta1:
      sakinių_grupė_1;
      break;
  case konstanta2:
      sakinių_grupė_2;
      break;
  . . .
  default:
      default_sakinių_grupė
}
```

Jis vykdomas taip: visų pirma apskaičiuojama išraiška ir tikrinama, ar gauta reikšmė lygi konstantai1. Jeigu taip, vykdoma sakinių_grupė_1 iki sutinkamas break sakinys. Sutikus break, peršokama į switch sakinio galą.

Jei išraiškos reikšmė nėra lygi konstantai1, tuomet tikrinama, ar ji nėra lygi konstantai2. Jei lygu, tai vykdoma sakinių_grupė_2 iki break ir tuomet peršokama į switch sakinio galą.

Galų gale, jei išraiškos reikšmė nėra lygi nė vienai iš nurodytų konstantų (case konstantų gali būti tiek, kiek mums reikia), vykdomi sakiniai, esantys po žymės default, jei ji yra. default dalis switch sakinyje nebūtina.

Abu žemiau pateikti programų fragmentai duoda tokį patį rezultatą:

<table>
<tr><th>switch pavyzdys</th><th>if-else ekvivalentas</th></tr>
<tr><td><pre>switch (x)
{ case 1:
      cout << "x lygu 1";
      break;
  case 2:
      cout << "x lygu 2";
      break;
  default:
      cout << "x nezinomas";
}</pre></td><td><pre>if (x == 1)
  { cout << "x lygu 1";
  }
else if (x == 2)
        { cout << "x lygu 2";
        }
       else
        { cout << "x nezinomas";
        }</pre></td></tr>
</table>

switch sakinys yra kažkiek neįprastas C++ kalboje, nes jame vartojamos žymės, o ne blokai (sudėtiniai sakiniai). Tai verčia mus naudoti break po kiekvienos

sakinių_grupės, kurią reikia atlikti esant specifinei sąlygai. Praleidus break būtų vykdoma ir kita iš eilės einanti sakinių_grupė, kol nepasiektume switch sakinio galo arba nesutiktume break.

Pvz., jei mes nerašysime break po pirmos sakinių grupės, tai programa automatiškai neperšoks į switch sakinio pabaigą, o vykdys toliau iš eilės einančius sakinius. Tokiu būdu išvengiama bereikalingų riestinių skliaustų {} kiekvieno varianto sakinių grupei. Taip pat tai gali būti naudinga norint vykdyti tą pačią sakinių grupę, esant įvairioms išraiškos reikšmėms. Pvz.:

```
switch (x)
 { case 1:
   case 2:
   case 3:
     cout << "x lygu 1, 2 arba 3";
     break;
   default:
     cout << "x nelygu 1, 2 nei 3";
 }
```

Pastebėkime, kad switch sakinyje vietoje konstantų negalima vartoti kintamųjų, pvz., n, arba reikšmių intervalų, pvz., (1..5). Jei reikalingi skaičiavimai tikrinant reikšmes iš kažkurio intervalo, naudokime if arba else if sakinius.

6.5. Paprastų programų pavyzdžiai

Prieš rašant programą visų pirma reikia sudaryti sprendžiamo uždavinio algoritmą, t.y. planą, kokius veiksmus ir kokia eilės tvarka numatome atlikti. Žemiau pateiktuose pavyzdžiuose, kur veiksmų atlikimo tvarka yra aiški iš uždavinio sąlygos, algoritmo neaprašome. Kai kuriuose iš jų, programavimo niuansams suprasti, algoritmai aiškinami plačiau.

1 pavyzdys.

Išspausdinkime sveikųjų skaičių nuo 1 iki N žingsniu 1 kvadratų, kūbų ir kvadratinių šaknų lentelę. N reikšmę įveskime klaviatūra.

```
 /* Programa skaičių laipsnių lentelei išvesti. Autorius - V.Pavardenis,
    tel. 222555,   data - 2007 06 26 */

#include <iostream>        // iš čia ims cin ir cout
#include <cmath>           // matematinių funkcijų biblioteka. Ims sqrt
#include <cstdlib>         // iš čia ims funkciją  system
using namespace std;
#define bruksnys "\n----------------------------------\n"

main ()
{ int  n, j, kvadratas, kubas;
   float  saknis;
   cout << "Iveskite sveikaji skaiciu N: ";
   cin >> n ;              // įvedame skaičių, iki kurio skaičiuosime laipsnius
                           // toliau išvedame lentelės antraštę
   cout << bruksnys;
   cout << "Skaic.  Kvadr.  Kubas   Saknis  \n";
   cout << bruksnys;
                           // toliau skaičiuojame ir išvedame rezultatus
   for ( j = 1; j <= n; j++ )
    { kvadratas = j*j;
      kubas = kvadratas*j ;
      saknis = sqrt(j) ;          // sqrt – kvadr. šaknies traukimo funkcija
      cout << j << "\t" << kvadratas << "\t"
            << kubas << "\t" << saknis << endl;
    }
   cout << bruksnys;
   system ("PAUSE");              // stabdo programą rezultatams pažiūrėti
   return 0;
}
```

2 pavyzdys.

Apskaičiuoti sumą $S = \sum_{k=1}^{N} \frac{1}{k} = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{N}$,

ir faktorialą $N! = 1 * 2 * 3 * \ldots * N$,

kur N > 0. N reikšmę įveskime klaviatūra.

```
 /* programa skaičiuoja sumą ir faktorialą;
  autorius - V.Pavardenis;   tel.  2144555;  data  2007 06 26 */

#include <iostream>
#include <cstdlib>              // iš čia ims funkciją  system
using namespace std;

main ()
{ int  k, n;
   float  suma;
   long  fakt;
   cout << "Iveskite sveikaji skaiciu N: ";
   cin >> n;
      // toliau parengiami kintamieji suma ir fakt sumai ir faktorialui kaupti
   suma = 0 ;
   fakt = 1 ;
   for (k=1; k<=n; k++)
     { suma = suma + 1.0/k ;   // 1.0, kad dalybos rezultatas būtu float tipo
        fakt = fakt * k ;
     }
                                  // toliau išvedami skaičiavimo rezultatai
   cout << "Kai N = " << n << ", suma = " << suma << endl;
   cout << "Faktorialas = " << fakt << endl;
   system ("PAUSE");              // stabdoma programa rezultatams pažiūrėti
   return 0;
}
```

3 pavyzdys.

Apskaičiuoti funkciją

$$F(x) = \begin{cases} x + (x+1)^2 + (x+2)^3, & \text{kai } x < 1, \\ 1 + \frac{1}{x} + \frac{1}{x^2} + \frac{1}{x^3} + \ldots + \frac{1}{x^{10}}, & \text{kai } 1 <= x <= 3, \\ 1 + \sin(x), & \text{kai } x > 3. \end{cases}$$

x reikšmę įveskime klaviatūra.

```
        /* programa išsišakojančios funkcijos reikšmei apskaičiuoti */
#include <iostream>
#include <cmath>          // matematinių funkcijų biblioteka. Ims sin
#include <cstdlib>        // iš čia ims funkciją  system
using namespace std;
main ()
{ float  a, x, f;
  int  k;
  cout << “Iveskite x reikšme: “;
  cin >> x;
  if (x < 1)
    f = x + (x+1)*(x+1) + (x+2)*(x+2)*(x+2);
  if ((1 <= x) && (x <= 3))
   { f = 1;
     a = 1;
     for ( k=1; k<=10; k++ )
      { a = a/x;                 // vardiklyje augs x  laipsnis
        f = f+a;
      }
   }
  if (x > 3)
    f = 1 + sin(x);
                // toliau išvedame skaičiavimo rezultatus
  cout << “Kai  x = “ << x << “, funkcijos reiksme = “
        << f << endl;
  system ("PAUSE");       // stabdoma programa rezultatams pažiūrėti
  return 0;
}
```

4 pavyzdys.

Trigonometrinės funkcijos *cos (x)* , kai x išreiškiamas radianais, reikšmė randama sumuojant tokios matematinės eilutės narius:

$$cos(x) \approx 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots$$

Nežiūrint to, kad C++ kalboje matematinių funkcijų bibliotekoje <cmath> yra *cos(x)* standartinė funkcija, kosinusui apskaičiuoti sudarykime programą patys. x reikšmę įveskime klaviatūra, o kosinusą skaičiuokime tikslumu $\varepsilon = 0.00001$.

Algoritmas.

Matematinės eilutės, kurios narius sumuojant gaunama cos(x) reikšmė, *k*-tasis narys, remiantis prieš buvusiu nariu, randamas taip: $a_k = -a_{k-1} * \frac{x^2}{2k(2k-1)}$, kur

$a_0=1$; $k=1, 2, \ldots$

Taigi, reikia apskaičiuoti sumą $\cos(x) \approx a_0 + a_1 + a_2 + \ldots$

Čia sumuojamų narių kiekis iš anksto nėra žinomas. Skaičiavimai nutraukiami tada, kai tik eilinis (paskutinis) sumos narys absoliutiniu dydžiu tampa mažesnis už pasirinktą ε.

Jeigu įvesta x reikšmė yra didelė, ji redukuojama keletu periodų 2π.

Taip pat palyginimui išveskime kosinuso reikšmę, gautą standartine funkcija.

```
// programa kosinusui apskaičiuoti;  autorius V.Pavardenis;  tel. 2222228
#include <iostream>
#include <cmath>           // matematinių funkcijų biblioteka. Ims fabs, cos
#include <cstdlib>         // iš čia ims funkciją  system
using namespace std;
#define eps 0.00001        // tokiu tikslumu skaičiuosime
#define PI 3.14159

main ()
{ float  a, s, x;
  int  k;
  cout << “Iveskite  x  reiksme: “;
  cin >> x;
  while (fabs(x) > 2*PI)      // redukuojam x reikšmę, atimdami periodą 2*PI
    { if (x>0)  x-=2*PI;
      else  x+=2*PI;
    }
     /* toliau pasiruošiam kintamųjų pradines reikšmes: k – matem.eilutės
        nario numeriui, a – nulinis narys, s – narių sumai kaupti  */
```

```
  k = 0;  a = 1 ;   s = 1 ;
  do
   { k = k + 1 ;                    // didinamas matem.eilutės nario numeris
      a = -a*x*x/(2*k*(2*k-1));    // gaunama eilinio sumos nario a reikšmė
      s = s + a ;                   // kaupiama narių suma
    }
   while (fabs(a) > eps);           // fabs – duoda a absoliutinį dydį float tipo
   cout << "Musu programa gautas cos(x)= " << s << endl;
   cout << "Standart. f-ja gautas cos(x)= " << cos(x) << endl;
   system ("PAUSE");                // stabdoma programa rezultatams pažiūrėti
   return 0;
}
```

7. Funkcijos

Funkcijos įgalina skaidyti sudėtingas programas į atskiras dalis - modulius. Naudodami jas galime išnaudoti visą C++ kalbos struktūrinio programavimo potencialą, geriau organizuoti programuotojų kolektyvo darbą.

Funkcija – tai sakinių grupė, kuri gali būti vykdoma kreipusis į ją iš bet kurios programos vietos. Bendrasis jos pavidalas yra toks:

tipas vardas (parametras1, parametras2, ...) { sakiniai }

kur:

- tipas – tai bazinis žodis, nurodantis funkcijos apskaičiuotos ir grąžinamos reikšmės tipą. Jis yra nebūtinas (*optional*);
- vardas, kurį duodam funkcijai ir kuriuo reikės kreiptis į ją;
- parametrai – tai kintamieji (jų gali būti tiek, kiek reikia), nurodant kiekvieno jų tipą (pvz., int a). Šie kintamieji yra lokalūs ir galioja tik funkcijos ribose. Parametrai įgalina perduoti funkcijai argumentų reikšmes, kai yra kreipiamasi į ją. Parametrai vienas nuo kito skiriami kableliu;
- sakiniai, apkabinti riestiniais skliaustais {}. Jie dar vadinami funkcijos kūnu.

Štai pirmasis funkcijos pavyzdys:

```
    // funkcijos pavyzdys
#include <iostream>
using namespace std;

int sudetis (int a, int b)
{ int  r;
  r=a+b;
  return (r);
}

int main ()
{ int  z;
  z = sudetis (5,3);
  cout << "Rezultatas yra " << z;
  return 0;
}
```

Rezultatas yra 8

Pradėdami nagrinėti šį pavyzdį prisiminkime, kad C++ programa visada pradedama vykdyti nuo main funkcijos. Čia main funkcija prasideda int tipo kintamojo z paskelbimu. Po to, priskyrimo sakinio dešinėje pusėje yra kreipinys į

funkciją sudetis. Pastebėkime, kad kreipinys į funkciją ir funkcijos antraštė (pirmasis funkcijos sakinys) yra panašūs:

```
int  sudetis ( int a , int b )
z = sudetis (     5 ,     3   );
```

Funkcijos antraštės parametrai ir kreipinio į funkciją argumentai turi aiškų atitikimą.

main funkcijoje kreipusis į funkciją sudetis, programos vykdymas iš main funkcijos persikelia į funkciją sudetis, o abiejų argumentų reikšmės 5 ir 3 atitinkamai perduodamos parametrams a ir b.

Funkcijoje sudetis yra paskelbtas lokalus kintamasis int r , kuris priskyrimo sakiniu r=a+b; įgyja kintamųjų a ir b sumos reikšmę, t.y. 8, nes kreipimosi į funkciją sudetis metu a įgijo reikšmę 5, o b – reikšmę 3.

Sakiniu

```
return (r);
```

baigiamas funkcijos sudetis darbas ir grįžtama į tą programos vietą, iš kur buvo kreiptasi (šiuo atveju į main funkcijos atitinkamą vietą). Tuo pačiu į kreipinio vietą grąžinama return sakinyje nurodyto kintamojo r reikšmė 8. Pastaroji reikšmė main funkcijoje priskiriama kintamajam z ir išvedama į ekraną.

Panagrinėkime dar vieną funkcijos pavyzdį:

```
    // kitas funkcijos pavyzdys
#include <iostream>
using namespace std;

int atimtis (int a, int b)
{ int  r;
  r=a-b;
  return (r);
}

int main ()
{ int  x=5, y=3, z;
  z = atimtis(7,2);
  cout << "Pirmas rezultatas: " << z << '\n';
  cout << "Antras rezultatas: " << atimtis(7,2) << '\n';
  cout << "Trecias rezultatas:" << atimtis(x,y) << '\n';
  z= 4 + atimtis(x,y);
  cout << "Ketvirtas rezultatas: " << z << '\n';
  return 0;
}
```

```
Pirmas rezultatas:  5
Antras rezultatas:  5
Trecias rezultatas: 2

Ketvirtas rezultatas:  6
```

Šiame pavyzdyje turime funkciją atimtis. Ji skaičiuoja abiejų parametrų gaunamų reikšmių skirtumą, kuris ir grąžinamas kaip rezultatas.

Nagrinėdami pagrindinę funkciją main pastebėsime, kad joje yra keletas kreipinių į funkciją atimtis. Kreipinys į funkciją pakeičiamas reikšme, kurią ji grąžina return() sakiniu. Matome, kad kreipiniai į funkcijas gali būti įvairiose programos vietose, kur leistina rašyti išraiškas: cout sakinyje, priskyrimo sakinyje ir daug kur kitur.

7.1. *void* tipo funkcijos

Prisiminkime bendrąjį funkcijos pavidalą:

tipas vardas (parametras1, parametras2, ...) { sakiniai }

Matome, kad pradžioje nurodomas tipas, t.y. funkcijos grąžinamos reikšmės tipas. Bet ką daryti, jei mums nereikia grąžinamos reikšmės?

Tarkime, mes norime sudaryti funkciją, kuri tik išvestų pranešimą į ekraną. Mums nereikia jokios jos gražinamos reikšmės. Tokiais atvejais funkcijos tipas nurodomas baziniu žodžiu void (angl. *void* – tuščias, neturintis).

Programa	Rezultatas

`// void  funkcijos pavyzdys`
`#include <iostream>`
`using namespace std;`
 	
`void pranesu ()`
`{ cout << "As taip pat esu funkcija!";`
`}`
 	
`int main ()`
`{ pranesu ();`
`  return 0;`
`}`
	As taip pat esu funkcija!

Bazinis žodis void taip pat gali būti funkcijos parametrų sąraše, kai norime aiškiai parodyti, kad jokie parametrai kreipiantis į funkciją nereikalingi. Pvz., funkcija pranesu galėtų būti paskelbta taip:

```
void pranesu (void)
{ cout << "As taip pat esu funkcija!";
}
```

Tačiau rašyti void parametrų sąraše nebūtina. C++ kalboje funkcijos parametrų sąrašas tiesiog gali būti tuščias, jei nereikia parametrų.

Taip pat būtina atsiminti, kad kreipinio į void tipo funkciją pavidalas yra atskiras sakinys, susidedantis tik iš funkcijos vardo ir suskliaustų parametrų. Nesant

parametrų, skliaustai vis viena turi būti rašomi. Todėl kreipinys į funkciją pranesu yra toks:

```
pranesu ();
```

Skliaustai aiškiai parodo, kad tai yra kreipinys į funkciją, o ne C++ kalbos kintamasis. Kreipinys

```
pranesu;
```

yra klaidingas.

7.2. Argumentų reikšmių ir adresų perdavimas parametrams

Lig šiol nagrinėtose funkcijose jų parametrams būdavo perduodamos kreipinių argumentų reikšmės. Kitaip tariant, reikšmės iš argumentams skirtos vietos atmintyje buvo nukopijuojamos į parametrams paskiriamas atskiras atminties vietas. Pvz., tarkime, kad mes kreipiamės į mūsų anksčiau turėtą funkciją sudetis tokiu būdu:

```
int  x=5, y=3, z;
z = sudetis (x, y);
```

Jos parametrams perduodamos kreipinio argumentų x ir y reikšmės, t.y. 5 ir 3 atitinkamai. Kadangi funkcijos antraštė yra int sudėtis (int a, int b), tai 5 → a, 3 → b.

Šiuo atveju jokie funkcijos sudetis lokaliųjų kintamųjų a ir b reikšmių keitimai neturi įtakos kintamųjų x ir y reikšmėms, kurie yra išoriniai funkcijos atžvilgiu.

Bet kartais būtų naudinga, jei funkcija galėtų pakeisti išorinių kintamųjų reikšmes. Šiam tikslui pasiekti kreipimosi į funkciją metu parametrams reikia paskirti ne atskirą vietą atmintyje, bet tą pačią, kuri jau paskirta argumentams. Tokiu būdu parametrą patalpinus argumento adresu, keičiant vieno reikšmę, natūralu, keisis ir kito reikšmė. Pvz.:

```
      // argumentų adreso perdavimas funkcijos parametrams
#include <iostream>
using namespace std;

void dvigub (int& a, int& b, int& c)
{ a*=2;                  // prisiminkime, kad tai  a = a*2;
  b*=2;
  c*=2;
}

int main ()
{ int  x=1, y=3, z=7;
```

```
    dvigub (x, y, z);
    cout << "x=" << x << ", y=" << y << ", z=" << z;
    return 0;
}
```

x=2, y=6, z=14

Pirmas dalykas, į ką čia reikėtų atkreipti dėmesį, kad skelbiant funkciją dvigub, parametro tipą nurodančio bazinio žodžio gale pridedamas simbolis &. Tai reiškia, kad kreipiantis į funkciją parametras atmintyje bus talpinamas tuo pačiu adresu kaip ir kreipinio argumentas.

Kadangi sutapatinamos argumento ir parametro atminties sritys, tai keičiant parametro reikšmę keisis ir argumento reikšmė. Taigi, turint tokią funkcijos antraštę ir kreipinį į ją

```
void dvigub ( int& a, int& b, int& c )
dvigub ( x, y, z );
```

reikšmių perdavimas tarp argumentų ir parametrų vyks abiem kryptimis: x ↔ a, y ↔ b, z ↔ c.

Tai ir yra priežastis, kodėl aukščiau pateiktos programos main funkcijos kintamiesiems x, y ir z pradžioje priskyrus reikšmes 1, 3 ir 7 atitinkamai, po kreipinio į funkciją dvigub į ekraną išvedami dvigubai didesni skaičiai. Jei funkciją dvigub būtume paskelbę po parametrų tipo nerašydami &,

```
void dvigub (int a, int b, int c)
```

tai main funkcijoje po kreipinio į funkciją dvigub į ekraną būtų išvestos nepakeistos kintamųjų x, y ir z reikšmės.

Argumentų ir parametrų atminties sutapatinimas yra efektyvus būdas pasiekti, kad funkcija apskaičiuotų daugiau kaip vieną reikšmę. Žemiau pateikiamoje programoje yra funkcija priespo, apskaičiuojanti prieš ir po einančius skaičius duotam skaičiui.

```
    // daugiau kaip vieną reikšmę apskaičiuojanti funkcija
#include <iostream>
using namespace std;

void priespo (int x, int& pries, int& po)
{ pries = x-1;
  po = x+1;
}

int main ()
{ int  x=100, y, z;
   priespo (x, y, z);
   cout << "Skaicius=" << x << endl;
```

Skaicius=100

```
  cout << "Ankstesnis=" << y << endl;
        << "Kitas=" << z;
  return 0;
}
```

```
Ankstesnis=99
Kitas=101
```

7.3. Numatytųjų reikšmių parametrai

Skelbiant funkciją, kiekvienam jos parametrui galima nurodyti numatytąją (*default*) reikšmę. Funkcijos antraštėje tokiems parametrams numatytoji reikšmė suteikiama priskyrimo operatoriumi. Ši reikšmė naudojama tada, jei kreipinyje į funkciją nėra parametrą atitinkančio argumento. Jei kreipinyje į funkciją yra tokį parametrą atitinkantis argumentas, numatytoji reikšmė ignoruojama ir parametrui perduodama argumento reikšmė. Pvz.:

```
        // numatytosios funkcijų parametrų reikšmės
#include <iostream>
using namespace std;

int dalyba (int a, int b=2)
{ int  r;
  r=a/b;
  return (r);
}

int main ()
{ cout << dalyba (12) << endl;
  cout << dalyba (20,4);
  return 0;
}
```

```
6
5
```

Šioje programoje yra du kreipiniai į funkciją dalyba. Pirmajame kreipinyje

```
dalyba (12)
```

yra tik vienas argumentas, nors pagal funkcijos aprašą galėtų būti du. Taigi, funkcijai dalyba pradėjus darbą, jos antrojo parametro reikšmė bus lygi 2, o rezultatas bus 6.

Antrajame kreipinyje

```
dalyba (20,4)
```

yra du argumentai. Todėl funkcijos dalyba antrojo parametro b (int b=2) numatytoji reikšmė 2 ignoruojama, o b įgyja kreipinio argumento reikšmę 4. Funkcijos rezultatas bus 5.

7.4. Funkcijos vienodais vardais

C++ kalboje dvi skirtingos funkcijos gali turėti tokį pat vardą, jei jų parametrų tipai arba kiekis skiriasi. Pvz.:

```
    // funkcijos vienodais vardais
#include <iostream>
using namespace std;

int daryk (int a, int b)
{ return (a*b);
}

float daryk (float a, float b)
{ return (a+b);
}

int main ()
{ int  n=5, m=2;
  float x=5.0, y=2.0;
  cout << daryk (x,y) << endl;
  cout << daryk (n,m) << endl;
  return 0;
}
```

```
7
10
```

Šiame pavyzdyje yra apibrėžtos dvi funkcijas tokiu pačiu vardu daryk. Tačiau vienos iš jų parametrai yra int tipo, o kitos - float tipo. Į kurią iš jų reikia kreiptis, kompiliatorius nustato išnagrinėjęs kreipinio argumentų tipus. Pirmo kreipinio rezultatas yra argumentų reikšmių suma 7, o antro − sandauga 10.

Atkreipkime dėmesį į tai, kad negalima naudoti tokio paties vardo dviem funkcijoms, jei skiriasi tik jų gražinamų reikšmių tipai.

7.5. Funkcijų šablonai (*template*)

Funkcijos šablonas − tai specialus funkcijos pavidalas, kuris įgalina pritaikyti vieną funkciją darbui su įvairaus tipo kintamaisiais. Tai padeda išvengti pakartotino funkcijos programos rašymo skirtingo tipo duomenims.

Šitokia funkcijų savybė gaunama panaudojus šablono parametrus. Tai specialūs parametrai, kuriais galima perduoti funkcijai informaciją apie duomenų tipą.

Funkcijos šablono bendrasis pavidalas yra toks:

```
template <typename vardas> funkcijos_programa;
```

arba

```
template <class vardas> funkcijos_programa;
```

Šablono parametras gali būti apibūdinamas baziniu žodžiu typename arba class.

Pvz., sudarykime šabloną funkcijos, kuri iš gautų dviejų skaičių gražina didesnįjį:

```
template <typename tpv>
tpv didesnis (tpv a, tpv b)
{ tpv x;
  if (a>b)  x=a;
  else  x=b;
  return (x);
}
```

Čia panaudotas šablono parametras tpv, skirtas duomenų tipui nurodyti, tačiau konkreti jo reikšmė dar nėra aiški. Ji tampa aiški tik po kreipinio į funkciją.

Bendrasis kreipinio į šablonu apibrėžtą funkciją pavidalas yra:

```
funkcijos_vardas <tipas> (argumentai);
```

Todėl į mūsų funkcijos didesnis šabloną turi būti kreipiamasi, pvz., taip:

```
int  j=5, k=6, m;
m = didesnis <int> (j,k);
```

Kompiliatorius, sutikęs tokį kreipinį, funkcijos šablone automatiškai vietoje parametro tpv įstato mūsų atveju int ir tik tada kreipiasi į funkciją.

Imkime ištisos programos pavyzdį:

```
    // funkcijos šablonas
#include <iostream>
using namespace std;

template <typename tpv>
tpv didesnis (tpv a, tpv b)
{ tpv  x;
  if (a>b)  x=a;
  else  x=b;
  return (x);
}

int main ()
{ int  j=5, k=6, m;
  float p=13.83, q=5.47, r;
  m = didesnis <int> (j,k);
  r = didesnis <float> (p,q);
  cout << m << endl;
```

6

```
  cout << r << endl;
  return 0;
}
```

13.83

Čia į funkcijos didesnis šabloną kreipiamasi du kartus. Pirmo kreipinio argumentai yra int tipo, o antro – float tipo. Todėl kompiliatorius parengia dvi funkcijos versijas ir kiekvieną kartą kreipiamasi į reikiamą funkcijos versiją.

Turėto pavyzdžio atveju kompiliatorius sugeba parengti dvi funkcijos didesnis versijas, netgi jei kreipiniuose į jas akivaizdžiai nenurodysime tipo parametro, t.y. jei vietoje didesnis<int>(j,k) rašysime tiesiog didesnis(j,k), o vietoje didesnis<float>(p,q) rašysime didesnis(p,q). Jei kreipinyje yra, pvz., int tipo argumentai j ir k, tai ir funkcijos šablone juos atitinkantys parametrai a ir b taip pat turi būti int tipo. Todėl šablono parametras tpv tampa int.

Kadangi mūsų funkcijos šablone yra tik vienas šablono parametras tpv, o pati funkcija turi du parametrus a ir b, kurie yra vienodo tipo, mes negalime kreiptis į funkciją naudodami skirtingo tipo argumentus. Pvz.,

```
  int  k;
  float  p;
  q = didesnis (k,p);        // neleistinas kreipinys
```

Tačiau galima apibrėžti funkcijos šabloną, kuriame būtų daugiau kaip vienas šablono parametras. Pvz.:

```
        // funkcijos šablonas su dviem šablono parametrais
#include <iostream>
using namespace std;

template <typename t1, typename t2>
t2 mazesnis (t1 a, t2 b)
  { float  x;
    t2  y;
    if (a<b) x=a;
    else  x=b;
    y=t2(x);              // tam atvejui, kai t2 tampa int, nes x – float tipo
    return (y);
  }

int main ()
{ int  j=5, k=6, m, n;
  float  p=13.83, q=5.47, r, v;
  r = mazesnis <int,float> (j,p);
```

```
  m = mazesnis <float,int> (q,k);
  v = mazesnis <float,float> (p,q);
  n = mazesnis <int,int> (j,k);
  cout << r << endl;
  cout << m << endl;
  cout << v << endl;
  cout << n << endl;
  return 0;
}
```

Šiame pavyzdyje funkcijos mazesnis šablone yra du šablono parametrai t1 ir t2. Kreipiniai į funkciją mazesnis taip pat gali būti rašomi ir paprasčiau, t.y. akivaizdžiai nenurodant tipų šablono parametrams:

```
r = mazesnis(j,p);
k = mazesnis(q,k);
v = mazesnis(p,q);
n = mazesnis(j,k);
```

Pastebėkime, kad funkcijos mazesnis gražinamos reikšmės tipas sutampa su antro argumento tipu.

Taip pat atkreipkime dėmesį į funkcijos mazesnis šablono sakinį y=t2(x); Jame t2 panaudotas kaip funkcijos vardas, kuri konvertuoja argumento x reikšmę iš float tipo į t2 tipą (žiūr. 4.10 skyrių "Aiškiai nurodomi duomens tipo keitimo operatoriai").

7.6. *inline* funkcijos

Funkcijos apraše inline baziniu žodžiu nurodoma kompiliatoriui, kad jis funkcijos sakinius (funkcijos kūną) įterptų į kiekvieną programos vietą, kur yra kreipinys į tą funkciją. inline nuoroda nekeičia funkcijos elgesio. Ji tik padeda išvengti kintamųjų steko sudarymo ir šokinėjimų programoje iš vienos vietos į kitą. inline funkcijos yra makrokomandų ekvivalentas. Patartina inline naudoti tik trumpoms funkcijoms.

Bendrasis inline funkcijų pavidalas:

```
inline tipas vardas(parametras1,parametras2,...){sakiniai}
```

Į jas kreipiamasi taip pat, kaip ir kitas funkcijas.

7.7. Rekursijos

Rekursija – tai funkcijos savybė jos viduje esančiais sakiniais kreiptis į save pačią. Tai gali būti naudinga įvairiuose uždaviniuose, pvz., skaičiaus n faktorialui n! gauti:

n! = 1*2*3*…(n-1)*n.

Tai galima užrašyti ir taip:

n! = n*(n-1)!

Rekursyvi funkcija skaičiaus faktorialui skaičiuoti atrodo taip:

```
    // faktorialo skaičiavimas rekursyviai
#include <iostream>
using namespace std;

long faktorialas (long a)
{ if (a > 1)
     return (a * faktorialas (a-1));
   else
     return (1);
}

int main ()
{ long  skaicius;
   cout << "Iveskite skaiciu: ";
   cin >> skaicius;
   cout << skaicius << "! = "
           << faktorialas (skaicius);
   return 0;
}
```

```
Iveskite skaiciu: 9

9! = 362880
```

Pastebėkime, kad funkcijoje faktorialas kreipiamasi į save pačią, jei argumento reikšmė yra didesnė už 1. Priešingu atveju funkcijos reikšmė būtų lygi 0 ir turėtume nesibaigiantį ciklą, nes toliau vyktų daugyba iš neigiamų skaičių, kol neįvyktų steko perpildymas.

Apskritai, faktorialo skaičiavimo funkcijos ribotumas yra tas, kad ja, naudojant netgi long tipo kintamuosius, galima gauti rezultatą, ne didesnį kaip 15!.

7.8. Funkcijų skelbimas

Lig šiol mūsų nagrinėtose programose funkcijos buvo paskelbtos anksčiau, negu sutikdavome pirmą kreipinį į jas. Šie kreipiniai buvo pagrindinėje main funkcijoje, kurią rašėme programos pabaigoje. Bet jeigu main funkciją rašytume anksčiau nei kitas programos funkcijas, tai gautume kompiliavimo klaidą. Priežastis yra ta, kad funkcija turi būti paskelbta anksčiau, nei sutinkamas pirmas kreipinys į ją.

Yra būdas leidžiantis nerašyti viso funkcijos programos teksto (kūno) anksčiau už main ar kitokią funkciją, kuriose yra kreipiniai į pirmąją. Iki kreipinio į funkciją užtenka paskelbti tik funkcijos prototipą, o ne visą funkcijos tekstą. Funkcijos

prototipe nurodomas tik jos tipas, vardas ir parametrų tipai. To visiškai pakanka kompiliatoriui. Taigi, bendrasis funkcijos prototipo pavidalas yra:

tipas vardas (parametro1_tipas, parametro2_tipas, ...);

Funkcijos prototipas - tai tik funkcijos antraštė, kurioje gali būti praleisti parametrų vardai (parametrų tipai turi būti). Kadangi funkcijos sakiniai (kūnas) praleidžiami, tai prototipo gale rašomas kabliataškis (;). Pvz., funkcijos daugyba, turinčios du int tipo parametrus a ir b, prototipas gali būti rašomas vienu iš dviejų būdų:

```
int daugyba (int a, int b);
int daugyba (int, int);
```

Rašant ir parametrų vardus, funkcijos prototipas būna aiškesnis.

Žemiau pateikiamas C++ programos pavyzdys, kurioje naudojami funkcijų prototipai, o patys funkcijų tekstai rašomi po pagrindinės funkcijos main:

```
      // funkcijų prototipų naudojimas
#include <iostream>
using namespace std;

void nelyg (int a);      // funkcijos nelyg prototipas
void lyg (int a);        // funkcijos lyg prototipas

int main ()
{ int  i;
  cout << "Darbui baigti iveskite 0.\n" << endl;
  do
    { cout << "Iveskite skaiciu: ";
      cin >> i;
      nelyg (i);
    }
  while (i!=0);
  return 0;
}

void nelyg (int a)       // visas funkcijos nelyg tekstas
{ if ((a%2)!=0)  cout << "Skaicius nelyginis.\n";
  else  lyg (a);
}

void lyg (int a)         // visas funkcijos lyg tekstas
{ if ((a%2)==0)  cout << "Skaicius lyginis.\n";
  else  nelyg (a);
}
```

```
Darbui baigti iveskite 0.

Iveskite skaiciu: 9
Skaicius nelyginis.
Iveskite skaiciu: 6
Skaicius lyginis.
Iveskite skaiciu: 1030
Skaicius lyginis.
Iveskite skaiciu: 0
Skaicius lyginis.
```

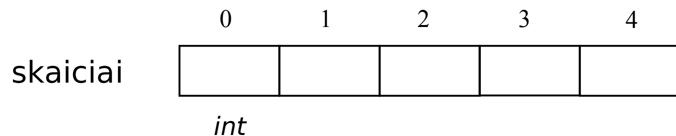
Savo esme ši programa nėra efektyvi. Ja tik norėta pailiustruoti prototipų veikimą. Iš pradžių į akis krenta funkcijų nelyg ir lyg paskelbimas:

```
void nelyg (int a);
void lyg (int a);
```

Tai įgalina kreiptis į šias funkcijas, pvz., pagrindinėje main funkcijoje. Daugumai žmonių atrodo logiškiau, kai main funkcija, nuo kurios pradedamas visos programos darbas, yra priekyje. Visų programoje naudojamų funkcijų prototipų skelbimas programos pradžioje praktiškai yra naudingas. Taip elgiasi dauguma programuotojų.

8. Masyvai

Masyvas − tai tokio paties tipo elementų visuma, talpinamų gretimose atminties srityse. Atskiras masyvo elementas nurodomas po masyvo vardo rašant indeksus. Pvz., masyvas vardu skaiciai, turintis penkis int tipo elementus, gali būti pavaizduotas tokia schema:



Čia kiekvienas stačiakampis atitinka masyvo elementą, kuris mūsų atveju yra int tipo. Masyvo pirmojo elemento indeksas visada yra lygus 0, nepriklausomai nuo masyvo dydžio.

Kaip ir visi kintamieji, taip ir masyvai turi būti paskelbti prieš naudojant juos. C++ kalboje bendrasis masyvų paskelbimo pavidalas yra toks:

tipas vardas [kiekis];

Stačiakampiuose skliaustuose nurodoma, kiek elementų norime turėti masyve.

Aukščiau schemoje pavaizduotas masyvas skaiciai turi būti skelbiamas taip:

```
int  skaiciai [5];
```

Atkreipkime dėmesį, kad masyvo elementų kiekis turi būti nurodomas konstanta. Apie kitokias galimybes šioje mokymo priemonėje rašoma vėliau.

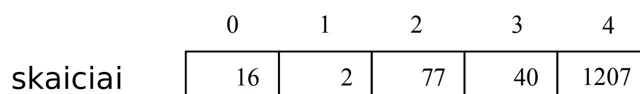
8.1. Masyvų inicializacija

Įprastu būdu paskelbus lokalųjį (pvz., galiojantį tik funkcijos ribose) masyvą, jo elementų reikšmės yra neapibrėžtos. Kita vertus, globalieji ir statiniai masyvai automatiškai užpildomi nuliais.

Lokaliųjų ir globaliųjų masyvų atvejais yra galimybė suteikti jų elementams pradines reikšmes, t. y. inicializuoti masyvus, rašant tas reikšmes tarp riestinių skliaustų {}. Pvz.,

```
int  skaiciai [5] = { 16, 2, 77, 40, 1207 };
```

Šiuo skelbimu sukuriamas masyvas, kurį schematiškai galima pavaizuoti taip:



Reikšmių kiekis tarp riestinių skliaustų {} turi būti ne didesnis, nei tarp stačiakampių skliaustų [] nurodytas masyvo elementų kiekis.

Kai yra nurodomos masyvo elementų pradinės reikšmės, masyvo elementų kiekio tarp stačiakampių skliaustų [ ] galima ir nenurodyti. Tokiu atveju kompiliatorius laikys, kad masyvo dydis yra toks, kiek pradinių reikšmių nurodyta riestiniuose skliaustuose {}:

```
int  skaiciai [] = { 16, 2, 77, 40, 1207 };
```

Taip paskelbtame masyve skaiciai bus 5 elementai, nes tiek pradinių reikšmių nurodėme.

8.2. Kreipimasis į masyvo elementus

Bet kurioje programos vietoje, kur yra masyvo galiojimo sritis, galima kreiptis į jo elementus individualiai, kaip tai daroma su įprastais kintamaisiais. Atskiras masyvo elementas nurodomas taip:

vardas [indeksas]

Remiantis anksčiau turėtu pavyzdžiu, kur masyvas skaiciai turi penkis int tipo elementus, kiekvienas iš jų gali būti nurodomas taip: skaiciai [0], skaiciai [1],..., skaiciai [4]. Norint priskirti reikšmę 97 trečiam masyvo skaiciai elementui, reikėtų rašyti tokį sakinį:

```
skaiciai [2] = 97;
```

o norint trečio elemento reikšmę priskirti, pvz., kintamajam a, reikėtų rašyti taip:

```
a = skaiciai [2];
```

Taigi, išraiška skaiciai [2] visais atvejais traktuojama kaip int tipo kintamasis.

Pastebėkime, kad paskelbus 5 elementų masyvą, paskutinio jo elemento indeksas yra vienetu mažesnis, pvz., skaiciai [4]. Jeigu užrašysime skaiciai [5], tai išeisime už masyvo ribų. C++ kalbos sintaksės prasme tai yra leistina, kreipiantis į elementus už masyvo ribų kompiliatorius klaidos nefiksuos, tačiau programos rezultatai gali būti klaidingi. Viso to priežastį pamatysime vėliau, kai susipažinsime su rodyklėmis (*pointer*).

Taip pat atkreipkime dėmesį į du stačiakampių skliaustų [ ] naudojimo atvejus dirbant su masyvais. Pirma, skelbiant masyvą tarp jų nurodomas masyvo dydis, antra − juose nurodomas išraiškose naudojamo atskiro masyvo elemento indeksas. To nereikėtų painioti.

```
int  skaiciai [5];          // skelbiamas masyvas
skaiciai [2] = 97;       // kreipiamasi į masyvo elementą
```

Skelbiant masyvą, pradžioje rašomas tipą nurodantis bazinis žodis, o prieš atskirą masyvo elementą tokio žodžio niekada nebūna.

Toliau pateikama keletas su masyvais leistinų veiksmų:

```
skaiciai [0] = a;
skaiciai [a] = 75;
b = skaiciai [a+2];
skaiciai [ skaiciai [a] ] = skaiciai [2] + 5;
```

<code>// programa su masyvu
#include <iostream>
using namespace std;
 	
int skaiciai [] = {16, 2, 77, 40, 1207};
int n, rezult=0;
 	
int main ()
{ for (n=0 ; n<5 ; n++)
      rezult += skaiciai [n];
   cout << rezult;
   return 0;
}</code>
	1342

Ši programa suskaičiuoja visų masyvo skaiciai elementų sumą.

8.3. Daugiamačiai masyvai

Daugiamačiai masyvai gali būti apibūdinami kaip “masyvų masyvai“. Pvz., dvimatį masyvą galime įsivaizduoti kaip vienodo tipo elementų lentelę.

ruta	0	1	2	3	4
0					
1					
2					

Čia pavaizduotas dvimatis 3 iš 5 masyvas ruta. Toks int tipo skaičių masyvas C++ kalboje skelbiamas taip:

```
int  ruta [3] [5];
```

Jo antros eilutės ketvirtas (ketvirto stulpelio) elementas nurodomas kaip

```
ruta [1] [3]
```

Atsiminkime, kad indeksai visada prasideda nuo 0.

Daugiamačiai masyvai neapsiriboja dviem indeksais. Jų gali būti tiek, kiek reikia. Bet būkime atidūs. Masyvui reikalingas atminties dydis greitai didėja sulig kiekvienu nauju indeksu.

Daugiamačiai masyvai yra tik programuotojų abstrakcija. Tokį patį rezultatą galima gauti su vienmačiu masyvu, kurio dydis būtų lygus daugiamačio masyvo indeksų sandaugai:

```
int  ruta [3] [5];     //  ekvivalentiški masyvai
int  ruta [15];        //   3 * 5 = 15
```

Panagrinėkime toliau du programų fragmentus, duodančius tokį patį rezultatą. Viename iš jų naudojamas dvimatis masyvas, kitame − vienmatis masyvas.

Dvimatis masyvas:

```
#define plotis 5
#define aukstis 3
int  ruta [aukstis] [plotis];
int  n,m;

int main ()
{ for (n=0; n<aukstis; n++)
    for (m=0; m<plotis; m++)
      ruta [n] [m] = (n+1)*(m+1);
    return 0;
}
```

Vienmatis masyvas:

```
#define plotis 5
#define aukstis 3
int  ruta [aukstis * plotis];
int  n,m;

int main ()
{ for (n=0; n<aukstis; n++)
     for (m=0; m<plotis; m++)
        ruta [n*plotis+m] = (n+1)*(m+1);
   return 0;
}
```

Nė viena iš šių programų neišveda rezultatų į ekraną, tačiau patalpina juos masyvui ruta skirtoje atmintyje tokia tvarka:

ruta	0	1	2	3	4
0	1	2	3	4	5
1	2	4	6	8	10
2	3	6	9	12	15

Šiose programose apibrėžtosios konstantos (#define) pavartotos tam, kad būtų paprasčiau modifikuoti programą. Pvz., prireikus masyvo indekso aukstis reikšmę 3 pakeisti į 4, tai galima padaryti labai paprastai. Tereikia programos eilutę

```
#define aukstis 3
```

pakeisti į

```
#define aukstis 4
```

Daugiau jokių pakeitimų programoje daryti nereikia.

8.4. Masyvai kaip funkcijų parametrai

Funkcijų parametrais gali būti ir masyvai, nežiūrint to, kad C++ kalboje nėra galimybės perduoti ištisų masyvų. Todėl funkcijos parametrai-masyvai patalpinami

atmintyje tik kreipinio į funkciją argumentams-masyvams skirtais adresais, t.y. vyksta tik argumento-masyvo adreso perdavimas funkcijai.

Masyvas, kaip funkcijos parametras, skelbiamos funkcijos antraštėje nurodomas po jo vardo rašant stačiakampius skliaustus [] , tačiau juose nieko nenurodant. Pvz.:

```
void procedura ( int para [] )
```

Čia parametras vardu para yra int tipo masyvas. Norint šiai funkcijai perduoti masyvą, paskelbtą kaip int mas[40]; , pakanka tokio kreipinio:

```
procedura (mas);
```

Žemiau pateikiama programa, kurioje yra funkcija su parametru-masyvu:

```
    // masyvai kaip funkcijų parametrai
#include <iostream>
using namespace std;

void masyvo_isvedimas (int para [], int ilgis)
{ for ( int n=0; n<ilgis; n++ )
      cout << para [n] << " ";
  cout << "\n";
}

int main ()
{ int  mas1 [] = {5, 10, 15};
  int  mas2 [] = {2, 4, 6, 8, 10};
  masyvo_isvedimas (mas1, 3);
  masyvo_isvedimas (mas2, 5);
  return 0;
}
```

```
5 10 15
2 4 6 8 10
```

Čia funkcijos masyvo_isvedimas pirmas parametras (int para []) gali priimti bet kokio ilgio int tipo masyvą. Tam yra įvestas antras parametras, kuriuo funkcijai perduodamas apdorojamo masyvo ilgis. Tai įgalina for ciklu išvesti į ekraną tiek elementų, kokio dydžio masyvas buvo kreipinyje į funkciją masyvo_isvedimas.

Funkcijos parametru gali būti ir daugiamatis masyvas. Pvz., trimačio masyvo, kaip parametro, formatas yra:

tipas vardas [] [skaičius] [skaičius]

Funkcijos su daugiamačiu masyvu kaip parametru antraštės pavyzdys:

```
void procedura ( int trimas [] [3] [4] )
```

Pastebėkime, kad tarp pirmųjų stačiakampių skliaustų nerašoma nieko, kai tuo tarpu kituose skliaustuose turi būti sveikieji skaičiai.

8.5. Programų su masyvais pavyzdžiai

1 pavyzdys.

Turime 20 skaičių masyvą. Klaviatūra įveskime juos į kompiuterio atmintį, o po to apskaičiuokime įvestų skaičių vidurkį bei raskime didžiausią ir mažiausią skaičius.

Rezultatus išveskime į monitorių taip: visų pirma išveskime visą masyvą po 8 skaičius kiekvienoje eilutėje, o po to vidurkį, didžiausią ir mažiausią skaičius.

Algoritmas.

Ieškant didžiausio elemento, visų pirma daroma prielaida, kad pirmas masyvo elementas (arba bet kuris kitas masyvo elementas) yra didžiausias. Jis įsimenamas atskirame kintamajame. Po to šio atskiro kintamojo reikšmė lyginama su visais masyvo elementais. Kai tik randamas dar didesnis masyvo elementas, atskirame kintamajame įsimenama jo reikšmė.

Analogiškai randamas ir mažiausias elementas.

```
     /* Skaičių masyvo vidurkio, didžiausio ir mažiausio elementų radimas */
#include <iostream>
#include <cstdlib>          //  iš čia ima funkciją  system
using namespace std;
#define KIEKIS 20
#define BRUKSNYS “--------------------------------“
float  mas [KIEKIS];
float  did, maz, vid;

main ()
{ int  j;
  cout << “Iveskite “ << KIEKIS << “ skaiciu:“ << endl;
  for (j=0; j<KIEKIS; j++)  cin >> mas [j];
  vid = 0;                 // skaičių sumai kaupti
  did = mas [0];           // prielaida, kad didžiausias elementas yra mas[0]
  maz = mas [0];           // prielaida, kad mažiausias elementas yra mas[0]
   for ( j=0; j<KIEKIS; j++ )
   { vid += ma s[j];
     if (mas [j]>did) did = mas [j];
     if (mas [j]<maz) maz = mas [j];
   }
   vid /= KIEKIS;      // gaunamas skaičių vidurkis
                   // toliau išvedame į ekraną masyvą po 8 elementus eilutėse
   cout << “Skaiciu masyvas:” << endl;
   cout << BRUKSNYS << BRUKSNYS << endl;
```

```
    for (j=0; j<KIEKIS; j++)
      { if ( (j % 8) == 0 )  cout << endl;          // pereinam į naują eilutę
        cout << mas [j] << “\t”;
      }
    cout << endl << BRUKSNYS << BRUKSNYS << endl;
              // toliau išvedami rezultatai
    cout << “Skaiciu vidurkis: “ << vid << endl;
    cout << “Diziausias skaicius: “ << did << endl;
    cout << “Maziausias skaicius: “ << maz << endl;
    system ("PAUSE");              // stabdo programą rezultatams pažiūrėti
    return 0;
  }
```

2 pavyzdys.

Tarkime, turime ne didesnį kaip 100 skaičių masyvą. Įveskime norimą masyvo elementų kiekį, o po to išveskime juos į monitorių, surikiuotus nemažėjimo tvarka (tolesnis elementas turi būti didesnis arba lygus prieš buvusiam). Prieš įvedant masyvo elementus, visų pirma įveskime masyvo elementų kiekį nurodantį skaičių.

Algoritmas.

Masyvo elementams rikiuoti yra daug algoritmų. Imkime vieną iš paprastesnių − burbulo algoritmą. Rikiuojant lyginami du gretimi masyvo elementai ir, jeigu reikia, jie sukeičiami vietomis. Tokiu būdu “prabėgus” visą masyvą, didžiausias elementas perkeliamas į masyvo galą. Po to viskas daroma iš naujo, ir į antrą vietą nuo masyvo galo perkeliamas antras pagal dydį masyvo elementas, ir t.t. Tam programoje reikia ciklo cikle.

```
      // skaičių masyvo elementų rikiavimas
  #include <iostream>
  #include <cstdlib>          // iš čia ima funkciją  system
  using namespace std;
  #define BRUKSNYS "--------------------------------"
  float  skm [100];

  main ()
  { float  t;
    int  k, m, n;
```

```
   cout << “Kiek skaiciu noresite ivesti: “;
   cin >> k;
   cout << “Iveskite “ << k << “ skaiciu:“ << endl;
   for (m=0; m<k; m++)
     cin >> skm [m];
        //  toliau rikiuojame masyvo skm elementus nemažėjimo tvarka
   for (m=0; m<(k-1); m++)
     for (n=0; n<(k-1-m); n++)
       if (skm [n] > skm [n+1])
        { t = skm [n];                  //   masyvo elementų
          skm [n] = skm [n+1];      //   keitimas
          skm [n+1] = t;              //   vietomis
        }
                 // toliau į ekraną išvedamas surikiuotų skaičių masyvas
   cout << “\nSurikiuoti skaičiai:\n“;
   for (m=0; m<k; m++)
    cout << skm [m] << “  “;
   cout << endl;
   system ("PAUSE");          // stabdo programą rezultatams pažiūrėti
   return 0;
 }
```

Šios programos if sakinyje pakeitus palyginimo ženklą priešingu, masyvo elementai būtų rikiuojami nedidėjimo tvarka.

3 pavyzdys.

Turime 6 x 8 dydžio matricą (dvimatį masyvą). Raskime jos didžiausią elementą ir į ekraną išveskime visus to stulpelio ir tos eilutės elementus, kuriuose yra didžiausias matricos elementas.

Algoritmas.

Matricos didžiausias elementas ieškomas panašiai, kaip ir vienmačio masyvo atveju (žiūr. 1 pavyzdį šiame skyriuje). Skirtumas toks, kad matricos atveju reikia ciklo cikle.

Čia mums rūpi ne pati didžiausio elemento reikšmė, bet jo vieta matricoje, t.y. indeksai. Todėl pradžioje, papildomam kintamajam suteikus kurio nors matricos elemento reikšmę (prielaida, kad jis yra didžiausias), būtina užfiksuoti ir jo indeksus. Tikrinimo metu radus didesnį elementą, užfiksuojami ir jo indeksai.

```
      // matrica
#include <iostream>
#include <cstdlib>              // iš čia ima funkciją  system
using namespace std;
#define STULP 8
#define EILUT 6
float  matr [EILUT] [STULP];

main ()
{ float  t;
  int  st, eil, m, n;
  cout << "Iveskite " << EILUT << 'x' << STULP
       << " dydzio matricos elementus:\n";
  for (m=0; m<EILUT; m++)
   for (n=0; n<STULP; n++)
     cin >> matr [m] [n];
  cout << "\nIvesta matrica\n";
  for (m=0; m<EILUT; m++)
   { for (n=0;n<STULP; n++)
       cout << matr [m] [n] <<"\t";
     cout << endl;
   }
          // ieškome didžiausio elemento matricoje matr
  t = matr [0] [0];          // prielaida, kad didžiausias elementas yra matr[0][0]
  st = 0;  eil = 0;          // didžiausio elemento indeksų reikšmėms
  for (m=0; m<EILUT; m++)
   for (n=0; n<STULP; n++)
    if (matr [m] [n] > t)
     { t = matr [m] [n];
       st = n;   eil = m;
      }
          // išvedame atitinkamo matricos matr stulpelio ir eilutės elementus
  cout << "\nStulpelis su didz. matricos elementu:\n";
  for (m=0; m<STULP; m++)  cout << matr [m] [st] << "  ";
  cout << "\nEilute su didz. matricos elementu:\n";
  for (m=0; m<EILUT; m++)  cout << matr [eil] [m] << "  ";
  cout << endl;
  system ("PAUSE");                // stabdo programą rezultatams pažiūrėti
  return 0;
}
```

9. Simbolių sekos

C++ kalboje yra galinga standartinė biblioteka <string>, kurioje yra patogios priemonės darbui su simbolių eilutėmis. Apie tai trumpai rašyta 2.6 skyriuje. Kadangi eilutės iš tikro yra simbolių sekos, jas galima pavaizduoti kaip char tipo elementų masyvus. Pvz., masyvas

```
char  mikas [20];
```

gali įsiminti 20 simbolių. Jį galima pavaizduoti taip:

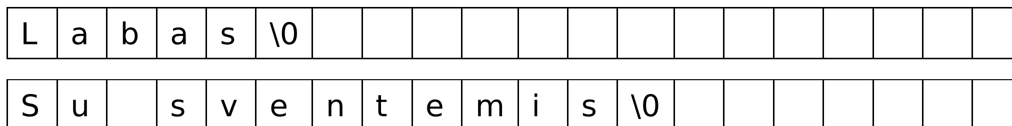
mikas

Taigi, tokiame masyve galėsime saugoti iki 20 simbolių ilgio sekas.

Kadangi simbolių masyve galima saugoti ir trumpesnes simbolių sekas nei nurodytas masyvo dydis, yra naudojamas specialus simbolis ‘\0’ (*null*) sekos pabaigai nurodyti.

Mūsų char tipo 20 elementų dydžio masyvui mikas priskirtas simbolių sekas “Labas” ir “Su sventemis” galima būtų pavaizduoti taip:

mikas



Pastebėkime, kad priskirtos simbolių sekos gale įterpiamas *null* simbolis (‘\0‘). Po jo likusiuose baituose yra neapibrėžta informacija.

9.1. Simbolių sekų inicializacija

Simbolių masyvams yra taikomos įprastos masyvų taisyklės. Tokiam masyvui norint suteikti pradinę reikšmę (norint inicializuoti jį) jo skelbimo metu, turėtume elgtis taip pat, kaip ir su kitokio tipo masyvais. Riestiniuose skliaustuose {} reikėtų nurodyti kiekvieno masyvo elemento reikšmę:

```
char zodis [] = { 'L', 'a', 'b', 'a', 's', '\0' };
```

char tipo masyvo atveju pradinės reikšmės simbolių gale dar reikėtų nurodyti ir *null* simbolį ‘\0’.

Tačiau char tipo masyvams yra dar vienas būdas pradinei reikšmei nurodyti. Tai simbolių eilutės.

Ankstesniuose šios mokymo priemonės skyriuose jau daug kartų naudojome eilutes. Jų tekstas buvo apkabinamas dvigubomis kabutėmis (“), pvz., “Labas“.

Dvigubomis kabutėmis apkabinta eilutė yra ne kas kita, kaip simbolių seka, kurios pabaiga automatiškai nurodoma *null* simboliu (‘\0‘). Todėl turėtą char tipo masyvą zodis lengviau yra inicializuoti taip:

```
char zodis [] = "Labas";
```

Abiem inicializavimo atvejais masyvas zodis paskelbiamas 6 elementų dydžio: 5 elementai žodžiui “Labas“ plius jo pabaigą nurodančiam *null* simboliui (‘\0‘).

Pastebėkime, kad po char tipo masyvo paskelbimo priskyrimas jam reikšmės priskyrimo sakiniu skiriasi nuo inicializacijos skelbimo metu. char tipo masyvai ir simbolių eilutės yra ištisi duomenų blokai, o priskyrimo operatoriumi kopijuoti blokų negalima. Todėl tokie priskyrimo sakiniai, kaip

```
zodis = "Labas";
zodis [] = "Labas";
zodis = { 'L', 'a', 'b', 'a', 's', '\0' };
```

yra neleistini. Viso to priežastis taps aiškesnė, kai susipažinsime su rodyklėmis (*pointer*).

9.2. Simbolių sekų naudojimas

Natūralu, kad C++ kalboje eilutės yra traktuojamos kaip simbolių sekos, pasibaigiančios *null* simboliu (‘\0‘). Todėl jos gali būti naudojamos daug kur. Faktiškai eilutės yra char tipo elementų masyvai ir taip pat gali būti naudojamos daugeliu atvejų.

Pvz., įvedimo/išvedimo sakiniai cin ir cout palaiko *null* simboliu pasibaigiančias simbolių sekas, todėl jos gali būti naudojamos šiuose sakiniuose. Pvz.:

```
    // null simboliu pasibaigiančios simbolių sekos
#include <iostream>
using namespace std;

int main ()
{ char  klausk [] = "Iveskite savo varda: \n";
  char  atsakyk [] = "Labas, ";
  char  vardas [80];
  cout << klausk;
  cin >> vardas;
  cout << atsakyk << vardas << "!";
  return 0;
}
```

```
Iveskite savo varda:
Ona
Labas, Ona!
```

Kaip matome, šioje programoje yra paskelbti trys char tipo masyvai. Pirmieji du yra inicializuoti panaudojus eilutes, o trečias paliktas neinicializuotas. Tačiau bet kuriuo atveju turi būti nurodomas masyvo dydis. Pirmų dviejų masyvų klausk ir atsakyk dydis nustatomas pagal inicializacijai panaudotas eilutes, o trečio masyvo vardas dydis nurodytas aiškiai − 80.

Belieka pridurti, kad char tipo masyvuose saugomos simbolių sekos lengvai gali būti konvertuojamos į eilutes, t.y. string tipo objektus, priskyrimo operatoriais:

```
string  eilut;
char  masyv [] = "kazkoks tekstas";
eilut = masyv;
```

10. Rodyklės (pointers)

Lig šiol į kintamuosius žiūrėjome kaip į atminties sritis. Į jas galėjome kreiptis kintamųjų vardais. Mums nereikėjo rūpintis fiziniu mūsų duomenų išdėstymu atmintyje, o prireikus jų kreipėmės tiesiog kintamųjų vardais.

Kompiuterio atmintį galima įsivaizduoti kaip baitų seką. Kiekvienas baitas turi savo numerį, kuris vadinamas adresu. Adresai eina iš eilės: 0, 1, 2, ...

10.1. Adreso operatorius (&)

Kai tik paskelbiame kintamąjį, jam numatoma reikiamo dydžio atminties sritis (kažkiek baitų). Paleidus programą, ją automatiškai paskiria operacinė sistema. Tam tikrais atvejais mus gali dominti, nuo kurio adreso atmintyje yra patalpintas mūsų kintamasis programos vykdymo metu. To gali reikėti, norint paslinkti kintamąjį atmintyje.

Kintamojo adresui gauti naudojamas adreso operatorius (& - ampersendo ženklas), kuris rašomas kintamojo vardo priekyje. Pvz.:

```
tom = &alge;
```

Šiuo sakiniu kintamojo alge adresas priskiriamas kintamajam tom. Atkreipkime dėmesį, kad čia kalbame ne apie kintamojo alge turinį, o apie jo adresą, nes vardo priekyje yra &.

Tarkime, vykdant programą kintamasis alge atmintyje buvo patalpintas adresu 2476. Šis skaičius (2476) yra tik atsitiktinai sugalvotas, kad būtų lengviau aiškinti kai kurias idėjas. Iš tikro iki programos vykdymo pradžios mes negalime žinoti kintamojo tikrojo adreso.

Tarkime yra toks programos fragmentas:

```
alge = 25;
pran = alge;
vyt = &alge;
```

Jis duoda tokias kintamųjų reikšmes: alge – 25, pran – 25, vyt – 2476.

Pirmose dviejose šio programos fragmento eilutėse yra standartiniai priskyrimo sakiniai. Pirmuoju kintamajam alge priskiriama reikšmė 25 (jo adresas, tarėme, yra 2476). Antrajame sakinyje imama kintamojo alge reikšmė ir priskiriama kintamajam pran. Na o trečiajame sakinyje imamas kintamojo alge adresas ir jis priskiriamas kintamajam vyt.

Kintamasis, kuris gali įsiminti kito kintamojo adresą, yra vadinamas rodykle (*pointer*). Rodyklės yra labai galinga C++ kalbos priemonė, ir programuojant jos

daug kur naudojamos. Kaip šitokie kintamieji paskelbiami ir naudojami, sužinosime truputį vėliau.

10.2. Duomens nurodytu adresu operatorius (*)

Naudodami rodykles mes galime tiesiogiai kreiptis į duomenis, kurie atmintyje laikomi jose nurodytais adresais. Tam prieš rodyklės vardą turi būti rašoma žvaigždutė (*), kuri atlieka duomens ėmimo/įsiminimo nurodytu adresu operatoriaus vaidmenį.

Todėl, pratęsdami aukščiau turėtą programos fragmentą sakiniu

```
beta = *vyt;
```

gausime tokį rezultatą: kintamasis beta įgis reikšmę 25, nors vyt reikšmė yra 2476. Taip yra todėl, kad atmintyje adresu 2476 yra užrašytas skaičius 25.

Įsidėmėkime šį skirtumą: užrašas vyt atitinka reikšmę 2476 (tai adresas), o užrašas *vyt atitinka 25 (tai reikšmė, užrašyta nurodytu adresu). Todėl

```
beta = vyt;        // beta pasidaro lygi vyt   ( 2476 )
beta = *vyt;       // beta igyja duomens reikšmę  ( 25 )
```

Atkreipkime dėmesį į adreso operatoriaus (&) ir duomens nurodytu adresu operatoriaus (*) skirtumą: jie yra vienas kitą papildantys ir kartu vienas kitam priešingi. Imkime tokį atvejį:

```
a = 25;
b = a;
c = *&a;
```

Čia kintamieji a, b ir c įgyja tokią pačią reikšmę. Užrašas *&a reiškia, kad operatoriumi & pirmiausiai gaunamas kintamojo a adresas, o po to operatoriumi * paimama reikšmė, užrašyta tuo adresu. Taigi, užrašas *&a duoda tą patį kaip ir a.

Imkime dar vieną pavyzdį. Anksčiau turėjome tokius du priskyrimo sakinius:

```
alge = 25;
vyt = &alge;
```

kur tarėme, kad kintamojo alge adresas atmintyje yra 2476. Įvykdžius juos, visos žemiau parodytos palyginimo išraiškos duos rezultatą true.

```
alge == 25
&alge == 2476
vyt == 2476
*vyt == 25
```

10.3. Rodyklės tipo kintamųjų paskelbimas

Kadangi rodyklės įgalina tiesiogiai kreiptis į duomenis jose nurodytais adresais, paskelbiant jas būtina nurodyti, kokio tipo duomenys bus laikomi tais adresais. Rodyklių paskelbimo bendrasis pavidalas yra:

```
tipas *vardas;
```

Čia tipas nurodo duomenų tipą, kurie bus saugomi rodykle nurodomu adresu, o ne pačios rodyklės tipą. Pvz.:

```
int  *skaicius;
char  *simbolis;
double  *didelissk;
```

Tai trijų rodyklių aprašai. Kiekvieną iš jų ketinama naudoti skirtingo tipo duomenims nurodyti, bet visos jos yra rodyklės ir kiekviena iš jų atmintyje užims tokį patį baitų kiekį (jis priklauso nuo skaičiavimo platformos, naudojamos programai vykdyti; dažniausiai – 4 baitai). Tačiau patiems duomenims, kurių adresus nurodys šios rodyklės, atmintyje reikės skirtingo dydžio sričių: int tipo duomuo užims 4 baitus, char – 1 baitą, double – 8 baitus. Todėl ir sakoma, kad rodyklės yra skirtingo tipo.

Pastebėkime, kad žvaigždutės simbolis (*) apraše rodo tik tai, kad paskelbtas vardas yra rodyklė. Nepainiokime jo su ankstesniame skyriuje nagrinėtu duomens nurodytu adresu operatoriumi, kuris taip pat žymimas žvaigždute (*).

Panagrinėkime tokią programą:

```
// pirmoji programa su rodyklėmis
#include <iostream>
using namespace std;

int main ()
{ int  sk1, sk2;
  int  *rod;
  rod = &sk1;
  *rod = 10;
  rod = &sk2;
  *rod = 20;
  cout << "Pirmas skaicius " << sk1 << endl;
  cout << "Antras skaicius " << sk2 << endl;
  return 0;
}
```

```
Pirmas skaicius 10
Antras skaicius 20
```

Nors tiesiogiai nė vienam iš kintamųjų sk1 ir sk2 reikšmės nesuteikėme, abu jie reikšmes įgijo netiesiogiai, naudojant rodyklę rod.

Imkime sudėtingesnį pavyzdį:

```
  // programa su daugiau rodyklių
#include <iostream>
using namespace std;

int main ()
{ int sk1 = 5, sk2 = 15;
  int  *p1, *p2;

  p1 = &sk1;     //  p1 = sk1 adresas
  p2 = &sk2;     //  p2 = sk2 adresas
  *p1 = 10;      // 10 įraso adresu p1
  *p2 = *p1;    // ima reikšmę adresu p1 ir įraso adresu p2
  p1 = p2;       // kopijuoja rodyklės reikšmę
  *p1 = 20;      // 20 įrašo adresu p1
  cout << "Pirmas skaicius " << sk1 << endl;
  cout << "Antras skaicius " << sk2 << endl;
  return 0;
}
```

```
Pirmas skaicius  10
Antras skaicius  20
```

Atkreipkime dėmesį į šią eilutę:

```
int  *p1, *p2;
```

Joje yra paskelbtos dvi rodyklės. Jei rašytume

```
int  *p1, p2;
```

`p1` būtų rodyklė, o `p2` – `int` tipo kintamasis. Skelbiant rodykles, prieš kiekvieną būtina rašyti žvaigždutę (*).

10.4. Rodyklės ir masyvai

Masyvai turi glaudų ryšį su rodyklėmis. Masyvo vardas faktiškai atitinka pirmo jo elemento adresą ir jis yra traktuojamas kaip rodyklė. Tarkime, turime tokius aprašus:

```
int  skaiciai [20];
int  *p;
```

Todėl priskyrimo sakinys

```
p = skaiciai;
```

yra teisingas. Tačiau priešingas priskyrimas

```
skaiciai = p;            //  konstantai negalima priskirti kintamojo reikšmės
```

yra neleistinas, nes <u>masyvo vardas traktuojamas kaip rodyklės konstanta</u>.

Toliau pateikiamame programos pavyzdyje visos išraiškos, kuriose naudojamos rodyklės, yra teisingos:

```
    // dar apie rodykles
#include <iostream>
using namespace std;

int main ()
{ int  skaiciai [5];
  int  *p;
  p = skaiciai;  *p = 10;
  p++;  *p = 20;
  p = &skaiciai [2];  *p = 30;
  p = skaiciai + 3;  *p = 40;
  p = skaiciai;  *(p+4) = 50;
  for ( int n=0; n<5; n++ )
    cout << skaiciai [n] << ", ";
  return 0;
}
```

```
10, 20, 30, 40, 50,
```

Priskyrimo sakinyje `p = skaiciai;` užrašas `skaiciai` atitinka `&skaiciai [0]`, t. y. masyvo vardas reiškia pirmojo elemento adresą. Sakinio `p++;` rezultatas yra kito elemento adresas. Kadangi rodyklė `p` yra `int` tipo, tai adreso skaitinė reikšmė padidinama `4`, nes `int` tipo kintamieji atmintyje užima 4 baitus. Išraiškomis `skaiciai+3` ir `p+4` adresai padidinami tiek, kad gautume reikiamo masyvo elemento adresą.

Prisiminkime, kad masyvo elementas nurodomas po vardo stačiakampiuose skliaustuose `[]` rašant indeksą. Todėl šiuos skliaustus galima traktuoti kaip reikšmės ėmimo/įsiminimo nurodytu adresu operatorių, kuris, kaip žinome, žymimas žvaigždute (*). Pvz., išraiškos

```
a [5] = 0;
*(a+5) = 0;
```

yra ekvivalentiškos, jei `a` yra rodyklė arba masyvas.

10.5. Rodyklių inicializacija

Skelbiant rodykles yra galimybė aiškiai nurodyti kintamąjį, kurio adresą ji turėtų įsiminti visų pirma. Pvz.,

```
int  skaic;
int  *tom = &skaic;
```

Ekvivalentišką rezultatą gautume užrašę taip:

```
int  skaic;
int  *tom;
tom = &skaic;
```

Masyvų atveju inicializuoti rodyklę galima priskiriant konstantą.

```
char  *vera = "Labas";
```

Čia visų pirma rezervuojama atminties sritis konstantai “Labas” saugoti, o po to jos pirmo simbolio adresas priskiriamas rodyklei vera.

Rodyklė vera nurodo simbolių sekos (eilutės) pradžią ir todėl traktuojama kaip masyvas (prisiminkime, kad masyvas traktuojamas kaip rodyklės konstanta). Todėl į turimos eilutės penktą elementą galima kreiptis tokiais būdais:

```
*(vera+4)
vera [4]
```

Abiem atvejais gausime reikšmę ‘s’ (penktoji eilutės “Labas” raidė).

10.6. Rodyklių aritmetika

Aritmetinės operacijos su rodyklėmis atliekamos šiek tiek kitaip, nei su įprastais sveikaisiais skaičiais. Su jomis leistini tik sudėties ir atimties veiksmai. Tačiau ir jie atliekami kitaip, priklausomai nuo duomens tipo, kuriam rodyklė yra skirta, užimamų baitų kiekio.

Nagrinėdami pagrindinius duomenų tipus matėme (žiūr. 2.2 skyrių), kad vieni užima daugiau, kiti mažiau baitų atmintyje. Pvz., char tipo duomuo atmintyje užima 1 baitą, short int – 2 baitus, long int – 4 baitus.

Tarkime, turime tris rodykles

```
char  *simb;
short  *trump;
long  *ilg;
```

kurios yra įgiję tokias atminties adresų reikšmes atitinkamai: 1000, 2000 ir 3000. Taigi, jei rašysime

```
simb++;
trump++;
ilg++;
```

tai simb reikšmė pasidarys lygi 1001, trump – 2002, o ilg – 3004. Priežastis yra ta, kad didindami rodyklės reikšmę vienetu mes siekiame, kad ji įgytų adreso reikšmę kito iš eilės einančio tokio paties tipo elemento. Tą patį gautume, jei rašytume

```
simb = simb + 1;
trump = trump + 1;
ilg = ilg + 1;
```

Visi šie samprotavimai tinka ir atimties operacijos atveju.

Reikšmės didinimo (++) ir mažinimo (--) operatorių prioritetas yra žemesnis už duomens nurodytu adresu ėmimio/įsiminimo operatorių (*). Tačiau abu jie turi specifinių ypatybių, kai rašomi po kintamojo vardo (juk galime rašyti a++ ir ++a). Taigi, užrašas

```
*p++
```

yra ekvivalentiškas užrašui

```
*(p++)
```

Tai reiškia, kad pirma pasinaudojama esama rodyklės reikšme ir tik po to jos reikšmė yra didinama.

Atkreipkime dėmesį į tokį užrašą:

```
(*p)++
```

Šiuo atveju yra imamas duomuo nurodytu adresu, jis panaudojamas ir po to jo reikšmė didinama vienetu. Rodyklės reikšmė lieka nepakitusi.

Imkime dar vieną pavyzdį. Užrašai

```
*p++ = *q++;
```

ir

```
*p = *q;
++p;
++q;
```

yra ekvivalentiški.

Tokiais atvejais rekomenduojama naudoti skliaustus, kad išvengtume klaidų.

10.7. Rodyklių rodyklės

C++ kalboje galima naudoti rodykles, kuriose būtų saugomi adresai kitų rodyklių, skirtų konkretiems duomenims. Aprašant tokias rodykles tereikia pridėti papildomą žvaigždutę (*). Pvz.:

```
char  a,  *b,  **c;
a = 'z';
b = &a;
c = &b;
```

Čia sakiniu b = &a; sužinojome, kokiu adresu atmintyje patalpintas char tipo kintamasis a, o sakiniu c = &b; gavome atminties adresą, kur laikoma rodyklė b.

10.8. *void* rodyklės

C++ kalboje void tipo rodyklės gali įsiminti įvairaus tipo duomenų adresus. Tačiau jų naudojimas yra smarkiai apribotas. Į duomenis, kurių adresą nurodo tokios rodyklės, negali būti kreipiamasi tiesiogiai, t.y. naudojant duomenų nurodytu adresu ėmimo/įsiminimo operatorių (*). Tai yra logiška, nes šiuo atveju mes nežinome, kiek baitų užima duomuo pradedant nurodytu adresu. Todėl void tipo rodyklė prieš kreipiantis į konkretaus tipo duomenis visada turi būti pakeista į konkretaus tipo rodyklę. Tai daroma tipų suderinimo būdu.

Vienas iš void tipo rodyklių panaudojimų yra funkcijos parametro, kuris tiktų įvairaus tipo duomenims, nurodymas. Panagrinėkime tokį programos pavyzdį:

```
     //  void tipo rodyklės naudojimo pavyzdys
#include <iostream>
using namespace std;

void didink (void *duomuo, int dydis)
{ switch (dydis)
   { case sizeof(char) : (*((char*)duomuo))++; break;
     case sizeof(int)  : (*((int*)duomuo))++; break;
   }
}

 int main ()
 { char  a = 'x';
   int  b = 1602;
   didink (&a, sizeof(a) );
   didink (&b, sizeof(b) );
   cout << a << ", " << b << endl;
   return 0;
 }
```

```
y, 1603
```

Čia standartine sizeof() funkcija gaunamas jos argumento užimamos atminties srities dydis baitais. Pagrindinio tipo duomenims šie dydžiai yra konstantos. Pvz., sizeof(char) reikšmė yra 1, nes char tipo duomuo atmintyje užima 1 baitą.

Funkcijos didink() parametras duomuo, kuris yra void tipo rodyklė, prieš naudojimą išraiškomis (char*)duomuo ir (int*)duomuo pakeičiamas į reikiamo tipo rodyklę.

10.9. *null* rodyklė

null rodyklė yra tokia, kurios reikšmė yra lygi 0. Sakoma, kad tokia rodyklė dar nėra įgijusi reikšmės. Tai jokio kintamojo adresas. Pvz.:

```
int  *p;                  //   paskelbus, p būna null rodyklė
int  a;
if (p = 0) p = &a;        //  p igijo reikšmę
p = 0;                    //  p vėl tapo null rodykle
```

10.10. Funkcijų rodyklės

C++ kalboje rodyklė gali įsiminti ir funkcijos adresą. Tipiškas tokios rodyklės panaudojimo atvejis yra funkcijos vardo, kaip argumento, perdavimas kreipiantis į kitą funkciją.

Funkcijos rodyklės skelbimo bendrasis pavidalas yra toks:

tipas (*vardas) (parametro1_tipas, parametro2_tipas, ...);

Matome, kad funkcijos rodyklė skelbiama kaip funkcijos prototipas (antraštė), išskyrus tai, kad funkcijos vardas dar suskliaudžiamas ir prieš jį yra žvaigždutė (*):

```
   //  funkcijų rodyklės naudojimo pavyzdys
#include <iostream>
using namespace std;

int sudek (int a, int b)
  {  return (a+b);  }

int atimk (int a, int b)
  { return (a-b);  }

int (*minus) (int, int) = atimk;

int veiksmas ( int x, int y, int (*funkc)(int,int) )
{ int g;
   g = (*funkc)(x,y);
   return (g);
}

int main ()
{ int m,n;
  m = veiksmas (7, 5, sudek);         //  rezultatas - 12
  n = veiksmas (20, m, minus);        //  rezultatas - 8
  cout << n;
  return 0;
}
```

Rezultatas
8

Šiame pavyzdyje minus yra globalioji du int tipo parametrus turinčios funkcijos rodyklė. Jai čia iš karto suteikiama pradinė reikšmė (ji inicializuojama), kuri lygi funkcijos atimk adresui.

```
int (*minus) (int,int) = atimk;
```

Pastebėkime, kad funkcijų vardai yra traktuojami kaip rodyklės. Prisiminkime, kad panašiai yra ir su masyvų vardais.

11. Dinaminė atmintis

Lig šiol visoms mūsų parašytoms programoms vykdyti skiriamos atminties dydis žymia dalimi priklausė nuo to, kiek jos reikėjo paskelbtiems kintamiesiems. Tas atminties dydis būdavo apibrėžtas programos tekste (*source code*) dar prieš jos vykdymą. Bet ką daryti, jei kintamajam reikalingos atminties dydį įmanoma apibrėžti tik programos vykdymo eigoje? Pvz., kai visų pirma reikia įvesti kažkokius duomenis, kad galėtume apibrėžti reikiamos atminties srities dydį.

Atsakymas į šį klausimą yra toks: naudokime dinaminę atmintį. Dinamine atmintimi vadinama ta operatyviosios atminties dalis, kuri programos vykdymo metu yra laisva ir prireikus gali būti paskiriama kintamiesiems ir išlaisvinama, kai jos nebereikia. C++ kalboje tam yra operatoriai new ir delete.

11.1. Operatoriai *new* ir *new[ ]*

Dinaminė atminties sritis paskiriama operatoriumi new, kuris rašomas prieš kintamojo tipą nurodantį bazinį žodį. new grąžina naujai paskirtos atminties srities adresą, kuris priskiriamas rodyklei. Bendrasis sakinių new pavidalas:

rodyklė = new tipas;

arba

rodyklė = new tipas [kiekis];

Pirmasis naudojamas paskirti atminties sritį pavieniam nurodyto tipo elementui, o antrasis − nurodyto tipo ir nurodyto elementų kiekio masyvui. Pvz.:

```
int  *bob;
bob = new int [5];
```

Šiuo atveju sistema dinamiškai (programos eigoje) paskirs atminties sritį penkiems int tipo elementams ir jos adresą įsimins rodyklėje bob.

Į pirmąjį elementą dinamiškai paskirtoje atmintyje galima kreiptis bob[0] arba *bob, į antrąjį – bob[1] arba *(bob+1) ir t.t.

Dinaminis atminties skyrimo būdas elementų masyvui turi tą privalumą prieš įprastą masyvų skelbimą, kad čia elementų kiekis gali būti nurodomas konstanta arba kintamuoju (skelbiant masyvą, jo dydis nurodomas tik konstanta). Pvz.:

```
int  *bob;
int  a;
cin >> a;
bob = new int [a];
```

Reikiamas dinaminės atminties sritis sistema skiria laisvoje atminties dalyje (ją dar vadina *heap* atmintim). Tačiau kompiuterio atmintis yra riboto dydžio, ir ji gali būti visa užimta. Todėl svarbu turėti tikrinimo priemonę, ar prašomo dydžio atminties sritis iš tikro buvo paskirta.

C++ kalboje yra du standartiniai dinaminės atminties sėkmingo paskyrimo kontrolės būdai. Vienas iš jų yra nesėkmingų arba ypatingų situacijų (*exceptions*) apdorojimas. Negalėdama paskirti dinaminės atminties srities, sistema suformuoja bad_alloc tipo situaciją, susidarius kuriai programuotojas gali numatyti savo veiksmus. Ypatingų situacijų apdorojimo šioje mokymo priemonėje mes plačiau nenagrinėsime. Tačiau žinokime, kad programuotojui nenumačius savo veiksmų ypatingos situacijos atveju, programa nutraukiama.

Programos nutraukimas yra ypatingos situacijos, skiriant dinaminę atmintį new operatoriumi, apdorojimo numatytasis (*default*) būdas. Jis naudojamas tokiuose sakiniuose, kaip, pvz.,

```
bob = new int [5];
```

Kitas dinaminės atminties paskyrimo kontrolės būdas yra žinomas vardu nothrow. Nesėkmės atveju new operatorius grąžina adresą lygų 0 (*null*) ir programa tęsiama. Šis kontrolės būdas pradeda veikti, jei po new rašomas parametras nothrow. Pvz:

```
bob = new (nothrow) int [5];
```

Ar šiuo sakiniu dinaminė atmintis sritis nurodytam elementų kiekiui iš tikro buvo paskirta, galima įsitikinti patikrinus rodyklės bob reikšmę:

```
int  *bob;
bob = new (nothrow) int [5];
if (bob == 0)
 {    // atminties nepaskyrė. Reakcija į tai.
 };
```

11.2. Operatoriai *delete* ir *delete[ ]*

Dinaminė atminties sritis paprastai būna reikalinga tik tam tikrais programos vykdymo momentais. Kai tik jos nebereikia, turėta atminties sritis gali būti išlaisvinta ir panaudota kitiems reikalams. Šiam tikslui yra operatorius delete, kurio pavidalai:

```
delete rodyklė;
delete [] rodyklė;
```

Pirmoji išraiška naudojama pavieniam elementui, o antroji − elementų masyvui paskirtai dinaminei atminčiai išlaisvinti. Išraiškose nurodoma rodyklė anksčiau turi būti įgijusi reikšmę new operatoriumi arba ji gali turėti *null* reikšmę (esnt *null* rodyklei, operatorius delete nieko nedaro). Pavyzdys:

```
     // dinaminės atminties skyrimas ir išlaisvinimas
#include <iostream>
using namespace std;

int main ()
{ int k,n;
  int * p;
  cout << "Kiek skaiciu vesite? ";
  cin >> k;
  p = new (nothrow) int [k];     // paskiria atmintį
  if (p == 0)
    { cout << "Klaida: atmintis nepaskirta";
      exit(0);
    }
  else
   { for (n=0; n<k; n++)
       { cout << "Iveskite skaiciu: ";
         cin >> p [n];
       }
      cout << "Jus ivedete: ";
      for (n=0; n<k; n++)
      cout << p [n] << ", ";
      delete [] p;                 // išlaisvina atmintį
   }
  return 0;
}
```

```
Kiek skaiciu vesite? 4

Iveskite skaiciu: 75
Iveskite skaiciu: 436
Iveskite skaiciu: 1067
Iveskite skaiciu: 8

Jus ivedete: 75, 436,
1067, 8,
```

Atkreipkime dėmesį į sakinį

```
p = new (nothrow) int [k];
```

Jame masyvo dydis nurodytas kintamuoju k, kuriam reikšmė įvedama programos vykdymo metu. Įvedus didelę kintamojo k reikšmę, gali trūkti dinaminės atminties ir ji nebus paskirta. Tai kontroliuojama nothrow metodu.

11.3. Dinaminė atmintis ANSI-C++ standarte

C++ kalboje operatoriai new ir delete yra atsiradę neseniai. Jų nėra C kalboje. Tačiau C kalboje darbui su dinamine atmintimi yra funkcijos malloc, calloc, realloc ir free. Pastarasios yra laikomos <cstdlib> bibliotekoje. Kadangi C++ yra C kalbos viršaibis, minėtos funkcijos yra ir C++ kalboje.

```
Int  *x;  x = (int *) malloc (n * sizeof (int) );     arba
             x = new (nothrow) int [n];
```

Dinaminės atminties sričių skyrimas minėtomis funkcijomis kažkiek skiriasi nuo atminties skyrimo new operatoriumi. Todėl programuojant reikia naudoti vieną kurį nors funkcijų ar operatorių rinkinį, bet ne jų kombinaciją.

Žinotina, kad C ir C++ kalbose kai kurių standartinių bibliotekų pavadinimai skiriasi. Pvz., C++ kalbos biblioteką <cstdlib> atitinka C kalbos biblioteka <stdlib.h>.

12. Duomenų struktūros

Mes jau nagrinėjome, kaip C++ kalboje naudojami duomenų rinkiniai (pvz., masyvai), kurių visi elementai yra tokio paties tipo. Tačiau labai dažnai yra reikalingi rinkiniai, kuriuose būtų įvairaus tipo duomenys.

12.1. Duomenų struktūros

Duomenų struktūra – tai įvairaus tipo duomenų grupė, turinti vieną vardą. C++ kalboje duomenų struktūros skelbimo sintaksė yra tokia:

```
struct  struktūros_vardas
 { tipas1  elemento_vardas1;
   tipas2  elemento_vardas2;
   tipas3  elemento_vardas3;
   . . .
 }  objektų_vardai;
```

Čia struktūros_vardas – tai struktūros tipo vardas, objektų_vardai – tai objektų vardų rinkinys, kuriuos norime paskelbti kaip nurodyto tipo duomenų struktūras. Riestiniuose skliaustuose išvardinami duomenų elementai, kurių kiekvienas turi savo tipą ir vardą. Visų pirma reikėtų atkreipti dėmesį į tai, kad skelbiant struktūrą sukuriamas naujas duomenų tipas, kurio vardas bus struktūros_vardas. Toliau programoje jį galima bus naudoti kaip ir bet kurį kitą tipą. Pvz.:

```
struct  produktas
 { int  svoris;
   float  kaina;
 };

produktas  suris;
produktas  desra, zuvis;
```

Čia mes pirma paskelbėme iš dviejų elementų svoris ir kaina susidedančios struktūros tipą, kurį pavadinome produktas. Po to naudodami šį tipo vardą paskelbėme tris objektus: suris, desra ir zuvis.

Po šio paskelbimo produktas tapo nauju galiojančiu duomenų tipo vardu, kaip ir mums žinomi pagrindinių tipų vardai int, char, float ar kt. Taip mes įgijome galimybę skelbti šio naujo sudėtinio tipo objektus (kintamuosius) kaip suris, desra ir kt.

Struktūros skelbimo pabaigoje po uždarančio riestinio skliausto iš karto galima nurodyti objektus. Todėl mūsų turėtą pavyzdį galima rašyti ir taip:

```
struct  produktas
  { int svoris;
     float kaina;
  }  suris, desra, zuvis;
```

Svarbu yra skirti, kur yra struktūros tipo vardas ir kur yra tokios struktūros objektas (kintamasis). Vieno struktūros tipo objektų galime paskelbti daug.

Kadangi mes paskelbėme tris apibrėžtos struktūros tipo objektus suris, mesa ir zuvis, tai galime tiesiogiai dirbti su kiekvieno iš jų elementais. Tam tarp objekto vardo ir elemento turi būti dedamas taškas. Pvz., mes galime dirbti su bet kuriuo elementu, nes jie yra standartinio tipo kintamieji:

```
suris.svoris            desra.svoris          zuvis.svoris
suris.kaina             desra.kaina           zuvis.kaina
```

Kiekvieno šių duomenų tipas atitinka struktūros elementui nurodytą tipą. suris.svoris, desra.svoris ir zuvis.svoris yra int tipo, o suris.kaina, desra.kaina ir zuvis.kaina yra float tipo.

Išnagrinėkime pavyzdį, kuriame struktūros tipas naudojamas lygiai taip pat, kaip ir pagrindiniai duomenų tipai.

```
        // pavyzdys su struktūromis
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

struct  knyg_t
  { string  pavadinimas;
     int  metai;
  }  mano, tavo;

void spausdink (knyg_t  knyga);      //  funkcijos prototipas

int main ()
{ string  eilut;
   mano.pavadinimas = "Odisejo keliones";
   mano.metai = 1968;
     cout << "Iveskite pavadinima: ";
     getline (cin, tavo.pavadinimas);
     cout << "Iveskite metus: ";
     getline (cin, eilut);
```

```
Iveskite
pavadinima:
Pelene
Iveskite metus:
1979
```

```
    stringstream (eilut) >> tavo.metai;
    cout << "Mano megiama knyga yra:\n ";
    spausdink (mano);
    cout << "Jusu megiama knyga yra:\n ";
    spausdink (tavo);
    return 0;
 }
void spausdink (knyg_t  knyga)
 { cout << knyga.pavadinimas;
   cout << " (" << knyga.metai << ")\n";
 }
```

```
Mano megiama
knyga yra:
Odisejo keliones
(1968)
Jusu megiama
knyga yra:
Pelene  (1979)
```

Šis pavyzdys rodo, kad objektų elementai gali būti naudojami kaip paprasti kintamieji. Pvz., tavo.metai yra int tipo kintamasis, o mano.pavadinimas yra string tipo kintamasis.

Objektai mano ir tavo gali būti traktuojami kaip knyg_t tipo kintamieji. Pvz., mes juos, kaip ir įprastus kintamuosius, naudojome kreipdamiesi į funkciją spausdink. Tačiau viena iš svarbiausių duomenų struktūrų savybių yra ta, kad mes galime kreiptis į kiekvieną jos elementą individualiai arba kreiptis į visą struktūrą tik vienu vardu.

Duomenų struktūros ypatingos tuo, kad jos gali būti naudojamos duomenų bazėms pavaizduoti, ypač jei turėsime galvoje galimybę sudaryti duomenų struktūrų masyvus:

```
      // struktūrų masyvas
#include <iostream>
#include <string>
#include <sstream>
using namespace std;
#define N_KNYGU 3
struct  knyg_t
  { string  pavadinimas;
    int  metai;
  } knygos [N_KNYGU];
void spausdink (knyg_t kng);   // f-jos prototipas
int main ()
{ string eilut;
  int  n;
```

```
    for (n=0; n<N_KNYGU; n++)
      { cout << "Iveskite pavadinima: ";
        getline (cin,knygos [n].pavadinimas);
        cout << "Iveskite metus: ";
        getline (cin,eilut);
        stringstream(eilut) >> knygos [n].metai;
      }
    cout << "\n Jus ivedete siu knygu duomenis:"
           << endl;
    for (n=0; n<N_KNYGU; n++)
        spausdink (knygos [n]);
    return 0;
  }
  void spausdink (knyg_t kng)
  { cout << kng.pavadinimas;
    cout << " (" << kng.metai << ")\n";
  }
```

```
Iveskite pavadinima: Kopos
Iveskite metus: 1982
Iveskite pavadinima: Matrica
Iveskite metus: 1999
Iveskite pavadinima: Taksi vairuotojas
Iveskite metus: 1976
Jus ivedete siu knygu duomenis:
Kopos (1982)
Matrica (1999)
Taksi vairuotojas (1976)
```

12.2. Struktūrų rodyklės

Struktūrų, kaip ir kitokio tipo duomenų adresui atmintyje nurodyti gali būti naudojamos rodyklės:

```
struct  knyg_t
  { string  pavadinimas;
    int  metai;
  };
knyg_t   aknyga;
knyg_t   *pkn;
```

Čia aknyga yra knyg_t tipo objektas, o pkn yra knyg_t tipo struktūros rodyklė. Todėl sakinys

```
pkn = &aknyga;
```

yra teisingas. Rodyklei pkn bus priskirtas objekto aknyga adresas atmintyje.

Panagrinėkime programos su struktūrų rodyklėmis pavyzdį, kuris padės susipažinti su nauju operatoriumi, t.y. rodyklės operatoriumi (->):

```
     // struktūrų rodyklės
#include <iostream>
#include <string>
#include <sstream>
using namespace std;
```

```
struct  knyg_t
  { string  pavadinimas;
    int  metai;
  };

int main ()
{ string  eilut;
   knyg_t  aknyga;
   knyg_t  *pkn;

   pkn = &aknyga;
   cout << "Iveskite pavadinima: ";
   getline (cin, pkn->pavadinimas);
   cout << "Iveskite metus: ";
   getline (cin, eilut);
   stringstream(eilut) >> pkn->metai;
   cout << "\nJus ivedete:\n";
   cout << pkn->pavadinimas;
   cout << " (" << pkn->metai << ")\n";
   return 0;
}
```

```
Iveskite pavadinima:
Altoriu sesely
Iveskite metus: 1978


Jus ivedete:
Altoriu sesely  (1978)
```

Šiame pavyzdyje naudojamas rodyklės operatorius (->) yra reikšmės nurodytu adresu ėmimo/įsiminimo operatorius, kuris išimtinai naudojamas tik struktūrų elementams. Rodyklės operatoriumi kreipiamasi į elementą to objekto, kurio adresą turi įsiminusi struktūros rodyklė. Nepainiokime savokų "rodyklės operatorius" (->) ir "rodyklė" (*pointer*). Pavyzdyje mes naudojome išraišką

```
pkn->pavadinimas
```

kuri yra ekvivalentiška tokiai išraiškai:

```
(*pkn).pavadinimas
```

Šios abi išraiškos yra teisingos ir jomis kreipiamasi į elementą pavadinimas tos struktūros, kurios adresą yra įsiminusi rodyklė pkn. Būtina aiškiai skirti minėtas išraiškas nuo šių dviejų ekvivalentiškų išraiškų:

```
*pkn.pavadinimas
*(pkn.pavadinimas)
```

Pastarosios reikalauja, kad pavadinimas būtų rodyklė (o ne string, kaip esame paskelbę), ir todėl būtų kreipiamasi į duomenį, kurio adresą ji nurodytų.

Žemiau lentelėje reziumuojamos rodyklių ir struktūros elementų galimos kombinacijos:

Išraiška	Į ką kreipiamasi	Ekvivalentas
a.b	į objekto a elementą b	
a->b	į objekto, kurio adresas yra rodyklėje a, elementą b	(*a).b
*a.b	į duomenį, kurio adresas yra objekto a elemente b (elementas b turėtų būti rodyklė)	*(a.b)

12.3. Struktūros struktūrose

Viena struktūra gali būti kitos struktūros elementu. Pvz.:

```
struct  knyg_t
 { string  pavadinimas;
   int  metai;
 };

struct  draugai_t
 { string  vardas;
   string  epastas;
   knyg_t  mkn;
 }  kostas, morta;

draugai_t   *pdraug = &kostas;
```

Turint taip paskelbtas struktūras, galima bet kuri iš šių išraiškų:

```
kostas.vardas
morta.mkn.pavadinimas
kostas.mkn.metai
pdraug->mkn.metai
```

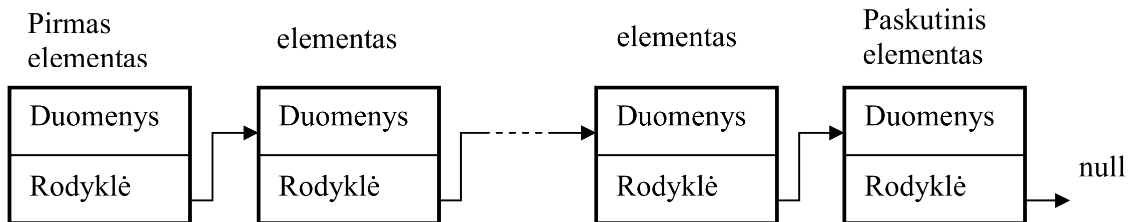
Čia paskutinės dvi išraiškos nurodo tą patį elementą.

12.4. Dinaminės duomenų struktūros: sąrašai, medžiai ir kita

Sąrašas – tai atitinkamai sudėlioti duomenys. Paprasčiausias pavyzdys yra masyvas, kuriame nuosekliai vienas po kito (vientisoje atminties srityje) yra talpinami jo elementai. Masyvo savybė yra ta, kad aprašant jį būtina nurodyti elementų kiekį (indeksų kitimo ribos nurodomos konstantomis). Tuo tarpu sąrašai neturi fiksuoto ilgio, jų elementai gali būti išmėtyti atmintyje (laikomi ne vientisoje srityje). Todėl jiems organizuoti reikia lankstesnio būdo. Lankstumą duoda dinaminiai kintamieji ir

rodyklės, įgalinantys kurti įvairias dinamines duomenų struktūras: tiesinius sąrašus, žiedinius sąrašus, eiles, stekus, medžius, piramides.

Tiesinis sąrašas. Paprasčiausio vienos krypties tiesinio sąrašo elemente turi būti du laukai: duomenys (elemento informacija) ir rodyklė (kur atmintyje laikomas kitas iš eilės sąrašo elementas). Sąrašo ilgis gali būti keičiamas įterpiant bet kurioje sąrašo vietoje naują elementą arba pašalinant bet kurį esantį. Nupieškime vienos krypties tiesinį sąrašą grafiškai.



Žinant kur yra pirmas sąrašo elementas (užfiksavus atitinkamos rodyklės reikšmę), galima susirasti bet kurį sąrašo elementą. Paskutiniame elemente esančiai rodyklei priskiriama *null* reikšmė, kaip požymis, kad sąrašas baigėsi.

Pavyzdžiui, sudarykime C++ programą gyventojų, kurių pavardė prasideda raide ‘A‘, telefonų numerių sąrašui kaupti:

```
        //  vienos krypties duomenų sąrašo sudarymas
#include <iostream>
#include <string>
using namespace std;

struct  telsar
  { string  pavarde, adresas, tnr;
    telsar  *kitorod;
  };

main()
{ telsar  *p1, *x, *y;
   p1 = 0;
    do
      { x = new telsar;
        getline (cin, (*x).pavarde);           //  galima rašyti ir  x->pavarde
        getline (cin, (*x).adresas);
        getline (cin, (*x).tnr);
        (*x).kitorod = 0;
        if (p1 == 0)  p1 = x;            // užfiksuojam pirmo elemento adresą p1
        else  (*y).kitorod = x;
        y = x;
       }
     while ( (*x).pavarde [0] == ‘A‘ );
```

```
    if (y == p1)               //  jei nėra nė vieno elemento su ‘A‘ pavarde
      { cout << “Nera ne vieno elemento“ << endl;  exit(1); }
    x = p1;                    // toliau išvedame į ekraną sąrašo elementus
    do
     { cout << (*x).pavarde << „\t“ << (*x).adresas
             << „\t“ << (*x).tnr << endl;
       x = (*x).kitorod;
      }
    while ( x != 0);
    return 0;
}
```

Sudarykime kitą programą, kuri iš sudaryto sąrašo šalina elementą su nurodyta pavarde. Pavardę įveskime klaviatūra.

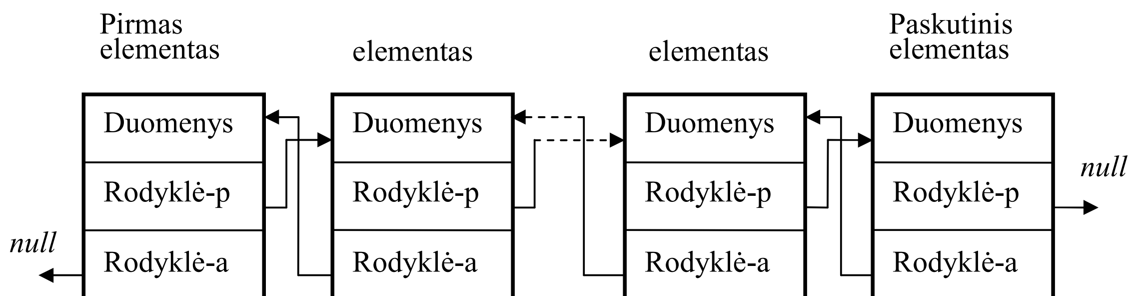
```
     // vienos krypties duomenų sąrašo elemento šalinimas
#include <iostream>
#include <string>
using namespace std;

struct  telsar
 { string  pavarde, adresas, tnr;
   telsar  *kitorod;
 };

main()
{ telsar  *p1, *x, *y;
   p1 = 0;
   string  salinti;
                //  toliau  įvedam sąrašą kaip ir prieš buvusioje programoje
   do
     { x = new telsar;
        getline (cin, (*x).pavarde);
        getline (cin, (*x).adresas);
        getline (cin, (*x).tnr);
        (*x).kitorod = 0;
        if (p1 == 0)   p1 = x;     //  p1 užfiksuojam viso sąrašo pradžios adresą
        else (*y).kitorod = x;
        y = x;
      }
    while ( (*x).pavarde [0] == ‘A‘ );
    if (y == p1)  p1=0;       // jei nėra nė vieno elemento su ‘A‘ pavarde
```

```
                    // toliau iš sukaupto sąrašo šaliname nurodytą elementą
    x = p1;    y = 0;
    if  (x == 0)
      { cout <<  “Sarase nera ne vieno elemento“ << endl;
        system (“PAUSE“);  exit(1);
      }
    cout << “Iveskite pavarde, kurio duomenis salinsite: “;
    getline (cin, salinti);            // įvedame, ka reikes šalinti
    do                                 //  ieškom šalinamojo elemento adreso
      { if ( (*x).pavarde != salinti )
           {  y = x;   x = (*x).kitorod; }
      }
    while ( (x != 0) && ( (*x).pavarde != salinti) );
    if ( (*x).pavarde == salinti )                     //   jei šalinam:
      { if ( (*x).kitorod == 0 )   (*y).kitorod = 0;   //   paskutinį
        else  if (x == p1)   p1 = (*x).kitorod;        //   pirmą
              else  (*y).kitorod = (*x).kitorod;       //   vidurinį
      }
    else  cout << “Tokios pavardes sarase nera“ << endl;
                 //    išvedame į ekraną sąrašą be jau pašalinto elemento
    x = p1;
    do
      { cout << (*x).pavarde << “\t“ << (*x).adresas << “\t“
              << (*x).tnr << endl;
        x = (*x).kitorod;
      }
    while ( x!=0);
    return 0;
}
```

Tiesinio sąrašo elemente pavartojus dvi rodykles, kurių viena būtų skirta nurodyti kitam iš eilės einančiam sąrašo elementui, o antra - prieš einančiam sąrašo elementui, turėtume dviejų krypčių tiesinį sąrašą. Pavaizduokime jį grafiškai.



Trumpai apibūdinkime kitokias dinamines duomenų struktūras:

žiedinis sąrašas yra panašus į tiesinį sąrašą, tačiau jame yra papildomas ryšys (nuoroda) tarp paskutinio ir pirmo elementų;

eilė – vienos krypties tiesinio sąrašo variantas, kuriam leistini tik du veiksmai: naujo elemento pridėjimas tik sąrašo gale ir elemento šalinimas tik nuo sąrašo pradžios;

stekas – vienos krypties tiesinio sąrašo variantas, kuriame naujas elementas gali būti pridėtas tik gale, o pašalintas taip pat tik nuo galo. Šis elementas vadinamas steko viršūne;

medis - tai pasirinktos konfigūracijos hierarchinė duomenų struktūra. Medžio elementai vadinami viršūnėmis (mazgais);

piramidė (surikiuotas medis) – tai medis, kurio viršūnių reikšmės visada nemažėja arba nedidėja pereinant į kitą lygį.

13. Kiti duomenų tipai

13.1. Apibrėžtieji duomenų tipai (*typedef*)

C++ kalba leidžia programuotojams apsibrėžti savus duomenų tipus, remiantis esamais pagrindiniais ar sudėtiniais tipais. Tai galima padaryti naudojant bazinį žodį typedef:

```
typedef  esamas_tipas  naujo_tipo_vardas;
```

Pvz.:

```
typedef  char  SIMBOLIS;
typedef  unsigned int  ZODIS;
typedef  char  *pChar;
typedef  char  laukas [50];
```

Čia mes apibrėžėme keturis duomenų tipus tokias vardais: SIMBOLIS, ZODIS, pChar ir laukas, kurie reiškia char, unsigned int, char* ir char [50] tipus atitinkamai. Juos toliau galima naudoti skelbiant kintamuosius:

```
SIMBOLIS  raid, sp, *ptc1;
ZODIS  pavad;
pChar  ptc2;
laukas  vrd;
```

Šiame pavyzdyje paskelbtas kintamasis ptc2 bus char tipo rodyklė, o kintamasis vrd bus char tipo masyvas, turintis 50 elementų.

typedf sakiniu nekuriami nauji tipai. Juo kuriami tik sinonimai esamiems tipams. Todėl, pvz., ZODIS ir unsigned int faktiškai nurodo tokį patį duomenų tipą.

typedef gali būti naudojamas duomenų tipui pavadinti slaptažodžiu, kas dažnai daroma programose. Jis taip pat gali būti naudingas, kai reikia suderinti duomenų tipus skirtingose programos versijose arba kai vietoje pernelyg ilgo arba painaus tipo norime sukurti patogesnį.

13.2. Bendrosios atminties sritys (*union*)

union baziniu žodžiu nurodoma, kad ta pati atminties sritis gali būti naudojama keliems skirtingo tipo kintamiesiems. Tokios srities skelbimo sintaksė yra panaši į duomenų struktūros (struct) skelbimą, tačiau savo esme jie yra visiškai skirtingi dalykai:

```
union  bendros_srities_vardas
  { tipas1  elemento_vardas1;
    tipas2  elemento_vardas2;
    tipas3  elemento_vardas3;
    . . .
   }  objektų_vardai;
```

Visi union sakinyje paskelbti elementai atmintyje bus talpinami toje pačioje srityje. Jos dydis yra lygus didžiausiam baitų kiekiui, kuris reikalingas kažkuriam iš įvardintų kintamųjų. Pvz.:

```
union  mantip
 { char  c;
   int  k;
   float  f;
 }  mt;
```

Apibrėžia tris elementus:

```
mt.c            mt.k            mt.f
```

kurie yra skirtingų tipų. Kadangi jų reikšmės saugomos toje pačioje atminties vietoje, tai keičiant vieno elemento reikšmę keisis ir kitų elementų reikšmės. Pastebėkime, kad mantip yra bendros atminties srities tipo vardas, o mt – šio tipo kintamojo vardas. Tokio tipo kintamųjų galime paskelbti ir daugiau, pvz.:

```
mantip  dsa, laukas;
```

Vienas iš union naudojimo pavyzdžių:

```
union  svk
{ long  k;
  struct
   { short  virs;
     short  apat;
   }  s;
  char  c [4];
}  mix;
```

Čia paskelbti trys elementai mix.k, mix.s ir mix.c, kurie naudos tą pačią 4 baitų dydžio atminties sritį. Vienu atveju juose laikysime long tipo kintamąjį, kitu – du

short tipo kintamuosius, o trečiu − char elementų masyvą. Schemiškai tai galima būtų pavaizduoti taip:

mix				
	mix.k			
	mix.virs		mix.apat	
	mix.c [0]	mix.c [1]	mix.c [2]	mix.c [3]

Čia derėtų prisiminti tai, kad įsimenant sveikuosius skaičius jų dvejetainio pavidalo jauniausios skiltys dedamos į kairėje esantį baitą, o vyriausios skiltys − į dešinėje esantį. Todėl, pvz., priskyrus mix.k = 97, tačiau išvedant mix.c [0], ekrane matysime raidę ‘a’ (raidės ‘a’ kodo skaitinė reikšmė yra lygi 97). Ir atvirkščiai, priskyrus mix.c [0] = ’a’, tačiau išvedant mix.k, ekrane gausime 97.

13.3. Bevardės bendrosios atminties sritys *union*

C++ kalboje yra galimybė skelbti bevardes union atmities sritis. Turint tokią, į jos elementus galima kreiptis tiesiog naudojant jų vardus. Pvz., atkreipkime dėmesį į šių dviejų duomenų struktūrų skelbimus:

<table>
<tr><th>Struktūra su įprastu union</th><th>Struktūra su bevardžiu union</th></tr>
<tr><td><pre>struct
 { char pavadinimas [50];
   char autorius [50];
   union
     { float litai;
       int eurai;
     } kaina;
 } knyga;</pre></td><td><pre>struct
 { char pavadinimas [50];
   char autorius [50];
   union
     { float litai;
       int eurai;
     };
 } knyga;</pre></td></tr>
</table>

Skirtumas tarp šių dviejų programos fragmentų yra tas, kad kairiajame yra nurodytas union srities vardas kaina, o dešiniajame − ne. Todėl į šių sričių elementus litai ir eurai kreipiamasi nevienodai. Kairėje esančio programos fragmento atveju kreipiamasi

```
knyga.kaina.litai
knyga.kaina.eurai
```

o dešinėje esančio fragmento atveju

```
knyga.litai
knyga.eurai
```

Prisiminkime, kad union nėra tas pat kaip struct. Elementai litai ir eurai užims fiziškai tą pačią atminties sritį. Todėl kainą galima bus nurodyti arba litais, arba eurais, bet ne abiem valiutomis iš karto.

13.4. Išvardijimai (*enum*)

Išvardijimai – tai dar vienas duomenų tipas. Jo aprašo pavidalas:

```
enum  išvardijimo_vardas
  { reikšmė1,
    reikšmė2,
    reikšmė3,
    - - -
  }  objekto_vardas;
```

Pvz., mes galime paskelbti kintamųjų tipą spalvų pavadinimams saugoti:

```
enum  spalvos { juoda, melyna, zalia, cyan, raudona, purpurine };
```

Pastebėkime, kad čia nenaudojami jokie pagrindiniai duomenų tipai. Išvardijimo tipo kintamieji skelbiami įprastu būdu:

```
spalvos  mikosp, roze, batosp;
```

Reikšmės, kurias gali įgyti tokie kintamieji yra pastovūs dydžiai, išvardinti tipo apraše tarp riestinių skliaustų. Pvz., naudojant mūsų paskelbtą tipą spalvos, galima rašyti tokius sakinius:

```
spalvos  roze;
roze = melyna;
if (roze == purpurine)  roze = raudona;
```

Išvardintųjų reikšmių tipas atitinka skaitinio tipo kintamuosius, todėl šioms reikšmėms visada priskiriama ir skaitinė reikšmė. Nesant papildomų nurodymų, pirmos reikšmės ekvivalentas yra 0, antros 1 ir t. t. Mūsų pavyzdyje išvardijimo tipe spalvos išvardintosios reikšmės įgyja tokius skaitinius atitikmenis: juoda – 0, melyna – 1, zalia – 2 ir t.t.

Bet kuriai išvardintąjai reikšmei galima aiškiai nurodyti ją atitinkantį sveikąjį skaičių. Jei po išvardintosios reikšmės sveikasis skaičius nenurodomas, tai tokios išvardintosios reikšmės skaitinis atitikmuo yra vienetu didesnis nei prieš buvusios išvardintosios reikšmės. Pvz.:

```
enum  menesiai
  { sausis=1, vasaris, kovas, balandis, geguze, birzelis,
    liepa, rugpjutis, rugsejis, spalis, lapkritis, gruodis
  }  m2007;
```

14. Klasės ir objektai

Klasė – tai išplėsta duomenų struktūra, kurioje laikomi ne tik duomenys, bet dar gali būti ir funkcijos. Objektai yra klasių atstovai. Kalbant kintamųjų terminais, klasė atitinka tipą, o objektą – kintamasis.

Klasės skelbiamos naudojant bazinį žodį class. Jų pavidalas:

```
class  klasės_vardas
  { kreipties_atributas1: narys1;          //  kreipties = prieigos
    kreipties_atributas2: narys2;
    . . .
  }  objekto_vardas;
```

Čia klasės_vardas – tai klasės pavadinimas, o objekto_vardas – tai šios klasės objektų vardų sąrašas (šioje vietoje jo gali ir nebūti). Klasės nariai – tai kintamųjų ir funkcijų aprašai, prieš kuriuos nurodomi kreipties atributai. Kreipties (prieigos) atributas nurodomas baziniu žodžiu private, public arba protected. Juo apibrėžiama klasės nario galiojimo sritis:

- į private narius gali kreiptis tik kiti tos klasės nariai arba jų *draugai* (friend, apie juos – vėliau);
- į protected narius gali kreiptis tik kiti tos klasės nariai arba jų *draugai,* o taip pat ta klase paremtų išvestinių klasių nariai;
- į public narius galima kreiptis iš bet kurios vietos, kur yra objekto galiojimo sritis.

Kai kreipties atributas nenurodomas, numatytasis yra private. Pvz.:

```
class  Cstaciakampis
  { int  x, y;
    public:
    void nurodyk_krastines (int, int);
    int  plotas (void);
  }  sta;
```

Čia apibrėžėme klasę (kitaip tariant – tipą), kurią pavadinome Cstaciakampis, ir šios klasės objektą (kitaip tariant – kintamąjį) sta. Šioje klasėje yra keturi nariai: du int tipo kintamieji (x ir y) ir dvi public kreipties funkcijos (nurodyk_krastines ir plotas). Čia nurodėme tik funkcijų prototipus (antraštes), nors galima būtų rašyti ir visą funkcijos programos tekstą. Kadangi kintamiesiems x ir y kreipties atributo nenurodėme, tai jiems galioja numatytasis private.

Į Cstaciakampis klasės sta objekto narius programoje galima kreiptis kaip į įprastus kintamuosius ar funkcijas, žinoma, prieš jų vardą dar nurodant objekto vardą. Tarp šių vardų dedamas taškas. Pvz.:

```
sta.nurodyk_krastines (3,4);
mapl = sta.plotas();
```

Tiesiogiai kreiptis į mūsų pavyzdžio narius x ir y negalėsime, nes jie yra private kreipties.

Panagrinėkime sudėtingesnį programos su Cstaciakampis klase pavyzdį:

```
        //  Programos su klase pavyzdys
#include <iostream>
using namespace std;

class Cstaciakampis
{ int  x, y;
  public:
  void nurodyk_krastines (int,int);
  int plotas ()  { return (x*y); }
};

void Cstaciakampis :: nurodyk_krastines (int a, int b)
{ x = a;
  y = b;
}

int main ()
{ Cstaciakampis  sta;
  sta.nurodyk_krastines (3,4);
  cout << "plotas: " << sta.plotas();
  return 0;
}
```

```
plotas: 12
```

Šioje programoje nauja yra tai, kad joje pavartotas priklausymo klasei operatorius (**::** - du dvitaškiai) funkcijos nurodyk_krastines apraše. Pastarasis operatorius naudojamas tada, kai klasės nario aprašas yra už klasės ribų, ir jis padeda atskirti klasėms priklausančias funkcijas nuo įprastų globalių funkcijų.

Pastbėkime, kad šioje programoje klasės Cstaciakampis viduje yra visas funkcijos plotas tekstas, nes jis labai paprastas, o funkcijos nurodyk_krastines - tik prototipas.

Priklausymo klasei operatorius (::) nurodo, kuriai klasei aprašyta funkcija priklauso. Todėl tokioje funkcijoje galima naudoti tos klasės private kreipties narius, kaip kad x ir y mūsų pavyzdyje.

Jei klasėje yra private kreipties narių, tai jiems reikšmės gali būti priskiriamos ir keičiamos tik tos klasės funkcijomis. Tokiu būdu likusiose programos vietose tiesiogiai jų reikšmių keisti negalima. Tai padeda apsaugoti tokių narių reikšmes nuo nenumatyto pakeitimo.

Didelis klasių privalumas yra tas, kad, kaip ir kitokių tipų atveju, mes galime paskelbti keletą vienos klasės objektų. Pvz., žemiau pateikiamoje programoje klasės Cstaciakampis objektų turime du: stc1 ir stc2:

```
    // viena klasė, du objektai
#include <iostream>
using namespace std;
class  Cstaciakampis

 { int x, y;
   public:
   void nurodyk_krastines (int,int);
   int plotas () {return (x*y);}
 };

void Cstaciakampis :: nurodyk_krastines (int a, int b)
 { x = a;
   y = b;
 }

int main ()
{ Cstaciakampis  stc1, stc2;
  stc1.nurodyk_krastines (3,4);
  stc2.nurodyk_krastines (5,6);
  cout << "stc1 plotas: " << stc1.plotas() << endl;
  cout << "stc2 plotas: " << stc2.plotas() << endl;
  return 0;
}
```

```
stc1 plotas: 12
stc2 plotas: 30
```

Pastebėkime, kad kreipiniai stc1.plotas() ir stc2.plotas() duoda skirtingus rezultatus. Taip yra todėl, kad kiekvienas Cstaciakampis klasės objektas operuoja su jam paskirtais kintamaisiais x ir y bei funkcijomis nurodyk_krastines ir plotas.

Tai yra pagrindinė *objektinio programavimo* idėja, kai duomenys ir funkcijos yra objekto nariai. Pastebėkime, kad kreipiniuose stc1.plotas() ir stc2.plotas() jau nebereikėjo argumentų. Šios funkcijos atitinkamai naudojo savų objektų stc1 ir stc2 duomenis.

14.1. Konstruktoriai ir destruktoriai

Objektų sudėtyje esančius kintamuosius reikia inicializuoti (priskirti jiems reikšmes), kad nebūtų operuojama su neapibrėžtomis reikšmėmis. Pvz., kas būtų atsitikę, jei prieš buvusiame mūsų pavyzdyje būtume kreipęsi į funkciją plotas anksčiau, negu į funkciją nurodyk_krastines? Tikriausiai būtume gavę neapibrėžtą rezultatą, nes nariams x ir y dar nebuvo priskirtos reikšmės.

Tokiems dalykams išvengti klasėse gali būti speciali funkcija, kuri vadinama konstruktoriumi. Į tokią funkciją kreipiamasi automatiškai, duotos klasės naujo objekto skelbimo metu. Konstruktoriaus vardas turi būti toks pat kaip ir klasės vardas, ir jis kaip funkcija negrąžina jokios reikšmės. Prieš jį nenurodomas joks tipas, netgi void.

Pakeiskime anksčiau turėtą klasę Cstaciakampis įtraukdami konstruktorių:

```
        // klasės konstruktoriaus pavyzdys
#include <iostream>
using namespace std;

class  Cstaciakampis
{  int  x, y;
    public:
    Cstaciakampis (int, int);
    int plotas () { return (x*y); }
};

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

int main ()
{ Cstaciakampis  stc1 (3,4);
   Cstaciakampis  stc2 (5,6);
   cout << "stc1 plotas: " << stc1.plotas() << endl;
   cout << "stc2 plotas: " << stc2.plotas() << endl;
   return 0;
}
```

```
stc1 plotas: 12
stc2 plotas: 30
```

Matome, kad šio pavyzdžio programa duoda tokius pačius rezultatus kaip ir ankstesnioji. Tačiau čia mes pašalinome funkciją nurodyk_krastines ir vietoje jos įtraukėme konstruktorių, kuris atlieka panašius veiksmus: inicializuoja x ir y, priskirdamas objektų stc1 ir stc2 aprašuose nurodytas reikšmes.

Atkreipkime dėmesį į tai, kaip nurodytos reikšmės perduodamos konstruktoriui skelbiant objektą:

```
 Cstaciakampis  stc1 (3,4);
Cstaciakampis  stc2 (5,6);
```

Į konstruktorių, kaip į funkciją, atvirai kreiptis negalima. Jis savo vaidmenį atlieka tik tuomet, kai skelbiamas naujas duotos klasės objektas.

Tikriausiai pastebėjote, kad prieš konstruktoriaus prototipą klasės apraše ir konstruktoriaus apraše void nevartojamas.

Destruktorius atlieka priešingą vaidmenį nei konstruktorius. Į jį kreipiamasi automatiškai, kai norima pašalinti nebereikalingą objektą (pvz., baigus darbą kuriai nors programos funkcijai, naudojančiai lokaliuosius objektus). Destruktoriaus vardas turi būti toks pat, kaip ir klasės vardas, tačiau prieš jį dar turi būti bangelės simbolis (~).

Destruktoriai ypač naudingi yra tuomet, kai objektai naudoja dinaminę atmintį. Kai paskelbtas objektas tampa nebereikalingas, objekto užimama atmitis išlaisvinama. Pvz.:

```
  //  konstruktoriaus ir destruktoriaus pavyzdys
#include <iostream>
using namespace std;

class Cstaciakampis
{  int  *x, *y;
   public:
   Cstaciakampis (int,int);
   ~Cstaciakampis ();
   int plotas () { return (*x * *y); }
};

Cstaciakampis :: Cstaciakampis (int a, int b) //konstruktorius
{ x = new int;   y = new int;
  *x = a;         *y = b;
}

Cstaciakampis :: ~Cstaciakampis ()            //  destruktorius
{ delete x;
  delete y;
}

int main ()
{ Cstaciakampis  stc1 (3,4), stc2 (5,6);
  cout << "stc1 plotas: " << stc1.plotas() << endl;
  cout << "stc2 plotas: " << stc2.plotas() << endl;
  return 0;
}
```

```
stc1 plotas: 12
stc2 plotas: 30
```

14.2. Klasės su keliais konstruktoriais

Prisiminkime, kad kelios skirtingos funkcijos gali turėti tokį patį vardą, jei skiriasi jų parametrų tipai arba kiekiai. Iš kreipinio pavidalo kompiliatorius nustato, į kurią iš jų kreipiamasi.

Konstruktorių, kurių vardas turi būti toks pat kaip ir klasės vardas, taip pat gali būti ne vienas, o keli. Skelbiant objektą bus pasirinktas tas konstruktorius, kuris atitinka skelbime nurodytus argumentus:

```
    //   pasirinktiniai konstruktoriai
#include <iostream>
using namespace std;

class Cstaciakampis
{  int  x, y;
   public:
   Cstaciakampis ();
   Cstaciakampis (int,int);
   int plotas () { return (x*y); }
};

Cstaciakampis :: Cstaciakampis ()
{ x = 5;  y = 5;
}

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

int main ()
{ Cstaciakampis  stc1 (3,4);
  Cstaciakampis  stc2;
  cout << "stc1 plotas: " << stc1.plotas() << endl;
  cout << "stc2 plotas: " << stc2.plotas() << endl;
  return 0;
}
```

```
stc1 plotas: 12
stc2 plotas: 25
```

Čia objektas stc2 paskelbtas nenurodant argumentų. Todėl jis inicializuojamas parametrų neturinčiu konstruktoriumi, kuris x ir y priskiria reikšmę 5.

Pastebėkime, kad skelbiant naują objektą ir naudojant numatytąjį (*default*) konstruktorių (pastarasis neturi parametrų), skliaustų () rašyti nereikia:

```
Cstaciakampis  stab;          //  teisingai
Cstaciakampis  stab ();       //  neteisingai!
```

14.3. Numatytasis (default) konstruktorius

Jeigu klasės apraše mes nenurodome jokio konstruktoriaus, tai kompiliatorius laiko, kad yra numatytasis (*default*) konstruktorius be parametrų. Paskelbus tokią klasę kaip ši:

```
class  Cpavyzdys
 { public:
   int a,b,c;
   void daugyba (int n, int m) { a=n; b=m; c=a*b; };
 };
```

kompiliatorius laikys, kad Cpavyzdys klasė turi numatytąjį konstruktorių. Šios klasės objektai skelbiami nenurodant jokių argumentų. Pvz.:

```
Cpavyzdys  abc;
```

Bet kai tik mes nurodome klasei jos konstruktorių, kompiliatorius daugiau nebepripažįsta numatytųjų konstruktorių. Todėl visiems mūsų skelbiamiems tos klasės objektams galios mūsų nurodytas tos klasės konstruktorius:

```
class  Cpavyzdys
 { public:
   int  a, b, c;
   Cpavyzdys (int n, int m) { a=n; b=m; };
   void daugyba () { c=a*b; };
 };
```

Čia mes paskelbėme konstruktorių su dviem int tipo parametrais. Todėl šis objekto skelbimas

```
Cpavyzdys  abc (2,3);                //  teisingas skelbimas
```

bus teisingas, o skelbimas

```
Cpavyzdys  abc;                      //  neteisingas skelbimas
```

jau bus neteisingas, nes klasė turi aiškiai nurodytą konstruktorių, kuris pakeičia numatytąjį.

Tačiau kompiliatorius sukuria ne tik numatytąjį konstruktorių, kai jo mes nenurodome, bet dar pateikia tris specialias funkcijas: *kopijuojantįjį konstruktorių*, *kopijuojantįjį priskyrimo operatorių* ir numatytąjį (*default*) destruktorių. Kopijuojantysis konstruktorius ir kopijuojantysis priskyrimo operatorius nukopijuoja

visus kito objekto duomenis ir perkelia į einamąjį objektą. Klasei Cpavyzdys kompiliatorius netiesiogiai sukurs kopijuojantįjį konstruktorių panašų į šį:

```
Cpavyzdys :: Cpavyzdys (const Cpavyzdys& rv)
 {  a=rv.a;  b=rv.b;  c=rv.c; }
```

Todėl toliau parodyti du objektų skelbimai yra teisingi:

```
Cpavyzdys  abc1 (2,3);
Cpavyzdys  abc2 (abc1);  /*  kopijuojantysis konstruktorius
                               duomenis kopijuos iš abc1      */
```

14.4. Klasių rodyklės

Kaip ir kitokių tipų atvejais (int, char, t.t.), taip ir klasėms gali būti kuriamos rodyklės. Jei jau turime paskelbę klasę, tai jos vardas gali būti naudojamas rodyklės tipui nurodyti. Pvz.:

```
Cstaciakampis *  psta;
```

psta yra rodyklė, kuri galės įsiminti Cstaciakampis klasės objekto adresą.

Kaip ir duomenų struktūrų (struct) atveju, norėdami tiesiogiai kreiptis į rodykle nurodyto objekto narį, mes galime naudoti rodyklės operatorių (->). Štai galimų kombinacijų pavyzdys:

```
    //   klasių rodyklių pavyzdys
#include <iostream>
using namespace std;

class  Cstaciakampis
{  int  x, y;
   public:
   void nurodyk_krastines (int, int);
   int plotas (void) { return (x * y); }
};

void Cstaciakampis :: nurodyk_krastines (int a, int b)
{ x = a;
  y = b;
}

int main ()
{ Cstaciakampis  a, *b, *c;
  Cstaciakampis  *d = new Cstaciakampis[2];
  b= new Cstaciakampis;
```

```
    c= &a;
    a.nurodyk_krastines (1,2);
    b->nurodyk_krastines (3,4);
    d->nurodyk_krastines (5,6);
    d [1].nurodyk_krastines (7,8);
    cout << "a plotas: " << a.plotas() << endl;
    cout << "*b plotas: " << b -> plotas() << endl;
    cout << "*c plotas: " << c -> plotas() << endl;
    cout << "d [0] plotas: " << d [0].plotas() << endl;
    cout << "d [1] plotas: " << d [1].plotas() << endl;
    return 0;
}
```

```
a plotas: 2
*b plotas: 12
*c plotas: 2
d [0] plotas: 30
d [1] plotas: 56
```

Toliau pateikiama suvestinė lentelė, kaip reikėtų suprasti kai kuriuos rodyklių ir klasių operatorius (*, &, ., ->, [ ]), panaudotus ankstesniame pavyzdyje:

išraiška	reikšmė
*x	reikšmė esanti adresu x
&x	x adresas
x.y	objektui x priklausantis narys y
x->y	objektui, kurio adresas yra rodyklėje x, priklausantis narys y
(*x).y	objektui, kurio adresas yra rodyklėje x, priklausantis narys y (ekvivalentiška išraiškai x->y)
x [0]	pirmasis objektas, kurio adresas yra rodyklėje x
x [1]	antrasis objektas, kurio adresas gaunamas remiantis rodyklėje x esančiu adresu
x [n]	(n+1)-sis objektas, kurio adresas gaunamas remiantis rodyklėje x esančiu adresu

14.5. Klasių skelbimas naudojant *struct* ir *union*

Klases galima paskelbti naudojant ne tik bazinį žodį class, bet ir bazinius žodžius struct ir union.

Klasių ir duomenų struktūrų idėjos yra tiek panašios, kad abiejų bazinių žodžių paskirtis C++ kalboje yra beveik vienoda. Skirtumas yra toks, kad paskelbus klasę struct baziniu žodžiu, jos nariai būna public kreipties, o paskelbus class baziniu žodžiu – private kreipties. Kitais požiūriais struct ir class yra ekvivalentiški.

Bendrosios atminties sričių union idėja skiriasi nuo struct ir class, nes union saugo tik vieną duomenų narį duotu metu. Tačiau union taip pat yra klasė ir jis gali saugoti narius-funkcijas. Numatytoji kreiptis į union klasės narius yra public.

14.6. *static* nariai klasėse

Klasėse bet kuris narys − duomuo (kintamasis) ar funkcija - gali turėti static požymį. Šį požymį turintys duomenys dar vadinami *klasių kintamaisiais*. Jie yra unikalūs tuo, kad jų reikšmė prieinama visiems tos klasės objektams, t.y. visiems tos klasės objektams jų reikšmė bus ta pati.

Pvz., toks kintamasis gali būti naudojamas kaip skaitliukas, kiek gi tos klasės objektų programoje buvo sukurta:

```
     //  static  nariai klasėse
#include <iostream>
using namespace std;

class  Cpypke
 { public:
    static int  n;
    Cpypke () { n++; };
    ~Cpypke () { n--; };
 };

int  Cpypke :: n = 0;

int main ()
{ Cpypke  a;
  Cpypke  b[5];
  Cpypke *  c = new Cpypke;
  cout << a.n << endl;                    7
  delete c;
  cout << Cpypke :: n << endl;            6
  return 0;
}
```

Faktiškai static nariai turi tokias pačias savybes kaip ir globalieji kintamieji, tačiau jų galiojimo sritis yra tik vienos klasės objektai. Dėl šios priežasties mes galime skelbti juos klasėje, bet be inicializavimo. static požymį turintys klasės kintamieji inicializuojami už klasės ribų esančiu formaliu apibrėžimu, kaip parodyta turėtame pavyzdyje:

```
int  Cpypke :: n=0;
```

Kadangi n yra unikalus kintamasis, galiojantis visuose tos klasės objektuose, į jį galima kreiptis dvejopai: kaip į kažkurio objekto narį arba kaip į klasės narį:

```
cout << a.n;              //  kreipiamasi kaip į objekto a narį
cout << Cpypke :: n;      //  kreipiamasi kaip į klasės Cpypke narį
```

Klasėse static požymį gali turėti ir funkcijos. Tačiau tokios funkcijos galės operuoti tik su tos klasės static kintamaisiais.

14.7. Klasių šablonai (*template*)

C++ kalboje klasių šablonai, panašiai kaip ir funkcijų šablonai (žiūr. 7.5 skyrių), įgalina pritaikyti klasę darbui su įvairaus tipo kintamaisiais. Pvz., apibrėžkime klasę, kuri gali įsiminti porą skaičių:

```
    //  klasės šablonas
template <class T>
class  pora
{   T  reiksmes[2];
   public:
     pora (T r1, T r2)
       { reiksmes[0]=r1; reiksmes[1]=r2;  }
};
```

Jei mes norime paskelbti šios klasės objektą sveikiesiems int tipo skaičiams 115 ir 36 įsiminti, turėtume rašyti taip:

```
pora <int> aobj (115, 36);
```

Ši klasė taip pat gali būti panaudota kuriant objektus ir kitokio tipo reikšmėms įsiminti:

```
pora <float> bobj (3.0, 2.18);
```

Šiame klasės šablono pavyzdyje jos nario-funkcijos programos tekstas yra klasės aprašo viduje. Jei nario-funkcijos programos tekstas būtų klasės aprašo išorėje, tuomet prieš tokios funkcijos programos tekstą reikėtų nurodyti tokį patį template < ... >, kokį užrašėme klasės šablono pradžioje:

```
      // klasių šablonai
#include <iostream>
using namespace std;
template <class T>
class  pora
{   T  a, b;
   public:
     pora (T r1, T r2)
       { a=r1; b=r2; }
    T didesnis ();
};
template <class T>
T pora<T>::didesnis()
{ T gr;
   gr = a>b? a : b;
   return gr;
}
int main ()
{ pora <int> aobj (100, 75);
  cout << aobj.didesnis();
  return 0;
}
```

```
100
```

Atkreipkime dėmesį į klasės nario-funkcijos didesnis() aprašą

```
template <class T>
T pora<T>::didesnis()
```

Jame yra net trys duomens tipo nuorodos T. Pirmasis T yra klasės šablono parametras. Antruoju T nurodomas funkcijos gražinamos reikšmės tipas. Trečiasis T, kuris apskliaustas ženklais <>, nurodo, kad šis klasės šablono parametras taip pat yra ir funkcijos šablono parametras.

15. Draugiškumas ir paveldimumas

15.1. Draugiškosios funkcijos (*friend*)

Kreipiniai į klasių private ir protected kreipties narius galimi tik tos pačios klasės funkcijose. Tačiau ši taisyklė negalioja draugiškosioms (friend) funkcijoms ir klasėms.

Išorinė duotai klasei, tačiau jai draugiškoji funkcija gali kreiptis į tos klasės private ir protected narius. Tokios funkcijos prototipas turi būti paskelbtas toje klasėje, priekyje rašant bazinį žodį friend, o pati funkcija skelbiama įprastu būdu:

```
        //  friend  funkcijos
#include <iostream>
using namespace std;

class  Cstaciakampis
{ int  x, y;
  public:
  void nurodyk_krastines (int, int);
  int plotas () { return (x * y); }
  friend  Cstaciakampis keturgubas (Cstaciakampis);
};

void  Cstaciakampis :: nurodyk_krastines (int a, int b)
{ x = a;
  y = b;
}

Cstaciakampis  keturgubas (Cstaciakampis stapar)
{ Cstaciakampis  starez;
  starez.x = 2*stapar.x;
  starez.y = 2*stapar.y;
  return (starez);
}

int main ()
{ Cstaciakampis  stc1, stc2;
  stc1.nurodyk_krastines (2,3);
  stc2 = keturgubas (stc1);
  cout << stc2.plotas();
  return 0;
}
```

```
24
```

Funkcija keturgubas yra draugiška klasei Cstaciakampis. Šioje funkcijoje mes galime kreiptis į įvairių tos klasės objektų narius x ir y, kurie yra private kreipties. Pastebėkime, kad funkcijos keturgubas skelbime ir kreipinyje į ją pagrindinėje funkcijoje (main) nėra jokių nuorodų, kad ji būtų klasės Cstaciakampis narys. Ji paprastai kreipiasi į private ir protected duomenis, nebūdama tos klasės narys.

Draugiškosios (friend) funkcijos gali praversti, pvz., atliekant operacijas su dviejų skirtingų klasių objektais. Apskritai, draugiškųjų funkcijų naudojimas išeina už objektinio programavimo metodologijos ribų, todėl, jei tik įmanoma, geriau yra naudoti tos pačios klasės narius atliekant operacijas. Mūsų turėtas programos pavyzdys būtų trumpesnis, jei funkcija keturgubas būtų klasės Cstaciakampis narys.

15.2. Draugiškosios klasės

C++ kalboje yra galimybė apibrėžti klasę, kuri būtų draugiška kitai. Taip pasiekiama, kad viena klasė gali kreiptis į kitos klasės private ir protected kreipties narius.

```
        // friend klasės
#include <iostream>
using namespace std;

class  Ckvadratas;

class  Cstaciakampis
{ int  x, y;
   public:
   int plotas () { return (x * y); }
   void konvertuok (Ckvadratas a);
};

class  Ckvadratas
{ int sonas;
   public:
   void nurodyk_sona (int a) { sonas = a; }
   friend class Cstaciakampis;
};

void Cstaciakampis :: konvertuok (Ckvadratas a)
{ x = a.sonas;
   y = a.sonas;
}
```

```
int main ()
{ Ckvadratas  kva;
  Cstaciakampis  sta;
  kva.nurodyk_sona(4);
  sta.konvertuok(kva);
  cout << sta.plotas();
  return 0;
}
```

16

Šiame pavyzdyje mes paskelbėme klasę Cstaciakampis, kuri yra draugiška klasei Ckvadratas. Todėl klasės Cstaciakampis funkcija konvertuok įgijo galimybę kreiptis į klasės Ckvadratas duomenį sonas, kuris yra numatytosios private kreipties.

Nauja šioje programoje yra dar ir tai, kad matome tuščią klasės Ckvadratas skelbimą, t.y. class Ckvadratas; Jis yra būtinas, nes skelbiamos klasės Cstaciakampis funkcijoje konvertuok() yra nuoroda į Ckvadratas, o pati klasė Ckvadratas aprašyta vėliau.

Draugiškumo tarp klasių nebus, jei to aiškiai nenurodysime. Mūsų pavyzdyje Cstaciakampis yra draugiškas klasei Ckvadratas, tačiau Ckvadratas nėra draugiškas klasei Cstaciakampis. Jeigu reikėtų, tai ir pastarąjį draugiškumą galima būtų nurodyti.

Kita draugiškumo savybė yra ta, kad jis nėra perduodamas. Draugo draugas nėra laikomas draugu, kol to aiškiai nenurodysime.

15.3. Klasių paveldimumas

Paveldimumas yra labai svarbi klasių savybė. Jis leidžia kurti klases, kurios yra išvestinės iš kitų klasių. Taip automatiškai perimami tų kitų klasių nariai, papildomai pridedant naujus. Pvz., mes norime paskelbti seriją klasių, kurios aprašo įvairius daugiakampius, kaip Cstaciakampis ar Ctrikampis. Jie tikriausiai turi bendrų savybių, pvz., gali būti aprašomi dviem dydžiais: aukštine ir pagrindu.

Tai gali būti atspindėta panaudojus klasę Cdaugiakampis, iš kurios būtų išvedamos kitos dvi klasės: Cstaciakampis ir Ctrikampis. Klasė Cdaugiakampis galėtų turėti savyje bendras abiejų daugiakampių savybes, pvz., mūsų atveju plotį ir aukštį. Tuo tarpu Cstaciakampis ir Ctrikampis galėtų būti išvestinės klasės su savo specifinėmis savybėmis, padedančiomis atskirti vieno tipo daugiakampį nuo kito.

Klasės, kurios gaunamos iš kitų klasių, vadinamų <u>bazinėmis klasėmis</u>, paveldi tų bazinių klasių narius. Tai reiškia, kad jeigu bazinė klasė turi narį A, o <u>išvestinė klasė</u> turi narį B, tai išvestinė klasė turės abu narius A ir B.

Išvestinei klasei gauti remiantis bazine, skelbime reikia rašyti dvitaškį (:) tarp šių klasių vardų:

class išvestinės_klasės_vardas **: public** bazinės_klasės_vardas
{ išvestinės_klasės_nariai }

Čia parodytas kreipties atributas public gali būti pakeistas į private ar protected. Jais nurodomas minimalus kreipties lygis, kuris bus taikomas iš bazinės klasės paveldėtiems nariams.

```
        // išvestinės klasės
#include <iostream>
using namespace std;

class  Cdaugiakampis
{ protected:
   int  x, y;          //   x – plotis,  y - aukštis
   public:
   void nurodyk_matavimus (int a, int b)
      { x=a; y=b;}
};

class Cstaciakampis : public Cdaugiakampis
{ public:
   int plotas ()
      { return ( x * y ); }
};

class Ctrikampis : public Cdaugiakampis
{  public:
    int plotas ()
       { return ( x * y / 2 ); }
};

int main ()
{ Cstaciakampis  sta;
   Ctrikampis  trk;
   sta.nurodyk_matavimus (4,5);
   trk.nurodyk_matavimus (4,5);
   cout << sta.plotas() << endl;
   cout << trk.plotas() << endl;
   return 0;
}
```

```
20
10
```

Klasių Cstaciakampis ir Ctrikampis objektai sta ir trk turi paveldėtus narius iš klasės Cdaugiakampis. Tai duomenys x, y ir funkcija nurodyk_matavimus.

Kreipties atributas protected yra panašus į private. Skirtumas tarp jų atsiranda tik paveldėjimo atveju. Kai viena klasė paveldi kitos klasės narius, išvestinės klasės nariai gali kreiptis į bazinės klasės protected narius, bet ne į private narius.

Kadangi mums reikėjo, kad x ir y būtų prieinamas ne tik bazinės klasės Cdaugiakampis nariams, bet ir išvestinių klasių Cstaciakampis ir Ctrikampis nariams, vietoje private (jis yra numatytasis, kai nieko nenurodoma) panaudojome protected.

Apibendrindami skirtingus kreipties tipus, žemiau lentelėje pateikiame, kas ir kokiu atveju gali kreiptis į klasės narį:

Kas kreipiasi	public	protected	private
tos pačios klasės narys	taip	taip	taip
išvestinės klasės narys	taip	taip	ne
ne nariai	taip	ne	ne

Šioje lentelėje “ne nariai” reiškia bet kokį kreipinį iš už klasės ribų, kaip iš main() funkcijos, kitos klasės ar funkcijos.

Mūsų pavyzdyje į narius, kuriuos paveldėjo klasės Cstaciakampis ir Ctrikampis, vienodas kreipimosi teises turi visos trys paskelbtos klasės.

```
Cdaugiakampis :: x                          // protected kreiptis
Cstaciakampis :: x                          // protected kreiptis
Cdaugiakampis :: nurodyk_matavimus()        // public kreiptis
Cstaciakampis :: nurodyk_matavimus()        // public kreiptis
```

Taip yra todėl, kad išvestinių klasių skelbimuose prieš bazinę klasę rašėme public:

```
class Cstaciakampis : public Cdaugiakampis { ... }
```

Baziniu žodžiu public, rašomu po dvitaškio (:), pažymimas minimalus kreipties lygis į bazinės klasės narius, leistinas išvestinei klasei. Kadangi public reiškia aukščiausią kreipties lygį, todėl išvestinės klasės kreipties į bazinės klasės narius lygis yra toks pat kaip ir bazinės klasės.

Jei išvestinės klasės skelbime prieš bazinę klasę nurodytume protected kreipties lygį, visi bazinės klasės public nariai išvestinei klasei pasidaro protected lygio. Analogiškai, jei nurodytume labiausiai apribojantį kreipties lygį private, tai visi bazinės klasės nariai pasidarytų private lygio.

Jei paveldint kreipties lygis prieš bazinę klasę nenurodomas, kompiliatorius laiko, kad class baziniu žodžiu paskelbtoms klasėms jis yra private, o struct baziniu žodžiu paskelbtoms – public.

15.4. Kas nepaveldima iš bazinės klasės?

Išvestinė klasė paveldi visus bazinės klasės narius, išskyrus:

- konstruktorių ir destruktorių;
- operator=() narius (apie juos šioje mokymo priemonėje nerašoma);
- draugiškuosius (friend) narius.

15.5. Daugeriopas paveldimumas

C++ kalboje įmanoma, kad viena išvestinė klasė paveldėtų keleto bazinių klasių narius. Išvestinės klasės skelbime tai nurodoma, bazines klases atskiriant kableliu. Pvz., jei mes turime kažkokią klasę duomenims išvesti į ekraną (Cisved) ir norime, kad ją paveldėtų mūsų turėtos klasės Cstaciakampis ir Ctrikampis, reikėtų rašyti:

```
class  Cstaciakampis : public Cdaugiakampis, public Cisved;
class  Ctrikampis : public Cdaugiakampis, public Cisved;
```

Toliau pateikiamas išsamesnis pavyzdys:

```
        //  daugeriopas paveldimumas
#include <iostream>
using namespace std;

class  Cdaugiakampis
{ protected:
  int  x, y;                  //   x – plotis,  y - aukštis
  public:
  void nurodyk_matavimus (int a, int b)
     { x=a; y=b;}
};

class  Cisved
{ public:
  void isved (int i);
};

void Cisved :: isved (int i)
{ cout << i << endl;
}

class  Cstaciakampis : public Cdaugiakampis, public Cisved
{ public:
  int plotas ()
     { return (x * y); }
};
```

```
class  Ctrikampis : public Cdaugiakampis, public Cisved
{ public:
   int plotas ()
      { return (x * y / 2); }
};

int main ()
{ Cstaciakampis  sta;
   Ctrikampis  trk;
   sta.nurodyk_matavimus (4,5);
   trk.nurodyk_matavimus (4,5);
   sta.isved (sta.plotas());
   trk.isved (trk.plotas());
   return 0;
}
```

16. Polimorfizmas

Skaitantieji šį skyrių turėtų būti gerai susipažinę su rodyklėmis ir klasių paveldimumu. Jei kurio nors iš žemiau pateiktų užrašų prasmė yra neaiški, peržiūrėkite nurodytus šios mokymo priemonės skyrius:

Užrašas	Žiūrėkite skyrių
int a::b(c) {};	14. Klasės ir objektai
a->b	10. Rodyklės (*pointers*)
class a : public b;	15. Draugiškumas ir paveldimumas

16.1. Bazinių klasių rodyklės

Išvestinių klasių viena iš pagrindinių savybių yra ta, kad išvestinės klasės rodyklės tipas yra suderinamas su bazinės klasės rodyklės tipu. Polimorfizmas yra pagrįstas šia paprasta, tačiau galinga ir lanksčia savybe, kuri suteikia objektiniam programavimui visą potencialą.

Toliau mes perrašysime ankstesnių šios mokymo priemonės skyrių programą apie stačiakampį ir trikampį, atsižvelgdami į minėtą rodyklių suderinamumo savybę.

```
        // bazinių klasių rodyklės
#include <iostream>
using namespace std;

class  Cdaugiakampis
{ protected:
   int x, y;              //   x – plotis,    y - aukštis
   public:
   void nurodyk_matavimus (int a, int b)
      { x=a; y=b; }
};

class  Cstaciakampis : public Cdaugiakampis
{ public:
   int plotas ()
      { return (x * y); }
};
```

```
class  Ctrikampis : public Cdaugiakampis
{ public:
   int plotas ()
      { return (x * y / 2); }
};

int main ()
{ Cstaciakampis  sta;
   Ctrikampis  trk;
   Cdaugiakampis * rodgk1 = &sta;
   Cdaugiakampis * rodgk2 = &trk;
   rodgk1->nurodyk_matavimus (4,5);
   rodgk2->nurodyk_matavimus (4,5);
   cout << sta.plotas() << endl;
   cout << trk.plotas() << endl;
   return 0;
}
```

```
20
10
```

Pagrindinėje funkcijoje main mes paskelbėme dvi Cdaugiakampis tipo rodykles rodgk1 ir rodgk2. Joms priskyrėme objektų sta ir trk adresus. Kadangi šie abu objektai yra išvestinių iš Cdaugiakampis klasių tipo, tai abu priskyrimai yra teisingi.

Vienintelis rodgk1 ir rodgk2 naudojimo vietoje sta ir trk ribotumas yra tas, kad rodgk1 ir rodgk2 yra Cdaugiakampis tipo, ir todėl šias rodykles galima naudoti tik nariams, kuriuos Cstaciakampis ir Ctrikampis yra paveldėję iš Cdaugiakampis. Dėl šios priežasties, kuomet programos pabaigoje mes kreipėmės į narius plotas(), turėjome tiesiogiai nurodyti objektus sta ir trk, o ne naudoti rodykles rodgk1 ir rodgk2.

Kad funkciją plotas() galima būtų naudoti drauge su Cdaugiakampis tipo rodyklėmis, ji turėtų būti paskelbta bazinės klasės Cdaugiakampis viduje, o ne išvestinėse klasėse Cstaciakampis ir Ctrikampis. Bet problema yra ta, kad klasėse Cstaciakampis ir Ctrikampis yra skirtingos funkcijų plotas() versijos, todėl perkelti jų į bazinę klasę negalima. Tokiais atvejais labai praverčia virtualūs klasių nariai.

16.2. Virtualūs klasių nariai

Bazinės klasės narys, kuris gali būti iš naujo apibrėžiamas išvestinėje klasėje, vadinamas virtualiu nariu. Virtualus klasės narys skelbiamas jo priekyje rašant bazinį žodį virtual:

```
      //  virtualūs nariai
#include <iostream>
using namespace std;

class  Cdaugiakampis
{ protected:
   int x, y;             //   x – plotis,   y – aukštis
   public:
   void nurodyk_matavimus (int a, int b)
      { x=a; y=b; }
    virtual int plotas ()
      { return (0); }
};

class  Cstaciakampis : public Cdaugiakampis
{ public:
   int plotas ()
      { return (x * y); }
};

class  Ctrikampis : public Cdaugiakampis
{ public:
   int plotas ()
      { return (x * y / 2); }
};

int main ()
{ Cstaciakampis sta;
   Ctrikampis  trk;
   Cdaugiakampis  dgk;
   Cdaugiakampis * rodgk1 = &sta;
   Cdaugiakampis * rodgk2 = &trk;
   Cdaugiakampis * rodgk3 = &dgk;
   rodgk1->nurodyk_matavimus (4,5);
   rodgk2->nurodyk_matavimus (4,5);
   rodgk3->nurodyk_matavimus (4,5);
   cout << rodgk1->plotas() << endl;
   cout << rodgk2->plotas() << endl;
   cout << rodgk3->plotas() << endl;
   return 0;
}
```

```
20
10
0
```

Dabar visos trys klasės (Cdaugiakampis, Cstaciakampis, Ctrikampis) turi tuos pačius narius: x, y, nurodyk_matavimus() ir plotas().

Funkcija plotas() yra paskelbta kaip virtualus bazinės klasės narys, nes vėliau ji iš naujo apibrėžiama išvestinėse klasėse. Taigi, žodis virtual įgalina kreiptis ne tik į bazinės klasės narį, bet ir į išvestinės klasės narį, kurio vardas sutampa su bazinės klasės nario vardu. Į kurios klasės narį bus kreipiamasi priklausys nuo to, kokios klasės objekto adresas priskirtas bazinės klasės rodyklei.

Klasės, kuriose yra paskelbtos arba kurios paveldi virtualias funkcijas, vadinamos polimorfinėmis klasėmis.

Nežiūrint virtualumo, mes taip pat galime skelbti klasės Cdaugiakampis tipo objektus ir kreiptis į jos nuosavą funkciją plotas(), kuri visada grąžina 0.

16.3. Abstrakčiosios bazinės klasės

Abstrakčioji bazinė klasė yra labai panaši į klasę Cdaugiakampis mūsų turėtame pavyzdyje. Vienintelis skirtumas yra tas, kad mūsų turėtame pavyzdyje yra paskelbta įprasto pavidalo virtuali funkcija plotas() ir ji atlieka minimalias funkcijas Cdaugiakampis klasės objektams (pvz., dgk), o abstrakčiosiose bazinėse klasėse virtualią funkciją plotas() galėtume rašyti be jos įgyvendinamosios dalies. Tam funkcijos skelbimo gale reikia prirašyti =0 (lygu nuliui).

Abstrakčioji bazinė klasė Cdaugiakampis atrodytų taip:

```
        // abstrakčioji bazinė klasė
class  Cdaugiakampis
{ protected:
  int x, y;
  public:
  void nurodyk_matavimus (int a, int b)
     { x=a; y=b; }
  virtual int plotas() =0;
};
```

Pastebėkime, kad po virtual int plotas() vietoje funkcijos įgyvendinamosios dalies užrašėme =0. Šitokio tipo funkcijos yra vadinamos grynosiomis virtualiomis funkcijomis, o visos klasės, turinčios bent vieną grynąją virtualią funkciją, vadinamos abstrakčiosiomis bazinėmis klasėmis.

Pagrindinis skirtumas tarp abstrakčiosios bazinės klasės ir įprastos polimorfinės klasės yra tas, kad abstrakčiojoje bazinėje klasėje bent vienas jos narys neturi įgyvendinamosios dalies, dėl ko negalima skelbti tokių klasių objektų.

Tačiau klasės, kurių objektai negalimi, nėra visiškai beprasmės. Mes galime kurti tokio tipo rodykles ir gauti naudos iš polimorfiškumo galimybių. Taigi, skelbimas pavidalo:

```
Cdaugiakampis  dgk;
```

naudojant abstrakčiąją bazinę klasę nėra leistinas, nes juo norime paskelbti objektą. Tačiau rodyklių skelbimai:

```
Cdaugiakampis * rodgk1;
Cdaugiakampis * rodgk2;
```

yra visiškai leistini.

Be to, klasė Cdaugiakampis turi grynąją virtualią funkciją ir todėl ji yra abstrakčioji bazinė klasė. Tačiau šios bazinės klasės tipo rodyklės gali būti naudojamos išvestinių klasių objektų adresams įsiminti.

Panagrinėkime sudėtingesnį pavyzdį:

```
   // abstrakčioji bazinė klasė
#include <iostream>
using namespace std;

class  Cdaugiakampis
{ protected:
   int x, y;
   public:
   void nurodyk_matavimus (int a, int b)
      { x=a; y=b; }
   virtual int plotas (void) =0;
};

class  Cstaciakampis : public Cdaugiakampis
{ public:
   int plotas (void)
      { return (x * y); }
};

class  Ctrikampis : public Cdaugiakampis
{ public:
   int plotas (void)
      { return (x * y / 2); }
};
```

```
int main ()
{ Cstaciakampis  sta;
  Ctrikampis  trk;
  Cdaugiakampis * rodgk1 = &sta;
  Cdaugiakampis * rodgk2 = &trk;
  rodgk1->nurodyk_matavimus (4,5);
  rodgk2->nurodyk_matavimus (4,5);
  cout << rodgk1->plotas() << endl;
  cout << rodgk2->plotas() << endl;
  return 0;
}
```

```
20
10
```

Pasigilinę į šią programą pamatysime, kad mes kreipėmės į skirtingų, bet susijusių klasių objektus, naudodami vieną rodyklės tipą (Cdaugiakampis*). Tai gali duoti didžiulę naudą. Pvz., dabar mes galime paskelbti abstrakčiosios bazinės klasės Cdaugiakampis naują narį-funkciją, kuri išvestų į ekraną funkcijos plotas() rezultatus, netgi jei plotas() įgyvendinamosios dalies klasėje Cdaugiakampis nėra.

```
  /* abstrakčioji bazinė klasė gali kreiptis į grynuosius virtualius narius  */
#include <iostream>
using namespace std;

class  Cdaugiakampis
{ protected:
  int  x, y;              //   x- plotis,   y – aukštis
  public:
  void nurodyk_matavimus (int a, int b)
    { x=a; y=b; }
  virtual int plotas (void) =0;
  void isved (void)
    { cout << this->plotas() << endl; }
};

class  Cstaciakampis : public Cdaugiakampis
{ public:
  int plotas (void)
    { return (x * y); }
};

class  Ctrikampis: public Cdaugiakampis
{ public:
   int plotas (void)
    { return (x * y / 2); }
};
```

```
int main ()
{ Cstaciakampis  sta;
  Ctrikampis  trk;
  Cdaugiakampis * rodgk1 = &sta;
  Cdaugiakampis * rodgk2 = &trk;
  rodgk1->nurodyk_matavimus (4,5);
  rodgk2->nurodyk_matavimus (4,5);
  rodgk1->isved();
  rodgk2->isved();
  return 0;
}
```

```
20
10
```

Bazinės klasės Cdaugiakampis naryje-funkcijoje isved() pavartotas bazinis žodis this, atitinkantis rodyklę. Jis įgyja to objekto rodyklės reikšmę, kuris yra apdorojamas.

Virtualūs nariai ir abstrakčiosios klasės suteikia C++ kalbai polimorfines charakteristikas ir daro objektinį programavimą labai naudingu instrumentu dideliuose projektuose. Žinoma, čia mes susipažinome tik su paprastu šių savybių panaudojimu. Šios savybės gali būti taikomos objektų masyvams ir dinaminiams objektams.

Pabaigai pateikiame tą patį pavyzdį, tačiau su dinaminiais objektais:

```
      // dinaminis atminties skyrimas ir polimorfizmas
#include <iostream>
using namespace std;

class  Cdaugiakampis
{ protected:
  int x, y;                // x – plotis, y - aukštis
  public:
  void nurodyk_matavimus (int a, int b)
    { x=a; y=b; }
  virtual int plotas (void) =0;
  void isved (void)
    { cout << this->plotas() << endl; }
};

class  Cstaciakampis : public Cdaugiakampis
{ public:
  int plotas (void)
    { return (x * y); }
};
```

```
class Ctrikampis: public Cdaugiakampis
{ public:
  int plotas (void)
    { return (x * y / 2); }
};

int main ()
{ Cdaugiakampis * rodgk1 = new Cstaciakampis;
  Cdaugiakampis * rodgk2 = new Ctrikampis;
  rodgk1->nurodyk_matavimus (4,5);
  rodgk2->nurodyk_matavimus (4,5);
  rodgk1->isved();
  rodgk2->isved();
  delete rodgk1;
  delete rodgk2;
  return 0;
}
```

```
20
10
```

Pastebėkime, kad rodyklės rodgk1 ir rodgk2 yra Cdaugiakampis tipo, o patys objektai yra paskelbti dinamiškai, nurodant tiesiog išvestinių klasių tipus.

17. Dar apie duomenų failų įvedimą/išvedimą

17.1. Įvedimo/išvedimo funkcijos *read()* ir *write()*

C++ kalboje bibliotekiniame <fstream> faile laikomose standartinėse ofstream ir ifstream klasėse yra nariai-funkcijos write() ir read(), skirtos duomenų mainams su failais bitų lygiu (įvedimo/išvedimo metu jokia duomenų transformacija neatliekama).

Pavyzdys. Pateikiama programa užrašo į failą duotos struktūros Studentas duomenis, po to skaito juos iš sukurto failo ir išveda į ekraną. Funkcijose write() ir read() užrašas (char*) nurodo kompiliatoriui, kad parametro rodyklės reikšmė, pvz., &B, gali nurodyti bet kokio tipo duomenims skirtą adresą.

```
      // write() ir read() funkcijų naudojimo pavyzdys
#include <iostream>     //  iš šios bibliotekos imami cin, cout
#include <fstream>  // iš šios bibliotekos imami ifstream, ofstream, write, read
using namespace std;

struct Studentas { char pav [20]; int metai; float ugis; };
int  main()
{ Studentas  A;
  Studentas  B = {"Pavardenis", 21, 187.5};
  cout << B.pav << "  " << B.metai << "  "
        << B.ugis << endl;
  ofstream  R ("c:\\katalog\\duom1.d");
  ifstream  D ("c:\\katalog\\duom1.d");
  R.write ( (char *)&B, sizeof(Studentas) );      // rašo į failą  duom1.d
  R.close();
  D.read ( (char*)&A, sizeof(Studentas));       // skaito iš failo  duom1.d
  cout << A.pav << "  " << A.metai << "  "
        << A.ugis << endl;
  D.close();
  return 0;
}
```

write() ir read() sakiniai iš atminties nurodytu adresu perduoda nurodytą baitų kiekį, t.y. tiek, kiek užima struktūra Studentas. Faile c:\katalog\duom1.d užrašytų duomenų vizualiai pažiūrėti negalėsime, nes jie ten užrašyti dvejetainiu pavidalu.

17.2. Įvedimo/išvedimo funkcijos *fscanf(), fprintf()* ir kitos

Apdorojant didesnius duomenų kiekius, patogu juos laikyti failuose (pvz., diskuose). Failo tipo rodyklės aprašomos taip:

```
FILE  *vardas;
```

Failai paruošiami darbui funkcija (ji laikoma bibliotekoje <cstdio>), kurios prototipas yra:

```
FILE  *fopen (const char *Failo_Vardas, const char *M);
```

Failo_Vardas nurodomas simbolių eilute, vardine konstanta arba kintamuoju. Antruoju parametru M (taip pat simbolių eilute, vardine konstanta arba kintamuoju) nurodomas failo stovis, t.y. leistina failo duomenų perdavimo kryptis. Galimi failų stovio požymiai parodyti lentelėje:

Požymis	Failo stovis
r	Failą galima tik skaityti
w	Failas skirtas tik rašymui. Jei toks failas jau buvo, tai jis ištrinamas ir kuriamas naujas.
a	Papildyti failą duomenimis, rašant juos failo gale. Jei nurodyto failo dar nebuvo, jis sukuriamas.

Stovio požymį papildžius raide t (rašant rt, wt, at), failas bus paruoštas darbui kaip tekstinis. Stovio reikšmę papildžius raide b (rašant rb, wb, ab), failas bus paruoštas darbui kaip binarinis. Jeigu nėra nei t, nei b raidės, failo parengimas darbui priklauso nuo globaliojo kintamojo fmode reikšmės (žr. failą fcntl.h). Kiekviena stovio nuoroda gali būti papildyta ženklu + , pvz., r+, w+, wt+. Tai reiškia, kad failai gali būti atnaujinami, t.y. leidžiama juos skaityti ir rašyti.

Žemiau lentelėje pateikiame pagrindinių buferizuotų įvedimo/išvedimo funkcijų prototipus (antraštes). Jos yra laikomos cstdio bibliotekoje.

Funkcijos prototipas (antraštė)	Funkcijos atliekami veiksmai
int **printf** (const char *f [,kintamieji]);	Išveda į ekraną kintamųjų reikšmes pagal šabloną f ;
int **scanf** (const char *f [,kint._atminties_laukai]);	Įveda klaviatūra reikšmes į kintamųjų atminties laukus pagal šabloną f ;
int **fclose** (FILE *rod);	Uždaro failą
int **putc** (int simbolis, FILE *rod);	Išveda simbolį į failą rod
int **getc** (FILE *rod);	Įveda simbolį iš failo rod

int **putw** (int sk, FILE *rod);	Išveda skaičių į failą rod
int **getw** (FILE *rod);	Įveda skaičių iš failo rod
char ***fgets** (char*s, int n, FILE *rod);	Įveda n simbolių ilgio eilutę s iš failo rod
int **fputs** (const char *s, FILE *rod);	Išveda eilutę s į failą rod
int **fread** (void *buf, int t, int n, FILE *rod);	Įveda n blokų, turinčių po t baitų, iš failo rod į buferį buf; funkcijos fread reikšmė - įvestų blokų skaičius
int **fwrite** (void *buf, int t, int n, FILE *rod);	Išveda n blokų, turinčių po t baitų, iš buferio buf į failą rod; funkcijos fwrite reikšmė - išvestų blokų skaičius
int **fprintf** (FILE *rod, const char *f [,sąrašas]);	Išveda į failą rod išvardinto sąrašo elementų reikšmes pagal šabloną f ; šablonas f ir kintamųjų sąrašas sudaromi taip pat, kaip ir funkcijoje printf
int **fscanf** (FILE *rod, const char *f [,sąrašas]);	Įveda sąrašo elementų reikšmes iš failo rod pagal šabloną f; šablonas f ir elementų sąrašas sudaromi taip pat, kaip ir funkcijoje scanf. Funkcijos fscanf reikšmė – įvestų sąrašo elementų reikšmės
int **feof** (FILE *rod);	Tikrina skaitomo failo rod pabaigą. Jei dar ne failo pabaiga, funkcijos reikšmė nelygi nuliui
int **ferror** (FILE *rod);	Tikrina darbo su failu rod klaidas. Jei klaidų nebuvo, funkcijos reikšmė lygi nuliui
int **remove** (const char *vardas);	Šalina failą vardas

Programai apdorojamų failų vardai gali būti perduodami per pagrindinės funkcijos main parametrus. Tokios pagrindinės funkcijos prototipas yra:

```
int main ( int n, char *argv [] );
```

Čia n − argumentų skaičius, o argv − argumentams skirto eilučių masyvo rodyklė. Argumentų reikšmės yra nurodomos programos iškvietimo komandoje ir yra atskiriamos tarpais. Nulinę indekso reikšmę turinčioje argumentų masyvo eilutėje užrašomas pačios programos pavadinimas.

1 pavyzdys. Tarkime, turime skaičių failą c:\ktlg\skc.txt. Reikia įvesti skaičius iš šio failo, užrašyti juos į programoje paskelbtą masyvą ir išvesti į ekraną. Toliau - apskaičiuoti masyvo elementų sumą ir išvesti ją į ekraną. Skaičiai faile yra žmogui perskaitomo tekstinio (simbolių) pavidalo.

```
#include <cstdio>  // iš šios bibliotekos ims FILE, fscanf, fopen, foef, printf
#include <cstdlib>        // iš šios bibliotekos ims  system
using namespace std;

const  int Kiek = 15;       // šia konstanta apribojam apdorojamų skaičių kiekį
typedef  int mas [Kiek];
FILE  *Fskc;

void  Sp (mas, int);          //  masyvo išvedimo į ekraną funkcijos prototipas
int  Suma (mas, int);       // masyvo elementų sumavimo funkcijos prototipas

int  main()
{ mas  A;
  int  n = 0;
  Fskc = fopen("c:\\ktlg\\skc.txt", "r");
  if (Fskc == NULL)  { printf("Failas neatidarytas"); exit (); }
  else
    { while ( !feof(Fskc) && ( n < Kiek) )
        { fscanf ( Fskc, "%d", &A [n] );         // įveda iš failo Fskc
          if ( !feof(Fskc) )  n++;
        }
      Sp(A, n);
      printf ("\n \n  Suma = %d \n ", Suma(A, n));
    }
  system ("PAUSE");       //  stabdo programą jos rezultatams ekrane pažiūrėti
  return 0;
}

void  Sp(mas X, int n)             // masyvo išvedimo į ekraną funkcija
{ int i = 0;
  while (i<n)   printf("  %d", X [i++]);
}
```

```
int  Suma(mas R, int k)           // masyvo elementų sumavimo funkcija
{ int i = 0, S = 0;
  while (i<k)    S += R [i++];
  return S;
}
```

2 pavyzdys. Faile c:\ktlg\pradf.txt yra užrašyti skaičiai. Reikia skaičius iš šio failo įvesti į programoje paskelbtą masyvą. Po to masyvo elementų reikšmes išvesti į failą c:\ktlg\kitf.txt .

```
#include <cstdio>
#include <cstdlib>                      //  iš šios bibliotekos ims system
using namespace std;

const  int Kiek = 15;
const  char finved[] = "c:\\ktlg\\pradf.txt";     //  failas ivedimui
const  char fisved[] = "c:\\ktlg\\kitf.txt";      //  failas išvedimui
typedef  int mas [Kiek];

void  SpF (mas, int, char);        //  išvedimo į failą funkcijos prototipas
int  Ivesti (mas, int *);          //  įvedimo iš failo funkcijos prototipas

int  main()
{ mas A;
  int n = 0;
  if ( Ivesti(A, &n) )
    { printf ("\n** Failo neatidare arba nera duomenu\n");
      system ("PAUSE");  exit(1);         //  exit()  nutraukia programą
    }
  SpF (A, n, 'A');
  printf("Masyvas uzrasytas i faila %s\n", fisved);
  system ("PAUSE");       // stabdo programą jos eigos pranešimams pažiūrėti
 return(0);
}

int  Ivesti (mas A, int *n)                    //  įvedimo iš failo funkcija
{ FILE *Fin;
  Fin = fopen(finved, "r");
  if ( Fin == NULL )
    { printf ( " ** Failas neatidarytas" );    return 1; }
```

```
  else while ( !feof(Fin) && (*n < Kiek) )
            { fscanf(Fin, "%d", &A [*n]);
              if (!feof(Fin))  (*n)++;
            }
  fclose (Fin);
  if ( (*n) == 0 )
    { printf ( " ** Faile nera duomenu" ); return 1; }
  return 0;
}

void  SpF (mas X, int n, char R)          //  išvedimo į failą funkcija
{ FILE *Fis;
  int  i = 0;
  Fis = fopen (fisved, "w");
  if (Fis == NULL)
    {  printf ( "Failas neatidarytas isvedimui" ); exit(1);  }
  else
    { while (i < n)
        fprintf (Fis, "%c[%2d]=%4d\n", R, i, X [i++]);
        fclose (Fis);
    }
}
```

Tarkime, faile c:\ktlg\pradf.txt buvo užrašyti tokie skaičiai:	Į failą c:\ktlg\kitf.txt ši programa išves tokius duomenis:
5 6 98 -5	A[1] = 5 A[2] = 6 A[3] = 98 A[4] = -5

Šaltiniai

[Sou06] Juan Soulie. C++ Language Tutorial. 2006. http://www.cplusplus.com/doc/tutorial/

[Vid02] A. Vidžiūnas. C++ ir C++ Builder pradmenys. Smaltijos leidykla, 2002.

[BBK+01] J.Blonskis, V.Bukšnaitis, J.Končienė, D.Rubliauskas. C++ praktikumas. Kauno technologijos universitetas. Praktinės informatikos katedra. Kaunas, 2001.

Priedas. C++ standartinės bibliotekos.

Tarptautinis ANSI-C++ standartas nustato C++ kalbos standartinių bibliotekų failų (*header files*) vardų sudarymo taisykles, kurios kažkiek skiriasi nuo anksčiau naudotų C kalboje:

- standartinių bibliotekų failų varduose nebenaudojamas plėtinys .h, kurį turėjo C kalbos standartinių bibliotekų failai, pvz., <stdio.h>, <stdlib.h>;
- iš C kalbos perimtų standartinių bibliotekų failų vardų priekyje rašoma raidė c, kad juos galėtume atskirti nuo grynai C++ kalbos standartinių bibliotekų failų vardų. Pvz., vietoje buvusio <stdio.h> dabar rašoma <cstdio>;
- visi standartinėse bibliotekose esančių klasių ir funkcijų elementai apibrėžti namespace std vardų srityje ir todėl nėra globalūs. Tai netaikoma C kalbos makrosams.

C++ kalboje yra tokie nauji standartinių bibliotekų failai:

<algorithm>, <bitset>, <deque>, <exception>, <fstream>, <functional>, <iomanip>, <ios>, <iosfwd>, <iostream>, <istream>, <iterator>, <limits>, <list>, <locale>, <map>, <memory>, <new>, <numeric>, <ostream>, <queue>, <set>, <sstream>, <stack>, <stdexcept>, <streambuf>, <string>, <typeinfo>, <utility>, <valarray>, <vector>.

Standartinių bibliotekų failai, perimti iš C kalbos:

Failo vardas. ANSI-C++ standartas	Failo vardas. ANSI-C standartas
<cassert>	<assert.h>
<cctype>	<ctype.h>
<cerrno>	<errno.h>
<cfloat>	<float.h>
<ciso646>	<iso646.h>
<climits>	<limits.h>
<clocale>	<locale.h>
<cmath>	<math.h>
<csetjmp>	<setjmp.h>
<csignal>	<signal.h>
<cstdarg>	<stdarg.h>
<cstddef>	<stddef.h>
<cstdio>	<stdio.h>
<cstdlib>	<stdlib.h>
<cstring>	<string.h>
<ctime>	<time.h>
<cwchar>	<wchar.h>
<cwtype>	<wtype.h>