Vilniaus universitetas
Fizikos fakultetas
Radiofizikos katedra

R. Grigalaitis

Programavimas

(Programavimo C++ kalba paskaitų konspektas)

Vilnius 2010

Turinys

Programavimo samprata

Žodis programa gali būti vartojamas dvejopa prasme: jis gali reikšti tam tikrą programuotojo parašytų instrukcijų rinkinį (kitaip išeities kodą) arba vykdomąją programą ar jos dalį. Išeities kodas gali būti paverstas vykdomąja programa naudojant interpretatorių, paverčiantį šį kodą kompiuteriui suprantamomis instrukcijomis, kurias jis tuoj pat ir įvykdo arba kompiliatoriumi, transliuojančiu išeities kodą į vykdomąją programą, kuri gali būti paleista vykdyti vėliau. Pagrindinis interpretatorių privalumas – jais paprasta naudotis, tačiau kompiliuotos programos veikia daug greičiau.

Paprastai programuotojai rašo programas kokiai nors konkrečiai problemai spręsti. Prieš keliasdešimt metų jos buvo rašomos norint apdoroti didelius duomenų kiekius. Žmonės, rašantys programas bei jas naudojantys buvo kompiuterių profesionalai. Vėliau kompiuteriais pradėjo naudotis vis daugiau žmonių, kur kas mažiau išmanančių kaip jie veikia, norinčių spręsti savo verslo ar kitas problemas, o ne aiškintis kompiuterių veikimo subtilybes. Ironiška, tačiau „draugiškos vartotojui“ programos tapo kur kas sudėtingesnės. Jose gausu įvairių langų, meniu ir t.t. Keičiantis reikalavimams rašomoms programoms evoliucionavo ir programavimo kalbos bei priemonės joms kurti.

Programavimo strategijos

Dar pakankamai neseniai programos buvo procedūrų, veikiančių kokius nors duomenis, seka. Procedūra ar funkcija yra tam tikrų instrukcijų, vykdomų viena po kitos, grupė. Duomenys būdavo atskirti nuo funkcijų ir pagrindinis programavimo būdas buvo kviesti vienas funkcijas, iš jų kitas ir t.t. Kadangi sudėtingoms užduotims spręsti šia būdas buvo pakankamai painus, buvo pereita prie struktūrinio programavimo. Jo pagrindinis principas remiasi paprasta ir seniai žinoma „skaldyk ir valdyk“ idėja. Kiekviena užduotis, kuri yra pakankamai sudėtinga, kad ją būtų pakankamai lengva aprašyti, skaidoma į paprastesnių užduočių grupę tol kol pastarosios tampa pakankamai paprastos ir lengvai suprantamos. Struktūrinis programavimas buvo labai sėkmingas kompleksinių problemų sprendimo būdas, tačiau, bėgant laikui, ėmė ryškėti jo trūkumai. Pirmiausia, yra įprasta galvoti apie turimus duomenis (pvz. studentų sąrašą) ir apie tai, ką su jais galima padaryti (pvz. redaguoti, surūšiuoti), kaip apie susijusius dalykus. Be to programuotojai laikas nuo laiko pradėjo „išradinėti dviratį“ – siūlydavo vis naujus sprendimus senoms

problemoms spręsti. Sprendimas šioms problemoms buvo programose panaudoti žinomų savybių komponentus. Šis modelis buvo „nusižiūrėtas“ iš techninės įrangos kūrimo srities: jei inžinieriui reikia tranzistoriaus, jis paprastai nekuria naujo, o pasirenka iš „didelės pintinės“ jau egzistuojančių tokį, kurio savybės yra tinkamos. Objektinio programavimo esmė ir yra susieti duomenis bei juos veikiančias procedūras į vieną komponentą, vadinamą objektu. Toks programavimo būdas suteikia naujas interaktyvumo galimybes. Šiuolaikinės programos yra įvykiais valdomos programos, t.y. jos reaguoja į vartotojo daromus veiksmus (pvz. mygtuko paspaudimą) naudodamos objektus šiems įvykiams apdoroti.

C++ kalbos kilmė ir raida

C++ kalba išsivystė iš C kalbos, kuri savo ruožtu kilo iš BCPL ir B programavimo kalbų. Pastarosios dvi kalbos buvo skirtos operacinių sistemų ir kompiliatorių programoms kurti. C kalba, sukurta 1972 m. panaudojo ne tik geriausias BCPL ir B kalbų savybes, bet buvo papildyta ir aukšto lygmens programavimo idėjomis. Pradžioje C kalba beveik išimtinai buvo naudojama UNIX operacinės sistemos kūrimui bei tobulinimui. Vėliau, dėl savo lankstumo – gebėjimo susieti aukšto lygmens programavimo struktūras su žemo lygmens asemblerio lygio programiniu kodu – ji paplito ir kaip kitų operacinių sistemų kūrimo priemonė. Vienas iš C kalbos privalumų yra tas, kad ji nėra susieta su jokia operacine sistema ar kompiuterio architektūra.

C++ kalba atsirado praėjusio amžiaus 8-jame dešimtmetyje – tuomet, kai pradėjo plisti objektinio programavimo idėjos. Jos kūrėjas Bjarne Stroustrup iš Bell laboratories papildė C kalbą objektinio programavimo priemonėmis. Pirmasis komercinis C++ leidinys buvo išleistas 1985 m., o 1989 m. patvirtintas ir pirmas ANSI C kalbos standartas, ženkliai įtakojęs C++ vystymąsi. C++ išlaikė beveik visišką suderinamumą su C kalba, todėl vadinama C kalbos viršaibiu. 1998 m. buvo sukurtas C++ kalbos standartas, papildytas 2003m.

Programos rašymas

C++, kaip ir daugelyje kitų programavimo kalbų, svarbu suprojektuoti programą prieš pradedant ją rašyti. Paprastos užduotys nereikalauja rimto projektavimo, tačiau sudėtigoms, t.y. tokioms, kurias sprendžia profesionalūs programuotojai, geras projektas sutaupo daug laiko bei pinigų. Be to taip galima išvengti daugelio klaidų. Pirmiausia reikia iškelti klausimą: „kokią problemą reikia išspręsti?“. Kitas svarbus

klausimas yra: „ar šios problemos sprendimas dar nėra sukurtas?“. Jau egzistuojančios programos panaudojimas dažnai sutaupo laiko.

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 klaviaturą ir ekraną. Visi C++ kompiliatoriai gali apdoroti konsolės tipo programas. Vartotojams tereikia pasiskaityti darbo su kompiliatoriumi vadovą (user‘s manual). Yra daug komercinių, o taip pat ir 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). Ja galima rasti adresu:

http://www.bloodshed.net/dev/devcpp.html.

Failai, kuriami teksto rengykle, vadinami išeities failais. Tai tekstiniai failai, praktiškai visuose C++ kompiliatoriuose įvardijami naudojant .cpp, .cp ar .c plėtinius. Tai nėra programos, todėl norint „priversti“ juos veikti, reikia sukompiliuoti. Kompiliatorius sugeneruoja objektinį failą, kurio plėtinys paprastai būna *.obj. Tai vis dar ne vykdomoji programa ir norint ją gauti naudojama ryšių redagavimo programa (angl. linker), sukomponuojanti objektinį failą su reikalingomis standartinėmis bibliotekomis.

Pirmoji C++ programa

Tradiciškai pirmoji programa išveda į ekraną žodžius „sveikas pasauli“ ar pan. Jos išeities kodas atrodo taip:

```
#include <iostream.h>

int main()
{
        cout <<"Sveikas pasauli!\n";
        return 0;
}
```

Sukompiliavus ir paleidus šią programą į ekraną ji išves žodžius „Sveikas pasauli!“. Panagrinėkime ją detaliau. Pirmoji eilutė `#include <iostream.h>` yra nuoroda išankstiniam procesoriui (angl. preprocessor) įtraukti į programą iostream.h failą. Visos nuorodos išankstiniam procesoriui prasideda # simboliu, po kurio seka žodis, nurodantis, ką reikia atlikti. Programos pradžia yra funkcija `int main()`. Kiekviena programa turi šią funkciją. Aplamai funkcija yra programinio kodo blokas, atliekantis tam tikras užduotis. Paprastai funkcijos yra kviečiamos iš kitų funkcijų, tačiau main() yra

išimtis, nes paleidus programą ji iškviečiama automatiškai. Visos funkcijos prasideda atidarančiais riestiniais skliaustais „{“ ir baigiasi uždrančiais „}“. Viskas, kas parašyta tarp šių skliaustų, priklauso funkcijai. Bene svarbiausia programos eilutė yra `cout <<"Sveikas pasauli!\n";`. Ši eilutė yra C++ kalbos sakinys (angl. statement). Sakinys – tai paprasta arba sudėtinga išraiška, kuria nurodomi konkretūs veiksmai, kuriuos reikia atlikti. C++ kalboje visi sakiniai baigiami kabliataškiu. „cout“ yra objektas skirtas informacijai į ekraną išvesti, o „<<“ – išvedimo operatorius. Šiuo atveju į ekraną išvedama simbolių eilutė „Sveikas pasauli!“ (be kabučių) ir perkėlimo į kitą eilutę simbolis „\n“. Norint išvesti simbolių eilutę į ekraną reikia nepamiršti jos apgaubti dvigubomis kabutėmis. Paskutinis sakinys `return 0;` parodo, kad funkcija main() baigia darbą ir grąžina vertę, lygią nuliui. Iš tiesų kiekviena iškviesta funkcija privalo grąžinti kokią nors vertę išskyrus tą atvejį, kai ji yra tuščio t.y. „void“ tipo. Funkcijos tipas nurodomas specialiu žodeliu prieš jos pavadinimą (šiuo atveju tai yra „int“, žymintis sveikųjų skaičių aibę).

Ką tik išnagrinėta programa nėra parašyta visai korektiškai. Paleidus ją veikti pvz. iš „WINDOWS“ operacinės sistemos bus sukurtas „DOS“ pavidalo langas, išvestas jau minėtas pranešimas į jo ekraną ir programa baigsis, t.y. langas bus uždarytas, taigi pranešimo perskaityti nepavyks. Norint, kad taip neatsitiktų, programą reikia modifikuoti. Viena iš galimų modifikacijų yra tokia:

```
#include <iostream.h>

int main()
{
        cout <<"Sveikas pasauli!\n";
        int skaicius;
        cin>>skaicius;
        return 0;
}
```

Čia sakiniu `int skaicius;` sukuriamas sveikojo tipo kintamasis „skaicius“, o sekančioje eilutėje objektas „cin“ ir įvedimo iš klaviatūros operatorius „>>“ naudojami, norint priskirti šiam kintamajam klaviatūra įvesto sveikojo skaičiaus vertę. Programa šiuo atveju laukia vartotojo veiksmų, todėl jos langas išlieka monitoriaus ekrane tol, kol bus įvestas skaičius ir paspaustas „ENTER“ klavišas.

Ta pati programa, sugeneruota „DevC++“ kompiliatoriuje, jo meniu pasirinkus „File->New->Project->Console application“ bei pridėjus sakinį `cout<<"Sveikas pasauli!\n";` būtų tokia:

```
#include <cstdlib>
#include <iostream>
```

```
using namespace std;

int main(int argc, char *argv[])
{
   cout<<"Sveikas pasauli!\n";
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Čia sakinys `using namespace std;` reikalingas tam, kad galėtume panaudoti „cout“ objektą, aprašytą „iostream“ faile, įtrauktame į „std“ vardų erdvę. Programos darbas stabdomas `system("PAUSE");` sakiniu, išvedančiu į ekraną užrašą „Press any key to continue“. Vietoj nulio `return EXIT_SUCCESS;` sakinyje naudojama konstanta `EXIT_SUCCESS`, kurios vertė lygi tam pačiam nuliui.

Komentarai programoje

Rašant programą visada aišku ir savaime suprantama, ką norima padaryti. Praėjus kuriam laikui parašyta programa dažniausia tampa nebe tokia aiški. Norint išvengti painiavos ir sumaišties programose bei padėti kitiems geriau jas suprasti, programose rašomi komentarai. Komentarai yra paprasčiausi teksto, kurį kompiliatorius ignoruoja, intarpai. Komentarai būna dviejų tipų: du priekin pasvirę brūkšniai „//“ arba pasvirusio brūkšnio ir žvaigždutės kombinacija „/* */“. Pirmuoju atveju kompiliatoriui nurodoma ignoruoti tekstą, esantį už dvigubo brūkšnio iki eilutės pabaigos. Šiuo atveju išvedimo į ekraną sakinys su komentaru galėtų atrodyti taip: `cout<<"Sveikas pasauli!\n"; //į ekraną išvedamas pranešimas`. Antruoju atveju ignoruojama visas tekstas už „/*“ simbolių tol, kol sutinkama „*/“ simbolių grupė. Šiuo būdu rašomas komentaras gali užimti ne vieną eilutę. Bendros taisyklės rašant komentarus būtų tokios: kiekviena programa turi turėti komentarą jos pradžioje, paaiškinantį, ką ji daro; kiekviena funkcija turi turėti komentarą, parodantį, ką ji skaičiuoja, kokią reikšmę ji grąžina ir kokie parametrai jai turi būti perduodami; sudėtingos išraiškos taip pat turi būti pakomentuotos. Vengti reikia komentarų, teigiančių tai, kas ir taip yra aišku.

Funkcijų iškvietimas

Paprastai funkcijos (išskyrus „main()“) iškviečiamos programos vykdymo metu. Programa vykdoma eilutė po eilutės, kol pasiekiama funkcija. Tada nuoseklus programos sakinių vykdymas yra pertraukiamas ir vykdomi funkcijai priklausantys sakiniai. Užbaigusi funkcijos vykdymą, programa tęsia darbą vykdydama iškart už

funkcijos iškvietimo esantį sakinį. Funkcijos iškvietimo antraštę sudaro funkcijos grąžinamas tipas, funkcijos vardas ir jai perduodamų parametrų sąrašas, aprašomas skliausteliuose. Net jei perduodamų parametrų nėra, skliausteliai privalo būti. Taip funkcija atskiriama nuo kitų programos elementų, pvz. kintamųjų. Funkcijos, kuri negrąžina jokios vertės, tipas yra „void“. Programos, naudojančios kvadratinės šaknies traukimo funkciją pavyzdys:

```
#include <iostream>
#include <cmath>

using namespace std;

int main(int argc, char *argv[])
{
   int rezultatas = sqrt(16);
   cout<<"Saknis is 16 yra "<<rezultatas<<"\n";
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Čia sakinyje `int rezultatas = sqrt(16);` sveikojo tipo kintamajam „rezultatas“ priskiriama funkcijos „sqrt“ grąžinama vertė. Argumentas, perduodamas šiai funkcijai yra sveikasis skaičius 16. Programa į ekraną išveda pranešimą „*Saknis is 16 yra 4*“. Reikia atkreipti dėmesį, kad norint pasinaudoti šia bei kitomis matematinėmis funkcijomis, į programą būtina įtraukti <cmath> failą.

Kintamieji ir konstantos

Visoms programoms reikia kur nors išsaugoti duomenis, su kuriais jos dirba. C++ kalboje kintamuoju vadinama kompiuterio atminties sritis, kurioje saugomi programai reikalingi duomenys. Kompiuterio atmintinę galima įsivaizduoti kaip eilute išrikiuotus kubelius. Visi kubeliai yra sužymėti. Šis žymėjimas vadinamas adresu. Kintamasis rezervuoja vieną ar kelis kubelius, kuriuose gali išsaugoti informaciją. Kintamojo vardas yra tiesiog žymė, naudojama tam, kad rastume tuos rezervuotus kubelius nežinodami tikro jų adreso atmintinėje.

Kintamieji gali būti skirtingų tipų. Apibrėžiant programoje kintamąjį reikia nurodyti komiliatoriui kokio tipo jis yra: sveikasis skaičius, simbolis ar kt. Tada kompiliatorius žino kiek kubelių jam skirti atmintinėje. Informacija kompiterio atmintinėje matuojama baitais, taigi keturių baitų kintamajam reikia rezervuoti keturis kubelius. Akivaizdu, kad kintamųjų tipų bei vardų naudojimas gerokai palengvina duomenų saugojimą bei manipuliavimą jais.

Sveikojo tipo kintamieji

Pagrindiniai C++ sveikojo tipo kintamieji žymimi baziniais žodžiais *int* ir *char*. *int* duomenų tipas naudojamas sveikųjų skaičių vertėms įsiminti. Jis gali užimti 2 arba 4 baitus atminties, priklausomai nuo kompiuterio tipo, tačiau tam pačiam komiuteriui jis visada tas pats. *int* gali būti naudojamas su papildomais baziniais žodžiais *short* arba *long*. Pirmuoju atveju jis užims 2, o antruoju – 4 baitus atminties. *char* duomenų tipas, paprastai naudojamas simboliams saugoti, užima 1 baitą. Reikia pažymėti, kad iš tiesų *char* tipo kintamasis saugo ne patį simbolį, o jo ASCII kodą, t.y. sveikąjį skaičių.

Vieni sveikojo tipo kintamieji gali įgyti tiek teigiamas, tiek ir neigiamas reikšmes, tuo tarpu kiti – tik teigiamas. Tai nurodoma baziniais žodžiais *signed* ir *unsigned*. *int* tipo kintamieji pagal nutylėjimą yra *signed* tipo (gali būti tiek teigiami, tiek ir neigiami). *unsigned* tipo kintamieji turi privalumą prieš *signed* – tokio tipo kintamuosiuose galima patalpinti dvigubai didesnį teigiamą skaičių (nuo 0 iki 65535, kai pirmuoju atveju tik iki 32767).

Slankaus kablelio tipo kintamieji

Slankaus kablelio(kitaip realaus) tipo kintamieji gali saugoti ne tik sveikąją bei ir trupmeninę skaičiaus dalis, t.y. realiuosius skaičius. Šie kintamieji žymimi baziniu žodžiu *float*, o dvigubo tikslumo – baziniu žodžiu *double*. Kompiuterio atmintyje jie užrašomi eksponentine forma – ženklas *s*, mantisė *M* bei laipsnio rodiklis *e*. Taigi, slankaus kablelio skaičius atvaizduojamas taip: $s \times M \times 2^{e-E}$. *E* yra pastovi laipsnio rodiklio dedamoji, leidžianti pastumti kablelį taip, kad būtų galima užrašyti skaičių mažesne už 1 mantise. *float* tipo kintamajam kompiuterio atmintyje skiriami 4 baitai, iš kurių 3 naudojami ženklui ir mantisei (ženklui 1 bitas, mantisei -23), o likęs – eksponentės laipsnio rodikliui. Tokiu būdu išsaugant slankiuosius skaičius jų tikslumą lemia mantisei skiriamos atminties dydis, taigi pvz. *float* tipo kintamųjų tikslumas paprastai neviršija 7 skaitmenų.

Visi pagrindiniai C++ duomenų tipai bei jų užimamas atminties kiekis pateiktas 1 lentelėje:

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	3.4 e +/– 38 (7 skaitmenys)	**4**
double	1.7 e +/– 308 (15 skaitmenų)	**8**
long double	1.7 e +/– 308 (15 skaitmenų)	**8**
wchar_t	platusis simbolis	**2**

Kintamųjų paskelbimas

Kintamieji sukuriami arba paskelbiami nurodant jo tipą, po kurio seka vienas ar keli tarpai bei kintamojo vardas ir kabliataškis. Vardas gali būti bet kokių raidžių, skaičių bei pabraukimo simbolių, išskyrus tarpo simbolį, seka, tačiau negali prasidėti skaičiumi. Skelbiant kintamąjį pravartu parinkti jam prasmingą vardą, programose reikia stengtis apriboti vieno simbolio vardų naudojimą. C++ skiria didžiąsias bei mažąsias raides, todėl tokie kintamųjų vardai kaip Indeksas ir indeksas yra skirtingi. Kai kurie žodžiai C++ kalboje, tokie kaip if, else, for, while, main ir kt. yra rezervuoti (tarnybiniai) t.y. jų negalima vartoti kintamųjų vardams skelbti, tačiau parenkant prasmingą vardą galima būti beveik visiškai tikram, kad taip neatsitiks. Standartinis tarnybinių žodžių sąrašas pateiktas 1 priede (kai kuriems kompiliatoriams jis gali šiek tiek skirtis). Viename sakinyje galime sukurti daugiau nei vieną to paties tipo kintamąjį rašydami kntamjo tipą, o po jo kintamųjų vardus, atskirtus kableliais. Kintamųjų paskelbimo pavyzdžiai:

```
int rezultatas;    unsigned int ilgis, plotis, aukstis;      double nuokrypioKvadratoVidurkis;
```

Reikšmių priskyrimas kintamiesiems

Reikšmės kintamiesiems priskiriamos naudojant priskyrimo operatorių „=“ (pvz. ilgis=5;). Tai vadinama kintamojo inicializacija, kadangi paskelbus kintamąjį, jo reikšmė gali būti neapibrėžta. Kintamajam priskirti reikšmę galima jau jį paskelbiant. Tai pat galima paskelbti ir inicializuoti iš karto kelis kintamuosius:

```
int rezultatas; rezultatas = 0;     int ilgis = 1, plotis , aukstis = 2;
```

Typedef naudojimas

Apibrėžti pvz. unsigned long int tipo kintamuosius programoje daug kartų nėra patogu, todėl C++ buvo įvestas naujas būdas, leidžiantis sukurti sinonimus egzistuojantiems

duomenų tipams įvardinti. Tai atliekama rašant bazinį žodį *typedef*, po kurio seka egzistuojančio bei naujo, vartotojo sugalvoto jam, tipo pavadinimai:

```
#include <iostream.h>
typedef unsigned short int USHORT

int main()
{
USHORT ilgis = 5;
cout<<“Ilgis = “<<ilgis<<“\n“;
return 0;
}
```

Svarbu atkreipti dėmesį, kad *typedef* nekuria naujų tipų, o tik leidžia pervardinti jau esamus.

Dar apie sveikuosius skaičius

Faktas, kad unsigned long int tipo kintamiesiems egzistuoja vertės, kurią jie gali saugoti limitas, dažniausia nesukelia problemų. Vis dėlto pravartu žinoti, kas atsitinka sveikajam skaičiui viršijus maksimaliai įmanomos saugoti vertės limitą. Jei taip atsitinka, sveikojo tipo kintamasis „persiverčia“ panašiai, kaip spidometro kilometražo indikatorius automobilyje ir pradeda rodyti mažiausią įmanomą vertę, pvz. unsigned short int tipo kintamajame saugant 65535 vertę ir pridėjus 1 pastaroji pakis į 0, o signed short int nuo 32767 pakis į -32768.

char tipo kintamieji, skirti simboliams saugoti, yra 1 baito, taigi gali įgyti 265 skirtingas vertes. Kompiuteriai nežino nieko apie simbolius, skyrybos ženklus, sakinius, todėl visi simboliai transformuojami į skaičius, t.y. kiekvienam jų priskiriama tam tikra fiksuota skaitinė vertė (pvz. „a“ - 97), apibrėžiama ASCII standartu.

Konstantos

Konstantos iš dalies panašios į kintamuosius, kadangi jos taip pat skirtos duomenims saugoti. Skirtingai, nei kintamųjų, jų vertės negali kisti, todėl konstantas reikia inicializuoti jų sukūrimo metu. Konstantos gali būti įvairių tipų: sveikojo, slankaus kablelio, simbolinės, loginės, pvz.:

```
int ilgis = 1;        float svoris = 70.5;        char pirmaRaide = `A`;
```

Šiame kode paskelbti trys *int, float* bei *char* tipų kintamieji ir jiems priskirtos 1, 70.5 ir `A` konstantos atitinkamai.

Konstantos gali būti skelbiamos naudojant tarnybinį žodį *#define*, po kurio rašomas konstantos pavadinimas bei jos vertė, pvz.:

```
#define Pi  3.14159265
```

Reikia įsidėmėti, kad konstanta Pi nėra kokio nors tipo (šiuo atveju ji galėtų būti *float* arba *double*). *#define* tiesiog „priverčia“ pakeisti visus Pi į 3.14159265 programoje, todėl kompiliatorius mato tiesiog skaičių. Kitas būdas konstantai programoje apibrėžti yra naudoti bazinį žodį *const*:

```
const int ilgis = 1;
```

Šiuo atveju konstanta *ilgis* yra sveikojo tipo, todėl kompiliatorius gali tikrinti tipų suderinamumą programoje ir įspėti apie potencialiai neteisingą jos vartojimą.

Išvedant tekstinę informaciją į monitoriaus ekraną ar failą naudojamos specialios simbolinės konstantos:

\n	pereiti į naują eilutę
\t	tabuliacija
\b	grįžti atgal pervieną poziciją
\r	grįžti į eilutės pradžią
\‘	vienguba kabutė
\“	dvigubos kabutės
\?	klaustukas
\\	atgal pasviręs brūkšnys

Išvardijimai

Išvardijimai (angl. enumerations) leidžia apibėžti naujus duomenų tipus bei skelbti šių tipų kintamuosius įgyjančius tik tam tikrą verčių rinkinį, pasirinktą aprašant išvardijimą. Išvardijimai skelbiami baziniu žodžiu *enum*, pvz.:

```
enum spalvos {juoda, zalia, raudona, melyna, balta };
```

Pastebėkime, kad čia nenaudojami jokie pagrindiniai duomenų tipai. Išvardijimo tipo kintamieji skelbiami įprastu budu:

```
spalvos vaivorykste, roze;
```

Reikšmės, kurias gali įgyti tokie kintamieji yra pastovūs dydžiai, išvardinti tipo apraše tarp riestinių skliaustų. Pvz., galima rašyti tokius sakinius:

```
spalvos roze; roze = blue;
```

Išvardintų reikšmių tipas atitinka skaitinio tipo kintamuosius, todėl šioms reikšmėms visada priskiriama 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, zalia – 1, raudona – 2 ir t.t.

Bet kuriai išvardintajai reikšmei galima aiškiai nurodyti ją atitinkantį sveikąjį skaičių. Jei po išvardintosios reikšmės sveikasis skaičius nenurodomas, tai tokios reikšmės skaitinis atitikmuo yra vienetu didesnis nei prieš buvusios išvardintosios reikšmės, pvz.:

```
enum spalvos {juoda=100, zalia=300, raudona=500, melyna=700, balta };
```

Išraiškos ir sakiniai

Kaip jau minėta, programa yra nuosekliai vykdomų instrukcijų seka. Esminė programavimo „galia“ yra galimybė vykdyti vieną ar kitą instrukcijų seką, priklausomai nuo tam tikrų sąlygų.

C++ kalboje sakiniai kontoliuoja programos vykdymo seką, apskaičiuoja išraiškas arba nevykdo nieko (tuščiasis sakinys). Visi sakiniai būtinai turi būti baigiami kabliataškiu. Vienas iš dažniausiai naudojamų sakinių yra priskyrimo sakinys:

```
x = y + z;
```

Priešingai, nei algebroje, šis sakinys nereiškia, kad *x* yra lygus *y* ir *z* sumai. Jis sako: „priskirk kintamajam *x* kintamųjų *y* ir *z* sumą“. Net jei jis ir daro du dalykus (sudeda *y* su *z*, rezultatą priskiria *x*) tai yra vienas sakinys ir jo gale dedamas vienas kabliataškis. Priskyrimo operatorius priskiria tai, kas yra dešinėje lygybės pusėje tam, kas yra jos kairėje. Tarpai priskyrimo sakinyje (o taip pat ir kituose) yra ignoruojami, todėl jie paprastai naudojami tik padaryti programą lengviau skaitomą.

Bet kurioje programos vietoje užuot rašius vieną sakinį, galima parašyti jų grupę (bloką). Blokas prasideda atsidarančiu riestiniu skliaustu „{“ ir baigiasi užsidarančiu „}“. Nors kiekvienas sakinys bloko viduje baigiamas kabliataškiu, pats blokas – ne, todėl tokią struktūrą galima vadinti vienu sudėtiniu sakiniu:

```
{ laik = a;
  a = b;
  b = laik; }
```

Šiame programos bloke demonstruojamas *a* ir *b* kintamųjų reikšmių sukeitimas vietomis. Išraiška C++ kalboje vadinama visa tai, kas yra apskaičiuojama, pvz. `x = y + 5`. Sakoma, kad išraiška visada grąžina vertę.

Operatoriai

Operatoriumi C++ kalboje vadinamas simbolis, verčiantis kompiliatorių atlikti tam tikrą veiksmą. Operatoriai veikia operandus. Pagal veikimo būdą jie skirstomi į keletą

kategorijų. Bene dažniausiai naudojami dviejų grupių – priskyrimo bei matematiniai operatoriai.

Priskyrimo operatorius veikia operandą, esantį jo kairėje, keisdamas pastarojo vertę į dešinėje pusėje esančio operando (išraiškos) vertę, pvz.:

```
x = y + z;
```

išraiškoje operandui x priskiriama y ir z sumos vertė. Kadangi konstantoms negalima priskirti verčių, jos negali būti kairėje operatoriaus pusėje.

Matematinių operatorių aibę sudaro 5 operatoriai: sudėties „+“, atimties „-“, daugybos „*“, dalybos „/“ ir modulio „%“. Sudėties ir atimties operatoriai veikia taip, kaip įprasta matematikoje, išskyrus atvejį, kai atimami unsigned tipo sveiki skaičiai. Šiuo atveju neigiamas rezultatas bus interpretuojamas kaip didelis teigiamas skaičius.

Daugybos bei dalybos operatoriai su slankaus kablelio skaičiais taipogi yra klasikiniai, tačiau dalijant sveikuosius skaičius rezultatas yra taip pat sveikasis, pvz. 5 dalijant iš 2 rezultatas bus 2 (o ne 2.5, kaip įprasta matematikoje). Dalybos liekaną galime gauti, pasinaudoję modulio operatoriumi.

```
int Atsakymas = 10 / 3;     //Rezultatas 3
int Liekana = 10 % 3;       //Rezultatas 1
```

C++ matematiniai operatoriai dažnai kombinuojami su priskyrimo operatoriais:

```
int manoCentai = manoCentai + 5;
```

Čia sudedamos kintamojo manoCentai vertė su 5 ir rezultatas priskiriamastam pačiam kintamajam manoCentai. Galima šią išraišką užrašyti dar kompaktiškiau:

```
int manoCentai  += 5;
```

Čia pavartotas „+=“ operatorius vadinamas sau priskiriančiu sudėties operatoriumi. Egzistuoja ir sau priskiriantys atimties, daugybos, dalybos bei liekanos operatoriai „–=“, „*=“, „/=“, „%=“.

Dažniausia C++ programose pridedama ar atimama vertė yra 1, todėl šiai matematinei operacijai atlikti yra specialūs operatoriai, vadinami inkremento „++“ bei dekremento „--" operatoriais. Taigi, norint padidinti kokio nors kintamojo, pvz. c, vertę vienetu, galima parašyti c++. Tiek inkremento, tiek ir dekremento operatoriai gali būti vartojami dviem būdais – rašomi prieš kintamąjį arba po jo (pvz. ++c arba c++). Paprastose išraiškose nėra jokio skirtumo, kurį užrašymo būdą pasirinksime, tačiau sudėtingose – tokiose, kur po reikšmės padidinimo ar sumažinimo seka priskyrimas – netgi labai. Jei pvz. inkremento opreatorius parašytas prieš kintamąjį – pastarojo vertė padidinama prieš priskyrimą, o jei po kintamojo – tik alikus priskyrimo operaciją. Pavyzdžiui programos fragmente:

```
int manoCentai = 5;
int manoPinigine = ++manoCentai;
```
kintamajam manoPinigine bus priskirta vertė, lygi 6, o jei antroji fragmento eilutė būtų `int manoPinigine = manoCentai++;` – 5.

Operatorių pirmumo lygiai

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

```
int x = 2 + 3 * 4;
```

neaišku, ar pirmiausia bus atliekama sudėties, ar daugybos operacija. Yra nustatyti ne tik aritmetinių, bet ir kiekvieno C++ kalboje sutinkamo operatoriaus pirmumo lygiai. Kadangi daugybos lygis aukštesnis, nei sudėties, pirmiausia bus atliekama ji ir *x* vertė bus lygi 14. Vienodo pirmumo lygio operacijos atliekamos iš kairės į dešinę. Sudėtingose išraiškose pravartu naudoti skliaustus operacijoms grupuoti bei jų skaičiavimo eiliškumui keisti. Išraiškose su skliaustais operacijos atliekamos iš vidaus į išorę, pvz. išraiškoje:

```
int x = ((2 + 3) * 4 – (1+2) * 2) / 5;
```

pirmiausia sudedami 2 ir 3 bei 1 ir 2. Po to gauti rezultatai dauginami iš 4 ir 2 atitinkamai. Galiausiai atliekamos atimties bei dalybos operacijos. Pastaroji išraiška lengvai suprantama kompiliatoriui, bet ne žmogui, nagrinėjančiam programą, todėl kartais pravartu panaudoti keletą kintamųjų tarpiniams rezultatams saugoti tam, kad programa būtų suprantamesnė. Visų C++ operatorių ppirmumo lygiai pateikti 2 priede.

Palyginimo sąlygos bei loginiai operatoriai

C++ kalboje nulis vadinamas „melu“ (*false)*, o visos kitos vertės – „tiesa“ (*true*). Taigi, jei išraiška yra *false*, ji lygi nuliui. Palyginimo operatoriai naudojami nustatyti, ar du palyginami skaičiai lygūs, ar vienas didesnis (mažesnis) už kitą. Palyginimo operatoriai:

= =	lygu	10 = = 20 – *false*; 10 = = 10 – *true*;
! =	nelygu	10 ! = 20 – *true*; 10 ! = 10 – *false*;
>	daugiau	20 > 10 – *true*; 10 > 20 – *false*;
> =	daugiau arba lygu	10 > = 20 – *false*; 20 > = 20 – *true*;
<	mažiau	10 < 20 – *true*; 20 < 10 – *false*;
< =	mažiau arba lygu	10 < = 20 – *true*; 20 < = 10 – *false*;

Svarbu įsidėmėti, kad palyginimo operatorius „= =“ ir priskyrimo operatorius „=“ yra skirtingi.

Sąlygos operatorius yra vienintelis C++ operatorius, naudojantis 3 operandus:

(išraiška 1) ? (išraiška 2) : (išraiška 3)

Ši eilutė suprantama taip: jei „išraiška 1“ yra *true*, grąžinama „išraiška 2“, jei *false* – „išraiška 3“. Pavyzdžiui šiame programos fragmente:

```
int Rezultatas = (5 > 4) ? 10 : -10;
```

kintamajam Rezultatas bus priskirta vertė 10.

Kartais patogu vienu metu tikrinti ne vieną o kelias sąlygas, pvz., jei norima įsitikinti, kad koks nors kintamasis *x* didesnis ir už *y* ir už *z*. C++ kalboje tam tikslui skirti 3 loginiai operatoriai:

IR	–	&&	(išraiška 1) && (išraiška 2)
ARBA	–	||	(išraiška 1) || (išraiška 2)
Ne	–	!	! (išraiška)

Loginis operatorius „&&“ (AND) grąžina *true* vertę, jei abi išraiškos yra tenkinamos, o „||“ (OR) – jei bet kuri iš jų. Loginis „!“ (NOT) operatorius atlieka neigimo funkciją – jis grąžina *false* vertę, jei tikrinama išraiška yra *true*.

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, įvykdę šį sakinį *a* reikšmę gausime lygią 5, o

b reikšmę – 3.

Valdantieji sakiniai

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

Sąlygos sakinys

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

if (*sąlygos_išraiška*) sakinys;

Jis vykdomas taip: visu 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";
```

išves į ekraną užrašą “x yra lygus 100”, jei kintamojo *x* reikšmė iš tikro bus lygi 100.

Taip pat yra galimybė nurodyti, ka reikėtu daryti, kai *salygos_išraiškos* rezultatas yra false. Tam gali buti 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 buti rašomas naujas *if* sakinys. Kitas 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 norint *if* ar *else* šakose atlikti kelis sakinius, juos būtina apskliausti riestiniais skliaustais {}.

Ciklai: while , do-while , for.

Ciklų paskirtis yra kartoti sakinį arba jų grupę tam tikra kiekį kartų arba kol nebus įvykdyta tam tikra sąlyga.

while ciklo bendrasis pavidalas yra:

while (salygos_išraiška) sakinys;

Jame nurodytas sakinys turi būti kartojamas tol, kol prieš tai apskaičiuota *salygos_išraiška* duoda rezultatą *true*. Programa *while* ciklui pailiustruoti:

```
#include <iostream>
using namespace std;
int main ()
```

```
{
        int n;
        cout << "Iveskite pradini skaiciu > ";
        cin >> n;
        while (n>0)
        {
                cout << n << ", ";
                --n;
        }
        system(„PAUSE“);
        return 0;
}
```

Šios programos vykdymo pradžioje vartotojui į ekraną išvedamas priminimas „Iveskite pradini skaiciu“. Įvedus skaičių, *while* ciklo sudėtinis sakinys kartojamas tol, kol n>0. Kai tik n pasidaro lygus 0, ciklas nutraukiamas.

Svarbu atkreipti dėmesį į tai, kad *while* ciklo sudėtiniame sakinyje turi buti sakinys, turintis įtaka tam, kad ciklo sąlygos išraiškos rezultatas taptu *false*. Kitokiu atveju turėtume nesibaigiantį ciklą. Šiuo atveju tai yra sakinys `--n;`

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

do-while ciklo bendrasis pavidalas yra toks:

do

sakinys;

while (salygos_išraiška);

Jis vykdomas taip pat, kaip ir *while* ciklas. Skirtumas yra tik toks, kad čia *salygos_išraiška* apskaičiuojama po to, kai įvykdomas sakinys. Taip garantuojama, kad sakinys bus įvykdytas bent vieną kartą. *do-while* ciklo nutraukimo sąlyga suformuojama ciklo bloke.

for ciklo bendrasis pavidalas yra:

for (inicializacija; salyga; 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 sukurtas pakartotinam veiksmu vykdymui vadovaujantis skaitikliu. Skaitiklis visu pirma yra inicializuojamas (jam suteikiama pradinė reikšmė), o vėliau jo reikšmė keičiama tam tikru dydžiu kiekvienos iteracijos metu. *for* ciklo veiksmų sekos punktai yra tokie:

1. Atliekamas inicializacijos veiksmas. Paprastai tai pradinės reikšmės suteikimas skaitiklio kintamajam. Šis veiksmas atliekamas tik vieną kartą.
2. Tikrinama sąlyga, ar skaitiklio 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 buti paprastas arba sudėtinis sakinys (blokas).

4. Atlikus sakinį, vyksta reikšmės_keitimas. Dažniausia keičiama skaitiklio reikšmė (didinama arba mažinama) ir grįžtama į ciklo veiksmų sekos 2 punktą. Žemiau pateikiama programa su *for* ciklu, kuriame skaitliuko reikšmė yra mažinama:

```
#include <iostream>
using namespace std;
int main ()
        {
         for (int n=10; n>0; n--)
                {
                 cout << n << ", ";
        }
        cout << "Startas!";
        return 0;
}
```

for cikle inicializacijos ir reikšmės_keitimo sakinių gali ir nebūti. Jie gali buti tušti, tačiau laukus skiriantys kabliataškiai (;) turi buti 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 (gal inicializacija buvo atlikta dar iki *for* ciklo pradžios). *for* cikle inicializacija ir reikšmės_keitimas gali susidėti iš keleto išraiškų. Išraiškos skiriamos kableliu (,). Pvz., tarkime, cikle norime inicializuoti daugiau nei vieną kintamąjį:

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

Šis ciklas bus kartojamas 50 kartu, jei *n* ir *i* reikšmės nebus keičiamos ciklo vidiniais sakiniais.

Visų paminėtų ciklų darbą galima valdyti sakiniais *break* bei *continue*. Cikle sutikus *break* sakinį, jis nutraukiamas nepaisant to, kad jo tęsimo sąlyga ir yra įvykdyta. *break* gali būti naudojamas begaliniams ciklams nutraukti arba nutraukti ciklams dar jiems normaliai nepasibaigus. *continue* sakinys įgalina nutraukti ciklo iteraciją jos nebaigus tuo pačiu peršokant į kitos iteracijos pradžią ir tęsti ciklą tol, kol jis pasibaigs normaliai. Pvz. žemiau pateiktoje programoje begalinis *while* ciklas bus nutrauktas *break* sakiniu kintamajam *index* sumažėjus iki nulio. Be to esant *index* daugiau už 5, sekanti ciklo iteracija bus pradedama iš naujo neišvedant į ekraną *index* vertės.

```
#include <iostream>
```

```
using namespace std;
int main ()
{
 int index = 10;
            while (1)
            {
                        index--;
                        if (index  < 1)
                                    break;
                        if (index > 5)
                                    continue;
            cout << index << ", ";

            }
            cout << "Startas!";
            return 0;
}
```

Išrinkimo sakinys

Išrinkimo sakinio *switch* struktūra yra kažkiek savita. Juo, priklausomai nuo kintamojo reikšmės, vykdoma viena iš sakinių grupių, kuri gali būti norimas kiekis. Tai kažkiek 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 jį, peršokama į *switch* sakinio galą. Jei išraiškos reikšmė nėra lygi konstantai1, tuomet tikrinama, ar ji nėra lygi konstantai2. Jei lygu, tai vykdoma sakiniu_grupė_2 iki *break* ir peršokama i 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.

switch sakinys yra kažkiek neiprastas C++ kalboje, nes jame vartojamos žymės, o ne blokai (sudėtiniai sakiniai). Tai verčia naudoti *break* po kiekvienos sakinių grupės, kurią reikia atlikti esant specifinei sąlygai. Kitu atveju likusios kitas konstantas atitinkančios sakinių grupės būtų vykdomos taip pat, kol nepasiektume *switch* sakinio galo arba nesutiktume *break*. Jei mes nerašysime *break* po pirmos sakiniu 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. Tai taip pat gali buti naudinga, norint vykdyti tą pačią sakinių grupę, esant įvairioms išraiškos reikšmėms. Pvz.:

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

Pastebėtina, kad *switch* sakinyje vietoje konstantų negalima vartoti kintamųjų arba reikšmių intervalų. Jei reikalingi skaiciavimai tikrinant reikšmes iš kažkurio intervalo, reikia naudoti *if* arba *else if* sakinius.

Funkcijos

Fukcija yra subprograma, galinti apdoroti duomenis bei grąžinti vertę. Kiekviena C++ programa turi bent vieną funkciją – *main*(), kuri yra iškviečiama automatiškai. *main*() gali iškviesti kitas funkcijas, o jos savo ruožtu dar kitas. Kiekviena funkcija turi savo pavadinimą, kurį sutikus programoje jos nuoseklus darbas pertraukiamas ir atliekama funkcijai priklausanti programos dalis. Funkcijai baigus darbą grįžtama į tą programos vietą, kur buvo iškviesta funkcija ir vykdomas po jos iškvietimo sekantis sakinys.

Norint programoje naudoti savo sukurtą funkciją, pirmiausia reikia ją paskelbti, o po to aprašyti. Funkcijos paskebimas (deklaravimas) kompiliatoriui nurodo jos vardą, grąžinamo rezultato tipą, parametrus. Funkcijos aprašymas parodo, kaip ji yra realizuojama. Nė viena funkcija negali būti iškviečiama, prieš tai jos nedeklaravus. Funkcijos deklaravimas dar vadinamas jos prototipu.

Funkciją paskelbti galima šiais būdais:

a) įrašant jos prototipą į atskirą failą ir įtraukiant pastarąjį į programą tarnybiniu žodžiu *#include*;
b) įrašant jos prototipą į tą patį failą, kurioje yra programa;
c) aprašant fukciją anksčiau, nei ji yra panaudojama kurioje nors kitoje funkcijoje – šiuo atveju jos aprašymas yra kartu ir paskelbimas.

Funkcijos prototipo struktūra yra tokia:

grąžinamos_reikšmės_tipas funkcijos_vardas(*parametrų_sąrašas*);

Fukcijos prototipas turi tiksliai sutapti su jos aprašymo dalimi grąžinamo rezultato tipu, pavadinimu bei perduodamų parametrų tipu. Kintamųjų, kurie perduodami kaip parametrai, vardų prototipe rašyti nebūtina, pvz.:

```
int Suma(int,int);
```

tačiau prasmingi parametrų vardai daro ją suprantamesnę. Parametrai vienas nuo kito skiriami kableliais.

Funkcijos aprašymas susideda iš jos antraštės bei realizacijos dalies. Antraštė atrodo lygiai taip pat, kaip ir prototipas, išskyrus tai, kad parametrai privalo būti įvardinti ir nėra kabliataškio pabaigoje. Funkcijos realizaciją sudaro sakinys ar jų grupė tarp riestinių skliaustų, pvz.:

```
int Suma(int demuo1, int demuo2)
{
        return demuo1 + demuo2 ;
}
```

Kai kurios funkcijos gali neturėti parametrų. Tuo atveju skliaustai paliekami tušti. Visi sakiniai funkcijos realizacijos dalyje turi būti baigiami kabliataškiu, tačiau pati funkcija – ne. Ji baigiama užsidarančiu riestiniu skliaustu. Jei funkcija grąžina vertę, tai atliekama naudojant *return* bazinį žodį, po kuri seka grąžinama išraiška. *return* gali būti ir ne funkcijos pabaigoje, tačiau įvykdžius jį funkcijos darbas baigiamas. Kiekviena funkcija turi grąžinamą tipą. Jei jis nėra aiškiai nurodomas, jis yra *int*. Jei funkcija neturi grąžinti jokios reikšmės jos grąžinamas tipas turi būti *void*.

Iškvietus funkciją, jos vykdymas prasideda nuo sakinio, sekančio po „{“. Funkcijos gali iškviesti kitas funkcijas, taip pat ir save. Funkcijai galima ne tik perduoti kintamųjų reikšmes iš pagrindinės programos, bet ir skelbti lokalius kintamuosius jos viduje. Pastarieji kintamieji vadinami lokaliais, nes jų veikimo sritis apribota riestiniais funkcijos skliaustais, t.y. jie egzistuoja tik tos funkcijos viduje. Funkcijai baigus darbą jie tampa nepasiekiamais. Detaliau funkcijų veikimas aprašomas 3 priede. Programos, demonstruojančios lokalius kintamuosius, pavyzdys:

```
#include <iostream>
using namespace std;
float Konvertuok(float);
int main ()
{
 float TempKelv, TempCels;
cout<<“Iveskite temperatura Celsijais “;
cin>>TempCels;
TempKelv = Konvertuok(TempCels);
cout<<“\nTemperatura Kelvinais “<<TempKelv;
        return 0;
```

```
}

float Konvertuok(float Temp)
{
float TempKelv = Temp;
TempKelv += 273.15;
return TempKelv;
}
```

Čia funkcijoje *Konvertuok()* yra paskelbtas lokalus kintamasis *TempKelv*, kurio vardas sutampa su *main()* funkcijoje paskelbtu analogišku kintamuoju. Nors šių kintamųjų vardai ir vienodi, jie nėra tapatūs, kadangi kiekvienas apibrėžtas tik savo bloko viduje.

Globalūs ir lokalūs kintamieji

Kintamieji, apibrėžti ne funkcijoje (įskaitant ir *main()*), vadinami globaliaisiais ir galioja bet kurioje funkcijoje. Lokalūs kintamieji, kurių vardai sutampa su globaliųjų vardais, „paslepia“ pastaruosius, nekeisdami jų įgytų verčių, t.y. jei programa šiuo atveju kreipiasi į kintamąjį, tai tik į lokalų.

Kintamieji, apibrėžti kokioje nors funkcijoje vadinami lokaliais. Tai reiškia, kad jie yra matomi ir naudojami tik tos funkcijos viduje. C++ kintamuosius galima apibrėžti ne tik funkcijos realizacijos pradžioje, bet ir bet kurioje jos vietoje. Kintamojo veikimo sritį apibrėžia blokas, žymimas „{}“. Verta įsidėmėti, kad ta pati taisyklė galioja ir kintamiesiems, apibrėžtiems sąlygos ar ciklo bloko viduje.

Funkcijos parametrai (argumentai)

Funkcijai perduodami parametrai gali būti įvairaus tipo. Kiekviena teisinga C++ išraiška, pvz. konstantos, matematinės ar loginės išraiškos, kitos, grąžinančios vertę funkcijos gali būti funkcijos argumentu. Argumentai, perduodami funkcijai, yra lokalūs tai funkcijai. Jų keitimas nepakeičia pastarųjų verčių iškvietusioje funkcijoje. Tai vadinama argumentų reikšmių perdavimu. Tokiu būdu kiekvienam perduodamam argumentui sukuriama jo lokali kopija, kuri traktuojama kaip kitas lokalus kintamasis, pvz.:

```
#include <iostream>
using namespace std;
void swap(int x, int y);
int main ()
{
 int x = 5, y = 10;
cout<<“Pries sukeitima:\n x=“<<x<<“ y=“<<y<<“\n“;
swap(x,y);
```

```
cout<<“Po sukeitimo: \n x=“<<x<<“ y=“<<y<<“\n“
        return 0;
}

void swap(int x, int y)
{
cout<<“Funkcijoje pries sukeitima: \n x=“<<x<<“ y=“<<y<<“\n“;
int temp;
temp = x;
x = y;
y = temp;
cout<<“Funkcijoje po sukeitimo: \n x=“<<x<<“ y=“<<y<<“\n“;
}
```

Čia funkcija *swap()* naudojama kintamųjų *x* ir *y* vertėms sukeisti vietomis. Nors šioje funkcijoje jų vertės ir sukeičiamos vietomis, *main()* funkcijoje jos išlieka nepakitusios.

Kartais gali būti naudinga, kad funkcija galėtų pakeisti jai perduodamų kintamųjų reikšmes. Šiam tikslui pasiekti parametrams turi buti perduodami argumentų adresai atmintyje, o ne vertės. Tokiu būdu funkcijos parametrą patalpinus toje pačioje atminties vietoje, kuri skirta perduodamam argumentui, keičiant vieno reikšmę, pasikeis ir kito reikšmė. Norint tai realizuoti aukščiau pateiktame pavyzdyje reiktų pakeisti funkcijos prototipą bei antraštę realizacijos dalyje, pridedant simbolį & kiekvieno parametro tipą nurodančio bazinio žodžio gale, t.y.:

`void swap(int& x, int& y);` ir `void swap(int& x, int& y){...`

Iškviečiant funkciją kiekvienam deklaruotam parametrui funkcijos prototipe turi būti suteikta atitinkamo duomenų tipo reikšmė. Jei kviečiant funkciją perduodamų jai kaip parametrai argumentų reikšmių tipai ar skaičius nesutampa, kompiliatorius generuoja klaidą. Norint to išvengti, galima suteikti parametrams reikšmes pagal nutylėjimą funkcijos prototipe, pvz.:

```
int suma(int x=1, int y=1);
```

Funkcijos aprašymo dalis lieka nepakitusi. Jei iškvietimo metu tokios funkcijos parametrui nesuteikiama vertė, kompiliatorius naudoja vertę pagal nutylėjimą. Bet kuriam ar net visiems funkcijos parametrams gali būti suteiktos vertės pagal nutylėjimą. Vienintelis apribojimas čia yra toks, kad jei kuris parametras neturi vertės pagal nutylėjimą, joks prieš tai esantis parametras negali jos turėti. Aukščiau pateiktame pavyzdyje kintamajam *x* gali būti suteikta reikšmė pagal nutylėjimą tik tada, kai ji suteikta ir kintamajam *y*.

Funkcijos vienodais vardais

C++ leidžia naudoti daugiau nei vieną funkciją tuo pačiu vardu, jei jų parametrų tipai ar skaičius (ar abu kartu) skiriasi, pvz.:

```
int manoFunkcija(int, int ); int manoFunkcija(float, float ); int manoFunkcija(float);
```

Tai vadinama funkcijų užklotimi. Funkcijų grąžinami tipai gali būti tie patys arba skirtingi, tačiau dvi funkcijos vienodais vardais ir parametrų rinkiniu, bet skirtingais grąžinamo rezultato tipais sukels kompiliavimo klaidą.

Funkcijos vienodais vardais įgalina atlikti tą pačią užduotį esant skirtingiems jų parametrų tipams, taigi kreipdamasis į vieną iš funkcijų vienodais vardais, kompiliatorius išrenka „teisingą“ lygindamas perduodamų argumentų bei funkcijos parametrų tipus bei skaičių.

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

float plotas(float a) {return a*a;}
float plotas(float a, float b) {return a*b;}

int main ()
{
 float x = 0, y = 0;
cout<<“Iveskite krastines\n “;
cin>>x>>y;
if (x = = y)
        cout<<“Kvadrato plotas yra „<<plotas(x)<<endl;
else
        cout<<“Staciakampio plotas yra „<<plotas(x,y)<<endl;
return 0;
}
```

Šioje programoje funkcijos vienodais vardais naudojamos skirtingų figūrų plotams apskaičiuoti. Tinkama funkcija parenkama, atsižvelgiant į perduodamų argumentų skaičių.

Inline tipo funkcijos

Programoje parašius funkciją kompiliatorius paprastai sukuria jos kodui vieną instrukcijų seką atmintyje. Iškviečiant šią funkciją, programa „peršoka“ į tą atminties vietą, įvykdo reikiamas instrukcijas ir grįžta atgal, t.y. vykdant funkciją pvz. 5 karus, bus atlikti 5 tokie peršokimai. Kartais, kai funkcijos kodas yra labai trumpas, o kreipimųsi į ją daug, tai gali sulėtinti programos darbą. To galima išvengti, parašant bazinį žodį *inline* tokios funkcijos deklaravime. Tuomet kompiliatorius nekuria

realios funkcijos, o kopijuoja jos kodą į tą programos vietą (vietas), kur ši funkcija yra iškviečiama.

Rekursija

Funkcijos gali iškviesti pačios save – tai vadinama rekursija. Rekursija naudinga sprendžiant specifines užduotis, pvz. tokias, kuriose funkcija veikia duomenis, o po to ir gautą rezultatą tuo pačiu būdu. Rekursija gali būti tiesioginė – kai funkcija iškviečia pati save ir netiesioginė – kai ji iškviečia kitą funkciją, kuri savo ruožtu iškviečia pirmąją. Būtina žinoti, kad funkcijai kviečiant save, sukuriama nauja jos kopija ir visi naujosios versijos lokalūs kintamieji yra kiti, nei senesnių, taigi negali vieni kitų įtakoti. Rekursijos būdu galima skaičiuoti pavyzdžiui faktorialą:

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

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

int main ()
{
long n = 0;
cout<<“Iveskite skaiciu\n “;
cin>>n;
if (n > 0)
        cout<<“Skaiciaus „<<n<<“ faktorialas yra „<<faktorialas(n)<<endl;
return 0;
}
```

Šioje programoje funkcija faktorialas kreipiasi į save pačią tol, kol kintamojo *n* vertė didesnė už 1. Tuo būdu lokalusis *n* kintamasis nuosekliai mažėja iki *1*. Vėliau tiek pat kartų vyksta daugyba *n*(n-1)*.

Klasės

Klasės praplečia standartines C++ galimybes, leisdamos kurti naujus, vartotojo apibrėžtus duomenų tipus, išlaikančius visas bazinių duomenų tipų savybes bei funkcionalumą bei leisdamos lengviau spręsti realias kompleksines užduotis.

Klasė iš esmės yra kintamųjų (dažniausia skirtingų tipų) bei juos veikiančių funkcijų rinkinys. Klasikinis pavyzdys galėtų būti automobilis – jį sudaro kėbulas, durys, ratai, variklis ir t.t. Automobilį galima apibūdinti ir pagal jo atliekamus veiksmus – jis gali važiuoti, greitėti, stabdyti, stovėti ir t.t. Klasė leidžia susieti visas šias dalis bei funkcijas į vieną visumą, vadinamą objektu. Programuotojui toks susiejimas yra naudingas tuo, kad įgalina paprasčiau manipuliuoti duomenimis.

Klasės paskelbimas

Klasę sudaro bet kokia bet kokio tipo kintamųjų bei juos veikiančių funkcijų, vadinamų klasės nariais, kombinacija. Skelbiant klasę naudojamas *class* bazinis žodis. Klasės skelbimo pavidalas yra toks:

```
class klasės_vardas
{ kreipties_atributas: narys1;
  kreipties_atributas: narys2();
  . . .
 } objekto_vardas;
```

Čia klasės vardas – klasės pavadinimas, objekto vardas – šios klasės objektų vardų sąrašas. Klasės paskelbimas neišskiria atminties jai. Jis tik informuoja kompiliatorių kas ši klasė yra, kokie duomenys ją sudaro ir ką ji gali daryti. Klasės nariai – tai kintamųjų ir funkcijų aprašai, prieš kuriuos nurodomi kreipties atributai, nurodomi baziniu žodžiu *private*, *public* arba *protected*, apibrėžiančiu klasės nario galiojimo sritį.

Klasės objektai skelbiami taip, kaip ir įprasti kintamieji, pvz.:

```
class Auto {.....};     Auto Ferrari, Porshe;
```

Čia paskelbti du klasės *Auto* objektai, pavadinti *Ferrari* ir *Porshe*. Reikia įsidėmėti, kad klasės vardas ir objekto vardas yra du skirtingi dalykai. Klasė gali būti apibūdinama kaip idėja, o objektai yra jos realizacijos produktai, kuriais galima realiai manipuliuoti.

Apibrėžus klasės objektą į jo narius kreipiamasi naudojantis „.“ (taško) operatoriumi, pvz. jei anksčiau minėtos klasės *Auto* nariai yra kintamasis *masė* bei funkcija *juda()*, jos realizacija bei kreipimasis į objektą *Porshe* atrodys taip:

```
class Auto {
int mase;
public:
        int greitis;
         bool juda() {if (greitis > 0) return true; else return false;}
};

int main() {
Auto Porshe;
Auto.greitis = 100;  //klaida
Porshe.greitis = 100;
if (Porshe.juda())
        cout<<”Vaziuoja Porshe “<<Porshe.greitis<<” km/h greičiu \n”;
return 0;
}
```

Klasės skelbime jos nariams gali būti nurodomi kreipties atributai. Jei šie atributai nėra nurodomi akivaizdžiai, pagal nutylėjimą kompiliatorius jiems priskiria *private* atributą. Tai reiškia, kad jie gali būti pasiekiami tik tos pačios klasės narių. *public* nariai gali būti pasiekiami bet kuriuo tos klasės objektu. Taigi sakinys `Porshe.mase = 1000;` *main()* funkcijoje bus klaidingas. Kartą nurodytas kreipties atributas galija visiems tos klasės nariams iki jos skelbimo galo arba ko nebus nurodytas naujas kreipties atributas.

Paprastai kuriant klases stengiamasi visus jų duomenis daryti *private* kreipties. Todėl reikalingos *public* funkcijos (kitaip metodai), kurios galėtų šiuos duomenis pasiekti bei modifikuoti. Toks klasės duomenų pasiekimo būdas leidžia atskirti duomenų saugojimo įpatumus nuo manipuliavimo jais įpatumų. Programuotojui visai nebūtina žinoti, kaip tie duomenys yra saugomi ir, jei jų saugojimo būdas bus kaip nors pakeistas, tai nesugadins tuos duomenis naudojančios programos veikimo. Taigi „viešos“ (*public*) duomenų pasiekimo funkcijos sukuria sąsajas tarp privačių klasės duomenų ir likusios programos, naudojančios juos, dalies. kiekviena tokia, kaip ir kitos, nesusijusios su duomenų mainais, klasės funkcijos turi turėti realizacijos dalį.

Klasei priklausančios funkcijos realizacija skiriasi nuo įprastos funkcijos realizacijos tuo, kad jos antraštė prasideda klasės pavadinimu, po kurio seka du dvitaškiai „::“ ir funkcijos-nario vardas, pvz. papildytas aukščiau pateiktos *Auto* klasės pavyzdys atrodys taip:

```
class Auto {
int mase;
public:
        int greitis;
         bool juda() {if (greitis > 0) return true; else return false;}
        int skaitykMase();
        void raskykMase(int m);
};

int Auto::skaitykMase()
{ return mase; }

void Auto::rasykMase(int m)
{mase = m;}

int main() {
Auto Porshe;
Porshe.greitis = 100;
if (Porshe.juda())
        cout<<”Vaziuoja Porshe “<<Porshe.greitis<<” km/h greičiu \nJos masė -
”<<Porshe.skaitykMase()<<” kg\n”;
return 0;
}
```

Čia funkcija *skaitykMase()* grąžina klasės kintamojo *mase* vertę, kadangi ji yra tos klasės narys ir turi priėjimą prie visų kitų jos narių. Analogiškai, funkcija *rasykMase()* turi tas pačias teises ir priskiria kintamajam *mase* skaitinę vertę.

Konstruktoriai ir destruktoriai

Skelbdami bazinio tipo kintamąjį programoje, galime jam iš karto priskirti ir vertę, pvz. `int mase = 1000;`. Klasės sukurtiems objektams to taip paprastai padaryti negalima, todėl šiam tikslui egzistuoja specialios funkcijos, vadinamos konstruktoriais. Konstruktorius gali turėti parametrų, tačiau jis negali grąžinti vertės (netgi *void*). Tai klasės metodas, kurio vardas yra toks pat, kaip ir klasės. Kadangi konstruktorius sukuria objektą bei jį inicializuoja, šiam objektui sunaikinti (kai jo jau nebereikia) naudojamas destruktorius. Jis atlaisvina objekto užimtą atmintį. Destruktorius skelbiamas panašiai, kaip ir konstruktorius – naudojant klasės vardą, prieš kurį seka tildės „~“ ženklas. Konstruktoriaus ir destruktoriaus skelbimo pavyzdys:

```
class Auto { Auto();  ~Auto() ....};
```

Jei konstruktorius ar destruktorius nėra tiesiogiai skelbiami, kompiliatorius tai atlieka pagal nutylėjimą. Tokie konstruktorius bei destruktorius neturi parametrų ir nedaro nieko kito, tik sukonstruoja bei sunaikina objektą.

Kiekvieną kartą sukuriant kokį nors objektą, iškviečiamas tos klasės konstruktorius. Jei pvz. *Auto* klasės konstruktorius turi du parametrus, iškvietimas bus `Auto Porshe(100,1200);`, jei vieną – tai `Auto Porshe(100);`, o jei nė vieno – tiesiog `Auto Porshe;`. Tai yra išimtis iš taisyklės, kad visos funkcijos turi turėti skliaustus, net jei ir neturi perduodamų parametrų. Net ir konstruktorius be parametrų gali turėti realizacijos dalį, kurioje klasės nariai kintamieji gali būti inicializuojami. Klasės konstruktorius, kaip ir bet kuri kita klasės funkcija, gali turėti keletą skirtingų realizacijų, besiskiriančių perduodamų parametrų tipu ar skaičiumi. Taigi, galimos visos trys aukščiau aprašytos *Auto* klasės *Porshe* konstruktoriaus realizacijos toje pačioje klasėje. Klasės su konstruktoriumi pavyzdys:

```
class Auto
{
private:
        int mase;
        int greitis;
public:
        Auto();
        Auto(int,int);
        ~Auto(){};

        int skaitykMase(){ return mase; }
        int skaitykGreiti() {return greitis;}

};

Auto::Auto()
{mase = 1000; greitis = 100;}
Auto::Auto(int m, int v)
{mase = m; greitis = v;}

int main()
{
int mase, greitis;
Auto Porshe;
cout<<"Porshe greitis "<<Porshe.skaitykGreiti()<<" km/h\nJos masė - "<<Porshe.skaitykMase()<<"
kg\n";
cin>>mase>>greitis;
```

```
Auto Ferrari(mase,greitis);
cout<<"Ferrari greitis "<< Ferrari.skaitykGreiti()<<" km/h\nJos masė - "<< Ferrari.skaitykMase()<<"
kg\n";
return 0;
}
```

Čia inicializuojami du klasės *Auto* objektai *Porshe* ir *Ferrari*. Pirmajam iškviečiamas konstuktorius be parametrų, priskiriantis šios klasės nariams *mase* bei *greitis* iš anksto numatytas vertes. Antrojo – *Ferrari* objekto minėtų kintamųjų inicializacijai konstruktoriui perduodamos klaviatūra įvestos vertės.

Klasės kintamuosius inicializuoti konstruktoriuje galima dar prieš riestinius skliaustus, taip vadinamoje konstruktoriaus inicializacijos fazėje:

```
Auto::Auto():
mase(1000),
greitis(100)
{ }
```

Tai atliekama rašant po konstruktoriaus antraštės dvitaškį „:“, po kurio išvardijami kintamųjų, kuriuos norima inicializuoti, vardai bei jiems priskiriamos vertės skliaustuose. Jei kintamųjų daugiau, nei vienas, jie atskiriami kableliais.

Be konstruktoriaus ir destruktoriaus pagal nutylėjimą kompiliatorius sukuria dar ir kopijų konstruktorių. Jis naudojamas tuo atveju, kai daroma objekto kopija ir kopijuoja visas tos klasės narių vertes į naują objektą.

Klasės nariais kintamaisiais gali būti ne tik bazinių tipų kintamieji, bet ir kitos klasės. Pavyzdžiui turime stačiakampį aprašančią klasę. Stačiakampis sudarytas iš atkarpų, kurios savo ruožtu turi pradžios ir pabaigos taškus, apibrėžiamus koordinatėmis. Taigi galima sukurti *Taskas* klasę, kurioje bus saugomos stačiakampio viršūnių koordinatės, ir įtraukti ją į *Keturkampis* klasę:

```
class Taskas
{
private:
        int tskX;
        int tskY;
public:
        void rasykX(int x) {tskX = x;}
        void rasykY(int y) {tskY = y;}
        int gaukX() {return tskX;}
```

```
        int gaukY() {return tskY;}
};

class Keturkampis
{
private:
        Taskas VirsusKaire;
        Taskas VirsusDesine;
        Taskas ApaciaKaire;
        Taskas ApaciaDesine;
public:
        void rasykVK(Taskas tsk) {VirsusKaire = tsk;}
        Taskas gaukVK() {return VirsusKaire;}
        ..........................................................
};

int main()
{
 int x,y;
 Taskas a;
 cin>>x>>y;
 a.rasykX(x);
 a.rasykY(y);
 Keturkampis kv;
 kv.rasykVK(a);
 cout<<"rasom koordinates :"<<kv.gaukVK().gaukX()<<" , "<<kv.gaukVK().gaukY()<<endl;
 return 0;
}
```

Struktūros

Struktūros yra daugeliu aspektų panašios į klases, išskyrus tai, kad jų nariai pagal nutylėjimą yra *public* kreipties. Struktūra skelbiama taip pat, kaip ir klasė, išskyrus tai, kad šiuo atveju skelbimui vartojamas bazinis žodis *struct* o ne *class*. C++ kalbos struktūros yra paveldėtos iš C kalbos, kurioje struktūros negalėjo turėti savo nariais funkcijų. Kaip jau minėta, C++ kalboje tarp jų nėra esminių skirtumų, išskyrus kreipties į narius būdą pagal nutylėjimą, todėl visas aukščiau aprašytas klasių formalizmas tinka ir struktūroms.

Masyvai

Programuojant būna situacijų, kai reikia sukurti ne vieną tam tikro bazinio tipo kintamąjį, bet visą jų grupę. C++ kalboje tokia to paties tipo kintamųjų grupė vadinama masyvu. Kintamieji vadinami masyvo elementais. Kiekvienas masyvas, prieš juo naudojantis, turi būti paskelbtas. Masyvo skelbimo sintaksė panaši į įprasto

kintamojo skelbimo sintaksę, t.y. pirmiausia rašomas masyvo kintamųjų tipas, po kurio seka masyvo vardas ir masyvo elementų kiekis, nurodomas sveikuoju skaičiumi laužtiniuose skliaustuose, pvz.:

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

Svarbu įsidėmėti tai, kad masyvo elementų kiekis nurodomas konstanta, t.y. jis turi būti žinomas dar prieš kompiliuojant programą. Kompiliatorius skiria masyvui kompiuterio atmintyje tiek vietos, kad jos užtektų visiems masyvo elementams patalpinti vieną po kito, pvz. prieš tai skelbto masyvo *skaiciai* elementams atmintyje reikės 5x4=20 baitų.

Masyvų inicializacija

Įprastu būdu paskelbus lokalų (pvz., galiojantį tik funkcijos ribose) masyvą, jo elementų reikšmės yra neapibrėžtos. Kita vertus, globalūs masyvai automatiškai užpildomi nuliais. Skelbiant masyvą yra galimybė suteikti jo elementams pradines reikšmes (t.y. jį inicializuoti), rašant jas riestiniuose skliaustuose „{}“ir atskiriant kableliais, pvz.:

```
int skaiciai [5] = { 16, 2, 77, 40, 127 };
```

Reikšmių kiekis riestiniuose skliaustuose „{}“ turi būti ne didesnis, nei stačiakampiuose skliaustuose „[]“ nurodytas masyvo elementų kiekis. Kai yra nurodomos masyvo elementu pradinės reikšmės, masyvo elementu kiekio stačiakampiuose skliaustuose galima ir nenurodyti. Tokiu atveju kompiliatorius skirs masyvui vietą atmintyje atsižvelgdamas į tai, kiek pradinių reikšmių nurodyta riestiniuose skliaustuose. Taigi, paskelbtame masyve *skaiciai* nenurodžius elementų skaičiaus, jame vis tiek bus 5 elementai, nes tiek pradinių reikšmių yra nurodyta.

Kreipimasis į masyvo elementus

Programoje galima kreiptis į bet kurį masyvo elementą nurodant jo indeksą. Indeksas nurodo, kiek konkretus masyvo elementas yra „pasislinkęs“ nuo masyvo pradžios. Svarbu įsidėmėti tai, kad masyvo elementų indeksacija prasideda nuo 0, t.y. pirmojo elemento indeksas yra 0, antrojo – 1 ir t.t. Taigi masyvo, kurį sudaro *n* elementų indeksų aibę sudarys sveikieji skaičiai nuo 0 iki *n*-1. Kreipiantis į masyvo elementą rašomas masyvo vardas ir to elemento indeksas laužtiniuose skliaustuose. Norint priskirti naują reikšmę pvz. trečiajam masyvo *skaiciai* elementui, reiktų rašyti tokį sakinį:

```
skaiciai [2] = 33;
```

Norint priskirti kintamajam *a* ketvirtojo to paties masyvo elemento reikšmę, rašomas toks sakinys:

```
int a = skaiciai [3];
```

Jeigu bandysime kreiptis į skaiciai[5], t.y. šeštąjį masyvo elementą – išeisime už masyvo ribų. C++ kalbos sintaksės prasme tai yra leistina – kreipiantis i elementus už masyvo ribų kompiliatorius klaidos nefiksuos, tačiau programos rezultatai bus klaidingi.

Svarbu įsidėmėti tai, kad skelbiant masyvą prieš jo vardą yra elementų tipą nurodantis bazinis žodis, tuo tarpu kreipiantis į tam tikrą masyvo elementą tokio žodžio nebūna.

Masyvo užpildymo atsitiktiniais skaičiais pavyzdys:

```
#include <iostream>
int main()
{
 const int kiekis = 10;
 int mas[kiekis];
 for (int i=0; i<kiekis; i++)
        mas[i] = rand() % 100 +1;
 cout << "Spausdinam rezultatus:\n“;
 for (int i=0; i<kiekis; i++)
        cout<<“mas[“<<i<<“] = “<<mas[i]<<endl;
system("PAUSE");
return 0;
}
```

Skelbti galima ne tik bazinių tipų, bet ir vartotojo kuriamų klasių objektų masyvus. Klasė šiuo atveju būtinai turi turėti konstruktorių be parametrų tam, kad kompiliatorius galėtų išskirti atmintį konstruojant masyvą. Kreipimasis į tokio masyvo elementus susideda iš dviejų dalių: pirmiausia identifikuojamas masyvo elementas laužtiniais skliaustais, po to taško operatoriumi kreipiamasi į tam tikrą klasės narį. Pavyzdys:

```
class Taskas
{
private:
        int tskX;
        int tskY;
public:
        void rasykX(int x) {tskX = x;}
        void rasykY(int y) {tskY = y;}
        int gaukX() {return tskX;}
        int gaukY() {return tskY;}
};

int main()
{
 Taskas koordinates[4] ;
```

```
 koordinates[0].rasykX(10) ;
koordinates[0].rasykY(5) ;
 cout<<"rasom koordinates :"<<koordinates[0].gaukX()<<" , "<< koordinates[0].gaukY()<<endl;
 return 0;
}
```

Daugiamačiai masyvai

C++ yra galimybė skelbti daugiau, nei vieno matmens masyvą. Kiekvienas papildomas matmuo žymimas naujais laužtiniais skliaustais, taigi dvimatis masyvas bus skelbiamas sakiniu `int mas[4][4] ;`, trimatis - `int mas[5][5][2] ;` ir t.t. Iš tiesų daugiamačiai masyvai yra tik programuotojų abstrakcija. Tokį patį rezultatą galima gauti naudojant vienmatį masyvą, kurio dydis būtų lygus daugiamačio masyvo indeksų sandaugai, pvz. `int mas[4][4] ;` galima pakeisti `int mas[16];`. Aišku, šiuo atveju keičiasi masyvo elementų indeksacija.

Masyvai kaip funkcijų parametrai

Kartais naudojant funkcijas gali prireikti perduoti joms masyvą kaip parametrą. C++ kalboje nėra galimybės perduoti visų masyvo elementų reikšmes. Funkcijos parametrui leidžiama perduoti tik kreipinio į funkciją argumentui – masyvui skirtos atminties adresą.

Kad masyvą galėtume naudoti kaip funkcijos parametrą, skelbiamos funkcijos antraštėje

toks parametras paskelbiamas kaip masyvas, tik po jo vardo stačiakampiuose skliaustuose „[]“ nerašoma nieko, pvz.:

```
void manoFunkcija (int mas[]);
```

Čia parametras vardu *mas* yra int tipo masyvas. Norint šiai funkcijai perduoti masyvą, paskelbtą kaip `int masyvas [10];`, pakanka tokio kreipinio:

```
manoFunkcija (masyvas);
```

Reikia pažymėti, kad perduodamo masyvo ilgis funkcijai nėra žinomas, todėl tam, kad indeksuodami masyvą neišeitume už jo ribų, funkcijai reikia perduoti ir jo ilgį.

Funkcijos parametru gali buti ir daugiamatis masyvas. Pvz., trimačio masyvo, kaip parametro, formatas yra: *tipas vardas [][skaičius][skaičius]*.

Funkcijos su daugiamačiu masyvu kaip parametru antraštės pavyzdys:

```
void manoFunkcija (int mas[][4][4]);
```

Šiuo atveju pirmuosiuose stačiakampiuose skliaustuose nerašoma nieko, o kituose skliaustuose turi būti sveikieji skaičiai, atitinkantys masyvo dydį.

Masyvo duomenų išvedimo į ekraną funkcijoje pavyzdys (komentarais pažymėti sakiniai, kuriuos turėtume naudoti esant dvimačiam masyvui):

```
#include <iostream>
const int kiekis = 10;

//  void spausdinti(int mas[][kiekis],int kiek)  //dvimačiam masyvui
void spausdinti(int mas[],int kiek)                //vienmačiam masyvui
{
cout << "Spausdinam rezultatus:\n";
 for (int i=0; i<kiek; i++)
            cout<<"mas["<<i<<"] = "<<mas[i]<<endl;   //vienmačiam masyvui
/*{
             for (int j=0; j<kiek; j++)
                        cout<<mas[i][j]<<" ";       //dvimačiam masyvui
             cout<<endl;
 }*/
}

int main()
{
  int masyvas[kiekis];         // vienmačiam masyvui
 // int masyvas[kiekis] [kiekis];        // dvimačiam masyvui
 for (int i=0; i<kiekis; i++)
            masyvas[i] = i*i;  //vienmačiam masyvui
/*for (int j=0; j<kiekis; j++)
                        masyvas[i][j] = (i+1)*(j+1);*/         //dvimačiam masyvui
 spausdinti(masyvas,kiekis);
system("PAUSE");
return 0;
}
```

Šioje programoje vienmačio masyvo atveju masyvo elementams bus priskirtos ciklo kintamojo *i* kvadrato vertės, o dvimačio masyvo atveju bus sugeneruota daugybos lentelė.

Veiksmai su masyvais

Masyvai yra galinga duomenų saugojimo priemonė. Turint didelį duomenų kiekį masyve dažnai reikia suskaičiuoti tam tikrą statistinę informaciją, pvz. masyvo elementų verčių vidurkį, rasti didžiausią ar mažiausią elementą ir pan. Žemiau pateiktame pavyzdyje pateikiama šių dydžių radimo realizacija:

```
int main()
{
  const int kiekis = 10;
 int masyvas[kiekis];
 for (int i=0; i<kiekis; i++)
            masyvas[i] = rand() % 100 +1;
int suma = 0;
int didziausias = masyvas[0];
for (int i=0; i<kiekis; i++)
{
            suma += masyvas[i];
            didziausias = (masyvas[i] > didziausias) ? masyvas[i] : didziausias;
```

```
}
cout<<"Didžiausias elementas : "<<didziausias
<<"\nMasyvo elementų vidurkis : "<<suma * 1.0 / kiekis<<"\n";
system("PAUSE");
return 0;
}
```

Kita svarbi masyvuose laikomų duomenų panaudojimo sritis yra jų rūšiavimas. Rūšiuotame masyve galima daug greičiau atlikti norimo elemento (pvz. didžiausio) paiešką, elementų išrinkimą pagal tam tikrą požymį ir pan. Rūšiavimo algoritmų yra gana nemažai. Žemiau pateikiamas vieno iš paprasčiausių ir žinomiausių – „burbuliuko“ algoritmas:

```
int main()
{
int kiek;
float skmas[100];
float tarpsk;
cout << "Kiek skaiciu noresite ivesti: ";
cin >> kiek;
cout << "Iveskite " << kiek << " skaicius:" << endl;
for (int m=0; m<kiek; m++)
cin >> skmas[m];

for (int m=0; m<(kiek-1); m++)
for (int n=0; n<(kiek-1-m); n++)
if (skmas[n] > skmas[n+1])
{
  tarpsk = skmas[n];
  skmas[n] = skmas[n+1];
  skmas[n+1] = tarpsk;
}
cout << "\nSurikiuoti skaiciai:\n";
for (int m=0; m<kiek; m++)
cout << skmas[m] << " ";
cout << endl;
return 0;
}
```

Rikiuojant lyginami du gretimi masyvo elementai ir, jeigu reikia, jie sukeičiami vietomis. Tokiu budu vieną kartą “prabėgus” visą masyvą, didžiausias elementas perkeliamas į jo galą. Po to šis ciklas cikle kartojamas, ir į antrą vietą nuo masyvo galo perkeliamas antras pagal dydį masyvo elementas ir t.t. Šiuo būdu galima rikiuoti ir daugiamačius masyvus. Skirtumas yra tas, kad pvz. dvimačiam masyvui surikiuoti prireiks vieno papildomo ciklo cikle, trimačiam – dviejų ir t.t.

Rodyklės

Viena iš labiausiai galingų C++ programavimo priemonių yra galimybė tiesiogiai manipuliuoti kompiuterio atmintimi. Tai atliekama naudojantis rodyklėmis. Kompiuterio atmintis yra padalinta į sritis, kurių kiekviena turi savo numerį, vadinama adresu. Adresai kompiuterio atmintyje numeruojami nuosekliai. Paprastai programuotojams nebūtina žinoti kintamųjų adresų atmintyje – pakanka jų vardų, kurie kompiliatoriaus yra transformuojami į adresus. Norint sužinoti kokio nors kintamojo adresą galima naudoti adreso paėmimo operatorių „&“:

```
int x;
cout << "Kintamojo x adresas : "<<&x;     //Rezultatas 0x22ff745
```

Net ir nežinant kintamojo adreso, jį galima išsaugoti rodyklėje. Rodyklė taip pat yra kintamasis, todėl ir jos skelbimas yra analogiškas įprastų kintamųjų skelbimui išskyrus tai, kad prieš kintamojo vardą rašoma „*“, pvz.:

```
int *rod = 0;
```

Čia skelbiama *int* tipo rodyklė **rod*. *int* reiškia, kad ji gali įsiminti *int* tip kintamojo adresą, t.y. kokio tipo yra rodyklė, tokio tipo adresą ji ir gali įsiminti. Rodyklė inicializuojama nuliu. Tokios rodyklės vadinamos nulinėmis. Rodyklės, kurioms nepriskirtos jokios vertės vadinamos „laukinėmis“ ir yra potencialiai labai pavojingos.

Norint rodyklę, inicializuotą nuliu, „priversti“ rodyti į kokio nors kintamojo adresą atmintyje, reikia jai tą adresą priskirti. Tai atliekama taip:

```
int x = 50;
int *rod = 0;
rod = &x;
```

Čia sukuriami *int* tipo kintamasis *x*, kuriam priskiriama vertė, lygi 50, bei nulinė rodyklė *rod*, kuriai priskiriamas kintamojo *x* adresas atmintyje (ne jo vertė!). Jei šiuo atveju nebūtų naudojamas adreso paėmimo operatorius, rodyklei *rod* būtų priskirta kintamojo *x* vertė, t.y. skaičius 50, kuris būtų traktuojamas, kaip adresas (greičiausiai neteisingas).

Turint rodyklę su priskirtu jai adresu, galima nesunkiai gauti toje atminties vietoje įrašytą vertę. Aukščiau nagrinėtame pavyzdyje, tai būtų kintamojo *x* vertė:

```
cout << "Kintamojo x vertė : "<<*rod;     //Rezultatas 50
```

Šiuo atveju „*“, dar vadinama nuorodos operatoriumi, prieš rodyklės vardą rodo, kad reikia išvesti į ekraną ne rodyklės, kuri iš tiesų saugo *x* kintamojo adresą atmintyje,

bet tuo adresu esančio duomens vertę, t.y. *x* reikšmę. Tai vadinama netiesioginiu kreipimųsi į kintamąjį. Taigi, nuorodos operatorius „*“ su rodyklėmis naudojamas dviem atvejais: kai reikia skelbti rodyklę jis nurodo, kad tai yra būtent rodyklė; kai reikia panaudoti duomenis, esančius rodyklės turimu adresu jis nurodo kreiptis į tą kompiuterio atminties adresą, kurį yra įsiminusi rodyklė. Svarbu įsidėmėti, kad nuorodos operatorius žymimas tuo pačiu simboliu, kaip ir daugybos operatorius. Kompiliatorius, nagrinėdamas programą, nustato kurį operatorių panaudoti kiekvienu konkrečiu atveju.

Rodyklių naudojimas

Kartą priskyrus rodyklei kintamojo adresą, naudojantis ja galima manipuliuoti to kintamojo duomenimis, pvz.:

```
int x = 5, y = 0;
int *rod_x = 0;
rod_x = &x;
int *rod_y = &y;
int z;
z = *rod_x;
cout<<”x =”<<x<<”, y = ”<<y<<”, z = ”<<z
<<", rod_x = "<<*rod_x<<", rod_y = "<<*rod_y<<endl ;endl;
y = 10;
cout<<"y ="<<y<<", *rod_y = "<<*rod_y<<endl ;
```

Pateiktame programos fragmente sukuriami trys *int* tipo kintamieji bei dvi to paties tipo rodyklės. Kintamieji *x* ir *y* inicializuojami tiesiogiai, o jų adresai priskiriami rodyklėms. Kintamasis *z* inicializuojamas netiesiogiai naudojant rodyklę *rod_x*, todėl pirmasis išvedimo į ekraną sakinys bus: *x = 5, y = 0, z = 5, rod_x = 5, rod_y = 0*. Kintamajam *y* priskyrus 10, tiek pat pakis ir rodykle *rod_y* rodoma vertė, kadangi ji susieta su šio kintamojo adresu. Kas gi saugoma pačioje rodyklėje:

```
cout<<"y adresas ="<<&y<<", rod_y = "<<rod_y<<", rod_y adresas = "<<&rod_y <<endl ;
```

Tiek kintamojo *y* adresas, tiek ir *rod_y* reikšmė bus ta pati (pvz. 0x22ff741), tuou tarpu pačios *rod_y* adresas kompiuterio atmintyje bus kitas (pvz. 0x22ff39). Reikia pažymėti, kad rodyklei skiriama tiek vietos atmintyje, koks yra jos tipas. Šiuo atveju tai yra 4 baitai, kadangi rodyklės tipas yra *int*. *double* tipo rodyklės atveju ji užimtų 8 baitus ir t.t.

Peržvelgus pateiktus pavyzdžius gali kilti klausimas, kam gi reikalingos tos rodyklės, jei kintamųjų vertes galima pasiekti tiesiogiai? Iš tiesų rodyklės naudojamos kitiems tikslams, t.y. išnaudoti globalią amintį, esančią už dėklo, skiriamo programos

lokaliems kintamiesiems saugoti, ribų, bei perduoti funkcijoms parametrus pagal nuorodą.

Dinaminis atminties skyrimas

Įprastai lokaliems kintamiesiems atmintis skiriama dėkle. Pagrindinis jų trūkumas yra tai, kad jie neegzistuoja, t.y. jie egzistuoja tik funkcijoje, kurioje jie apibrėžti. Funkcijai baigus darbą, šie kintamieji „dingsta“ – jie yra sunaikinami. Šią problemą išsprendžia dinaminis kintamųjų skelbimas, įgalinantis skirti jiems vietą atmintyje už dėklo ribų, t.y. taip vadinamoje „krūvoje“ (nuo angl. žodžio „heap“).

C++ atmintis dinamiškai skiriama operatoriumi *new*, po kurio seka objekto, kuriam norite skirti atmintį, tipas. Šio operatoriaus grąžinama vertė yra kintamojo adresas globalioje dinaminėje atmintyje. Skiriamos atminties dydis priklauso nuo objekto tipo, taigi tipo nurodymas tuo pačiu sako kompiliatoriui, kiek atminties reikia rezervuoti kiekvienu konkrečiu atveju. Norint paskelbti dinaminį pvz.*int* tipo kintamąjį, reikia rašyti:

```
int *px = new int ;
```

Jei operatorius *new* negali skirti atminties (atmintis visgi ribotas resursas), jis grąžina nulinę rodyklę. Taigi, visada pravartu patikrinti ar *new* rezultatas nėra nulis. Taip paskelbtą kintamąjį galima naudoti kaip ir bet kurią kitą rodyklę bei priskirti jam kokią nors vertę, pvz.:

```
*px = 50;
```

Tai reiškia, kad skaičius 50 bus įrašytas į atmintį tuo adresu, į kurį rodo rodyklė *px*. Kai duomuo išskirtu dinamiškai adresu nebereikalingas, atmintis turi būti išlaisvinama. Tai atliekama operatoriumi *delete*, pvz.:

```
delete px ;
```

Reikia įsidėmėti, kad pati rodyklė yra lokalus kintamasis – priešingai, nei išskiriama atmintis, todėl funkcijai, kurioje deklaruota ši rodyklė, baigus darbą, pastaroji išnyksta, kai tuo tarpu jos saugotu adresu esantys duomenys – ne ir ši atminties sritis tampa nebepasiekiama. Tai sukelia kompiuterio atminties praradimus (kitaip nuotėkius).

delete operatorius, veikdamas kokią nors rodyklę, iš tiesų tik panaikina duomenis tuo adresu, kurį ji rodo, todėl ji gali būti panaudota vėl. Svarbu nepanaudoti šio opreatoriaus rodyklei, kuriai atmintis jau buvo išlaisvinta, kadangi tai sukeltų

programos veiklos sutrikimus. Todėl kiekvienai „laisvai“ rodyklei pravatu priskirti 0. Dinaminio atminties skyrimo pavyzdys:

```
int main()
{
 int lokalusX = 10;
 int *lokaliRod = &lokalusX;
 int *globaliRod = new int;
 if (globaliRod == NULL)
            { cout<<"Klaida ! Nepaskirta atmintis";
              return 1 ;}
*globaliRod = 5 ;
 cout<<"Lokali rodyklė "<<*lokaliRod<<"\nGlobali rodyklė "<<*globaliRod;
delete globaliRod ;
 return 0;
}
```

Nors šiuo atveju globalios rodyklės naudotos atminties išlaisvinimas nėra būtinas, kadangi pasibaigus programai tai bus padaryta automatiškai, tai yra gero programavimo stiliaus pavyzdys.

Dinamiškai atmintis gali būti skiriama ne tik paprastiems kintamiesiems, bet ir objektams. Aminties skyrimo ir išlaisvinimo sintaksė išlieka nepakitusi, pvz.:

```
class Taskas {.....}
int main()
{
Taskas *pCentras = new Taskas;
.....
delete pCentras;
return 0;
}
```

Norint pasiekti lokaliai (dėkle) sukurto objekto narius naudojamas taško „.“ operatorius. Dinaminių objektų nariams pasiekti reikia naudoti rodyklę į konkretų objektą, todėl kreipimosi sintaksė tampa sudėtingesnė:

```
 (*pCentras).tskX = 10;
```

Kadangi toks kreipimosi būdas nėra labai patogus, C++ naudojamas specialus operatorius „–>“, sudarytas iš „minus“ ir „daugiau už“ simbolių dinamiškai sukurto objekto nariams pasiekti. Kreipimasis į *tskX* narį naudojant šį peratorių pakis į:

```
pCentras->tskX = 10;
```

Rodyklės, kaip funkcijų parametrai

Kaip jau buvo minėta, perduodant funkcijoms argumentus, perduodamos jų reikšmės ir nėra galimybių jas pakeisti. Norint tai padaryti, reikia perduoti argumentų adresus. Vienas iš būdų yra argumento adreso perdavimas naudojantis adreso paėmimo operatoriumi „&“, kitas – rodyklių panaudojimas. Abiem atvejais funkcijai perduodamas argumento adresas ir ji įgauna galimybę keisti perduoto kintamojo

vertę. Tai naudinga tuo atveju, kai norima grąžinti iš funkcijos daugiau nei vieną vertę, pvz.:

```
void swap(int *x, int *y);

int main()
{
 int x = 5, y = 10;
 cout << "Main() prieš swap(), x: " << x << " y: " << y << "\n";
 swap(&x,&y);
 cout << "Main() po swap(), x: " << x << " y: " << y << "\n";
 return 0;
}

void swap (int *px, int *py)
{
 int temp;
 cout << "Swap() prieš sukeitimą, *px: " << *px << " *py: " << *py << "\n";
 temp = *px;
 *px = *py;
 *py = temp;
 cout << "Swap() po sukeitimo, *px: " << *px << " *py: " << *py << "\n";
}
```

Čia funkcijoje *swap()* sukeičiamos kintamųjų *x* ir *y* vertės. Šiuo atveju kreipiantis į funkciją tiems parametrams, kurie yra rodyklės, perduoti būtina panaudoti adreso paėmimo operatorių. Priešingu atveju kompiliaorius sugeneruos klaidą.

Kitas svarbus rodyklių panaudojimo funkcijų parametrams atvejis yra sudėtingų struktūrų, klasių objektų perdavimas. Perduodant funkcijai pvz. klaės objektą įprastu būdu, daroma jo kopija, t.y. iškviečiamas kopijų konstruktorius, nukopijojantis visų tos klasės narių duomenų reikšmes į dėklą. Jei funkcija grąžina objektą, tai ši procedūra pakartojama dar kartą (be to, abu lokalius objektus dar reikia ir ištrinti iš atminties – dukart iškviečiamas destruktorius). Esant pakankamai dideliems objektams tai labai neefektyvu tiek naudojamos atminties, tiek ir programos greitaeigiškumo požiūriu. Objekto perdavimo funkcijai sintaksė nesiskiria nuo paprasto kintamojo perdavimo.

Rodyklės ir masyvai

Masyvai yra glaudžiai susiję su rodyklėmis. C++ masyvo vardas yra const tipo rodyklė į pirmąjį masyvo elementą. Taigi skelbiant masyvą, pvz.:

```
int skaiciai[10];
```

masyvo vardas *skaiciai* yra tiesiog rodyklė į pirmąjį jo elementą `&skaiciai[0]`. Taigi galimas toks priskyrimas:

```
int *p = skaiciai;
```

Priešingas priskyrimas negalimas, nes rodyklė *skaiciai* traktuojama kaip konstanta. Su rodyklėmis galima atlikti sudėties bei atimties veiksmus, tačiau jos turi savų ypatybių. Priešingai, nei kitiems kintamiesiems, rodyklės reikšmės padidinimas ar sumažinimas vienetu perstumia ją atmintyje per tiek baitų, kiek užima tos rodyklės duomenų tipas (pvz. per 4 *float* atveju), todėl jos didinimas ar mažinimas leidžia pasiekti bet kurį su šia rodykle susieto masyvo elementą, pvz.:

```
int antras = *(p++);
```

kintamajan *antras* bus priskirta antro masyvo *skaiciai* elemento vertė. Žemiau pateiktame pavyzdyje parodoma keletas manipuliavimo masyvo elementais naudojantis rodykle būdų:

```
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;
}
```

Ši programa išveda į ekraną vertes 10, 20, 30, 40, 50. Šios skaitinės vertės masyvo elementams priskiriamos rodykle *p*.

Masyvai ir dinaminė atmintis

Visi aukščiau aprašyti masyvai buvo sukurti dėkle, tačiau masyvą skelbti galima ir už dėklo ribų. Tai atliekama operatoriumi *new*, po kurio seka masyvo elementų tipas ir jų kiekis laužtiniuose skliaustuose, pvz.:

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

Tokio masyvo elementus galima pasiekti naudojantis tiek rodyklių aritmetika, tiek ir naudojantis indeksais laužtiniuose skliaustuose, pvz.:

```
int main ()
{
 int *skaiciai = new int[5];
skaiciai[0] = 10;
*(skaiciai+1) = 20;
cout<<”masyvo[0] elementas:”<<*skaiciai<<endl;
cout<<”masyvo[1] elementas:”<<skaiciai[1]<<endl;
return 0;
}
```

Kai dinaminis masyvas tampa nebereikalingas, jam skirta atmintis išlaisvinama operatoriumi *delete []*:

```
delete [] skaiciai;
```

Laužtiniai skliaustai reiškia, kad išlaisvinama masyvui, o ne paprastam kintamajam skirta atmintis.

Simbolių masyvai ir eilutės

C++ kalboje galima sukurti ir *char* tipo (simbolių) masyvą. Toks masyvas, kurio paskutinis simbolis yra '\0', vadinamas simbolių eilute. *char* masyvo skelbimo pavyzdys:

```
char mas[5] = {'A','l','u','s','\0'};
```

Toks simbolių eilutės skelbimo būdas nėra patogus, todėl C++ galimas kompaktiškesnis eilutės skelbimas naudojant dvigubas kabutes:

```
char mas[5] = ”Alus”;
```

Šiuo atveju nebereikia nulio simbolio eilutės pabaigoje – kompiliatorius jį prideda pats. Galima deklaruoti ir neinicializuotus masyvus. Tokio masyvo pavyzdys:

```
char buffer[80];
cout << "Iveskite eilute: ";
cin >> buffer;
cout << "Buferio turinys:  " << buffer << endl;
```

Šioje programoje simboliai įvedami klaviatūra jos vykdymo metu. Šis programos fragmentas turi du trūkumus: pirmiausia, vartotojas turi įvesti ne daugiau 79 simbolių tam, kad neperpildytų masyvo, be to, įvedus tarpo simbolį *cin* traktuos jį kaip eilutės pabaigą ir nutrauks įvedimą. Norint ištaisyti šiuos trūkumus, galima naudoti specialų *cin* objekto metodą – *get()* funkciją. Šiai funkcijai reikia perduoti tris parametrus: rodyklę į buferį, saugantį įvestą informaciją, maksimaliai galimą įvesti simbolių skaičių, simbolį, kuris baigia įvedimą (pagal nutylėjimą tai yra naujos eilutės simbolis). Naudojant *get()* funkciją aukščiau esantis programos fragmentas turi būti transformuotas į:

```
char buffer[80];
cout << "Iveskite eilute: ";
cin.get(buffer, 79);
cout << "Buferio turinys:  " << buffer << endl;
```

Simbolių masyvai dar vadinami C stiliaus (arba tiesiog C) eilutėmis. C++ paveldėjo iš C funkcijų biblioteką darbui su šiomis eilutėmis. Panagrinėsime kai kurias iš jų:

strcpy(eilute1, eilute2) – kopijuoja eilutę *eilute2* į *eilute1*;

strcat(eilute1, eilute2) – prideda eilutę *eilute2* į *eilute1* galą;

strlen(eilute1) – grąžina eilutės *eilute1* ilgį;

strcmp(eilute1, eilute2) palygina eilutes *eilute1* ir *eilute2*. Jei eilutės lygios, funkcija grąžina 0, jei *eilute1* daugiau už *eilute2* – teigiamą skaičių, *eilute1* mažiau už *eilute2* – neigiamą skaičių.

Pavyzdys, iliustruojantis šių funkcijų veikimą:

```
int main ()
{
 char Eilute1[] = "Vardenis";
char Eilute2[20];
strcpy(Eilute2,"Pavardenis");
cout <<Eilute1<< " , " << Eilute2 << endl;
int ilgis = strlen(Eilute1)+strlen(Eilute2);
char *Eilute3 = new char[ilgis];
strcpy(Eilute3,Eilute1);
strcat(Eilute3,Eilute2);
cout << "Sudedam eilutes :" << Eilute3 << endl;
if (strcmp(Eilute1,Eilute2) > 0)
        cout<<Eilute1<<" daugiau uz "<<Eilute2<< endl;
else
        cout<<Eilute1<<" maziau uz "<<Eilute2<< endl;
return 0;
}
```

Čia sukuriamos dvi eilutės *Eilute1* ir *Eilute2*. Pirmajai reikšmė priskiriama inicializacijos metu, o antrajai – naudojant *strcpy()* funkciją. Trečioji eilutė *Eilute3* skelbiama dinamiškai skiriant jai tiek vietos atmintyje, kiek užima *Eilute1* ir *Eilute2* kartu sudėjus. Tam dukart panaudojama *strlen()* funkcija. Į *Eilute3* patalpinamos *Eilute1* ir *Eilute2* pirmiausia kopijuojant į ją *Eilute1*, o po to pridedant *Eilute2*. Galiausiai *Eilute1* ir *Eilute2* yra palyginamos tarpusavyje *strcmp()* funkcija. Ši funkcija lygina eilutes po vieną simbolį nuo jų pradžios, palygindama jų ASCII kodus. Taigi, pateiktame pavyzdyje jau pirmasis *Eilute1* simbolis didesnis už *Eilute2* simbolį.

Pagrindinis tokių eilučių trūkumas yra tas, kad jos yra riboto ilgio ir, naudojant tokias funkcijas, kaip *strcpy()* ar *strcat()* lengva viršyti masyvo, skirto tos eilutės simboliams saugoti, ribas. Tokiu atveju pravartu naudoti *strncpy(eilute1, eilute2, ilgis)* bei *strncat(eilute1, eilute2 , ilgis)* funkcijas, kurios kopijuoja ar prideda tiek simbolių iš *eilute2* į *eilute1*, kiek nurodyta *ilgis* parametre. Pvz. saugus *Eilute2* reikšmės priskyrimas paskutiniajame pavyzdyje būtų:

```
strncpy(Eilute2,"Pavardenis",19);
```

Naudojantis šiomis funkcijomis reikia turėti omenyje, kad tuo atveju, kai kopijuojama eilutė iš tiesų ilgesnė už tą, į kurią kopijuojama, pastarosios gale nėra patalpinamas `'\0'` simbolis, Taigi, tuo reikia pasirūpinti pačiam programuotojui.

C++ eilės

C++ yra sukurta klasių, manipuliuojančių simbolių eilutėmis, biblioteka. Norint ja pasinaudoti, reikia įtraukti į programą nuorodą kompiliatoriui:

```
#include <string>
```

Po to jau galima skelbti eilutes, kaip ir bet kuriuos kitus kintamuosius bei priskirti joms pradines vertes:

```
string Eilute1 = "Vardenis";
```

C++ eilutės neturi tokių apribojimų, kaip jau anksčiau aptartos C eilutės. Eilutė gali būti praktiškai begalinio ilgio, kurį riboja tik kompiuterio atminties resursai. Šios eilutės taip pat gali būti traktuojamos kaip simbolių masyvai, taigi kiekvienas jų simbolis gali būti pasiekiamas „[]“ operatoriumi. Kadangi šiuo būdu galima kreiptis ir už eilutės ribų, tokio kreipimosi rezultatai gali būti nenuspėjami, todėl dažnai naudojamas saugesnis būdas pasitelkiant *string* klasės funkciją *at(index),* pvz.:

```
string Eilute1 = "Vardenis";
cout<<”Pirmas simbolis: “<<Eilute1.at(0);
```

C++ eilutės ilgis gali būti gaunamas naudojantis *string* klasės funkciją *length(),* pvz.:

```
cout<<”Eilutes ilgis: “<<Eilute1.length();
```

Eilutės gali būti lengvai sudedamos pasinaudojant sudėties „+“ operatoriumi. Žemiau pateiktas pavyzdys demonstruoja vardo bei pavardės (taip pat tarpo tarp jų) sumavimą į vieną eilutę:

```
int main ()
{
 string Eilute1 = "Vardenis";
 string Eilute2 = "Pavardenis";
 string Eilute3 = Eilute1 + “  “ + Eilute2;
cout << "Sudedam eilutes :" << Eilute3 << endl;
return 0;
}
```

Norint nukopijuoti dalį eilutės, naudojama *string* klasės funkcija *substr(first, count).* Ši funkcija grąžina *count* eilutės simbolių pradedant *first* indeksu. Jei *first* indeksas yra už eilutės ribų, generuojama išskirtinė situacija ir programa baigia darbą. Jei *count* yra per didelis, funkcija grąžina visus simbolius nuo *first* iki eilutės galo. Toks pat rezultatas gaunamas, jei *count* iš viso neįvedamas, pvz.:

```
string Eilute = "Mano knyga";
cout<<”Pirmas zodis: “<<Eilute.substr(0, 4)<<endl;
```

C++ eilutės gali būti lengvai palyginamos. Tai atliekama operatoriais „>“, „<“, „>=“, „<=“, „==“, „!=“. Norint rasti simbolį ar eilutės dalį kitoje eilutėje, galima naudoti

string klasės funkciją *find(str, offs)*. Čia *str* yra eilutė arba simbolis, kurio ieškoma, *offs* – poslinkis nuo eilutės pradžios ieškant *str*, pvz.:

```
int main ()
{
 string Eilute = "Tai yra testas";
 cout << "Skaidom eilute zodziais:" << endl;
unsigned int ind, pos = 0;
do
{
            ind = Eilute.find(' ',pos);
            cout<< Eilute.substr(pos,ind-pos)<<endl;
            pos = ind + 1;
}
while(ind<Eilute.length());
 return 0;
}
```

string klasėje yra ir daugiau naudingų funkcijų: *replace(first, count,subst_eil)* keičia *count* simbolių eilutėje eilutės *subst_eil* simboliais pradedant *first*-uoju; *insert(index,ins_eil)* įterpia *ins_eil* eilutę pradedant *index* simboliu; *erase(index,count)* pašalina iš eilutės *count* simbolių pradedant nuo *index* ir t.t.

string eilutėms galima lengvai priskirti anksčiau aptartus simbolių masyvus:

```
char mas[20]="Simboliu masyvas";
string Eilute = mas;
```

Atvirkštinis priskyrimas yra negalimas, kadangi *string* eilutė nėra simbolių masyvas todėl šiam tikslui naudojama *c_str()* funkcija:

```
char mas[20];
const char *rodykle=mas;
string Eilute = “Cia yra C++ eilute”;
rodykle=Eilute.c_str();
```

Norint eilutės turinį konvertuoti į skaitinę reikšmę arba atvirkščiai, galima pasinaudoti *stringstream* biblioteka įterpiant nuorodą kompiliatoriui <*sstream*>. Pavyzdys:

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

int main ()
{
int nr,metai,menuo,diena;
string vardas,pavarde;
stringstream srautas;
string Eilute("Studento id");

cin>>nr;
srautas<<Eilute<<":"<<nr;
cout<<srautas.str()<<endl;

Eilute="Vardenis Pavardenis 2000 01 01";
srautas.str(Eilute);
srautas>>vardas>>pavarde>>metai>>menuo>>diena;
cout<<"Duomenys:\n"<<vardas<<"\t"<<pavarde<<"\ngimimo data : "
```

```
   <<metai<<"."<<menuo<<"."<<diena<<endl;
system("PAUSE");
return 0;
}
```

Čia sukuriamas *stringstream* tipo objektas *srautas* ir į jį ivedama simbolių eilutės *Eilute, ':'* bei *int* tipo kintamojo *nr*, įvesto klaviatūra, vertės operatoriais „<<“. Norint išvesti suformuotą simbolių srautą į ekraną, naudojama funkcija *str()*, paverčianti jį *string* tipo eilute. Srauto objektas taip pat gali būti inicializuotas *string* eilute. Tai atliekama iškviečiant tą pačią funkciją *str()* bei, skirtingai nei pirmame kvietime, jai perduodant eilutę *Eilute*. Vėliau iš simbolių srauto duomenys paimami „>>“ operatoriais ir priskiriami atitinkamo tipo kintamiesiems, kurių vertės išvedamos į ekraną.

Duomenų srautai

C++ kalboje nėra nustatyta, kaip duomenys yra rašomi į ekraną ar išorinį failą, taipogi, kaip iš jo įvedami į programą. Vietoj to C++ yra klasių biblioteka *iostream*, kuri palengvina įvedimą bei išvedimą. Tai leidžia lengvai pritaikyti C++ skirtingoms operacinėms sistemoms – tereikia pateikti skirtingas bibliotekas.

iostream klasės traktuoja perduodamus duomenis kaip duomenų srautą (bitas po bito) kuris gali būti nukreiptas iš programos į monitoriaus ekraną ar failą, arba atvirkščiai – iš klaviatūros ar failo į programos kintamiesiems skirtą vietą atmintyje. Svarbiausia srauto užduotis yra išlaisvinti programuotoją nuo išvedimo ar įvedimo į išorinį įrenginį detalių. Programuotojas tik siunčia duomenis į srautą arba ima iš jo.

Rašymas į išorinį diską yra pakankamai „brangus“ – jis sunaudoja santykinai daug laiko, todėl rašymo į diską ar skaitymo iš jo metu programa yra praktiškai blokuojama. Tam, kad išvengti šios problemos srautai naudoja buferius. Siunčiami į srautą duomenys yra persiunčiami į buferį ir tik jam visiškai užsipildžius vyksta duomenų rašymas į diską.

Įvedimo ir išvedimo srautai

C++ srautams ir buferiams kurti buvo panaudoti objektinio programavimo principai bei sukurtos klasės srautams ir buferiams realizuoti:

- *streambuf* klasė valdo buferį. Šios klasės nariai funkcijos sudaro galimybę jį užpildyti, išvalyti, paleisti į srautą ir pan.
- *ios* klasė yra bazinė įvedimo/išvedimo srautų klasė.

- *iostream* klasė, paveldinti *istream* ir *ostream* klasių funkcionalumą, skirta duomenų srautams iš klaviatūros ir į ekraną valdyti.
- *fstream* klasė valdo duomenų įvedimą bei išvedimą iš išorinio failo.

Kai C++ programa, į kurią įtrauktas <*iostream*> failas pradedama vykdyti, sukuriami keturi objektai:

1) *cin* valdo standartinį įvedimą iš klaviatūros;
2) *cout* valdo standartinį išvedimą į ekraną;
3) *cerr* valdo nebuferizuotą išvedimą į standartinį klaidų įrenginį (ekraną);
4) *clog* valdo buferizuotą klaidų išvedimą į standartinį kladų įrenginį (ekraną), kuris dažnai būna nukreipiamas į išorinį *log* failą.

Kiekvienas iš standartinių įvedimo bei išvedimo įrenginių gali būti peradresuotas, t.y. nukreiptas kita kryptimi, pvz. yra įprasta į standartinį klaidų įrenginnį siunčiamus duomenis nukreipti į išorinį failą.

Įvedimas naudojant *cin*

Globalų objektą *cin* galima naudoti programoje įtraukus <*iostream*> failą. Iš anksčiau pateiktų pavyzdžių žinoma, kad paskelbus kokį nors kintamąjį, pvz.:

```
int Kintamasis;
```

cin objektui galima panaudoti >> operatorių, kuris paima iš buferio duomenis ir bando juos įrašyti į *Kintamasis* kintamąjį. Kadangi šis kintamasis gali būti įvairiausių tipų *cin* turi „mokėti“ tinkamai juos prskirti, todėl egzistuoja daug >> operatoriaus variantų skirtingiems duomenų tipams apdoroti (analogiškai funkcijoms vienodais vardais). Kadangi iš tiesų įrašant duomenis į kintamąjį naudojamasi jo adresu, o ne verte, >> operatorius gali tiesiogiai veikti orginalų kintamąjį.

cin gali įrašyti duomenis ne tik į bazinių tipų kintamuosius, bet ir simbolių masyvus:

```
char eilute[20];  cin>>eilute;
```

Įvedus žodį, pvz. *Vardas*, kintamasis *eilute* bus užpildytas simboliais *V,a,r,d,a,s,\0*. Paskutinį – nulio sumbolį – *cin* prideda automatiškai (aišku, masyve turi būti pakankamai vietos jam įrašyti). Kaip jau buvo minėta, toks įvedimo būdas turi ir savų trūkumų – įvedus tarpo simbolį *cin* traktuoja jį kaip eilutės pabaigą ir įterpia toje vietoje nulio simbolį.

cin gali būti naudojamas daugkartiniam įvedimui, pvz.:

```
cin>>KintVienas>>KintDu>>KintTrys;
```

Čia duomenys į vedami iš karto į tris kintamuosius *KintVienas*, *KintDu* ir *KintTrys*. Įvedamos vertės atskiriamos naujos eilutės ar tarpo simboliais. Tai įmanoma, kadangi paėmimo iš srauto operatorius >>, kurį galime traktuoti kaip tam tikrą funkciją, grąžina *istream* objektą. Kadangi *cin* priklauso *istream*, tai šis grąžinamas objektas gali būti naudojamas naujam paėmimo iš srauto operatoriui. Tai galima pavaizduoti kaip:

```
((cin>>KintVienas)>>KintDu)>>KintTrys;
```

Be >> operatoriaus *cin* turi ir kitų narių funkcijų. Viena iš tokių – *get()* – yra skirta simboliui iš srauto paimti. *get()* gali būti naudojama be parametrų. Tokiu atveju naudojama jos grąžinama vertė, kuri gali būti įvestas simbolis arba konstanta *EOF* (end of file), jei failo pabaiga buvo pasiekta. Tipinio *cin. get()* naudojimo pavyzdys:

```
int main ()
{
char ch;
while ( (ch = cin.get()) != EOF)
{
        cout << "ch: " << ch << endl;
}
cout << "\nBaigta!\n";
return 0;
}
```

Ši programa skaito įvedamus klaviatūra simbolius, kol failo pabaigos simbolis aptinkamas (*ctrl+Z* Windows ar DOS sistemoje). Kadangi funkcijos *get()* grąžinamas tipas nėra *istream* objektas, ji negali būti naudojama daugkartiniam įvedimui.

cin. get(char &) gali būti naudojama ir su parametru, kuris šiuo atveju yra nuoroda į simbolį, užpildoma sekančia simbolio iš srauto verte. Funkcija grąžina *istream* objektą, todėl šiuo atveju galimas daugkartinis įvedimas:

```
int main ()
{
char a,b,c;
cout<<”Iveskite tris raides:”<<endl;
cin.get(a).get(b).get(c);
cout << "a: " << a << "b: " << b<< "c: " << c << endl;
return 0;
}
```

Funkcija *get()* gali būti naudojma ir simbolių masyvui (eilutei) užpildyti. Bendru atveju *get()* funkcijai reikia perduoti tris parametrus: rodyklę į simbolių masyvą, maksimaliai galimą nuskaityti simbolių skaičių bei simolių sekos nutraukimo simbolį, kuris pagal nutylėjimą yra *'\n'*. Jei šis simbolis gaunamas iš srauto anksčiau, nei pasiekiamas maksimaliai nuskaityti leistinas simbolių skaičius, masyve įrašomas nulio simbolis, o sekos nutraukimo simbolis paliekamas buferyje, pvz.:

```
char eil[256];
cout<<”Iveskite eilute:”<<endl;
cin.get(eil,255);
```

Kitas būdas įvesti simbolių eilutei yra *getline()* funkcijos naudojimas. Ji beveik identiška *get()* funkcijai išskyrus tai, kad nepalieka sekos nutraukimo simbolio buferyje. Simbolių eilutės įvedimo pavyzdys:

```
int main ()
{
char eilute1[256];
char eilute2[256];
char eilute3[256];
cout << "Iveskite  pirma eilute: ";
cin.get(eilute1,255,'Q');
cout << "Pirma eilute: " << eilute1 << endl;
cout << "Iveskite  antra eilute: ";
cin >> eilute2;
cout << "Antra eilute: " << eilute2 << endl;
cout << "Iveskite  trecia eilute: ";
cin.getline(eilute3,255);
cout << "Trecia eilute: " << eilute3 << endl;
return 0;
}
```

Šioje programoje skelbiami trys simbolių masyvai. Jie užpildomi skirtingais būdais: pirmasis – *get()* funkcija, kurios sekos nutraukimo simboliu pasirinktas 'Q' simbolis, antrasis – paėmimo iš srauto operatoriumi ir paskutinis – funkcija *getline()*. Taigi, įvedus pvz. tokią pirmą eilutę: „Vienas Qdu trys“, *eilute1* masyvui bus priskirtas „Vienas “ žodis. Kadangi Q simbolis paliekamas įvedimo buferyje, tai *eilute2* automatiškai įgis “Qdu”, o *eilute3* – „trys“ vertes, nes >> operratoriui aptikus įvedimo sraute tarpo simbolį, įvedimas nutrūksta ir įvedimo buferis lieka netuščias. Norint, kad antroje eilutėje nebūtų Q simbolio, reiktų naudoti *getline()* funkciją, kuri pašalina skiriamąjį simbolį iš srauto.

Išvedimas naudojant *cout*

Iki šiol *cout* buvo vartojamas kartu su išvedimo į srautą operatoriumi << išvesti į ekraną įvairiausių tipų duomenims. Kaip *cin* atveju, *cout* objektas taipogi gali būti naudojamas su jo nariais funkcijomis *put()* ir *write()*. *put()* funkcija tiesiog išveda simbolį į srautą. Kadangi ji grąžina *ostream* objektą, o *cout* yra pastarojo narys, tai grąžinamą reikšmę galima panaudoti pakartotiniam išvedimui, pvz.:

```
cout.put('L').put('a').put('b').put('a').put('s');
```

write() funkcija išveda duomenis panašiai, kaip ir << operatorius, išskyrus ta, kad jai reikia papildomai pateikti išvedamų simbolių kiekį:

```
char eilute[] = “Labas”;
cout.write(eilute, strlen(eilute));
```

Iki šiol nedarydavome skirtumo tarp perkėlimo į naują eilutę simbolio *\n* ir *endl*. Iš tiesų *endl* papildomai iškviečia *cout* funkciją *flush()*, kuri priverstinai nukreipia buferyje esančius duomenis į išvedimo srautą net jei buferis nėra iki galo užpildytas. Tą galima padaryti ir tiesogiai, pvz. naudojantis manipuliatoriumi *flush*:

```
cout<<flush;
```

cout gali valdyti išvedamos informacijos vaizdavimą ekrane. Paprastai išvedant informaciją į ekraną jame skiriama tiek vietos, kiek reikia tai informacijai pavaizduoti. Viena iš šį vaizdavimą keičiančių funkcijų yra *width()*. Jos skliausteliuose nurodomas argumentas yra išvedamo lauko plotis (simbolių kiekis). Jei nurodyto simbolių kiekio nepakanka informacijai išvesti, *width()* yra ignoruojama. Ši funkcija pakeičia tik artimiausiai išvedamai informacijai skirto lauko dydį, o po to vėl grįžtama prie įprastinio išvedimo.

Kitas būdas išvedimui valdyti realizuojamas naudojantis manipuliatoriais. Norint jais naudotis, į programą reikia įterpti <*iomanip*> failą. Čia *setw()* manipuliatorius yra analogiškas *width()* funkcijai ir nurodo informacijai išvesti skirto lauko dydį, (jis nurodomas skliausteliuose). *setprecision()* manipuliatoriaus skirtas skaitmenų skaičiui išvedamoje reikšmėje nustatyti. Šiuo atveju manipuliatoriaus skliausteliuose nurodoma reikšmė galioja iki sekančio nurodymo. Išvedimo formatavimo pavyzdys:

```
int main ()
{
float A = 3.1415;
cout << "Pradinis A=" << A << endl;
cout << "wide(9)  A=";
cout.width(9);
cout << A << endl;
cout << "precision(2) A=" << setprecision(2)
<< A << endl;
cout << "wide(10) A=" << setw(10) << A << endl;
A = 12.345678;
cout << "Naujas A=" << A << endl;
cout << "precision(4) A="
<< setprecision(4)<< A << endl;
return 0;
}
```

Išvedimas/įvedimas į išorinį failą

Srautai suteikia galimybę operuoti duomenimis, gaunamais iš klaviatūros ar kietojo disko, o taip pat duomenimis, siunčiamais į ekraną ar išorinį failą naudojant tuos pačius paėmimo/išvedimo į srautą operatorius >> ir <<. Norint paruošti (atidaryti)

išorinį failą skaitymui ar rašymui būtina sukurti *ifstream* ar *ofstream* objektą. Tuo tikslu į programą reikia įterpti <*fstream*> failą.

Rašant į failą ar skaitant iš jo pirmiausia sukuriamas *ofstream* ar *ifstream* tipo objektas ir susiejamas su konkrečiu failu išoriniame diske nurodant jo vardą (ir kelią iki jo, jei yra ne tame pačiame aplanke, kaip ir programa) konstruktoriui, pvz.:

```
ifstream sfailas("duomenys.txt");
ofstream rfailas("duomenys.txt");
```

Kiekvienas srautas, nesvarbu ar atidarytas įrašyti, ar nuskaityti duomenims, turi būti uždarytas baigus darbą funkcija *close()*. Ji užtikrina, kad naudotas failas nebus sugadintas ir visi rašomi duomenys iš tiesų nusiųsti į srautą.

Susiejus srautų objektus su išoriniais failais, galima juos naudoti taip pat, kaip ir anksčiau aptartus srautų į ekraną bei iš klaviatūros objektus *cout* ir *cin*. Pavyzdys:

```
#include <fstream>

int main ()
{
char fVardas[80];
char tekstas[256];
int sk1;
float sk2;
cout << "Iveskite failo pavadinima: ";
cin >> fVardas;

ofstream rfailas(fVardas);
rfailas << "Skaiciai, ivesti klaviatura...\n";
cout << "Iveskite du skaicius: ";
cin >> sk1 >> sk2;
rfailas << sk1 << " " << sk2;
rfailas.close();

ifstream sfailas(fVardas);
cout << "\nFailo turinys:\n";
sfailas.get(tekstas,255);
cout << tekstas<<endl;
sfailas >> sk1;
cout << "pirmas skaicius: " << sk1 << endl;
sfailas >> sk2;
cout << "antras skaicius: " << sk2 << endl;
cout << "\n*****Failo pabaiga.*****\n";
sfailas.close();
return 0;
}
```

Šioje programoje naudojami objektai *rfailas* ir *sfailas*, rašymui į tekstinio tipo failą ir skaitymui iš jos. Duomenys siunčiami į srautą operatoriumi <<, kuris kintamųjų *sk1* ir *sk2* vertes transformuoja į simbolių srautą, todėl juos būtina atskirti, pvz. tarpo simboliu. Nuskaitant pirmą teksto eilutę faile naudojama *get()* (galima ir *getline()*) funkcija, nes naudodami šiam tikslui >> operatorių gautume tik pirmą žodį.

Pateiktos programos ribotumas – nuskaitant informaciją reikia tiksliai žinoti, kiek ir kokio tipo duomenų yra pasirinktame faile. Dažnai ši informacija būna iš anksto nežinoma, todėl pravartu žinoti, ar skaitant iš failo jau pasiekta jo pabaiga ir at tiesiog pavyko jį atidaryti. *iostream* objektai turi galimybę tikrinti įvedimo bei išvedimo būseną loginio tipo funkcijomis *eof(), bad(), fail(), good(), is_open()*. Funkcija *eof()* grąžina *true* vertę, jei pasiekta failo pabaiga, t.y nuskaitytas „*EOF*“ simbolis. *bad()* funkcija grąžina *true* vertę, jei buvo atlikta neteisinga operacija, o *fail()* – jei buvo atlikta neteisinga operacija arba operacija nepavyko. Galiausiai *good()* funkcija grąžina *true* vertę, jei prieš tai minėtos trys funkcijos grąžina *false*. Funkcija *is_open()* tikrina, ar pavyko norimą failą atidaryti skaitymui ar rašymui. Programos, nuskaitančios nežinomo ilgio tekstą iš failo pavyzdys:

```
#include <fstream>

int main ()
{
char fVardas[80];
char tekstas[256];
cout << "Iveskite failo pavadinima: ";
cin >> fVardas;
ifstream sfailas(fVardas);
if (sfailas.is_open())
  cout<<"Atidarem, skaitom\n";
else
{
  cout<<"Nepavyko atidaryti\n";
  return 1;
}
cout << "\nFailo turinys:\n";
while (!sfailas.eof())
{
 sfailas.getline(tekstas,255);
 cout << tekstas<<endl;
}
cout << "\n*****Failo pabaiga.*****\n";
sfailas.close();
return 0;
}
```

Visi iki šiol pateikti darbo su išoriniais failais pavyzdžiai buvo atliekami su simbolinio tipo duomenimis, kitaip vadinamais tekstiniais failais. Juose informacija išsaugoma teksto pavidalu. Egzistuoja ir kitokio tipo failai, kuriuose informacija saugoma tokia, kokia ji ir buvo pateikta. Tai – dvejetainiai (binariniai) failai. Tokiuose failuose galima saugoti ne tik skaičius ar eilutes, bet ir visas duomenų struktūras. Duomenys į dvejetainį failą gali būti įrašomi ar iš jo nuskaitomi vienu ypu. Tam naudojamos *write()* bei *read()* funkcijos atitinkamai. Abiems joms reikia dviejų parametrų, vienas kurių yra rodyklė į simbolį, o kitas yra rašomų ar skaitomų

simbolių skaičius. Norint perduoti šioms funkcijoms objektą, reikia jo adresą paveikti tipo keitimo operatoriumi, kurio struktūra yra: *(naujas_tipas) objektas*. Tokiu būdu galima pvz. įrašyti visas kokios nors klasės objekto duomenų vertes. Reikia pažymėti, kad išsaugomi tik duomenys (ne funkcijos). Pavyzdys:

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

class Studentas
{
     char Vardas[20];
     int BilNr;
     bool Dieninis;
public:
      void Duomenys(char v[],int nr,bool d);
      void Informacija();
};

void Studentas::Duomenys(char v[],int nr,bool d)
{
    strcpy(Vardas,v);
    BilNr = nr;
    Dieninis = d;
}

void Studentas::Informacija()
{
    cout<<"Studento vardas: "<<Vardas<<"\nPazymejimo Nr. "<<BilNr;
    if (Dieninis) cout<<"\nDieninio skyriaus\n";
    else cout<<"\nVakarinio skyriaus\n";
}

int main ()
{
Studentas stud;
stud.Duomenys("Jonas",1234,true);
ofstream fout("Klase.dat");
fout.write( (char*) &stud, sizeof(stud));
fout.close();
stud.Duomenys("Petras",5678,false);
cout<<"Spausdinam\n";
stud.Informacija();
ifstream fin("Klase.dat");
fin.read( (char*) &stud, sizeof(stud));
fin.close();
cout<<"\nSpausdinam is failo\n";
stud.Informacija();
system("PAUSE");
return 0;
}
```

Šioje programoje sukuriamas klasės *Studentas* objektas *stud*, kuriam priskiriami duomenys išsaugomi dvejetainiame faile *„Klase.dat“*. Kreipinyje į *write()* funkciją perduodamas *stud* objekto adresas, pakeičiant jo tipą į *char* rodyklę, bei įrašomų

duomenų kiekis, gaunamas *sizeof()* funkcija. Kreipimosi į *read()* funkciją, kuri iš failo įkelia šiuos duomenis atgal į programą, sintaksė yra analogiška.

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

Duomenims įvesti/išvesti dar naudojamos iš C paveldėtos funkcijos *printf()* bei *scanf()*. Jas galima naudoti įterpus į program biblioteką <stdio.h>. Kreipinių į *printf()* ir *scanf()* pavidalas yra toks:

printf (šablonas , kintamųjų_sąrašas);

scanf (šablonas, kintamųjų_adresų_sąrašas);

Pastebėkime, kad duomenų išvedimo funkcijoje *printf()* po šablono nurodomi kintamųjų vardai, o ivedimo funkcijoje *scanf()* – jų adresai. Šablonas – tai simbolių eilutės konstanta su joje įterptais duomenų formatais ir valdančiaisiais simboliais.

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

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

Čia stačiakampiai skliaustai reiškia, kad to formato elemento gali ir nebuti. 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 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	simboliu eilutė;
e	rodiklinės formos skaičius (float);
p	rodyklė (pointer)

Išvedimo funkcijoje *printf()* valdantieji simboliai gali būti įterpiami bet kurioje šablono vietoje. *printf()* naudojimo pavyzdys:

```
int a = 1023;
double g = 3.1415926535898;
char e = ‘X’;
printf("dešimtainis sveikasis  a=%d\n", a); // rezultatas a=1023
printf("trupmena g=%11.8f\n", g); // rezultatas g= 3.141592654
printf("simbolis e=%c\n", e); // rezultatas e=X
```

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 (tab) arba perėjimu į kitą eilutę (enter). Pvz., ivedus “45 8.56”, kintamasis *a* igis vertę, lygią 45, o kintamasis b – 8.56.

Jei funkcijos *scanf()* šablone prieš duomenų formatus bus nurodyti kokie nors simboliai, tai juos taip pat reikės ivesti. Pvz.:

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

Čia klaviatūra įvedami duomenys turės atrodyti taip: “a= 45 f= 856”.

Priedai

1 priedas: tarnybiniai žodžiai

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,
Žodžiai, kurių taip pat negalima vartoti kintamųjų vardams:
and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, xor_eq

2 priedas: operatorių pirmumo lygiai

Pirmumo lygis	Operatorius	Paaiškinimai	Atlikimo išraiškoje kryptis
1	::	Veikimo srities (scope)	Iš kairės i dešinę
2	() [] . –> ++ -- dynamic_cast static_cast reinterpret_cast const_cast typeid	Rašomi po vardo (postfix)	Iš kairės i dešinę
3	++ -- ~ ! sizeof new delete * & + -	Vieno operando (prefix) Netiesioginiai ir nuorodos (pointers) Operando ženklo	Iš dešinės i kairę
4	(type)	Aiškus tipo keitimo	Iš dešinės i kairę
5	.* ->*	Rodyklės į narį	Iš kairės i dešinę
6	* / %	Dauginamieji	Iš kairės i dešinę
7	+ -	Pridedantieji	Iš kairės i dešinę
8	<< >>	Postūmio	Iš kairės i dešinę
9	< > <= >=	Palyginimo	Iš kairės i dešinę
10	= = !=	Palyginimo	Iš kairės i dešinę
11	&	Loginis IR su bitais	Iš kairės i dešinę
12	^	Loginis griežtasis-ARBA su bitais	Iš kairės i dešinę
13	|	Loginis ARBA su bitais	Iš kairės i dešinę
14	&&	Loginis IR	Iš kairės i dešinę
15	||	Loginis ARBA	Iš kairės i dešinę
16	?:	Salygos	Iš kairės i dešinę
17	= *= /= %= += -= >>= <<= &= ^= !=	Priskyrimo	Iš dešinės i kairę
18	,	Kablelio	Iš kairės i dešinę

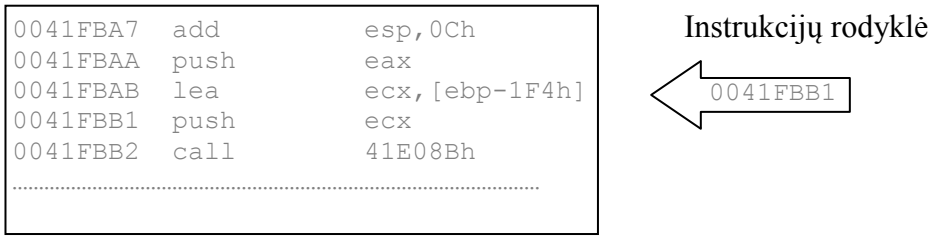
3 priedas: funkcijos išsamiau

Iškviečiant funkciją, atliekamas programos kodas pertraukiamas ir ji peršoka į funkciją, kuriai perduodami reikiami parametrai ir atliekami funkcijos realizacijos dalyje esantys sakiniai. Kaip tai yra atliekama? Pirmiausia reikia suvokti, kad aukšto lygio (kokia yra ir C++) kalbose naudojamos daugiapakopės abstrakčios struktūros, ką daugeliui pradedančiųjų programuotojų sunku suvokti. Kompiuteriai iš tiesų yra tik elektroninės mašinos. Jie neturi jokio supratimo apie langus ir meniu, programas ar instrukcijas ir netgi 0 ir 1. Viskas, ką jie gali, tai nustatyti įtampos lygį tam tikruose integrinio grandyno

taškuose. Netgi ir tai yra abstrakcija – elektra yra tik intelektuali koncepcija elektronų judėjimo medžiagose ypatumams pavaizduoti. Paprastai programuotojui nebūtina žinoti subatominių dalelių fizikos tam, kad galėtų sėkmingai programuoti. Jam visai pakanka suprasti, kokia yra laisvo išrinkimo atminties (RAM) struktūra kompiuteryje ir kaip ji funkcionuoja.

Pradedant veikti programai, operacinė sistema paskiria atskiras laisvo išrinkimo atminties dalis pagal kompiliatoriaus reikalavimus. C++ programuotojui svarbiausios atminties dalys yra: globalių kintamųjų, dėklo, programinio kodo, dinamiškai skiriama („heap“) bei registrų. Registrai yra speciali atminties dalis, esanti tiesiog centriniame procesoriuje. Daugelis jų kalbant apie funkcijas nėra svarbūs, tačiau svarbūs tie registrai, kurie yra skirti nurodyti, kokia programinio kodo eilutė bus vykdoma sekanti. Jie vadinami instrukcijų rodykle. Programinis kodas yra laikomas atmintyje atskirai nuo globalių kintamųjų, dėklo ir t.t. programinio kodo dalyje. Kiekvienas kodo sakinys yra paverčiamas instrukcijų procesoriui seka ir kiekviena ši instrukcija yra patalpinta tam tikru adresu kompiuterio atmintyje, pvz.,:

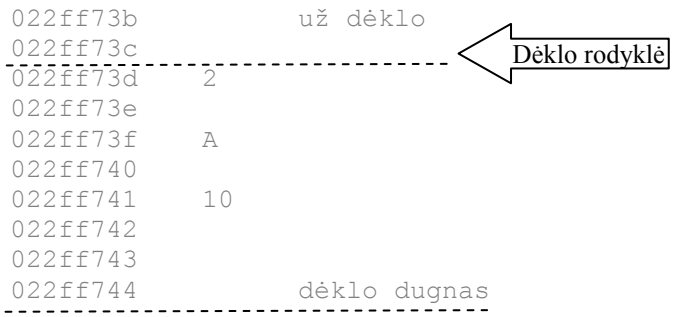
Programinis kodas



Dėklas yra speciali vieta atmintyje, sauganti duomenis, kurių reikia programos funkcijoms vykdyti. Jį galime įsivaizduoti kaip kokį indą – tai, kas į jį įdėta pirmiausia, bus galima paimti paskiausiai ir atvirkščiai. „Stumiant“ (nuo angl. termino

„push“) duomenis į dėklą, jis didėja, o „traukiant“ iš jo – mažėja. Norint ištraukti tam tikrą duomenį iš dėklo, pirmiausia reikia ištraukti visus aukščiau jo esančius. Dėklo pavyzdys:

Dėklas



Dėklo viršūnės adresą saugo dėklo rodyklė. Visi adresai, didesni už saugomą joje priklauso jam. Kai nauji duomenys talpinami į dėklą, jie talpinami virš dėklo rodyklės, o ji pati perstumiama į naują vietą. Imant duomenis iš dėklo jo rodyklė tiesiog perstumiama žemyn.

Programai kreipiantis į funkciją atliekami tokie veiksmai:

1. Programos instrukcijų rodyklės adresas yra padidinamas iki sekančios, einančios po funkcijos iškvietimo ir jos adresas yra talpinamas dėkle. Tai reikalinga, norint žinoti, kur programa turės grįžti funkcijai baigus darbą.
2. Dėkle rezervuojama vieta funkcijos grąžinamai vertei. Rezervuotos vietos dydis priklauso nuo grąžinamos vertės tipo, pvz. jei funkcijos grąžinama vertė yra *float* tipo jis bus 4 baitai.
3. Iškviečiamos funkcijos adresas yra įrašomas į instrukcijų rodyklę, taip pasiekiant, kad sekanti vykdoma instrukcija būtų pirmoji funkcijos instrukcija.
4. Esamas dėklo viršus yra pažymimas ir laikomas specialioje rodyklėje, vadinamoje dėklo kadru. Visi kintamieji, dedami į jį nuo šio momento iki tol, kol funkcija baigs darbą, vadinami lokaliais tai funkcijai.
5. Perduodami funkcijai argumentai patalpinami į dėklą.
6. Pradedamos vykdyti funkcijos instrukcijos.
7. Lokalūs kintamieji talpinami į dėklą ta tvarka, kuria jie sutinkami funkcijos programiniame kode.

Funkcijai baigus darbą, jos grąžinama vertė patalpinama 2 punkte rezervuotoje dėklo atminties dalyje ir pastarasis sumažinamas iki dėklo kadro rodyklės rodomo adreso.

Tuo būdu panaikinami visi lokalūs kintamieji. Iš dėklo paimama grąžinama išraiška ir priskiriama funkcijos iškvietimo išraiškai, taipogi paimamas ir 1 punkte paimtas adresas, nusakantis į kurią iškvietusios funkcijos vietą reikia grįžti. Jis patalpinamas instrukcijų skaitiklyje ir funkcija baigia darbą.

Rekomenduojama literatūra

1. Programavimas C++ : vadovėlis / J. Blonskis .V. Bukšnaitis, V. Jusas, R. Marcinkevičius, D. Rubliauskas; Technologija 2008.
2. C, C++, OOP : mokomės programuoti / A. Matulis; Puslaidininkių fizikos institutas 2005.
3. C++ praktikumas : mokomoji knyga / J. Blonskis, V. Bukšnaitis, J. Končienė, D. Rubliauskas 2001.
4. C++ duomenų tipai ir struktūros / A. Vidžiūnas 1999.
5. C++ ir C++ Builder pradmenys. A. Vidžiūnas. Smaltijos leidykla, 2002.
6. Programavimo kalbos ir skaičiavimų receptai. Ž. Kancleris; Vilniaus universitetas. I d. 2004.
7. C++ and object oriented numeric computing for scientists and engineers / Daoqi Yang; Springer –Verlag New York, 2001.
8. A first course in computational physic and object-oriented programming with C++ / David Jevick; Cambridge University press, 2005.
9. J. Soulie. C++ Language Tutorial. 2006. (http://www.cplusplus.com/doc/tutorial/).
10. http://www.cppreference.com/wiki/ .