11. RODYKLĖS

Rodyklė (angl. pointer) – kintamasis, saugantis kito konkretaus tipo kintamojo adresą. Rodyklės taikomos:

- Netiesioginei prieičiai prie kintamojo
- Prieičiai prie masyvo elementų
- Argumentų perdavimui į funkcijas
- Funkcijų kvietimui rodykle
- Dinaminės atminties skyrimui
- Duomenų struktūrų (pavyzdžiui, saitinio sąrašo) kūrimui

Daugeliu atvejų šiuos dalykus galima atlikti ir be rodyklių, bet rodyklėmis – efektyviau. C++ programose tradiciškai rodyklės taikomos labai plačiai. Jos būtinos dinaminei atminčiai skirti, sąrašams kurti ir kai kurioms virtualių funkcijų galimybėms realizuoti. Vis tik rodyklių konstrukcijos reikalauja didelio atidumo; jos lemia daugybę sunkiai aptinkamų programavimo klaidų. Matyt, todėl vėlesnė C++ pagrindu sukurta kalba Java rodyklių atsisakė (kompiliatorius jas, aišku, taiko, bet programuotojas disponuoti tokia konstrukcija negali).

10.1 ĮVADINĖS ŽINIOS APIE RODYKLES

Visi programos kintamieji kompiuterio atmintyje saugomi skirtingo ilgio ląstelėse. Ląsteles sudarantys baitai turi unikalius adresus. Kuriuose baituose kintamasis bus saugomas, priklauso nuo sisteminės atminties apimties, pačios programos apimties, programos duomenų apimties, nuo to, kelios programos šiuo metu yra vykdomos ir pan. – kiekvienąkart paleidus programą, kintamojo adresas bus skirtingas.

Kintamojo *k* adresą galima sužinoti adreso operacija *&k* (nemaišykite jos su konstrukcija *duomens formatas &k* arba *duomens formatas& k* – tai yra nuorodos (angl. reference) operatorius). Adresai išvedami šešioliktainėje skaičiavimo sistemoje. Šiems adresams saugoti skirti kintamieji ir vadinami rodyklėmis. Skelbiant rodyklę būtina nurodyti, kokio formato duomens adresas bus joje talpinamas. Tokios informacijos reikia, pavyzdžiui, kreipiantis į masyvo elementus rodykle: perslenkant rodyklę nuo vieno elemento prie kito, kompiliatorius turi žinoti, per kiek baitų turi pakisti adresas.

Rodyklės skelbimo alternatyvos:

```
int* pint1;
int *pint2;
```

Kelios rodyklės skelbiamos:

```
double *pdouble1, *pdouble2, *pdouble3, double4;
```

Taigi, prieš kiekvieną rodyklę būtinas simbolis ‚*‘. Šis pavyzdys kartu rodo, kad viename operatoriuje galima skelbti ir rodykles, ir paprastus kintamuosius: kadangi žvaigždutės prieš *double4* nėra, jis laikomas *double* formato kintamuoju.

Kaip į rodyklės tipo duomenį galima įkelti kintamojo adresą (sakysime: nukreipti rodyklę į kintamąjį), parodo 1 pavyzdys:

```
#include <iostream>
using namespace std;
//
int main( ) {
        //
        int k1 = 10,
            k2 = 20,
           k3 = 30;
        cout<<"Kintamuju adresai:\n"<<&k1<<endl<<&k2<<endl<<&k3<<endl;
        //
        int* p;   // rodyklė
        p = &k1;  // rodyklė nukreipiama į k1
        cout<<"Rodykles reiksme:  "<<p<<endl;
        p = &k3;  // ta pati rodykle dabar nukreipiama į k3
        cout<<"Rodykles reiksme:  "<<p<<endl;
        p+= 1;  // rodykle perstumiama į tolesnę 4B ilgio ląstelę
        cout<<"Perstumtos rodykles reiksme:  "<<p<<endl;
        //
        system("pause");
        return 0;
}
```

Programos spausdiniai:

```
Kintamuju adresai:
0048FB2C
0048FB20
0048FB14
Rodykles reiksme:  0048FB2C
Rodykles reiksme:  0048FB14
Perstumtos rodykles reiksme:  0048FB18
```

Paleidę šią programą savo kompiuteryje, jūs, aišku matysit visai kitus adresus. Kitą kartą paleidus programą, tie adresai vėl bus kitokie. Paskutinis programos veiksmas su rodykle *p+= 1;* parodo, kodėl skelbiant rodyklę būtina nurodyti ląstelės formatą: šis veiksmas rodyklę perstumia per vieną ląstelę, o iš spausdinių matome, kad adresas pakito 4 vienetais. Taip yra dėl to, kad *int* formato duomeniui skiriama 4B ilgio ląstelė.

Neinicializavus rodyklės, ji rodo atsitiktinį adresą. Galima rodyklę inicializuoti nuline reikšme:

```
double *pdouble = NULL;
```

Dabar aišku, kaip patikrinti, ar rodyklė yra į kurį nors kintamąjį nukreipta:

if(pdouble == NULL) . . . // true reiškia - nenukreipta arba
if(pdouble != 0) . . . // true reiškia – nukreipta

Dar kartą grįžkime prie adreso operacijos ir nuorodos bei jų skirtumų nuo rodyklių konstrukcijų. Nuoroda skelbiama analogiškai rodyklei, po kintamojo formato nurodžius ženklą *&*. Gi tas pat ženklas prieš jau paskelbtą kintamąjį reiškia to kintamojo adresą. Nuorodos esminis skirtumas nuo rodyklės: nuoroda yra tik kito kintamojo alternatyvus vardas. Tą matėme perduodami argumentus adresu į funkcijas: jau tada minėjome, kad faktiškieji argumentai-kintamieji ir atitinkami formalūs argumentai-nuorodos yra tik tų pačių ląstelių alternatyvūs vardai. Panašią konstrukciją galima taikyti ir vienoje funkcijoje:

```
. . .
double d = 5.;
double& rd = d;  // rd ir d – tos pat ląstelės alternatyvūs vardai
rd+= 10.;          // tos pat ląstelės turinys didinamas 10.
cout<<d<<“  “<<rd;     // bus išspausdinta 10  10
. . .
```

10.2 NETIESIOGINĖ KREIPTIS Į KINTAMĄJĮ

Žinant kintamojo adresą, galima gauti ir jo reikšmę, arba jam priskirti norimą reikšmę. Tam taikoma konstrukcija **rodyklė* (ją vadinsime įreikšminimo operacija, angl. indirection operator).

2 pavyzdys:

```
#include <iostream>
using namespace std;
//
int main( ) {
        //
        int k1 = 10,
                    k2 = 20;
        int *p = &k1; // rodyklę nukreipiam į k1
        cout<<"Adresas ir reiksme  "<<p<<"  "<<*p<<endl;
        //
        *p = 30; // atitinka k1 = 30
        k2 = *p; // atitinka k2 = 30
        cout<<"Kintamuju k1 ir k2 reiksmes  "<<*p<<"  "<<k2<<endl;
        //
        system("pause");
        return 0;
}
```

Šioje programoje rodyklė *p* nutaikyta į kintamąjį *k1*, o konstrukcija **p = 30;* yra netiesioginė kreiptis į kintamojo ląstelę per rodyklės įreikšminimo operaciją. Aišku, programa spausdins vienodas abiejų kintamųjų reikšmes: 30.

10.3 RODYKLĖS Į MASYVUS. OPERATORIUS *sizeof*

Visi prieigos prie masyvo elementų rodykle būdai parodyti 3 pavyzdyje:

```
#include <iostream>
using namespace std;
//
int main( ) {
        //
        int array[ ] = { 10,20,30,40,50 };
        int i, n = (sizeof array) / (sizeof array[0]);
        int *p = array; // rodyklę nukreipiam į masyvo pirmojo elemento
                        // pirmojo baito adresą
        //
        // 1 būdas
        for( i = 0; i < n; i++ )
                cout<<array[i]<<"  ";
        cout<<endl;
        //
        // 2 būdas
        for( i = 0; i < n; i++ )
                cout<<p[i]<<"  ";
        cout<<endl;
        //
        // 3 būdas
        for( i = 0; i < n; i++ )
                cout<<*( p+i )<<"  ";
        cout<<endl;
        //
        // 4 būdas
        for( i = 0; i < n; i++ )
                cout<<*( array+i )<<"  ";
        cout<<endl;
        //
        // 5 būdas
        for( i = 0; i < n; i++ )
                cout<<*p++<<"  ";
        cout<<endl;
        //
        // klaidingas būdas
        //for( i = 0; i < n; i++ )
        //              cout<<*array++<<"  ";
        //cout<<endl;
        //
        system("pause");
        return 0;
}
```

Visais 5 atvejais programa spausdina masyvo *array* visus elementus. Bet kokio masyvo elementų kiekį galima sužinoti taip, kaip parodyta: *sizeof* operatoriumi. Šis operatorius, jei jo argumentu nurodyti masyvą, grąžins masyvui skirtą baitų kiekį, o jei argumentas yra vienas kuris masyvo elementas – to elemento ilgį. Taigi, dalybos rezultatas yra masyvo elementų kiekis.

Pirmasis būdas kreiptis į masyvo elementą yra mums jau gerai pažįstama adreso operacija *masyvoVardas[indeksas].*

Antrasis būdas rodo, kad adreso operaciją galima taikyti ir rodyklei, nukreiptai į masyvą. Prieskyra *int *p= array;* rodyklę nukreipia į atminties srities, skirtos masyvui saugoti, pirmojo baito adresą. Tuo būdu, iš programos teksto net sunku atskirti, kur masyvas, kur rodyklė (palyginkit: *array[i]* ir *p[i]*).

Trečiasis būdas: prie rodyklės pradinės reikšmės – masyvo atminties srities pirmojo baito adreso – kiekviename ciklo žingsnyje pridedama po *1*, o *1 int* formato rodyklei reiškia 4B. Elemento reikšmė gaunama teikiant šiam suskaičiuotam adresui įreikšminimo operaciją.

Ketvirtasis būdas: panašus metodas – juk *array* yra masyvui skirtos atminties pirmojo baito adresas.

Penktasis būdas: rodyklei galima taikyti aritmetines sudėties ir atimties operacijas, tarp jų ir čia naudotą inkremento (postinkremento forma) operaciją. Čia tinka tik postinkremento operacija: pirmiau turi būti atlikta įreikšminimo operacija rodyklei, o tik po to reikia rodyklės reikšmę paauginti. Kai masyvui skirsime dinaminę atmintį, toks vaikščiojimo po masyvą būdas netiks, nes rodyklė pasibaigus ciklui rodys už masyvo atminties srities ribų – negalėsime tinkamai išlaisvinti paskirtos dinaminės atminties.

Pagaliau yra parodytas ir klaidingas kreipinio į masyvo elementus būdas: *array* yra adresas, t.y. konstanta, o jai taikyti bet kokių aritmetinių operacijų nevalia.

Sugrįžkime prie programos operatoriaus *int *p = array;* .Taip rodyklę galima nukreipti tik į vienmačio masyvo atminties srities pirmojo baito adresą. Sakykim, norim rodyklę nukreipti į masyvo *array* tretįjį elementą. Atrodytų, turi veikti operatoriai

```
. . .
int array[ 5 ];
int *p;
. . .
p = array[ 3 ];
. . .
```

Vis tik taip rodyklės nukreipti į konkretų masyvo elementą nevalia, nes čia painiojami du skirtingi prieskyros operandų tipai: kairėje prieskyros pusėje yra *int* formato rodyklė, o dešinėje – adreso operacijos *[ ]* teikiamas aritmetinis *int* duomuo. Viskas veiks, jei dešinėje prieskyros pusėje parašysime elemento adresą: *p = &array[3];* .

Sudėtingesni reikalai su daugiamačiais masyvais, kurie C++ kalboje yra vienmatis masyvas su elementais – savo ruožtu kitais vienmačiais masyvais, ir t.t. Sakykim, į dvimačio sveikojo masyvo atminties srities pradžią norime nukreipti rodykles *p1, p2, p3*. Galimi rodyklės nukreipimo būdai:

```
. . .
int array[ 2 ] [ 3 ];
```

```
int *p1, *p2;
int (*p3)[ 3 ];
. . .
// p1 = array;   // klaida!
p1 = &array[ 0 ] [ 0 ];  // 1
p2 = array[ 0 ];         // 2
p3 = array;              // 3
. . .
```

Intuityviai parašytas operatorius *p1 = array;* klaidingas: rodyklės formatas *yra int**, o vienmačio masyvo su elementais-masyvais *array* adresas yra formato *int*[3]*: juk norint surasti šio masyvo kito elemento adresą, reikia 4 (*int* duomens ilgis baitais) dauginti iš elemento-masyvo elementų skaičiaus 3. Rodyklė *p3* paskelbta kaip tik tokio formato, todėl trečiasis būdas – teisingas. Pirmasis būdas akivaizdus: čia rodyklę *p1* nukreipiame į elemento adresą. Antrasis būdas irgi tampa aiškus, prisiminus dvimačio masyvo struktūrą: *array[0]* yra vienmačio masyvo vardas, t.y. adresas.

10.4 ARGUMENTŲ PERDAVIMAS Į FUNKCIJĄ RODYKLĖMIS

Prisimename, kad argumentus į funkcijas galima perduoti reikšmės ir adreso mechanizmais. Daugelyje knygų teigiama, kad yra ir trečiasis būdas – rodykle. Faktiškai yra ne visai taip, nes į funkcijas galima perduoti rodykles į kintamuosius ar masyvus, tačiau pati rodyklė perduodama reikšmės mechanizmu. Todėl funkcijoje galima drąsiai keisti formaliojo argumento-rodyklės reikšmę – šie pokyčiai kviečiančiojoje programoje nebus matomi.

4 pavyzdys. Prisiminkime abu argumentų perdavimo būdus paprasčiausiam uždavinukui: vienos Farenheito temperatūros, kurios reikšmė vardan programos trumpumo suteikiama tiesiog tekste, perskaičiavimui į Celsijaus skalę.

Perdavimas reikšme:

```
#include <iostream>
using namespace std;
//
int main( ) {
        //
        double convert( double );
        //
        double temp = 32.;
        cout<<"Farenheito laipsniai: "<<temp<<endl
                <<"Celsijaus laipsniai:  "<<convert( temp )<<endl;
        //
        system("pause");
        return 0;
}
//
double convert( double t ) {
```

```
        return ( (t-32.)*5./9. );
}
```

Perdavimas adresu:

```
#include <iostream>
using namespace std;
//
int main( ) {
        //
        void convert( double& );
        //
        double temp = 32.;
        cout<<"Farenheito laipsniai: "<<temp<<endl;
        convert( temp );
        cout<<"Celsijaus laipsniai:  "<<temp<<endl;
        //
        system("pause");
        return 0;
}
//
void convert( double& t ) {
        t = (t-32.)*5./9.;
}
```

Rodyklės perdavimas reikšme:

```
#include <iostream>
using namespace std;
//
int main( ) {
        //
        void convert( double* );
        //
        double temp = 32.;
        cout<<"Farenheito laipsniai: "<<temp<<endl;
        convert( &temp );
        cout<<"Celsijaus laipsniai:  "<<temp<<endl;
        //
        system("pause");
        return 0;
}
//
void convert( double* pd ) {
        *pd = (*pd-32.)*5./9.;
}
```

Kaip matote, perdavimas rodykle sintaksiškai panašus į perdavimą adresu, tik kviečiant funkciją vietoje formalaus argumento-rodyklės, aišku, reikia teikti adresą – tas čia ir padaryta adreso operacija *&temp.* Pačioje funkcijoje prie kintamojo reikšmės galima prieiti tik įreikšminimo operacija **pd.* Jos pagalba pakeičiama ląstelės, kurios adresas yra *&temp*, reikšmė, todėl antrasis funkcijos *main* spausdinimo operatorius išspausdins jau pakitusią ląstelės *temp* reikšmę.

5 pavyzdys. Argumentų-masyvų perdavimas rodykle. Tegu uždavinio sąlyga lieka ta pat, tik turime pervesti į kitą skalę kelias temperatūros reikšmes.

```
#include <iostream>
#include <iomanip>
using namespace std;
//
int main( ) {
        //
        void convert( double*, int );
        //
        double temp[ ] = { 32., 70., 0., 45., 15. };
        double *pd = temp;
        //
        int n = (sizeof temp)/(sizeof temp[0]);
        //
        cout<<"Farenheito laipsniai: ";
        for( int i = 0; i < n; i++ )
                cout<<setw( 7 )<<setprecision( 3 )<<temp[ i ]<<"  ";
        cout<<endl;
        //
        convert( pd, n );
        //
        cout<<"Celsijaus laipsniai:  ";
        for( int i = 0; i < n; i++ )
                cout<<setw( 7 )<<setprecision( 3 )<<temp[ i ]<<"  ";
        cout<<endl;
        //
        system("pause");
        return 0;
}
//
void convert( double* p, int n ) {
        for( int i = 0; i < n; i++ )
                p[ i ] = (p[ i ]-32.)*5./9.;    // 1 būdas
                //*(p+i) = (*(p+i)-32.)*5./9.;  // 2 būdas
                //*p++ = (*p-32.)*5./9.;        // 3 būdas
}
```

Visi trys būdai „vaikščioti“ po masyvo elementus teikia visiškai vienodus programos rezultatus. Logiškai sudėtingesnis yra trečiasis būdas: reikia prisiminti, kad postinkrementas taikomas tik tada, kai visa šios operacijos aplinka jau įvykdyta. Tuo būdu, pirmiau atliekamas veiksmas **p = (*p-32.)*5./9.;* ir tik tada paauginama rodyklės reikšmė. Pasibaigus ciklui dabar rodyklė bus nukreipta už masyvui skirtos atminties srities ribų, tačiau kviečiančioji programa apie tokį rodyklės reikšmės pokytį nieko nežinos, nes pati rodyklė į funkciją atiduodama reikšmės mechanizmu. Galite tuo įsitikinti spausdindami rodyklės *pd* reikšmę prieš funkcijos *convert* kvietimą ir iškart po kvietimo bei pačioje funkcijoje palikę neužkomentuotą tik trečiąjį masyvo elementų išrinkimo būdą.

6 pavyzdys. Dar kartą parašysime programą masyvui rūšiuoti didėjimo tvarka burbulo metodu: imame pirmąjį masyvo elementą ir jį lyginam paeiliui su visais likusiais elementais. Jei tolesnis masyvo elementas didesnis už pirmesnį – nieko nedarom, jei nei – elementus sukeičiam vietomis. Po šių veiksmų mažiausias masyvo elementas atkeliaus į pirmojo elemento vietą, tačiau likusi masyvo dalis dar nebus tinkamai išrikiuota, todėl turime tokius veiksmus kartoti dabar jau antrąjį pertvarkyto masyvo elementą lygindami su likusiais elementais, ir t.t. iki *n-1*-ojo elemento lyginimo su paskutiniu. Panašų algoritmą esame užrašę 6-ame skyriuje, tik atvirkščiai: ten į masyvo galą plukdėme didžiausią elementą.

Masyvą į rikiavimo funkciją atiduosime rodykle, o du masyvo elementus lyginsime ir, jei reiks, keisime vietomis dar kitoje funkcijoje. Šiai funkcijai masyvo elementai perduodami taip pat rodyklėmis. Kad programa būtų trumpesnė, masyvą suformuosime tiesiog programos tekste.

```
#include <iostream>
#include <iomanip>
using namespace std;
//
int main( ) {
        // Burbulo algoritmo funkcijos prototipas
        void bubbleSort( int, double* );
        //
        double array[ ] = { 32., 70., 0., 45., 15. };
        //
        int n = (sizeof array)/(sizeof array[0]);
        //
        cout<<"Pradinis masyvas:\n";
        for( int i = 0; i < n; i++ )
                cout<<setw( 7 )<<setprecision( 3 )<<setiosflags( ios::showpoint )
                  <<array[ i ]<<"  ";
        cout<<endl;
        //
        bubbleSort( n, array );
        //
        cout<<"Isrikiuotas masyvas:\n ";
        for( int i = 0; i < n; i++ )
                cout<<setw( 7 )<<setprecision( 3 )<<setiosflags( ios::showpoint )
                  <<array[ i ]<<"  ";
        cout<<endl;
```

```
        //
        system("pause");
        return 0;
}
//
void bubbleSort( int n, double* p ) {
        // 2 skaičių rikiavimo funkcijos prototipas
        void sort2numbers( double*, double* );
        //
        for( int i = 0; i < n-1; i++ )
                for( int j = i+1; j < n; j++ )
                        sort2numbers( p+i, p+j );
}
//
void sort2numbers( double* n1, double* n2 ) { // elementai atiduodami rodykle
        if( *n1 > *n2 ) {
                double temp = *n1;
                *n1 = *n2;
                *n2 = temp;
        }
}
```

10.5 DINAMINĖS ATMINTIES SKYRIMAS

Visose programose iki šiol kintamiesiems ir masyvams skyrėme statinę atmintį. Jį paskiriama kompiliavimo metu ir lieka paskirta visam programos (ar matomumo srities, kur kintamieji ar masyvai paskelbti) gyvavimo laikui. Tai gali būti labai neefektyvu, kai iš anksto nežinome, kiek programos vykdymo metu iš tikrųjų reiks atminties, pavyzdžiui, masyvui saugoti. Galimas problemos sprendimo būdas – paskirti masyvui atminties tiek, kad bet kokiu atveju jos pakaktų. Todėl dažnai uždavinių sąlygas mes panašiai ir formulavome: „...masyvui, turinčiam iki 100 elementų...“. Vis tik panašiais atvejais žymiai racionaliau masyvui skirti dinaminę atmintį programos vykdymo metu, kai jau bus žinoma, kiek konkrečiai atminties prireiks. Kai masyvas nebereikalingas, jam skirtą atmintį iškart galima išlaisvinti ir naudoti kitiems programos tikslams.

Dinaminė atmintis skiriama operatoriumi *new*, kuris grąžina atminties srities kintamajam (ar masyvui) pirmojo baito adresą, todėl tą adresą galima priskirti tik rodyklei. Kai atmintis nebereikalinga, ji išlaisvinama operatoriumi *delete*. Visas dinaminės atminties gyvavimo ciklas paprastam *double* kintamajam būtų:

```
. . .
double* pd = NULL;  // skelbiama rodyklė. Inicializuojama nuline reikšme
pd = new double;    // skiriama dinaminė atmintis – 8B
. . .
*pd = 123.4;          // į atmintį įkeliama reikiama reikšmė
. . .
delete pd;          // kai kintamojo nebereikia – išlaisvinama atmintis
```

. . .

Aišku, programuotojui žymiai svarbesnis yra dinaminės atminties skyrimas vienmačiam masyvui. Masyvui skirdami atmintį turėsim nurodyti, kiek elementų turi masyvas. Jei skirdami masyvui statinę atmintį, turėjome nurodyti konkretų ląstelių skaičių sveikosios konstantos pavidalu, tai dabar tą galima padaryti ir sveikojo formato kintamuoju. Išlaisvindami atmintį, taip pat turime kompiliatoriui parodyti, kad išlaisvinama masyvo atmintis; tam pakanka po operatoriaus *delete* parašyti skliaustelius *[ ]*.

Paprasčiausias programos su dinamine masyvo atmintim 7 pavyzdys: sudaromas sveikųjų skaičių pradedant nuo 1 kubų masyvas. Kiek skaičių kubų skaičiuoti – nurodo programos vartotojas programos vykdymo metu.

```
#include <iostream>
using namespace std;
//
int main( ) {
        //
        int* array;  // rodyklė, kurią nukreipsime į masyvo pradžią
        int size;     // masyvo ilgis - kintamasis
        //
        cout<<"Nurodykite, kiek kubu skaiciuosime\n";
        cin>>size;
        cout<<"Skaiciuosime "<<size<<" kubu\n";
        //
        array = new int[ size ];   // skiriama dinaminė atmintis
        //
        for( int i = 0; i < size; i++ ) {
                array[ i ] = (i+1)*(i+1)*(i+1);      // 1 adresavimo būdas
                //*(array + i) = (i+1)*(i+1)*(i+1);   // 2 būdas
                //*array++ = (i+1)*(i+1)*(i+1);      // 3 būdas – klaida!
        }
        //
        for( int i = 1; i <= size; i++ )
                cout<<"Skaiciaus "<<i<<" kubas yra "<<array[ i-1 ]<<endl;
        //
        delete [ ] array;  // atmintis išlaisvinama
        //
        system("pause");
        return 0;
}
```

Atkreipkite dėmesį, kad kai numatome programoje dinaminę masyvo atmintį išlaisvinti, netinka trečiasis šiaip jau sintaksiškai visiškai tvarkingas „vaikščiojimo“ po masyvo elementus būdas. Inkremento operacija rodyklei ją perstumia į kitus nei masyvo atminties bloko pirmojo baito adresus, o *delete* operatorius išlaisvina *size* ląstelių atminties nuo ląstelės, į kurią rodyklė nukreipta. Taigi, jei rodyklė nukreipta tolyn nuo atminties srities pirmojo baito adreso, bus išlaisvinta ne masyvui saugoti skirta atmintis – įvyks neprognozuojamo pobūdžio programos

vykdymo meto klaida. Gi pirmais dviem būdais kreipiantis į masyvą, rodyklės reikšmė nekeičiama.

Dinaminės atminties skyrimas daugiamačiams masyvams. Jei masyvų matmenys yra konstantiniai, arba kintamasis yra tik pats kairysis masyvo matas, atminties skyrimo sintaksė labai panaši į vienmačiam masyvui taikomą:

```
. . .
double* dm = new double[ 10 ] [ 100 ];  // skiriami 8000B
. . .
int n;
. . .
cin>>n;
double* tm = new double[ n ] [ 10 ] [ 10 ]; // kintamasis gali būti tik kairysis matas
. . .
delete [ ] dm;
delete [ ] tm;
. . .
```

Jei dvimačio ar dar didesnio matų skaičiaus masyvų bent keli matai yra žinomi tik kintamųjų pavidalu, dinaminės atminties skyrimo sintaksė tampa žymiai sudėtingesnė. Panagrinėkime dvimačio masyvo atvejį. Tegu dvimatis *double* formato masyvas *dm* turi *n* eilučių ir *m* stulpelių. Faktiškai masyvo struktūra yra tokia: tai yra vienmatis *n* elementų masyvas, kurio visi elementai savo ruožtu yra kiti vienmačiai masyvai iš *m* elementų. Jei masyvo elementai išrenkami indeksais *i* ir *j*, tai adreso operacija *dm[i][j]* teikia *ij*-ąjį masyvo elementą, kurio formatas yra *double*. Kreipinys *dm[i]* būtų *i*-ojo vienmačio masyvo vardas, t.y. to masyvo pirmojo baito adresas, kurio formatas – *double**. Analogiškai kreipinys *dm* jau būtų visų *i*-ųjų, *i=0, 1, ..., n-1* vienmačių masyvų pirmųjų baitų adresų masyvo vardas, kurio formatas jau – rodyklė į rodyklių masyvą, arba *double***. Todėl dinaminę atmintį tenka skirti keliais etapais: pirmiausia reikia paskirti dinaminę atmintį *double** formato adresų masyvui, o tada jau jo elementams – dinaminę atmintį *double* formato elementams. Pati rodyklė, nukreipiama į dvimačio masyvo pirmąjį baitą, būtų *double*** formato. Atmintis išlaisvinama taip pat keliais etapais atvirkščia tvarka.

Visi šie dalykai paaiškinti 8 pavyzdžio programoje. Kad galėtume dėmesį sutelkti tik į dinaminės atminties skyrimo ir išlaisvinimo sintaksę, programos paskirtis tegu bus paprastutis uždavinys: vien tik formuosime *nxm* dvimatį masyvą, kurio elementai bus *i*j*; *i* ir *j* – atitinkamai eilutės ir stulpelio matematiniai indeksai.

```
#include <iostream>
using namespace std;
//
int main( ) {
        //
        int** array; // dvimatis masyvas-rodyklė
        int n, m;  // masyvo matai
        //
        cout<<"Iveskite masyvo matus\n";
        cin>>n>>m;
```

```
        cout<<"Masyvo matai yra "<<n<<" x "<<m<<endl;
        //
        // Dinaminės atminties skyrimas
        //
        array = new int*[ n ]; // vieta adresų masyvui
        for( int i = 0; i < n; i++ )
                array[ i ] = new int[ m ]; // vietos vienmačiams masyvams
        //
        // Masyvo formavimas ir spausdinimas
        //
        for( int i = 0; i < n; i++ )
                for ( int j = 0; j < m; j++ )
                        array[ i ][ j ] = (i+1)*(j+1);
                        //*(*(array+i)+j) = (i+1)*(j+1); // alternatyvi sintaksė
        //
        cout<<"Suformuotas masyvas:\n";
        for( int i = 0; i < n; i++ ){
                for ( int j = 0; j < m; j++ )
                        cout<<array[ i ][ j ]<<"  ";
                cout<<endl;
        }
        //
        // Atminties išlaisvinimas
        //
        for( int i = 0; i<n; i++ )
                delete [ ] array[ i ];
        delete [ ] array;
        //
        system("pause");
        return 0;
}
```

Šioje programoje taip pat parodyta, kaip į dvimačio masyvo elementus kreiptis taikant įreikšminimo operaciją: **(array+i)* teikia *i*-ojo vienmačio masyvo adresą, o **(*(array+i)+j)* – jau patį *ij*-ąjį elementą.

10.6 RODYKLĖS Į EILUTES

Kaip ir realizuojant eilutės duomenis vienmačiais simboliniais masyvais, taip ir realizuojant eilutes simbolinio formato rodyklėmis, taikoma kiek supaprastinta sintaksė. Palyginkit: 9 pavyzdys:

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

```
        //
        char eilm[ ] = "Realizavimas masyvu";
        char* eilr   = "Realizavimas rodykle";
        //
        cout<<eilm<<endl;
        cout<<eilr<<endl;
        //
        //eilm++; klaida: eilm yra adresas, t.y. konstanta
        eilr+=13; // viskas tvarkoj
        //
        cout<<eilr<<endl;
        //
        system("pause");
        return 0;
}
```

Dvimačio eilučių masyvo alternatyva – vienmatis *char* rodyklių masyvas; ir tai yra efektyviau. Vėlgi palyginimui – 10 pavyzdys:

```
#include <iostream>
using namespace std;
//
int main( ) {
        //
        char sdm[ 7 ][ 15 ] = { "Pirmadienis", "Antradienis", "Treciadienis",
                                "Ketvirtadienis", "Penktadienis", "Sestadienis",
                                "Sekmadienis" };
        char* sdr[ 7 ] = { "Pirmadienis", "Antradienis", "Treciadienis",
                           "Ketvirtadienis", "Penktadienis", "Sestadienis",
                           "Sekmadienis" };
        //
        for( int i = 0; i < 7; i++ )
                        cout<<sdm[ i ]<<endl;
        cout<<endl;
        for( int i = 0; i < 7; i++ )
                        cout<<sdr[ i ]<<endl;
        //
        system("pause");
        return 0;
}
```

Pirmuoju atveju eilutės atmintyje būtų saugomos kaip dvimatis 7 x 15 baitų masyvas, t.y

Pirmadienis\0␣␣␣ Antradienis\0␣␣␣ . . .

Antruoju atveju atmintis būtų eikvojama racionaliau:

Pirmadienis\0Antradienis\0 . . .

11 pavyzdys. Perrašysime 9-ojo skyriaus 2-ojo pavyzdžio programą – eilutės pertvarkymo vien į didžiąsias ar vien į mažąsias raides nekeičiant kitų simbolių – su rodyklių konstrukcijomis.

```
#include <iostream>
#include <cstring>
using namespace std;
//
void showString( char*, char* ); // funkcijos eilutei spausdinti prototipas
void toUpperCase( char*, int );  // funkcijos eilutei pertvarkyti į didžiąsias raides
void toLowerCase( char*, int );  // funkcijos eilutei pertvarkyti į mažąsias raides
//
int main( ) {
   char text[ ] = "This is a Mixed Case";  // pradinis tekstas
   //
   int a = 'a',   A = 'A',    space = ' ', difference = a - A;  // simbolių aritmetiniai kodai
   cout<<"ASCII codes for a, A and ' '   "<<a<<"  "<<A<<"  "<<space
       <<endl<<"  a - A = "<<difference
       <<endl<<endl;
     //
   showString( "Initial text", text );
   toUpperCase( text, difference );
   showString( "Changed to upper case", text );
   toLowerCase( text, difference );
   showString( "Changed to lower case", text );
     //
   system("pause");
          return 0;
}
//
void showString( char* t, char* s ) {
          while( *t )
                cout<<*t++;
          cout<<":  ";
          while( *s )
               cout<<*s++;
          cout<<endl<<endl;
}
//
void toUpperCase( char* s, int d ) {
          while( *s ) {
            if( *s >= 'a' )
                        *s -= d;
            s++;
          }
```

```
}
//
void toLowerCase( char* s, int d ) {
        while( *s ) {
          if( *s >= 'A' && *s <= 'Z' )
                        *s += d;
          s++;
        }
}
```

Taigi, į visas funkcijas eilutės duomenys perduodami *char* rodyklėmis. Panagrinėkim paskutiniąją funkciją. Funkcijos viduje visus eilutės simbolius perrenka ciklas *while(*s){ . . .s++; }* , kurio sąlyga yra tiesiog simbolio reikšmė. C++ kalboje tai galima, nes simbolio reikšmė yra simbolio aritmetinis kodas ASCII kodų lentelėje, o bet koks nenulinis duomuo loginiame reiškinyje interpretuojamas kaip *true*. Atlikus reikiamus veiksmus su simbolio reikšme, toliau pereinama prie kito simbolio: tam operacija *s++;* keičiama jau rodyklės reikšmė (bet šis pokytis kviečiančiojoje programoje nebus matomas; pagaliau ir atitinkamas faktiškasis argumentas yra konstanta – masyvo vardas). Galiausiai atėjus iki baigiamojo eilutės simbolio *\0*, ši reikšmė operatoriuje *while* bus interpretuota kaip *false* ir ciklas baigsis.

12 pavyzdys rodo, kaip galima operuoti tiesiog atminties ląstelių adresais. Suskaičiuosime, kiek simbolių sudaro vienoje ekrano eilutėje išdėstytą eilutės duomenį. Tegu simbolių eilutės didžiausias galimas ilgis yra 80.

```
#include <iostream>
using namespace std;
//
int main( ){
        const int MAX = 81;
        char eil[ MAX ];
        char* pchar = eil;
        //
        cout<<"Viena eilute iveskite teksta iki 80 simboliu\n";
        cin.get( eil, 81, '\n' );
        //
        while( *pchar )          // ciklas iki eilutės pabaigos \0
                        pchar++; // baigiant ciklą – galinio eilutės simbolio adresas
        //
        cout<<"Tekste \""<<eil<<" yra "<<pchar-eil<<" simboliu\n";
        //
        system( "pause" );
        return 0;
}
```

Objektus, kaip ir paprastus kintamuosius ar masyvus, taip pat operatoriumi *new* galima kurti dinaminėje atmintyje. Aišku, dabar į operatoriumi paskirtą atminties sritį reiks nutaikyti atitinkamo – klasės – tipo rodyklę. Metodai objekto rodyklei kviečiami taikant įreikšminimo operaciją arba operacija ->. Visa ši sintaksė pagrečiui su mums jau žinoma sintakse kuriant objektus statinėje atmintyje parodyta 13 pavyzdyje, formuojant jau žinomos klasės *Distance* (10 skyriaus 5-7 programų pavyzdžiai) objektus.

```
#include <iostream>
using namespace std;
//
class Distance {
private:
        int feet;
        double inches;
        //
public:
        Distance ( ): feet( 0 ), inches( 0. ) { }
        Distance ( int ft, double in ): feet( ft ), inches( in ) { }
        //
        ~Distance ( ) { }
        //
        void getDistance( );
        void showDistance( ) const;
        void addDistances1( const Distance&, const Distance& );
        Distance addDistances2 ( const Distance& );
        //
};
//
//
int main( ) {
        //
        Distance d1( 1, 2.3 ); // objektas statinėje atmintyje
        d1.showDistance( );
        //
        Distance* d2 = new Distance; // dinaminėje atmintyje
        d2->getDistance( );    // metodo kvietimo sintaksė
        (*d2).showDistance( );  // alternatyvi sintaksė
        delete d2; // esant reikalui dinaminę atmintį galima išlaisvinti
        //
        system("pause");
        return 0;
}
//
//
void Distance::getDistance ( ) {
        cout<<"Iveskite pedas ir colius\n";
```

```
        cin>>feet>>inches;
}
void Distance::showDistance ( ) const {
        cout<<"Pedos: "<<feet<<"  coliai: "<<inches<<endl;
}
void Distance::addDistances1 ( const Distance& d1, const Distance& d2 ) {
        inches = d1.inches + d2.inches;
        feet   = d1.feet   + d2.feet;
        if( inches >= 12. ) {
                inches-= 12.;
                feet++;
        }
}
Distance Distance::addDistances2 ( const Distance& d ) {
        int ft = feet + d.feet;
        double in = inches + d.inches;
        if( in >= 12. ) {
                in-= 12.;
                ft++;
        }
        return Distance( ft, in );
}
```

Jei turime kelis vienodo tipo objektus, juos galime sudėti į masyvus (kaip 10-ojo skyriaus 8 pavyzdyje), arba galime skelbti rodykles į objektus ir jas talpinti į masyvus. Dažnai antrasis būdas yra lankstesnis ir efektyvesnis: objektai atmintyje gali būti išdėstyti bet kur, kur tik yra laisvos vietos, o jų pirmųjų baitų adresai bus vienmačiame rodyklių masyve. Šiuos dalykus iliustruosime paprastos klasės *Person*, turinčios tik vieną lauką asmens pavardei talpinti, pavyzdžiu. Tegu klasės objektai kuriami interaktyviai, įvedant pavardes klaviatūra, o objektų gali būti iki 100. Tolesniame programos pavyzdyje pabandysime šias pavardes išrikiuoti pagal abėcėlę.

14 pavyzdys:

```
#include <iostream>
#include <cstring>
using namespace std;
//
class Person{
private:
        char name[ ];
public:
        Person( ) {
                strcpy( name, "  " );
        }
        Person( char p[ ] ) {
                strcpy(name, p );
        }
        //
```

```
        ~Person( ) { };
        //
        void getName( ) {
                cout<<"Iveskite pavarde:\n";
                cin>>name;
        }
        void showName( ) {
                cout<<name<<endl;
        }
};
//
int main( ) {
        Person* personp[ 100 ]; // rodyklių į objektus masyvas
        int n = 0;              // objektų skaičius
        char s;                 // įvedimui tęsti/nutraukti
        //
        do {
                personp[ n ] = new Person;  // konstruktoriumi be argumentų
                                            // formuojama n-ojo objekto atmintis
                personp[ n ] -> getName( ); // į atmintį įvedamos objekto
                                            // laukų reikšmės
                n++;
                cout<<"Ar testi pavardziu ivedima? t/n\n";
                cin>>s;
        } while( s == 't' );
        //
        for( int i = 0; i < n; i++ ){
                cout<<"Asmuo Nr. "<<i+1<<"   ";
                personp[ i ] -> showName( );
        }
        //
       system("pause");
       return 0;
}
```

10.8 RODYKLĖS Į RODYKLES

Pratęsim 13-ojo pavyzdžio programą: išrūšiuosime *Person* objektus abėcėlės tvarka pagal pavardes. Ši programa aiškiai rodo, kodėl rodyklių į objektus masyvas yra žymiai lankstesnė ir efektyvesnė konstrukcija už objektų masyvą. Rūšiuosime ne pačius objektus, t.y. objektų vieta kompiuterio atmintinėje liks ta pat, o tik rodyklių į tuos objektus masyvą – todėl reiks gerokai mažiau veiksmų.

Rūšiuosime jau mums pažįstamu burbulo algoritmu (žr. šio skyriaus 6-ąją programą). Klasę *Person* truputį pakeisime: pavardės lauką parašysime ne *char* masyvo formato, o vidinės C++ klasės *string* formato. Taip yra patogiau, kadangi šiai klasei yra perkrauta santykio operacija > (z > . . . > b > a > Z . . . > B > A) ir prieskyros operatorius. Darbui su klase būtinas antraštinis failas *string*.

15 pavyzdys:

```
#include <iostream>
#include <string>
using namespace std;
//
class Person{
private:
        string name;
public:
        Person( ){
                name = "  ";
        }
        Person( string s ){
                name = s;
        }
        //
        ~Person( ) { };
        //
        void getName( ) {
                cout<<"Iveskite pavarde:\n";
                cin>>name;
        }
        void showName( ) {
                cout<<name<<endl;
        }
        string returnName( ) {
                return name;
        }
};
//
int main( ) {
        //
        void pointerSort( Person**, int );  // 1
        //
        Person* personp[ 100 ];
        int n = 0;
        char s;
        //
        do {
                personp[ n ] = new Person;
                personp[ n ] -> getName( );
                n++;
                cout<<"Ar testi pavardziu ivedima? t/n\n";
                cin>>s;
        } while( s == 't' );
        //
```

```
        cout<<"Pradinis pavardziu masyvas\n";
        for( int i = 0; i < n; i++ ){
                cout<<"Asmuo Nr. "<<i+1<<"  ";
                personp[ i ] -> showName( );
        }
        //
        pointerSort( personp, n );  // 2
        //

        cout<<endl;
        cout<<"Isrusiuotas pavardziu masyvas\n";
        for( int i = 0; i < n; i++ ){
                cout<<"Asmuo Nr. "<<i+1<<"  ";
                personp[ i ] -> showName( );
        }
        //
   system("pause");
        return 0;
}
//
void pointerSort( Person** p, int n ) {
        //
        void sort2pointers( Person**, Person** );  // 3
        //
        for( int i = 0; i < n-1; i++ )
                for( int j = i+1; j < n; j++ )
                        sort2pointers( p+i, p+j );  // 4
}
//
void sort2pointers( Person** p1, Person** p2 ) {
        if( (*p1) -> returnName( ) > (*p2) -> returnName( ) ) {  // 5
                Person* temp = *p1;
                *p1 = *p2;
                *p2 = temp;
        }
}
```

Kad suprastume, kaip veikia programa, turime turėti galvoje, kad:

1. Jei *personp[k]* elementas yra *Person** tipo, t.y. rodyklė į *k*-ąjį objektą kompiuterio atmintyje, tai *Person*** tipo kintamasis yra rodyklė į vienmatį rodyklių masyvą,rodanti į to masyvo atminties srities pirmąjį baitą.
2. *personp*, teikiamas kviečiant funkciją *pointerSort*, yra tokios rodyklės į rodyklių masyvą reikšmė – rodyklių masyvo pirmojo baito adresas.
3. Rodyklių masyve jo elementai atrenkami rodyklėmis, todėl tų elementų tipas turi būti *Person***.
4. Faktiškieji argumentai *p+i* ir *p+j* leidžia atrinkti rodyklių masyvo elementus pradedant nuo jo pirmojo baito adreso *p*.

5. Būtinos dvi įreikšminimo operacijos lauko reikšmei gauti: *p1* yra rodyklės į rodyklę tipo, todėl operacija **p1* teiks rodyklių masyvo elemento reikšmę – rodyklę; o toliau jau *(…)* - > operacija teiks atminties srities, į kurią ši nutaikyta, reikšmę. Tai atminties sričiai- klasės objektui kviečiamas metodas *returnName* ir grąžina lauko reikšmę – *string* klasės objektą – pavardę. *string* klasės objektams yra perkrauta > operacija.

10.9 RODYKLĖS Į FUNKCIJAS

Galima skelbti rodyklę į funkciją ir ją kviesti per rodyklę; rodyklė šiuo atveju rodo į funkcijai skirtos atminties srities pirmąjį baitą. Skelbimo metu nurodomas funkcijos grąžinamos reikšmės formatas, skliausteliuose – pati rodyklė, ir funkcijos argumentų sąrašas (kaip ir funkcijos prototipe). Tokią rodyklę galima nukreipti tik į tas funkcijas, kurių grąžinama reikšmė ir prototipas tiksliai sutampa.

16 pavyzdys. Programa vien tik sintaksei parodyti: parašysim funkcijas dviejų sveikųjų skaičių sumavimui ir sandaugai ir jų pagalba suskaičiuosim dviejų aritmetinių reiškinių reikšmes.

```
#include <iostream>
using namespace std;
//
int sum( int, int );         // abiejų funkcijų prototipai
int product( int, int );   // turi tiksliai sutapti
//
int main( ){
        int (*pfunction ) ( int, int );  // rodyklė į funkciją; prototipas sutampa su
                                         //  funkcijų sum ir product prototipais
        pfunction = product;   //  rodyklę privaloma inicializuoti
        cout<<"10*11= "<<pfunction( 10, 11 )<<endl;
        pfunction = sum;      // rodyklę nukreipiame į kitą funkciją
        cout<<"3*(4+5)+6= "<<
                        pfunction( product( 3, pfunction( 4, 5 )), 6 )<<endl;
        //
        system( "pause" );
        return 0;
}
//
int sum( int s1, int s2 ) {
        return s1+s2;
}
//
int product( int s1, int s2 ) {
        return s1*s2;
}
```

Rodyklės į funkcijas savo ruožtu gali būti argumentai, masyvo elementai ir pan. 17 pavyzdys su rodyklėmis į funkcijas-argumentais, kuriame skaičiuojamos masyvo elementų kvadratų ir kubų sumos:

```
#include <iostream>
using namespace std;
//
double square( double );  // abiejų funkcijų prototipai
double cube( double );    // tiksliai sutampa
double sum( double array[ ], int n, double (*pfunction) (double) ); //argumentų sąraše
                                                    // teikiamas rodyklės į funkciją prototipas
//
int main( ){
        double array[ ] = { 12., -5., 10., 2., 3. };
        int n = ( sizeof array ) / ( sizeof array[ 0 ] );
        //
        cout<<"Sum of squares  "<<sum( array, n, square )<<endl;
        cout<<"Sum of cubes  "<<sum( array, n, cube )<<endl;
        //
        system( "pause" );
        return 0;
}
//
double square( double x ){
        return x*x;
}
//
double cube( double x ){
        return x*x*x;
}
//
double sum( double x[ ], int n, double (*pf) (double) ){
        double s = 0.;
        for( int i = 0; i < n; i++ )
                s+= pf( x[i] );
        return s;
}
```

10.10 FUNKCIJOS GRĄŽINAMA REIKŠMĖ-RODYKLĖ

Funkcijos kaip ir bet kokią kitą reikšmę gali grąžinti rodyklę arba adresą. Vienintelis dalykas, į kurį reikia atkreipti dėmesį – negalima grąžinti lokalaus funkcijos kintamojo adreso. Panašūs dalykai jau buvo aptarti 5.8 skyriuje, kalbant apie funkcijos grąžinamą reikšmę-nuorodą. Taigi, lokalus kintamasis išėjus iš funkcijos bus sunaikintas, todėl rodyklė rodys nežinia į ką. Panagrinėkite šį klaidingą programos pavyzdį.

18 pavyzdys. Funkcijoje skaičiuojamas skaičiaus kubas.

```
#include <iostream>
using namespace std;
//
double* cube( double );  //
//
int main( ){
        double number = 5.;
        double* p = NULL;
        //
        p = cube( number );
        cout<<"Skaiciaus  "<<number<<" kubas yra "<<*p<<endl;
        //
        system( "pause" );
        return 0;
}
//
double* cube( double x ){
        double result = x*x*x;
        return &result;
}
```

Kad programa netvarkinga, rodo ir kompiliatoriaus įspėjimas:

warning C4172: returning address of local variable or temporary

Funkcijos kintamasis *result* pasibaigus funkcijai turi būti sunaikintas (bet kartais, kai programa eikvoja nedaug atminties – nebus), todėl funkcijos grąžinamas adresas bus atsitiktinis. Tokia programa kelis kartus ją leidžiant gali teikti vis kitus nesuprantamus rezultatus. Geležinė taisyklė grąžinant iš funkcijos rodyklę – niekada negalima grąžinti lokalaus kintamojo adreso; galima – tik globalaus, arba per argumentų sąrašą ateinančio kintamojo. Panašus tvarkingas programos pavyzdys yra 5.8 skyriuje. Vis tik jei dėl kažkokių priežasčių norėtume išlaikyti ankstesnę programos ir funkcijos struktūrą, sprendimas yra. Funkciją turėtume perrašyti kad ir taip:

```
double* cube( double x ){
        double* result = new double;
        *result = x*x*x;
        return result;
}
```

Dabar lokalus kintamasis sukuriamas dinamiškai su operatoriumi *new* ir gyvuos, kol bus sunaikintas operatoriumi *delete* arba (kaip mūsų programoje) pasibaigus programai – viskas veiks tvarkingai.

Tačiau tokia programos struktūra gali prišaukti kitą bėdą: vadinamąjį atminties nuotėkį (angl. memory leak). Kadangi kintamasis *result* nėra sunaikinamas, kviečiant panašios struktūros funkciją cikle, kiekvienąkart bus skiriama dinaminė atmintis vis naujam kintamajam

result, ir laisvosios kompiuterio atminties apimties mažės sulig kiekvienu ciklo žingsniu. Todėl programoje po rezultato spausdinimo turėtume sunaikinti rodyklę: *delete p;* .

10.11 RAKTAŽODIS *this*

this yra rodyklė į šiuo metu apdorojamą objektą. Tokia rodyklė pirmiausia reikalinga pačiam kompiliatoriui, kai klasės metode kreipiamasi į bet kurį klasės lauką: reikia žinoti lauko atminties srities pirmojo baito adresą. Pavyzdžiui, 10-ame skyriuje kelis kartus perrašytoje angliškų ilgio matų klasėje *Distance* yra metodas

```
. . .
void getDistance ( ) {
        cout<<"Iveskite pedas ir colius\n";
        cin>>feet>>inches;
}
. . .
```

Čia *feet* ir *inches* – darbinio, šiuo metu programoje apdorojamo objekto laukai. Kad tie laukai klasės metodams būtų prieinami, jiems neišreikštai perduodama rodyklė *this* į pačios klasės objektą. Tuo būdu, įvesties operatorius metode yra ekvivalentiškas operatoriui

```
cin>>this->feet>>this->inches;
```

Kai kuriose situacijose *this* praverčia ir programuotojams. Kai reikia iš metodo sugrąžinti rodyklę į darbinį objektą – galima rašyti operatorių *return this;* , o kai norima sugrąžinti darbinį objektą – *return *this;* . Kita situacija, kai patogu taikyti this – rašyti metodus-konstruktorius su argumentų vardais, sutampančiais su klasės laukų vardais. 10-ojo skyriaus 2 programos laukai pavadinti buvo *data1* ir *data2*, o konstruktorių parašėme taip:

```
C ( int d1, double d2 ): data1( d1 ), data2( d2 ) { }
```

Daugelis programuotojų konstruktorius rašo kitokiu, jiems aiškesniu stiliumi. Minėtąjį konstruktorių šiuo stiliumi galėtume perrašyti taip:

```
C ( int data1, double data2 ) {
        this -> data1 = data1;
        this -> data2 = data2;
}
```

Daugiau pavyzdžių su rodykle *this* rasite 14-ame skyriuje.

10.12 C++/CLI KALBOS DINAMINĖS ATMINTIES YPATYBĖS

C++/ISO/ANSI ir C++/CLI kalbų kompiliatoriai šiuo aspektu labai stipriai skiriasi. C++/CLI kompiliatorius automatiškai atima atmintį iš nebenaudojamų atminties blokų – nebereikalinga *delete* operacija; tai sumažina atminties nuotėkio klaidų. Be to *heap* atmintis automatiškai kompaktizuojama. Tam vietoj *new* operacijos gali būti taikoma operacija *gcnew* (pirmosios raidės – nuo angl. garbage collector – atminties rinktuvo).

Tačiau automatiškai kompaktizuojant atmintį anksčiau nustatytos rodyklės rodys jau į tas atminties ląsteles, kuriose bus visai kitas turinys nei anksčiau. Reikia, kad rodyklės sektų kompaktizavimo metu iš vienos atminties vietos į kitas persiunčiamų duomenų adresus – todėl šiame kalbos standarte vietoje rodyklių naudojami jų atitikmenys – rankenėlės (vertinys iš angl. termino handle) – „^“. Sintaksė iš pirmo žvilgsnio atrodo visai nesuprantama. Pavyzdys: dinaminė atmintis sveikaskaitiniam dvimačiam masyvui iš 4 eilučių ir 10 stulpelių skiriama operatoriumi

array< int, 2 >^ = gcnew array< int, 2 >(4, 10)

Beje, šiame kalbos standarte masyvo elementus galima išrinkti FORTRANo stiliumi:

array[i, j].