5. FUNKCIJOS

Funkcija – įvardinta savarankiška operatorių grupė, kurią startinė C++ funkcija *main* (ar kuri kita žemesnio lygio funkcija) gali pakviesti darbui; pasibaigus funkcijos darbui, valdymas grąžinamas ją kvietusiai programai. Funkcijos darbas gali priklausyti nuo vadinamųjų argumentų. Argumentų reikšmes kvietimo metu nurodo kviečiančioji programa funkcijos argumentų sąraše. Funkcija baigiantis jos darbui kviečiančiajai programai gali grąžinti savo darbo rezultatą – reikiamo tipo duomenį. Jei funkcijos darbo rezultatas yra masyvas(-ai), funkcija grąžina rezultatus per argumentų sąrašą arba per grąžinamą reikšmę – rodyklę. Išsamiai apie tai – toliau.

Kokia prasmė naudoti funkcijas? Funkcijų paskirtis –

- Konceptualizuoti programos struktūrą – o tai yra pamatinis procedūrinio programavimo principas. Kiekvieną konkrečiam tikslui įgyvendinti skirtą operatorių grupę apiforminus kaip funkciją, bus žymiai paprasčiau suprasti programos struktūrą.
- Lengviau suvaldyti programą. Sakykim, kelių tūkstančių operatorių programą sudalinus kelias funkcijas, turinčias po keliasdešimt (iki poros šimtų) operatorių, ją bus nepalyginti paprasčiau derinti (galima derinti atskirai kiekvieną funkciją) ir skaityti. Sakoma, kad programa (funkcija) yra tvarkingai užprogramuota tada, kai visą ją galima aprėpti vienu žvilgsniu, t.y. jos tekstas telpa į kompiuterio ekraną.
- Trumpinti programos tekstą. Jei programa turi atlikti kelias panašių veiksmų grupes su skirtingais duomenimis, tuos veiksmus patogu apiforminti kaip funkciją ir prireikus ją kviesti darbui su skirtingomis argumentų reikšmėmis.

5.1 FUNKCIJOS IDĖJA

1 pavyzdys. Sakykim, norime į kompiuterio ekraną išvesti lentelę su sveikųjų aritmetinių duomenų charakteristikomis – ląstelės ilgiu ir duomens diapazonu. Lentelės antraštę ir paskutinę eilutę norime aprėminti 60-ies simbolių “*“ eilutėmis. Kad nereikėtų tris kartus kartoti to paties ciklo žvaigždutėms išvesti, tą apiforminsime kaip funkciją *stars* ir ją tris kartus reikiamu metu iškviesime darbui. Tuo būdu, programa bus trumpesnė ir aiškesnė.

```
#include <iostream>
using namespace std;
//
void stars();   // funkcijos prototipas
//
int main( ){
        //
        stars();   // kviečiama funkcija pirmajai eilutei išvesti
        cout<<"Duomens tipas     Ilgis     Diapazonas\n";
        stars();  // eilutė lentelės antraštei atriboti
```

```
                    cout<<"char         1B     -128 - +127"<<endl
                        <<"short        2B     -32768 - +32767"<<endl
                        <<"int          4B     ~-2e9 - ~+2e9"<<endl
                       <<"long          4B     ~-2e9 - ~+2e9"<<endl;
        stars();  // eilutė lentelei užbaigti
        //
        system( "pause" );
       return 0;
}
//
void stars(){      // funkcijos kūnas
        for( int i = 0; i < 60; i++ ) cout<<'*';
        cout<<endl;
}
```

Čia pateikėme paprasčiausios funkcijos pavyzdį: funkcijos darbo rezultatas nepriklauso nuo jokių parametrų, todėl funkcijos argumentų sąrašas *( )* yra tuščias; funkcija kviečiančiai programai negrąžina jokio duomens reikšmės-rezultato – todėl ji skelbiama „tuščia“ funkcija – *void*, ir neturi jokio operatoriaus *return*. Beje, kad būtų dar aiškiau, jog funkcija argumentų neturi, sąraše galima įrašyti atitinkamą raktažodį: *(void).* Funkcijai, kaip ir paprastam kintamajam, reikia suteikti vardą, kuriam keliami lygiai tokie pat reikalavimai, kaip ir kintamojo vardui: vardą sudaro tik raidės, skaitmenys ir apatinis brūkšnelis “ _“ , o pirmasis simbolis gali būti tik raidė arba brūkšnelis. Kad nekiltų jokių problemų, sutarkime funkcijoms parinkti tik unikalius programos rėmuose vardus.

Kad kompiliatorius tinkamai parengtų funkcijos iškvietimą, jam reikia pranešti funkcijos vardą, argumentų skaičių ir jų tipus, grąžinamos reikšmės tipą (arba pranešti, kad funkcija negrąžina nieko). Tai galima padaryti dviem būdais: prieš funkcijos kvietimą (arba tiesiog prieš *main*) užrašyti vadinamąjį funkcijos prototipą kaip programos pavyzdyje, arba pačią funkciją užrašyti prieš ją kviečiančią programą kaip 2-ame pavyzdyje:

```
#include <iostream>
using namespace std;
//
void stars(){      // funkcijos kūnas
        for( int i = 0; i < 60; i++ ) cout<<'*';
        cout<<endl;
}
//
int main( ){
        //
        stars();   // kviečiama funkcija pirmajai eilutei išvesti
        cout<<"Duomens tipas    Ilgis    Diapazonas\n";
       stars();  // eilutė lentelės antraštei atriboti
                    cout<<"char         1B     -128 - +127"<<endl
                        <<"short        2B     -32768 - +32767"<<endl
                        <<"int          4B     ~-2e9 - ~+2e9"<<endl
                        <<"long         4B     ~-2e9 - ~+2e9"<<endl;
        stars();  // eilutė lentelei užbaigti
```

```
        //
        system( "pause" );
        return 0;
}
```

Toliau visose programose rinksimės pirmąjį funkcijos apibūdinimo su prototipu variantą: tokiu atveju yra aiškesnė programos struktūra. Nepamirškite *;* po prototipo.

5.2 FUNKCIJOS ARGUMENTAI

Argumentai reikalingi, kai funkcijos darbas priklauso nuo tam tikrų duomenų reikšmių.

3 pavyzdys. Sakykim, norime parašyti universalesnę funkciją *stars*, kad ji galėtų spausdinti bet kokių nurodytų simbolių eilutę, o ta eilutė galėtų būti nurodyto ilgio. Tokiu atveju į funkcijos argumentų sąrašą (jis vadinamas formaliųjų argumentų sąrašu) reikia įtraukti du argumentus: vienas, *char* formato, skirtas simboliui perduoti į funkciją, o antrasis – *int* – reikš simbolių eilutės ilgį.

```
#include <iostream>
using namespace std;
//
void stars( char, int );
//
int main( ){
        //
        stars('-', 65 );  // pirmoji eilutė - iš 65 simbolių '-'
        cout<<"Duomens tipas    Ilgis    Diapazonas\n";
       stars('=', 60 );  // eilutė iš 60 simbolių '='
                        cout<<"char          1B      -128 - +127"<<endl
                            <<"short         2B      -32768 - +32767"<<endl
                            <<"int           4B      ~-2e9 - ~+2e9"<<endl
                            <<"long          4B      ~-2e9 - ~+2e9"<<endl;
        stars( '-', 65 );  // eilutė iš 65 simbolių '-'
        //
        system( "pause" );
        return 0;
}
//
void stars( char c, int n ){      // funkcijos kūnas
        for( int i = 0; i < n; i++ ) cout<<c;
        cout<<endl;
}
```

Kaip matyti iš šios programos, prototipe būtina pateikti informaciją apie funkcijos argumentus ir jų tipus. Pačių argumentų vardų nurodyti nebūtina. Vardus rašysim tik tada, kai tie vardai leis susigaudyti, kuriam tikslui argumentai skirti, o ir tie vardai gali skirtis nuo argumentų vardų formaliųjų argumentų sąraše ir argumentų vardų iškviečiant funkciją. Pastarieji vadinami faktiškaisiais argumentais. Šioje programoje funkcija kviečiama teikiant vietoje formaliųjų

argumentų konkrečias konstantines reikšmes. Toliau, kaip ir visose savo programose, vengsime operuoti konkrečiomis konstantomis ir funkcijų kvietimuose faktiškuosius argumentus teiksime kintamųjų pavidalu kaip 4-ame pavyzdyje:

```
#include <iostream>
using namespace std;
//
void stars( char s, int sk );
//
int main( ){
        //
        char symbol;
        int number;
        //
        cout<<"Iveskite simboli ir ju skaiciu\n";
        cin>>symbol>>number;
        cout<<"Eilute bus formuojama is "<<number<<" simboliu\n\n";
        //
        stars( symbol, number );   // pirmoji eilutė - iš 65 simbolių '-'
        cout<<"Duomens tipas    Ilgis    Diapazonas\n";
       stars( symbol, number );  // eilutė iš 60 simbolių '='
                    cout<<"char          1B     -128 - +127"<<endl
                     <<"short         2B     -32768 - +32767"<<endl
                     <<"int             4B     ~-2e9 - ~+2e9"<<endl
                     <<"long          4B     ~-2e9 - ~+2e9"<<endl;
        stars( symbol, number );  // eilutė iš 65 simbolių '-'
        //
        system( "pause" );
       return 0;
}
//
void stars( char c, int n ){      // funkcijos kūnas
        for( int i = 0; i < n; i++ ) cout<<c;
        cout<<endl;
}
```

Šioje programoje faktiškųjų argumentų reikšmės funkcijai pranešamos kintamųjų pavidalu. Formaliųjų, faktiškųjų argumentų vardai ir vardai prototipe tiems patiems duomenims programoje parinkti skirtingi, nors galima rinktis ir visus vienodus vardus.

Akivaizdu, kad formaliųjų ir faktiškųjų argumentų sąrašai sutapti pagal ilgį, prasmę ir tipą. Atitinkamų formaliųjų ir faktiškųjų argumentų pora, kai argumentai perduodami funkcijai reikšme (apie tai – 5.4 skyriuje), gali būti ir skirtingų tipų: tada faktiškasis argumentas pervedamas į formaliojo argumento tipą automatiškai. Vienas toks pavyzdys pateiktas 5.3 skyriuje; tačiau tokių situacijų savo programose vengsime.

Ši programa jau aiškiai rodo, kad funkcija – tai formali veiksmų seka, siejanti formalius duomenis, kuri pradedama vykdyti tik funkciją iškvietus darbui ir nurodžius, su kokiais faktiškaisiais duomenimis atlikti veiksmus.

Visi funkcijoje paskelbti kintamieji (šiame pavyzdyje – tik ciklo skaitiklis *i*) yra lokalūs funkcijos kintamieji, matomi tik iš pačios funkcijos. Šie kintamieji baigia egzistuoti sulig funkcijos baigiamuoju skliausteliu (arba, jei funkcijoje paskelbti vidinėje matomumo srityje – sulig jos baigtimi). Tokių kintamųjų vardai kitose programos funkcijose gali būti taikomi visai kitiems tikslams.

5.3 GRĄŽINAMA REIKŠMĖ

Jei funkcijos darbo rezultatas – vieno duomens reikšmė, patogu tokį rezultatą grąžinti iš funkcijos kviečiančiajai programai operatoriumi *return*, panašiai kaip startinė funkcija *main* korektiškai baigdama savo darbą grąžina sutartinę reikšmę *0*.

5 pavyzdys. Parašysime programą, kuri Farenheito temperatūrą perskaičiuotų į Celsijaus skalę. Farenheito temperatūra įvedama iš klaviatūros, o rezultatas – atitinkama Celsijaus temperatūra išvedama į kompiuterio ekraną. Skaičiavimas apiformintas kaip funkcija *F_to_C*:

```
#include <iostream>
using namespace std;
//
double F_to_C( double );
//
int main( ){
        //
        double f, c;   // Farenheito ir Celsijaus temperatūros
        char s;        // dialogo kintamasis
        //
        do{
            cout<<"Iveskite temperatura Farenheito laipsniais   ";
            cin>>f;
            cout<<"Farenheito temperatura: "<<f;
            c = F_to_C( f );
            cout<<"  Celsijaus temperatura: "<<c<<endl;
                cout<<"Ar testi skaiciavima? t/n\n";
                cin>>s;
        } while( s != 'n' );
        //
        system( "pause" );
        return 0;
}
  //
double F_to_C( double ft ){
        // Argumentai: ft – Farenheito temperatūra
        // Grąžinama reikšmė - Celsijaus temperatūra
        double ct = ( ft - 32. )*5./9.;
        return ct;
}
```

Kaip matyti iš programos, funkcija operatoriumi *return* grąžina *double* tipo rezultatą; grąžinamos reikšmės tipą būtina nurodyti ir funkcijos prototipe. Jei *void* funkcijos kviečiamos paskiru operatoriumi, tai į grąžinančias reikšmę funkcijas reikia kreiptis vykdomajame C++ operatoriuje (čia – prieskyros operatoriuje. Prieskyros operatorius grąžintą reikšmę nukopijuoja į ląstelę *c*).

6 pavyzdys. Ankstesnę programą galima gerokai sutrumpinti, atsisakant nebūtinų kintamųjų:

```
#include <iostream>
using namespace std;
//
double F_to_C( double );
//
int main( ){
        double f;   // Farenheito temperatūra
        char s;         // dialogo kintamasis

        do{
            cout<<"Iveskite temperatura Farenheito laipsniais  ";
            cin>>f;
            cout<<"Farenheito temperatura: "<<f
                <<"  Celsijaus temperatura: "<<F_to_C( f )<<endl;
            cout<<"Ar testi skaiciavima? t/n\n";
            cin>>s;
       } while( s != 'n' );
        //
        system( "pause" );
        return 0;
}
  //
double F_to_C( double ft ){
        // Argumentai: ft – Farenheito temperatūra
        // Grąžinama reikšmė - Celsijaus temperatūra
        return  ( ( ft - 32. )*5./9. );
}
```

Dabar funkcija kviečiama tiesiog išvesties operatoriuje. Kompiliatorius grąžinamai reikšmei saugoti *main* funkcijoje skiria bevardę ląstelę, o *cout* išveda į ekraną šios ląstelės turinį. Lygiai taip funkcijoje *F_to_C* aritmetinio reiškinio reikšmė išsaugoma bevardėje ląstelėje, o *return* jos turinį praneša kviečiančiai programai.

7 pavyzdys iliustruoja atvejį, kai formaliojo ir faktiškojo argumentų pora gali būti skirtingų tipų. Parašysim programą, kuri įvestą realųjį skaičių pakels įvestu teigiamu sveikuoju rodikliu, o vėliau tą patį skaičių pakels gautuoju rezultatu.

```
#include <iostream>
using namespace std;
//
double power( double x, int n );
```

```
//
int main( ){
        double number;
        int index;
        //
        cout<<"Iveskite skaiciu ir laipsnio rodikli\n";
        cin>>number>>index;
        cout<<"Pradinis skaicius "<<number<<" rodiklis  "<<index<<endl;
          //
        cout<<"Rezultatas number ^ index "
            <<power( number, index )<<endl;
        cout<<"Rezultatas number ^ (number ^ index) "
            <<power( number, power( number, index ) )<<endl;
        //
        system( "pause" );
        return 0;
}
  //
double power( double x, int n ){
        // Argumentai: x – pradinis skaičius
        //          n – skaičiaus laipsnio rodiklis
        //Grąžinama reikšmė : r – kėlimo laipsniu rezultatas
        double r = 1.;
        for( int i = 1; i <= n; i++ )    r*= x;
        return r;
}
```

Funkcija *main* antrąjį kartą kviesdama funkciją *power* kaip antrąjį faktiškąjį argumentą teikia kreipinį į tą pačią funkciją *power*. Antrasis argumentas yra *int* formato, o kreipinys į *power* grąžina *double* reikšmę. Šiuo atveju automatiškai bus sužadintas statinis tipų pertvarkymo mechanizmas *static_cast* ir *double* reikšmė bus pervesta į *int* reikšmę atmetant trupmeninę dalį. Galite tuo įsitikinti, leisdami programą ir įvesdami pradinį skaičių su trupmenine dalimi.

Tokie dalykai neleistini taikant duomenų perdavimą į funkciją ir iš jos ne reikšmės mechanizmu, todėl toliau vengsime tokių situacijų.

5.4 ARGUMENTŲ PERDAVIMO MECHANIZMAI

Argumentų perdavimo mechanizmai yra du: reikšme (mechanizmas pagal numatymą; taip argumentai buvo perduodami aukščiau parašytuose pavyzdžiuose), ir adresu. Kai kuriose knygose teigiama, kad yra ir trečiasis būdas – perdavimas rodykle. Rodyklė – tai specialus kintamasis, saugantis kompiuterio atminties ląstelės adresą; per ją netiesiogine kreiptimi galima prieiti ir prie ląstelėje saugomos duomens reikšmės. Tačiau pati argumento-rodyklės (t.y. adreso) reikšmė perduodama reikšmės mechanizmu, todėl faktiškai vis tik yra du argumentų perdavimo mechanizmai.

Pirmuoju atveju kompiuterio atmintyje padaroma kopija faktiškiesiems argumentams skirtos atminties srities, ir kviečiant funkciją vietoje formaliųjų argumentų perduodami šios tarpinės atminties srities atitinkamų faktiškųjų argumentų kopijų pirmųjų ląstelių (tai svarbu masyvams.

Antraip – tiesiog ląstelių) adresai. Dėl to, jei funkcijoje formalieji argumentai keičiami, kviečiančioji programa apie tuos pokyčius nieko nežino – tokiu būdu, faktiškieji argumentai yra apsaugoti nuo nesankcionuotos prieities iš kitų programų pusės. Kartais šis mechanizmas gali būti itin neracionalus: jei, pavyzdžiui, reikia perduoti daug atminties užimančius struktūrų *struct* duomenis – tada jų kopijoms teks skirti irgi daug atminties. Dėl šios priežasties masyvai pagal numatymą perduodami tik adresu, o ne reikšme.

Antruoju atveju kviečiant funkciją vietoje formaliųjų argumentų perduodami atitinkamų faktiškųjų argumentų pirmųjų ląstelių adresai. Todėl dabar funkcijoje formaliesiems argumentams padaryti pokyčiai bus matomi ir kviečiančiojoje programoje: atitinkamai pasikeis faktiškųjų argumentų reikšmės. Tokį mechanizmą reikia taikyti, kai funkcija turi daugiau nei vieną rezultatą. Be to, toks mechanizmas racionalesnis – nereikia tarpinės atminties, bet ne toks saugus: į funkciją atiduodamų duomenų reikšmes funkcijoje netyčia galima pakeisti. O visuotinai priimtas programavimo stilius teigia, kad programos pradiniai duomenys privalo išlikti nepakeisti. Supaprastintai galima sakyti, kad perduodant argumentus adresu formalieji ir faktiškieji vardai tėra tos pačios atminties ląstelės(-ių) skirtingi vardai, sinonimai. Kviečianti programa naudoja vieną vardą, o kviečiamoji – kitą.

8 pavyzdys. Realiame skaičiuje turime išskirti sveikąją ir trupmeninę dalis; abi turi būti gautos to paties formato. Taigi, funkcijos darbo rezultatai – du; teks taikyti rezultatų perdavimo per argumentų sąrašą mechanizmą adresu.

```
#include <iostream>
using namespace std;
//
void parts( double, double& intpart, double& fractpart );  // 1 pastaba
//
int main( ){
        double number, intpart, fractpart;
        //
        cout<<"Iveskite skaiciu\n";
        cin>>number;
        //
        parts( number, intpart, fractpart );  // 2 pastaba
        //
        cout<<"Skaicius: "<<number<<", jo sveikoji dalis: "
            <<intpart<<", jo trupmenine dalis: "<<fractpart<<endl;
        //
        system( "pause" );
        return 0;
}
        // Funkcija
void parts( double n, double& intp, double& fractp ){  //  3 pastaba
        // Argumentai: n - skaicius
        //          intp – rezultatas, sveikoji dalis
        //          fractp – rezultatas, trupmeninė dalis
        long t = static_cast< long >( n );      // sveikojo formato tarpinė ląstelė
        intp = static_cast< double >( t );
```

```
        fractp = n - intp;
    }
```

Pastabos:

1. Argumentai, po kurių formatų yra *&* ženklai, perduodami adresu. Pirmasis argumentas (jo vardas nenurodytas) perduodamas reikšme. Konstrukcijos kaip čia parašyta konstrukcija *double&* yra vadinamos nuorodomis (angl. reference), o *&* – adreso operatoriumi. Paskelbtam kintamajam *k* konstrukcija *&k* reiškia ląstelės, kuriame saugoma kintamojo reikšmė, adresą. Kaip vėliau matysim, toks adresas galės būti priskirtas rodyklei.
2. Kvietimo forma abiems mechanizmams vienoda. Kvietimas čia toks, nes funkcija *void*.
3. Funkcijos apibrėžime adresu perduodamus argumentus irgi būtina pažymėti nuorodų ženklais.

Adresu galima perduoti ne tik argumentus, bet ir funkcijos grąžinamą reikšmę. Tada funkcijos kvietimas galimas ir kairėje prieskyros operatoriaus pusėje. Tokių pavyzdžių ieškokite 5.8 skyriuje.

Kaip minėta, perdavimas adresu dažnai taikomas dėl efektyvumo, tačiau gali būti nesaugus. Saugumui užtikrinti tie adresu perduodami argumentai, kurių reikšmės nenorima keisti, gali būti paskelbti konstantiniais. Pavyzdys-schema:

```
. . .
void f1( int& k1, const int& k2 );  // raktažodis const antrajam argumentui
int main( ){
        int mk1 = 1, mk2 = 2;
        f1( mk1, mk2 );
        return 0;
}
void f1( int& fk1, const int& fk2 ){  // raktažodį reikia pakartoti
        fk1 = 10;  // gerai, keisti galima – argumentas nekonstantinis
        fk2 = 20; // kompiliavimo klaida, argumentas apsaugotas
}
. . .
```

5.5 MASYVAI – FUNKCIJŲ ARGUMENTAI

Esminis dalykas čia – masyvai tarp funkcijų perduodami tik adresu. Nepaisant to, nei funkcijų prototipuose, nei funkcijų formaliųjų argumentų sąrašuose skelbti nuorodų į masyvų vardus nereikia ir negalima: masyvo vardas C kalbose savaime yra masyvo pirmojo elemento pirmojo baito adresas. Kadangi kompiliatoriui kažkaip reikia pranešti, kad kažkuris prototipo ar formaliųjų argumentų sąrašo elementas yra adresas, sutarta masyvus žymėti vadinamąja masyvo forma. Vienmačio masyvo forma yra tiesiog *MasyvoVardas[ ]*, o dvimačio – *Masyvo Vardas [ ][m]*; čia *m* yra masyvo stulpelių skaičius. Funkcijos kvietimo metu atitinkamas faktiškasis argumentas yra tiesiog *MasyvoVardas* abiem atvejais.

Taigi funkcijai, kuri per argumentų sąrašą gauna vienmatį masyvą, yra visiškai nesvarbus masyvo elementų skaičius: C kalbų kompiliatoriai netikrina, ar masyvo elemento indeksas išeina už

masyvui skirtos atminties srities ribų, ar ne. Garantuoti, kad masyvo elementų adresavimas bus tinkamas – paties programuotojo atsakomybė. Dvimačio masyvo atveju, kaip matyti, funkcijai būtina pranešti tikslų masyvo antrąjį matmenį (tiksliai tokį matmenį, koks nurodytas skelbiant masyvą kviečiančiojoje programoje). Taip yra dėl dvimačio masyvo konstravimo principo: tai yra vienmatis masyvas, kurio kiekvienas elementas savo ruožtu yra kitas vienmatis masyvas, t.y. dvimatis masyvas kompiuterio atmintyje saugomas kaip vienmatė duomenų seka. Programoje į dvimačio masyvo elementą kreipiantis dviem indeksais *i* ir *j*, kompiliatorius iš tos sekos reikiamą elementą atrenka pagal perskaičiuotą indeksą $k = i*m + j$, kur m yra masyvo antrasis matas. Taigi, jei kviečianti programa žinotų vienokį to paties dvimačio masyvo stulpelių kiekį, o kviečiamoji – kitokį, tai būtų atrenkami visai ne tie masyvo elementai kaip reikia. Analogiškai trimačiams masyvams jau reikėtų masyvo formoje nurodyti tikslų trečiąjį ir antrąjį matus, ir t.t.

Visas šias nuostatas iliustruosime 9 pavyzdžiu. Sakykim, turime dvimatį sveikųjų skaičių masyvą iš ne daugiau kaip 10 eilučių ir ne daugiau kaip 10 stulpelių. Reikia suskaičiuoti kiekvienos eilutės elementų vidurkius ir tarp jų surasti didžiausiąjį vidurkį bei užfiksuoti tos eilutės indeksą. Visi skaičiavimai atliekami funkcijoje, o įvedimo ir išvedimo operacijos – startinėje funkcijoje.

```
#include <iostream>
#include <iomanip>
using namespace std;
//
void avg( const int[ ][ 10 ], double[ ], double& vmax, int& imax, int n, int m );  // prototipas
//
int main( ){
        int n,m;                // masyvo tikrieji matai
        int x[ 10 ][ 10 ];    // pradinis masyvas
        double v[ 10 ], vmax; // vidurkių masyvas ir didžiausias vidurkis
        int imax;           // eilutės, turinčios didžiausią vidurkį, indeksas
                        // Įvedimas
        cout<<"Iveskite masyvo matus n , m: ";
        cin>>n>>m;
        cout<<"n, m = "<<n<<"  "<<m<<endl;
        cout<<"Iveskite \n";
        for( int i = 0; i < n; i++ ){
                        for( int j = 0; j < m; j++ ){
                                        cout<<"elementa "<<i+1<<"  "<<j+1<<" : ";
                                        cin>>x[ i ][ j ];
                        }
        }
//
        cout<<"\nIvestas masyvas:\n";
        for( int i = 0; i < n; i++ ){
                        cout<<"Eilute "<<i+1<<endl;
                        for( int j = 0; j < m; j++ )
                                        cout<<setw(5)<<x[i][j];
                        cout<<endl;
        }
//
```

```
//      Skaičiavimas
//
        avg( x, v, vmax, imax, n, m );
//
//      Rezultatų išvedimas
//
        cout<<"\nRezultatai\n";
        for( int i = 0; i <n; i++ )
                        cout<<i+1<<" eilutes vidurkis "<<setw( 10 )<<v[ i ]<<endl;
        cout<<"Didziausia vidurki  "<<setw( 10 )<<vmax<<"  turi "<<setw( 4 )
           <<imax+1<<"  eilute\n";
//
        system( "pause" );
        return 0;
}
//
void avg( const int a[ ][ 10 ], double b[ ], double& v, int& max, int n, int m ){
        //
        // Argumentai: a – duomenys, pradinis masyvas
        //              b – rezultatas, eilučių vidurkių masyvas
        //              v – rezultatas, didžiausias vidurkis
        //              max – rezultatas, eilutės su didžiausiu vidurkiu indeksas
        //              n,m – duomenys, tikrieji masyvu matai
        //
        for( int i = 0; i < n; i++ ){
                b[ i ]=0.;
                        for( int j = 0; j < m; j++ ){
                                        b [ i ] += a[ i ][ j ];
                        }
                        b[ i ] /= m;
        }
        //
        v  = b[ 0 ];
        max = 0;
        for( int i = 1; i < n; i++ ){
            if( v < b[ i ] ){
                        v = b[ i ];
                        max = i;
              }
       }
//
}
```

Kaip matyti, šioje programoje duomenys tarp funkcijų perduodami abiem galimais mechanizmais: pradinis masyvas ir vidurkių masyvas – adresu pagal numatymą, jokių nuorodų ženklų rodyti nereikia; didžiausias vidurkis ir atitinkamos eilutės indeksas – taip pat adresu, ties šiais skaliariniais duomenimis rašant nuorodų ženklus, o pradinio masyvo eilučių ir stulpelių

skaičiai – reikšme. Kadangi funkcijos duomenys – pradinis masyvas – atiduodamas funkcijai nesaugiu adreso mechanizmu, pageidautina šį argumentą skelbti konstantiniu.

5.6 FUNKCIJŲ PERKROVIMAS

Toks programavimo mechanizmas labai patogus, reikia užprogramuoti keletą funkcijų, atliekančių iš esmės tuos pačius dalykus ir tarpusavyje besiskiriančių tik argumentų sąrašu ar bent argumentų formatais. Pavyzdžiui, funkcijų perkrovimo mechanizmo neturinčioje C kalboje absoliutinio dydžio matematinė funkcija priklausomai nuo argumento ir grąžinamos reikšmės formato yra užprogramuota keliais variantais: *int iabs(int), long labs(long), float fabs(float)* ir t.t. C++ kalboje į šią funkciją kreipiamasi tik vienu bendriniu vardu *abs* teikiant bet kokio formato argumentą, o funkcija sugrąžina tokio pat formato rezultatą: funkcija yra perkrauta visiems reikiamiems formatams, o kompiliatorius iš visos tos puokštės funkcijų reikiamą programos variantą atsirenka pagal teikiamo argumento formatą.

10 pavyzdys. Perrašysime 1-ojo pavyzdžio programą. Dabar funkcija *stars* turės vieną vardą, bet po tuo vardu slėpsis trys skirtingi funkcijos variantai: funkcija be argumentų, su vienu argumentu (simboliu linijai formuoti; linijos ilgis fiksuotas), su dviem argumentais (ir simboliu, ir linijos ilgiu simboliais).

```
#include <iostream>
using namespace std;
//
void stars( );            // funkcijos prototipai
void stars( char );
void stars( char, int );
//
int main( ){
        stars( );          // kviečiamas pirmasis funkcijos variantas ...
        cout<<"Duomens tipas    Ilgis    Diapazonas\n";
        stars( '-' );      // ...antrasis ...
        cout<<"char          1B     -128 - +127"<<endl
            <<"short         2B     -32768 - +32767"<<endl
            <<"int           4B     ~-2e9 - ~+2e9"<<endl
            <<"long          4B     ~-2e9 - ~+2e9"<<endl;
        stars( '=', 60 );  // ...trečiasis
        //
        system( "pause" );
        return 0;
}
//  pirmasis funkcijos variantas
void stars( ){
        for( int i = 0; i < 60; i++ ) cout<<'*';
        cout<<endl;
}
//  antrasis funkcijos variantas
void stars( char c ){
```

```
        for( int i = 0; i < 60; i++ ) cout<<c;
        cout<<endl;
    }
    //  trečiasis funkcijos variantas
    void stars( char c, int n ){
        for( int i = 0; i < n; i++ ) cout<<c;
        cout<<endl;
}
```

5.7 REKURSINĖS FUNKCIJOS

Tai – funkcija besikreipiant pati į save. Akivaizdu, kad tokioje funkcijoje turi būti numatyta sąlyga išeiti iš vadinamosios rekursinės grandinės, nes antraip užprogramuosime begalinį, niekad nesibaigiantį ciklą. Jokių papildomų sintaksės taisyklių aprašant rekursines funkcijas nėra; užtenka tik kad funkcijos kūne yra kreipinys į ją pačią.

10 pavyzdys – standartinis pavyzdys aiškinant rekursiją – faktorialo skaičiavimas:
n! = n* (n-1) * (n-2) * ... * 1, o 0! = 1. Nors faktorialui saugoti naudosim ilgiausią galimą sveikųjų skaičių formatą – *unsigned long*, bandydami programą turėkit omenyje, kad net ir nedidelių skaičių faktorialai gali įgyti milžiniškas reikšmes, todėl programa teisingai skaičius tik faktorialus skaičiams iki 15. Didesniems skaičiams faktorialų reikšmės viršija ląstelės (*Windows* operacinėms sistemoms *unsigned long* formato ląstelei skiriami tik 4B) diapazoną - ~4.2e9.

```
#include <iostream>
using namespace std;
//
unsigned long  factorial( int );
//
int main( ){
        int number;
        //
        cout<<"Iveskite teigiama skaiciu\n";
        cin>>number;
        //
        cout<<"\nSkaiciaus "<<number<<" faktorialas yra "<<factorial( number )<<endl;
        //
        system( "pause" );
        return 0;
}
//
unsigned long  factorial( int n ){
        //
        // Argumentas: n – pradinis skaičius
        // Grąžinama reikšmė: rezultatas, faktorialas
        //
        if( n > 1 )
```

```
                return n*factorial( n-1 );  // rekursinis funkcijos kreipinys į save pačią
        else                                // išėjimas iš rekursinės grandinės
                return 1;
}
```

Kompiuterio atliekami veiksmai su šia funkcija, kai paleidę programą įvesime skaičių *5*, tokie:
1 žingsnis: iškviečiama funkcija su argumento reikšme *5* –
2 žingsnis: grąžinama reikšmė *5* (iškviečiama funkcija su argumento reikšme 4)* –
3 žingsnis: grąžinama reikšmė *4* (iškviečiama funkcija su argumento reikšme 3)* –
4 žingsnis: grąžinama reikšmė *3* (iškviečiama funkcija su argumento reikšme 2)* –
5 žingsnis: grąžinama reikšmė *2* (iškviečiama funkcija su argumento reikšme 1 – grąžinama reikšmė 1).*
Dabar visa rekursinė grandinė vykdoma atbulai:
1 žingsnis: reikšmė *1* perduodama 5-ajam funkcijos kvietimui: grąžinama reikšmė *2*1=2* –
2 žingsnis: reikšmė *2* perduodama 4-ajam funkcijos kvietimui: gražinama reikšmė *3*2=6* –
3 žingsnis: reikšmė *6* perduodama 3-ajam funkcijos kvietimui: grąžinama reikšmė *4*6=24* –
4 žingsnis: reikšmė *24* perduodama 2-ajam funkcijos kvietimui: grąžinama reikšmė *5*24=120*.

11 pavyzdys. Rekursiškai užrašysime realaus skaičiaus kėlimo sveikuoju laipsnio rodikliu funkciją. Programa vykdoma, kol programuotojas iš klaviatūros neįves *n*.

```
#include <iostream>
using namespace std;
//
double  power( double number, int index );
//
int main( ){
        double number;
        int index;
        char s;
        //
        do{
            cout<<"Iveskite pradini skaiciu ir laipsnio rodikli\n";
            cin>>number>>index;
            cout<<"\nSkaicius "<<number<<" pakeltas laipsnio rodikliu "<<index
                <<"  bus  "<<power( number, index )<<endl;
            cout<<"Ar testi skaiciavima? t/n\n";
            cin>>s;
        } while( s != 'n' );
        //
        system( "pause" );
        return 0;
}
//
double power( double x, int n ){
        //
```

```
        // Argumentai:      x – pradinis skaičius
        //                  n – laipsnio rodiklis
        // Grąžinama reikšmė: rezultatas, x pakeltas laipsniu n
        //
        if( n < 0 ) {
            x = 1./x;
            n = -n;
        }
        //
        if( n > 0 )
            return x*power( x, n-1 );  // rekursinis funkcijos kreipinys į save pačią
        else                           // išėjimas iš rekursinės grandinės: x^0=1
            return 1.;
}
```

5.8 KAI KURIOS KITOS FUNKCIJŲ GALIMYBĖS

Vidinės funkcijos

Kompiliavimo metu tokių funkcijų tekstas paprasčiausiai įkeliamas į programą vietoje kreipinio į jas. Trumpoms funkcijoms tai leidžia sutaupyti programos vykdymo laiko: juk kad parengtų funkcijos kvietimą, kompiliatorius turi paruošti duomenų buferį argumentų mainams, paruošti valdymo perdavimo funkcijai ir atgal kviečiančiai programai komandas, duomenų buferį grąžinamai reikšmei, ir t.t. – visa tai imlu laikui. Dabar gi funkcijos kviesti nebereikėtų. Funkcija paskelbiama vidine prie jos prototipo prirašius raktažodį *inline*. Pavyzdžiui, trumpą 5-ojo pavyzdžio funkciją temperatūrai iš Farenheito skalės į Celsijaus skalę perskaičiuoti galėtume paskelbti vidine taip:

```
inline double F_to_C( double );
```

Numatytieji argumentai

Šis mechanizmas kiek panašus į funkcijų perkrovimą: galima funkcijos prototipe užrašyti numatytąsias formaliųjų argumentų reikšmes ir vėliau kviesti funkciją faktiškųjų argumentų nenurodant – tada ir bus taikomos numatytosios jų reikšmės, arba nurodant kitokias dalies arba visų argumentų vertes. Jei nurodomos naujos vertės tik daliai argumentų, tai praleisti galima tik galinius argumentus, o argumentų iš sąrašo pradžios ar vidurio – ne. Visas šias galimybes demonstruoja 12 pavyzdys: perrašoma 10-ojo pavyzdžio programa.

```
#include <iostream>
using namespace std;
//
void stars( char = '*', int = 60 ); // prototipas su numatytosiomis argumentų reikšmėmis;
                                    // argumentų vardus nurodyti nebūtina
//
int main( ){
        stars( );          // argumentų reikšmės yra * ir 60
```

```
        cout<<"Duomens tipas    Ilgis    Diapazonas\n";
        stars( '-', 50 );   // argumentų reikšmės yra – ir 50
        cout<<"char          1B      -128 - +127"<<endl
            <<"short         2B     -32768 - +32767"<<endl
            <<"int           4B     ~-2e9 - ~+2e9"<<endl
            <<"long          4B     ~-2e9 - ~+2e9"<<endl;
        stars( '=' );       //  argumentų reikšmės yra = ir 60
        //stars( 100 );     // klaida: negalima praleisti pradinių ar vidurinių argumentų
        //
        system( "pause" );
        return 0;
}
//
void stars( char c, int n ){
        for( int i = 0; i < n; i++ ) cout<<c;
        cout<<endl;
}
```

Funkcijos gražinama reikšmė-nuoroda

Iš funkcijos galima grąžinti ne tik įprastą reikšmę (aritmetinę, loginę, tekstinę), bet nuorodą į atminties ląstelės adresą. Nuorodos (primename: *formatas&*. O paskelbtam kintamajam *&kintamasis* – tai jau kintamojo *kintamasis* ląstelės adresas) nėra savarankiški programos objektai: tai tėra nuoroda į kažkurią atminties ląstelę, todėl ir argumentų perdavimo adresu atveju sakėme, kad atitinkamų formaliojo argumento-nuorodos ir faktiškojo argumento vardai yra tų pat atminties ląstelių sinoniminiai vardai.

Esminė nauja galimybė grąžinant iš funkcijos nuorodą – kad dabar kreipinys į funkciją gali būti ir kairėje prieskyros operatoriaus pusėje; tai svarbu, pavyzdžiui, objektiniame programavime, perkraunant operatorius.

Reikia įsidėmėti: negalima grąžinti nuorodos į lokalų funkcijoje paskelbtą kintamąjį: juk toks kintamasis baigs gyvuoti sulig savo matomumo sritimi – funkcija, ir turėsim nuorodą į neaišku ką. Galima grąžinti tik nuorodą į globalųjį kintamąjį (žr. tolesnį skyrių) arba per argumentų sąrašą ateinantį kintamąjį.

13 pavyzdys: turime realių skaičių masyvą, kurio elementų skaičius neviršija 100. Vietoje jo mažiausio elemento reikia įrašyti reikšmę *100.* . Dabar jau taip pakeistame masyve vėl išrenkamas mažiausias elementas ir vietoje jo turime įrašyti reikšmę *200.* .

```
#include <iostream>
#include <iomanip>
using namespace std;
//
double& lowest( double[ ],  int n );  // prototipas
//
int main( ){
        int n;                  // masyvo matas
        double x[ 100 ];    // pradinis masyvas
            // Įvedimas
        cout<<"Iveskite masyvo mata:   ";
```

```
        cin>>n;
        cout<<"n = "<<n<<endl;
        cout<<"Iveskite \n";
        for( int i = 0; i < n; i++ ){
                cout<<"masyvo elementa "<<i+1<<" :   ";
                cin>>x[ i ];
        }
//
        cout<<"\nIvestas masyvas:\n";
        for( int i = 0; i < n; i++ )
                cout<<setw( 10 )<<x[ i ];
        cout<<endl;
//
//          Masyvo keitimas
//
        lowest( x, n ) = 100.;  // kreipinys į funkciją – kairėje prieskyros operatoriaus pusėje!;
        lowest( x, n ) = 200.;  // šis kreipinys ir mažiausias masyvo elementas faktiškai yra tas
                                // pat
//
//            Rezultatų išvedimas
//
        cout<<"\nRezultatai: pakeistas masyvas\n";
        for( int i = 0; i <n; i++ )
                cout<<setw( 10 )<<x[ i ];
        cout<<endl;
//
        system( "pause" );
        return 0;
}
//
double& lowest( double x[ ], int n ){
        //
        // Argumentai: x – duomenys, pradinis masyvas
        //                n – duomenys, masyvo matas
        //Grąžinama reikšmė: nuoroda į mažiausią elementą
        //
        int imin = 0;  // prielaida: mažiausio elemento indekso vertė
        for( int i = 1; i < n; i++ )
            if( x[ imin ] > x[ i ] )
                imin = i;
        return x[ imin ]; // grąžinama nuoroda į mažiausią elementą: imin-tojo elemento
                          // adresas
}
```

Kaip matome, dabar kreipinys į funkciją atsiduria kairėje prieskyros operatoriaus pusėje. Taip ir turi būti: juk funkcija grąžina nuorodą į surastąjį mažiausiąjį masyvo elementą, ir dabar galime reikiamai pakeisti to elemento reikšmę.