10-11 PASKAITOS

Turinys: Funkcijos. Argumentų sąrašas. Grąžinama reikšmė. Argumentų perdavimas reikšme ir adresu.
Masyvai-funkcijų argumentai.
Funkcijų perkrovimas.
Rekursinės funkcijos.
Vidinės funkcijos. Numatytieji argumentai.

Funkcijos

Tai – įvardinta operatorių grupė. Prasmė: konceptualizuoti programos struktūrą – o tai yra pamatinis procedūrinio programavimo principas. Be to, leidžia sutrumpinti programos tekstą, jei pasikartojančios teksto dalys forminamos kaip funkcijos.

Pavyzdys: paprastos nepilnos lentelės, į kurią surašomos duomenų tipų charakteristikos, spausdinimas.

```
#include <iostream>
using namespace std;
//
void stars( );  // pastaba 1
//
int main( ){
        stars( );  // 2
        cout<<"Duomens tipas          Diapazonas"<<endl;
        stars( );  // 2
        cout<<"char                  -128   -   +127"<<endl<<
              "short                 -32768   -   +32767"<<endl<<
              "int                   ~-2e9    -   ~+2e9"<<endl<<
              "long                  ~2e9     -   ~+2e9"<<endl;
        stars( );  // 2
        return 0;
}
        // Funkcija
void stars( ){    //  3
        for( int i = 0; i < 50; i++ ) cout<<'*';
        cout<<endl;
}  // 4
```

Pastabos:

1. Funkcijas kaip kintamuosius būtina paskelbti. Galimi būdai: kaip čia; arba apibrėžti prieš pirmąjį kvietimą. Jei taikomas pirmasis būdas: nurodyti grąžinamos reikšmės tipą, o jei reikšmės negrąžina – *void*; skliaustuose įrašyti argumentų sąrašą, o jei argumentų nėra – įrašyti *void* arba nerašyti nieko. Toks skelbimas – "funkcijos prototipas", arba "funkcijos parašas". Prasmė: nurodo, kaip kompiliatorius turi kviesti funkciją, kurios kūnas bus aptiktas vėliau.

2. Valdymas atiduodamas funkcijai, vykdomi visi funkcijos operatoriai, o juos baigus – pirmam operatoriui kviečiančioje programoje po kreipiniu. Kreipinio forma: taip kviesti galima tik *void* funkcijas.
3. Tai – funkcijos apibrėžimas. Susideda iš antraštės ir kūno. Antraštė sutampa su prototipu.
4. *void* funkcijai *return* nebūtinas.

Pastaba dėl bibliotekinių funkcijų: jų prototipai yra antraštiniuose failuose (pavyzdžiui, funkcijai *getche* – faile *conio.h*). Apibrėžimai yra jau sukompiliuoti ir saugomi bibliotekiniuose failuose; saistymo metu jie prijungiami prie programos objektinio kodo.

Argumentai

– reikalingi, kai funkcijos darbas priklauso nuo tam tikrų parametrų.
Pavyzdys: perrašoma ta pati programa, tik dabar funkcija turi turėt galimybę spausdinti bet kokius simbolius ir bet kokį jų skaičių.

```
#include <iostream>
using namespace std;
//
void varchars( char, int );  // pastaba 1
//
int main( ){
        varchars( '-', 50  );  // 2
        cout<<"Duomens tipas          Diapazonas"<<endl;
        varchars( '-', 30 );  // 2
        cout<<"char                   -128   -   +127"<<endl<<
              "short                  -32768   -   +32767"<<endl<<
              "int                    ~-2e9    -   ~+2e9"<<endl<<
              "long                   ~2e9     -   ~+2e9"<<endl;
        varchars( '=', 50 );  // 2
        return 0;
}
        // Funkcija
void varchars( char c, int n ){    //  3
        for( int i = 0; i < n; i++ ) cout<<c;
        cout<<endl;
}
```

Pastabos:
1. Skelbiant argumentus, galima apsiriboti tik jų tipais. Jei būtų vaizdžiau, galima nurodyti ir jų formalius vardus.
2. Taip nurodomi faktiškieji argumentai. Čia tie argumentai – konstantos, o gali būti ir kintamieji. Formalieji ir faktiškieji argumentų sąrašai privalo sutapti pagal kiekį, tipą ir prasmę. Patys argumentų vardai (jei faktiškieji argumentai – kintamieji) gali skirtis.
3. Formaliųjų argumentų sąraše funkcijoje jau būtina nurodyti jų vardus.

Pavyzdys: ta pati programa, tik faktiškieji argumentai – kintamieji. Faktiškųjų ir atitinkamų formaliųjų argumentų vardai čia skiriasi.

```
#include <iostream>
using namespace std;
//
void varchars( char, int );  // pastaba 1
//
int main( ){
        char ch;
        int m;
        cout<<"Iveskite simboli ir ju skaiciu";
        cin>>ch>>m;
        varchars( ch, m  );  // 2
        cout<<"Duomens tipas            Diapazonas"<<endl;
        varchars( ch, m );  // 2
        cout<<"char                  -128   -   +127"<<endl<<
                "short               -32768   -   +32767"<<endl<<
                "int                 ~-2e9    -   ~+2e9"<<endl<<
                "long                ~2e9     -   ~+2e9"<<endl;
        varchars( ch, m );  // 2
        return 0;
}
        // Funkcija
void varchars( char c, int n ){    //  3
        for( int i = 0; i < n; i++ ) cout<<c;
        cout<<endl;
}
```

Grąžinama reikšmė

Tai – funkcijos darbo rezultatas.
Pavyzdys: temperatūros perskaičiavimas iš Farenheito į Celsijaus skalę.

```
#include <iostream>
using namespace std;
//
double f_to_c( double );  // pastaba 1
//
int main( ){
        double f, c;
        cout<<"Iveskite temperatura Farenheito laipsniais";
        cin>>f;
        c = f_to_c( f );  // 2
        cout<<"Farenheito temperatura: "<<f<<"Celsijaus temperatura: "<<
                c<<endl;
        return 0;
}
```

```
// Funkcija
double f_to_c( double fd ){
        // Argumentai: fd – Farenheito temperatura
        //Grazinama reiksme- Celsijaus temperatura
        double cd = ( fd - 32 )*5./9.;
        return cd;   // 3
}
```

Pastabos:

1. Prototipe nurodomas grąžinamos reikšmės tipas. Jei tipas nenurodomas, pagal nutylėjimą laikoma, kad bus grąžinama *int* reikšmė. Nerekomenduojama naudotis tokia galimybe – nevaizdu.
2. Kai funkcija grąžina kokią nors reikšmę, ją kviesti reikia kokiame nors vykdomajame operatoriuje (čia kviečiama prieskyros operatoriuje).
3. Taip grąžinama reikšmė perduodama kviečiančiajai programai. Taigi, taip grąžinti galima tik vieną rezultatą. Jei rezultatų daugiau – reikia taikyti kitokį duomenų mainų mechanizmą (žr. žemiau).

Aukščiau parašytą programą galima sutrumpinti – atsisakyti kelių nebūtinų kintamųjų:

```
#include <iostream>
using namespace std;
//
double f_to_c( double );
//
int main( ){
        double f;
        cout<<"Iveskite temperatura Farenheito laipsniais";
        cin>>f;
        cout<<"Farenheito temperatura: "<<f<<"Celsijaus temperatura: "<<
                f_to_c( f )<<endl;
        return 0;
}
        // Funkcija
double f_to_c( double fd ){
        // Argumentai: fd – Farenheito temperatura
        //Grazinama reiksme- Celsijaus temperatura
        return ( ( fd-32 )*5./9. );
}
```

Argumentų perdavimo mechanizmai

Du: reikšme (mechanizmas pagal numatymą; taip argumentai buvo perduodami aukščiau parašytuose pavyzdžiuose), ir adresu.
Pirmu 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 *struct* duomenis – tada jų kopijoms teks skirti irgi daug atminties. Dėl šios priežasties masyvai pagal numatymą perduodami tik adresu, o ne reikšme.
Antru 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 pakis 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. Supaprastintai: dabar formalieji ir faktiškieji vardai tėra tos pačios atminties ląstelės (-ių) skirtingi vardai, sinonimai. Kviečianti programa naudoja vieną vardą, o kviečiamoji – kitą.

Pavyzdys: *float* formato skaičiuje išskirti sveikąją ir trupmeninę dalis; abi – *float* formato. Taigi, funkcijos darbo rezultatai – du; teks taikyti rezultatų perdavimo per argumentų sąrašą mechanizmą adresu.

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

Pastabos:

1. Argumentai, po kurių formatų yra & ženklai, perduodami adresu. Pirmasis argumentas (jo vardas nenurodytas) perduodamas reikšme.
2. Kvietimo forma abiems mechanizmams vienoda. Kvietimas čia toks, nes funkcija *void*.
3. Funkcijos apibrėžime adresu perduodamus argumentus irgi būtina pažymėti &.

Adresu galima perduoti ne tik argumentus, bet ir funkcijos grąžinamą reikšmę. Tada funkcijos kvietimas galimas ir kairėje prieskyros operatoriaus pusėje; tai bus reikalinga operacijų perkrovime.
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 );  // raktazodis const antrajam argumentui
int main( ){
        int mk1 = 1, mk2 = 2;
        f1( mk1, mk2 );
        return 0;
}
void f1( int& fk1, const int& fk2 ){  // raktazodi pakartoti
        fk1 = 10;  // gerai, keisti galima - nekonstantinis
        fk2 = 20; // kompiliavimo klaida, argumentas apsaugotas
}
. . .
```

Masyvai-funkcijų argumentai

Pavyzdys: dvimačio masyvo, turinčio ne daugiau kaip 10 eilučių ir 10 stulpelių, eilučių vidurkių skaičiavimas.

```
#include <iostream>
#include <iomanip>
using namespace std;
//
const int MS = 10;
//
void avg( double[ ][ MS ],double[ ],int n,int m );  // pastabos 1,4
//
int main( ){
        int n,m;
        double x[ MS ][ MS ], v[ MS ];  // 2
                // Ivedimas
        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<<"Ivestas masyvas:\n";
        for( 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;
        }
//
        avg( x, v, n, m );  // 3
//
        cout<<"Rezultatai\n";
        for( i=0; i<n; i++ ){
                cout<<i+1<<" eilutes vidurkis "<<setw(5)<<v[ i ]<<endl;
        }
        return 0;
}
//
void avg( double a[ ][ MS ], double b[ ], int n, int m ){  // 1,4
        //
        // Argumentai: a – pradinis masyvas
        //              b – rezultato masyvas, eiluciu vidurkiai
        //              n,m – 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;
        }
}
```

Pastabos:

1. Prototipe būtina nurodyti masyvo formą ir antrąjį dvimačio masyvo matą; pirmasis matas nesvarbus (nesvarbus jis ir vienmačiam masyvui). Analogiškai trimačiam masyvui reikėtų nurodyti trečiąjį ir antrąjį matus, ir t.t. Tas pat galioja ir masyvų aprašui funkcijos apibrėžime. Taip yra dėl dviejų priežasčių: kompiliatorius netikrina, ar kreipiantis į masyvo elementą, kreipinys neišeina už masyvo ribų; antra, daugiamačiai masyvai atmintyje saugomi kaip vienmatės sekos, iš kurių reikiamas elementas atrenkamas taikant perskaičiuotą indeksą. Pavyzdžiui, kreipiantis į dvimačio masyvo elementą indeksais *i* ir *j*, faktiškai taikomas perskaičiuotas indeksas $k = i*M + j$, kur M

yra stulpelių skaičius. 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.

2. Šioje vietoje masyvams skiriama atmintis. Rekomenduojamas toks programavimo stilius – apibrėžti masyvus konstantiniais kintamaisiais.
3. Kreipinyje į funkciją masyvo forma nenurodoma.
4. Nors & prie argumentų-masyvų niekur neparašyti, masyvai automatiškai funkcijoms perduodami tik adresu.

Funkcijų perkrovimas

– kai vienu vardu yra kelios funkcijos, besiskiriančios argumentų kiekiu ar bent argumentų tipais. Pagal argumentų sąrašą kompiliatorius atrenka reikiamą funkciją.
Pavyzdys: perrašoma pati pirmoji šio skyriaus programa duomenų charakteristikų lentelei braižyti. Dabar funkcija – trijų variantų: be argumentų, su vienu argumentu (simboliu linijos formavimui; linijos ilgis fiksuotas), su dviem argumentais (ir simboliu, ir linijos ilgiu).

```
#include <iostream>
using namespace std;
//
void varchars( );  // pastaba 1
void varchars( char ); // 1
void varchars( char, int );  // 1
//
int main( ){
        varchars(  );
        cout<<"Duomens tipas          Diapazonas"<<endl;
        varchars( '-', 30 );  // 2
        cout<<"char                  -128   -   +127"<<endl<<
              "short                 -32768   -   +32767"<<endl<<
              "int                   ~-2e9    -   ~+2e9"<<endl<<
              "long                  ~2e9     -   ~+2e9"<<endl;
        varchars( '=' );  // 3
        return 0;
}
        // Funkcijos
void varchars(  ){    // 4
        for( int i = 0; i < 50; i++ ) cout<<'*';
        cout<<endl;
}
void varchars( char c ){    // 4
        for( int i = 0; i < 50; i++ ) cout<<c;
        cout<<endl;
}
void varchars( char c, int n ){    // 4
        for( int i = 0; i < n; i++ ) cout<<c;
        cout<<endl;
}
```

Pastabos:

1. Visi funkcijos prototipai, besiskiriantys argumentų sąrašais.
2. Kviečiamas 3-iasis funkcijos variantas.
3. Kviečiamas 2-asis funkcijos variantas.
4. Visi trys funkcijų apibrėžimai.

Toks mechanizmas žymiai supaprastina funkcijų šeimų vartojimą. Pavyzdžiui, C kalboje perkrovimo nėra, todėl modulis realizuojamas atskiromis funkcijomis *int abs(int), long labs (long), double fabs(double), … .* C++ kalboje į visas funkcijas galima kreiptis vienu vardu: *abs( ).*

Rekursinės funkcijos

Tai – funkcijos, besikreipiančios į save. Standartinis pavyzdys rekursijos aiškinimui – faktorialo skaičiavimas: n! = n*(n-1)!, (n-1)! = (n-1)*(n-2)!, …

```
#include <iostream>
using namespace std;
//
unsigned long  factorial( int );
//
int main( ){
        int number;
        unsigned long f; // faktorialo reiksme
        cout<<"Iveskite skaiciu";
        cin>>number;
        f = factorial( number );
        cout<<"Skaiciaus "<<number<<" faktorialas yra "<<f<<endl;
        return 0;
}
        // Funkcija
unsigned long  factorial( int n );
        //
        // Argumentas: n - skaicius
        // Grazinama reiksme: faktorialas
        //
        if( n > 1 )
                return n*factorial( n-1 );  // pastaba 1
        else
                return 1;
}
```

Pastaba:

1. Čia yra funkcijos kreipinys į save pačią, tik su sumažinta argumento reikšme. Visada rekursinė funkcija privalo turėti išėjimą iš rekursinės grandinės – kaip čia po operatoriaus *else*: funkcija daugiau nebekviečiama, tik grąžinama konkreti reikšmė.

Kompiuterio atmintyje rekursinės funkcijos kūnas saugomas vieną kartą, o visų kvietimų argumentai ir grąžinamos reikšmės – atskirai. Jei rekursinė grandinė ilga, ji gali pareikalauti didelių kompiuterių resursų.

Vidinės funkcijos

Funkcijos kvietimui parengti kompiliatorius turi eikvoti tam tikrą laiką: parengiamos komandos perėjimui į funkciją, komandos argumentų reikšmių perdavimui į stekus, komandas jų išėmimui iš stekų ir pan. Todėl trumpoms funkcijoms efektyviau gali būti tiesiog į programos tekstą įrašyti visą funkcijos tekstą. Taigi dabar programuotojas galėtų rašyti funkcijas, o kompiliatorius, parengdamas galutinį pradinį kodo failą, vietoje kreipinių į funkcijas įrašytų jų tekstus. Kad funkcija taptų vidine, ją reikia skelbti su raktažodžiu *inline*:

```
. . .
inline double f_to_c( double );
. . .
```

Numatytieji argumentai

Mechanizmas, kiek panašus į funkcijų perkrovimo mechanizmą. Funkciją galima kviesti visai nenurodant argumentų reikšmių, tačiau tam prototipe būtina įrašyti "argumentų reikšmes pagal numatymą".
Vėl tas pats pirmasis pavyzdys:

```
#include <iostream>
using namespace std;
//
void varchars( char = '*', int = 50 );  // pastaba 1
//
int main( ){
        varchars(  ); // 2
        cout<<"Duomens tipas          Diapazonas"<<endl;
        varchars( '-' );  // 3
        cout<<"char                   -128   -   +127"<<endl<<
              "short                  -32768   -   +32767"<<endl<<
              "int                    ~-2e9    -   ~+2e9"<<endl<<
              "long                   ~2e9     -   ~+2e9"<<endl;
        varchars( '=', 60 );  // 4
        return 0;
}
        // Funkcija
void varchars( char c, int n ){    //  4
        for( int i = 0; i < n; i++ ) cout<<c;
        cout<<endl;
}
```

Pastabos:

1. Taip įrašomi argumentai pagal numatymą.
2. Taikomos argumentų reikšmės * ir 50.
3. Taikomos argumentų reikšmės – ir 50.
4. Taikomos argumentų reikšmės = ir 60.

Svarbu: praleisti galima tik argumentus iš galinės pusės. Negalimas kvietimas *varchars(60);* .