4 – 5 PASKAITOS

Turinys: Operacijų perkrovimas ir tipų pertvarkymas
Unarinių operacijų perkrovimas. Binarinių operacijų perkrovimas: aritmetinės, santykio ir prieskyros operacijos.
Duomenų tipų pertvarkymas.

Bendros žinios

Operacijų perkrovimo prasmė – supaprastinti ir padaryti intuityviai labiau suprantamą programą. Pavyzdžiui, jau nagrinėtos angliškų matų klasės *Distance* dviejų objektų sudėčiai, kaip buvo parodyta pirmajame semestre, reikia arba taip kviesti nieko negrąžinantį sudėties metodą

d3.addDistances(d1, d2);

arba taip – *Distance* duomenį gražinantį sudėties metodą

d3 = d1.addDistances(d2);

Tai – nevaizdu. Vaizdesnė forma

d3 = d1 + d2;

ir pasiekiama operacijos "+" perkrovimu *Distance* tipo duomenims; o jei operacija apjungtų aritmetinius duomenis – būtų atliekama įprastinė sudėtis.
Faktiškai operacijos perkrovimas tėra tik kitokia metodo kvietimo forma; tas metodas turi turėti raktažodį *operator* ir iškart po jo einantį operacijos ženklą, pavyzdžiui,

void operator-( ){ . . . }

Perkraunant unarines operacijas, šis metodas argumentų neturi, o binarines – turi turėti vieną argumentą.

Unarinių operacijų perkrovimas

Unarinės operacijos turi tik vieną operandą, pavyzdžiui, inkremento ar dekremento operacijos.
Pavyzdys: nagrinėsim jau praėjusiame semestre aptartą skaitiklio klasę *Counter*. Jos lauko *count* reikšmės didinimui perkrausime priešinkremento operaciją.

```
#include <iostream>
using namespace std;
//
class Counter{
        private:
                unsigned int count;
```

```
        public:
                Counter( ): count( 0 ) { }      // konstruktorius be argumentų
                unsigned int getCount( ) {    // metodas lauko reikšmės grąžinimui
                        return count;
                }
                void operator++( ){            // "++" operacijos perkrovimas
                        ++count;
                }
};
//
int main( ){
        Counter c1, c2;
        cout<<"c1: "<<c1.getCount( )<<endl;
        cout<<"c2: "<<c2.getCount( )<<endl;
                ++c1;
                ++c2;
                ++c2;
                cout<<"c1: "<<c1.getCount( )<<endl;
                cout<<"c2: "<<c2.getCount( )<<endl;
                        return 0;
}
```

Programos rezultatai:

```
c1: 0
c2: 0
c1: 1
c2: 2
```

Kaip kompiliatorius sužino, kad operacijai "++" reikia kviesti parašytąjį metodą? Pagal duomens prieš operacijos ženklą tipą. Jei operacijos ženklas būtų prieš aritmetinį vidinį duomenį (pavyzdžiui, *int* tipo) – būtų atliekama įprastinė priešinkremento operacija. Kadangi programoje operacija eina prieš vartotojo apibrėžto tipo *Counter* duomenį *c1* ar *c2*, tai atitinkamoje klasėje ieškoma su operacija susieto metodo.
Taip parašius perkrovimo metodą, negalima operacijos rezultato priskirti kitam objektui, pavyzdžiui

```
c2 = ++c1;
```

- juk metodas yra *void* ir nieko negrąžina. Norint, kad perkrautą operaciją būtų galima taikyti ir taip, reikėtų kiek pakeisti patį metodą (kad šis grąžintų tokio pat tipo duomenį, kaip ir prieskyros kairėje esantis duomuo):

```
. . .
Counter operator++( ){
        ++count;
        Counter temp;
        temp.count = count;
        return temp;
}
```

```
. . .
int main( ){
        Counter c1,c2;
        ++c1;
        c2 = ++c1;
        cout<<"c1: "<<c1.getCount( )<<endl;
        cout<<"c2: "<<c2.getCount( )<<endl;
        return 0;
}
```

Rezultatai:

```
        c1: 2
        c2: 2
```

Dar paprasčiau tai galima padaryti kiek modifikavus klasę – įvedus metodą-konstruktorių su argumentais. Kartu šiame pavyzdyje perkraunama ir podekremento operacija:

```
#include <iostream>
using namespace std;
//
class Counter{
        private:
                unsigned int count;
        public:
                Counter( ): count( 0 ) { }
                Counter( int c ): count( c ){ }  // konstruktorius su argumentu
                unsigned int getCount( ) {
                        return count;
                }
                Counter operator++( ){        // priešinkremento operacija
                        return Counter( ++count ); // grąžinamas sukonstruotas
                }                                  // objektas, kurio count reikšmė
                                                   // padidinta priešinkrementu
                Counter operator++( int ){   //poinkremento operacija
                        return Counter( count++ );
                }
};
//
int main( ){
        Counter c1, c2;
        cout<<"c1: "<<c1.getCount( )<<endl;
        cout<<"c2: "<<c2.getCount( )<<endl;
                ++c1;
                c2 = ++c1;
                c2 =  c1++;
                cout<<"c1: "<<c1.getCount( )<<endl;
                cout<<"c2: "<<c2.getCount( )<<endl;
                        return 0;
}
```

Rezultatai:

c1: 0
c2: 0
c1: 3
c2: 2

Podekremento metode argumentų sąrašo vietoje nurodytas raktažodis *int* nėra joks argumentas – tai tik sutartinis C++ sintaksės dalykas operacijos po- formai žymėti.

Binarinių operacijų perkrovimas

Nagrinėsim, kaip perkrauti aritmetines, santykio ir prieskyros operacijas.

Aritmetinių operacijų perkrovimas
Pavyzdys: angliškų matų klasė *Distance*, jau nagrinėta praėjusiame semestre. Perkrausime sudėties operaciją "+", kad du *Distance* objektus būtų galima taip sudėti:

d3 = d1 + d2;

Pirmiausia priminsim originalią anksčiau rašytą programą ir kelių *Distance* tipo objektų sudėtį išreikštai kviečiant atitinkamus metodus:

```
#include <iostream>
using namespace std;
//
class Distance{
        private:
                int feet;
                float inches;
        public:
                Distance( ): feet( 0 ), inches( 0.f ){ }
                Distance( int f, float in ): feet( f ), inches( in ){ }
                void showDistance( ){
                        cout<<"Distance: "<<feet<<"  "<<inches<<endl;
                }
                Distance addDistances( Distance d2 ) const {
                        Distance temp;
                        temp.feet = feet + d2.feet;
                        temp.inches = inches + d2.inches;
                        if( temp.inches >= 12.f ) {
                                temp.inches -=12.f;
                                temp.feet++;
                        }
                        return temp;
                }
//
int main( ){
        Distance d3, d4;
```

```
        Distance d1( 1, 1.f );
        Distance d2( 2, 2.f );
        d3 = d1.addDistances( d2 );
        d4 = d3.addDistances( d3 );
        d1.showDistance( );
        d2.showDistance( );
        d3.showDistance( );
        d4.showDistance( );
        return 0;
}
```

Rezultatai:

```
Distance: 1 1
Distance: 2 2
Distance: 3 3
Distance: 6 6
```

Distance objektų sumavimo metodo – sudėties operacijos perkrovimas:

```
#include <iostream>
using namespace std;
//
class Distance{
        private:
                int feet;
                float inches;
        public:
                Distance( ): feet( 0 ), inches( 0.f ){ }
                Distance( int f, float in ): feet( f ), inches( in ){ }
                void showDistance( ){
                        cout<<"Distance: "<<feet<<"  "<<inches<<endl;
                }
                Distance operator+ ( Distance d2 ) const {
                        Distance temp;
                        temp.feet = feet + d2.feet;             // pastaba
                        temp.inches = inches + d2.inches;   // pastaba
                        if( temp.inches >= 12.f ) {
                                temp.inches -=12.f;
                                temp.feet++;
                        }
                        return temp;
                }
//
int main( ){
        Distance d3, d4;
        Distance d1( 1, 1.f );
        Distance d2( 2, 2.f );
        d3 = d1 + d2;           // pastaba
        d4 = d1 + d2 + d3;
```

```
        d1.showDistance( );
        d2.showDistance( );
        d3.showDistance( );
        d4.showDistance( );
        return 0;
}
```

Programos rezultatai – tokie pat.
Pastaba: kaip suprasti, kas yra darbinis objektas metode "*operator+*" (t.y. kurio objekto laukas išskiriamas kreipiantis tiesiog į laukus *feet* ir *inches* be jokio objekto vardo)? C++ sintaksėje sutarta, kad objektas iš perkrautosios operacijos kairiosios pusės (t.y. *d1* iš reiškinio *d1 + d2*) laikomas darbiniu objektu, o argumento objektu laikomas objektas iš perkrautosios operacijos dešinės pusės (t.y. *d2*). Todėl, tarkim, reiškinyje *d3 + d1* darbinis objektas būtų *d3*, o metode *d2* formaliuoju vardu vadinamas objektas faktiškai būtų *d1*.
Sudėties metodą galima perrašyti ir kitokia forma, nenaudojant tarpinio objekto, o grąžinant konstruktoriumi su argumentais formuojamą bevardį objektą:

```
. . .
Distance operator+ ( Distance d2 ) const {
        int f = feet + d2.feet;
        float in = inches + d2.inches;
        if( in >= 12.f ) {
                in -=12.f;
                f++;
        }
        return Distance( f, in );
}
. . .
```

Likusi programos dalis liktų nepakitusi.

Taip parašius programą, ženklas "+", jei jis jungtų du *Distance* tipo objektus, būtų interpretuojamas kaip metodo *operator+* kvietimas, o jei jis jungtų du aritmetinius duomenis – būtų interpretuojamas kaip paprastas sudėties operatorius.

Santykio operacijų perkrovimas
Perkrausime operatorių < dviejų anksčiau nagrinėtos klasės *Distance* objektų palyginimui. Operatoriaus neperkrovus, dviejų *Distance* objektų taip lyginti nebūtų galima:

```
… d1 < d2 …
```

Kadangi klasės kodas beveik tas pat, čia pateikiami tik reikiami teksto pakeitimai. Klasėje *Distance*:

```
…
bool operator< ( Distance ) const;  // prototipas
…
```

Metodas (už klasės ribų):

```
bool Distance::operator< ( Distance d2 ) const{
        float temp1 = feet + inches/12.f;
        float temp2 = d2.feet + d2.inches/12.f;
        if( temp1 < temp2 )
                return true;
        else
                return false;
}
```

main funkcijoje:

```
int main( ){
        …
        if( d1 < d2 )
                cout<<"d1 < d2"<<endl;
        else
                cout<<"d1 >= d2"<<endl;
        …
}
```

Prieskyros operacijų perkrovimas

Perkrausime operatorių += *Distance* objektų laukų sumavimui į pirmojo objekto laukus. Taip pat pateikiami tik reikiami tekstų pakeitimai ankstesnei programai.

Klasėje *Distance*:

```
…
void operator+= ( Distance );  // prototipas
…
```

Metodas už klasės ribų:

```
void Distance::operator+= ( Distance d2 ){
        feet += d2.feet;
        inches+= d2.inches;
        if( inches >= 12.f ){
                inches-=12.f;
                feet++;
        }
}
```

main funkcijoje:

```
…
int main( ){
        …
        d1 += d2;     // t. y. d1 = d1 + d2
        …
}
```

Duomenų tipų pertvarkymas

Pasitelkiamas, pavyzdžiui, prieskyros operatoriuje *k1 = k2;* kai prieskyra apjungti operandai yra skirtingų tipų:

```
int i1;
float f2;
i1 = f2;
```

Tai – neišreikštinis tipų pertvarkymas. Jam atlikti kompiliatorius kompiliavimo metu automatiškai kviečia specialias vidines funkcijas. Tas pat funkcijas galima kviesti ir išreikštiniu būdu:

```
i1 = static_cast< int >( f2 );  arba
i1 = int ( f2 );
```

Kurį šių būdų rinktis – tik programavimo stiliaus dalykas.
Klasių objektams tokie pertvarkymai iš vienos klasės tipo į kitos klasės tipą arba į vidinius kalbos tipus be programuotojo įsikišimo neįmanomi. Pertvarkymui iš vidinių tipų į klasės tipą, t. y. vartotojo sugalvotą tipą tenka rašyti vadinamąjį pertvarkymų konstruktorių, o pertvarkymui iš klasės tipo į vidinius tipus – rašyti pertvarkymo funkciją panašiai kaip perkraunant operatorius.
Pavyzdžiai: pertvarkyti atstumą metrais į *Distance* tipo atstumą, o *Distance* tipu teikiamą atstumą – į metrinį.

```
#include <iostream>
using namespace std;
//
const float MTF = 3.28083f;     //daugiklis m -> pėdos (dešimtainės)
//
class Distance{
        private:
                int feet;
                float inches;
        public:
                Distance( int f, float in ): feet( f ), inches( in ) { }
                Distance( float m ){            //pertvarkantysis konstruktorius
                        float floatfeet = MTF * m;
                        feet = int( floatfeet );
                        inches = ( floatfeet - feet )*12.f;
                }
                void getDistance(){
                        cout<<"Enter feet and inches:";
                        cin>>feet>>inches;
                }
                void showDistance() const {
                        cout<<feet<<"  "<<inches<<endl;
                }
```

```
                operator float(){      // pertvarkymo funkcija; jokio void ar tipo!
                        float partfeet = inches/12.f;
                        float fullfeet = partfeet + static_cast<float>( feet );
                        return fullfeet/MTF;
                }
};
//
int main( ){
        float m1, m2;
        Distance d1 = 1.23f;       // pertvarkantysis konstruktorius - 1
        cout<<"d1 in feets: ";
        d1.showDistance();
                Distance d2( 2.46f );       // tas pat - 2
                cout<<"d2 in feets: ";
                d2.showDistance();
                        m1 = static_cast<float>( d1 );   // atvirkščias išreikštinis
                                                         // pertvarkymas - 1
                        cout<<"d1 in meters: "<<m1<<endl;
                                m2 = d2;                                  // tas pat - 2
                                cout<<"d2 in meters: "<<m2<<endl;
                                        return 0;
}
```

Panašiai gali būti pertvarkomi ir objektų tipai iš vienos klasės tipo į kitos klasės tipą.