## 15-16 PASKAITOS

**Turinys:**
**Programuotojo duomenų tipai: struktūros, sąrašai *enum*, klasės.**
**Klasės: konstruktoriai, jų perkrovimas;**
**metodai, metodai ne klasės kūne;**
**objektai-funkcijų argumentai, funkcijos grąžinami objektai; kopijuojantysis konstruktorius;**
**konstantiniai metodai, objektai ir metodų argumentai;**
**statiniai klasės laukai.**

### Struktūros

Tai – programuotojo iš standartinių duomenų arba kitų struktūrinių duomenų sudarytas duomens tipas. Duomens komponentai vadinami laukais. Jo gyvavimo ciklas: skelbimas – nurodoma duomens struktūra, o atmintis duomeniui dar neskiriama; apibrėžimas – skiriama atmintis; inicializavimas; darbas su duomeniu; duomens sunaikinimas – atminties išlaisvinimas.
Pavyzdys: tarkim, sudaroma struktūra *student*, į kurią bus įtalpinti du komponentai: studento identifikavimo numeris ir jo pažymių vidurkis.

```
#include <iostream>
using namespace std;
//
struct student{   // skelbimas
        long id;
        float avg;
};
//
int main( ){
        student s1, s2, s3;  //  apibrezimas
        s1 = { 12345L, 8.5f }  // inicializavimas
        s2.id = 6789L;   // kitas inicializavimo budas;
        s2.avg = 9.5F;   // prieitis prie lauku
        cout<<"Id. Numeris "<<s1.id<<" vidurkis "<<s1.vid<<endl;
        s3 = s2;  // prieskyra tarp vieno tipo duomenu galima
        s3.avg = (s1.avg + s2.avg)/2  // taip operuojama laukais
        cout<<"Id. Numeris "<<s3.id<<" vidurkis "<<s3.vid<<endl;
        return 0;
}
```

Struktūros gali būti ir funkcijų argumentų sąrašuose; jos perduodamos reikšme arba adresu.
Įdėtinės struktūros. Pavyzdžiui, trikampį būtų galima aprašyti tokiomis struktūromis:

```
. . .
struct vertex{  // virsune: dvi koordinates
        double x,y;
}
struct triangle{   // trikampis: trys virsunes
        vertex a, b, c
}
. . .
```

## Sąrašai *enum*

Faktiškai tai yra kintamasis, galintis įgyti reikšmę iš programuotojo nurodytų reikšmių tarpo. Tos reikšmės įvardinamos, tačiau operuojama visada tik atitinkamomis skaitinėmis reikšmėmis – lyg ir tų reikšmių vienmačio masyvo indeksu.
Pavyzdys-schema: savaitės dienų sąrašas:

```
. . .
enum sd{ Pirm, Antr, Trec, Ket, Penk, Sest, Sekm };
sd d1, d2, d3;
d1 = Pirm;
d2 = Antr;
d3 = Abc;  // klaida
int skirtumas = d2-d1;  //bus 1-0 = 1
cout<<Sekm<<endl;  // spausdins 6
. . .
```

Norint galima pradėti skaičiuoti indeksą ne nuo nulio, o nuo duotosios reikšmės:

```
Enum sd{ Pirm =1, Antr, Trec, Ket, Penk, Sest, Sekm };
```

## KLASĖS

```
#include <iostream>
using namespace std;
        //
class Class1{  //  pastaba 1
        private:    // 2
                int data1;
                double data2;
        public:     // 2
                void set_data( int d1, double d2 ){    // 3
                        data1 = d1;
                        data2 = d2;
                }
                void show_data( ){
                        cout<<"Data: "<<data1<<"    "<<data2<<endl;
                }
};  // 6
        //
int main( ){
        Class1 c1, c2;   // 4
        c1.set_data( 1, 2.3 );   // 5
        c2.set_data( 10, 20.3 );
        c1.show_data( );
        c2.show_data( );
        return 0;
}
```

Pastabos:

1. Klasės turi laukus ir funkcijas-narius – metodus. Čia – klasės skelbimas: tik nustatoma klasės struktūra, o atmintis neskiriama.
2. Nėra būtina, bet tradicinė programavimo schema: skelbti visus klasės laukus uždarais klasės duomenimis,
3. o metodus – viešais. Tokiu atveju būtini metodai, kuriais būtų galima suteikti reikšmes laukams ir tas reikšmes peržiūrėti (atitinkamai *set_data* ir *show_data*). Metodai laukus mato, kadangi patys yra klasių nariais. Metodų kūnai paprastai talpinami į klases, bet galima juos talpinti ir už klasės ribų (žr. žemiau).
4. Klasės *Class1* objektų (egzempliorių) *c1* ir *c2* apibrėžimas. Būtent čia objektams paskiriama atmintis. Faktiškai tik kiekvieno objekto laukai užima skirtingas atminties sritis, o visi klasės metodai, nežiūrint, kiek būtų apibrėžta objektų, bus saugomi vienoje atminties srityje. <Klasės>/<jos objekto> santykis atitinka <standartinį duomens tipą>/<jo kintamąjį> arba <struktūrą>/ <jos objektą>.
5. Metodai kviečiami taip. Visada metodai kviečiami tik kuriam konkrečiam objektui, o ne klasei.
6. Sintaksė: visada po duomenų skelbimo (kaip ir *struct, enum* atveju) reikalingas *;*.

Vietoje *set_data* tipo metodų dažnai laukai formuojami įvestimi, tokio tipo metodais:

```
. . .
void get_data( ){
        cout<<"Iveskite klases laukus";
        cin>>data1>>data2;
}
. . .
```

## Konstruktoriai

Faktiškai visų (ir vidinių, ir programuotojo sukurtų tipų) C++ duomenų egzemplioriai suformuojami specialiais metodais-konstruktoriais. Tokie konstruktoriai parengiami automatiškai, kompiliatoriaus: jų vardas sutampa su duomens vardu, argumentų sąrašas tuščias, jokių reikšmių jie negrąžina. Metodus galima perrašyti ir perkrauti.
Duomenys sunaikinami (t.y. išlaisvinama jų užimta kompiuterio operatyvioji atmintis) specialiais metodais-destruktoriais. Jų forma analogiška konstruktoriams, tik prieš vardą yra simbolis ~. Juos galima perrašyti.

Pavyzdys: klasė-skaitiklis. Kuriant klasės objektą, skaitiklį reikia inicializuoti nuline reikšme – tam perrašomas konstruktorius.

```
#include <iostream>
using namespace std;
//
class Counter{
        private:
                int count;
        public:
                Counter( ): count( 0 ) { }  // pastaba 1
                void inc_counter( ){
                        count++;
                }
                int get_counter( ){
                        return count;
                }
};
//
int main( ){
        Counter c1, c2;
        cout<<"c1: "<<c1.get_counter( )<<endl;
        cout<<"c2: "<<c2.get_counter( )<<endl;
        c1.inc_counter( );
        c1.inc_counter( );
        c2.inc_counter( );
                cout<<"c1: "<<c1.get_counter( )<<endl;
                cout<<"c2: "<<c2.get_counter( )<<endl;
                        return 0;
}
```

Pastaba:
Prieš konstruktorių jokio raktažodžio nereikia. Jei jo neperrašyti, bus taikomas konstruktorius pagal nutylėjimą: atmintis skiriama, o inicializavimas – bet kokia reikšme. Čia, perrašant konstruktorių, inicalizacija teikiama "inicializacijos sąraše" tarp argumentų sąrašo ir konstruktoriaus kūno (šiuo atveju jis tuščias). Jei reikėtų inicializuoti kelis laukus:

*Class2( ): d1( 0 ), d2 (100 ), d3( 0 ) { }*

ir pan. Tai – rekomenduojamas programavimo stilius. Pačiame konstruktoriaus kūne galima užprogramuoti, pavyzdžiui, spausdinimo veiksmus, braižymą ar grafinės vartotojo sąsajos parengimą. *Counter* konstruktorių galima būtų perrašyti:

```
Counter( ): count( 0 ){
        Cout<<"Vykdomas konstruktorius\n";
}
```

**Konstruktorių perkrovimas. Metodai už klasės ribų.**
**Objektai – funkcijų argumentai**

Pavyzdys: angliškų ilgio matų sudėtis.

```
#include <iostream>
using namespace std;
//
class Distance{
        private:
                int feet;
                float inches;
        public:
                Distance( ): feet( 0 ), inches( 0.f ) { }  //pastaba 1
                Distance( int ft, float in ): feet( ft ), inches( in ) { }
                void get_distance( ){
                        cout<<"Iveskite pedas ";
                        cin>>feet;
                        cout<<"Iveskite colius ";
                        cin>>inches;
                }
                void show_distance( ){
                        cout<<"Pedos "<<feet<<" coliai "<<inches<<endl;
                }
                void add_distances( Distance, Distance ); // 2, 4
};
//
void Distance::add_distances( Distance d1, Distance d2 ){  // 3
        feet = 0.f;
        inches = d1.inches + d2.inches;
        if( inches >= 12.f ){
                inches-= 12.f;
                feet = 1;
        }
        feet+= d1.feet + d2.feet;
}
//
int main( ){
        Distance d1, d2;  // 5
        Distance d3( 10, 11.12f );  // 5
        d2.get_distance( );    // 6
                d1.add_distances( d2, d3 );
                        cout<<"d1: "<<d1.show_distance( );
                        cout<<"d2: "<<d2.show_distance( );
                        cout<<"d3: "<<d3.show_distance( );
                                return 0;
}
```

Pastabos:

1. Konstruktorius dukart perkrautas: yra galimybė inicializuoti laukus nulinėmis reikšmėmis (pirmasis variantas), ir argumentų sąraše teikiamomis reikšmėmis (antrasis).
2. Klasėje teikiamas tik metodo protoptipas, o pats kūnas apibrėžiamas kitur.
3. Metodo priklausomybė klasei nurodoma standartiniu būdu (kaip *setf* metode *ios* vėliavėlėms ir pan.): dviem dvitaškiais.
4. Čia argumentai – tos pat klasės objektai; tai įmanoma, nes klasės laukai jau yra paskelbti aukščiau. Objektus funkcijai galima perduoti reikšme ir adresu.
5. Čia apibėžiami (t.y skiriama atmintis) ir inicializuojami (t.y. skiriamos kokios nors reikšmės) trys *Distance* objektai: iš pradžių *d1* ir *d2* inicializuojami nulinėmis reikšmėmis, po to *d3 – 10* ir *11.12f*, o po to *d2* reikšmės keičiamos: įvedamos iš klaviatūros metodu *get_distance*.
6. *d1* objekto laukai gaunami sudedant atitinkamus *d2* ir *d3* laukus.

## Grąžinami objektai. Kopijuojantysis konstruktorius

Perrašomas tas pats pavyzdys:

```
#include <iostream>
using namespace std;
//
class Distance{
        private:
                int feet;
                float inches;
        public:
                Distance( ): feet( 0 ), inches( 0.f ) { }
                Distance( int ft, float in ): feet( ft ), inches( in ) { }
                void get_distance( ){
                        cout<<"Iveskite pedas ";
                        cin>>feet;
                        cout<<"Iveskite colius ";
                        cin>>inches;
                }
                void show_distance( ){
                        cout<<"Pedos "<<feet<<" coliai "<<inches<<endl;
                }
                Distance add_distances( Distance );
};
//
Distance Distance::add_distances( Distance d ){
        Distance t;
        t.feet = 0.f;
        t.inches = inches + d.inches;
        if( t.inches >= 12.f ){
                t.inches-= 12.f;
                t.feet = 1;
        }
        t.feet+= feet + d.feet;
        return t;
}
//
int main( ){
        Distance d1, d2;
        Distance d3( 10, 11.12f );
        d2.get_distance( );
                d1 = d2.add_distances(  d3 );  // pastaba
                        cout<<"d1: "<<d1.show_distance( );
                        cout<<"d2: "<<d2.show_distance( );
                        cout<<"d3: "<<d3.show_distance( );
                                return 0;
}
```

Programos rezultatai – tokie pat.
Pastaba:
Čia prieskyros operatorius atlieka sudėtingus veiksmus: juk kopijuoti reikia ne vieną kintamąjį, o kelis klasės laukus. Tam kompiliatorius parengia "kopijuojantįjį konstruktorių". Jis gali būti kviečiamas taip, kaip parašyta, arba tokiu būdu:

*Distance d4( d1 );*

- objekto *d4* laukai taptų lygūs *d1* laukams.

## Modifikatorius *const* klasėje

Anksčiau buvo taikytas skelbiant konstantinius kintamuosius, arba norint padaryti nekintamais adresu perduodamus funkcijų argumentus.
Klasėje juo galima apibrėžti konstantinius metodus, konstantinius objektus ir konstantinius metodų argumentus.
Konstantiniai metodai: negali keisti klasės laukų reikšmių. Jų prasmė: padeda analizuoti programos tekstą ir supaprastina klaidų ieškojimą. Pavyzdys-schema:

```
. . .
class Class1{
        private:
                int f;
        public:
                void m1( ){ f = 1; }  // gerai
                void m2( ) const { f = 2; }  // klaida
};
. . .
```

Konstantiniai argumentai ir metodai: perrašomas ankstesnis pavyzdys:

```
#include <iostream>
using namespace std;
//
class Distance{
        private:
                int feet;
                float inches;
        public:
                Distance( ): feet( 0 ), inches( 0.f ) { }
                Distance( int ft, float in ): feet( ft ), inches( in ) { }
                void get_distance( ){
                        cout<<"Iveskite pedas ";
                        cin>>feet;
                        cout<<"Iveskite colius ";
                        cin>>inches;
                }
                void show_distance( ) const {  // pastaba 1
                        cout<<"Pedos "<<feet<<" coliai "<<inches<<endl;
                }
                Distance add_distances( const Distance& ) const; //  2
};
//
Distance Distance::add_distances( const Distance& d ) const { // 3
        Distance t;
        t.feet = 0.f;
        t.inches = inches + d.inches;
        if( t.inches >= 12.f ){
                t.inches-= 12.f;
                t.feet = 1;
        }
        t.feet+= feet + d.feet;
        return t;
}
//
int main( ){
        Distance d1, d2;
        Distance d3( 10, 11.12f );
        d2.get_distance( );
                d1 = d2.add_distances(  d3 );
                        cout<<"d1: "<<d1.show_distance( );
                        cout<<"d2: "<<d2.show_distance( );
                        cout<<"d3: "<<d3.show_distance( );
                                return 0;
}
```

Pastabos:

1. Taip skelbiamas konstantinis metodas. Šį metodą tokiu skelbti galima ir rekomenduojama, kadangi jis nekeičia klasės laukų.
2. Tarkim, argumentas dėl efektyvumo perduodamas adresu. Kad būtų galima apsisaugoti nuo netyčinio jo reikšmės pakeitimo, jį taip reikia skelbti konstantiniu. Patį metodą taip pat galima skelbti konstantiniu, kadangi jis nekeičia darbinės klasės laukų (kvietimo metu tai – *d2*).
3. Jei metodas skelbiamas ir apibrėžiamas skirtingose vietose, raktažodį reikia rašyti abiejose vietose.

Konstantiniai objektai – objektai, kurių laukų keisti negalima. Tokiems objektams todėl galima kviesti tik konstantinius metodus. Pavyzdys-schema ankstesnės programos pagrindu:

```
. . .
const Distance d4( );
. . .
d4.show_distance( );  // gerai
d4.get_distance( );   // klaida
. . .
```

## Statiniai klasės laukai

Statinis klasės laukas visiems tos klasės egzemplioriams turi tą pat reikšmę ir yra saugomas atmintyje vienoje vietoje (visi gi kiti laukai užimtų atskiras atminties sritis atskiriems objektams). Panašus į *static* kintamąjį: matomumo sritis – tik klasės vidus, gyvavimo laikas – programos laikas.

Pavyzdys: skaitiklis, skaičiuojantis, kiek klasės objektų sukurta iš viso.

```
#include <iostream>
using namespace std;
//
class Class1{
        private:
                static int count;  // butinas raktazodis static
        public:
                Class1( ) {  // neinicializuojantis konstruktorius;
                        count++;  // konstruojamas objektas priskaiciuojamas
                }
                void show_count( ){
                        cout<<"Objektu yra: "<<count<<endl;
                }
};
//
int Class1::count = 0;  // statinio lauko apibrezimas – uz klases ribu
//
int main( ){
        Class1 c1, c2;
        c1.show_count( );
        c2.show_count( );
        Class1 c3;
        c3.show_count( );
                return 0;
}
```

Statiniai laukai skelbiami ir apibrėžiami skirtingose vietose, sekant C++ ideologija: skelbimo metu atmintis neturi būti išskiriama. Be to, taip yra aiškiau, kad laukas priklauso visiems klasės egzemplioriams. Inicializuojamas tik vienąkart.