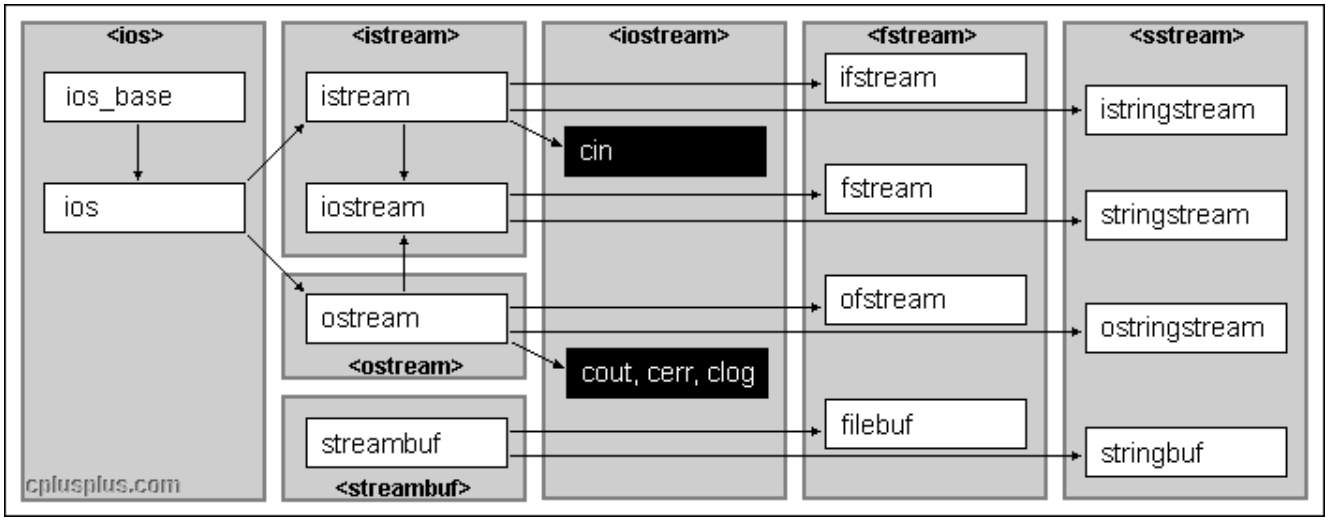
FAILŲ ĮVEDIMAS – IŠVEDIMAS (tęsinys).

Darbui su failais (išoriniais) skirtas *iostream* bibliotekos klasių rinkinys: **ifstream, ofstream, fstream**. Taip taikomi ir klasių protėvių **istream** ir **ostream,** skirtų bendram srautiniam įvedimui – išvedimui metodai (pvz., įvedimui – *get(), getline(), read()* ir išvedimui – *put(), write()*).



Darbas su failais prasideda nuo srauto objekto aprašymo, pvz.:

```
ofstream my_out;
ifstream my_in;
```

Toliau susieti sukurtą srautą su failu galima failų srautų klasių atidarymo metodais. Jų prototipai yra tokie:

Srautas, skirtas skaitymui iš failo:

```
ifstream::open(const char *name, ios::openmode = ios::in);
```

Srautas, skirtas rašymui į failą:

```
ofstream::open(const char *name, ios::openmode = ios::out | ios::trunc);
```

Failo skaitymo-rašymo srautas:

```
fstream::open(const char *name, ios::openmode = ios::in | ios::out);
```

Pvz., failo atidarymas rašymui galėtų atrodyti taip:

```
ofstream my_out;
my_out.open("pavyzdys.out", ios::out);
```

Kadangi klasės **ifstream, ofstream** ir **fstream** turi konstruktorius, kurie atidaro failus automatiškai, galima supaprastinti failo atidarymo aprašą:

```
ofstream my_out("pavyzdys.out");
```

Jei atidarymas atliktas sėkmingai, srautui priskiriama reikšmė 1, jei failo atidaryti nepavyko – 0. Tuo galima pasinaudoti nustatant ar failas buvo atidarytas:

```
ofstream my_out("pavyzdys.out");
if (!my_out) {
    //Aprodoti klaidą, išvesti pranešimą
};
```

Arba taikant failų srautų klasių metodą **is_open()** :

```
if (!my_out.is_open()) {
    //Aprodoti klaidą, išvesti pranešimą
};
```

Jei failas **pavyzdys.out** egzistuoja**,** jo turinys bus panaikintas (perrašytas iš naujo). Kai norima pildyti jau egzistuojantį failą, jį reikėtų atidaryti tokiose režimuose:

```
ofstream my_out("pavyzdys.out", ios::app);
```
arba
```
ofstream my_out("pavyzdys.out", ios::ate);
```

Patikrinti ar jau pasiekta failo pabaiga galima taikant loginę funkcija *eof()*, uždaryti failą funkcija *close()*. Failas taip pat uždaromas destruktoriumi automatiškai, kai srauto objektas išeina iš matomumo srities.

Pavyzdys 2.1. Skaičių sekos rašymas į failą.

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
const char* output_file_name = "pavyzdys.out";
ofstream my_out(output_file_name);
if (my_out.fail()) {
cerr << "Negalima atverti failo "<< output_file_name<< " rasymui." << endl;
return 1;
 }
for (int k=1; k<=10; k++) {
        my_out << k << " ";
 }
my_out << endl;
return 0;
}
```

Pavyzdys 2.2. Failo skaitymas.

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
const char* input_file_name = "pavyzdys.out";

ifstream my_in(input_file_name);
if (my_in.fail()) {
cerr << "Negalima atverti failo " << input_file_name
<< " ivedimui" << endl;
return 1;
}
int i;
int sum = 0;
while (true) {
        my_in >> i;
        if (my_in.fail()) break;
        sum += i;
}
cout << "Skaiciu suma = " << sum << endl;
getchar();
return 0;
}
```

ĮVADAS Į C++ KLASĖS

KLASĖS APRAŠAS. Tipinė klasės aprašo struktūra yra tokia:

class <*Klasės_vardas*>
{
private:
<*Kintamųjų (duomenų, atributų) sąrašas*>;
public:
<*Metodų (funkcijų) sąrašas*>;
} <*klasės objektų sąrašas*>;

Nurodant tokį tipinį prieigos prie klasės elementų variantą, įgyvendinamas inkapsuliacijos principas, t. y., prieiti prie klasės duomenų laukų galima tik per jos atvirus metodus (per vadinamąją *klasės sąsają*).

Pavyzdys 2.3. Paprastos klasės apibrėžimas ir naudojimas.

```
#include <iostream>
using namespace std;
class Paprasta {
      private:
        int duomenys;
      public:
        void priskirti(int a) { duomenys = a; }
        void parodyti()    {cout<<"Duomenys ="<<duomenys <<endl;  }
     };
void main()
        {
        Paprasta p1, p2;
        p1.priskirti (2014);  p2.priskirti (2017);
        p1.parodyti(); p2.parodyti();
}
```

Realiose programose metodai yra sudėtingesni, todėl klasėje metodai tik deklaruojami (rašomi tik metodų prototipai), o pačių metodų aprašai iškeliami už klasės ribų. Išorinė metodo aprašo forma:

*<Tipas> <**klasės_vardas**> **::**<**metodo_vardas**>(parametrų_sąrašas)*
{ metodo tekstas }

void Paprasta :: priskirti(int a)
{ duomenys = a; }

Prieiga prie klasės duomenų ir metodų galima organizuoti ir naudojant rodykles į šios klasės objektus.

Pavyzdys 2.4. Rodyklių naudojimas prieigai prie klasės elementų.

```
class Taskas {
      public:
        int x,y;
        void nustatyti(int _x, int _y)
     };
void Taskas::nustatyti(int _x, int _y)
     {
        x=_x;
        y= _y;
       }
int main()
       {
         Taskas t1;
         Taskas *t1ptr = &t1;
         //
         t1.x = 0;              //objektas.klasės_elementas
         t1ptr –> y = 0;        //rodyklė–> klasės_elementas
         t1ptr –>nustatyti(3,4);
         return 0;
}
```

Iš esmės abu prieigos būdai yra analogiški, nes, pvz.,

```
t1.nustatyti(3,4);
```

kompiliatorius supranta kaip

```
(&t1) –>nustatyti(3,4);
```

VARDŲ MATOMUMAS

Klasės elementų vardų galiojimo sritis yra ši klasė, tad klasės metodai gali kreiptis į visus klasės elementus tiesiog pagal jų vardus. Tačiau kitos klasės turi kreiptis į šios klasės elementus naudojant matomumo praplėtimo operaciją **::**

<klasės_vardas>::<kintamojo_ar_metodo_vardas>

Klasės duomenys negali būti apibrėžiami, naudojant modifikatorius **auto**, **extern** ir **register**; klasės kintamuoju negali būti tos pačios klasės objektas, tačiau gali būti rodyklė į objektą ar kitos klasės objektas.

Pagal nutylėjimą vardų matomumas yra lokalus, t.y. **private**. Jei vardų matomumo požymis nurodytas, jis galioja iki kito požymio nurodymo arba iki klasės apibrėžimo pabaigos. Klasės elementų vardai gali būti lokalūs, globalūs ir apsaugotieji :

private
// Duomenys ir metodai prieinami tik šios klasės metodams, t.y. lokalūs.

public
//Duomenys ir metodai prieinami tiek klasės metodams, tiek ir išorinėms //funkcijoms, t.y. globalūs.

protected
//Klasės elementai prieinami klasėje, kuri paveldi duotąją klasę. //Paveldėtoje klasėje jie galioja **private** teisėmis.

C++ kalboje struktūros ir junginiai taip pat traktuojami kaip klasės, t.y. gali turėti tiek duomenų laukus, tiek metodus darbui su šiais duomenimis, tačiau skirtingai nuo klasių jų elementai yra globalūs:

Savybės	Klasės	Struktūros	Junginiai
Bazinis žodis	**class**	**struct**	**union**
Priėjimas prie duomenų (pagal nutylėjimą)	**private**	**public**	**public**
Duomenų perdengimas	ne	ne	taip

KONSTRUKTORIAI IR DESTRUKTORIAI

Kuriant klasių objektus, juos reikia inicializuoti. Tam tikslui naudojami specialūs metodai – **konstruktoriai**, kurie vykdomi automatiškai kiekvieną kartą, kai yra sukuriamas tam tikros klasės objektas (objekto skelbimas reiškia objekto sukūrimą). Konstruktorius turi tokį pat vardą, kaip ir klasė ir neturi grąžinamų reikšmių.

Kai objektas šalinamas, iškviečiamas specialus atvirkštinis konstruktoriui metodas – destruktorius. Destruktorius turi tokį pat vardą, kaip ir klasė, o prieš vardą rašomas simbolius **~**

Destruktorius neturi parametrų ir grąžinamų reikšmių. Kiekviena klasė turi tik vieną destruktorių.

Konstruktoriai ir destruktoriai nėra paveldimi ir negali būti apibrėžiami naudojant modifikatorius **const**, **volatile**, **static**, konstruktorius – ir **virtual**.

Pavyzdys 2.5. Konstruktorius ir destruktorius.

```
#include <iostream.h>
class Taskas
{
        int x, y;
  public:
        Taskas(int _x, int _y);                 //Konstruktorius
        ~Taskas();                              //Destruktorius
        void parodyti();
}
Taskas::Taskas(int _x, int _y)
        {x = _x;
         y = _y;
                cout<<“Vykdomas konstruktorius \n”;
        }
Taskas:: ~ Taskas()
        {       cout<<“Vykdomas destruktorius \n”;
        }

void Taskas::parodyti()
        { cout << x << ‘ ‘ <<  y << endl;}

void main()
{
        Taskas t1(3, 4);
        t1.parodyti();
 }
```

Klasės duomenims reikšmės gali būti priskiriamos naudojant vadinamąjį *inicializacijos sąrašą*. Toks sąrašas nurodomas po konstruktoriaus parametrų sąrašo ir dvitaškio. Naudojant inicializacijos sąrašą, kintamiesiems reikšmės priskiriamos prieš konstruktoriaus vykdymą, todėl pats konstruktorius dažnai lieka tuščias. Nuorodos ir *const* tipo duomenys gali būti inicializuojamos tik naudojant inicializacijos sąrašą.

Pavyzdys 2.6. Inicializacijos sąrašas.

```
class circle
{
        int radius;
```

```
        int c_x, c_y;
  public:
        circle (int a, int b, int c) : radius(a), c_x(b), c_y(c)
            {  }
};
```

Be automatinio konstruktoriaus, kuris vykdomas pagal nutylėjimą ir neturi parametrų, C++ kalboje realizuotas **kopijavimo konstruktorius**. Jis sukuria klasės objektą kopijuodamas jau egzistuojančio šios klasės objekto duomenis į naują objektą. Tam tikslui kopijavimo konstruktorius naudoja vienintelį parametrą – nuorodą į klasės objektą, pvz.: *(const Taskas&)* arba *(Taskas&)* .

Jei kopijavimo konstruktorius nėra apibrėžtas programuotojo, kompiliatorius sukuria kopijavimo konstruktorių pagal nutylėjimą. Toks konstruktorius sukuria naują objektą ir atlieka tiesioginį pobitinį egzistuojančio objekto kopijavimą.

Pavyzdys 2.7. Kopijavimo konstruktorius.

```
class Taskas
{
        int x, y;
  public:
        Taskas(const  Taskas&  src);        //Kopijavimo konstruktorius
};

Taskas::Taskas(const  Taskas&  src)
    {
        x = src.x;
        y = src.y;
    }

void main()
{
        Taskas  t1 (3, 4);
        Taskas t2 = t1;
        Taskas  t3 (t1);
        // …
 }
```

Jei objektas turi savyje nuorodas arba rodyklės, programuotojas būtinai turi aprašyti savo kopijavimo konstruktorių. Toks konstruktorius taip pat būtinas, kai klasėje perkraunamas priskyrimo operatorius.

Kiekvienam C++ objektui kompiliatorius sukuria specialią rodyklę vadinimą **this** , kuri rodo į patį objektą. Tokiu būdu klasės metoduose objekto adresas prieinamas kaip rodyklė **this**. Rodyklės **this** galiojimo sritis – klasė, kurioje ji apibrėžta.

Iš esmės, rodyklė **this** yra paslėptas klasės parametras, kurį pats kompiliatorius prideda prie klasės apibrėžimo. Kai iškviečiamas klasės metodas, **this** rodyklė jam perduodama kaip pirmas metodo parametras, t.y.:

Objekto_vardas.metodas (par1, par2);

traktuojama kaip

Objekto_vardas.metodas (Objekto_vardas&, par1, par2);

Pvz.

```
{ cout<<par1<<endl;
  cout<<this–>par1<<endl;          // išvedamas tas pats rezultatas
  cout << this<< endl; }           // išvedamas objekto adresas
```

Paprastai rodyklė **this** naudojama operatorių perkrovime ir kitais atvejais, kai metodas gražina rodyklę arba nuorodą į objektą, atitinkamai: *return this;* arba *return *this;*

STATINIAI KLASĖS ELEMENTAI

Klasės elementai gali būti aprašyti naudojant modifikatorių **static**. Tokie elementai tampa globalias kintamaisiais arba funkcijomis ir yra saugomi iki programos vykdymo pabaigos. Jų vardai galioja klasės matomumo srityje, todėl pasiekiami tik iš klasės. Statiniai duomenys naudojami visais klasės objektais, nes iš tikrųjų kuriamas tik vienas, visiems prieinamas tokių duomenų egzempliorius.

Jei statinis kintamasis paskelbtas **public** srityje galimi tokie kreipiniai:

<objekto_vardas> **.** <kintamojo_vardas>

<nuoroda_į_objektą> –> <kintamojo_vardas>

Tačiau pabrėžiant, kad toks kintamasis yra vienintelis visai klasei, patartina kreiptis į jį taip:

<klasės_vardas> **::** <kintamojo_vardas>

Jei statinis kintamasis paskelbtas **private** ar **protected** srityje prieigai prie jo naudojami klasės metodai**,** tačiau turi būti sukurtas nors vienas šios klasės objektas. Taigi, statinius kintamuosius patartina aprašyti, kai numatyta, kad keli klasės objektai naudos bendrus duomenis (dalinsis informacija).

Statinės funkcijos taip pat egzistuoja vieninteliame egzemplioriuje ir nepriklauso konkrečiam objektui, todėl jiems neperduodama rodyklė **this**. Statinės funkcijos pasižymi tokiomis ypatybėmis:

- jos gali būti iškviečiamos ir tada, kai nėra sukurtas joks klasės atstovas;
- statinės funkcijos gali naudoti tik statinius klasės duomenis ir kreiptis tik į kitas statines funkcijas;
- statinė funkcija negali būti aprašyta naudojant modifikatorių **virtual** .

Pavyzdys 2.8. Statinių duomenų ir metodų naudojimas. Objektų skaitiklis.

```
#include <iostream>
using namespace std;

class St
  {
      public:
        St() {ObCount++;};
        ~St(){ObCount– –;};
        static int GetCounts() {return ObCount;}

      private:
        int x;
        static int ObCount;
   };
int St :: ObCount = 0;   //
int main()
{
   St* pOb = new St[5];
   cout<< “Yra sukurta“ << St :: GetCounts()<<“klasės St objektų“ << endl;
   delete[] pOb;
   return 0;
 }
```

PASTOVŪS KLASĖS ELEMENTAI

Klasės metodai gali būti apibrėžti su modifikatoriumi **const**. Toks metodas negali modifikuoti klasės kintamųjų reikšmių, todėl **const** metodai paprastai naudojami reikšmių išvedimui. Tokių metodų argumentų sąraše dažniausiai būna nuorodos.

Pavyzdys 2.9. Pastovūs (const) metodai.

```
class Taskas
  {
       int x,y;
     public:
       Taskas(int _x, int _y) ;
       void Nustatyti(int _x, int _y) ;
       void Gauti(int _x, int _y) const ;        //Pastovus metodas
   }
```

Su modifikatoriumi **const** gali būti apibrėžiami ir *konstantiniai objektai*. Taip apibrėžtą objektą negalima modifikuoti, tai reiškia kad galima naudoti tik **const** metodus, nes tik tokiu atveju garantuojama, kad duomenys nebus modifikuoti.

```
const Taskas ob(10,20) ;
```

C++ standarte taip pat apibrėžtas tipo modifikatorius **mutable**, kuris leidžia nurodyti kokie klasės kintamieji gali būti modifikuojami konstantinėmis klasės funkcijomis. Šis modifikatorius negali būti taikomas su statiniais ir konstantiniais kintamaisiais.

Pavyzdys 2.10. Tipo modifikatorius **mutable**.

```
#include <iostream>
using namespace std;
class AnyClass
  {
       mutable int count;
       mutable const int* iptr;
     public:
       int func(int i=0) const
       {
           count = i++;
           iptr =&i;
           cout<< iptr;
           return count;
   }
```

Apibendrinant, galima pateikti tokias *const* modifikatorių taikymo rekomendacijas (žr. *Distance4.cpp* programos kodą):

- kai metodas neturi keisti klasės laukų reikšmių, rekomenduojama jį skelbti konstantiniu metodu (tokiu atveju, jei bus klaidingai bandoma keisti šias reikšmes, kompiliatorius aptiks tokią klaidą, pvz., (pvz., *void showDistance() const*);
- kai metodo parametrai perduodami pagal adresą ir jie yra objektai, kurių reikšmių nežadama keisti, patartina juos skelbti konstantiniais objektais (pvz., *const Distance&*);
- kai sudaryto objekto reikšmių toliau keisti nebereikėtų, jį rekomenduojama skelbti konstantiniu objektu (pvz., *const Distance d3(1, 2.4).*