

# Objektinis Programavimas

**Vartotojo apibrėžti tipai**



**Iš čia kylama  
į žvaigždes**

# Turinys

1. Procedūrinis vs. objektinis programavimas
2. Duomenų tipai: struct'ūros ir class'ės
3. Inkapsuliavimas
4. Konstruktoriai
5. Destruktoriai
6. "this" rodyklė
7. Klasės realizacija ir header failai

# Procedūrinis vs. objektinis programavimas (1)

- Objekto samprata C++ kalboje priklauso nuo konteksto.
- Procedūrinio programavimo kontekste:
  - **Objektas** yra atminties "gabalas", kuriame saugomos reikšmės.
  - Objektas turintis vardą vadinamas **kintamuoju** (**variable**).
- Objektiniame programavime objektas yra suprantamas kaip objektas pagal struktūrinį programavimą, bet tuo pačiu kaip objektas apjungiantis **savybes** (kintamuosius) ir **elgseną** (funkcijas).

## Procedūrinis vs. objektinis programavimas (2)

- Procedūriniame programavime programos yra kompiuterio instrukcijų sąrašai, kuriuose **duomenys apibrėžiami per objektus**, o su duomenimis dirbame per **išraiškas (statement) ir funkcijas**.
- Duomenys ir funkcijos yra atskiri subjektai, kurie yra apjungiami norint pasiekti pageidaujamą rezultatą.
- Programuotojas susieja **savybes** (kintamuosius) su **elgesiu** (funkcijomis):

```
vaziuoja(Petras, "į mokyklą"); // funkcija vaziuoja()
```

# Procedūrinis vs. objektinis programavimas (3)

- **Objektai** supa mus: maisto produktai, žmonės, automobiliai ir t.t.
- Objektai turi **dvi pagrindines sudedamąsias dalis**:
  1. atitinkamų savybių sąrašą, pvz.: **svoris, spalva, dydis, forma ir t.t.**
  2. tam tikrą elgesį (veiksmus), kurį jie gali atlikti, pvz., būti sugedęs, sportuoti, važiuoti ir pan.
- Objektiškai orientuotas programavimas (**OOP**) suteikia galimybę kurti objektus, kurie **apjungia savybes (kintamuosius) ir elgseną (funkcijas) į vieną visumą**:

```
Petras.vaziuoja("į mokyklą"); // nario funkcija .vaziuoja()
```

# Procedūrinis vs. objektinis programavimas (4)

- Objektiniame programavime aišku, kas yra objektas (Petras) ir ką jis veikia (važiuoja į mokyklą).
- Užuoat orientavęsi į funkcijas (struktūrinis programavimas), mes orientuojamės į objektus, kurie turi aiškiai apibrėžtą elgesių visumą.
- Tokia programavimo paradigma vadinama "**objektiškai-orientuota**".
- OO nepakeičia tradicinio programavimo metodų, o jį praplečia ir leidžia programas padaryti aiškesnėmis ir bendresnėmis.

# Duomenų tipai

- **Pagrindiniai (fundamental) duomenų tipai:** char, int, long, float, double, ir t.t.
- **Integruoti (build-in) tipai:** sudaryti naudojant **pagrindinius duomenų tipus** apjungiant juos su: const, \*, &, [].
- **Vartotojo apibrėžti tipai:** sudaryti iš **integruotų tipų** naudojant C++ abstrakcijos mechanizmus.

# struct'ūros tipas

```
struct Vector {  
    int sz;           // elementų skaičius  
    double* elem;    // rodyklė į elementus  
};
```

// Naują Vector'ių susikuriame:

```
Vector v; // SVARBU: v.elem rodyklė point'ina į niekur!
```

// Norint išvengti katastrofos, Vektorių reikia inicializuoti:

```
void VectorInit(Vector& v, int s) {  
    v.elem = new double[s]; // išskirti s dydžio double tipo masyvą  
    v.sz = s;  
}
```



# struct'ūros tipas

## Vector panaudojimo pavyzdys

```
// nuskaityti (s > 0) skaičių iš std::cin stream'o ir grąžina jų sumą:
double readAndSum(int s) {
    Vector v;
    VectorInit(v, s);
    for (int i=0; i!=s; ++i)
        cin >> v.elem[i];

    double sum = 0.0;
    for (int i=0; i!=s; ++i)
        sum += v.elem[i];
    return sum;
}
```

// išskirti atminties s elementams

// nuskaityti reikšmes į elem masyvą

// apskaičiuoti elementų sumą

# struct'ūros tipas

## Struktūros narių pasiekiamumas

- Naudojame . (tašką), norėdami pasiekti struktūros narius (**members**) per vardą (ir nuorodą &), tačiau -> norėdami pasiekti per rodyklę \*:

```
void func(Vector v, Vector& rv, Vector* pv) {  
    int i1 = v.sz;           // pasiekiamame per vardą  
    int i2 = rv.sz;          // pasiekiamame per nuorodą  
    int i3 = pv->sz;          // pasiekiamame per rodyklę  
}
```

# class'ės tipas

- Objektiniame pasaulyje įprasta, kad naujai sukurti tipai ne tik saugo duomenis, bet ir turi funkcijas, kurios veiktų su tais duomenimis.
- Kuriant naujus tipus yra siekiama atskirti naujo duomenų tipo **realizaciją** (vartotojams nesvarbi) nuo vartotojo sąsajos (**interfeiso**).
- C++ kalboje tam skirtas mechanizmas vadinamas klase (class).
- Klasės nariai gali būti: duomenys, funkcijos bei tipo nariai.
- Klasės interfeisą nusako **public** nariai, o **private** nariai yra pasiekiami tik per šį interfeisą.

# struct **vs.** class tipai

```
struct Vector { // struct default narių pasiekiamumas yra: public
    int sz;      // elementų skaičius
    double* elem; // rodyklė į elementus
};
```

Ekvivalenti realizacija naudojant class'ę:

```
class Vector {
    public:      // class default narių pasiekiamumas yra: private
        int sz;      // elementų skaičius
        double* elem; // rodyklė į elementus
};
```

# class'ės tipas

## public **vs.** private narių pasiekiamumas

// class nariai yra private pagal nutylėjimą

```
class Vector {  
    int sz;           // elementų skaičius  
    double* elem;    // rodyklė į elementus  
};
```

```
int main() {  
    Vector v;  
    int s = 10;  
    v.sz = s;           // Negalima!  
    v.elem = new double[s]; // Negalima!  
}
```

// Įvykdžius šią realizaciją, gauname:  
error: 'int Vector::sz' is private within this context  
 v.sz = s;

# class'ės tipas

**Kam tada iš viso reikalingas** private **pasiekiamumas**?

- Realiam gyvenime nuolat **sėkmingai** naudojame sudėtingus technologinius sprendimus, pvz. vairuojame automobilį, naudojames telefonu, ar žiūrime televizorių nesuprasdami kaip jie realizuoti.
- Juk lygiai taip pat sėkmingai atliktami 2-ąjį darbą naudojome `std::vector` ar `std::cout` tipus ir vėl nesigilindami, kaip jie realizuoti!
- Pasirodo vartotojo sąsajos (**interfeiso**) atskyrimas nuo realizacijos **yra labai svarbus ne tik gyvenime, bet ir programavime!**

# Inkapsuliavimas (1)

- OOP **encapsulation** yra procesas kurio metu nuo objekto vartotojų paslepiana informacija kaip objektas yra realizuotas.
- Objekto vartotojai naudojami objektu per viešą (`public`) interfeisą.
- C++ inkapsuliacija realizuojama per pasiekiamumo specifikacijas.
- Įprastai (visi) klasei priklausančys **kintamieji yra privatūs** (`private`), o **dauguma funkcijų yra viešos** (`public`).
- Tokiu būdu klasės vartotojams pakanka žinoti, kokios nario funkcijos egzistuoja, visiškai nesirūpinant kaip jos realizuotos!

# Inkapsuliavimas (2)

## Padedu lengviau atnaujinti programos realizaciją (1)

```
class Vector {  
    public:  
        int sz;           // elementų skaičius  
        double* elem;     // rodyklė į elementus  
};
```

```
// Tuomet:  
Vector v;  
v.sz = 10; // Viskas OK, sz yra public
```

- Bet kas jeigu nuspręsimė sz pervadinti į size arba dydis?
- Sugadintume ir šią ir kitas programas, kurios naudoja Vector klasę!



# Inkapsuliavimas (3)

## Padedą lengviau atnaujinti programos realizaciją (2)

```
class Vector {  
    private:  
        int sz;          // elementų skaičius  
        double* elem;    // rodyklė į elementus  
    public:  
        void setSize(int size) { sz = size; }  
        double getSize() const { return sz; } // read-only  
};  
  
// Tuomet:  
Vector v;  
v.setSize(10); // Viskas OK, setSize() yra public  
v.getSize();   // Viskas OK, getSize() yra public
```

— Tuomet pakeitus **int sz;** į **int dydis**, mes turime atnaujinti tik klasę!

# Inkapsuliavimas (4)

## Padedą lengviau atnaujinti programos realizaciją (3)

```
class Vector {  
    private:  
        int dydis;      // elementų skaičius  
        double* elem;   // rodyklė į elementus  
    public:  
        void setSize(int size) { dydis = size; }  
        double getSize() const { return dydis; } // read-only  
};  
  
// Tuomet:  
Vector v;  
v.setSize(10); // Viskas OK, setSize() yra public  
v.getSize();   // Viskas OK, getSize() yra public
```

— Programa, kaip ir kitos naudojančios Vector lieka nepakitusios!

# Inkapsuliavimas (5)

**Padedą apsaugoti duomenis ir užkirsti kelią klaidoms (1)**

```
class Vector {  
    public:  
        int sz;           // elementų skaičius  
        double* elem;     // rodyklė į elementus  
};  
  
Vector v;  
VectorInit(v, 10); // išskirti atminties 10 elem.  
v.elem[15] = 10;    // WRONG! Perrašėme atmintį kurios nevaldome!
```

# Inkapsuliavimas (6)

## Padedą apsaugoti duomenis ir užkirsti kelią klaidoms (2)

```
class Vector {
private:
    int sz;          // elementų skaičius
    double* elem;    // rodyklė į elementus
public:
    void init(int s) { // inicializuojame vektorių
        elem = new double[s];
        sz = s;
    }
    void setElem(int idx, double val) {
        // Jei idx blogas, pvz. nieko negražinti
        if (idx < 0 || idx >= sz) return;
        elem[idx] = val;
    }
};
```

# Prieigos (public) funkcijos: get'eriai ir set'eriai

```
class Vector {  
    // realizacijos  
private:  
    int sz;           // elementų skaičius  
    double* elem;     // rodyklė į elementus  
    // interfeisas  
public:  
    // set'er funkcija  
    void init(int s) {  
        elem = new double[s];  
        sz = s;  
    }  
    // get'er funkcija  
    double getElem(int i) { // kas būtų jeigu grąžintume double& ?  
        return elem[i];  
    }  
    // get'er funkcija  
    int size() const { return sz; }  
};
```

# Konstruktoriai (1)

**Kai visi klasės (arba struct) nariai yra vieši, galime inicializuoti klasę tiesiogiai 1 iš 2 būdų:**

```
#include <iostream>

class Vector {
public:
    int sz;          // elementų skaičius
    double* elem;    // rodyklė į elementus
};

int main() {
    int s = 10;
    Vector v = {s, new double[s]}; // 1: per inicializacijos sąrašą
    // Vector v{s, new double[s]}; // 2: nuo C++11: panaudojant universalią inicializaciją
    std::cout << v.sz << std::endl;
    std::cout << v.elem[0] << std::endl;
}
```

// Rezultatas:

10

0

## Konstruktoriai (2)

- Tačiau, jei kintamieji yra privatūs, tokiu būdu mes negalėsime inicializuoti klasės.
- **Logiška:** jei negalima tiesiogiai pasiekti kintamojo (yra privatus), tai negalima ir tiesiogiai inicializuoti tokios klasės.
- Taigi, tai kaip tada inicializuoti klasę su privačiais kintamaisiais?
- **Atsakymas:** naudojant konstruktorius!

## Konstruktoriai (3)

- **Konstruktorius** yra speciali klasės funkcija, kuri **automatiškai iškviečiama tik tuomet kai klasės objektas yra sukuriamas**.
- Konstruktoriai yra naudojami inicializuoti klasės narių kintamuosius numatytomis (**default**) arba vartotojo pateiktoms reikšmėms.
- Skirtingai nuo įprastų nario funkcijų, konstruktoriai turi specialias taisykles, kaip jie turi būti pavadinti:
  - **Konstruktoriai visada turi tą patį pavadinimą kaip ir klasė (ta pati kapitalizacija)**.
  - Konstruktoriai neturi tipo grąžinimo (net ir **void** negalima).



# Numatytasis (**default**) konstruktorius (1)

- Konstruktorius neturintis jokių parametrų reikšmių (arba turintis visas numatytas) yra vadinamas **numatytasis konstruktorius**.
- Iškviečiamas kai objektas inicializuojamas be parametrų reikšmių:

```
class Vector {  
    private:  
        int sz;           // elementų skaičius  
        double* elem;     // rodyklė į elementus  
    public:  
        Vector() {        // default konstruktorius  
            sz = 0;  
            elem = new double[sz];  
        }  
};
```

# Numatytasis (**default**) konstruktorius (2)

```
#include <iostream>

class Vector {
private:
    int sz;          // elementų skaičius
    double* elem;    // rodyklė į elementus
public:
    Vector() {        // default konstruktorius
        sz = 0;
        elem = new double[sz];
    }
    int size() const { return sz; }
    double getElem(int i) { return elem[i]; }
};

int main() {
    Vector v;        // kviečia numatytąjį Vector() konstruktorių
    std::cout << v.size() << '\n';    // Rezultatas: 0
    std::cout << v.getElem(0) << '\n'; // Rezultatas: 0, bet šiaip undefined!
}
```

# Konstruktorių persidengimas (1)

```
#include <iostream>

class Vector {
private:
    int sz;          // elementų skaičius
    double* elem;    // rodyklė į elementus
public:
    Vector() {        // default konstruktorius
        sz = 0;
        elem = new double[sz];
    }
    Vector(int s) {   // konstruktorius su 1 parametru
        sz = s;
        elem = new double[s](); // value initialization
    }
    int size() const { return sz; }
    double getElem(int i) { return elem[i]; }
};

int main() {
    Vector v(10); // kviečia konstruktorių su 1 parametru
    std::cout << v.size() << '\n'; // Rezultatas: 10
    std::cout << v.getElem(0) << '\n'; // Rezultatas: 0
}
```

# Konstruktorių persidengimas (2)

```
#include <iostream>
#include <algorithm> // std::fill_n

class Vector {
private:
    int sz;          // elementų skaičius
    double* elem;    // rodyklė į elementus
public:
    Vector() {        // default konstruktorius
        sz = 0;
        elem = new double[sz];
    }
    Vector(int s) {    // konstruktorius su 1 parametru
        sz = s;
        elem = new double[s](); // value initialization
    }
    Vector(int s, double val) { // konstruktorius su 2 parametrais
        sz = s;
        elem = new double[s];
        std::fill_n(elem, s, val); // užpildome val reikšmėmis
    }
    int size() const { return sz; }
    double getElem(int i) { return elem[i]; }
};

int main() {
    Vector v(10, 5.5); // kviečia Vector(int, double) konstruktorių
    std::cout << v.size() << '\n'; // Rezultatas: 10
    std::cout << v.getElem(0) << '\n'; // Rezultatas: 5.5
}
```

# Kompiliatoriaus sukurtas (**synthesized**) konstruktorius

- Kas nutinka kai mes nesukuriamo klasės tipui konstruktoriaus(-ių)?
- Tuomet kompiliatoriai sukuria (public) konstruktorių automatiškai:  
`Vector() {} ; // susintetintas konstruktorius`
- Sintezuotas konstruktorius automatiškai inicializuoja objektų (kintamųjų) reikšmes priklausomai nuo jų tipo:
  - Jei objektas yra klasės tipo, tada atinkamas konstruktorius kontroliuoja tos klasės objektų inicializavimą.
  - Jei objektas yra integruoto (**built-in**) tipo tuomet value-initializacija priskiria nulį (0), o **default** yra neapibrėžta.

# Konstruktorių vykdymo eiliškumas (1)

```
#include <iostream>
class Vector { // Sutrumpinta Vector versija
public:
    Vector() {
        std::cout << "Kuriame Vector tipo objektą\n";
    }
};
class Studentas {
private:
    Vector v; // Vector tipo konteineris
public:
    Studentas() {
        std::cout << "Kuriame Studentas tipo objektą\n";
    }
};
int main() {
    Studentas s; // Ką atspausdins ši programa?
}
```

## Konstruktorių vykdymo eiliškumas (2)

- Kai kintamasis **Studentas s;** yra sukuriamas, Studentas() konstruktorius yra iškviečiamas.
- Prieš vykdant konstruktoriaus išraiškas esančias tarp {} pirma yra (private) kintamasis **Vector v;** inicializuojamas iškviečiant numatytąjį Vector() konstruktorių.
- Tai atspausdina "**Kuriame Vector tipo objektą**" ir tuomet sugrįžta kontrolė pas Studentas() konstruktorių ir vykdo išraiškas tarp {}.
- **Kodėl tokia tvarka?** Pagalvokime kas būtų, jei konstruktorius norėtų naudoti Vector tipo kintamojo v reikšmę, o jis būtų ne inicializuotas?

# Konstruktorių vykdymo eiliškumas (3)

```
class Vector {  
    private:  
        int sz;  
        double* elem;  
    public:  
        Vector() {                // default konstruktorius  
            sz = 0;                // priskiriame (ne inicializuojame)  
            elem = new double[sz]; // priskiriame (ne inicializuojame)  
        }  
};  
  
int main() { Vector v; /* kviečia numatytąjį Vector() konstruktorių */ }
```

Kai vykdomė Vector(), pirmiausia kintamieji (sz ir \*elem) sukuriami (ir inicializuojami), o tik po to vykdomas konstruktoriaus kodas tarp {}.



## Konstruktorių vykdymo eiliškumas (4)

- Nesunku suprasti, kad ši dviejų žingsnių "inicializacija" yra mažiau efektyvu negu inicializacija ir reikšmių priskyrimas tuo pačiu metu.
- Dar daugiau: const ir nuorodos & tipo kintamieji turi būti inicializuoti jų sukūrimo metu:

```
class Vector {  
    private:  
        const int sz;  
    public:  
        Vector() {  
            sz = 0; // klaida: const turi būti inicializuotas  
        }  
};
```

# Konstruktorių inicializavimo sąrašai (**member initializer lists**)

- Narių inicializavimo sąrašas įterpiamas po konstruktoriaus parametrų (po skliaustų) ir prasideda dvitaškiu ":", o tuomet inicializuojame kiekvieną kintamąjį, atskiriant juos kableliu:

```
class Vector {  
private:  
    int sz;  
    double* elem;  
public:  
    Vector() : sz(0), elem(new double[sz]) {}  
    Vector(int s) : sz{ s }, elem{ new double[sz] } {} // C++11 stilius  
    Vector(int s, double val) : sz(s), elem(new double[sz]) {  
        std::fill_n(elem, s, val); // užpildome val reikšmėmis  
    }  
};
```

# Destruktoriai (1)

- **Destruktorius** yra kita speciali klasės funkcija, kuri **automatiškai iškviečiama tik tuomet kai klasės objektas yra sunaikinamas**.
- Destruktorius naudojamas atlaisvinti konstruktorių išskirtą atmintį.
- Destruktorius turi specialias taisykles, kaip jis turi būti pavadintas:
  - **Destruktorius visada turi tą patį pavadinimą kaip ir klasė (ta pati kapitalizacija) prasidedantį bangelės simboliu (~).**
  - Konstruktorius neturi nei tipo grąžinimo nei input parametrų.
- **Koks svarbus skirtumas tarp konstruktorių ir destruktoriaus?**

## Destruktoriai (2)

```
class Vector {
private:
    int sz;
    double* elem;
public:
    // konstruktoriai: įgyja resursus
    Vector() : sz(0), elem(new double[sz]) {}
    Vector(int s) : sz{ s }, elem{ new double[sz] } { // C++11 stilius
        std::fill_n(elem, s, 0.0); // inicializuojame
    }
    Vector(int s, double val) : sz(s), elem(new double[sz]) {
        std::fill_n(elem, s, val); // užpildome val reikšmėmis
    }
    ~Vector() { delete[] elem; } // destruktoriaus: atlaisvina resursus
};
```



## Destruktoriai (3)

The technique of acquiring resources in a constructor and releasing them in a destructor, known as Resource Acquisition Is Initialization or RAII.

- Bjarne Stroustrup
- Tokiu būdu galime visiškai išvengti pavojingo **new** ir **delete** naudojimo (resursų leakinimo), paslepiant jį klasės realizacijoje - **C++ Garbage collector'ius**
- Tokiu būdu klasės objektas elgiasi kaip **build-in** tipo kintamasis.



Jesslyn  
@jtannady

Follow



I'm from the island of Java, Indonesia.

I am the Java Garbage Collector.



# "this" rodyklė (pointer) (1)

— Tradicinis OOP naujokų klausimas yra: **kai iškviečiama klasės nario funkcija, kaip C++ žino, kuris objektas ją iškvietė?**

```
class Vector {  
    int sz;  
public:  
    Vector(int s) : sz{s} {}  
    void setSize(int size) { sz = size; }  
};
```

```
int main() {  
    Vector v1{1}, v2{2};  
    v1.setSize(5); // Kaip C++ žino, kad setSize() yra v1, o ne v2 funkcija?  
}
```

## "this" rodyklė (pointer) (2)

— **Atsakymas:** C++ naudoja papildomą užslėptą rodyklę "this"!

```
class Vector {  
    int sz;  
public:  
    Vector(int s) : sz{s} {}  
    void setSize(int size) { sz = size; }  
};
```

```
int main() {  
    Vector v1{1}, v2{2};  
    v1.setSize(5); // Kiek input parametru turi funkcija setSize()?  
}
```

## "this" rodyklė (pointer) (3)

— Atrodo, kad `setSize()` turi tik vieną input parametą:

```
v1.setSize(5);
```

— Tačiau tikrovėje kompiliatorius šį kodą paverčia į:

```
setSize(&v1, 5);
```

— t.y., konvertuoja į tradicinę funkciją, kurios papildomas parametras yra nuoroda (`&v1`) į objektą!



## "this" rodyklė (pointer) (4)

- Bet nesutampa su nario funkcijos deklaracija: `void setSize(int) !`
- Vadinasi kompiliatorius ir šią member nario funkciją:

```
void setSize(int size) { sz = size; }
```

- konvertuoja į:

```
void setSize(Vector* const this, int size) { this->sz = size; }  
// setSize(&v1, 5);
```

- paslėpta **this** const tipo rodyklė saugo objekto adresą.

## "**this**" rodyklė (pointer) (5)

- Galiausiai klasės viduje kintamieji (kitos funkcijos) taip pat turi būti pakoreguoti, kad jie atspindėtų objekto kintamuosius/funkcijas.
- Tai pasiekama pridėjus "**this->**" prefiksą kiekvienam iš jų.
- Todėl klasės nario funkcijos `setSize()` viduje kintamasis `sz` buvo konvertuotas į `this->sz`.
- Kadangi **this** saugo objekto adresą, todėl `this->sz` yra ekvivalentu: `v1.sz`.

## "this" rodyklė (pointer) (6)

- Tai kiek iš viso **this** rodyklių (pointer) egzistuoja?
- Kiekviena objekto nario funkcija turi \*this rodyklę, kurios adresas yra objekto, kuriam priklauso ta nario funkcija, adresas:

```
int main() {  
    Vector v1{1}; // *this = &v1 Vector konstruktoriaus viduje  
    Vector v2{2}; // *this = &v2 Vector konstruktoriaus viduje  
    v1.setSize(5); // *this = &v1 nario funkcijos viduje  
    v2.setSize(5); // *this = &v2 nario funkcijos viduje  
}
```

## "this" rodyklė (pointer) (7)

- Dažniausiai apie visa tai, jums nereikia galvoti, nes kompiliatorius viską tą atlieka automatiškai!
- Tačiau yra keli atvejai, kuomet tai gali būti labai naudinga. Pirma:

```
class Vector {  
    int size;  
    public:  
    Vector(int s) : size{s} {}  
    void setSize(int size) { size = size; } // Painu?  
};
```

- Ar gali funkcijos parametro ir nario kintamojo vardai sutapti?

## "this" rodyklė (pointer) (8)

```
class Vector {  
    int size;  
    public:  
    Vector(int s) : size{s} {}  
    void setSize(int size) { this->size = size; } // 0 kaip dabar?  
};
```

- Patarimas: vengti tų pačių vardu, o nario kintamųjų vardus išskirti pagal tam tikrą stilių, pvz.
  - prirašant raidę "m" (**member**) vardų pradžioje: int mSize;
  - ar "\_" vardų pabaigoje: int size\_;

## "this" rodyklė (pointer) (9)

- Antra: gali būti naudinga, kad objekto nario funkcija gražintų paties objekto adresą.
- Ar esame mes jau tai kažkur naudoję? Jei taip, tai koku tikslu?

```
std::cout << "Man patinka" << "ketvirtadienis!" << std::endl;
```

- čia `std::cout` yra objektas, o `operator<<` yra nario funkcija, kuri dirba su tuo objektu.

# Klasės realizacija ir header failai (1)

Vector.h faile deklaracijos ir trumpų funkcijų realizacijos:

```
#ifndef VECTOR_H
#define VECTOR_H

class Vector {
private:
    int sz;
    double* elem;
public:
    Vector() : sz(0), elem(new double[sz]) {}
    Vector(int s);
    Vector(int s, double val);
    ~Vector() { delete[] elem; }
    void setElem(int idx, double val);
};

#endif
```

# Klasės realizacija ir header failai (2)

Vector.cpp faile ilgų funkcijų realizacijos, prasidedančios:

```
#include "Vector.h"

// Visur reikia nurodyti klasės scop'ą: Vector::
Vector::Vector(int s) : sz{ s }, elem{ new double[sz] }{
    std::fill_n(elem, s, 0.0);
}

Vector::Vector(int s, double val) : sz(s), elem(new double[sz]) {
    std::fill_n(elem, s, val);
}

void Vector::setElem(int idx, double val) {
    if (idx < 0 || idx >= sz) return;
    elem[idx] = val;
}
```



# Klausimai ?

