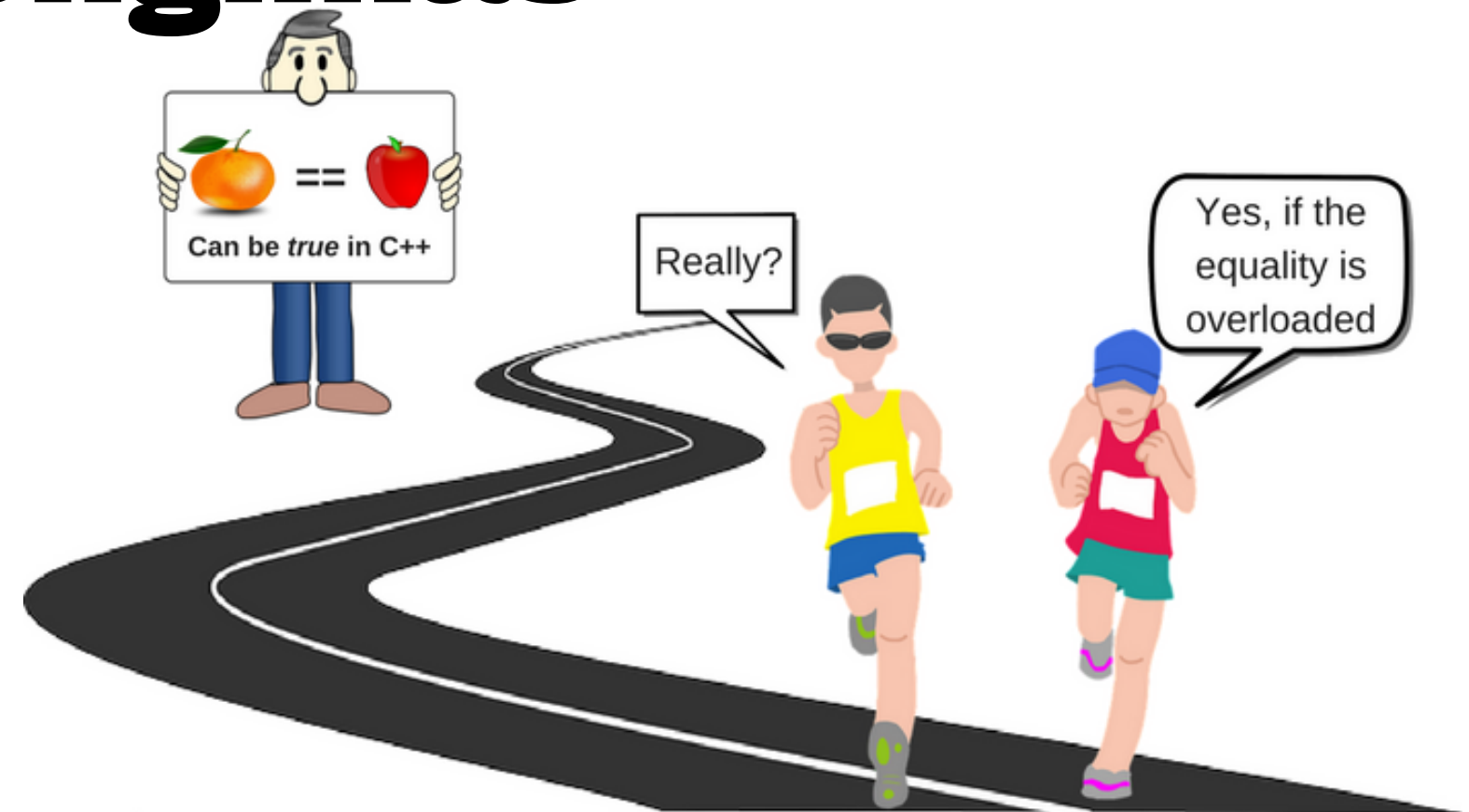


Objektinis Programavimas

Operatorių persidengimas



Iš čia kylama
į žvaigždes



Turinys

1. Kam reikalingas operatorių persidengimas?
2. Kompleksinių skaičių tipas
3. Aritmetiniai operatoriai
 - operator+, operator-, operator*, operator/
4. Skirtingi to paties operatoriaus realizavimo būdai
5. Ivedimo/išvedimo (**input/output**) operatoriai:
 - operator<< ir operator>>
6. Operatorių persidengimo rekomendacijos

Kam reikalingas operatorių persidengimas ? (1)

- Mes jau žinome, kad C++ kalboje funkcijos gali persidengti!
- Tai leidžia turėti kelias funkcijas su tuo pačiu pavadinimu, tačiau tinkamą dirbti su skirtingais duomenų tipais (unikalus prototipas):

```
double galBalas(double, double);
```

```
double galBalas(double, const std::vector<double>&);
```

```
double galBalas(double egzaminas, const vector<double>&,  
double (*)(vector<double>) = mediana)
```

```
double galBalas(const Studentas&, double (*) (vector<double>)  
= mediana);
```

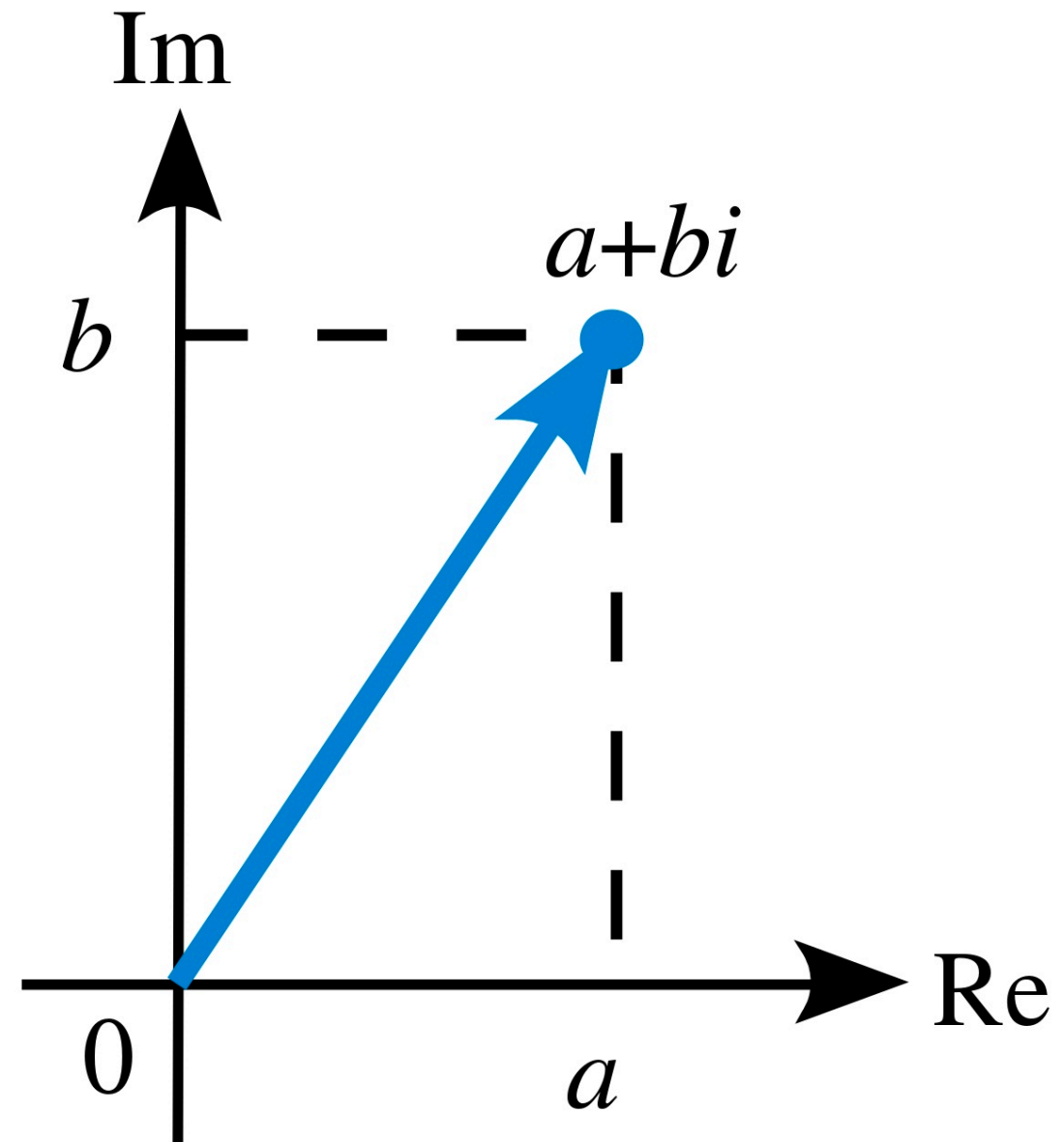
Kam reikalingas operatorių persidengimas ? (2)

- C++ kalboje operatoriai realizuoti kaip funkcijos!
- Panaudodami funkcijų persidengimą galime apsirašyti persidengiančius (**overloaded**) operatorius, dirbančius su įvairiais duomenų tipais, tame tarpe ir vartotojo sukurtais tipais (struktūros, klasės).
- Kaip pamatysime, galimi bent **trys skirtingi** operatorių persidengimo realizavimo būdai:
 - naudojant nario (**member**) funkcijas
 - naudojant draugiškas (**friend**) funkcijas
 - naudojant įprastas funkcijas.

Konkrečios klasės (**Concrete classes**)

- Pagrindinis konkrečių klasių bruožas yra tai, kad ji elgiasi kaip integruotas (**built-in**) C++ tipas.
- Pavyzdžiui, kompleksinių skaičių tipas elgiasi panašiai kaip **built-in** Integer tipas, išskyrus tai, kad turi savo semantiką ir operacijų rinkinius.
- Šio tipo klasių išskirtinė charakteristika yra tai, kad jų reprezentacija yra kartu ir klasės apibrėžimas. **Ką tai suteikia?**

Kompleksiniai skaičiai (1)



Kompleksiniai skaičiai (2)

Supaprastinta `std::complex` versija

```
// Complex.h (Complex.hpp) header failas
class Complex {
    double re, im; // realioji ir menamoji dalis
public:
    // konstruktorius iš dviejų skaliarų
    Complex(double r, double i) :re{r}, im{i} {}
    // konstruktorius iš vieno skaliaro
    Complex(double r) :re{r}, im{0} {}
    // default konstruktorius: {0, 0}
    Complex() :re{0}, im{0} {}
    double real() const { return re; }
    void real(double r) { re = r; }
    double imag() const { return im; }
    void imag(double i) { im = i; }
};
```

Kompleksiniai skaičiai (3)

Kompleksinių skaičių suma (1)

```
#include <iostream>
#include "Complex.h"

int main() {
    Complex a {1.0, 1.0}; // sukonstruojame {1.0, 1.0}
    Complex b {2.0};      // sukonstruojame {2.0, 0.0}
    a + b;                // Ką gausime čia?
}
```


Kompleksiniai skaičiai (4)

Kompleksinių skaičių suma (2)

```
#include <iostream>
#include "Complex.h"

int main() {
    Complex a {1.0, 1.0}; // sukonstruojame {1.0, 1.0}
    Complex b {2.0};      // sukonstruojame {2.0, 0.0}
    a + b;                // Ką gausime čia?
}
```

main.cpp: error: no match for 'operator+' (operand types are 'Complex' and 'Complex')

Aritmetiniai operatoriai

- Vieni iš dažniausiai naudojamų operatorių C++ kalboje yra **aritmetiniai operatoriai**:
 - plus operatorius (+)
 - minus operatorius (-)
 - daugybos operatorius (*)
 - dalybos operatorius (/)

operator+ realizacija

Naudojant friend funkciją

```
class Complex {
    double re, im; // realioji ir menamoji dalis: du double
public:
    Complex(double r, double i) :re{r}, im{i} {}
    Complex(double r) :re{r}, im{0} {}
    Complex() :re{0}, im{0} {}
    // sudėti: Complex + Complex naudojant friend funkciją
    friend Complex operator+(const Complex &a, const Complex &b);
};

// ši funkcija yra ne nario bet Complex klasės friend funkcija
// - galėtų būti apibrėžta (realizuota) ir Complex klasės viduje
// - nereikia friend žodelio prieš realizaciją už klasės ribų
Complex operator+(const Complex &a, const Complex &b) {
    // naudojame: Complex(double, double) konstruktorių ir operator+(double, double)
    // pasiekiamo privačias: re ir im reikšmes tiesiogiai nes tai yra friend funkcija!
    return Complex {a.re + b.re, a.im + b.im};
}
```

operator+() realizacija

Naudojant tradicinę (ne friend) funkciją

```
class Complex {
    double re, im; // realioji ir menamoji dalis
public:
    Complex(double r, double i) :re{r}, im{i} {}
    Complex(double r) :re{r}, im{0} {}
    Complex() :re{0}, im{0} {}
    double real() const { return re; }
    double imag() const { return im; }
};

// ši funkcija yra ne nario ir ne friend funkcija!
Complex operator+(const Complex &a, const Complex &b) {
    // naudojame: Complex konstruktorių ir operator+(double, double)
    // pasiekiamo: re ir im narius per public interfeisą!
    return Complex {a.real() + b.real(), a.imag() + b.imag()};
}
```

operator+() realizacija

Per klasės nario funkciją

```
class Complex {
    double re, im; // realioji ir menamoji dalis: du double
public:
    Complex(double r, double i) :re{r}, im{i} {}
    Complex(double r) :re{r}, im{0} {}
    Complex() :re{0}, im{0} {}
    Complex operator+(const Complex &a);
};

// Complex nario funkcijos realizacija
Complex Complex::operator+(const Complex &a) {
    return Complex {re + a.re, im + a.im};
}
```

Kompleksiniai skaičiai (5)

Kompleksinių skaičių suma (3)

```
#include <iostream>
#include "Complex.h" // su friend operator+(Complex, Complex)

int main() {
    Complex a {1.0, 1.0}; // sukonstruojame {1.0, 1.0}
    Complex b {2.0};      // sukonstruojame {2.0, 0.0}
    a + b;                // o ką dabar gausime?
}
```

Kompleksiniai skaičiai (6)

```
#include <iostream>
#include "Complex.h"          // su friend Complex operator+()

int main() {
    Complex a {1.0, 1.0};    // sukonstruojame {1.0, 1.0}
    Complex b {2.0};         // sukonstruojame {2.0, 0.0}
    a + b;                   // viskas OK
}
```

Kompleksiniai skaičiai (7)

```
#include <iostream>
#include "Complex.h"          // su friend Complex operator+()

int main() {
    Complex a {1.0, 1.0};    // sukonstruojame {1.0, 1.0}
    Complex b {2.0};         // sukonstruojame {2.0, 0.0}
    a + b;                   // viskas OK
    std::cout << a + b << std::endl; // O ką gausime dabar?
}
```

main.cpp: error:
no match for 'operator<<' (operand types are 'std::ostream {aka std::basic_ostream<char>}' and 'Complex')

Kompleksinių skaičių spausdinimas

Susikuriame tam dedikuotą `print()` nario funkciją

```
class Complex {
    double re, im; // realioji ir menamoji dalis
public:
    Complex(double r, double i) :re{r}, im{i} {}
    Complex(double r) :re{r}, im{0} {}
    Complex() :re{0}, im{0} {}
    friend Complex operator+(const Complex &a, const Complex &b);
    void print() const { // atspausdinti kompleksinį skaičių: a + bi forma
        std::cout << re << " + " << im << "i\n";
    }
};
```

Kompleksiniai skaičiai (8)

```
#include <iostream>
#include "Complex.h"          // su friend Complex operator+()

int main() {
    Complex a {1.0, 1.0};    // sukonstruojame {1.0, 1.0}
    Complex b {2.0};         // sukonstruojame {2.0, 0.0}
    a.print();
    b.print();
    // a + b;                // 0 kaip atspausdinti šią sumą?
}
```

Kompleksiniai skaičiai (9)

```
#include <iostream>
#include "Complex.h"          // su friend Complex operator+()

int main() {
    Complex a {1.0, 1.0};    // sukonstruojame {1.0, 1.0}
    Complex b {2.0};         // sukonstruojame {2.0, 0.0}
    a.print();
    b.print();
    Complex c = a + b;       // arba: (a+b).print();
    c.print();
}
```

Išvedimo operator<< realizacija

```
class Complex {
    double re, im; // realioji ir menamoji dalis
public:
    Complex(double r, double i) :re{r}, im{i} {}
    friend Complex operator+(const Complex &a, const Complex &b) {
        return Complex {a.re + b.re, a.im + b.im};
    }
    friend std::ostream& operator<<(std::ostream& out, const Complex &a) {
        out << a.re << (a.im < 0 ? " - " : " + ") << abs(a.im) << "i\n";
        return out;
    }
};

int main() {
    Complex a{1, 1}, b{2, 2};
    std::cout << a + b << std::endl; // Atspausdiname mums įprastu būdu
}
```

Kodėl operator<< grąžina std::ostream& tipą?

- Kompiliatoriai neleidžia nukopijuoti srauto std::ostream.
- Grąžindami kaip nuorodą std::ostream&, įgaliname galimybę atspausdinti kelias išraiškas naudojant operator<<

```
int main() {  
    Complex a {1.0, 2.0};  
    Complex b {2.0, 1.0};  
    std::cout << a << " " << b << std::endl;  
}
```

Įvedimo operator>> realizacija

```
class Complex {
    double re, im; // realioji ir menamoji dalis
public:
    Complex(double r, double i) :re{r}, im{i} {}
    friend Complex operator+(const Complex &a, const Complex &b) {
        return Complex {a.re + b.re, a.im + b.im};
    }
    friend std::ostream& operator<<(std::ostream& out, const Complex &a) {
        out << a.re << (a.im < 0 ? " - " : " + ") << abs(a.im) << "i\n";
        return out;
    }
    // Koks esminis skirtumas lyginant su operator<< ?
    friend std::istream& operator>>(std::istream& in, Complex &a) {
        in >> a.re >> a.im;
        return in;
    }
};

int main() {
    std::cout << "Įveskite kompleksinį skaičių: pirma realią, po to menamą dalį: \n";
    Complex c;
    std::cin >> c;
    std::cout << "Jūs įvedėte: " << c << std::endl;
}
```

operator- realizacija

Naudojant friend funkciją

```
class Complex {  
    double re, im; // realioji ir menamoji dalis  
public:  
    Complex(double r, double i) :re{r}, im{i} {}  
    Complex(double r) :re{r}, im{0} {}  
    Complex() :re{0}, im{0} {}  
    friend Complex operator+(const Complex &a, const Complex &b);  
    friend Complex operator-(const Complex &a, const Complex &b);  
};  
  
Complex operator-(const Complex &a, const Complex &b) {  
    return Complex {a.re - b.re, a.im - b.im};  
}
```

Jungtiniai operatoriai: +=, -=, *=

```
class Complex {
    double re, im; // realioji ir menamoji dalis
public:
    Complex(double r, double i) :re{r}, im{i} {}
    friend Complex operator+(const Complex &a, const Complex &b) {
        return Complex {a.re + b.re, a.im + b.im};
    }
    Complex& operator+=(Complex c) { re += c.re; im += c.im; return *this; }
    Complex& operator-=(Complex c) { re -= c.re, im -= c.im; return *this; }
    Complex& operator*=(Complex c); // Realizuokite patys!
};

int main() {
    Complex a{1, 1}, b{2, 2};
    std::cout << a + b << std::endl; // Ką čia gausime?
    a += b;                          // Ekvivalentu: a = a + b;
    std::cout << a << std::endl;     // O ką dabar čia gausime?
}
```


Kodėl reikia gražinti Complex&?

```
class Complex {
    double re, im; // realioji ir menamoji dalis
public:
    Complex(double r, double i) :re{r}, im{i} {}
    friend Complex operator+(const Complex &a, const Complex &b) {
        return Complex {a.re + b.re, a.im + b.im};
    }
    // Kas būtų jeigu vietoj Complex& naudotume Complex, t.y.?
    Complex operator+=(Complex c) { re += c.re; im += c.im; return *this; }

    Complex& operator-=(Complex c) { re -= c.re, im -= c.im; return *this; }
    Complex& operator*=(Complex c); // Realizuokite patys!
};

int main() {
    // Panagrinėkime ką gausime šiam pvz. abiem atvejams:
    Complex a{1,1}, b{2,2}, c{3,3};
    std::cout << ((a += b) += c) << std::endl;
    std::cout << a << std::endl;
}
```

Jungtinio operatoriaus *= realizacija (1)

```
class Complex {
    double re, im; // realioji ir menamoji dalis
public:
    Complex(double r, double i) :re{r}, im{i} {}
    friend Complex operator+(const Complex &a, const Complex &b) {
        return Complex {a.re + b.re, a.im + b.im};
    }
    Complex& operator+=(Complex c) { re += c.re; im += c.im; return *this; }
    Complex& operator-=(Complex c) { re -= c.re; im -= c.im; return *this; }
    Complex& operator*=(Complex c) { // Ar viskas čia gerai?
        re = re*c.re - im*c.im;
        im = re*c.im + im*c.re;
        return *this;
    }
};
```

Jungtinio operatoriaus *= realizacija (2)

```
class Complex {  
    double re, im; // realioji ir menamoji dalis  
public:  
    Complex(double r, double i) :re{r}, im{i} {}  
    friend Complex operator+(const Complex &a, const Complex &b) {  
        return Complex {a.re + b.re, a.im + b.im};  
    }  
    Complex& operator+=(Complex c) { re += c.re; im += c.im; return *this; }  
    Complex& operator-=(Complex c) { re -= c.re, im -= c.im; return *this; }  
    Complex& operator*=(Complex c) {  
        double real = re*c.re - im*c.im;  
        im = re*c.im + im*c.re;  
        re = real;  
        return *this;  
    }  
};
```

operator+ realizacija panaudojant operator+=

```
class Complex {  
    double re, im; // realioji ir menamoji dalis: du double  
public:  
    Complex(double r, double i) :re{r}, im{i} {}  
    Complex& operator+=(Complex c) { re += c.re, im += c.im; return *this; }  
};  
  
// panaudojame realizuotą jungtinį operatorių: operator+=  
Complex operator+(Complex a, const Complex& b) { // o kodėl ne const Complex& a?  
    return a += b; // Kviečiamas: operator+=  
}
```

Nebaigta Complex klasės realizacija

```
class Complex {
    double re, im; // realioji ir menamoji dalis
public:
    Complex(double r, double i) :re{r}, im{i} {}
    double real() const { return re; }
    double imag() const { return im; }
    Complex& operator+=(Complex c) { re += c.re, im += c.im; return *this; }
    Complex& operator-=(Complex c) { re -= c.re, im -= c.im; return *this; }
    Complex& operator*=(Complex c); // Realizuota kitur

    Complex operator+(Complex a, Complex b) { return a += b; }
    Complex operator-(Complex a, Complex b) { return a -= b; }
    Complex operator-(Complex a) const { // vienanaris minus
        return { -a.real(), -a.imag() }; // neiginys
    }
    Complex operator*(Complex a, Complex b) { return a *= b; }
    bool operator==(Complex a, Complex b) {
        return a.real() == b.real() && a.imag() == b.imag();
    }
    bool operator!=(Complex a, Complex b) { return !(a==b); }
```

Reziumė

- Vartotojų apibrėžti "perkrauti" operatoriai turi būti naudojami atsargiai ir turėti aprašytiems objektams prasmę.
- C++ yra fiksuota sintaksė, todėl negalima apibrėžti pvz. vienanario "/" operatoriaus.
- Negalima pakeisti egzistuojančių operatorių prasmės integruotiems (built-in) tipams. Todėl yra neįmanoma pvz. Int'ams "perkrauti" + operatoriaus, kad jis atliktų Int'ų atmintį.

Operatorių persidengimo rekomendacijos

Kada realizuoti naudojant friend funkcijas, kada tradicines, o kada klasės nario?

- Jei įmanoma (nepridedant tik dėl šio tikslo **get**'er funkcijų), vietoje friend funkcijų operatorių persidengimui realizuoti naudokite tradicines funkcijas. **Kodėl?**
- friend funkcijos turi tiesioginį priėjimą prie klasės private kintamųjų reikšmių, o tas yra "**pavojingiau**", negu suteikiant priėjimą prie jų vien tik per public interfeisą.
- Tačiau ne visi operatoriai gali būti realizuoti kaip **friend** funkcijos: priskyrimo (**=**), indeksavimo (**[]**), funkcijos iškvietimo (**()**) ir (**->**) operatoriai **turi būti realizuoti** (nes taip C++ nusprendė 😊) **kaip nario funkcijos**.
- Kita vertus įvedimo/išvedimo operatoriai negali būti realizuoti kaip nario funkcijos, nes kairinis operandas yra **std::ostream**, o ne klasės tipo!

Klausimai !?

