# Objektinis Programavimas

## Operatorių persidengimas


Iš čia kylama
į žvaigždes

© Remigijus Paulavičius @RemigPau, Vilniaus Universitetas - 2018

# Turinys

# Kam reikalingas operatorių persidengimas ? (1)

— Mes jau žinome, kad C++ kalboje funkcijos gali persidengti!

— Tai leidžia turėti kelias funkcijas su tuo pačiu pavadinimu, bet 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++ operatoriai realizuoti kaip funkcijos!
- Panaudodami funkcijų persidengimą galime apsirašyti persidengiančius (**overloaded**) operatorius, dirbančius su įvairiais duomenų tipais, tame tarpe ir vartotojo sukurtais (struktūros, klasės).
- Egzistuoja *trys skirtingi* operatorių persidengimo realizavimo būdai:
  - naudojant nario (**member**) funkcijas
  - naudojant draugiškas (**friend**) funkcijas
  - naudojant įprastas funkcijas.

# Kompleksiniai skaičiai (1)

## Konkrečioji klasė (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** `int`, išskyrus tai, kad turi savo semantiką ir operacijų rinkinius.

# Kompleksiniai skaičiai (2)

**Supaprastinta `std::complex` versija**

```
// Complex.h (Complex.hpp) faile
class Complex {
  double re, im;  // realioji ir menamoji dalis: du double
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
  Complex() :re{0}, im{0} {}
};
```

# Kompleksiniai skaičiai (3)

```
#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)

```
#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**:
  - plius 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 friend funkcija
// galėtų būti apibrėžta (realizuota) ir Complex klasės viduje
// nereikia friend žodelio prieš realizaciją už klasės
Complex operator+(const Complex &a, const Complex &b) {
  // naudojame: Complex konstruktorių ir operator+(double, double)
  // pasiekiame: re ir im tiesiogiai nes tai yra friend funkcija!
  return Complex {a.re + b.re, a.im + b.im};
}
```

# Kompleksiniai skaičiai (5)

```
#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;                 // 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;                 // čia viskas OK
  std::cout << a + b << std::endl;  // O ką gausime dabar?
}
```

# Kompleksiniai skaičiai (7)

```cpp
#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;                 // o ką dabar gausime?
  std::cout << a + b << std::endl;  // O ką tokiu atveju?
}
```

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

# Kompleksinių skaičių spausdinimas

## Nenaudojant operator<<

```
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} {}
  friend Complex operator+(const Complex &a, const Complex &b);
  void print() const { // atspausdinti kompleksinį skaičių forma: a + bi
    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;     // O kaip atspausdinti šį?
}
```

# 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;
  c.print();
}
```

# 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);
  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};
}
```

# operator+ realizacija

## Naudojant tradicinę (ne 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} {}
  void print() const { std::cout << re << " + " << im << "i\n"; }
  double getReal() const { return re; }
  double getImag() 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)
  // pasiekiame: re ir im narius per public interfeisą!
  return Complex {a.getReal() + b.getReal(), a.getImag() + b.getImag()};
}
```

# Operatorių persidengimas: `friend` funkcijos vs. tradicinės

— Jei įmanoma (pvz. nepridedant tik dėl šio tikslo **get**'er funkcijų), operatorių persidengimui realizuoti naudokite tradicines funkcijas vietoje `friend` funkcijų. Kodėl?

— `friend` funkcijos turi tiesioginį priėjimą prie `private` reikšmių, o tas yra "*pavojingiau*", negu suteikiant priėjimą prie jų vien tik per `public` interfeisą.

# Operatorių persidengimas naudojant klasės nario funkcijas

## Jungtiniai operatoriai: +=, -= ir kt.

```
class Complex {
  double re, im;  // realioji ir menamoji dalis: du double
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 << "i\n";
    return out;
  }
};

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 čia gausime?
}
```

# Operatorių persidengimas naudojant klasės nario funkcijas

**Jungtiniai operatoriai +=, -=**

```
class Complex {
  double re, im;  // realioji ir menamoji dalis: du double
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; }
};

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?
}
```

# operator+ realizacija panaudojant jungtinį 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; }
};

// ši funkcija yra ne nario ir ne friend funkcija!
Complex operator+(Complex a, Complex b) {  // o kodėl ne const Complex&?
  return a += b;  // Kviečiamas: operator+=
}
```

# Kitų `Complex` klasės aritmetinių operatorių realizacija

```
class Complex {
  double re, im;  // realioji ir menamoji dalis: du double
public:
  Complex(double r, double i) :re{r}, im{i} {}
  double getReal() const { return re; }
  double getImag() 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); // pabandykite realizuoti!
};

Complex operator+(Complex a, Complex b) { return a += b; }
Complex operator-(Complex a, Complex b) { return a -= b; }
Complex operator-(Complex a){              // vienanaris minus
  return { -a.getReal(), -a.getImag() }; // neiginys
}
Complex operator*(Complex a, Complex b) { return a *= b; }
bool operator==(Complex a, Complex b) {
     return a.getReal() == b.getReal() && a.getImag() == b.getImag();
}
bool operator!=(Complex a, Complex b); // pabandykite jį realizuoti!
```

# operator<< realizacija

```
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} {}
  friend Complex operator+(const Complex &a, const Complex &b);
  void print() const { // atspausdinti kompleksinį skaičių forma: a + bi
    std::cout << re << " + " << im << "i\n";
  }
  friend std::ostream& operator<<(std::ostream&, const Complex&);
};

std::ostream& operator<<(std::ostream& out, const Complex &a) {
  out << a.re << " + " << a.im << "i\n";
  return out;
}
```

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

— Negalima grąžinti `std::ostream` tipo reikšmės.

— Grąžindami nuorodą `std::ostream&` įgaliname galimybę atspausdinti kelias išraiškas naudojančias `operator<<`, pvz.:

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

# Kompleksiniai skaičiai (10)

**Spausdiname (`std::cout`) kompleksinius skaičius "įprastu" būdu**

```
#include <iostream>
#include "Complex.h"  // friend std::ostream& operator<<

int main() {
  Complex a {1.0, 1.0};  // sukonstruojame (1.0, 1.0)
  a.print();                          // atspausdina a: 1 + 1i
  std::cout << a << std::endl;  // atspausdina a: 1 + 1i
}
```

# Klausimai !?