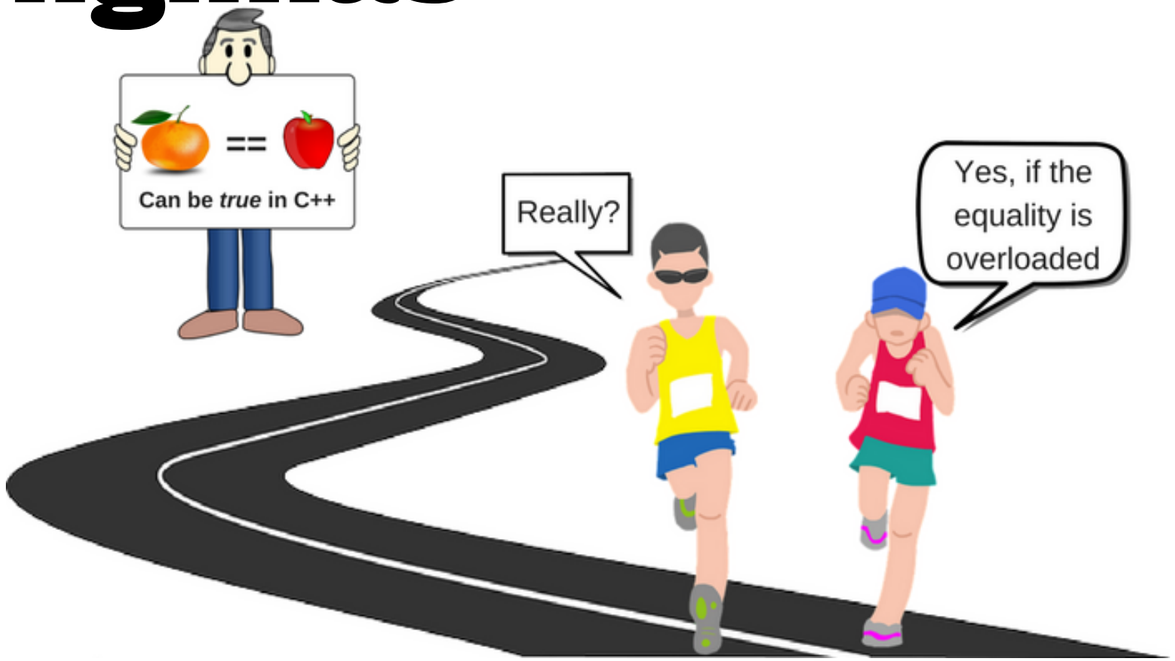
Objektinis Programavimas

Operatorių persidengimas



Iš čia kylama į žvaigždes

Turinys

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.

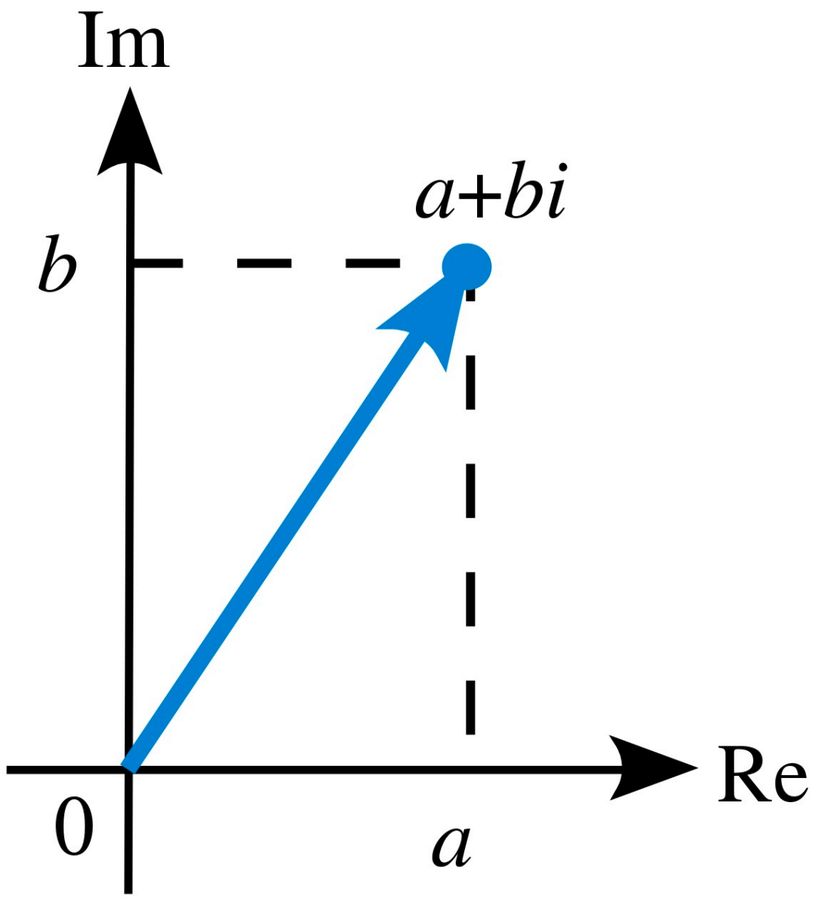
Konkrečios klasės (Concrete classes)

— 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**:
  — 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 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)
  // pasiekiame privačias: re ir im reikšmes tiesiogiai nes tai yra friend funkcija!
  return Complex {a.re + b.re, a.im + b.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;               // O 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 << "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?

— Grąžindami nuorodą `std::ostream&` įgaliname galimybę atspausdinti kelias grandinines išraiškas naudojant `operator<<`

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

Alternatyvi `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} {}
  void print() const { std::cout << re << " + " << im << "i\n"; }
  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)
  // pasiekiame: re ir im narius per public interfeisą!
  return Complex {a.real() + b.real(), a.imag() + b.imag()};
}
```

Operatorių realizavimas: `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 klasės `private` kintamųjų reikšmių, o tas yra "pavojingiau", negu suteikiant priėjimą prie jų vien tik per `public` interfeisą.

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};
}
```

Operatorių realizavimas per klasės nario funkcijas

Jungtiniai operatoriai: += ir -=

```
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; }
};

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 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
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); // Realizuokite patys!

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.

Klausimai !?