Informatika 3.
5. előadás: Operator overload, friends

Kovács Kristóf

Budapesti Műszaki és Gazdaságtudományi Egyetem

2024-04-16

Operator overloading

```
int main(void) {
  Complex a;
  Complex b = Complex(1,2);
  Complex c = a * b;
  Complex d = b + b;

  cout << "a: " << a << endl;
  cout << "b: " << b << endl;
  cout << "c: " << c << endl;
  cout << "d: " << d << endl;

  cout << b + c << endl;

  cout << b.abs() << endl;

  return 0;
}
```

Operator overloading

```
int main(void) {
  Complex a;
  Complex b = Complex(1,2);
  Complex c = a * b;
  Complex d = b + b;

  cout << "a: " << a << endl;
  cout << "b: " << b << endl;
  cout << "c: " << c << endl;
  cout << "d: " << d << endl;

  cout << b + c << endl;

  cout << b.abs() << endl;

  return 0;
}
```

Más szóval, szeretnénk átdefiniálni a műveleti jeleket/operátorokat.

Operator overloading

Eddigi megoldásunk az összeadásra:

```
Complex Complex::add(Complex other) {
  return Complex(this->re + other.re, this->im + other.im);
}
```

Operator overloading

Eddigi megoldásunk az összeadásra:

```
Complex Complex::add(Complex other) {
  return Complex(this->re + other.re, this->im + other.im);
}
```

Megoldás operator overload-al:

```
Complex Complex::operator+(Complex other) {
  return Complex(this->re + other.re, this->im + other.im);
}
```

Operator overloading

Eddigi megoldásunk az összeadásra:

```
Complex Complex::add(Complex other) {
  return Complex(this->re + other.re, this->im + other.im);
}
```

Megoldás operator overload-al:

```
Complex Complex::operator+(Complex other) {
  return Complex(this->re + other.re, this->im + other.im);
}
```

Az **operator** egyben egy kulcsszó és a függvény/metódusnevek része. Amilyen operátort teszünk a kulcsszó után azt tudjuk definiálni. Pár példa:

- `operator+`
- `operator-`
- `operator*`
- `operator/`
- `operator^`
- `operator=`
- `operator==`
- `operator<`
- `operator<<`
- `operator()`
- `operator[]`
- `operator++`

Operator overloading

- Az alapelve az operátoroknak ugyanaz. Bal és jobb értékeken operálnak és visszadhatnak valami értéket.

Operator overloading

- Az alapelve az operátoroknak ugyanaz. Bal és jobb értékeken operálnak és visszadhatnak valami értéket.
- Amikor osztályon belül írunk operator metódust a balérték mindig az adott objektum. Például az eddigi példánknál egy `Complex` objektum:

```
Complex Complex::operator+(Complex other) {
  return Complex(this->re + other.re, this->im + other.im);
}
```

Operator overloading

- Az alapelve az operátoroknak ugyanaz. Bal és jobb értékeken operálnak és visszadhatnak valami értéket.
- Amikor osztályon belül írunk operator metódust a balérték mindig az adott objektum. Például az eddigi példánknál egy `Complex` objektum:

```
Complex Complex::operator+(Complex other) {
  return Complex(this->re + other.re, this->im + other.im);
}
```

- A következő két kódsor ekvivalens, feltéve, hogy *c* és *d* Complex objektumok:

```
Complex e = c + d;
Complex e = c.operator+(d);
```

Operator overloading

- Használhatunk beépített típusokat is, például ha egy komplex és egy valós számot szeretnénk összeadni:

```
Complex Complex::operator+(float other) {
  return Complex(this->re + other, this->im);
}
```

Operator overloading

- Használhatunk beépített típusokat is, például ha egy komplex és egy valós számot szeretnénk összeadni:

```
Complex Complex::operator+(float other) {
  return Complex(this->re + other, this->im);
}
```

- Így már működik a következő:

```
Complex f = c + 5.3;
```

Operator overloading

- Használhatunk beépített típusokat is, például ha egy komplex és egy valós számot szeretnénk összeadni:

```
Complex Complex::operator+(float other) {
  return Complex(this->re + other, this->im);
}
```

- Így már működik a következő:

```
Complex f = c + 5.3;
```

- De ez még nem:

```
Complex g = 5.3 + c;
```

Operator overloading

- Használhatunk beépített típusokat is, például ha egy komplex és egy valós számot szeretnénk összeadni:

```
Complex Complex::operator+(float other) {
  return Complex(this->re + other, this->im);
}
```

- Így már működik a következő:

```
Complex f = c + 5.3;
```

- De ez még nem:

```
Complex g = 5.3 + c;
```

- A probléma, hogy a bal érték egy beépített típus. Az `(5.3).operator+(c)` hívásnak nincs értelme, mert az 5.3 nem egy objektum, nincsenek metódusai.

Operator overloading

- Használhatunk beépített típusokat is, például ha egy komplex és egy valós számot szeretnénk összeadni:

```
Complex Complex::operator+(float other) {
  return Complex(this->re + other, this->im);
}
```

- Így már működik a következő:

```
Complex f = c + 5.3;
```

- De ez még nem:

```
Complex g = 5.3 + c;
```

- A probléma, hogy a bal érték egy beépített típus. Az `(5.3).operator+(c)` hívásnak nincs értelme, mert az 5.3 nem egy objektum, nincsenek metódusai.
- A megoldás egy másik módja az operator overloadnak:

```
Complex operator+(float left, Complex right) {
  return Complex(left + right.re, right.im);
}
```

Két paraméteres operator overloading

- Használhatjuk az operator overload-ot osztályokon kívül is. Ekkor 2 paramétert kell megadnunk, a bal és a jobb értéket (ennek a függvénynek a Complex osztályon kívül kell lennie):

```
Complex operator+(float left, Complex right) {
  return Complex(left + right.re, right.im);
}
```

Két paraméteres operator overloading

- Használhatjuk az operator overload-ot osztályokon kívül is. Ekkor 2 paramétert kell megadnunk, a bal és a jobb értéket (ennek a függvénynek a Complex osztályon kívül kell lennie):

```
Complex operator+(float left, Complex right) {
  return Complex(left + right.re, right.im);
}
```

- Hasonlóan a korábbihoz, ez a két sor ekvivalens:

```
Complex g = 5.3 + d;
Complex g = operator+(5.3, d);
```

Két paraméteres operator overloading

- Használhatjuk az operator overload-ot osztályokon kívül is. Ekkor 2 paramétert kell megadnunk, a bal és a jobb értéket (ennek a függvénynek a Complex osztályon kívül kell lennie):

```
Complex operator+(float left, Complex right) {
  return Complex(left + right.re, right.im);
}
```

- Hasonlóan a korábbihoz, ez a két sor ekvivalens:

```
Complex g = 5.3 + d;
Complex g = operator+(5.3, d);
```

- Egy probléma van már csak. Mivel ez a függvény a `Complex` osztályon kívül van, ezért a *private* tagokhoz nem fér hozzá.
- Erre lesz a **friend** kulcsszó. Ha ezt a függvény deklarációt berakjuk a Complex osztályba, akkor kívül a definíciójában használhatjuk a *private* adattagokat:

```
friend Complex operator+(float left, Complex right);
```

Jelenlegi Complex osztály

```
class Complex {
private:
  float re;
  float im;
public:
  Complex();
  Complex(const Complex& other);
  Complex(float r);
  Complex(float r, float i);

  Complex operator+(Complex other);
  Complex operator+(float other);
  friend Complex operator+(float left, Complex right);
  Complex operator*(Complex other);
  float abs();

  void print();

  ~Complex();
};
```

cout overload

- Természetesen a szorzáshoz is megírhatjuk, hogy működjön *float*-okkal, de oldjuk meg most, hogy lehessen *Complex* objektumokat *cout*-al kiírni:

```
Complex a;
cout << "a: " << a << endl;
```

cout overload

- Természetesen a szorzáshoz is megírhatjuk, hogy működjön *float*-okkal, de oldjuk meg most, hogy lehessen *Complex* objektumokat *cout*-al kiírni:

```
Complex a;
cout << "a: " << a << endl;
```

- Ehhez is igazából csak egy operator overload kell, hogy működjön:

```
ostream& operator<<(ostream& os, Complex right) {
  os << right.re << " + " << right.im << "i";
  return os;
}
```

cout overload

- Természetesen a szorzáshoz is megírhatjuk, hogy működjön *float*-okkal, de oldjuk meg most, hogy lehessen *Complex* objektumokat *cout*-al kiírni:

```
Complex a;
cout << "a: " << a << endl;
```

- Ehhez is igazából csak egy operator overload kell, hogy működjön:

```
ostream& operator<<(ostream& os, Complex right) {
  os << right.re << " + " << right.im << "i";
  return os;
}
```

- Az **ostream** a *cout* típusa (az *iostream*-ben van definiálva). A bal érték mindig egy *ostream*, hisz `cout << valami;` az általános parancsunk.

cout overload

- Természetesen a szorzáshoz is megírhatjuk, hogy működjön *float*-okkal, de oldjuk meg most, hogy lehessen *Complex* objektumokat *cout*-al kiírni:

```
Complex a;
cout << "a: " << a << endl;
```

- Ehhez is igazából csak egy operator overload kell, hogy működjön:

```
ostream& operator<<(ostream& os, Complex right) {
  os << right.re << " + " << right.im << "i";
  return os;
}
```

- Az **ostream** a *cout* típusa (az *iostream*-ben van definiálva). A bal érték mindig egy *ostream*, hisz `cout << valami;` az általános parancsunk.
- Gondoljuk meg most, hogy miért kell visszaadnunk a balértéket (*ostream*/*cout*-ot).

Operator chaining

- Nézzünk rá mi történik itt a háttérben, feltéve, hogy a, b, c Complex objektumok:

```
Complex d = a + b + c;
```

Operator chaining

- Nézzünk rá mi történik itt a háttérben, feltéve, hogy a, b, c Complex objektumok:

  ```
  Complex d = a + b + c;
  ```

- Bontsuk fel függvény/metódus hívásokra, mint korábban:

  ```
  Complex d = (a.operator+(b)).operator+(c);
  ```

Operator chaining

- Nézzünk rá mi történik itt a háttérben, feltéve, hogy a, b, c Complex objektumok:

  ```
  Complex d = a + b + c;
  ```

- Bontsuk fel függvény/metódus hívásokra, mint korábban:

  ```
  Complex d = (a.operator+(b)).operator+(c);
  ```

- Ahhoz hogy ez működjön az `a.operator+(b)` hívásnak *Complex*-et kell visszaadnia, vagy legalábbis olyat amihez utána hozzá tudunk adni még egy *Complex*-et.

Operator chaining

- Nézzünk rá mi történik itt a háttérben, feltéve, hogy a, b, c Complex objektumok:

```
Complex d = a + b + c;
```

- Bontsuk fel függvény/metódus hívásokra, mint korábban:

```
Complex d = (a.operator+(b)).operator+(c);
```

- Ahhoz hogy ez működjön az `a.operator+(b)` hívásnak *Complex*-et kell visszaadnia, vagy legalábbis olyat amihez utána hozzá tudunk adni még egy *Complex*-et.
- Hasonló módon bontsuk fel ezt, feltéve, hogy az *a* Complex:

```
cout << "a: " << a;
```

Operator chaining

- Nézzünk rá mi történik itt a háttérben, feltéve, hogy a, b, c Complex objektumok:

```
Complex d = a + b + c;
```

- Bontsuk fel függvény/metódus hívásokra, mint korábban:

```
Complex d = (a.operator+(b)).operator+(c);
```

- Ahhoz hogy ez működjön az `a.operator+(b)` hívásnak *Complex*-et kell visszaadnia, vagy legalábbis olyat amihez utána hozzá tudunk adni még egy *Complex*-et.
- Hasonló módon bontsuk fel ezt, feltéve, hogy az *a* Complex:

```
cout << "a: " << a;
```

- Itt két paraméteres operátorok lesznek, de hasonlóan egymásba ágyazva:

```
operator<<(operator<<(cout, "a: "), a);
```

Operator chaining

- Nézzünk rá mi történik itt a háttérben, feltéve, hogy a, b, c Complex objektumok:

  ```
  Complex d = a + b + c;
  ```

- Bontsuk fel függvény/metódus hívásokra, mint korábban:

  ```
  Complex d = (a.operator+(b)).operator+(c);
  ```

- Ahhoz hogy ez működjön az `a.operator+(b)` hívásnak *Complex*-et kell visszaadnia, vagy legalábbis olyat amihez utána hozzá tudunk adni még egy *Complex*-et.
- Hasonló módon bontsuk fel ezt, feltéve, hogy az *a* Complex:

  ```
  cout << "a: " << a;
  ```

- Itt két paraméteres operátorok lesznek, de hasonlóan egymásba ágyazva:

  ```
  operator<<(operator<<(cout, "a: "), a);
  ```

- Ahhoz, hogy ez működjön, a belső `operator<<(cout, "a: ")` hívásnak a *cout*-ot kell visszaadnia, hisz ekkor tudja a külső hívás használni, hogy kiírja az *a*-t.

Fordítós keresési sorrendje

- Fordítási időben dől el, hogy adott műveletnél melyik **operator** metódust vagy függvényt használjuk.
- Tehát ha valamilyen operátorhoz nincs megfelelő függvény, akkor le se fordul a programunk.

Fordítós keresési sorrendje

- Fordítási időben dől el, hogy adott műveletnél melyik **operator** metódust vagy függvényt használjuk.
- Tehát ha valamilyen operátorhoz nincs megfelelő függvény, akkor le se fordul a programunk.
- Ha az operátor bal oldala osztály, akkor az adott osztályban keres megfelelő metódust, ilyen formában:

```
visszateresi_ertek operator+(jobb_ertek_tipus x);
```

Fordítós keresési sorrendje

- Fordítási időben dől el, hogy adott műveletnél melyik **operator** metódust vagy függvényt használjuk.
- Tehát ha valamilyen operátorhoz nincs megfelelő függvény, akkor le se fordul a programunk.
- Ha az operátor bal oldala osztály, akkor az adott osztályban keres megfelelő metódust, ilyen formában:

```
visszateresi_ertek operator+(jobb_ertek_tipus x);
```

- Ha nem talált ilyet, vagy a bal érték nem osztály (hanem pl *int* vagy *float*), akkor keres a típusoknak megfelelő globális függvényt:

```
visszateresi_ertek operator+(bal_tipus x, jobb_tipus y);
```

Fordítós keresési sorrendje

- Fordítási időben dől el, hogy adott műveletnél melyik **operator** metódust vagy függvényt használjuk.
- Tehát ha valamilyen operátorhoz nincs megfelelő függvény, akkor le se fordul a programunk.
- Ha az operátor bal oldala osztály, akkor az adott osztályban keres megfelelő metódust, ilyen formában:

```
visszateresi_ertek operator+(jobb_ertek_tipus x);
```

- Ha nem talált ilyet, vagy a bal érték nem osztály (hanem pl *int* vagy *float*), akkor keres a típusoknak megfelelő globális függvényt:

```
visszateresi_ertek operator+(bal_tipus x, jobb_tipus y);
```

- Következő előadáson még ebben a témában ránézünk az `operator[]`-ra és megnézzük mi a jelentősége annak, ha referenciát adunk vissza.

kisZH kérdések

- Mire való a **friend** kulcsszó?
- Mutassunk példát egy operator overload-ra, tetszőleges operátorral!
- Miért nem lehet az `5.3 + Complex(1,2)` műveletet a *Complex* osztályon belül megoldani?
- Mi az **ostream**?
- Mennyi $5 + 10 * 2/4$?