

OO Programozás

Operátor felültöltés, másolás és átalakítás

Dr. Hatwagner F. Miklós

Széchenyi István Egyetem, Győr

https://github.com/wajzy/GKxB_INTM085

2023. október 23.


```

6  class Complex {
7      double re, im;
8      public:
9          Complex() {
10             re = im = 0.;
11         }
12         Complex(double re, double im) {
13             this->re = re;
14             this->im = im;
15         }
16         double getRe() {
17             return re;
18         }
19         void setRe(double re) {
20             this->re = re;
21         }

```

Complex1.h

```
22     double getIm() {
23         return im;
24     }
25     void setIm(double im) {
26         this->im = im;
27     }
28     void print() {
29         std::cout << re << '+' << im << 'i' << std::endl;
30     }
31     Complex add(Complex right) {
32         return Complex(re + right.re, im + right.im);
33     }
34     Complex mult(Complex right) {
35         return Complex(re*right.re - im*right.im, im*right.re + re*right.im);
36     }
37 };
```

main1.cpp

```
1 #include <iostream>
2 #include "Complex1.h"
3
4 int main() {
5     Complex c(1., 1.);
6     Complex sum = c.add(Complex(2., 2.));
7     sum.print(); // 3+3i
8     Complex total = c.mult(Complex(3., 4.));
9     total.print(); // -1+7i
10 }
```

Operátor felültöltés

- A C++ megengedi, hogy az operátorok jelentését kiterjesszük a saját típusainkra (osztályainkra)
- Pl. ha értelmezhető az összeadás két int vagy float között, akkor két Complex objektum miért ne lehetne összeadható?
- Az operátorok működését (praktikusan nyilvános tag)függvények adják meg → `operatorX`, ahol X pl. +, *.

Complex2.h

```
31 // Use reference to avoid copy of 'right'
32 Complex operator+(const Complex& right) {
33     return Complex(re + right.re, im + right.im);
34 }
35 Complex operator*(const Complex& right) {
36     return Complex(re*right.re - im*right.im, im*right.re + re*right.im);
37 }
```

- + művelet bal operandusa: aktuális objektum, a jobb oldalt paraméterként kapja.
- Utóbbi felesleges (tagonkénti) másolásának elkerülésére referenciát használunk.
- Új, ideiglenes (eredmény) objektum jön létre e kettő alapján, ezt adja vissza.

main2.cpp

```
4  int main() {
5      Complex c(1., 1.);
6      // Complex sum = c.operator+(Complex(2., 2.));
7      Complex sum = c + Complex(2., 2.);
8      sum.print();
9      // Complex sum2 = c + 100.;
10     // error: no match for 'operator+'
11     // (operand types are 'Complex' and 'double')
12     // note: no known conversion for argument 1
13     // from 'double' to 'const Complex&'
14     Complex total = c * Complex(3., 4.);
15     total.print();
16 }
```


Probléma:

Összeadásnál a jobb oldali operandusnak `Complex`-nek kell lennie.

Megoldás:

További felültöltött operátor függvények hozzáadása, pl. `double`-t hozzáadhatunk a valós részhez.

Complex3.h

```
32     Complex operator+(double re) {  
33         return Complex(this->re + re, im);  
34     }
```

main3.cpp

```
4  int main() {
5      Complex c(1., 1.);
6      // Complex sum = c.operator+(Complex(2., 2.));
7      Complex sum = c + Complex(2., 2.);
8      sum.print();
9      // Complex sum2 = c.operator+(100.);
10     Complex sum2 = c + 100.;
11     sum2.print();
12     // Complex sum3 = 100. + c;
13     // error: no match for 'operator+'
14     // (operand types are 'double' and 'Complex')
```

- Ez sem segít, ha a *bal* operandus double típusú → *nem tag*, két paraméteres operator függvény.
- Nem éri el a privát/védett tagokat → barát (friend) függvény: mindenhez hozzáfér az osztályon belül.
- Másik osztály is lehet az osztályunk barátja, pl.:
`friend class FriendOfComplex;`

Complex4.h

```
6  class Complex {
7      double re, im;
8      public:

28     friend std::ostream& operator<<(std::ostream& os, const Complex& cplx);
29
30     Complex operator+(const Complex& right) {
31         return Complex(re + right.re, im + right.im);
32     }
33     Complex operator+(double re) {
34         return Complex(this->re + re, im);
35     }
36     friend Complex operator+(double re, const Complex& right);
```

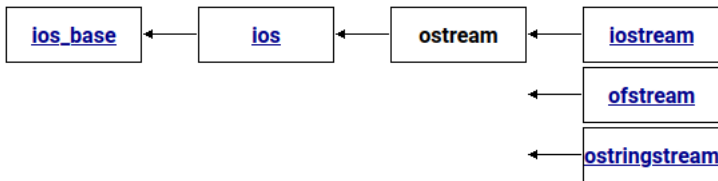
Complex4.cpp

```
3 std::ostream& operator<<(std::ostream& os, const Complex& c) {  
4     os << c.re << '+' << c.im << 'i';  
5     return os;  
6 }  
7  
8 Complex operator+(double re, const Complex& right) {  
9     return Complex(re + right.re, right.im);  
10 }
```

main4.cpp

```
4  int main() {
5      Complex c(1., 1.);
6      // Complex sum = c.operator+(Complex(2., 2.));
7      Complex sum = c + Complex(2., 2.);
8      std::cout << sum << std::endl; // 3+3i
9      // Complex sum2 = c.operator+(100.);
10     Complex sum2 = c + 100.;
11     std::cout << sum2 << std::endl; // 101+1i
12     // Complex sum3 = operator+(100., c);
13     Complex sum3 = 100. + c;
14     std::cout << sum3 << std::endl; // 101+1i
```

Az `ostream` a `cout`, `cerr` és `clog` objektumok típusa.



Bár különösebb haszna nincs, de csupa nem tag barát függvénnyel is megoldhattuk volna az operátorok felültöltését.

Complex5.h (Complex5.cpp)

```
6  class Complex {
7      double re, im;
8      public:

28     friend std::ostream& operator<<(std::ostream& os, const Complex& cplx);
29
30     friend Complex operator+(const Complex& left, const Complex& right);
31     friend Complex operator+(const Complex& left, double re);
32     friend Complex operator+(double re, const Complex& right);
33
34     friend Complex operator*(const Complex& left, const Complex& right);
35     friend Complex operator*(const Complex& left, double re);
36     friend Complex operator*(double re, const Complex& right);
37 };
```


Operátor felültöltés

- Definiáltuk az operátor *jelentését* olyan kifejezésben, melyben legalább egy operandus az osztály objektuma.
- Nem változtatható meg az operátor *szintakszisa*, *precedenciája*, *asszociativitása*, *operandusainak száma*.
- *Majdnem* minden operátor felültölthető.

- Az értékadás megengedett objektumok között → tagonkénti másolás.
- Probléma: dinamikusan foglalt területet több objektum adattagja is címezhet. Ha az egyik destruktora ezt felszabadítja, a másik ezen próbálhat műveletet végezni vagy újra felszabadítani.
- Egészítsük ki a **Message** osztályt felültöltött értékadás operátorral, készítsünk másolatot a tárolt szövegről!

message2.cpp

```
4  class Message {
5      private:
6          char* pStr;
7      public:
8          Message() {
9              pStr = new char( '\0' );
10             std::cout << "Created_[" << this << "]\n";
11         }
12
13         Message(const char* s) {
14             pStr = new char[ strlen(s) + 1 ];
15             strcpy(pStr, s);
16             std::cout << "Created_[" << this << ",_" << pStr << "]\n";
17         }

```

message2.cpp

```
19 ~Message() {
20     std::cout << "Freed [" << this << ", " << pStr << "]\n";
21     delete[] pStr;
22 }
23
24 friend std::ostream& operator<<(std::ostream& os, const Message& m);
25
26 // return reference instead of void to allow multiple assignments
27 Message& operator=(const Message& m) {
28     // do not copy to itself
29     if(&m == this) return *this;
30     setMessage(m.pStr);
31     return *this;
32 }
```

message2.cpp

```
34     void setMessage(const char* s) {
35         delete[] pStr;
36         pStr = new char[strlen(s) + 1];
37         strcpy(pStr, s);
38     }
39 };
40
41 std::ostream& operator<<(std::ostream& os, const Message& m) {
42     os << m.pStr;
43     return os;
44 }
```

message2.cpp

```
46 int main() {  
47     Message m1;  
48     m1.setMessage("Hello_C++_world!");  
49     std::cout << m1 << std::endl;  
50  
51     Message m2;  
52     m2 = m1;  
53     std::cout << m2 << std::endl;  
54  
55     Message m3;  
56     m3 = m2 = m1;  
57     std::cout << m3 << std::endl;  
58  
59     m1 = m1;  
60     std::cout << m1 << std::endl;  
61  
62     Message m4 = m1; // double free!  
63     std::cout << m4 << std::endl;  
64 }
```

Kimenet

```
Created [0x7ffcf5018618]  
Hello C++ world!  
Created [0x7ffcf5018620]  
Hello C++ world!  
Created [0x7ffcf5018628]  
Hello C++ world!  
Hello C++ world!  
Hello C++ world!  
Freed [0x7ffcf5018630, Hello C++ world!]  
Freed [0x7ffcf5018628, Hello C++ world!]  
Freed [0x7ffcf5018620, Hello C++ world!]  
Freed [0x7ffcf5018618, ]  
free(): double free detected in tcache 2  
Aborted (core dumped)
```

Ami jól működik:

Az értékadás operátor, ami másolatot hoz létre a karakterláncból,
nem engedi magát önmagára másolni, és
a többszörös értékadás is működik a visszaadott referencia miatt.

Ami hibás:

Az inicializáció nem értékadás, ilyenkor nem fut le a felültöltött függvényünk!

Másolási konstruktor (copy constructor):

Akár *egyetlen, azonos típusra vonatkozó referencia* paraméterrel is hívható konstruktor. (Lehetnek további paraméterek alapértelmezett értékekkel.)

Konverziós konstruktor (conversion constructor):

Ez is hívható egyetlen paraméterrel, de az más típusú.

Közös tulajdonság: objektum *inicializálásának* lehetősége kétféle szintaktikával:

```
Osztaly obj(42);    ≡    Osztaly obj = 42;
```


Másolási konstruktor további tulajdonságai:

- Ha mi nem készítünk, a fordító létrehoz egyet → tagonkénti másolás.
- Impliciten hívásra kerül:
 - Objektum átadása függvénynek paraméterként → a végtelen rekurzió elkerülése miatt kell az első paraméternek referenciának lennie.
 - Függvény adott osztály objektumát adja vissza.

message3.cpp

```
13     Message(const char* s) { // conversion ctor
14         pStr = new char[strlen(s) + 1];
15         strcpy(pStr, s);
16         std::cout << "Created " << this << ", " << pStr << "]\n";
17     }
18
19     /* for old compilers
20     Message(const Message& m) {
21         pStr = new char[strlen(m.pStr) + 1];
22         strcpy(pStr, m.pStr);
23         std::cout << "Created [" << this << ", " << m.pStr << "]\n";
24     } */
25
26     // C++11+ / delegating constructors
27     Message(const Message& m) : Message(m.pStr) {}
```

message3.cpp

```
56 int main() {  
57     Message m1;  
58     m1.setMessage("Hello_C++_world!");  
59     std::cout << m1 << std::endl;  
  
72     Message m4 = m1; // calls copy ctor  
73     std::cout << m4 << std::endl;  
74     Message m5(m1); // calls copy ctor  
75     std::cout << m5 << std::endl;  
76     Message m6 = "Conversion_+_copy_ctor";  
77     Message m7("Conversion_ctor");  
78 }
```

Kimenet

```
Created [0x7fff0778e740]  
Hello C++ world!  
...  
Created [0x7fff0778e758, Hello C++ world!]  
Hello C++ world!  
Created [0x7fff0778e760, Hello C++ world!]  
Hello C++ world!  
Created [0x7fff0778e768, Conversion ctor]  
Created [0x7fff0778e770, Conversion ctor]  
Freed [0x7fff0778e770, Conversion ctor]  
Freed [0x7fff0778e768, Conversion ctor]  
Freed [0x7fff0778e760, Hello C++ world!]  
Freed [0x7fff0778e758, Hello C++ world!]  
...  
Freed [0x7fff0778e740, Hello C++ world!]
```

Mikor hívjuk a konverziós konstruktort?

- Explicit hívás, más típusú adattal.
- Implicit hívás, a fordító egy adatot az osztály típusára akar alakítani → lehetővé teszi a felültöltött operátorfüggvények számának csökkentését!

Complex6.h

```
6  class Complex {
7      double re, im;
8  public:
9      // explicitly defaulted constructor
10     Complex() = default;
11     // conversion constructor
12     Complex(double re, double im = 0.) {
13         this->re = re;
14         this->im = im;
15     }
16
17
18
19
20
21
22
23
24
25
26
27
28
29     friend Complex operator+(const Complex& left, const Complex& right);
30     friend Complex operator*(const Complex& left, const Complex& right);
31 };
```

Complex6.h

```
10      Complex() = default;
```

Ha készítünk legalább egy konstruktort, a fordító nem hoz létre automatikusan *alapértelmezett* konstruktort. Ha mégis szükség lenne rá: `default` (C++11+; használható a kulcsszó más *speciális* tagfüggvényekhez is.)

Legtöbbször ha szükség van az alábbiak közül akár csak egynek is a definiálására, akkor a másik kettőre is szükség van ([Rule of Three](#)):

- destruktork
- másoló konstruktor
- hozzárendelés operátor

Vagy, átmeneti jelleggel akár le is tilthatjuk egyes függvények alapértelmezett előállítását → delete

Hasznos lehet nem kívánt típuskonverziók tiltására is.

message4.cpp

```
4  class Message {  
  
7      public:  
  
19         // C++11: deleted constructor  
20         Message(const Message& m) = delete;  
  
29         // deleted assignment operator  
30         Message& operator=(const Message& m) = delete;  
  
37     };
```


message4.cpp

```
44  int main() {  
  
49      Message m2;  
50      // m2 = m1; error: use of deleted function  
51      // 'Message& Message::operator=(const Message&)'  
52  
53      // Message m4 = m1; error: use of deleted function  
54      // 'Message::Message(const Message&)'  
55  
56      // Message m6 = "Conversion + copy ctor"; error: use of deleted function  
57      // 'Message::Message(const Message&)'  
58      // after user-defined conversion: Message::Message(const char*)  
59  
60      Message m7("Conversion_ctor"); // OK  
61  }
```