Nutzerdefinierte Ein- und Ausgabe

- warum `friend` ?
- warum `i/o-stream` Referenzen?
- warum `SomeClass` Referenzen?

```
class SomeClass { ...

friend std::ostream& operator<<
              (std::ostream&, [const] SomeClass [&]);

friend std::istream& operator>>
              (std::istream&, SomeClass &c)

};
SomeClass o;
cout<<o<<endl;       // op<< ( op<< ( cout, o ), endl );
cin>>o;              // op>> ( cin, o );
```

Member oder Friend?

```
class Complex { // wie oben, aber + und += in der jeweils anderen Variante
public:
    Complex operator+(const Complex&); // Member ?
    friend Complex& operator+=(Complex&, const Complex&); // Friend?
};

Complex z1, z2;

z2 = z1 + 1; // ok!
z2 = 1 + z1; // Fehler
// Typumwandlungen in beiden Argumenten erwünscht ! -> Friend

z1 += 1; // ok!
1 += z1; // auch ok, aber sinnlos
// Typumwandlungen nur im 2. Argument erwünscht ! -> Member
```

Überladung von `new` und `delete`

1. Replacement der impliziten globalen Operatoren

sämtliche Anforderungen und Freigaben von dynamischem Speicher nutzt dann diese: tiefer Eingriff in Laufzeitsystem, nichts für den Gelegenheitsprogrammierer

```
// Definition einer der impliziten Operationen
void* operator new(std::size_t); // throw(std::bad_alloc)
void* operator new[](std::size_t); // throw(std::bad_alloc)
void operator delete(void*) noexcept;
void operator delete[](void*) noexcept;
void* operator new(std::size_t, const std::nothrow_t&) noexcept;
void* operator new[](std::size_t, const std::nothrow_t&) noexcept;
void operator delete(void*, const std::nothrow_t&) noexcept;
void operator delete[](void* , const std::nothrow_t&) noexcept;
```

1. Replacement der impliziten globalen Operatoren

```
T* t = new T;       // ::operator new(sizeof(T)); !
delete t;           // ::operator delete(t);
t = new T[n];       // ::operator new[](sizeof(T)*n); !
delete[] t;         // ::operator delete[](t);

T* t = new (std::nothrow) T;  // returns 0 if it fails
delete(std::nothrow) t;
t = new (std::nothrow) T[n];  // returns 0 if it fails
delete[] (std::nothrow) t;
```

! throws std::bad_alloc if it fails

2. Neudefinition von globalen Operatoren

```
void* operator new/new[] (std::size_t, weitereParameter);
void operator delete/delete[] (void*, weitereParameter);
```

außer den sog. placement-Operationen, die nicht displaceable sind:

These functions are reserved, a C++ program may not define functions that displace the versions in the Standard C++ library.

```
void* operator new/new[] (std::size_t, void*);
void operator delete/delete[] (void*, void*);
...
char place[sizeof(Something)];
Something* p = new (place) Something();
delete (place) p;
```

2. Neudefinition von globalen Operatoren

```
// Beispiel: allocation trace
void* operator new (size_t s, const char* info = 0) {
        if (info)
                puts(info); // NOT cout<<info<<endl;
        void* p = calloc(s, 1); // zero-initialized
        if (!p) { ...some rescue action ... }
        return p;
}
void operator delete (void* p, const char* info = 0) {
        if (info)
                printf("%s\n", info);
        if (p) free(p);
}
```

3. Klassenlokale Operatoren `new` und `delete`

nur für dynamische Objekte dieses Typs, Vorteil: alle sind gleich groß
--> Pool Allocators

```
class X  {
public:
      void* operator new (size_t);     // bzw. Varianten
      void operator delete (void* p); // bzw. Varianten
};

X* px = new X; // X::operator new(sizeof(X));
delete px;     // X::operator delete(px);
```

Typumwandlungen

Konstruktoren (die mit einem Argument aufrufbar sind) fungieren als Typumwandler (von 1. Argumenttyp in den Klassentyp)

```
class X  {
public:
       X(int, double = 0);                  // int ---> X
       X(char*, int = 1, int = 2);          // char* ---> X
};
void f(X);
X g() { return 0;}
class Y       {
public: Y(X); };    // X ---> Y
```

Typumwandlungen

```
int main()
{
        X x1 = 1;               // 1 --> X
        X x2 = "ein X";         // "ein X" --> X
        f(2);                   // 2 --> X
        x2 = g();               // 0 --> X
        Y y = 0; //  ERROR: Cannot convert 'int' to 'Y'
}
```

Es kommt maximal EINE nutzerdefinierte Typumwandlung zum Einsatz !

Typumwandlungen

Falls automatische Umwandlung per Konstruktor unerwünscht ist, kann man solche als explicit spezifizieren:

```
class X  {
public:
        explicit X(int, double = 0);
        ...
};

...
f(2);          // ERROR: keine implizite Umwandlung 2 --> X
f(X(2));       // OK
```

Typumwandlungen

Ziel der Umwandlung durch Konstruktoren ist immer ein Klassentyp

Es gibt noch eine zweite Kategorie von nutzerdefinierten Umwandlungsoperationen, bei denen die Quelle der Umwandlung immer ein Klassentyp ist: Conversion Operators

```
class Bruch  { int z, n;
public:
        Bruch (int zaehler = 0, int nenner = 1)
                : z(zaehler), n(nenner) {}
        operator double() { return double(z)/n; }
        ...
};
Bruch halb(1,2); std::sqrt(halb); ....
```

kein Rückgabetyp ! keine Argumente !

Typumwandlungen durch Conversion Operators sind normalerweise mit Informationsverlust verbunden :-)

Umwandlung per Konstruktion und Konversion sind gleichberechtigt, jede Mehrdeutigkeit ist ein statischer Fehler!

```
class B { public: operator int(); };
class C { public: C(B); C(int); };
C operator+ (C c1, C c2) {return c1; } // mal kein friend !

C foo (B b1, B b2) { return b1+b2; }
// Ambiguity between 'operator +(C,C)' and 'B::operator int()' in function foo(B,B)
```

Ziel einer Konversion kann ein beliebiger Typ sein (z.B. auch ein Zeigertyp)

```
struct X {
  virtual operator const char*() { return "X"; }
};
struct Y : public X {
 virtual operator const char*() { return "Y"; };
};
int main() {
              X* p = new Y;
              cout <<  p << endl;
              cout << *p << endl;
}
```

```
C:\tmp>conv2
007B33E0
Y
```

Konversionen sind in C++98 nicht 'abschaltbar' (wie explicit ctors) ggf. Memberfunktionen `toType()` bevorzugen ! in C++11 auch erlaubt