Ziel: maximales Code-Sharing -- Weg: gemeinsame (aber ggf. in Ableitungen variierende) Funktionalität in Basisklassen festlegen

Problem: die so entstehenden Basisklassen sind oft so rudimentär, dass Objekterzeugung nicht sinnvoll und Implementation einiger Memberfunktionen (noch nicht) möglich ist:

abstract base class (ABC)

pure virtual function

Beispiel:

```
struct AbstractShape {
        virtual void draw() = 0;
        virtual void erase()= 0;
};
// no objects allowed:
// AbstractShape aShape; ERROR
AbstractShape *any; // ok
any = new Circle (Point(0,0), 100);
```

```
struct Circle : // real Shape
 public AbstractShape {
  virtual void draw() {...}
  virtual void erase() {...}
};
```

Neu in C++11 **final** (kein! reservierter Bezeichner - ein kontextsensitives Schlüsselwort)

finale Klassen: keine Ableitung möglich

finale Methoden: keine Redefinition in Ableitungen

```
class X {
public:
    virtual void foo();
};

class Y final : public X {
public:
    virtual void foo() override {}
};

void call (Y* p)
{
    p->foo(); // can bind statically !
}

// class Z : public Y {}; // not possible
```

finale Methoden müssen virtuell sein !

```
class Z {
    void virtual foo() const {}
};

class ZZ : public Z {
    void foo() const final override {};
} final, override;
```

2. Klassen in C++

```
class abstractBase { public:
        virtual void pure() = 0;
        void notPure() { pure(); }
        abstractBase() { notPure(); }
        virtual ~abstractBase() { notPure(); }
};

class concrete: public abstractBase { public:
        void pure() {}
        concrete() {}
};

int main() {
        cout<<"buggy:"<<endl;
        concrete c;
/*
        g++: pure virtual method called
        terminate called without an active exception
        Abort
*/
}
```

Scott Meyers, Effective C++ :
Item 9: "Never call virtual functions during construction or destruction."

Neu in C++11 **deleted/defaulted functions** (in Anlehnung an die Syntax von pure virtual functions)

```
class X {
public:
    X() = default;
    virtual ~X() = default;
    X(const X&) = delete;
    void foo(int);
    void foo(double) = delete;
};

X x;
X x1(x);
x.foo(1);
x.foo(1.0);

// delete auch für globale Funtionen
void bar(double);
void bar(int) = delete;
bar(1.9);
bar(19);
```

Call to deleted constructor of 'X'

Call to deleted member function 'foo'

Call to deleted function 'bar'

Im Kontext von Klassen können Operatoren mit nutzerdefinierter Semantik implementiert werden:

```
//Complex.h:                ∃  std::complex<T>
#include <iosfwd>
class Complex {
        double re, im;
public:
   Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
   friend Complex operator+(const Complex&, const Complex&);
   friend Complex operator*(const Complex&, const Complex&);
   friend bool operator==(const Complex&, const Complex&);
   friend bool operator!=(const Complex&, const Complex&);
   Complex& operator+=(const Complex&); // Member !
   Complex operator-(); // Member !
   friend std::ostream& operator<<(std::ostream&, const Complex&);
   friend std::istream& operator>>(std::istream&, Complex&);
....};
```

```
//complex.cpp: Auswahl
Complex operator+(const Complex& c1, const Complex& c2) {
        return Complex(c1.re+c2.re, c1.im+c2.im);
}
bool operator==(const Complex& c1, const Complex& c2) {
        return (c1.re==c2.re && c1.im==c2.im);
}
Complex& Complex::operator+=(const Complex& c) {
        re += c.re; im += c.im;
        return *this;
}
Complex Complex::operator-() {
        return Complex(-re, -im);
}
std::ostream& operator<< (std::ostream& o, const Complex &c) {
        return o << c.re << "+i*" << c.im;
}
```

```
//usecomplex.cpp:

int main() {
        Complex z1 (3, 4);
        Complex z2 (5, 6);
        Complex z3;
        cout << "z1=" << z1 << endl << "z2=" << z2 << endl;
        cout << "z1+z2=" << z1+z2 <<endl;
        cout << "gimme a Complex: ";
        cin >> z3;
        cout << "z3=" << z3 << endl;
}
```

Die Semantik von Operatoren kann nutzerdefiniert überladen werden, nicht dagegen deren Signatur, Priorität und Assoziativität

Es ist nicht möglich, neue Operatoren einzuführen (** %$@# ....)

Überladbar sind die folgenden Operatoren:

```
[]   ()   ->   ++   --   &    *    +
-    ~    !    /    %    <<   >>   <
>    <=   >=   ==   !=   ^    |    &&
||   =    *=   /=   %=   +=   -=   <<=
>>=  &=   ^=   |=   ,    new delete
```

nicht überladbar sind dagegen `.  .*  .->  ::  ?:`

Die vordefinierte Semantik von Operatoren für built in -Typen bleibt erhalten

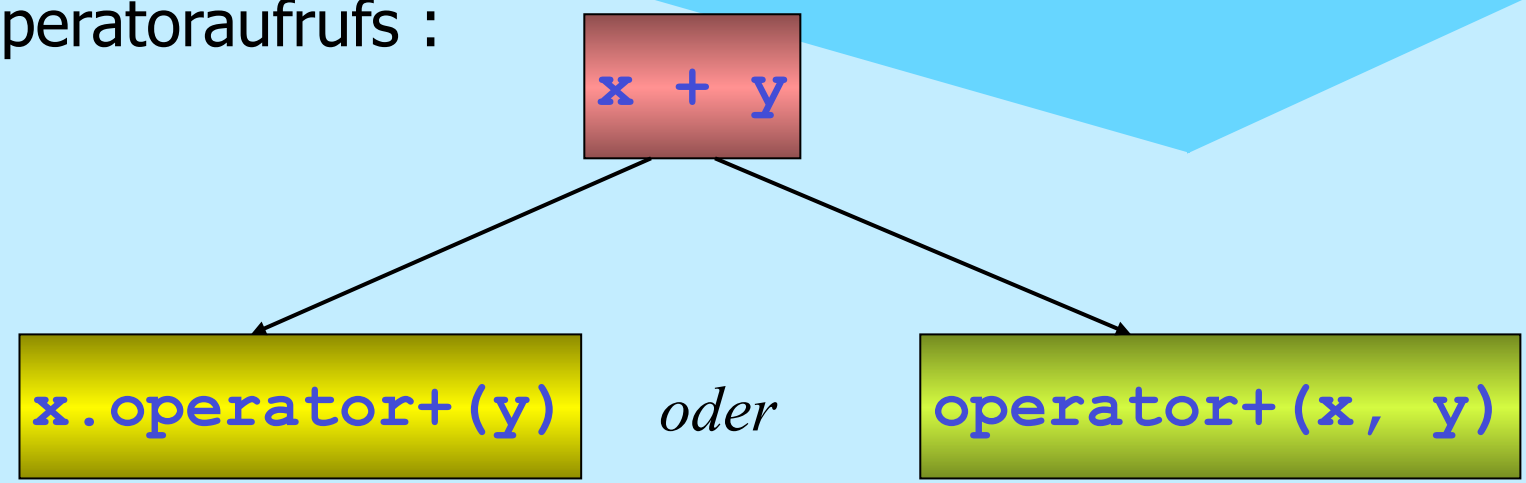
```
// falsch:
// int operator+ (int i, int j) {return i - j;}
```

durch die Forderung:

Ein Operator kann nur dann überladen werden, wenn in seiner Deklaration mindestens ein Parameter von einem Klassentyp (ggf. auch const / &) ist (dies kann auch das implizite `this`-Argument einer Memberfunktion sein) !

Member oder Friend (globale Funktion) ?

generell gibt es zwei Möglichkeiten der Auflösung eines Operatoraufrufs :



x muss von einem Klassentyp sein
(nur) y wird u.U. Typumwandlungen unterzogen

x oder y muss von einem Klassentyp sein
x und y werden u.U. Typumwandlungen unterzogen

Operatoren können NICHT static sein !

Syntax

Member

unäre Operatoren

```
class X { public:
 T operator ⊙ ();
};

⊙x; // Ergebnis: T
// (x).operator ⊙ ();
```

binäre Operatoren

```
class X { public:
 T1 operator ⊙ (T2);
};

x ⊙ y; // Ergebnis: T1
// (x).operator ⊙(y);
```

Friend

```
class X { public:
 friend T operator ⊙
          ([const]X[&]);
};

⊙x; // Ergebnis: T
// operator ⊙(x);
```

```
class X { public:
 friend T1 operator ⊙
          ([const]X[&], T2);
 friend T1 operator ✸
          (T2, [const]X[&]);
};
x ⊙ y; // Ergebnis: T1
// operator ⊙(x, y);
y ✸ x; // Ergebnis: T1
// operator ✸(y, x);
```

```
X x; T2 y;
```