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() {...}
};
```

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."

2. Klassen in C++

```
class abstractBase2 { public:
        virtual void pure() = 0;
        void notPure() { pure(); }
        abstractBase2() { /* no virtual function call ! */ }
        virtual ~abstractBase2() {/* no virtual function call ! */ }
};

class concrete2: public abstractBase2 { public:
        void pure() {}
        concrete2() {}
};

int main() {
        cout<<"buggy too:"<<endl;
        abstractBase2 *p = new concrete2;
        abstractBase2 *q = p;

        delete q;
        p->notPure();    // pure virtual method called
}
```

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

2. Klassen in C++

[http://www.artima.com/cppsource/pure_virtual.html]

This is another classic blunder: going indirect on a "dangling" pointer. That's a pointer to an object that's been deleted, or memory that's been freed, or both. C++ programmers never write such code ... unless they're clueless (unlikely) or rushed (all too likely).

So now p points to an ex-object. What does that thing look like? According to the C++ standard, it's "undefined". That's a technical term that means, in theory, anything can happen: the program can crash, or keep running but generate garbage results, or send Bjarne Stroustrup e-mail saying how ugly you are and how funny your mother dresses you. You can't depend on anything; the behavior might vary from compiler to compiler, or machine to machine, or run to run. In practice, there are several common possibilities (which may or may not happen consistently):

- *The memory might be marked as deallocated. Any attempt to access it would immediately be flagged as the use of a dangling pointer. That's what some tools (BoundsChecker, Purify, valgrind, and others) try to do. As we'll see, the Common Language Runtime (CLR) from Microsoft's .NET Framework, and Sun Studio 11's dbx debugger, work this way.*
- *The memory might be deliberately scrambled. The memory management system might write garbage-like values into the memory after it's freed. (One such value is "dead beef": 0xDEADBEEF, unsigned decimal 3735928559, signed decimal -559038737.)*
- *The memory might be reused. If other code was executed between the deletion of the object and the use of dangling pointer, the memory allocation system might have created a new object out of some or all of the memory used by the old object. If you're lucky, this will look enough like garbage that the program will crash immediately. Otherwise the program will likely crash sometime later, possibly after curdling other objects, often long after the root cause problem occurred. This is the kind of problem that drives C ++ programmers crazy (and makes Java programmers overly smug).*
- *The memory might have been left exactly the way it was.*

The last is an interesting case. What was the object "exactly the way it was"? In this case, it was an instance of the abstract base class; certainly that's the way the vtbl was left. What happens if we try to call a pure virtual member function for such an object? "Pure virtual function called".

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

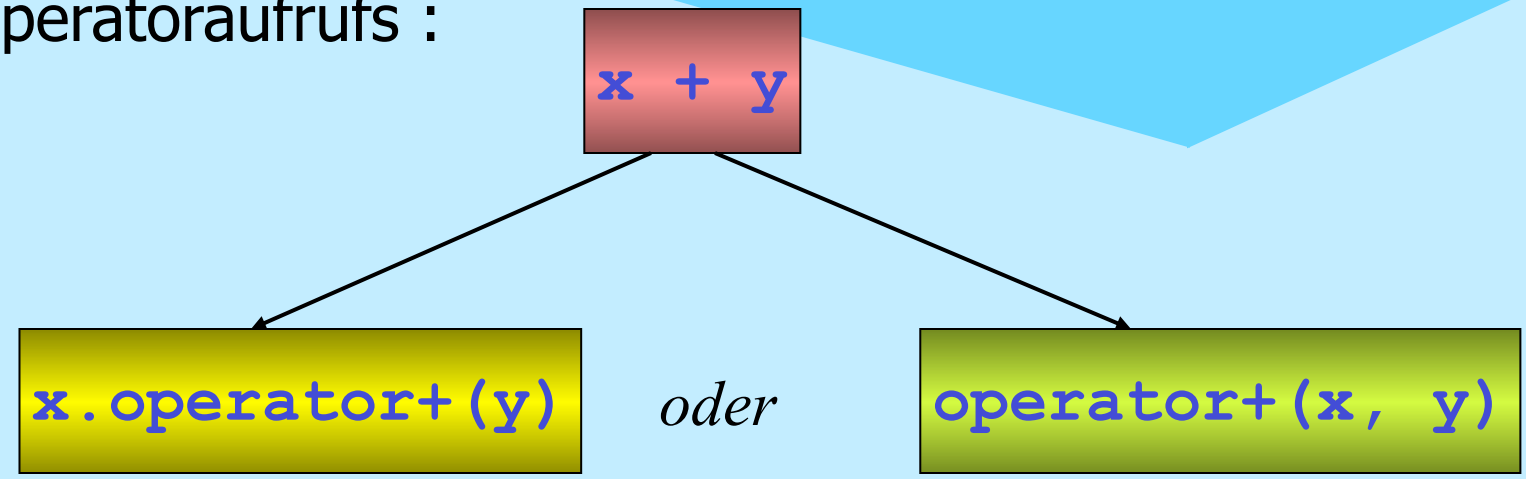
```
// falsch:
// int operator+ (int i, int j) {return i - j;}
```

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;
```

Operator	empfohlene Variante
alle einstelligen	Member
`= () [] ->`	müssen Member sein !
alle der Form `@=`	Member
alle anderen zweistelligen	Friend

Sonderfälle

```
T X::operator [] (IndexT);          X x; IndexT i; T t;
t = x[i]; // (x).op[](i)
T X::operator () (T1, T2, ....Tn); T1 t1; ... Tn tn;
t = x(t1,t2,....tn); // (x).op()(t1,t2, ....tn)
```

funktionale Objekte

```
X& X::operator++ ();     ++x;
X  X::operator++ (int); x++;
```

☺ **syntaktischer Hack**

```
T X::operator->(void);
x->selector  // (x).operator->()->selector
```

kanonischer Zuweisungsoperator copy assignment

```
X& X::operator= (const X&);          X x1, x2;
```

wird implizit bereitgestellt* (mit shallow assignment Semantik), kann neu definiert werden, dann ist die komplette Semantik von "Zuweisung" nutzerdefiniert zu implementieren, incl. Zuweisung von enthaltenen Objekten bzw. Basisklassenbestandteilen

```
class A { public: /* copy assignment implicit or explicit */ };
class B : public A {public: B& operator= (const B&); };
B& B::operator=(const B& src) { // assign B member
        // how to assign the A-part ???
        A::operator= (src); // oder
        A* thisA = this;
        *thisA = src;
        return *this;
}
```

* nicht, wenn die Klasse konstante Member oder Referenzen enthält, oder Basis-=-Operator(en) nicht aufrufbar ist !