

2. Klassen in C++

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 . .* .-> :: ?:

2. Klassen in C++

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

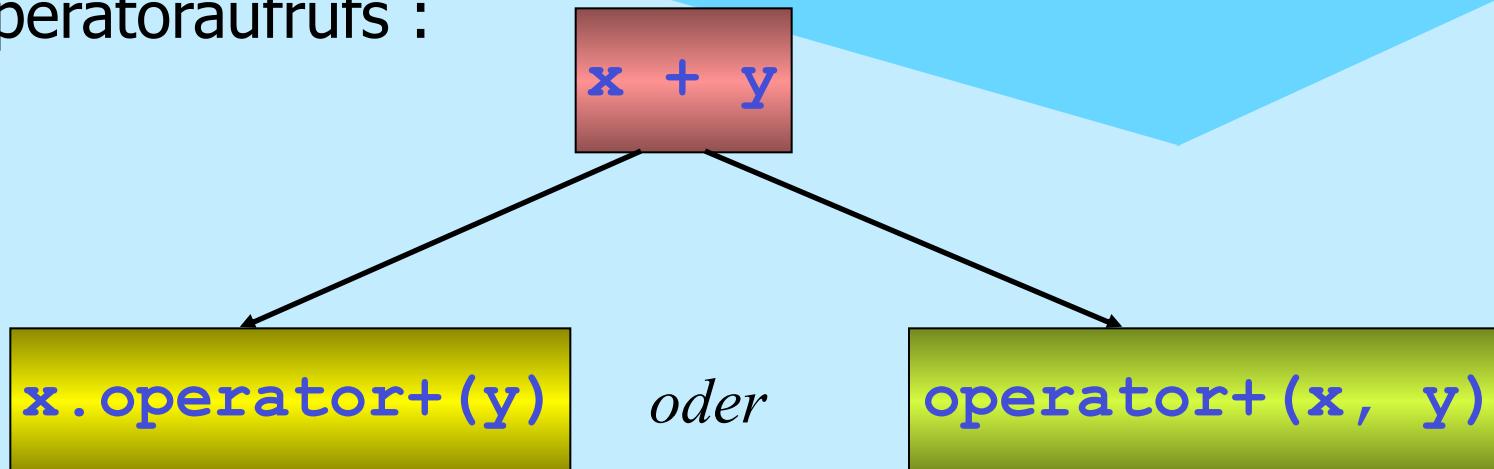
```
// falsch:  
// int operator+ (int i, int j) {return i - j;}
```

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 !

2. Klassen in C++

Syntax

unäre Operatoren

binäre Operatoren

Member

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

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

```
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;
```

2. Klassen in C++

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

```

2. Klassen in C++

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 !

2. Klassen in C++

wenn ein nutzedefinierter Copy-Konstruktor vorliegt, ist zumeist auch der Zuweisungsoperator nutzerdefiniert zu implementieren

```
Stack& Stack::operator= (const Stack& src)
{
    if (&src==this) return *this; // self assignment
    top = src.top;
    max = src.max;
    // NOT: data = src.data; as the implicit one does
    // leak in this->data, data sharing afterwards
    delete[] data;
    data = new int[max];
    for (int i=0; i<top; ++i) data[i]=src.data[i];
    return *this;
}
```

2. Klassen in C++

Copy-Konstruktor und Copy-Assignment-Operator sind semantisch verwandt: meist gemeinsam bereitzustellen!

Kanonische und exception safe Implementation:

GotW #59 (Sutter: mxC++ Item 22)

What is the canonical form of strongly exception safe copy assignment?

2. Klassen in C++

What are the three common levels of exception safety? Briefly explain each one and why it is important.

The canonical Abrahams^(*) Guarantees are as follows.

1. **Basic Guarantee:** If an exception is thrown, **no resources are leaked**, and **objects remain in a destructible and usable -- but not necessarily predictable -- state**. This is the weakest usable level of exception safety, and is appropriate where client code can cope with failed operations that have already made changes to objects' state.
2. **Strong Guarantee:** If an exception is thrown, **program state remains unchanged**. This level always implies global commit-or-rollback semantics, including that no references or iterators into a container be invalidated if an operation fails. In addition, certain functions must provide an even stricter guarantee in order to make the above exception safety levels possible:
3. **Nothrow Guarantee:** The function **will not emit an exception under any circumstances**. It turns out that it is sometimes impossible to implement the strong or even the basic guarantee unless certain functions are guaranteed not to throw (e.g., destructors, deallocation functions).

(^{*} http://www.boost.org/more/generic_exception_safety.html)

2. Klassen in C++

Exception safe copy assignment → two steps:

First, provide a nonthrowing Swap() function that swaps the guts (state) of two objects:

```
void T::Swap( T& other ) // throw()
{ /* ...swap the guts of *this and other... */ }
```

Second, implement operator=() using the "create a temporary and swap" idiom:

```
T& T::operator=( const T& other ) {
    T temp( other ); // do all the work off to the side
    Swap( temp );   // then "commit" the work using
                    // nonthrowing operations only
    return *this;
}
```

2. Klassen in C++

Beispiel Stack: stack.h

```
class Stack {
    int max, top;
    int *data;
    void swap(Stack&); // throw ();

protected:
    int* get_data() const {return data;}
    int  get_top()  const {return top;}
    int  get_max()  const {return max;}
public:
    explicit Stack(int dim=100);
    Stack(const Stack&);
    Stack& operator=(const Stack&);

    virtual ~Stack();
    virtual void push (int i);
    int pop();
    int full() const;
    int empty() const;
};
```

2. Klassen in C++

Beispiel Stack: stack.cc

```
//...
Stack::Stack(int dim): max(dim), top(0), data(new int[dim]) { }

Stack::Stack(const Stack& o):max(o.max),top(o.top),data(new int[o.max]) {
    for (int i=0; i<top; ++i) data[i] = o.data[i];
}

#include <algorithm>
void Stack::swap(Stack& other) {           // never fails:
    std::swap(max, other.max);           // swapping
    std::swap(top, other.top);           // builtin types
    std::swap(data, other.data);         // always succeeds
}

Stack& Stack::operator=(const Stack& src) {
    Stack temp (src); // in case of failure: no change to this
    swap(temp);       // succeeds always
    return *this;
}
//...
```

2. Klassen in C++

const reicht zur Unterscheidung von überladenen Funktionen (auch Operatoren) aus

typisches Idiom:

```
class Vector { int* data; int dim;
    void check(int i) const // WHY const?
    {if (i<0 || i>=dim) throw std::out_of_range("Vector");}
public:
    Vector(int d, int val=0): dim(d), data(new int[d])
    {for (int i=0; i<dim; ++i) data[i]=val;}
    Vector(const Vector&); // deep copy
    Vector& operator=(const Vector&); // deep assign
    int operator[] (int i) const { check(i); return data[i]; }
    int& operator[] (int i)      { check(i); return data[i]; }
};

const Vector cv(20, 3); int i = cv[11]; // NOT cv[11] = 3;
Vector v(20, 4); int j = v[13]; v[13] = 7;
```