2. Neudefinition von globalen Operatoren

```
// Beispiel: allocation trace
void* operator new (std::size_t s, const char* info = 0) {
        if (info)
                printf("%s\n", 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 (std::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 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 nicht 'abschaltbar' (wie explicit ctors) ggf. Memberfunktionen `to`*`Type`*`()` bevorzugen !

sogar eine Konversion nach `void*` kann u.U. sinnvoll sein

```
// bcc32: ios.h (ähnlich in anderen Impl.)
inline basic_iosT::operator void*() const {
    return fail() ? (void*)0 : (void*)1;
}

// basic_iosT ist Basiklasse von ostream, ofstream

ofstream output ("file.txt");
// wenn die Dateien nicht zum Schreiben eröffnet
// werden konnte, ist das intern in einem Status
// vermerkt, den fail() abfragt:
if (output.fail()) ...  // ODER: VIEL KOMPAKTER
if (!output) ...
```

sämtliche Typumwandlungen (Konstruktion und Konversion) werden bei Bedarf implizit (außer bei explicit ctors) veranlasst, aber auch bei expliziten Cast-Operationen

```
T1 t1;
T2 t2 = (T2) t1; // oder auch
T2 t2 = T2 (t1); // falls T2 ein Typname (kein Typkonstrukt) ist
```

Casts sind syntaktisch eher unauffällig, werden in unterschiedlichsten Absichten (und z.T. mit nicht erkennbarem Risiko!) eingesetzt

```
X = 2 / double(3);  // OK
class B: public A {....};
A *pa = new B; B* pb = (B*)pa; // OK
cout << (void*)pa; // OK
int *pi = new int; int i = int(pi); // ???
const X x; X* px = (X*)&x;  // ???
class X{}; class Y{};
X *px = new X; Y* py = (Y*)px; // ???
```

Um die Semantik besser ausdrücken zu können (und dem Compiler mehr Prüfmöglichkeiten zu geben) bietet C++ vier spezielle Cast-Operatoren

```
T1 t1;
T2 t2 = const_cast<T2> (t1);
T2 t2 = static_cast<T2> (t1);
T2 t2 = reinterpret_cast<T2> (t1);
T2 t2 = dynamic_cast<T2> (t1);
```

const_cast<T>

» die Konstantheit eines Objektes ignorieren «

verletzt eigentlich die "Spielregeln": alle schreibenden Zugriffe nach Brechung der constness haben undefined behaviour

aber manchmal aus praktischen Gründen unumgänglich

```
// use std::string instead of [const] char*

string s ("simsalabim");

// but:
extern "C" void someOldCfunction (char*);
...
someOldCfunction(s.c_str()); // ERROR: const ignored
someOldCfunction(const_cast<char*>(s.c_str())); // OK
```

für einige häufige Anwendungsszenarien bietet C++ eine bessere Variante: **mutable**

```
class X {
      int copies;
public:
      X(): copies(0){}
      // Copy-Ctor: one of
      X(X& other) { other.copies++; }
      // kann keine Kopien von Konstanten machen
      // or:
      X(const X& other) { other.copies++; }
                                  ^ Cannot modify a const object
      int cc() const {return copies;}
};
```

const_cast<X&>(other).copies++;

```
class X {
        mutable int copies;
public:
        X(): copies(0){}
        X(const X& other) { other.copies++; }
        // kann Kopien von Konstanten machen und dabei
        // dennoch other.copies ändern !
        int cc() const {return copies;}
};
```

`mutable` immer benutzen, wenn Objekte logisch konstant, aber in (Implementations-) Details veränderlich sein sollen (z.B. Objekte mit lazy evaluation gewisser Eigenschaften)