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

```
// traditionell z.b. bcc32: ios.h (ähnlich in anderen Impl.)
inline basic_iosT::operator void*() const {
    return fail() ? (void*)0 : (void*)1;
}
// ab C++11 (besser)
explicit operator bool() const {return !fail();}

// 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)

static_cast<T>

» den Compiler überreden, verwandte Typen verträglich zu verwenden «
das Ergebnis kann ohne erneute Umwandlung verwendet werden

```
class X { ... };
class Y : public X {};

// eine Y& ist auch immer eine X&
Y o;
X& x1 = o; // implizite Anpassung der Typen
X& x2 = static_cast<X&> (o); // dasselbe

// manchmal ist eine X& auch eine Y&
Y& y1 = static_cast<Y&> (x1); // ok, weil x1 ein Y ref.
// aber eben nicht immer:
X& x3 = *new X; Y& y2 = static_cast<Y&> (x3); // Crash ahead
```

reinterpret_cast<T>

» den Compiler überreden, nicht verwandte Typen verträglich zu verwenden «

das Ergebnis kann nur nach erneuter Rückumwandlung verwendet werden

die unveränderte Bitbelegung wird anders interpretiert; zumeist nicht portabel

```
int *pi = &someint;
void *v = reinterpret_cast<void*>(pi);
// don't use v, but:
int *p = reinterpret_cast<int*>(v);
*p = 337; // OK: sets someint
```

dynamic_cast<T>

» zur Laufzeit verwandte Typen verträglich und sicher verwenden «

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


// manchmal ist eine X& auch eine Y&
Y& y1 = dynamic_cast<Y&> (x1); //ok,weil x1 ein Y ref.
// aber eben nicht immer:
X& x3 = *new X;
Y& y2 = dynamic_cast<Y&> (x3); // NO Crash ahead
// exception std::bad_cast !!!
```

dynamic_cast<T>

Implementation setzt offenbar Auswertung von Laufzeittypinformationen (RTTI - run time type identification) voraus

Funktioniert für Zeiger und Referenzen polymorpher Typen (es muss virtuelle Funktionen in der Basisklasse geben!)

Ein downcast (von einem Zeiger/einer Referenz auf eine Basisklasse auf einen Zeiger/eine Referenz einer Ableitung) gelingt, wenn das referenzierte Objekt vom Typ der Ableitung oder einer Ableitung dieser ist.

Bei Zeigern liefert `dynamic_cast` den Wert 0, bei Referenzen wird die Ausnahme `std::bad_cast` geworfen, wenn der dynamische Typ nicht ausreicht

dynamic_cast<T>

```
class A {
public:  virtual void needed () {}
};
class B: public A {public: int i;};
class C: public B {public: int j;};
int main() {
        A *pa = new B;
        B *pb = dynamic_cast<B*>(pa);
        if (pb) pb->i = 12345; // ok, es ist ein B
        C *pc = dynamic_cast<C*>(pb);
        if (pc) pc->j = 54321; // wird nicht ausgefuehrt
        pa = new C;
        pb = dynamic_cast<B*>(pa);
        if (pb) pb->i = 12345; // ok, es ist ein B
}
```

Darüber hinaus kann man die Typidentität direkt abfragen:

dazu existiert der Operator `typeid` (wie `sizeof` vom Compiler umgesetzt und nicht überladbar), der eine (vergleichbare) Struktur des Typs `type_info` liefert, der Vergleich von `type_info` gelingt, wenn exakt der gleiche Typ vorliegt

auf `type_info` ist wiederum die Funktion `name()` definiert, die einen Klarnamen der Klasse (nicht notwendig identisch mit dem Klassennamen) erzeugt

`#include <typeinfo>` ist erforderlich

Die beteiligten Typen müssen wiederum polymorph sein, d.h. mindestens eine virtuelle Funktion in der gemeinsamen Basis besitzen

```
#include <typeinfo>
#include <iostream>
using namespace std;

class A { virtual void any () {} };
class B: public A { };
class C: public A { };
void check (A* p) {
        if (typeid(*p)==typeid(A))
                { cout << "es ist ein A\n"; return; }
        if (typeid(*p)==typeid(B))
                { cout << "es ist ein B\n"; return; }
        cout << "weder A noch B\n";
}
const char* get_name(A* p) {
        return typeid(*p).name();
}
```

```
int main()
{
        A *p;
        p = new A;
        check (p);
        cout << get_name(p) << endl;
        p = new B;
        check (p);
        cout << get_name(p) << endl;
        p = new C;
        check (p);
        cout << get_name(p) << endl;
}
```

```
C:\tmp>rtti
es ist ein A
A
es ist ein B
B
weder A noch B
C
```

ohne **any** !

```
C:\tmp>rtti
es ist ein A
A
es ist ein A
A
es ist ein A
A
```

dynamic_cast<T> ist manchmal nicht zu vermeiden

```
class B {
        // no functionality 'foo'
};
class D: public B {
        virtual void foo();
};
void register (B*);
B* next();
...
register(new D);
...
B* n = next();

// how to call foo ?
D* d;
if (d=dynamic_cast<D*>(n)) d->foo();
```

RTTI nur in Ausnahmefällen explizit benutzen

statt spaghetti code

```
Shape * s;
if (typeid(*s)== typeid(Circle))
((Circle*)s)->Circle::draw();
else
if (typeid(*s)== typeid(Rectangle))
        ((Rectangle*)s)-> Rectangle::draw();
else ...
```

benutze

```
Shape * s;
s->draw(); // late bound virtual
```