SEP

5. Tutorium

Type casting

The Good, the Bad and the Ugly

Implizite Konvertierung

- Standard Conversions
  - int ↔ char
  - float ↔ int
  - Derived* → Base*
  - …
- Konstruktor
  - Ermöglicht Typumwandlung
    - Kann verwendet werden um ein Objekt aus einem Objekt eines anderen Typs zu erzeugen
    - `explicit`

Explizite Konvertierung

- Den Typ eines Objektes (Ausdrucks) ändern
- In „gutem“ C++ nur selten notwendig
  - Ich hab ein A, brauch aber ein B
    - warum?

Function-Style Cast

- „Konstruktorschreibweise“
  - Sieht aus wie (und ist im Prinzip auch) ein Konstruktoraufruf
- Erzeugt ein temporäres Objekt von gegebenem Typ

```
int a = 42;
float b = float(a);
int c = 2 * int(3.14159265358979f);

std::string x = std::string(“hello ”) + “world!”;
```

static_cast

- „Normale Umwandlung“
- Castet *kompatible* Typen
  - Standard Conversions
  - Umkehrung von Standard Conversions
    - Downcast in Klassenhierarchie (unchecked!)
  - Konstruktor (expliziter Aufruf)

```
int a = 42;
float b = static_cast<float>(a);
int c = 2 * static_cast<int>(3.14159265358979f);

std::string x = static_cast<std::string>(“hello ”) + “world!”;
```

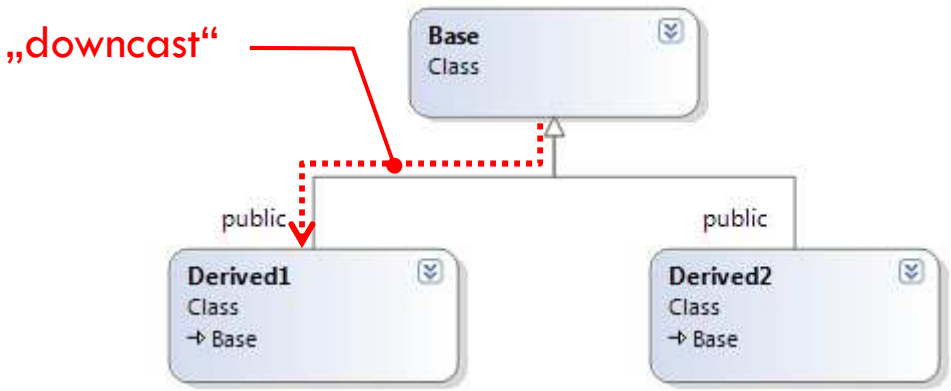
dynamic_cast

- Sicherer downcast in einer Klassenhierarchie
  - Typprüfung zur Laufzeit
    - Kostet!
- Oft Indikator für einen Fehler im Design
  - Eine Abstraktionsebene zu hoch?
  - Catch your breath and think before use!

```
class Base
{
  virtual void blub();
};

class Derived1 : public Base
{
};

class Derived2 : public Base
{
};
```




```
void MyFunction(Base* base_pointer)
{
  if (Derived1* derived = dynamic_cast<Derived1*>(base_pointer))
  {
    // base_pointer points to an object of type Derived1
    // do something with derived…
  }
  else
    // do something else…
}
```

const_cast

- Erlaubt es const/volatile Qualifizierer wegzucasten
- Macht in manchen exotischen Fällen Sinn
  - Finger weg, außer man weis was man tut!
  - Keine Lösung um mal schnell diese blöden Compilerfehler loszuwerden
    - Kabuuum!

reinterpret_cast

- The Bad Guy
- Castet
  - Inkompatible Typen
    - Pointer ↔ Integrale Typen (int, long, …)
- „Implementation defined“
  - Was genau passiert hängt vom Compiler bzw. dem Zielsystem ab
  - Finger weg, außer man weis was man tut!

What about C-Casts?

- C++ casts
  - Ersetzen C-Casts *vollständig*
  - Sind ausdrucksstärker/restriktiver
    - Man hat mehr Kontrolle darüber *was genau* passiert

- Fazit:
  - C-Casts stay home

Operator overloading

Operator overloading

- Intuitives Interface für eigene Typen
  - Gleich einfach zu verwenden wie built-in Typen
  - Arithmetik
    - Vektoren, Matritzen, Komplexe Zahlen, …
  - Strings
    - Verkettung, Vergleich
  - Streams
    - „Stream Operatoren“ << bzw. >>
  - …

Was kann überladen werden?

- Fast alles
  - Ausnahmen: `.  .*  ::  ?:  sizeof  typeid  *_cast`
- Nur für benutzerdefinierte Typen
- Unterscheidung
  - Binäre Operatoren
    - `+  *  ==  &&  ...`
  - Unäre Operatoren
    - `!  -  ...`
  - Sonstiges
    - `[]  ()  ++  =  ...`

Wie kann überladen werden?

- Memberfunktion

```
Blub::operator +(const Blub& b);
```

- Freie Funktion

```
Blub operator +(const Blub& a, const Blub& b);
```

- Beispiel…

The End