Streams


elif operation == "MIRROR_Z":
mirror_mod.use_x = False
mirror_mod.use_y = False
mirror_mod.use_z = True
#selection at the end -add back the deselected mirror modifier object
mirror_ob.select= 1
modifier_ob.select=1
bpy.context.scene.objects.active = modifier_ob
print("Selected" + str(modifier_ob)) # modifier ob is the active ob

Namespaces

```
namespace A
{
    class String
    {
        …
    };
}
```

```
namespace B
{
    typedef char* String;

    …
}
```

Zuordnung von **Namensbereichen** (sogenannte *„namespaces“*) zur Vermeidung von Kollisionen von globalen Identifiers

Zugriff erfolgt mit Hilfe des Bereichsoperators

```
A::String s1
B::String s2
```

Meist wird ein Namensbereich bevorzugt und über das Schlüsselwort `using` definiert:

```
using namespace A
String s1
B::String s2
```

`using` kann auch auf einzelne Identifier angewandt werden (z.B. `using A::String`)

Ein- und Ausgabe

C	
Funktion	**Verwendung**
`fopen()`	Datei öffnen
`fprintf()/printf()`	Formatierten Text ausgeben
`fgetc()/getc()`	Zeichen einlesen
`fclose()`	Datei schließen
Konstanten	**Verwendung**
`stdin`	Standard Eingabekanal
`stdout`	Standard Ausgabekanal
`stderr`	Standard Fehlerausgabekanal

Funktionieren auch unter C++

C++ hat allerdings eine bessere Alternative …

Streams (I)

C++ Alternative: **Streams**

- *„Datenstrom“* in dem Zeichenfolgen *„entlangfließen“*
- Teil der Standard Klassenbibliothek

C++	
Klassen	**Verwendung**
`istream`	*„Eingabestrom“* (input stream) Daten können gelesen werden.
`ostream`	*„Ausgabestrom“* (output stream) Daten können ausgegeben werden.

Weitere Klassen existieren …

Global existierende Objekte:

C++	
Objekte	**Verwendung**
`cin`	Klasse `istream` Standard Eingabekanal (entspricht `stdin` in C)
`cout`	Klasse `ostream` Standard Ausgabekanal (entspricht `stdout` in C)
`cerr`	Klasse `ostream` Standard Fehlerausgabekanal (entspricht `stderr` in C)

Streams (II)

Einbindung der entsprechenden Header Datei notwendig

- `iostream` (C++) statt `stdio.h` (C)

Alle Streams Klassen/Objekte/etc. sind im Namensbereich `std` definiert

- Zugriff via Bereichsoperator: `std::cin`
- Zur Vereinfachung kann es sinnvoll sein, den Namensbereich `std` mit Hilfe des `using` Schlüsselworts zu bevorzugen

```
…
#include <iostream>
…

using namespace std;

…
```

Streams (III)

Für das Objekt `cout` wird der Operator `<<` mit dem entsprechenden Argument aufgerufen

- `cout.operator<<("Ausgabe");`

Der Operator ist bereits für alle Standard Datentypen überladen

- Abhängig vom Datentyp wird automatisch die dazugehörige Funktion aufgerufen
- Umwandlung des zweiten Operanden in eine Zeichenfolge die ausgegeben wird

`endl` ist einer der **Manipulator**

- Manipulation am Stream
- `endl`: `'\n'` ausgeben und Ausgabepuffer leeren

```
…

int x = 100;
float f = 4.5;

cout << "Ausgabe";

cout << x;
cout << f;

cout << endl;

…
```

Streams (IV)

Verkettungen sind möglich

Was passiert hier:

- `<<` ist ein linksassoziativer Operator

```
(((cout << "Wert: ") << x) << endl);
```

- Operator liefert als Resultat eine Referenz auf `cout` zurück
- Auf dieses Objekt kann dann der nächste Operator aufgerufen werden

```
…

int x = 100;

cout << "Wert: " << x << endl;

…
```

Streams (V)

Für das Objekt `cin` wird der Operator `>>` mit dem entsprechenden Argument aufgerufen

- `cin.operator>>(a);`

Der Operator ist bereits für alle Standard Datentypen überladen

- Abhängig vom Datentyp wird automatisch die dazugehörige Funktion aufgerufen
- Einlesen eines Werts und Zuweisen an das Argument (als Referenz übergeben)

Führende Whitespaces werden überlesen

Auch hier ist eine Verkettung möglich

```
…

char a;
int x;

cin >> a >> x;

…
```

Streams (VI)

C++	
Methode	**Verwendung**
`good()`	Alles in Ordnung (`ios::goodbit`)
`eof()`	End-of-File (`ios::eofbit`)
`fail()`	Fehler (`ios::failbit` oder `ios::badbit`)
`bad()`	Fataler Fehler (`ios::badbit`)
`rdstate()`	Liefert aktuell gesetzte Flags
`clear()`	Löscht oder setzt einzelne Flags

Methoden der Klasse Stream

Typische Verwendung

```
cin >> x;
if ( ! cin.good() )
{
    /* Fehler */
}
```

Flags werden über ein Bit Feld verwaltet

- Jedes Bit zeigt anderen Status
- Verwendung der good() Method reicht oft

Versuche mit der eigenen Klasse

Was ist das Problem?

Aufgerufen wird:

- `cout.operator<<((Bruch)a);`
- `((int)1000).operator+((Bruch)a);`
- `cout.operator<<((Bruch)b);`

Keine der beiden Klassen kannte die Klasse Bruch

- `ostream` – Objekt `cout`
- *„Integer“* – Objekt `1000`

Keine der Klassen stellt daher entsprechende Operatoren

Geschlossene Klassenbibliothek – daher kann die Klasse auch nicht einfach erweitert werden

```
Bruch a;
Bruch b;

…

cout << a << endl;

b = 1000 + a;

cout << b << endl;

…
```

Compiler:
No acceptable conversion

Globale Operator Funktionen

Operator-Funktionen können auch als globale Funktionen implementiert werden (***„Globales Überladen“***)

- Außerhalb der Klassen
- Wie jede andere Funktion auch

Alle Operanden werden als Argumente übergeben

- Auch das erste Argument (das Objekt selbst)

Kein Zugriff auf private Komponenten der Klasse

- Funktion gehört ja nicht mehr zur Klasse

Erweiterung von Operatoren bestehender Klassen auf eigene Datentypen

```
ostream&
operator << (ostream& strm,const Bruch& b)
{
   b.printOn(strm);

   return(strm);
}


Bruch operator + (int i, const Bruch& b)
{
   return(Bruch(i) + b);
}
```

Zusammenfassung

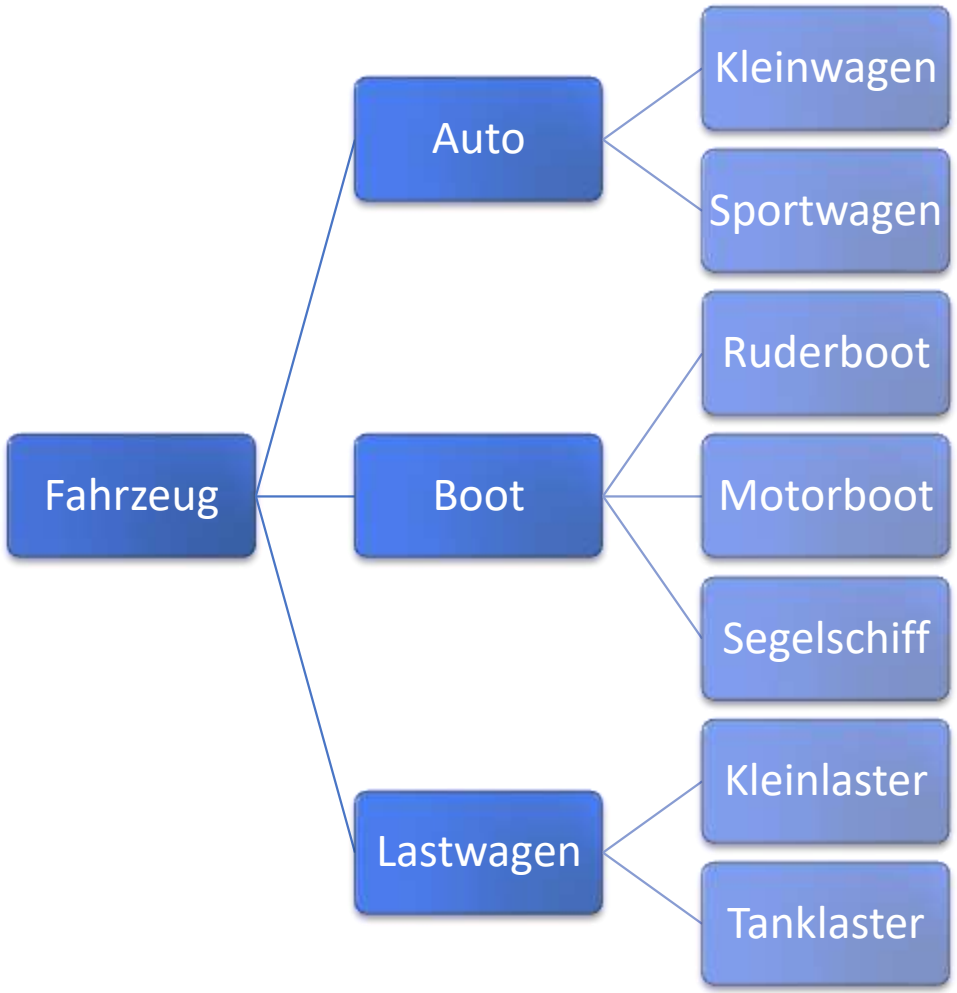
- Kollidierende Identifier können durch **Namespace** Definitionen gemeinsam in einem Programm verwendet werden.
  - Zugriff erfolgt über den Bereichsoperator.
  - Mit dem Schlüsselwort `using` kann ein bevorzugter Namespace definiert werden.

- Die Ein- und Ausgabe wird über verschiedene **Stream** Klassen realisiert.
  - Nutzen den Namespace `std`
  - Die globalen Objekte `cin`, `cout` und `cerr` sind als Standard Ein/Ausgabe Kanäle vordefiniert
  - Durch **globales Überladen** der I/O Operatoren (<< und >>) kann das Ein/Ausgabe Konzept auch auf eigene Klassen übertragen werden.

Vererbung


elif operation == "MIRROR_Z":
mirror_mod.use_x = False
mirror_mod.use_y = False
mirror_mod.use_z = True
#selection at the end -add back the deselected mirror modifier object
mirror_ob.select= 1
modifier_ob.select=1
bpy.context.scene.objects.active = modifier_ob
print("Selected" + str(modifier_ob)) # modifier ob is the active ob

Vererbung



Vererbung (*„inheritance“*): Ableitung der Eigenschaften einer Klasse von einer anderen Klasse

- Die neue Klasse *„erbt“* die Eigenschaften der bereits vorhandenen Klasse
- Nur noch neue Eigenschaften müssen neu implementiert werden (*„Spezialisierung“*)

Die Klasse von der geerbt wird, nennt man **Basisklasse** (auch **Oberklasse**, **Elternklasse**)

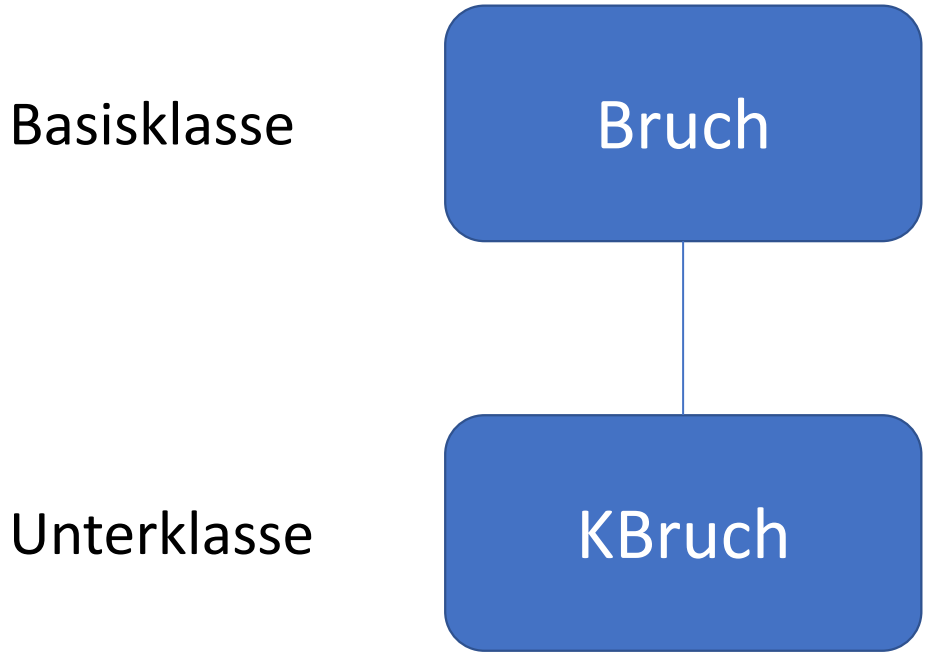
Die Klasse, die erbt, nennt man **abgeleitete Klasse** (auch **Unterklasse**, **Kind-Klasse**)

Eine abgeleitete Klasse kann wiederum eine Basisklasse sein ⟹ **Klassenhierarchie**

Erweiterung unseres Beispiels

Abgeleitete Klasse: KBruch

- *„kürzbarer Bruch“*
- Objekte: Brüche, die prinzipiell die Fähigkeit besitzen, gekürzt zu werden.



Private vs. Protected

Datenkapselung

- `private` – Zugriff nur für Methoden der Klasse
- `public` – öffentliche Schnittstelle
- `protected` – wie `private`, aber auch Zugriff für Methoden von abgeleiteten Klassen

Typischerweise wird der Zugriff für abgeleitete Klassen erlaubt

- Fälle in denen `private` sinnvoll ist, sind eher selten

```
class Bruch
{
    /* Unterklassen haben Zugriff */
protected:
    int zaehler;
    int nenner;

    /* oeffentliche Schnittstelle */
public:
    Bruch(int = 0);
    Bruch(int, int);
    ~Bruch();

    …
};
```

Vorüberlegungen zu KBruch

Neues Attribut

- `kuerzbar`

Neue Methoden

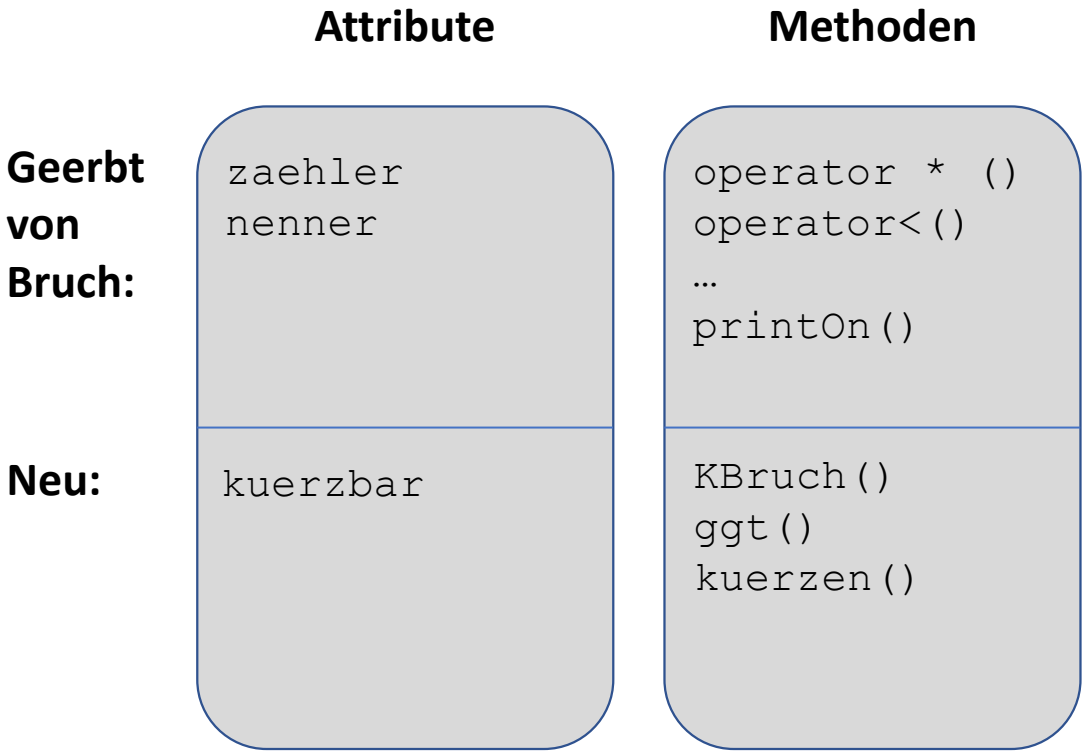
- `ggt()` liefert den größten gemeinsamen Teiler, mit dem der Bruch gekürzt werden könnte
- `kuerzen()` kürzt den Bruch (soweit möglich)

Konstruktoren / Destruktoren

- Werden grundsätzlich nicht geerbt

Existierende Methoden/Operatoren

- Werden geerbt
- Machen sie aber noch Sinn?



Deklaration der abgeleiteten Klasse

Allgemeine Form

```
class unterKlasse : [zugriff] basisKlasse
{
   deklarationen
}
```

Es spielt keine Rolle ob die Basisklasse selbst eine abgeleitete Klasse ist

[*zugriff*] schränkt optional den Zugriff auf geerbte Komponenten weiter ein

```
class KBruch : public Bruch
{
protected:
    bool kuerzbar;

    unsigned ggt() const;

public:
    KBruch(int = 0, int = 1);

    void kuerzen();
};
```

Zugriff auf geerbte Komponenten

Verhalten der Deklaration in der Basisklasse abhängig vom Schlüsselwort

Schlüssel-wort	Zugriffsdeklaration der Basisklasse		
	public	**protected**	**private**
public	public	protected	private
protected	protected	protected	private
private	private	private	private

Die abgeleitete Klasse kann direkt auf alle als `public` oder `protected` deklarierten Komponenten der Basisklasse zugreifen

Schlüsselwort definiert ob und inwiefern Zugriff auf geerbte Komponenten eingeschränkt wird

- Für Anwender der abgeleiteten Klasse
  - Schlüsselwort `private`: Anwender von KBruch könnten nicht mehr auf `printOn()` zugreifen
- Für weitere abgeleitete Klassen der Unterklasse
  - Schlüsselwort `private`: Keinerlei Zugriff auf Komponenten der Klasse Bruch

Defaultwert: `private` (!)

Konstruktoren (I)

Konstruktoren werden prinzipiell nicht geerbt

- Spielen aber bei der Initialisierung eine Rolle

Ablauf (*„Top Down Verkettung“*)

- Zuerst wird der Konstruktor der Basisklasse aufgerufen
- Danach der Konstruktor der abgeleiteten Klasse

Sollte die Basisklasse bereits eine abgeleitete Klasse sein, wird zunächst der Konstruktor ihrer Basisklasse aufgerufen

Argumente zur Konstruktion des Objekts werden aber nicht automatisch an den Konstruktor der Basisklasse weitergegeben

- Default Konstruktor der Basisklasse wird verwendet

```
KBruch::KBruch(int z, int n)
{
   if (n == 0)
   {
      cerr << "Fehler: Nenner 0" << endl;
      exit(1);
   }
   /* Nenner soll immer positiv sein */
   if (n < 0)
   { zaehler = -z; nenner = -n; }
   else
   { zaehler = z; nenner = n; }

   kuerzbar = (ggt() > 1);
}
```

Konstruktoren (II)

Argumente werden nicht automatisch weitergegeben …

… aber man kann sie durchreichen

Verwendung von Initialisierungslisten

- Parameter werden dabei an Konstruktor der Basisklasse durchgereicht

In der Klassendefinition werden Default Argumente für die beiden Parameter des KBruch Konstruktors definiert.

- Durchgereicht wird der Wert den der Konstruktor nutzen würde

Initialisierungsliste

```
KBruch::KBruch(int z, int n) : Bruch(z,n)
{
   kuerzbar = (ggt() > 1);
}
```

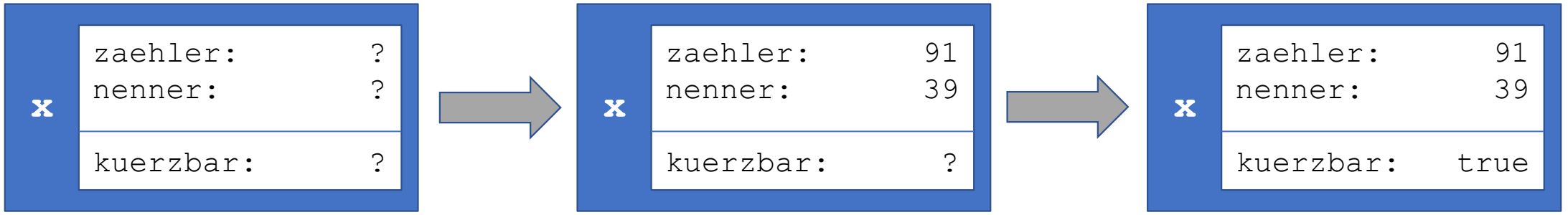
Konstruktoren (III)

```
KBruch x (91,39);
```

Anlegen des Speicherplatzes für das Objekt `x`,
Zustand ist undefiniert.

Aufruf des Konstruktors der Basisklasse Bruch mit den Parametern 91 und 39.
`zaehler` und `nenner` werden initialisiert.

Aufruf des Konstruktors der Klasse KBruch.
Die Variable `kuerzbar` wird initialisiert.



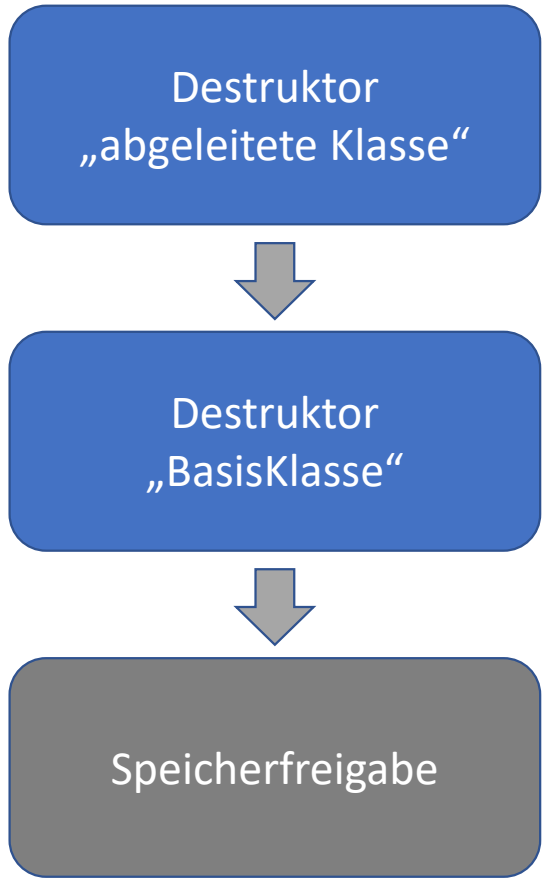
Destruktoren

Destruktoren werden ebenfalls nicht geerbt

Ablauf (*„bottom-up Verkettung“*)

- Zuerst wird der Destruktor der abgeleiteten Klasse aufgerufen
- Danach der Destruktor der Basisklasse

Sollte die Basisklasse bereits eine abgeleitete Klasse sind, wird vor der Speicherfreigabe der Destruktor ihrer Basisklasse aufgerufen



Implementierung der Methoden

```
unsigned KBruch::ggt() const
{
   if (zaehler == 0)
   {
      return(nenner);
   }

   unsigned teiler = min(abs(zaehler),
                         nenner);
   while ((zaehler % teiler != 0) ||
          (nenner % teiler != 0))
   {
      teiler--;
   }
   return(teiler);
}
```

```
void KBruch::kuerzen()
{
   if (kuerzbar)
   {
      int teiler = ggt();

      zaehler /= teiler;
      nenner /= teiler;

      kuerzbar = false;
   }
}
```

Einfacher Test

Auch die Ausgabe über `cout` funktioniert

- Obwohl wir keine neue `operator<<` Funktion implementiert haben
- Obwohl wir hier einfach einen KBruch übergeben

Anwendung der **is_a** Beziehung

- KBruch wurde von Bruch abgeleitet
- Damit ist KBruch auch ein Bruch und kann als solcher verwendet werden

Vererbung liefert eine Art *„Typumwandlung“* von einem Objekt einer abgeleiteten Klasse in ein Objekt seiner Basisklasse.

- Eigenschaften reduziert auf die Basisklasse
- Ignoriert neu hinzu gekommene Komponenten

```
KBruch a (2,6);
KBruch b (25,100);

cout << a << endl << b <<  endl;

a.kuerzen();
b.kuerzen();

cout << a << endl << b << endl;
```

```
2/6
25/100
1/3
1/4
```

Einschränkungen & Lösungen (I)

Einmal funktioniert die Addition, einmal nicht – warum?

- `operator +` wird von Bruch geerbt
  - `a.operator+(b)`
  - Resultat: ein Objekt der Klasse Bruch
- `operator <<` wird von Bruch *„übernommen“*
  - `operator<<(cout, a.operator+(b))`
  - Erwartetes Argument: ein Objekt der Klasse Bruch
- Default Zuweisungsoperator für KBruch
  - `c.operator=(a.operator+(b))`
  - Erwartetes Argument: ein Objekt der Klasse KBruch
  - KBruch ist zwar ein Bruch (*„is_a“*), aber Bruch ist kein KBruch
- Problematik: Initialisierung der zusätzlichen Komponenten von KBruch

```
…

KBruch a (1);
KBruch b (1,5);
KBruch c;

cout << a + b;

c = a + b;

…
```

Compiler: Fehler

Einschränkungen & Lösungen (II)

Lösungsmöglichkeiten

1. Neuimplementierung der `operator+` Methode für die Klasse KBruch
   - *„Überschreiben“* der Methode der Basisklasse
   - Evtl. auch anderer Operatorfunktionen?

2. Neuimplementierung der `operator=` Methode für die Klasse KBruch
   - Argument: Objekt der Klasse Bruch
   - Evtl. noch andere Operatoren/Funktionen?

3. Definition für die Typumwandlung von Bruch nach KBruch
   - Nutzung eines Konstruktors der ein Objekt von KBruch basierend auf einem Objekt der Klasse Bruch initialisiert.

```
KBruch::KBruch(const Bruch& b) : Bruch(b)
{
   kuerzbar = (ggt() > 1);
}
```

Typumwandlung vs. Neuimplementierung

- Typumwandlung
  - Neue Komponenten der abgeleiteten Klasse können über die existierenden Komponenten der Basisklasse initialisiert werden
  - Vorteil: wenig zusätzlicher Code, einfacher zu warten
  - Nachteil: potentiell schlechteres Lauzeitverhalten

- Neuimplementierung / *„Überschreiben“*
  - Neue Komponenten der abgeleiteten Klasse lassen sich **nicht** über die existierenden Komponenten der Basisklasse initialisieren
  - Vorteil: optimiertes Laufzeitverhalten
  - Nachteil: potentiell müssen viele Operatorfunktionen neu implementiert werden

Zusammenfassung

- Klassen können in einer Vererbungsbeziehung stehen.
  - Eine **abgeleitete Klasse** übernimmt (erbt) alle Eigenschaften der **Basisklasse** und ergänzt dies um neue Eigenschaften.
  - Objekte abgeleiteter Klassen besitzen die Komponenten der Basisklasse und die neu hinzugenommenen Komponenten.

- Kennzeichen der Vererbung ist die **is_a Beziehung**
  - Ein Objekt einer abgeleiteten Klasse ist ein Objekt der Basisklasse (mit zusätzlichen Eigenschaften).
  - Ein Objekt einer abgeleiteten Klasse darf jederzeit als Objekt der Basisklasse verwendet werden. Es reduziert sich dann auf die Eigenschaften der Basisklasse.

- Konstruktoren und Destruktoren werden nicht geerbt
  - Konstruktoren werden aber verkettet top-down aufgerufen, Destruktoren entsprechend bottom-up verkettet.
  - Mit **Initialisierungslisten** können den Konstruktoren einer Basisklasse Parameter durchgereicht werden. Erfolgt dies nicht, wird der Default-Konstruktor der Basisklasse aufgerufen.