4.1 Die erste Klasse: Bruch

In diesem Abschnitt wird als erste C++-Klasse die Klasse `Bruch` implementiert und angewendet. Die Klasse `Bruch` bietet dabei ein kleines, überschaubares Beispiel. Der Aufbau eines Bruchs und der Umgang mit Brüchen kann im Wesentlichen als bekannt vorausgesetzt werden. Dennoch gibt es einige Fallen, die in den nachfolgenden Versionen eine Verwendung von nichttrivialen Sprachmitteln erfordern.

Es kann gut sein, dass sich ein Leser, der bereits einiges über C++ weiß, über diese erste Version wundern wird. Mit fortgeschrittenen Kenntnissen der Sprache würde man die Klasse sicherlich anders implementieren. Aber alle Eigenschaften der Sprache können unmöglich auf einmal vorgestellt werden. Im weiteren Verlauf des Buchs wird die Klasse `Bruch` von Version zu Version immer weiter verbessert. Einige Programmiertechniken, die hier als Grundlage aufgezeigt werden, werden aufgrund der gewonnenen Erkenntnisse dann in Frage gestellt oder durch andere Mechanismen ersetzt.

4.1.1 Vorüberlegungen zur Implementierung

Wie bei jeder Klasse müssen zur Implementierung der Klasse `Bruch` zwei Dinge geklärt werden:

- Was soll man mit Objekten der Klasse machen können?
- Welche Daten repräsentieren Objekte dieser Klasse?

Die erste Frage betrifft die Anwendung der Klasse und definiert somit ihre Schnittstelle nach außen. Sie wird anhand der Anforderungen, die an die Klasse gestellt werden, in einer entsprechenden Spezifikation beschrieben.

Die zweite Frage führt zum internen Aufbau der Klasse und ihrer Instanzen. Die Daten müssen in irgendeiner Form als Attribute verwaltet werden. Diese Frage wird vor allem bei der Implementierung der Klasse relevant und muss so beantwortet werden, dass sie die Anforderungen, die sich aus der ersten Frage ergeben, möglichst geschickt lösen kann.

Anforderungen an die Klasse `Bruch`

Für unser erstes Beispiel sollen die Anforderungen an die Klasse `Bruch` möglichst gering gehalten werden. Es soll ja nur das Prinzip verdeutlicht werden. Aus diesem Grund wird nur eine einzige Operation realisiert:

- das Ausgeben eines Bruchs

In den folgenden Abschnitten kommen dann weitere Operationen hinzu, mit denen Brüche z.B. multipliziert oder eingelesen werden können.

Eines darf allerdings nicht vergessen werden: Um mit Brüchen eine Operation ausführen zu können, muss es diese Brüche erst einmal geben. Außerdem sollte ein Bruch, wenn er nicht mehr gebraucht wird, zerstört werden können. Es werden also noch zwei weitere Operationen gebraucht:

- Erzeugen (und Initialisieren) eines Bruchs
- Zerstören eines Bruchs

Insgesamt ergibt sich so die in Abbildung 4.1 dargestellte Schnittstelle.

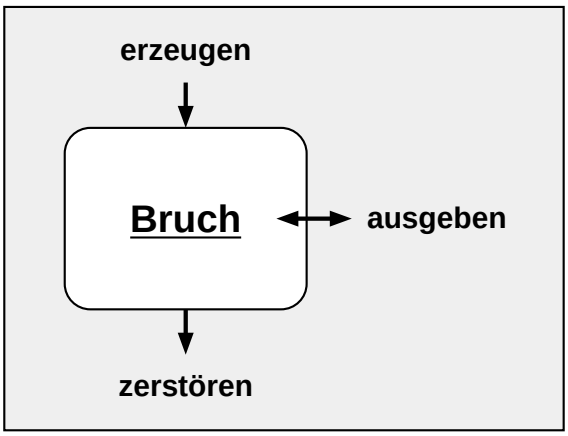


Abbildung 4.1: Erste Schnittstelle der Klasse `Bruch`

Aufbau von Brüchen

Bei der Frage nach den Attributen und dem internen Aufbau eines Objekts geht es letztlich darum, ein solches beliebig abstraktes Gebilde direkt oder indirekt auf bereits vorhandene Datentypen zurückzuführen. Der Informationsgehalt und der Zustand des Objekts müssen durch mehrere einzelne Komponenten, die bereits vorhandene Datentypen besitzen, beschrieben werden.

Dabei ist zunächst einmal die Frage wichtig, wie der Informationsgehalt eines Objekts grundsätzlich beschrieben wird. Was repräsentiert das Objekt? Daraus ergeben sich meist schon die ersten Komponenten, aus denen das Objekt besteht. Im Laufe der Implementierung können dann weitere Hilfskomponenten hinzukommen, die einen internen Objektzustand widerspiegeln und dadurch z.B. die Implementierung erleichtern oder Laufzeit einsparen.

Was ein Bruch repräsentiert bzw. woraus er besteht, kann relativ leicht beantwortet werden: Ein Bruch besteht aus einem Zähler und einem Nenner. Dafür können vorhandene fundamentale Datentypen wie `int` verwendet werden. Damit sind die beiden wesentlichen Komponenten eines Bruchs festgelegt:

- Ein `int` für den Zähler
- Ein `int` für den Nenner

Es kann durchaus sein, dass im Rahmen der Implementierung noch weitere Hilfskomponenten gebraucht werden. So könnte eine Komponente intern z.B. festhalten, ob Zähler und Nenner im aktuellen Zustand gekürzt werden können. Die für den Informationsgehalt wesentlichen Komponenten für den Zähler und den Nenner müssen also nicht unbedingt die einzigen Komponenten bleiben.

Insgesamt ergibt sich der in Abbildung 4.2 dargestellte Aufbau: Ein Bruch besteht aus einem Zähler und einem Nenner und kann erzeugt, ausgegeben und zerstört werden.

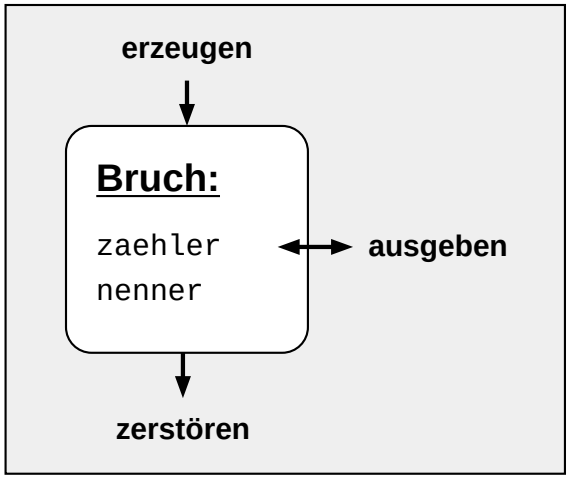


Abbildung 4.2: Schnittstelle und Aufbau der Klasse `Bruch`

Terminologie

In objektorientierter Lesart sind Zähler und Nenner die *Daten* oder *Attribute*, aus denen sich eine Instanz der Klasse `Bruch` zusammensetzt. Das Erzeugen, Ausgeben und Zerstören sind die *Methoden*, die für die Klassen definiert sind.

In der Lesart von C++ bedeutet das, dass eine Klasse `Bruch` gebraucht wird, die sich aus den *Komponenten* Zähler und Nenner sowie den Funktionen zum Erzeugen, Ausgeben und Zerstören zusammensetzt. Die Komponenten einer Klasse werden auch *Elemente* oder *Klassenelemente* genannt. Dies alles sind verschiedene Übersetzungen der englischen Bezeichnung *member*, die mitunter auch im Deutschen verwendet wird (siehe dazu auch die Anmerkungen zur Namensgebung auf Seite 22).

Die Methoden, also die Operationen, die in einer Klasse definiert sind, nennt man in C++ *Elementfunktionen*. Eine andere weit verbreitete Bezeichnung dafür ist *Memberfunktion*.

Für die Komponenten, die keine Elementfunktionen sind, gibt es in C++ eigentlich gar keine richtige Bezeichnung. Man spricht im Englischen im Allgemeinen nur von *member* und dem Spezialfall *member function*. Ich werde sie nachfolgend als *Attribute* oder *Datenelemente* bezeichnen.

Die Komponenten der Klasse `Bruch` unterteilen sich also in die Datenelemente Zähler und Nenner und die Elementfunktionen zum Erzeugen, Ausgeben und Zerstören.

Datenkapselung

Brüche besitzen eine grundsätzliche Eigenschaft: Ihr Nenner darf nicht 0 sein, da durch 0 nicht geteilt werden darf. Diese Randbedingung soll natürlich auch für die Klasse `Bruch` gelten.

Wenn aber für alle Anwender unbeschränkter Zugriff auf alle Komponenten der Klasse `Bruch` besteht, ist es möglich, den Nenner eines Bruchs auf 0 zu setzen. Aus diesem Grund ist es sinnvoll, bei der Verwendung eines Bruchs keinen direkten Zugriff auf den Nenner zu ermöglichen.

Solange der Zugriff nur über Funktionen möglich ist, kann der Versuch, dem Nenner 0 zuzuweisen, mit einer entsprechenden Fehlermeldung quittiert werden.

Generell ist es eine gute Idee, den Zugriff auf Objekte nur über dafür definierte Operationen zuzulassen. Durch deren Implementierung können nicht nur Fehler gefunden, sondern auch Inkonsistenzen vermieden werden. Wird für einen Bruch z.B. intern in einer Hilfskomponente festgehalten, ob er kürzbar ist, kann bei einer Änderung des Sachverhalts durch eine Zuweisung des Zählers von außen intern eine entsprechende Anpassung stattfinden.

Für Brüche ist es deshalb sinnvoll, den Zugriff auf die Komponenten Zähler und Nenner nur über die zur Klasse `Bruch` gehörenden Funktionen zu erlauben. Damit wird auch ein anderer Vorteil erreicht: Der Aufbau und das Verhalten eines Bruchs können intern modifiziert werden, ohne dass sich die Schnittstelle nach außen ändert.

4.1.2 Deklaration der Klasse `Bruch`

Um eine Klasse in C++ verwenden zu können, muss sie zunächst deklariert werden. Dies geschieht, indem eine Struktur deklariert wird, in der die prinzipiellen Eigenschaften der Klasse aufgelistet werden. Sie enthält sowohl die besprochenen Komponenten für den Zähler und den Nenner als auch die Funktionen, die die Schnittstelle ausmachen.

Damit die Deklaration zum Kompilieren aller Module, in denen die Klasse `Bruch` gebraucht wird, verwendet werden kann, wird sie in einer Headerdatei untergebracht. Sinnvollerweise trägt die Datei den Namen der Klasse und erhält deshalb am besten den Namen `bruch.hpp`.

Die erste Version der Headerdatei `bruch.hpp` hat insgesamt folgenden Aufbau, auf den anschließend im Einzelnen eingegangen wird:

```
// klassen/bruch1.hpp

#ifndef BRUCH_HPP
#define BRUCH_HPP

// **** BEGINN Namespace Bsp **********************************
namespace Bsp {

/* Klasse Bruch
 */
class Bruch {

  /* privat: kein Zugriff von außen
   */
  private:
    int zaehler;
    int nenner;

  /* öffentliche Schnittstelle
   */
```

```
  public:
    // Default-Konstruktor
    Bruch();

    // Konstruktor aus int (Zähler)
    Bruch(int);

    // Konstruktor aus zwei ints (Zähler und Nenner)
    Bruch(int, int);

    // Ausgabe
    void print();
};

} // **** ENDE Namespace Bsp **********************************

#endif    /* BRUCH_HPP */
```

Präprozessor-Anweisungen

Die komplette Headerdatei wird durch Präprozessor-Anweisungen eingeschlossen, mit deren Hilfe vermieden wird, dass die darin befindlichen Deklarationen bei mehrfachem Einbinden der Datei mehrfach durchgeführt werden:

```
#ifndef BRUCH_HPP        // ist nur beim ersten Durchlauf erfüllt, denn
#define BRUCH_HPP        // BRUCH_HPP wird beim ersten Durchlauf definiert
    ...
#endif
```

Mit `#ifndef` („if not defined“) werden die folgenden Zeilen bis zum korrespondierenden `#endif` nur verarbeitet, wenn die Konstante `BRUCH_HPP` nicht definiert ist. Da sie bei der ersten Verarbeitung definiert wird, werden die Anweisungen bei jedem weiteren Mal ignoriert.

Da es immer passieren kann, dass Headerdateien über unterschiedliche Wege von einer Quelldatei mehrfach eingebunden werden, sollten Deklarationen in Headerdateien immer durch derartige Anweisungen eingeschlossen werden.

4.1.3 Die Klassenstruktur

Die eigentliche Deklaration einer Klasse erfolgt in einer so genannten *Klassenstruktur*. Dort werden alle Komponenten (Attribute und Funktionen) der Klasse deklariert.

Das Schlüsselwort `class`

Die Deklaration beginnt mit dem Schlüsselwort `class`. Es folgen ein Symbol für den Namen der Klasse und der in geschweifte Klammern eingeschlossene und durch ein Semikolon abgeschlossene Klassenrumpf, in dem die Eigenschaften der Klasse deklariert werden:

```
class Bruch {
    ...
};
```

Wenn man eine Bibliothek definiert, die aus mehreren Klassen besteht, stellt jeder Klassenname ein globales Symbol dar, das überall verwendet werden kann. Dies kann bei der Verwendung unterschiedlicher Bibliotheken schnell zu Namenskonflikten führen. Aus diesem Grund wird die Klasse `Bruch` innerhalb eines *Namensbereiches* (englisch: *namespace*) deklariert:

```
namespace Bsp {      // Beginn Namensbereich Bsp

class Bruch {
    ...
};

}                    // Ende Namensbereich Bsp
```

Damit befindet sich lediglich das Symbol `Bsp` im globalen Gültigkeitsbereich. Genau genommen definieren wir also „`Bruch` in `Bsp`“. Die Notation, um diese Klasse anzusprechen, lautet in C++ `Bsp::Bruch`.

Alle Klassen und auch andere Symbole sollte man immer einem Namensbereich zuordnen. Dies vermeidet nicht nur Konflikte, sondern macht auch deutlich, welche Elemente eines Programms logisch zusammengehören. In der objektorientierten Modellierung bezeichnet man eine derartige Gruppierung auch als *Paket* (englisch: *package*).

Weitere Details zu Namensbereichen werden in Abschnitt 3.3.8 und Abschnitt 4.3.5 erläutert.

Zugriffsschlüsselwörter

Innerhalb der Klassenstruktur befinden sich Zugriffsschlüsselwörter, die die einzelnen Komponenten gruppieren. Damit wird festgelegt, welche Komponenten der Klasse intern sind und auf welche auch von außen, bei der Anwendung der Klasse, zugegriffen werden kann.

Die Komponenten `zaehler` und `nenner` sind privat:

```
class Bruch {
  private:
    int zaehler;
    int nenner;
    ...
};
```

Durch das Schlüsselwort `private` werden alle folgenden Komponenten als privat deklariert. In Klassen ist dies zwar die Defaulteinstellung, es sollte zur besseren Lesbarkeit aber dennoch explizit herausgestellt werden.

Die Tatsache, dass die Komponenten privat sind, bedeutet, dass auf sie nur im Rahmen der Implementierung der Klasse Zugriff besteht. Der Anwender, der einen Bruch verwendet, hat keinen direkten Zugriff darauf. Er kann diesen nur indirekt erhalten, indem entsprechende Funktionen zur Verfügung gestellt werden.

Zugriff hat der Anwender der Klasse `Bruch` nur auf die öffentlichen Komponenten. Diese werden durch das Schlüsselwort `public` gekennzeichnet. Es handelt sich typischerweise um die Funktionen, die die Schnittstelle der Objekte dieser Klasse nach außen zum Anwender bilden:

```
class Bruch {
  ...
  public:
    void print ();
};
```

Es ist nicht zwingend, dass alle Elementfunktionen öffentlich und alle anderen Komponenten privat sind. Bei größeren Klassen ist es durchaus nicht untypisch, dass es klasseninterne Hilfsfunktionen gibt, die als private Elementfunktionen deklariert werden.

Genauso ist es prinzipiell möglich, Komponenten, die keine Funktionen sind, mit einem öffentlichen Zugriff zu versehen. Dies ist aber im Allgemeinen nicht sinnvoll, da damit der Vorteil, die Objekte nur über eine wohldefinierte Schnittstelle manipulieren zu können, aufgehoben wird.

Zugriffsdeklarationen können in einer Klassendeklaration beliebig oft auftreten, wodurch der Zugriff auf die folgenden Komponenten mehrfach wechseln kann:

```
class Bruch {
  private:
    ...
  public:
    ...
  private:
    ...
};
```

Strukturen und Klassen

Aus der Sicht eines C-Programmierers kann eine Klasse als eine C-Struktur betrachtet werden, die einfach nur mit zusätzlichen Eigenschaften ausgestattet wurde (Zugriffskontrolle und Funktionen als Komponenten).

Es ist sogar so, dass die Eigenschaften von Klassen in C++ auch für Strukturen eingeführt werden. Statt des Schlüsselwortes `class` kann deshalb auch immer das Schlüsselwort `struct` verwendet werden. Der einzige Unterschied ist, dass mit dem Schlüsselwort `struct` alle Komponenten defaultmäßig öffentlich sind. Jede C-Struktur ist im C++-Sinne also eine Klasse mit lauter öffentlichen Komponenten.

In der Praxis empfehle ich, um unnötige Verwirrung zu vermeiden, für Klassen immer das Schlüsselwort `class` zu verwenden. Lediglich wenn in C++ Strukturen im herkömmlichen C-Sinne eingesetzt werden (ohne Funktionen als Komponenten und mit öffentlichem Zugriff auf alle Komponenten), verwende ich zur Abgrenzung das Schlüsselwort `struct`. Leider gibt es Literatur, in der selbst für Beispiele zur Programmierung mit Klassen das Schlüsselwort `struct` verwendet wird.

4.1.4 Elementfunktionen

Eine Komponente, die in der Struktur der Klasse `Bruch` deklariert wird, ist die Elementfunktion `print()`:

```
class Bruch {
    ...
    void print ();          // Ausgabe eines Bruchs
    ...
};
```

Sie dient dazu, einen Bruch auszugeben. Da sie keinen Wert zurückliefert, hat sie den Rückgabetyp `void`.

Man beachte dabei, dass der Bruch, der jeweils ausgegeben werden soll, nicht als Parameter übergeben wird. Er ist in einer Elementfunktion automatisch vorhanden, da der Aufruf einer Elementfunktion immer an ein bestimmtes Objekt dieser Klasse gebunden ist:

```
Bsp::Bruch x;
...
x.print();        // Elementfunktion print() für Objekt x aufrufen
```

Die Funktion `print()` wird mit dem Punkt-Operator für einen Bruch ohne (weitere) Parameter aufgerufen. Im objektorientierten Sprachgebrauch wird durch den Aufruf von `x.print()` dem *Objekt* `x` als *Instanz* der Klasse `Bruch` die *Nachricht* `print` ohne Parameter geschickt.

4.1.5 Konstruktoren

Die ersten drei Funktionen, die in der Klasse Bruch deklariert werden, sind spezielle Funktionen. Sie legen fest, auf welche Weise ein Bruch erzeugt („konstruiert“) werden kann. Ein solcher *Konstruktor* (englisch: *constructor*) ist daran zu erkennen, dass er als Funktionsnamen den Namen der Klasse trägt.

Aufgerufen wird ein Konstruktor immer dann, wenn eine Instanz (ein konkretes Objekt) der entsprechenden Klasse angelegt wird. Das ist z.B. beim Deklarieren einer entsprechenden Variablen der Fall. Nachdem für das Objekt der notwendige Speicherplatz angelegt wurde, werden die Anweisungen im Konstruktor ausgeführt. Sie dienen typischerweise dazu, das angelegte Objekt zu initialisieren, indem die Datenelemente des Objekts mit sinnvollen Startwerten versehen werden.

Bei der Klasse `Bruch` werden drei Konstruktoren deklariert:

```
class Bruch {
    ...
    // Default-Konstruktor
    Bruch ();

    // Konstruktor aus int (Zähler)
    Bruch (int);
```

```
    // Konstruktor aus zwei ints (Zähler und Nenner)
    Bruch (int, int);
    ...
};
```

Das bedeutet, dass ein Bruch auf dreierlei Arten angelegt werden kann:

- Ein Bruch kann ohne Argument angelegt werden.
  Der Konstruktor wird z.B. aufgerufen, wenn ein Objekt der Klasse `Bruch` ohne weitere Parameter deklariert wird:

  ```
  Bsp::Bruch x;          // Initialisierung mit Default-Konstruktor
  ```

  Einen Konstruktor ohne Parameter nennt man auch *Default-Konstruktor*.

- Ein Bruch kann mit einem Integer als Argument erzeugt werden. Wie wir bei der Implementierung des Konstruktors noch sehen werden, wird dieser Parameter als ganze Zahl interpretiert, mit der der Bruch initialisiert wird.
  Ein solcher Konstruktor wird z.B. durch eine Deklaration aufgerufen, bei der der Bruch mit einem Objekt des passenden Typs initialisiert wird:

  ```
  Bsp::Bruch y = 7;      // Initialisierung mit int-Konstruktor
  ```

  Diese Art der Initialisierung wurde von C übernommen.
  In C++ wurde aber auch eine neue Schreibweise zur Initialisierung eines Objekts eingeführt, die genauso verwendet werden kann:

  ```
  Bsp::Bruch y(7);       // Initialisierung mit int-Konstruktor
  ```

- Ein Bruch kann mit zwei ganzzahligen Argumenten erzeugt werden. Diese Parameter werden zum Initialisieren von Zähler und Nenner verwendet.
  Um einen solchen Konstruktor aufrufen zu können, wurde die gerade vorgestellte neue Schreibweise zur Initialisierung eines Objekts eingeführt, denn sie erlaubt es, mehrere Argumente zu übergeben:

  ```
  Bsp::Bruch w(7,3);     // Initialisierung mit int/int-Konstruktor
  ```

Konstruktoren haben noch eine ungewöhnliche Besonderheit: Sie besitzen keinen Rückgabetyp (auch nicht `void`). Es sind also keine Funktionen oder Prozeduren im herkömmlichen Sinne.

Klassen ohne Konstruktoren

Werden für eine Klasse keine Konstruktoren definiert, können trotzdem Instanzen (konkrete Objekte) der Klasse angelegt werden. Diese Objekte werden dann allerdings nicht initialisiert. Es gibt also sozusagen einen vordefinierten Default-Konstruktor, der nichts macht.

Damit Objekte keinen undefinierten Zustand erhalten, sollte dieser Fall unbedingt vermieden werden. Wenn es den Zustand „*nicht initialisiert*“ geben soll, dann sollte stattdessen eine Boolesche Komponente eingeführt werden, die diesen Sachverhalt über den Konstruktor definitiv festhält. Werden dann Operationen für einen Bruch mit nicht definiertem Wert aufgerufen, können entsprechende Fehlermeldungen ausgegeben werden. Bei einem Bruch, der, da er nicht initialisiert wurde, einen beliebigen Zustand besitzen kann, ist das nicht möglich.

Sobald Konstruktoren definiert werden, können Objekte nur noch über diese angelegt werden. Wird kein Default-Konstruktor, wohl aber ein anderer Konstruktor definiert, ist ein Anlegen eines Objekts der Klasse ohne Parameter also nicht mehr möglich. Auf diese Weise kann die Übergabe von Werten zur Initialisierung erzwungen werden.

4.1.6 Überladen von Funktionen

Die Tatsache, dass – wie im Fall der Konstruktoren – mehrere Funktionen den gleichen Namen besitzen dürfen, ist eine grundsätzliche Eigenschaft von C++. Funktionen (nicht nur Elementfunktionen) dürfen beliebig *überladen* werden.

Das Überladen (englisch: *overloading*) von Funktionen bedeutet, dass diese den gleichen Namen tragen und sich nur in ihren Parametern unterscheiden. Welche Funktion aufzurufen ist, wird durch Anzahl und Typ der Parameter entschieden. Eine Unterscheidung nur durch den Rückgabetyp ist nicht zulässig.

Das folgende Beispiel zeigt das Überladen einer globalen Funktion zum Berechnen des Quadrats für Integer und Gleitkommawerte:

```
// Deklaration
int    quadrat (int);              // Quadrat eines ints
double quadrat (double);           // Quadrat eines doubles

void f ()
{
    // Aufruf
    quadrat(713);                  // ruft Quadrat eines ints auf
    quadrat(4.378);                // ruft Quadrat eines doubles auf
}
```

Die Funktion ließe sich im gleichen Programm zusätzlich für die Klasse `Bruch` überladen:

```
// Deklaration
int        quadrat (int);          // Quadrat eines ints
double     quadrat (double);       // Quadrat eines doubles
Bsp::Bruch quadrat (Bsp::Bruch);   // Quadrat eines Bruchs

void f ()
{
    Bsp::Bruch b;
    ...
    // Aufruf
    quadrat(b);                    // ruft Quadrat eines Bruchs auf
}
```

Beim Überladen von Funktionen sollte natürlich darauf geachtet werden, dass die überladenen Funktionen im Prinzip das Gleiche machen. Da jede Funktion für sich implementiert wird, liegt dies in der Hand des Programmierers.

Parameter-Prototypen

Damit eine Unterscheidung der Funktionen über die Parameter überhaupt möglich ist, müssen diese mit Parameter-Prototypen deklariert werden. Das bedeutet, der Typ eines Parameters muss bei der Deklaration angegeben werden und steht bei der Definition einer Funktion direkt in der Liste der Parameter:

```
// Deklaration
int quadrat (int, int);

// Definition
int quadrat (int a, int b)
{
    return a * b;
}
```

Parameter-Prototypen können auch in ANSI-C definiert werden. In C brauchten die Parameter bei einer Deklaration allerdings nicht unbedingt angegeben zu werden. Im Vergleich zu C gibt es deshalb einen wichtigen Unterschied: Die Deklaration

```
void f ();
```

definiert in C eine Funktion mit beliebig vielen Parametern. In C++ bedeutet dies aber, dass die Funktion auf jeden Fall *keine* Parameter besitzt. Die Schreibweise von ANSI-C zur Deklaration einer Funktion ohne Parameter

```
void f (void);
```

kann in C++ ebenfalls verwendet werden, ist aber unüblich.

Falls eine Funktion eine variable Anzahl von Argumenten besitzt, kann dies in der Deklaration durch drei Punkte (mit oder ohne Komma davor) angegeben werden:

```
int printf (char*, ...);
```

Eine solche Angabe nennt man im englischen *Ellipsis*, was man im Deutschen etwa mit *Auslassung* oder *Unvollständigkeit* übersetzen kann.[1]

4.1.7 Implementierung der Klasse `Bruch`

Die in der Deklaration der Klasse `Bruch` deklarierten Elementfunktionen müssen natürlich auch implementiert werden. Dies geschieht typischerweise in einem eigenen Modul, das für die Klasse angelegt wird. Es trägt den Namen `Bruch.cpp`.

Es ist von Vorteil, für jede Klasse eine eigene Quelldatei zu verwenden. Damit müssen Programme nur die Module der Klassen dazubinden, die auch verwendet werden. Zusätzlich unterstützt diese Vorgehensweise natürlich auch die Idee der objektorientierten Programmierung,

[1] In vielen Büchern wird der Begriff „ellipsis“ auch als „Ellipse“ übersetzt, womit offensichtlich nicht das mathematische Objekt, sondern die sprachwissenschaftliche Bezeichnung für eine „Ersparung von Redeteilen“ gemeint ist (ein Duden ist manchmal eine wirklich interessante Sache).

da jede Art von Objekt für sich betrachtet wird (es gibt natürlich auch gute Gründe, von dieser Regel abzuweichen, etwa wenn zwei Klassen logisch und technisch sehr eng zusammengehören).

Die Quelldatei der ersten Version der Klasse `Bruch` sieht wie folgt aus:

```
// klassen/bruch1.cpp

// Headerdatei mit der Klassen-Deklaration einbinden
#include "bruch.hpp"

// Standard-Headerdateien einbinden
#include <iostream>
#include <cstdlib>

// **** BEGINN Namespace Bsp *********************************
namespace Bsp {

/* Default-Konstruktor
 */
Bruch::Bruch ()
 : zaehler(0), nenner(1)      // Bruch mit 0 initialisieren
{
    // keine weiteren Anweisungen
}

/* Konstruktor aus ganzer Zahl
 */
Bruch::Bruch (int z)
 : zaehler(z), nenner(1)      // Bruch mit z 1tel initialisieren
{
    // keine weiteren Anweisungen
}

/* Konstruktor aus Zähler und Nenner
 */
Bruch::Bruch (int z, int n)
 : zaehler(z), nenner(n)      // Zähler und Nenner wie übergeben initialisieren
{
    // 0 als Nenner ist allerdings nicht erlaubt
    if (n == 0) {
        // Programm mit Fehlermeldung beenden
        std::cerr << "Fehler: Nenner ist 0" << std::endl;
        std::exit(EXIT_FAILURE);
    }
}
```

```
/* print
 */
void Bruch::print ()
{
    std::cout << zaehler << '/' << nenner << std::endl;
}

} // **** ENDE Namespace Bsp ********************************
```

Am Anfang wird die Deklaration der Klasse eingebunden, deren Elementfunktionen hier definiert werden:

```
#include "bruch.hpp"
```

Dies ist notwendig, damit der Compiler z.B. weiß, welche Komponenten die Klasse `Bruch` besitzt.

Anschließend werden verschiedene andere Headerdateien eingebunden, die die Datentypen und die Funktionen deklarieren, die von der Klasse verwendet werden. In diesem Fall sind das zwei Dateien:

```
#include <iostream>
#include <cstdlib>
```

Die erste Include-Anweisung bindet die Standardelemente von C++ für Ein- und Ausgaben ein (siehe Seite 29 und Seite 202). Die zweite Include-Anweisung bindet eine Headerdatei ein, in der verschiedene Standardfunktionen deklariert werden, die auch schon in C zur Verfügung stehen. In C trägt diese Datei den Namen `<stdlib.h>`. Damit die Symbole zum Namensbereich der Standardbibliothek `std` gehören, werden sie für C++ in der Datei `<cstdlib>` deklariert. Generell stehen die Standardfunktionen von C auch in C++ zur Verfügung. Die dazugehörigen Headerdateien haben statt der Endung `.h` das Präfix `c`.

In diesem Programm wird `<cstdlib>` konkret für die Funktion `std::exit()` und die dazugehörige Konstante `EXIT_FAILURE` eingebunden. Mit `std::exit()` kann man ein Programm im Fehlerfall abbrechen (siehe Seite 128).

Man beachte, dass die Include-Anweisungen zwei unterschiedliche Formen haben. Im einen Fall wird die einzubindende Datei in doppelten Anführungszeichen, im anderen Fall in spitzen Klammern eingeschlossen:

```
#include "bruch.hpp"
#include <iostream>
```

Die Form mit den spitzen Klammern ist für externe Dateien vorgesehen. Die dazugehörigen Dateien werden in den Verzeichnissen gesucht, die vom Compiler als System-Verzeichnis angesehen werden.[2] Die in doppelten Anführungsstrichen angegebenen Dateien werden zusätzlich

[2] Auf Unix-Systemen kann man diese Verzeichnisse meistens mit der Option `-I` definieren. Microsoft Visual C++ bietet dazu die Möglichkeit der Angabe „zusätzlicher Include-Verzeichnisse“.

auch im lokalen Verzeichnis gesucht. Falls sie dort nicht gefunden werden, wird ebenfalls in den System-Verzeichnissen nachgesehen (siehe auch Seite 56).

Der Bereichsoperator

Beim Betrachten der Funktionen fällt zunächst auf, dass vor sämtlichen Funktionsnamen der Ausdruck

```
Bruch::
```

steht. Der Operator `::` ist der *Bereichsoperator*. Er ordnet einem nachfolgenden Symbol den Gültigkeitsbereich der davorstehenden Klasse zu. Er besitzt höchste Priorität.

In diesem Fall wird damit zum Ausdruck gebracht, dass es sich jeweils um eine Elementfunktion der Klasse `Bruch` handelt. Das bedeutet, dass die Funktion nur für Objekte dieser Klasse aufgerufen werden kann. Damit ist aber gleichzeitig definiert, dass es innerhalb der Funktionen immer einen Bruch gibt, für den die Funktion aufgerufen wurde und auf dessen Komponenten zugegriffen werden kann.

Implementierung der Konstruktoren

Zunächst werden die Konstruktoren implementiert. Sie sind, wie schon erwähnt, daran zu erkennen, dass der Funktionsname gleich dem Klassennamen ist, also `Bruch::Bruch` lautet.

Der Default-Konstruktor initialisiert den Bruch mit `0` (genauer mit $\frac{0}{1}$):

```
Bruch::Bruch ()
 : zaehler(0), nenner(1)      // Bruch mit 0 initialisieren
{
    // keine weiteren Anweisungen
}
```

Bei jedem Erzeugen eines Bruchs ohne Argumente wird dieser Konstruktor aufgerufen. Damit wird sichergestellt, dass auch ein Bruch, der ohne Argumente zur Initialisierung angelegt wird, keinen undefinierten Wert besitzt.

An dieser Stelle zeigt sich gleich eine Besonderheit der Konstruktoren. Da sie zur Initialisierung der Objekte dienen, verfügen sie über die Möglichkeit, noch vor den eigentlichen Anweisungen der Funktion die initialen Werte der Komponenten festzulegen. Getrennt durch einen Doppelpunkt und vor dem eigentlichen Funktionskörper können für jedes Attribut die initialen Werte angegeben werden. Es handelt sich dabei um eine so genannte *Initialisierungsliste*. Sie wirkt so, als hätte man für Brüche, die unter diesen Umständen angelegt werden, die folgenden Deklarationen in Bezug auf ihre Komponenten getätigt:

```
int zaehler(0);
int nenner(1);
```

Alternativ hätte man die Werte auch mit Hilfe herkömmlicher Anweisungen zuweisen können:

```
Bruch::Bruch ()
{
    zaehler = 0;
    nenner = 1;
}
```

Es gibt jedoch kleine Unterschiede in der Semantik dieser beiden Formen – so wie es Unterschiede zwischen

```
int x = 0;      // x anlegen und dabei sofort mit 0 initialisieren
```

bzw.

```
int x(0);       // x anlegen und dabei sofort mit 0 initialisieren
```

auf der einen und

```
int x;          // x anlegen
x = 0;          // in einem separaten Schritt 0 zuweisen
```

auf der anderen Seite gibt. Diese Unterschiede spielen hier zwar noch keine Rolle; man sollte sich aber gleich an die Syntax mit der Initialisierungsliste gewöhnen.

Der zweite Konstruktor wird aufgerufen, wenn beim Erzeugen des Objekts ein Integer übergeben wird. Dieser Parameter wird als ganze Zahl interpretiert. Entsprechend wird der Zähler mit dem übergebenen Parameter und der Nenner mit 1 initialisiert:

```
Bruch::Bruch (int z)
 : zaehler(z), nenner(1)      // Bruch mit z 1tel initialisieren
{
    // keine weiteren Anweisungen
}
```

Der dritte Konstruktor erhält zwei Parameter, die er zum Initialisieren von Zähler und Nenner verwendet. Dieser Konstruktor wird immer dann aufgerufen, wenn beim Anlegen eines Bruchs zwei Integer übergeben werden. Falls als Nenner allerdings 0 übergeben wird, wird das Programm mit einer entsprechenden Fehlermeldung beendet:

```
Bruch::Bruch (int z, int n)
 : zaehler(z), nenner(n)      // Zähler und Nenner wie übergeben initialisieren
{
    // 0 als Nenner ist allerdings nicht erlaubt
    if (n == 0) {
        // Programm mit Fehlermeldung beenden
        std::cerr << "Fehler: Nenner ist 0" << std::endl;
        std::exit(EXIT_FAILURE);
    }
}
```

Fehlerbehandlung in Konstruktoren

Die Tatsache, dass in diesem Beispiel bei einer fehlerhaften Initialisierung das Programm beendet wird, ist sehr radikal. Es wäre besser, den Fehler zu melden und abhängig von der Programmsituation darauf zu reagieren. Das Problem ist nur, dass Konstruktoren keine herkömmlichen Funktionen sind, die explizit aufgerufen werden und einen Rückgabewert zurückliefern können. Sie werden implizit durch Deklarationen aufgerufen – und Deklarationen haben keinen Rückgabewert.

Das Programm wird deshalb mangels besserer Alternativen abgebrochen. Um einen solchen Programmabbruch zu vermeiden, muss also in allen Programmen, in denen die Klasse verwendet wird, darauf geachtet werden, dass als Nenner nicht 0 übergeben wird.

Es gibt natürlich noch andere Alternativen zum Abbruch. So könnte der Bruch auch einen Default-Wert erhalten. Dies erscheint aber nicht sinnvoll, da damit ein offensichtlicher Fehler (Nenner dürfen nicht 0 sein) nicht behandelt, sondern einfach nur ignoriert wird.

Es gibt auch die Möglichkeit, eine weitere Hilfskomponente für Brüche einzuführen, die festhält, dass der Bruch noch nicht korrekt initialisiert wurde. Diese Komponente müsste dann aber auch ausgewertet und bei jeder Operation beachtet werden.

Die beste Möglichkeit, mit solchen Fehlern umzugehen, stellt das C++-Konzept zur Ausnahmebehandlung dar, das bereits in Abschnitt 3.6 vorgestellt wurde. Dieses Konzept erlaubt es, auch bei einer Deklaration eine Fehlerbehandlung auszulösen, die aber nicht unbedingt zum Abbruch führen muss, sondern von der Programmumgebung, in der sie auftritt, abgefangen und entsprechend der aktuellen Situation behandelt werden kann. Wir werden die Klasse `Bruch` in Abschnitt 4.7 auf diesen Mechanismus umstellen.

Implementierung von Elementfunktionen

Als Nächstes folgt die Definition der Elementfunktion `print()`, die einen Bruch ausgibt. Dazu gibt diese Funktion einfach den Zähler und den Nenner mit Hilfe der Techniken der C++-Standardbibliothek auf dem Standard-Ausgabekanal, `std::cout`, in der Form *„zähler/nenner“* aus:

```
void Bruch::print ()
{
    std::cout << zaehler << '/' << nenner << std::endl;
}
```

Da es sich um eine Elementfunktion handelt, muss sie immer für ein bestimmtes Objekt aufgerufen werden, damit `zaehler` und `nenner` zugeordnet werden können. Dies bedeutet, dass es bei Elementfunktionen eine neue Art von Gültigkeitsbereich gibt, der bei der Zuordnung von Symbolen berücksichtigt wird: den Gültigkeitsbereich der Klasse, zu der die Funktion gehört. Die Deklaration von einem in einer Elementfunktion verwendeten Symbol wird zunächst im lokalen Gültigkeitsbereich der Funktion, dann in der Klassendeklaration und schließlich im globalen Gültigkeitsbereich gesucht.

Durch `w.print()` wird diese Funktion für das Objekt `w` aufgerufen. Daraus folgt, dass in diesem Fall in der Funktion mit `zaehler` der Zähler von `w`, also `w.zaehler`, angesprochen wird. Bei einem Aufruf der Funktion durch `x.print()` wird entsprechend `x.zaehler` angesprochen.

Man beachte, dass die Komponente `zaehler` im Anwendungsprogramm nicht direkt angesprochen werden kann, da sie als privat deklariert ist. Nur die Elementfunktionen einer Klasse haben Zugriff auf die privaten Komponenten dieser Klasse.

4.1.8 Anwendung der Klasse `Bruch`

Die vorgestellte erste Version der Klasse `Bruch` kann bereits in einem Anwendungsprogramm eingesetzt werden.

Dazu muss in der entsprechenden Quelldatei die Headerdatei der Klasse eingebunden werden. Aufgrund der Typbindung von C++ wird ansonsten die Deklaration einer Variablen vom Typ Bruch abgelehnt. Der Compiler braucht außerdem Kenntnis über die Komponenten eines Bruchs, damit entsprechend viel Speicherplatz angelegt werden kann. Mit Hilfe der Klassendeklaration wird auch geprüft, ob alle Zugriffe auf Objekte der Klasse `Bruch` zulässig sind.

Unter der Annahme, dass das Anwendungsprogramm `btest.cpp` heißt, ergibt sich das in Abbildung 4.3 dargestellte Übersetzungsschema: Die Headerdatei `bruch.hpp` mit der Klassendeklaration wird von der Quelldatei mit der Klassenimplementierung und der Quelldatei mit dem Anwendungsprogramm eingebunden. Die beiden Quelldateien werden vom C++-Compiler zunächst in Objektdateien übersetzt (diese haben typischerweise die Endung `.o` oder `.obj`). Von einem Linker werden die Objektdateien dann zum ausführbaren Programm `btest` (oder `btest.exe`) zusammengebunden. Das Übersetzen und Binden ist auf den meisten Systemen auch in einem Schritt möglich.

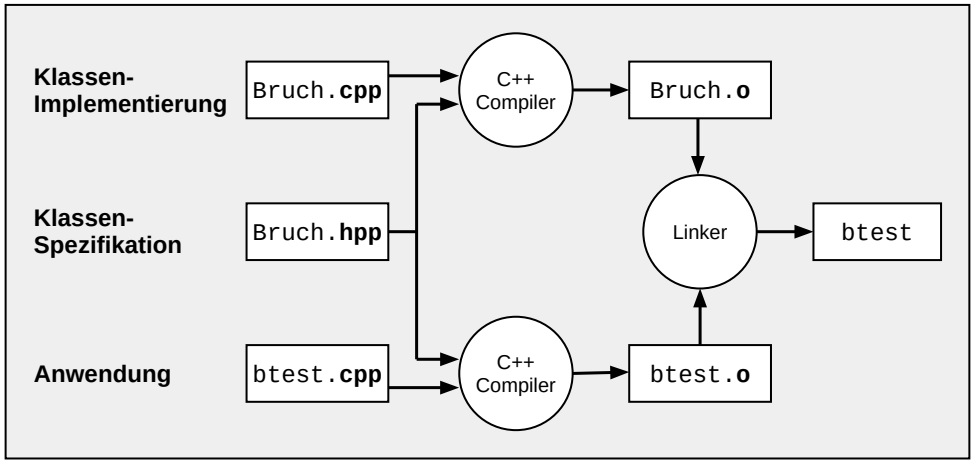


Abbildung 4.3: Übersetzungsschema für die Klasse `Bruch`

Das folgende Programm ist ein erstes Anwendungsbeispiel der ersten Version der Klasse `Bruch` und zeigt, was mit Brüchen bereits alles gemacht werden kann:

```
// klassen/btest1.cpp

// Headerdateien für die verwendeten Klassen einbinden
#include "bruch.hpp"

int main()
{
    Bsp::Bruch x;          // Initialisierung durch Default-Konstruktor
    Bsp::Bruch w(7,3);     // Initialisierung durch int/int-Konstruktor

    // Bruch w ausgeben
```

```
    w.print();

    // Bruch w wird an Bruch x zugewiesen
    x = w;

    // 1000 in einen Bruch umwandeln und w zuweisen
    w = Bsp::Bruch(1000);

    // x und w ausgeben
    x.print();
    w.print();
}
```

Die Ausgabe des Programms lautet:

```
7/3
7/3
1000/1
```

Deklarationen

Im Programm werden zunächst die Variablen `x` und `w` deklariert:

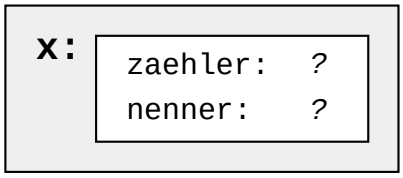
```
Bsp::Bruch x;          // Initialisierung durch Default-Konstruktor
Bsp::Bruch w(7,3);     // Initialisierung durch int/int-Konstruktor
```

Der Vorgang, der hier stattfindet, kann unterschiedlich benannt werden:

- Im nicht-objektorientierten Sprachgebrauch werden zwei *Variablen* vom *Typ* `Bsp::Bruch` angelegt.
- In objektorientierter Lesart werden zwei *Instanzen* (konkrete Objekte) der *Klasse* `Bruch` angelegt.

Konkret wird sowohl für `x` als auch für `w` Speicherplatz angelegt und der entsprechende Konstruktor aufgerufen. Dies geschieht für `x` in folgenden Schritten:

- Zunächst wird der Speicherplatz für das Objekt angelegt. Dessen Zustand ist zunächst nicht definiert:

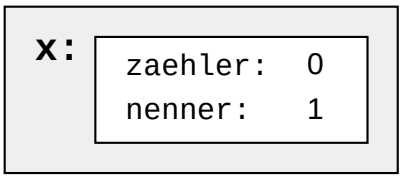


- Da `x` ohne einen Wert zur Initialisierung deklariert wird, wird anschließend der Default-Konstruktor aufgerufen:

```
Bruch::Bruch ()
  : zaehler(0), nenner(1)      // Bruch mit 0 initialisieren
```

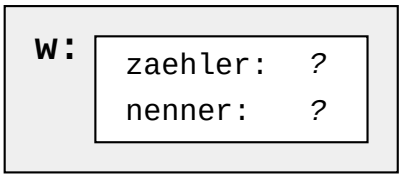
```
{
    // keine weiteren Anweisungen
}
```

Durch dessen Initialisierungsliste wird der Bruch `x` also mit `0` ($\frac{0}{1}$) initialisiert:



Die Initialisierung von `w` geschieht entsprechend in folgenden Schritten:

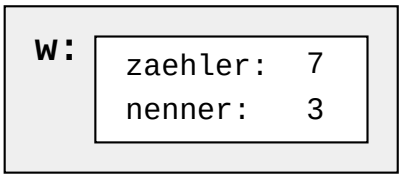
- Zunächst wird wieder der Speicherplatz für das Objekt angelegt, dessen Zustand zunächst nicht definiert ist:



- Da `w` mit zwei Argumenten initialisiert wird, wird der Konstruktor aufgerufen, der für zwei Integer als Parameter definiert ist:

```
/* Konstruktor aus Zähler und Nenner
 */
Bruch::Bruch (int z, int n)
 : zaehler(z), nenner(n)      // Zähler und Nenner wie übergeben initialisieren
{
    // 0 als Nenner ist allerdings nicht erlaubt
    if (n == 0) {
        // Programm mit Fehlermeldung beenden
        std::cerr << "Fehler: Nenner ist 0" << std::endl;
        std::exit(EXIT_FAILURE);
    }
}
```

Mit den übergebenen Parametern initialisiert der Konstruktor den Zähler und den Nenner und legt somit den Bruch $\frac{7}{3}$ an:



Anschließend wird noch geprüft, ob der Nenner 0 ist.

Zerstören von Objekten

Zerstört werden die angelegten Objekte automatisch, wenn vom Programm der Gültigkeitsbereich der Objekte verlassen wird. Genau so, wie beim Anlegen automatisch Speicherplatz für das Objekt angelegt wird, wird er bei der Zerstörung des Objekts auch wieder automatisch freigegeben.

Bei lokalen Objekten geschieht dies, wenn der Block, in dem sie deklariert werden, wieder verlassen wird:

```
void f ()
{
    Bsp::Bruch x;          // Erzeugung und Initialisierung von x
    Bsp::Bruch w(7,3);     // Erzeugung und Initialisierung von w
    ...
}                          // automatische Zerstörung von x und w
```

Es ist allerdings möglich, Funktionen zu definieren, die unmittelbar vor der Freigabe des Speicherplatzes aufgerufen werden. Sie bilden das Gegenstück zu den Konstruktoren und heißen Destruktoren. Sie können z.B. dazu dienen, eine entsprechende Rückmeldung auszugeben oder einen Zähler für die Anzahl der existierenden Objekte zurückzusetzen. Typisch ist auch die Freigabe von explizit angelegtem Speicherplatz, der zu einem Objekt gehört. Da bei der Klasse `Bruch` kein Bedarf für Destruktoren besteht, werden sie erst später in Abschnitt 6.1 eingeführt.

Statische und globale Objekte

Statische oder globale Objekte werden beim Programmstart angelegt und beim Programmende zerstört. Dies hat eine wichtige Konsequenz: Die Funktion `main()` ist nicht unbedingt die erste Funktion eines Programms, die aufgerufen wird. Vorher werden alle Konstruktoren für statische Objekte aufgerufen, die wiederum beliebige Hilfsfunktionen aufrufen können. Ebenso können nach dem Beenden von `main()` oder einem Aufruf von `exit()` noch Destruktoren für statische und globale Objekte aufgerufen werden.

Nachfolgend wird *vor* dem Aufruf von `main()` für das globale Objekt `zweiDrittel` der Konstruktor der Klasse `Bruch` aufgerufen:

```
/* globaler Bruch
 * - Erzeugung und Initialisierung (über Konstruktor) beim Programmstart
 * - automatische Zerstörung (über Destruktor) beim Programmende
 */
Bruch zweiDrittel(2,3);

int main ()
{
    ...
}
```

Da Konstruktoren beliebige Hilfsfunktionen aufrufen können, können auf diese Weise bei komplexen Klassen vor dem Aufruf von `main()` bereits erhebliche Dinge geschehen. Ein Beispiel dafür wäre eine Klasse für Objekte, die geöffnete Dateien repräsentieren. Wird ein globales Ob-

jekt dieser Klasse deklariert, wird durch den Aufruf des Konstruktors noch vor dem Aufruf von `main()` eine entsprechende Datei geöffnet.

Felder von Objekten

Konstruktoren werden für jedes erzeugte Objekt einer Klasse aufgerufen. Wenn ein Feld (Array) von zehn Brüchen deklariert wird, wird also auch zehnmal der Default-Konstruktor aufgerufen:

```
Bsp::Bruch werte[10];       // Feld von zehn Brüchen
```

Dabei können auch Werte zur Initialisierung der Elemente im Feld angegeben werden, wie das folgende Beispiel zeigt:

```
Bsp::Bruch bm[5] = { Bsp::Bruch(7,2), Bsp::Bruch(42),
                     Bsp::Bruch(), 13 };
```

Angelegt werden fünf Brüche (`bm[0]` bis `bm[4]`). `bm[0]` wird mit dem `int`/`int`-Konstruktor initialisiert, da zwei Parameter übergeben werden. `bm[1]` und `bm[3]` werden mit dem `int`-Konstruktor initialisiert, da ein Integer übergeben wird. `bm[2]` und `bm[4]` werden mit dem Default-Konstrukor initialisiert, da kein Argument angegeben wird (bei `bm[2]` wegen der leeren Klammer und bei `bm[4]`, da dafür überhaupt kein Wert zur Initialisierung mehr angegeben wird).

Auch diese fünf Brüche werden zerstört, wenn der Gültigkeitsbereich, in dem sie erzeugt werden, verlassen wird.

Aufruf von Elementfunktionen

Mit der Anweisung

```
w.print()
```

wird für `w` die Elementfunktion `print()` aufgerufen, die den Bruch ausgibt.

Für `w` wird sozusagen auf dessen Klassenkomponente `print()` zugegriffen. Da das in diesem Fall eine Funktion ist, bedeutet das, dass diese dadurch für `w` aufgerufen wird.

In C++ kann auch ein Zeiger (siehe Abschnitt 3.7.1) auf einen Bruch definiert werden. In einem solchen Fall kann der Zugriff auf eine Komponente bzw. der Aufruf einer Elementfunktion über den Operator `->` erfolgen:

```
Bsp::Bruch  x;    // Bruch
Bsp::Bruch* xp;   // Zeiger auf Bruch

xp = &x;          // xp zeigt auf x

xp->print();      // print() für das, worauf xp zeigt, aufrufen
```

Zuweisungen

Mit der Anweisung

```
x = w;
```

wird dem Bruch `x` der Bruch `w` zugewiesen.

Eine solche Zuweisung ist möglich, obwohl sie in der Klassenspezifikation nicht deklariert worden ist. Für jede Klasse ist nämlich automatisch ein *Default-Zuweisungsoperator* vordefiniert. Er ist so implementiert, dass er komponentenweise zuweist. Das bedeutet, dass dem Zähler von `x` der Zähler von `w` und dem Nenner von `x` entsprechend der Nenner von `w` zugewiesen wird.

Der Zuweisungsoperator kann für eine Klasse auch selbst definiert werden. Dies ist vor allem dann notwendig, wenn ein komponentenweises Zuweisen nicht korrekt wäre. Typischerweise ist das dann der Fall, wenn Zeiger als Komponenten verwendet werden. In Abschnitt 6.1.5 wird anhand der Klasse `String` ausführlich darauf eingegangen.

4.1.9 Erzeugung temporärer Objekte

In der folgenden Anweisung wird dem Bruch `w` der in einen Bruch umgewandelte Wert 1000 zugewiesen:

```
// 1000 in einen Bruch umwandeln und w zuweisen
w = Bsp::Bruch(1000);
```

Der Ausdruck

```
Bsp::Bruch(1000)
```

erzeugt einen temporären Bruch, wobei der dazugehörige Konstruktor für einen Parameter aufgerufen wird. Es handelt sich im Prinzip um eine erzwungene Typumwandlung der ganzzahligen Konstante 1000 in den Typ `Bruch`.

Durch die Anweisung

```
w = Bsp::Bruch(1000);
```

wird also ein temporäres Objekt der Klasse `Bruch` angelegt, über den entsprechenden Konstruktor mit $\frac{1000}{1}$ initialisiert, dem Objekt `w` zugewiesen und nach Beendigung der Anweisung wieder zerstört.

Beim Erzeugen temporärer Objekte einer Klasse muss ein entsprechender Konstruktor definiert sein, dem die Argumente zur Umwandlung übergeben werden. In diesem Fall können deshalb temporäre Brüche mit keinem, einem oder zwei Argumenten erzeugt werden:

```
Bsp::Bruch()          // temporären Bruch mit dem Wert 0/1 erzeugen
Bsp::Bruch(42)        // temporären Bruch mit dem Wert 42/1 erzeugen
Bsp::Bruch(16,100)    // temporären Bruch mit dem Wert 16/100 erzeugen
```

4.1.10 UML-Notation

Bei der Modellierung von Klassen hat sich die UML-Notation als Standard etabliert. Sie dient dazu, Aspekte von objektorientierten Programmen grafisch darzustellen. Abbildung 4.4 stellt die UML-Notation der Klasse `Bruch` dar.

Bei der UML-Notation werden Klassen durch Rechtecke dargestellt. In den Rechtecken befinden sich, jeweils durch horizontale Linien getrennt, der Klassenname, die Attribute und die Operationen. Führende Minuszeichen bei Attributen und Operationen bedeutet, dass diese privat

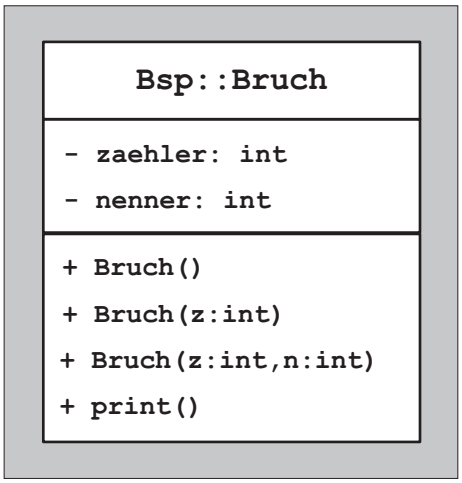


Abbildung 4.4: UML-Notation der Klasse `Bruch`

sind. Führende Pluszeichen stehen für öffentlichen Zugriff. Falls eine Operation einen Rückgabetyp besitzt, wird dieser durch einen Doppelpunkt getrennt, hinter der Operation angegeben.

Je nach Stand der Modellierung können bei der UML-Notation Details entfallen. So können z.B. der Paketname (der Namensbereich), die führenden Zeichen zur Sichtbarkeit, die Datentypen der Attribute oder die Parameter der Operationen entfallen. Auch kann ganz auf die Angabe von Attributen und Operationen verzichtet werden. Die kürzeste Form einer UML-Notation für die Klasse `Bruch` wäre also ein Rechteck, in dem nur das Wort `Bruch` steht.

4.1.11 Zusammenfassung

- *Klassen* werden durch Klassenstrukturen mit dem Schlüsselwort `class` deklariert.
- Klassen sollten wie alle anderen globalen Symbole in einem Namensbereich deklariert werden.
- Die *Komponenten* einer Klasse besitzen *Zugriffsrechte*, die durch die Schlüsselwörter `public` und `private` vergeben werden. Der Default-Zugriff ist `private`.
- Eine Struktur ist in C++ das gleiche wie eine Klasse, nur dass statt `class` das Schlüsselwort `struct` verwendet wird und der Default-Zugriff `public` ist.
- Die Komponenten der Klassen können auch Funktionen (so genannte *Elementfunktionen*) sein. Sie bilden typischerweise die öffentliche Schnittstelle zu den Komponenten, die den Zustand des Objekts definieren (es kann aber auch private Hilfsfunktionen geben). Elementfunktionen haben Zugriff auf alle Komponenten ihrer Klasse.
- Für jede Klasse ist ein *Default-Zuweisungsoperator* vordefiniert, der komponentenweise zuweist.

- *Konstruktoren* sind spezielle Elementfunktionen, die beim Erzeugen von Objekten einer Klasse aufgerufen werden und dazu dienen, diese zu initialisieren. Sie tragen als Funktionsnamen den Namen der Klasse und besitzen keinen Rückgabetyp, auch nicht `void`.
- Alle Funktionen müssen mit *Parameter-Prototypen* deklariert werden.
- Funktionen können *überladen* werden. Das bedeutet, dass der gleiche Funktionsname mehrfach vergeben werden darf, solange sich die Parameter in ihrer Anzahl oder ihren Typen unterscheiden. Eine Unterscheidung nur durch den Rückgabetyp ist nicht erlaubt.
- Der *Bereichsoperator* „`::`“ ordnet einem Symbol den Gültigkeitsbereich einer bestimmten Klasse zu. Er besitzt höchste Priorität.
- Zur Modellierung von Klassen wird standardmäßig die UML-Notation verwendet.