Programmiermethoden in der Mathematik
Klassen in C++

Implementierung einer Klasse am Beispiel der Klasse `Bruch`:

- **Deklaration** erfolgt in einer Header-Datei `bruch.h`:

```
#ifndef BRUCH_H           // bedingte "Ubersetzung:
#define BRUCH_H           // Klasse wird h"ochstens einmal eingebunden
class Bruch
{
   private:               // kein Zugriff von aussen
      int zaehler;
      int nenner;
   public:                // oeffentliche Schnittstellen
      void print();       // Ausgabemethode
}
#endif
```

- Attribute in Klassen sind standardmäßig `private`, d.h. vor dem Zugriff von außen geschützt. Es ist dennoch sinnvoll, dieses Schlüsselwort explizit aufzuführen.

- Folgende Methoden werden vom Compiler automatisch generiert, d.h. sie müssen nicht deklariert und implementiert werden:

  1. Ein Default-Konstruktor ohne Parameter. Man kann also mit

     ```
     Bruch b;
     ```

     einen neuen Bruch erzeugen, d.h. Speicherplatz bereitstellen. Sobald irgendein Konstruktor in der Klassendeklaration enthalten ist, ist der Default-Konstruktor nicht mehr vorhanden.

  2. Der Zuweisungsoperator, wie z.B. in

     ```
     Bruch b1,b2;
     b2=b1;
     ```

  3. Ein *Copy-Konstruktor*, er initialisiert das neu erzeugte Objekt mit den Werten des Parameters:

     ```
     Bruch b1;
     Bruch b2(b1);
     ```

     oder auch

     ```
     Bruch b2=b1;
     ```

  4. Ein Default-Destruktor.
  5. Der Adress- oder Referenzoperator `&`, der die Adresse des Objektes zurückgibt.

- Man kann **eigene Konstruktoren** definieren. Konstruktoren haben keinen Rückgabewert, auch nicht `void`!!! Sie sind also ein Sonderfall der bisher bekannten Funktionen.

```
Bruch();              // Konstruktor ohne Paramater
Bruch(int z);         // Konstruktor mit einem Parameter (Zaehler)
Bruch(int z,int n);   // Konstruktor mit zwei Parametern (Zaehler, Nenner)
```

  Aufruf:

```
Bruch b1(1), b2=1, b3(1,2);
```

- Methode aufrufen:

```
b.print();                // Methode print fuer Objekt b aufrufen
```

  Die Schreibweise `b.print` entspricht der für Elemente einer Struktur. Die Methode `print` bekommt keinen Parameter. Im OO-Sinne *sendet* die Anweisung `b.print()` *die Nachricht* "`print`" an das Objekt `b` der Klasse `Bruch`.

- Innerhalb der Klasse greift man mit `zaehler` etc. auf die Attribute des entsprechenden Objektes zu. Außerhalb der Methoden (z.B. mit `b.nenner=1`) ist dies nicht möglich, da diese `private` und damit geschützt sind.
- Es können Felder von Objekten einer Klasse angelegt werden:

```
Bruch b[10];
```

- Man kann einen Zeiger auf ein Objekt definieren:

```
Bruch b,*p;
p = &b;                    // p zeigt auf b
p->print();                // entspricht (*p).print(), also hier b.print();
```

  Innerhalb der Klassenmethoden existiert der Zeiger `this`, der auf das aktuelle Objekt zeigt.

- **Implementierung** der Klasse bzw. ihrer Methoden: Sie erfolgt in einer separaten Datei, z.B. `bruch.cc`:

```
#include "bruch.h"             // Klassendeklaration einbinden
#include <iostream>
void Bruch::print()
{
   std::cout << zaehler << '/' << nenner << std::endl;
}
```

- Man kann auch Operatoren für die Klasse deklarieren:

```
Bruch operator+(Bruch);
```

  Aufruf dann wie gewohnt mit `b3=b1+b2;`

- Evtl. selbstdefinierte Konstruktoren hinzufügen:

```
#include "bruch.h"
#include <iostream>
Bruch::Bruch()              // Konstruktor ohne Parameter
 : zaehler(0), nenner(1)    // Bruch z.B. mit 0 initialisieren
{                           // keine weiteren Anweisungen
}
Bruch::Bruch(int z)         // Konstruktor mit einem Parameter (Zaehler)
// ...
Bruch::Bruch(int z, int n)  // Konstruktor mit zwei Parametern
// ...
void Bruch::print()
// s.o.
Bruch Bruch::operator+(Bruch)
// ...
```

  Bemerkungen:

  - Der **Bereichsoperator** `::` ordnet dem nachfolgenden Namen als Gültigkeitsbereich die davorstehende Klasse zu. `Bruch::print()` bedeutet also: Die Methode `print` hat in dieser Form nur Gültigkeit, wenn sie auf ein Objekt der Klasse `Bruch` angewandt wird.
  - Die Konstruktoren können noch vor Beginn der eigentlichen Anweisungen (in { }) eine Initialisierung der Attribute (hier `zaehler`, `nenner`) vornehmen: In der sog. **Initialisierungsliste** nach dem Doppelpunkt. Daher ist z.B. beim ersten Konstruktor der eigentliche Anweisungsteil leer.
  - Ein Destruktor (Signatur `~Bruch()`) muss implementiert werden, wenn in einem selbstdefinierten Konstruktor zusätzlicher Speicherplatz bereitgestellt wurde. Ansonsten gilt:
    - *Lokale Objekte* werden automatisch zerstört und ihr Speicherplatz freigegeben, wenn ihr Gültigkeitsbereich (also die Funktion, in der sie instanziert wurden) verlassen wird.
    - *Globale Objekte* werden beim Beenden der Funktion `main` zerstört. Da darüber hinaus Konstruktoren andere Funktionen aufrufen können, kann also bei der Verwendung von globalen Objekten vor (bzw. nach) den Anweisungen in der Funktion `main` schon (bzw. noch) einiges passieren.