

Abstrakte Datentypen

In diesem Teil der Veranstaltung werden die Konzepte Klasse und Objekt beleuchtet, um abstrakte Datentypen verwirklichen zu können

Inhalt

1. Einleitung	3
2. Klasse und Objekt	10
3. Konstruktoren	18
3.1. Standardkonstruktor	19
3.2. Allgemeine Konstruktoren.....	21
3.2.1. Initialisierung mit Listen	24
3.3. Kopierkonstruktor (engl. copy constructor)	26
3.4. Typumwandlungskonstruktor	30
4. Destruktoren	32
5. Konstante Objekte und Methoden	37
6. Beispiel Rationale Zahlen	39

6.1. Header mit Klassendeklaration	41
6.2. Implementierung.....	43
7. Abstrakte Datentypen	48
7.1. Stack.....	48
7.1.1. Implementierung als Array	50
7.2. Schlangen	57

1. Einleitung

Zunächst werden einige Begriffe geklärt. Dazu wird ein Beispiel betrachtet, bei dem ein Bankkonto modelliert werden soll.

Ein **Bankkonto** hat einen **Kontostand** (Saldo) und eine **Überziehungslinie**. Man kann Geld **einzahlen, abheben** und den **Kontostand** abfragen.

Ein Bankkonto kann in **unstrukturierter Weise** realisiert werden, indem zwei globale Variable verwendet werden, auf die direkt an beliebiger Stelle im Programm zugegriffen wird.

```
$ cat konto01.cpp
#include <iostream.h>                                // Konto durch unstrukturierten Zugriff
double kontostand=0;                                // Kontosaldo bei Eröffnung
double linie;                                       // Linie für Überziehungen

int main() {
    char auswahl;
    while (true) {
        double betrag;
        cout << "---- Buchen ----" << endl;        // Menü
        cout << "e Einzahlung" << endl;
        cout << "a Auszahlung" << endl;
        cout << "k Kontostand" << endl;
        cout << "b Beenden" << endl;
        cout << "Auswahl: ";
        cin >> auswahl;
```

```

switch (auswahl) {
    case 'e': cout << "    Einzahlunsbetrag: "; // Einzahlung
              cin >> betrag;
              kontostand += betrag; // Kontostand erhöhen
              break;

    case 'a': cout << "    Auszahlungsbetrag: "; // Auszahlung
              cin >> betrag;
              if (kontostand - betrag >= linie) // Linienprüfung
                  kontostand -= betrag; // Kontostand verringern
              else cout << "    Linie überschritten!" << endl;
              break;

    case 'k': cout << "    Kontostand: " << kontostand << endl;
              break;

    case 'b': exit(0); // Beenden
    default: cout << "ungültige Auswahl" << endl;
}
}
}

```

Diese Lösung ist schwer erweiterbar und schlecht wartbar. Nach den ersten Erweiterungen ist der Code nicht mehr lesbar. Dies führt zu fehlerbehafteten Programmen.

Die grundlegenden Konzepte des Software-Entwurfs sind:

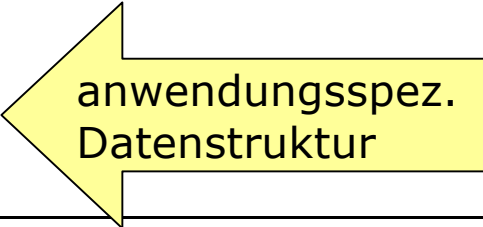
- Strukturierung,
- Modularität und
- Objektorientierung.

Strukturiertes Programmieren ist eine methodische Vorgehensweise, um Programm systematisch zu konstruieren: Die Verwendung einiger weniger Kontrollanweisungen für die Steuerung des Programmflusses und anwendungsbezogenen Datenstrukturen macht Programme übersichtlich und die Programmierung und Wartung produktiver.

(„**Programm = Algorithmus + Datenstruktur**“)

Mit dieser Vorgehensweise lässt sich unser Beispiel des Bankkontos nun formulieren:

```
$ cat konto02.cpp
#include <iostream.h> // Konto durch strukturierten Zugriff
struct Konto {
    double kontostand; // Kontosaldo
    double linie; // Linie für Überziehungen
} konto1;
```

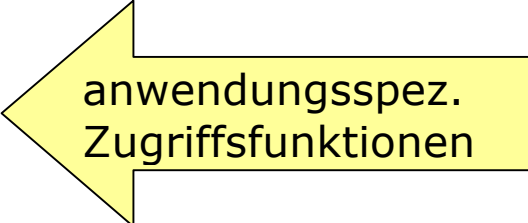


anwendungsspez.
Datenstruktur

```

void kontostandAnzeigen(Konto k) {
    cout << "    Kontostand: " << k.kontostand << endl;
}
void kontoInitialisieren(Konto &k) {
    k.kontostand = 0;
    k.linie = -100.0;
}
void einzahlen(Konto &k) {
    double betrag;
    cout << "    Einzahlunsbetrag: ";    // Einzahlung
    cin >> betrag;
    k.kontostand+= betrag;                // Kontostand erhöhen
}
void auszahlen(Konto &k) {
    double betrag;
    cout << "    Ausahlunsbetrag: ";    // Einzahlung
    cin >> betrag;
    if (k.kontostand - betrag >= k.linie) // Linienprüfung
        k.kontostand -= betrag;        // Kontostand verringern
    else cout << "    Linie überschritten!" << endl;
}

```



anwendungsspez.
Zugriffsfunktionen

```

int main() {
    char auswahl;
    kontoInitialisieren(kontol);           // Initialisierung
    while (true) {
        cout << "---- Buchen ----" << endl; // Menü
        cout << "e Einzahlung" << endl;
        cout << "a Auszahlung" << endl;
        cout << "k Kontostand" << endl;
        cout << "b Beenden" << endl;
        cout << "Auswahl: ";
        cin >> auswahl;
        switch (auswahl) {                // Auswahl
            case 'e':                      einzahlen(kontol);
            break;
            case 'a':                      auszahlen(kontol);
            break;
            case 'k':                      kontostandAnzeigen(kontol);
            break;
            case 'b':                      exit(0); // Beenden
            default:                       cout << "ungültige Auswahl" << endl;
        }
    }
}

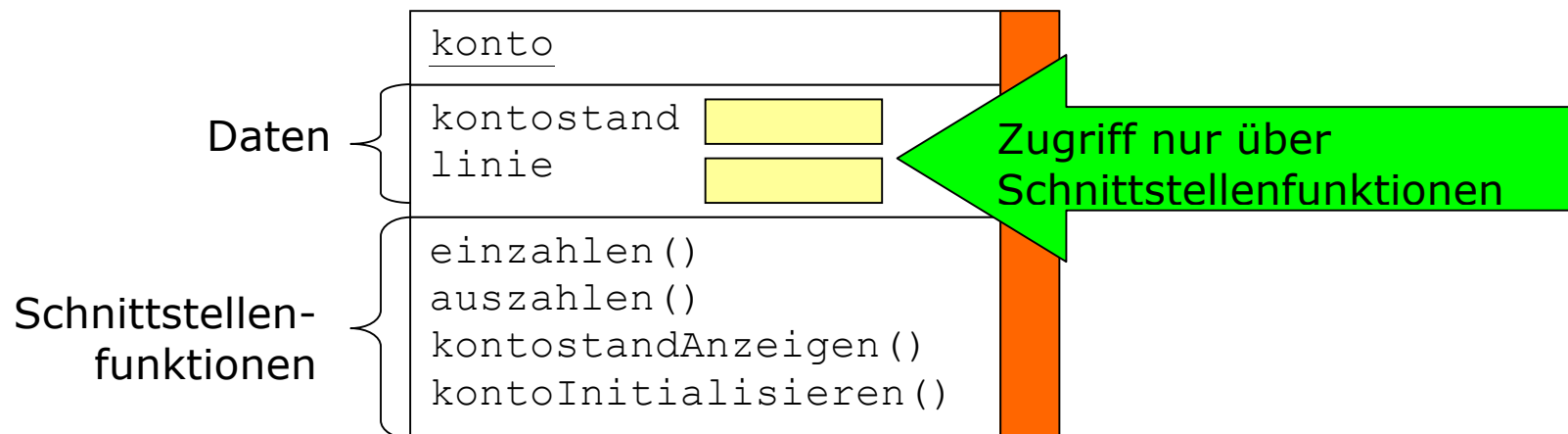
```

Trotz strukturierter Programmierung könnte in einer Erweiterung des Programms **direkt** auf die globale Variable `konto1` zugegriffen werden, **ohne** eine Funktion zu schreiben.

Die strukturierte Programmierung löst die Problem die beim „**Programmieren im Großen**“ auftreten **nicht**.

Eine Aufgabe ist in Teile (**Module**) zu zerlegen, die **nur** über definierte **Schnittstellen** zugreifbar sind. Der Zugriff auf eine Datenstruktur eines Moduls darf nur ausschließlich durch speziell für die Datenstruktur entwickelte Funktionen erfolgen.

Von außen her betrachtet ist ein **Modul** eine **Abstraktion**, dessen innere **Komponenten** verborgen und damit **geschützt** sind.



Durch den **ausschließlichen** Zugriff über die Schnittstellenfunktionen sind Änderungen im Inneren für den Benutzer des Moduls unwichtig. Dieses Prinzip, die Implementierung zu verbergen wird **Geheimhaltungsprinzip** genannt.

Das Prinzip der Zusammenfassung von Daten und sie verändernden Funktionen nennt man auch **Datenkapselung** (engl. encapsulation). Ein so gekapselter Datentyp (wie `konto`) wird **abstrakter Datentyp** (kurz ADT) genannt.

Somit werden Sprachkonstrukte gebraucht, mit denen man solche abstrakte Datentypen realisieren kann.

Wir werden in den Folgenden **C++ Sprachmittel** dafür besprechen. Weitere objektorientierte Konzepte, wie beispielsweise Vererbung und Polymorphismus werden im nächsten Semester behandelt.

2. Klasse und Objekt

Eine **Klasse** ist ein **abstrakter Datentyp**, der in einer Programmiersprache formuliert ist. Es ist somit die **Beschreibung** des **Verhaltens** und der **Eigenschaften** von ähnlichen **Objekten**.

In C++ dient eine Klasse dazu, dem Compiler die Beschreibung von Objekten mitzuteilen, die später durch eine Objektdefinition erzeugt werden.

Ein **Objekt** ist die **konkrete Ausprägung** einer **Klasse**; es belegt also Speicherplatz; die Klasse verbraucht keinen Speicherplatz.

Die **Eigenschaften** (**Attribute**) eines Objektes werden durch C++ Datentypen (elementare und zusammengesetzte) dargestellt. Der Zustand eines Objektes ist der Wert seiner Attribute.

Das **Verhalten** wird durch klassenspezifische Funktionen (**Methoden**) beschrieben. Eine Methode, angewendet auf ein Objekt, verändert den Objektzustand. Die **öffentlichen Methoden** der Klasse bilden die **Schnittstellen** des ADT, **private Methoden** sind klassenspezifische, von außen nicht zugreifbare „**Hilfsroutinen**“.

Objekte können durch die Position im Speicher **eindeutig identifiziert** werden; d.h. zwei Objekte, auch wenn sie zur gleichen Klasse gehören, haben immer unterschiedliche Adressen.

Beispiel (Klasse Konto)

```
$ cat konto03.cpp
#include <iostream.h> // Konto als Klasse
class CKonto {
    public:
        void kontoInitialisieren() {
            kontostand = 0;
            linie = -100.0;
        }
        void kontostandAnzeigen() {
            cout << "    Kontostand: " << kontostand << endl;
        }
        void einzahlen() {
            ...
        }
        void auszahlen() {
            ...
        }
    private:
        double kontostand;
        double linie;
};
```

öffentlich

privat

Methoden

Attribute

// Kontosaldo bei Eröffnung
// Linie für Überziehungen

Daten und **Methoden** sind durch `class {}` **zusammengefasst**. Der **Gültigkeitsbereich** von Klassenelementen ist **lokal** zu der Klasse. Das Schlüsselwort `private` könnte entfallen, wenn die Attribute vor `public`-Bereiche stehen würden (Default ist `private`).

Alle Elemente nach dem Schlüsselwort `public` sind öffentlich zugänglich.

Die Methoden nennt man auch „Elementfunktionen“.

Um ein Objekt einer Klasse verwenden zu können, muss es zuerst definiert werden. Dann kann man die öffentlichen Methode „aufrufen“, um den Zustand des Objekts zu verändern.

```

int main() {
    CKonto konto1;
    char auswahl;
    konto1.kontoInitialisieren(); // Initialisierung
    while (true) {
        cout << "---- Buchen ----" << endl; // Menü
        cout << "e Einzahlung" << endl;
        cout << "a Auszahlung" << endl;
        cout << "k Kontostand" << endl;
        cout << "b Beenden" << endl;
        cout << "Auswahl: ";
        cin >> auswahl;
        switch (auswahl) { // Auswahl
            case 'e':
                konto1.einzahlen();
                break;
            case 'a':
                konto1.auszahlen();
                break;
            ...
            default:
                cout << "ungültige Auswahl" << endl;
        }
    }
}

```

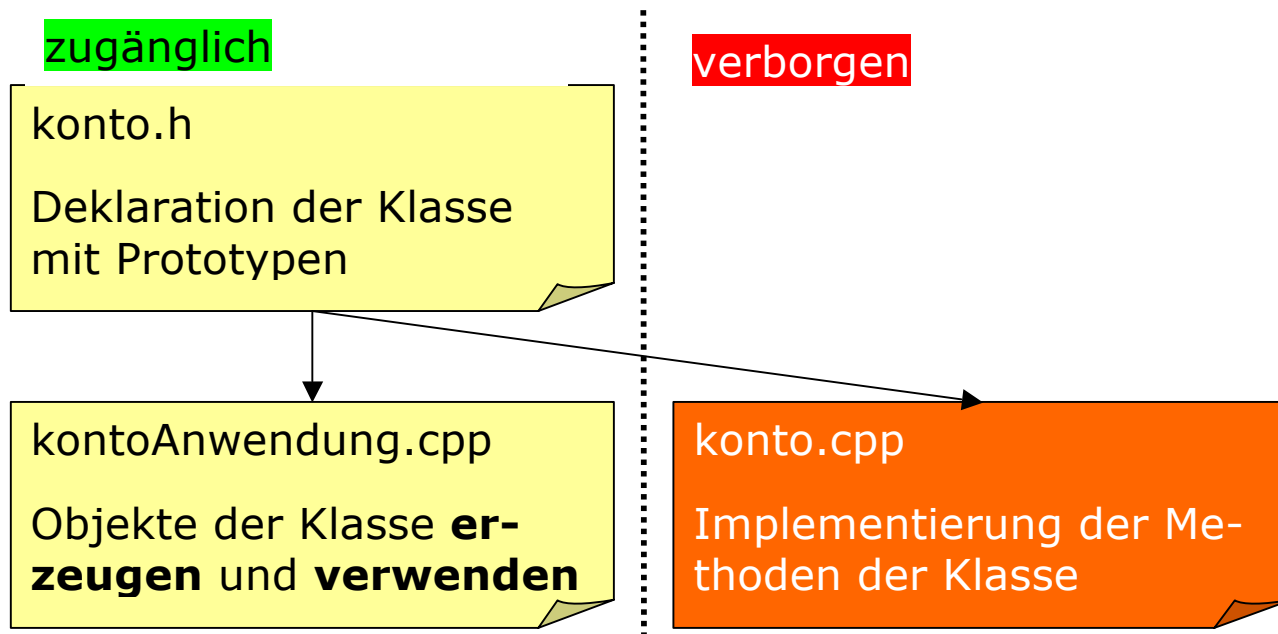
Erzeugung eines Objektes

Methodenaufrufe

So wie bei einer Variablendefinition wird durch „CKonto konto1“ hier Speicher bereitgestellt; eben Speicher, der Daten und Programmcode für das Objekt `konto1` vom Typ `CKonto` enthält.

Der **Zugriff** auf Elemente eines Objekts erfolgt in der Form „Objektname.Elementname“.

Bis jetzt ist also die Idee der ADT mit dem Konzept Klasse in C++ abbildbar. Die Geheimhaltung ist aber noch offen, da die Realisierung der Methoden direkt in den Code der Klasse integriert ist. Das **Verbergen** der **Implementierung** kann erreicht werden, in dem man in der Klasse nur die Prototypen verwendet und die Implementierung in eine eigene Datei verlagert.



```

$ cat CKonto/konto.h
#include <iostream.h>
class CKonto {
    public:
        void kontoInitialisieren();           // Initialisierung{
        void kontostandAnzeigen();           // Ausgabe des Kontostandes
        void einzahlen();                    // Einzahlen
        void auszahlen();                    // Auszahlen
    private:
        double kontostand;                   // Kontosaldo bei Eroeffnung
        double linie;                        // Linie für Überziehungen
};

```

Prototypen

Nun wird in der Implementierung der **Bereichsoperator** „`::`“ verwendet, um zu definieren, zu welcher Klasse die Implementierung der Methode gehört.

```

$ cat konto.cpp
#include "konto.h"
void CKonto::kontoInitialisieren() {
    kontostand = 0;
    linie = -100.0;
}
void CKonto::kontostandAnzeigen() {
    cout << "    Kontostand: " << kontostand << endl;
}
void CKonto::einzahlen() {
    double betrag;
    cout << "    Einzahlunsbetrag: ";           // Einzahlung
    cin >> betrag;
    kontostand+= betrag;                       // Kontostand erhöhen
}
void CKonto::auszahlen() {
    double betrag;
    cout << "    Ausahlunsbetrag: ";           // Einzahlung
    cin >> betrag;
    if (kontostand - betrag >= linie) // Linienprüfung
        kontostand -= betrag;           // Kontostand verringern
    else cout << "    Linie überschritten!" << endl;
}

```

Innerhalb von `main` ist jetzt nur die Headerdatei erforderlich:

```

$ cat kontoMain.cpp
#include "konto.h"
int main() {
    CKonto konto1;
    char auswahl;
    konto1.kontoInitialisieren(); // Initialisierung
    while (true) {
        cout << "---- Buchen ----" << endl; // Menü
        cout << "e Einzahlung" << endl;
        cout << "a Auszahlung" << endl;
        cout << "k Kontostand" << endl;
        cout << "b Beenden" << endl;
        cout << "Auswahl: ";
        cin >> auswahl;
        switch (auswahl) { // Auswahl
            case 'e':
                konto1.einzahlen();
                break;
            case 'a':
                konto1.auszahlen();
                break;
            case 'k':
                konto1.kontostandAnzeigen();
                break;
            case 'b':
                return 0; // Beenden
            default:
                cout << "ungültige Auswahl" << endl;
        }
    }
}

```

Wie würde eine Bank realisiert (welche Datenstruktur), bei der es eine Menge von Konten gibt?

3. Konstruktoren

Objekte sollten definierte **Initialwerte** für ihre **Attribute** haben, wenn sie erzeugt worden sind. Im letzten Beispiel `CKonto` war dazu eine Methode explizit implementiert worden (`kontoInitialisieren()`).

Wenn ein Objekt erzeugt wird, wird Speicherplatz für das Objekt angelegt und automatisch ein **Konstruktor** aufgerufen, der die Versorgung mit Anfangswerten übernimmt.

Ein Konstruktor ist so ähnlich aufgebaut wie eine Funktion, der Name ist immer identisch mit dem Namen der zugehörigen Klasse. Konstruktoren haben keine Return-Typen, auch nicht `void`.

3.1. Standardkonstruktor

Wenn im Programm kein Konstruktor explizit angegeben ist, wird ein Default-Konstruktor verwendet. Dieser Konstruktor **ohne formale Parameter**, heißt **Standardkonstruktor**.

Also ändern wir unser Beispiel und verwenden einen Standardkonstruktor anstelle der expliziten Methode `kontoInitialisieren()`.

```

$ cat CKonto/konto.h
#include <iostream.h>
class CKonto {
    public:
        CKonto();           // Initialisierung per Konstruktor
        void kontostandAnzeigen(); // Ausgabe des Kontostandes
        ...
    private:
        double kontostand; // Kontosaldo bei Eroeffnung
        ...
};

```

```

$ cat konto.cpp
#include "konto.h"
CKonto::CKonto() {
    kontostand = 0;
    linie = -100.0;
}
void CKonto::kontostandAnzeigen() {
...

```

3.2. Allgemeine Konstruktoren

Im Gegensatz zu Standardkonstruktoren, haben Allgemeine Konstruktoren **Argumente**. Weiterhin können Allgemeine Konstruktoren **überladen** werden, wobei es aber stets eine eindeutige Signatur geben muss.

Somit können wir unser Beispiel ändern, und die Initialwerte für Konten beim Erzeugen eines Objektes bestimmen:

```
$ cat CKonto02/konto.h
#include <iostream.h>
class CKonto {
    public:
        CKonto(); // Initialisierung per Konstruktor
        CKonto(double kontostand, double linie);
        void kontostandAnzeigen(); // Ausgabe des Kontostandes
        ...
    private:
        double kontostand; // Kontosaldo bei Eröffnung
        ...
};
```

```
$ cat konto.cpp
#include "konto.h"
CKonto::CKonto() {
    kontostand = 0;
    linie = -100.0;
}
CKonto::CKonto(double pKontostand, double pLinie) {
    kontostand = pKontostand;
    if (pLinie > 0) linie = -pLinie; // linie muss negativ oder null sein
    else linie = pLinie;
}
void CKonto::kontostandAnzeigen() {
...

```

```
int main() {
    CKonto konto1(100, 1000);
    CKonto konto2();
...

```

Wenn es mehrere Konstruktoren für eine Klasse gibt, wird der Compiler anhand der Anzahl und der Typen der Parameter (Signatur) des Aufrufs entscheiden, welcher Konstruktor verwendet wird.

Achtung:

Wie bei normalen Funktionen auch, kann man bei Konstruktoren die Anzahl der Parameter variabel halten. Dabei muss aber stets die Eindeutigkeit der Signatur über alle Konstruktoren einer Klasse hinweg gegeben sein.

```
CKonto(double kontostand, double linie=-1000.0);
```

```
...
```

```
CKonto konto3(0.0);
```

3.2.1. Initialisierung mit Listen

Beim Aufruf eines Konstruktors wird

1. zunächst Speicherplatz für die Attribute der Klasse reserviert,
2. dann erfolgt die Parameterübergabe (aktuelle Parameter -> formale Parameter).
3. Zuletzt wird dann der Block (Konstruktorrumpf) ausgeführt.

Da normalerweise durch die Parameter des Konstruktor klassenlokale Attribute initialisiert werden, kann Schritt 1 und 2 zusammen ausgeführt werden, wodurch bei vielen und/oder großen Objekten Laufzeitvorteile bei der Erzeugung der Objekte resultieren.

Dazu dienen **Initialisierungslisten**:

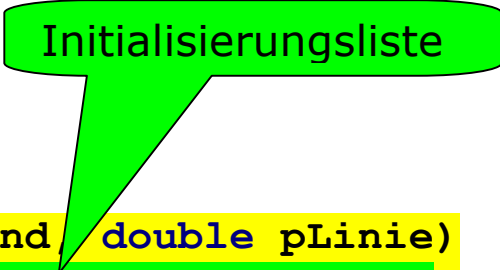
Vor dem Konstruktorrumpf wird eine Liste der Form

`: AttributName_i(Parameter_j), ... , AttributName_l(Parameter_m)`
angegeben.

```

$ cat CKonto03/konto.cpp
#include "konto.h"
CKonto::CKonto() {
    kontostand = 0;
    linie = -100.0;
}
CKonto::CKonto(double pKontostand, double pLinie)
    :kontostand(pKontostand), linie(pLinie) {
    if (linie > 0) linie = -linie; // linie muss negativ oder null sein
}
void CKonto::kontostandAnzeigen() {
...

```



Achtung:

Die Initialisierung innerhalb der Initialisierungsliste **erfolgt** in der **Reihenfolge** der **Attributdeklarationen** in der Klasse, **nicht** in der Aufschreibungsreihenfolge der Initialisierungsliste.

Ein weiteres Anwendungsfeld von Initialisierungslisten ist das Vorbesetzen von Konstanten einer Klasse.

3.3. Kopierkonstruktor (engl. copy constructor)

Der Kopierkonstruktor wird verwendet, um ein Objekt erzeugen zu können und es mit den Attributwerten eines anderen Objektes **derselben** Klasse initialisieren zu können.

Die allgemeine Form der **Deklaration** eines Kopierkonstruktors ist:

```
KlassenName( KlassenName& name ) {}
```

Im Normalfall sollte das Objekt, das man als Kopiervorlage nimmt, nicht verändert werden. Deshalb ist die gebräuchteste Form:

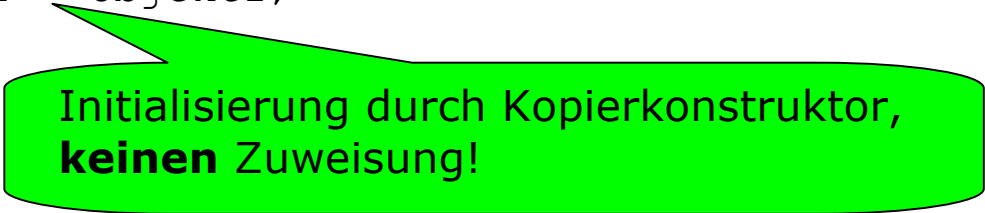
```
KlassenName( const KlassenName& name ) {}
```

Wird in einer Klasse kein eigener Kopierkonstruktor deklariert, wird ein **Standardkopierkonstruktor** verwendet. Grunddatentypen (es sind einfache Objekte) werden kopiert, indem das zugehörige Speicherabbild (Bitmuster) kopiert wird.

Ein **Objekt** wird mittels eines Kopierkonstruktors **erzeugt** durch:

```
Klassenname objekt1;
```

```
Klassenname objekt2 = objekt1;
```



Initialisierung durch Kopierkonstruktor,
keinen Zuweisung!

```
$ cat CKonto04/konto.h
#include <iostream.h>
class CKonto {
    public:
        CKonto(); // Initialisierung
        CKonto(double kontostand, double linie);
        CKonto::CKonto(const CKonto& pKonto); // Kopierkonstruktor
        void kontostandAnzeigen(); // Ausgabe des Kontostandes
        ...
    private:
        double kontostand; // Kontosaldo bei Eroeffnung
        ...
};
```

```

$ cat konto.cpp
#include "konto.h"
CKonto::CKonto() {
    kontostand = 0;
    linie = -100.0;
}
CKonto::CKonto(double pKontostand, double pLinie) {
    kontostand = pKontostand;
    if (pLinie > 0) linie = -pLinie; // linie muss negativ oder null sein
    else linie = pLinie;
}
CKonto::CKonto(const CKonto& pKonto) // Kopierkonstruktor
    :kontostand(pKonto.kontostand), linie(pKonto.linie) {
    cout << "Kopierkonstruktor wurde aufgerufen!" << endl;
}
void CKonto::kontostandAnzeigen() {
...

```

Der Kopierkonstruktor wird **nur** verwendet,

1. wenn ein **neues Objekt** erzeugt wird oder
2. wenn ein Objekt durch **Wertübergabe** an eine Funktion übergeben wird oder
3. wenn eine Funktion ein Objekt als **Ergebnis** zurück liefert.

Bei der Zuweisung eines Objektes oder der Initialisierung mit einem Allgemeinen Konstruktor wird der Kopierkonstruktor **nicht** verwendet!

```
$ cat CKonto04/kontoMain.cpp
main() {
    CKonto konto1(0, 100);
    CKonto konto2 = konto1;
    CKonto konto3(100, 100);
    CKonto konto4;

    konto2 = konto3;
    konto4 = Ckonto(10, 20);
}
```

Kopierkonstruktor wird verwendet

Zuweisung: Kopierkonstruktor wird nicht verwendet

Allgemeiner Konstruktor: Kopierkonstruktor wird nicht verwendet

Welche Ausgabe erzeugt das o.a. Programm?

3.4. Typumwandlungskonstruktor

Die Umwandlung eines Datentyps in eine Klasse kann durch den Typumwandlungskonstruktor erfolgen. Das erste Argument des Typumwandlungskonstruktor ist dabei ein Objekt, das nicht zur selben Klasse wie der Konstruktor gehört, die restlichen Argumente müssen Initialwerte haben.

Beispiel (Konto soll durch einen String, bestehend aus Saldo und Linie initialisiert werden):

```
$ cat CKonto05/kontoMain.cpp
#include "konto.h"
int main() {
    CKonto konto1(" 100 0 ");
    konto1.kontostandAnzeigen();
}
```

„ 100 0 „ soll in zwei int Werte transformiert werden.

```
$ kontoMain.exe
    Kontostand: 100
$
```

```

$ cat konto.cpp
...
CKonto::CKonto(string pSaldoUndLimit) { // Typumwandlungskonstruktor
    unsigned int pos=0;                // Position einer Ziffer
    for (int j=0; j<2; j++) {          // für Saldo und Limit ermittle jeweils Betrag
        while ( pos<pSaldoUndLimit.size() &&
                !isdigit(pSaldoUndLimit.at(pos))
                )                        // erste Ziffer suchen
            pos++;
        int betrag=0;
        while ( pos<pSaldoUndLimit.size() &&
                isdigit(pSaldoUndLimit.at(pos))
                ) {                      // Zahl bilden
            betrag = 10*betrag + pSaldoUndLimit.at(pos) - '0';
            pos++;
        }
        if (j==0)
            kontostand = betrag;
        else
            linie = -betrag;
    }
}
...

```

4. Destruktoren

Konstruktoren sind für die Aktionen bei der „**Geburt**“ eines Objektes verantwortlich.

Destruktoren dienen zum **Aufräumen**, bevor ein Objekt „**stirbt**“. Wenn ein Destruktor nicht explizit programmiert ist, wird der Default Destruktor verwendet.

Die wichtigste Aufgabe eines Destruktors ist es, den nicht mehr benötigten Speicherplatz wieder frei zu geben.

Destruktoren haben weder Argumente noch einen Rückgabewert. Sie werden durch den Klassennamen mit Vorangestellter Tilde (,~') gekennzeichnet.

Die Gültigkeit eines Objektes beginnt mit der öffnenden Klammer und endet mit der korrespondierenden schließenden Klammer; d.h. der **Konstruktor** wird beim **Eintreten** in einen Block aufgerufen, der **Destruktor** bei **Verlassen** des **Blocks**.

Die Reihenfolge der Aufrufe von Konstruktor und Destruktor ist umgekehrt.

Beispiel:

```

$ cat destruktork.cpp
#include<iostream>
using namespace std;
class CBeispiel {
    int zahl;
public:
    CBeispiel(int i=0);           // Konstruktor
    ~CBeispiel();               // Destruktor
};

CBeispiel::CBeispiel(int i) {   // Konstruktor
    zahl=i;
    cout << "Objekt " << zahl << " wird erzeugt.\n";
}

CBeispiel::~~CBeispiel() {     // Destruktor
    cout << "Objekt " << zahl << " wird zerstört.\n";
}
int main() {
    cout << "main wird begonnen\n" ;
    CBeispiel einBeispiel(1);
    cout << "main wird verlassen\n" ;
}
$

```

```

$ destrukt
main wird begonnen
Objekt 1 wird erzeugt.
main wird verlassen
Objekt 1 wird zerstört.
$

```

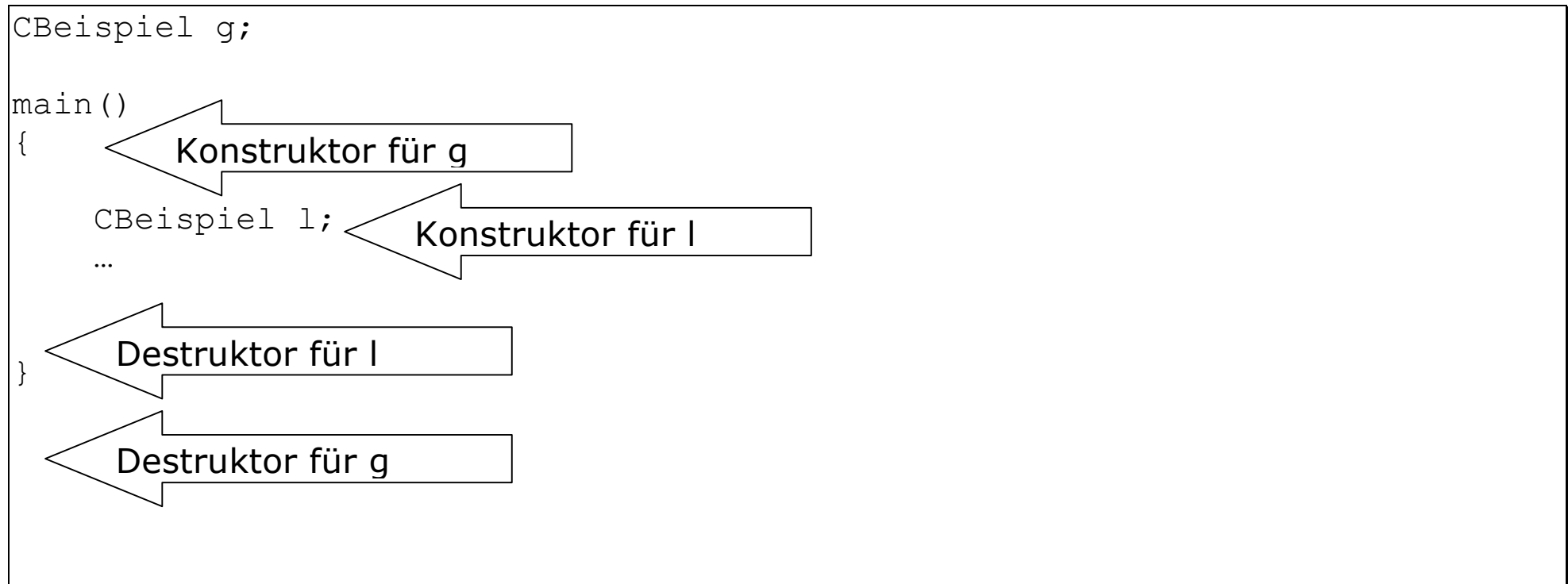
Dabei gilt dies nicht nur auf Ebene von Funktionen, sondern auch für beliebige Blöcke:

```
$ cat destruk02.cpp
class Beispiel {
    int zahl;
public:
    Beispiel(int i=0);           // Konstruktor
    ~Beispiel();                // Destruktor
};
Beispiel::Beispiel(int i) {    // Konstruktor
    zahl=i;
    cout << "Objekt " << zahl << " wird erzeugt.\n";
}
Beispiel::~~Beispiel() {      // Destruktor
    cout << "Objekt " << zahl << " wird zerstört.\n";
}
int main() {
    cout << "main wird begonnen\n" ;
    Beispiel einBeispiel(1);
    { cout << "    neuer Block\n" ;
      Beispiel einBeispiel(2);
      cout << "    Block wird verlassen\n" ;
    }
    cout << "main wird verlassen\n" ;
}
$
```

```
$ destruk02
main wird begonnen
Objekt 1 wird erzeugt.
    neuer Block
Objekt 2 wird erzeugt.
    Block wird verlassen
    Objekt 2 wird zerstört.
main wird verlassen
Objekt 1 wird zerstört.
$
```

Zu beachten ist, dass dadurch Aktionen **vor** Ausführung der Funktion `main` und **nach** ihrer Ausführung stattfinden:

- Wenn in einem Programm globale Objekte existieren, wird ihr Konstruktor vor der ersten Anweisung von `main` aufgerufen.
- Innerhalb des äußeren Blocks von `main` definierte Objekte werden erst nach Verlassen von `main` freigegeben.
- Wegen der umgekehrten Reihenfolge werden globale Objekte zuletzt freigegeben.



```

$ cat destrukt03.cpp
class Beispiel {
    int zahl;
public:
    Beispiel(int i=0);           // Konstruktor
    ~Beispiel();                // Destruktor
};
Beispiel::Beispiel(int i) {     // Konstruktor
    zahl=i;
    cout << "Objekt " << zahl << " wird erzeugt.\n";
}
Beispiel::~~Beispiel() {       // Destruktor
    cout << "Objekt " << zahl << " wird zerstört.\n";
}
Beispiel ein_globales_Beispiel; // globale Variable, durch Vorgabewert mit 0 initialisiert
int main() {
    cout << "main wird begonnen\n" ;
    Beispiel einBeispiel(1);
    {
        cout << "    neuer Block\n    ";
        Beispiel einBeispiel(2);
        cout << "    Block wird verlassen\n    ";
    }
    cout << "main wird verlassen\n" ;
}

```

```

$ destrukt03
Objekt 0 wird erzeugt.
main wird begonnen
Objekt 1 wird erzeugt.
    neuer Block
    Objekt 2 wird erzeugt.
    Block wird verlassen
    Objekt 2 wird zerstört.
main wird verlassen
Objekt 1 wird zerstört.
Objekt 0 wird zerstört.
$

```

5. Konstante Objekte und Methoden

Variablen können durch den Zusatz `const` als konstant deklariert werden. Dies ist bei Objekten genauso möglich.

Dabei sind **Änderungen des Zustandes** des Objektes über normale Methodenaufrufe nicht mehr möglich. Nur die **Konstruktoren** und **Destruktoren** und **konstante Elementfunktionen** sind erlaubt.

Ein `const` Qualifizierer wird beim Überladen von Methoden ausgewertet, d.h. die Signatur mit und ohne `const` ist verschieden.

Beispiel (Klasse für rationale Zahlen, vgl. 6):

```
void rationale::ausgabe();           // Methode 1
void rationale::ausgabe() const;     // Methode 2
```

Je nach Deklaration eines Objektes wird die entsprechende Methode aufgerufen:

```
rationale r;                          // „normale“ rationale Zahl
const rationale cr(1,2);               // konstante rationale Zahl 1/2
r.ausgabe();                           // Methode 1 wird ausgeführt
cr.ausgabe();                           // Methode 2 wird ausgeführt
```

Im nächsten Semester werden wir sehen, wie durch „mutable“ Attribute und „explizite Typumwandlung („casting the const away“) trotzdem Änderungen an konstanten Objekten möglich sind.

6. Beispiel Rationale Zahlen

Die bisherigen Sprachkonstrukte sollen nun an einem etwas größeren Beispiel im Zusammenhang dargestellt werden. Als Aufgabe nehmen wir ein Beispiel aus dem Mathematikunterricht im Gymnasium: Rechnen mit rationalen Zahlen:

Seien $a = \frac{a.z}{a.n}$, $b = \frac{b.z}{b.n}$ und $r = \frac{r.z}{r.n}$ rationale Zahlen,

dann sind die Grundrechenarten wie folgt definiert.

$$\blacksquare r = a + b = \frac{a.z * b.n + b.z * a.n}{a.n * b.n}$$

$$\blacksquare r = a - b = \frac{a.z * b.n - b.z * a.n}{a.n * b.n}$$

$$\blacksquare r = a * b = \frac{a.z * b.z}{a.n * b.n}$$

$$\blacksquare r = a / b = \frac{a.z * b.n}{a.n * b.z}$$

$$\blacksquare 1/a = \frac{a.n}{a.z}$$

Zu realisieren ist eine **Klasse** `rational`, die für eine rationale Zahl den **Zähler** und den **Nenner** als **private Attribute** verwaltet und die o.a. **Grundrechenarten** als **Schnittstelle** zur Verfügung stellt. Jede Rechenoperation soll stets ein gekürztes Ergebnis liefern. Dazu ist eine **private Methode** `kuerzen` zu entwickeln.

Damit man mit rationalen Zahlen in einem Programm umgehen kann sind folgende Methoden zu realisieren:

Methode	Beschreibung
<code>r.eingabe()</code>	Dialogeingabe einer rationalen Zahl
<code>r.ausgabe()</code>	Ausgabe einer rationalen Zahl
<code>r.definiere(z,n)</code>	Setzen von Zähler und Nenner
<code>r.kehrwert()</code>	Kehrwert von r
<code>r.add(a)</code>	<code>r += a</code>
<code>r.sub(a)</code>	<code>r -= a</code>
<code>r.mult(a)</code>	<code>r *= a</code>
<code>r.div(a)</code>	<code>r /= a</code>

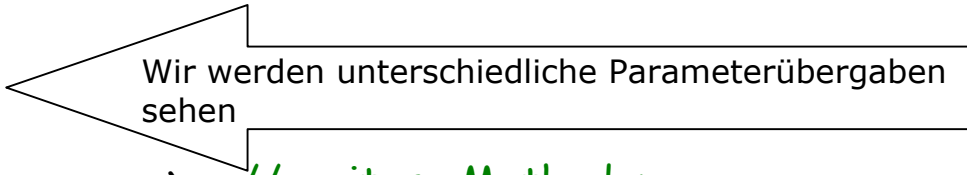
6.1.Header mit Klassendeklaration

```
$ cat ratio.h
#ifndef ratio_h
#define ratio_h ratio_h
class rational {
    public:
        rational();
        rational(long);
        long Zaehler() const;
        long Nenner() const;
        void add(rational r);
        void sub(const rational& r);
        void mult(const rational& r);
        void div(rational r);
        void definiere(long zaehler, long nenner);
        void eingabe();
        void ausgabe() const;
        void kehrwert();
    private:
        long zaehler, nenner;
        void kuerzen();
};
#endif
$
```

// Konstruktor
// Typumwandlungskonstruktor
// Abfragen

// arithmetische Methoden

// weitere Methoden



Wir werden unterschiedliche Parameterübergaben sehen

Mit dieser Klasse ist dann eine Anwendung programmierbar, in der eine Eingabe von zwei rationalen Zahlen mit Ausgabe des Ergebnisses erfolgt:

```
$ cat ratiomain.cpp
// alle 4 Operationen für a und b
void druckeTestfall(rational a, rational b) {
    rational erg;
    cout << "a = "; a.ausgabe();
    cout << "b = "; b.ausgabe();
    erg = a; erg.add(b);
    cout << "+  : "; erg.ausgabe();
    erg = a; erg.sub(b);
    cout << "-  : "; erg.ausgabe();
    erg = a; erg.mult(b);
    cout << "*  : "; erg.ausgabe();
    erg = a; erg.div(b);
    cout << "/  : "; erg.ausgabe();
    cout << endl;
}
int main() {
    rational a,b;
    cout << "Test der Eingabe\n";
    a.eingabe();
    b.eingabe();

    druckeTestfall(a,b);
}
$
```

```
$ ratiomain
Test der Eingabe
Zähler :2
Nenner :4
Zähler :2
Nenner :1
a = 2/4
b = 2/1
+   : 5/2
-   : -3/2
*   : 1/1
/   : 1/4
$
```

6.2. Implementierung

```
$ cat ratio.cpp
// Definition der Methoden
#include"ratio.h"
#include<iostream>
using namespace std;

void rational::add(rational r) { // Wertübergabe
    zaehler = zaehler*r.nenner + r.zaehler*nenner;
    nenner = nenner*r.nenner;
    kuerzen();
}

void rational::sub(const rational& s) { // Referenzübergabe, aber mit const
    rational r = s; // deshalb muss Hilfsvariable r verwendet werden
    r.zaehler *=-1;
    add(r);
}

void rational::mult(const rational& r) {
    zaehler = zaehler*r.zaehler;
    nenner = nenner *r.nenner;
    kuerzen();
}
```

```
void rational::div(rational n) { // Wertübergabe, also kann n verändert werden
    n.kehrwert();
    mult(n);
}

void rational::definiere(long z, long n) {
    zaehler = z;
    nenner = n;
    kuerzen();
}

void rational::eingabe() {
    cout << "Zähler :";
    cin >> zaehler;
    cout << "Nenner :";
    cin >> nenner;
}

void rational::ausgabe() const {
    cout << zaehler << "/" << nenner << endl;
}
```

```

void rational::kehrwert() {
    long temp = zaehler;
    zaehler = nenner;
    nenner = temp;
}

long ggt(long x, long y) {
    while (y > 0) {
        long rest = x % y;
        x=y; y=rest;
    }
    return x;
}

void rational::kuerzen() {
    int sign = 1;
    if (zaehler < 0) { sign=-sign; zaehler = -zaehler;}
    if (nenner < 0) { sign=-sign; nenner = -nenner;}
    long teiler = ggt(zaehler, nenner);
    zaehler = sign*zaehler/teiler;
    nenner = nenner/teiler;
}

```

```

rational::rational()           // Konstruktor
    : zaehler(1), nenner(1)    {
}

rational::rational(long ganzeZahl) // Typumwandlungskonstruktor
    : zaehler(ganzeZahl), nenner(1) {
}

long rational::Zaehler() const {
    return zaehler;
}

long rational::Nenner() const {
    return nenner;
}
$

```

Hörsaalübung

Modellieren Sie ein Klasse "**Lager**", die einen Bestand verwaltet.

Das Lager soll folgende Attribute besitzen:

- minimale und
- maximale Kapazität und
- aktuellen Bestand.

Als Methoden sind

- "einlagern",
- "entnehmen" und
- "bestandAnzeigen"

zu realisieren.

In einem Hauptprogramm soll ein Array von 10 Lager-Objekten angelegt werden.

In einem Menü sind folgende Punkte zu realisieren:

L	Lager auswählen (0-9)
E	Anzahl von Elementen in ausgewähltes Lager einlagern
A	Anzahl von Elementen aus dem ausgewählten Lager entnehmen
B	Bestand des ausgewählten Lagers anzeigen
V	Programm verlassen

7. Abstrakte Datentypen

Hier werden die ADTen Stapel und Schlange beschrieben und jeweils eine Implementierung gezeigt.

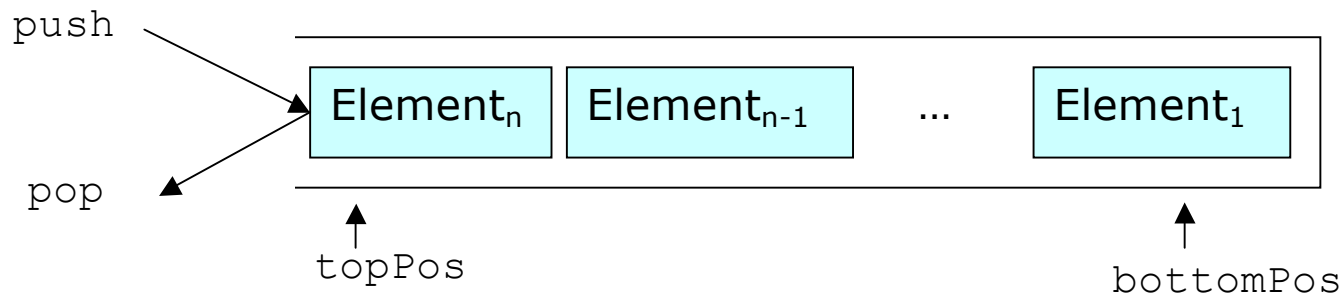
Stapel (engl. stack) werden häufig verwendet, um geschachtelte Strukturen zu bearbeiten. Die Reihenfolge der Bedienung ist prägnant durch die englische Bezeichnung **LIFO** (last in first out) charakterisiert.

Demgegenüber ist eine **Schlange** (engl. queue) eine Struktur mit Verarbeitungssystematik **FIFO** (first in first out). Verwendung finden solche Strukturen z.B. in Anwendungen, bei denen Warteschlangen verarbeitet werden müssen.

7.1.Stack

Die grundlegenden **Operationen** für Stacks sind `push` und `pop`. Durch `push` wird ein Datenelement auf dem Stack abgelegt, `pop` holt das zuletzt abgelegte Element und entfernt es vom Stapel.

Die Position des obersten Elements (=kleinste Verweildauer) bezeichnen wir mit `topPos`, die Position des Elementes mit der größten Verweildauer bezeichnen wir mit `bottomPos`.



Soll das oberste Element nur gelesen werden, könnte dies durch die Folge `pop` und anschließend `push` erfolgen. Einfacher ist es, eine zusätzliche Operation `top` dafür zu definieren. Zusätzlich sollen Operationen zur Abfrage des Status des Stack existieren.

Definition der Operationen:

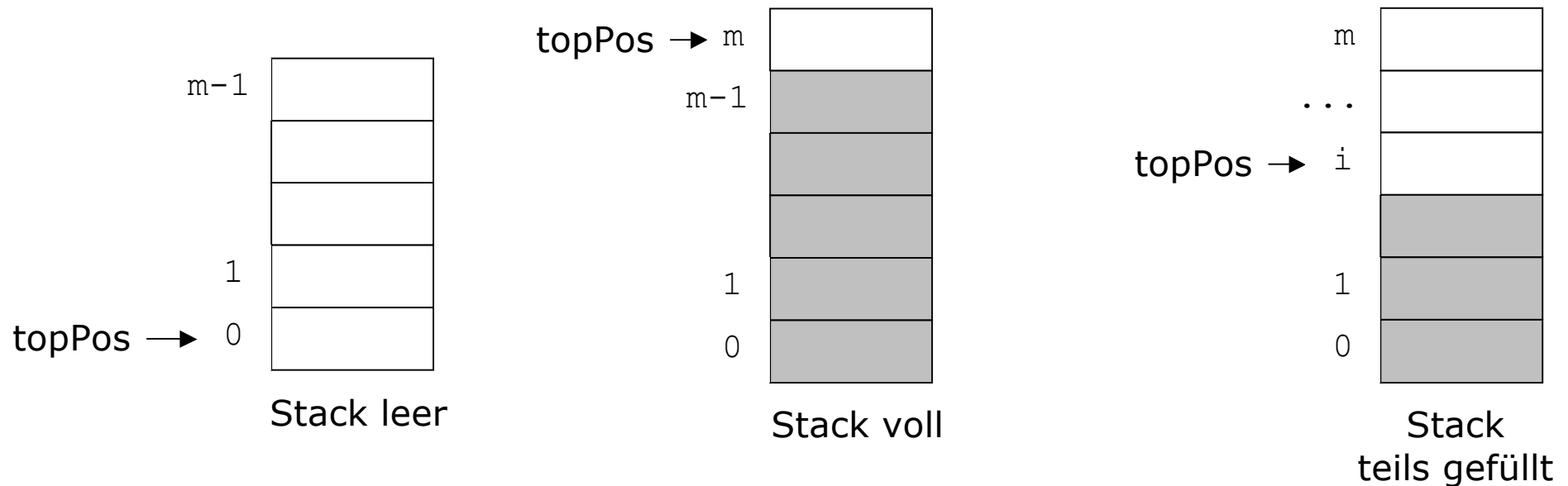
Sei $a = a_n, a_{n-1}, \dots, a_1$ der Stack. Dann sind die Operationen wie folgt definiert:

- `top()` == a_n
 $a = a_n, a_{n-1}, \dots, a_1$
- `push(b)`
 $a = b, a_n, a_{n-1}, \dots, a_1$
- `pop()` == a_n
 $a = a_{n-1}, \dots, a_1$
- `length()` == n
 $a = a_n, a_{n-1}, \dots, a_1$
- `isEmpty()` == `false` (true wenn a leer)
 $a = a_n, a_{n-1}, \dots, a_1$
- `isFull()` == `true`, wenn kein weitere Platz mehr frei ist

7.1.1. Implementierung als Array

Ein Stack wird als Array implementiert, wobei wir als Basistyp hier von `double` ausgehen. Im nächsten Semester werden wir sehen, wie man das mit Klassentemplates eleganter machen kann.

Die folgende Abbildung zeigt die unterschiedlichen Fälle, die bei der Realisierung zu berücksichtigen sind und wie die Variable `topPos` verwendet werden soll:



Zunächst die Definition des ADT Stack.

```
$ cat Stack.h
// Klasse "Stack" als Array von double Werten
class CStack {
    public:
        CStack(unsigned max);           // Stapel für Records vom Typ double
        ~CStack();                     // leerer Stapel für maximal max Records
        void push(double r);           // Destruktor
        double pop();                   // legt Record r oben auf Stapel
        double top();                   // holt obersten Record vom Stapel
        unsigned length();              // liefert den Wert des obersten Records
        bool isEmpty();                 // Anzahl der Records im Stapel
        bool isFull();                 // true, wenn Stapel leer, sonst false
    private:
        unsigned m;                    // true, wenn Stapel voll, sonst false
        unsigned topPos;                // Arraylänge; m: max Anzahl Records
        double *pa;                     // zeigt auf oberste Position im Stack
};                                       // Zeiger auf Array
$
```

Nun die Implementierung des ADT

```
$ cat Stack.cpp
// Elementfunktionen der Klasse "Stack": Implementation als Array
#include <iostream>
using namespace std;
#include "Stack.h"

CStack::CStack(unsigned max)           // Konstruktor: legt Speicherplatz für max Records an
    : topPos(0), m(max) {
    pa = new double[m];
}

CStack::~~CStack() {                   // Destruktor
    delete pa;
}

void CStack::push(double r) {          // oben im Stapel ablegen, Bedingung: Stapel nicht voll
    if (!isFull()) {
        pa[topPos] = r;
        topPos++;
    }
}
```

```

double CStack::pop() {                                // obersten Record holen und entfernen,
                                                        // Bedingung: Stapel nicht leer
    double r;                                          // initialer Record
    if (!isEmpty()) {
        topPos--;
        r = pa[topPos];
    }
    return r;                                         // Achtung: besser Ausnahme werfen, wenn
                                                        // Stack leer ist !
}

double CStack::top() {                                // Wert des obersten Records im nicht leeren Stapel holen
    double r;
    if (!isEmpty())
        r = pa[topPos-1];
    return r;
}

unsigned CStack::length() {                           // Anzahl der Records im Stapel
    return topPos;
}

bool CStack::isEmpty() {                             // prüfen, ob Stapel leer ist
    return topPos==0;
}

```

```
bool CStack::isFull() { // prüfen, ob Stapel voll ist
    return topPos==m;
}
$
```

Zum **Testen** werden Werte eingelesen und ausgegeben:

```
$ cat Stackapp.cpp
//      Main-Funktion für Stack-Beispiele
//      Implementation als Array
//      Eingabe von der Tastatur: maximal 20 Integers, Abschluss mit 9999.
//      Diese Integers werden in dem Stack abgelegt
//      Danach wird der Inhalt des Stacks ausgegeben
//
#include <iostream>
using namespace std;
#include "Stack.h"
#include "Stackdef.h"
```

```

void main() {
    double x;
    int i;

    CStack s(20); // Stack-Objekt erzeugen

    cout << "Stack als Array implementiert\n"
         << "=====\n";

    cout << endl << "Eingabe von Ganzzahlen, Abschluss mit 9999:" << endl;
    while ((cin >> i) && !s.isFull() && i!=9999) {
        x = i;
        s.push(x); // auf Stack ablegen
    }

    cout << endl << endl << "Stackinhalt" << endl;
    while (!s.isEmpty()) {
        cout << s.pop() << endl; // vom Stack holen
    }

}
$

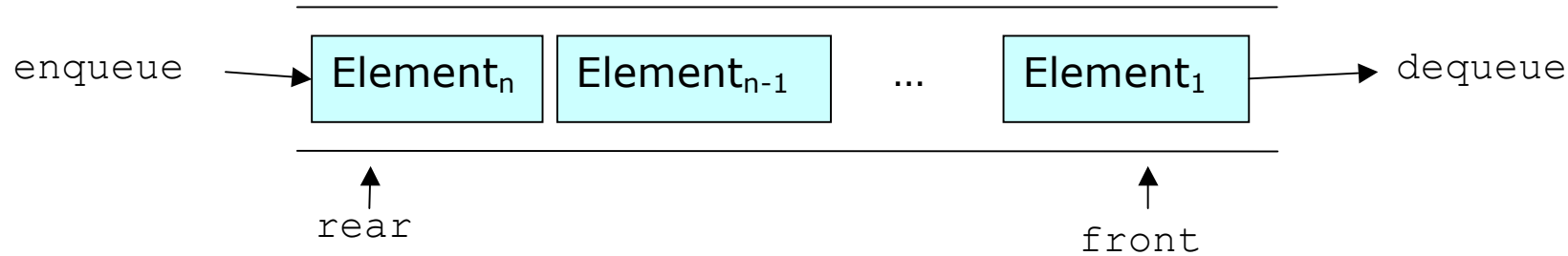
```

Hörsaalübung

Realisieren Sie einen ADT List für verkettete Listen

7.2.Schlangen

Schlangen verwalten Elemente in FIFO Manier.



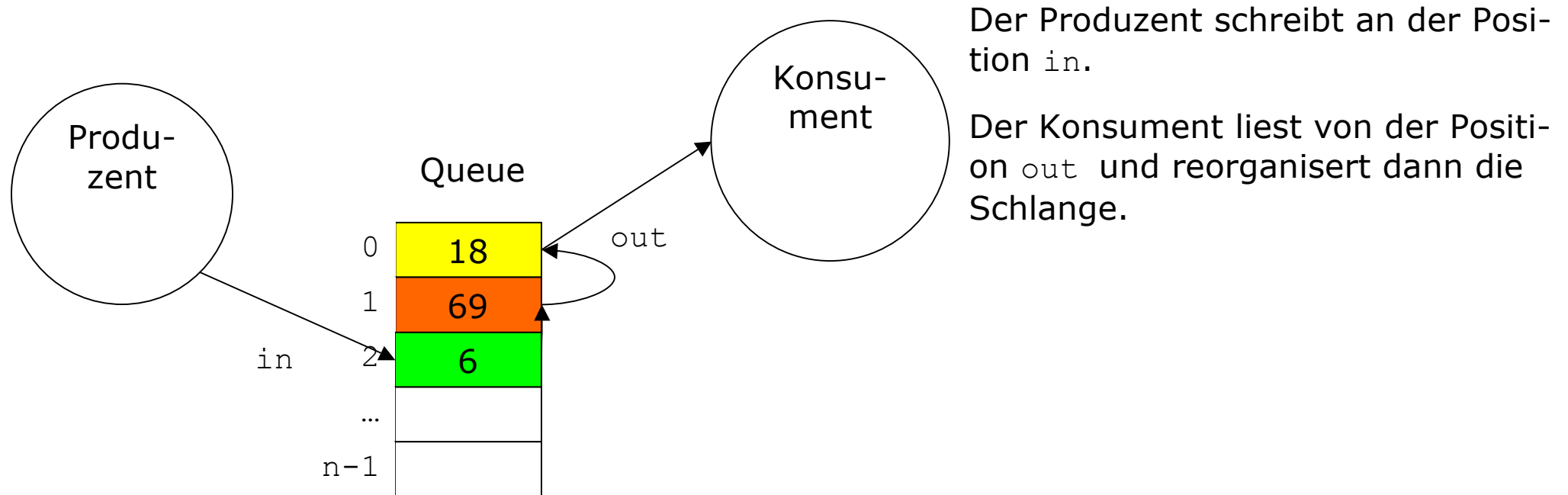
Definition der Operationen:

Sei $a = a_n, a_{n-1}, \dots, a_1$ die Schlange. Dann sind die Operationen wie folgt definiert:

- $\text{dequeue}() == a_1$ nimmt Element, das am längsten wartet und entfernt es
 $a = a_n, a_{n-1}, \dots, a_2$
- $\text{enqueue}(b)$ stellt neues Element in die Schlange „hinten an“
 $a = b, a_n, a_{n-1}, \dots, a_1$
- $\text{front}() == a_1$ Element, das am längsten in der Schlage ist
 $a = a_{n-1}, \dots, a_1$
- $\text{length}() == n$
 $a = a_n, a_{n-1}, \dots, a_1$

- `isEmpty() == false` (true wenn a leer)
 $a = a_n, a_{n-1}, \dots, a_1$
- `isFull() == true`, wenn kein weitere Platz mehr frei ist

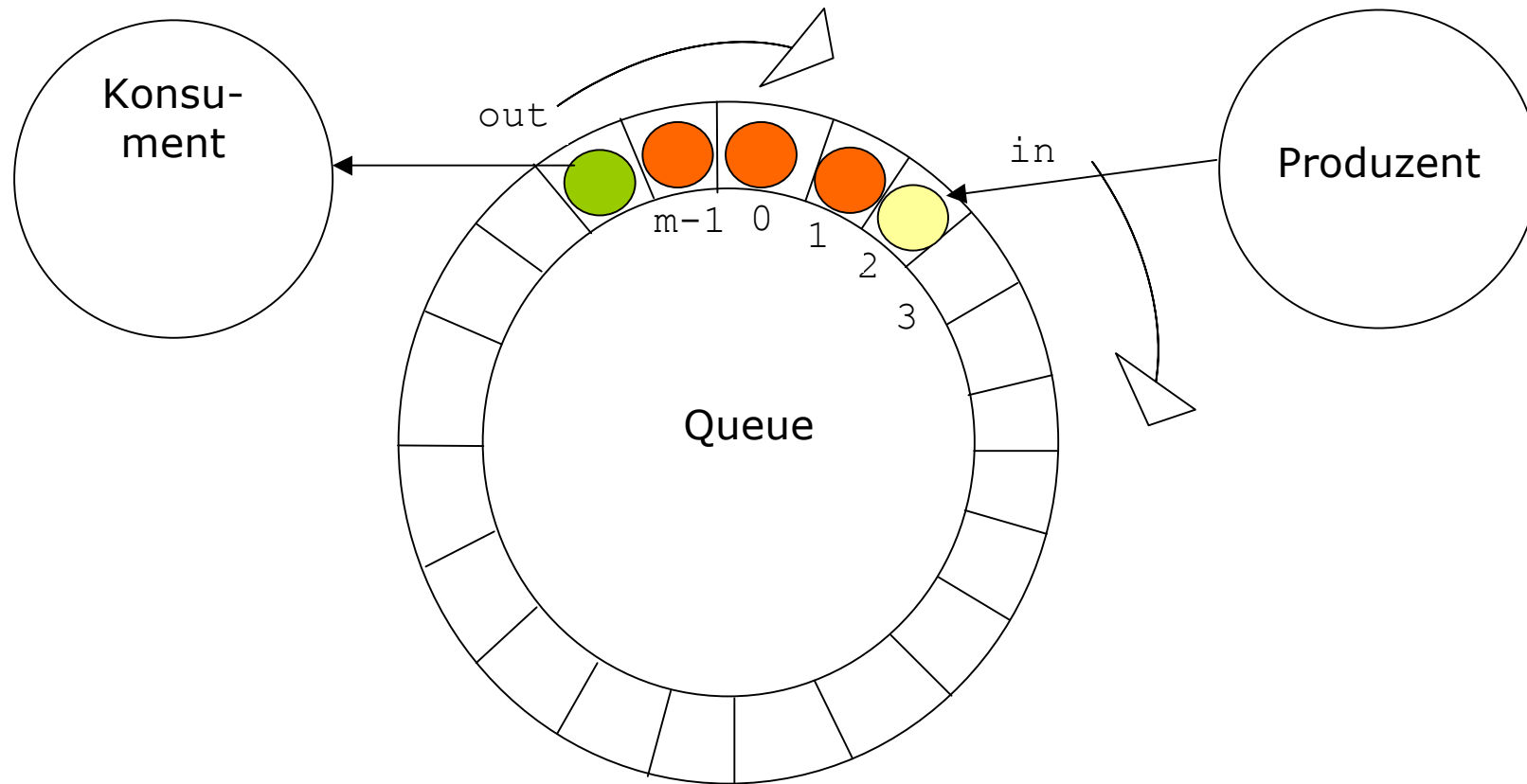
Schlangen könnten analog zu Stacks als Array implementiert werden, gelesen wird von `out`, geschrieben an `in`. Nach dem Lesen müssten dann alle Elemente nach vorne verschoben werden, um nicht schnell an die Arraygrenzen zu stoßen.



Der Produzent schreibt an der Position `in`.

Der Konsument liest von der Position `out` und reorganisiert dann die Schlange.

Dieses Reorganisieren kostet Zeit und Speicherzugriffe. Dies kann vermieden werden, wenn man den Zugriff zirkulär organisiert.



Die Implementierung des **ADT Schlange** als zirkuläres Array ist mit der o.a. Bemerkung analog zum Stack – wir werden eine Realisierung als **zirkulär verkettete Liste** betrachten.

Dabei hat ein Objekt der Klasse `CQueue` Komponenten, die zur Klasse `CQueueNode` gehören; dies sind `double`-Werte, die über Zeiger verkettet sind. Hier werden wir sehen, dass Klassen geschachtelt werden können.

Zunächst die Spezifikation der Klasse:

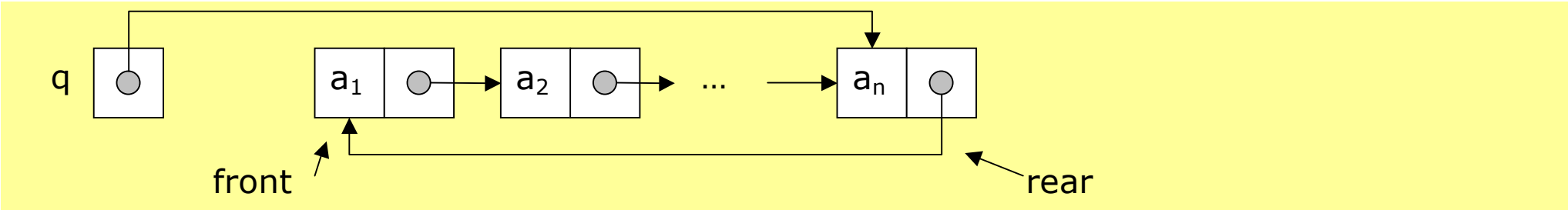
```
$ cat Qulist.h
//      Klasse "Queue" - als verkettete Liste
#ifndef QUEUE
#define QUEUE
class CQueue {
    public:
        CQueue ();                // leere Schlange
        ~CQueue ();              // gibt Schlange frei
        void enqueue (double &r); // fügt Record r am Ende ein
        double dequeue ();        // holt am längsten wartenden Record
        double front ();          // Wert des am längsten wartenden Records
        unsigned length ();       // Anzahl Records in der Schlange
        bool isEmpty ();          // true, wenn Schlange leer ist
        bool isFull ();           // true, wenn Schlange voll ist
        void printQueue ();       // Warteschlange ausgeben
};
```

```

private:
    class CQueueNode { // Knoten einer Liste
    public:
        CQueueNode(double r) // Konstruktor für Listenknoten
        : dvalue(r), next(NULL) { // leerer Body
        }
        double dvalue; // Knotenwert, Typ double
        CQueueNode *next; // Zeiger auf nächsten Knoten
    };
    CQueueNode *q; // Repräsentation der Schlange
                    // Zeiger auf letzten Knoten der Schlange
    unsigned n; // Anzahl Records in der Schlange
};
#endif
$

```

Die Idee der Implementierung des ADT Schlange ist es, eine **zirkulär verkettete Liste** zu verwenden, wobei immer am **Ende eingefügt** wird, und das Ende wieder auf den Anfang zeigt:



Der **Konstruktor** zum Erzeugen einer leeren Schlange muss lediglich die Anzahl der Elemente n auf 0 setzen und den Zeiger q mit NULL initialisieren:

q n

```
CQueue::CQueue () {           // Konstruktor: kreiert leere Schlange
    q = NULL;
    n = 0;
}
```

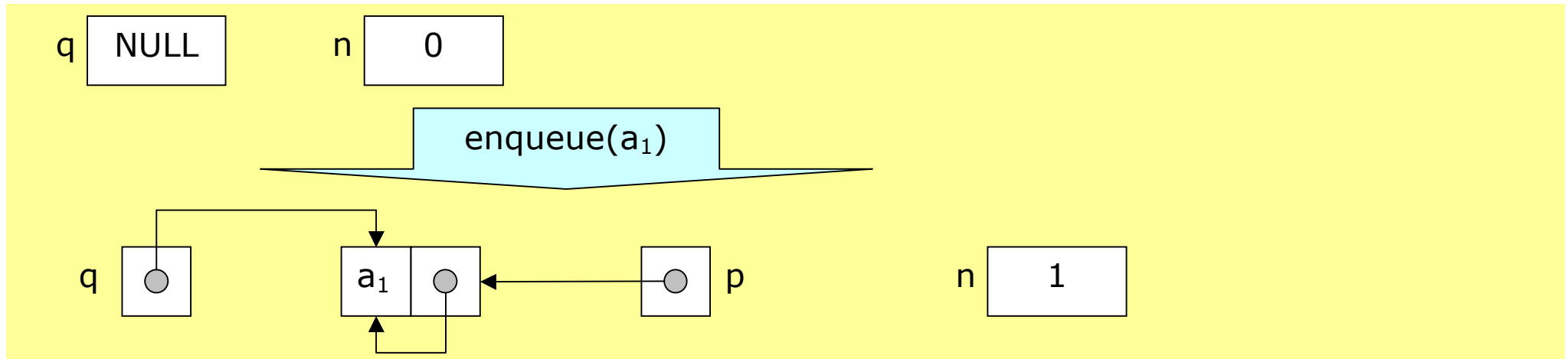
Die Methode `isEmpty()` fragt einfach, ob p auf ein Element zeigt oder nicht.

```
bool CQueue::isEmpty() {           // prüfen, ob die Schlange leer ist
    return q==NULL;
}
```

Eine Schlange, die als verkettete Liste realisiert ist, hat keine Größenbeschränkung, deshalb liefert die Methode `isFull()` stets `false` zurück.

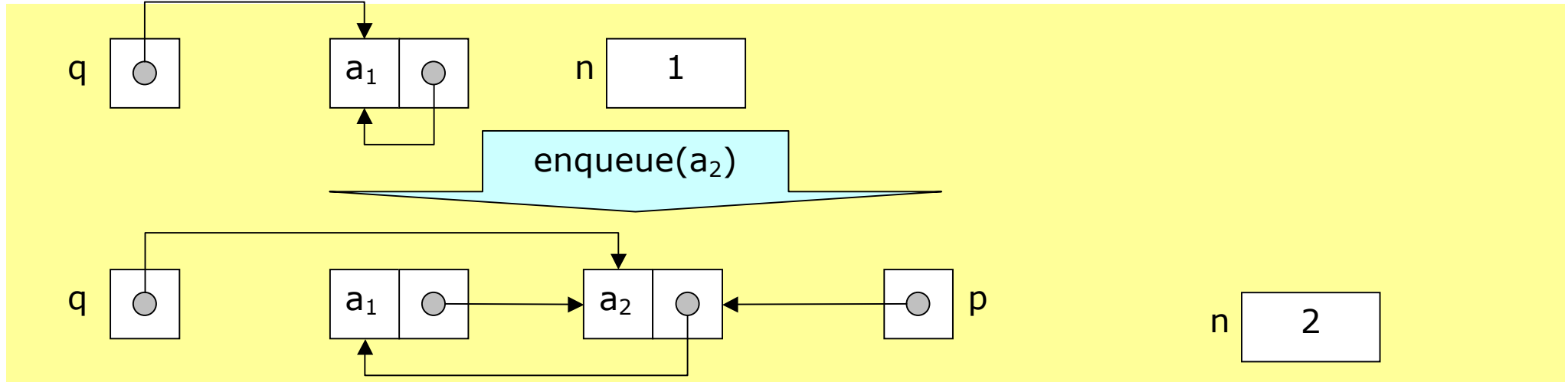
Die Methode **enqueue()** fügt ein neues Element am Ende ein. Auf das Ende zeigt das Attribut q. Dabei ist zu unterscheiden, ob die Liste leer ist oder nicht.

Einfügen in leere Schlange:



```
void CQueue::enqueue(double &r) { // am Queueende einfügen
    CQueueNode *p = new CQueueNode(r);
    if (isEmpty()) p->next = p;
    else {
        ...
    }
    q = p;
    n++;
}
```

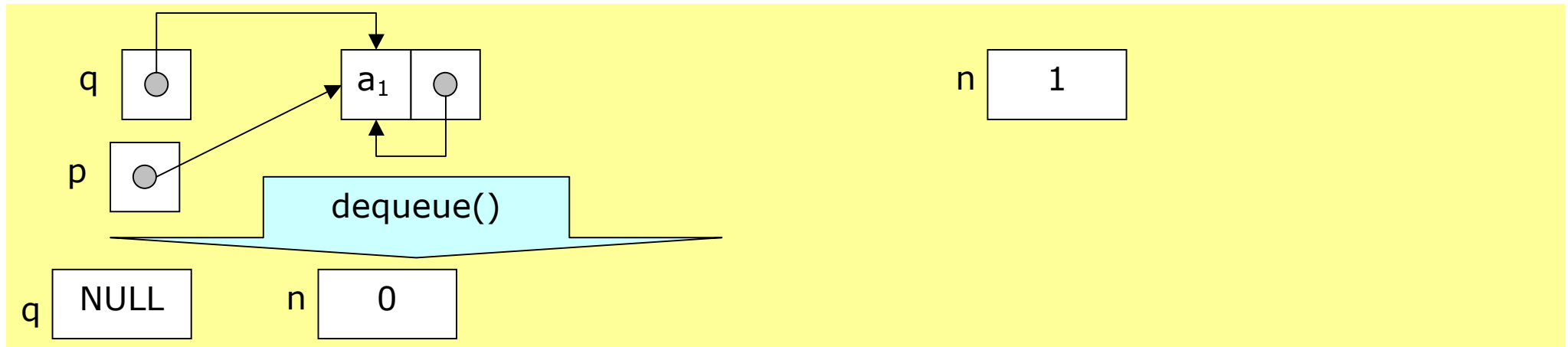
Einfügen in nicht leere Schlange:



```
void CQueue::enqueue(double &r) { // am Queueende einfügen
    CQueueNode *p = new CQueueNode(r);
    if (isEmpty()) p->next = p;
    else {
        p->next = q->next;
        q->next = p;
    }
    q = p;
    n++;
}
```

Die Methode **dequeue()** zum Entfernen eines Elementes muss unterscheiden, ob die Liste nach der Entnahme eines Elementes leer ist oder nicht.

Schlange nach Operation leer:



```
double CQueue::dequeue() {  
    CQueueNode *p = q->next;  
    double r = p->dvalue;  
    if (p==q) q = NULL;  
    else ...  
    delete p;  
    n--;  
    return r;  
}
```

// holt am längsten wartenden Record
// erster Record

// Schlange nach Entnahme leer

Schlange nach Operation **nicht** leer:



```
double CQueue::dequeue() {  
    CQueueNode *p = q->next;  
    double r = p->dvalue;  
    if (p==q) q = NULL;  
    else q->next = p->next;  
    delete p;  
    n--;  
    return r;  
}
```

// holt am längsten wartenden Record

// erster Record

// Schlange nach Entnahme leer

Der **Destruktor** gibt alle Elemente der Liste frei:

```
CQueue::~~CQueue() {                                     // gibt Schlange frei
    if (isEmpty()) return;
    if (n==1) {
        delete q;
        q = NULL;
        return;
    }
    CQueueNode *p = q;                                   // Zeiger auf ersten Record
    while (n) {
        q = p->next;
        delete p;
        p = q;
        n--;
    }
    q = NULL;
}
```

Damit lässt sich eine Testanwendung formulieren:

```

$ cat Qulapp.cpp
//      Main-Funktion für Queues - als Liste
#include <iostream.h>
#include "Qulist.h"

void main() {
    CQueue ws;                // ADT Schlange erzeugen
    for (double d=1; d<=6; d++)
        ws.enqueue(d);       // Element einfügen
    ws.printQueue();         // Schlange ausgeben
    cout << "Front Element: " << ws.front() << endl; // front Element
    cout << "Entferntes Element: " << ws.dequeue() << endl; // Element lesen
    ws.printQueue();         // Schlange ausgeben
}
$

```

```

$ Qulapp
1-ter Record: 1
2-ter Record: 2
...
5-ter Record: 5
6-ter Record: 6
Front Element: 6
Entferntes Element: 1
1-ter Record: 2
2-ter Record: 3
3-ter Record: 4
4-ter Record: 5
5-ter Record: 6
$

```