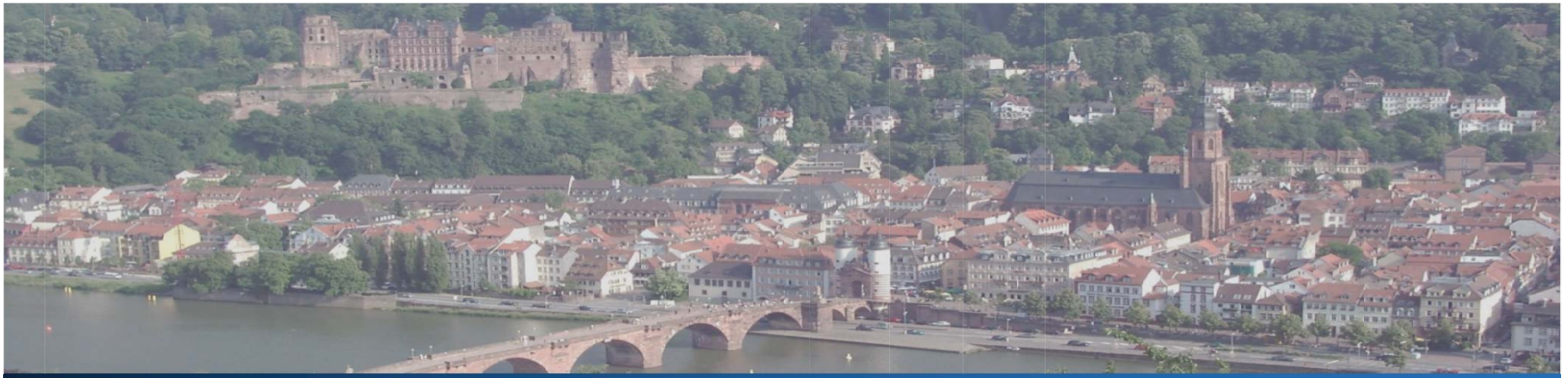
Einführung in die Praktische Informatik

Prof. Björn Ommer **HCI, IWR**

Computer Vision Group



Klassen

- Motivation, Klassendefinition, Objektdefinition
- Kapselung
- Konstruktoren und Destruktoren
- Implementierung der Klassenmethoden
- Klassen im Umgebungsmodell
- Beispiel: Monte-Carlo objektorientiert
- Initialisierung von Unterobjekten
- Selbstreferenz
- Überladen von Funktionen und Methoden
- Objektorientierte und funktionale Programmierung
- Operatoren
- Anwendung: rationale Zahlen objektorientiert
- Beispiel: Turingmaschine
- Abstrakter Datentyp

Motivation

Bisher:

- Funktionen bzw. Prozeduren (Funktion, bei welcher der Seiteneffekt wesentlich ist) als *aktive* Entitäten
- Daten als *passive* Entitäten.

Beispiel:

```
int konto1 = 100;
int konto2 = 200;
int abheben( int& konto, int betrag )
{
  konto = konto - betrag;
  return konto;
}
```

Kritik:

- Auf welchen Daten operiert `abheben`? Es könnte mit jeder **int**-Variablen arbeiten.
- Wir könnten `konto1` auch ohne die Funktion `abheben` manipulieren.
- Nirgends ist der Zusammenhang zwischen den globalen Variablen `konto1`, `konto2` und der Funktion `abheben` erkennbar.

Idee: Verbinde Daten und Funktionen zu einer Einheit!

Klassendefinition

Diese Verbindung von Daten und Funktionen wird durch Klassen (*classes*) realisiert:

Beispiel: Klasse für das Konto:

```
class Konto
{
public:
  int kontostand();
  int abheben( int betrag );
private:
  int k;
};
```

Sieht einer Definition eines zusammengesetzten Datentyps sehr ähnlich.

Syntax: **(Klassendefinition)** Die allgemeine Syntax der Klassendefinition lautet

$$<\text{Klasse}> \quad ::= \quad \underline{\text{class}} <\text{Name}> \underline{\{} <\text{Rumpf}> \underline{\}} \underline{;}$$

Im Rumpf werden sowohl Variablen als auch Funktionen aufgeführt. Bei den Funktionen genügt der Kopf. Die Funktionen einer Klasse heißen Methoden (*methods*). Alle Komponenten (Daten und Methoden) heißen Mitglieder. Die Daten heißen oft Datenmitglieder.

Bemerkung:

- Die Klassendefinition
  - beschreibt, aus welchen Daten eine Klasse besteht,
  - und welche Operationen auf diesen Daten ausgeführt werden können.

- Klassen sind (in C++) keine normalen Datenobjekte. Sie sind nur zur Kompilierungszeit bekannt und belegen daher keinen Speicherplatz.

Objektdefinition

Die Klasse kann man sich als Bauplan vorstellen. Nach diesem Bauplan werden Objekte (*objects*) erstellt, die dann im Rechner existieren. Objekte heißen auch Instanzen (*instances*) einer Klasse.

Objektdefinitionen sehen aus wie Variablendefinitionen, wobei die Klasse wie ein neuer Datentyp erscheint. Methoden werden wie Komponenten eines zusammengesetzten Datentyps selektiert und mit Argumenten wie eine Funktion versehen.

Beispiel:

```
Konto k1;
k1.abheben( 25 );

Konto* pk = &k1;
print( pk->kontostand() );
```

Bemerkung: Objekte haben einen internen Zustand, der durch die Datenmitglieder repräsentiert wird. Objekte haben ein Gedächtnis!

Klassen

- Motivation, Klassendefinition, Objektdefinition
- Kapselung
- Konstruktoren und Destruktoren
- Implementierung der Klassenmethoden
- Klassen im Umgebungsmodell
- Beispiel: Monte-Carlo objektorientiert
- Initialisierung von Unterobjekten
- Selbstreferenz
- Überladen von Funktionen und Methoden
- Objektorientierte und funktionale Programmierung
- Operatoren
- Anwendung: rationale Zahlen objektorientiert
- Beispiel: Turingmaschine
- Abstrakter Datentyp

Kapselung

Der Rumpf einer Klassendefinition zerfällt in zwei Teile:

1. einen öffentlichen Teil, und

2. einen privaten Teil.

Der öffentliche Teil einer Klasse ist die Schnittstelle (*interface*) der Klasse zum restlichen Programm. Diese sollte für den Benutzer der Klasse ausreichende Funktionalität bereitstellen. Der private Teil der Klasse enthält Mitglieder, die zur Implementierung der Schnittstelle benutzt werden.

Bezeichnung: Diese Trennung nennt man Kapselung (*encapsulation*).

Bemerkung:

- Sowohl öffentlicher als auch privater Teil können sowohl Methoden als auch Daten enthalten.
- Öffentliche Mitglieder einer Klasse können von jeder Funktion eines Programmes benutzt werden (etwa die Methode `abheben` in `Konto`).
- Private Mitglieder können nur von den Methoden der Klasse selbst benutzt werden.

Beispiel:

```
Konto k1;
k1.abheben( -25 );          // OK
k1.k = 1000000;             // Fehler !, k private
```

Bemerkung: Kapselung erlaubt uns, das Prinzip der versteckten Information (*information hiding*) zu realisieren. David L. Parnas[16] [CACM, 15(12): 1059–1062, 1972] hat dieses Grundprinzip im Zusammenhang mit der modularen Programmierung so ausgedrückt:

1. One must provide the intended user with all the information needed to use the module correctly, and with nothing more.

2. One must provide the implementor with all the information needed to complete the module, and with nothing more.

[16] David Lorge Parnas, geb. 1941, kanadischer Informatiker.

Bemerkung: Insbesondere sollte eine Klasse alle Implementierungsdetails „verstecken“, die sich möglicherweise in Zukunft ändern werden. Da Änderungen der Implementierung meist Änderung der Datenmitglieder bedeutet, sind diese normalerweise nicht öffentlich!

Zitat: Brooks[17] [The Mythical Man-Month: Essays on Software Engineering, Addison-Wesley, 1975, page 102]:

> . . . but much more often, strategic breakthrough will come from redoing the representation of the data or tables. This is where the heart of a program lies.

Regel: Halte Datenstrukturen geheim!

[17] Fred Brooks, geb. 1931, amerik. Informatiker.

Bemerkung:

- Die „Geheimhaltung“ durch die Trennung **public**/**private** ist kein perfektes Verstecken der Implementation, weil der Benutzer der Klasse ja die Klassendefinition einsehen kann/muss.
- Sie erzwingt jedoch bei gutwilligen Benutzern ein regelkonformes Verwenden der Bibliothek.
- Andererseits schützt sie nicht gegenüber böswilligen Benutzern! (z. B. sollte man nicht erwarten, dass ein Benutzer der Bibliothek ein **private**-Feld `password` nicht auslesen kann!)

Klassen

- Motivation, Klassendefinition, Objektdefinition
- Kapselung
- Konstruktoren und Destruktoren
- Implementierung der Klassenmethoden
- Klassen im Umgebungsmodell
- Beispiel: Monte-Carlo objektorientiert
- Initialisierung von Unterobjekten
- Selbstreferenz
- Überladen von Funktionen und Methoden
- Objektorientierte und funktionale Programmierung
- Operatoren
- Anwendung: rationale Zahlen objektorientiert
- Beispiel: Turingmaschine
- Abstrakter Datentyp

Konstruktoren und Destruktoren

Objekte werden – wie jede Variable – erzeugt und zerstört, sei es automatisch oder unter Programmiererkontrolle.

Diese Momente erfordern oft spezielle Beachtung, so dass *jede* Klasse die folgenden Operationen zur Verfügung stellt:

- Mindestens einen Konstruktor, der aufgerufen wird, nachdem der Speicher für ein Objekt bereitgestellt wurde. Der Konstruktor hat die Aufgabe, die Datenmitglieder des Objektes geeignet zu initialisieren.

- Einen Destruktor, der aufgerufen wird, bevor der vom Objekt belegte Speicher freigegeben wird. Der Destruktor kann entsprechende Aufräumarbeiten durchführen (Beispiele folgen).

Bemerkung:

- Ein Konstruktor ist eine Methode mit demselben Namen wie die Klasse selbst und kann mit beliebigen Argumenten definiert werden. Er hat $keinen$ Rückgabewert.
- Ein Destruktor ist eine Methode, deren Name mit einer Tilde $\sim$ beginnt, gefolgt vom Namen der Klasse. Ein Destruktor hat weder Argumente noch einen Rückgabewert.
- Gibt der Programmierer keinen Konstruktor und/oder Destruktor an, so erzeugt der Übersetzer Default-Versionen. Der Default-Konstruktor hat keine Argumente.

Beispiel: Ein Beispiel für eine Klassendefinition mit Konstruktor und Destruktor:

```
class Konto
{
public:
  Konto( int start );      // Konstruktor
  ~Konto();                // Destruktor
  int kontostand();
  int abheben( int betrag );
private:
  int k;
};
```

Der Konstruktor erhält ein Argument, welches das Startkapital des Kontos sein soll (Implementierung folgt gleich). Erzeugt wird so ein Konto mittels

```
Konto k1( 1000 );  // Argumente des Konstruktors nach Objektname
Konto k2;  // Fehler! Klasse hat keinen argumentlosen Konstruktor
```

Klassen

- Motivation, Klassendefinition, Objektdefinition
- Kapselung
- Konstruktoren und Destruktoren
- Implementierung der Klassenmethoden
- Klassen im Umgebungsmodell
- Beispiel: Monte-Carlo objektorientiert
- Initialisierung von Unterobjekten
- Selbstreferenz
- Überladen von Funktionen und Methoden
- Objektorientierte und funktionale Programmierung
- Operatoren
- Anwendung: rationale Zahlen objektorientiert
- Beispiel: Turingmaschine
- Abstrakter Datentyp

Implementierung der Klassenmethoden

Bisher haben wir noch nicht gezeigt, wie die Klassenmethoden implementiert werden. Dies ist Absicht, denn wir wollten deutlich machen, dass man nur die Definition einer Klasse *und die Semantik ihrer Methoden* wissen muss, um sie zu verwenden.

Nun wechseln wir auf die Seite des Implementierers einer Klasse. Hier nun ein vollständiges Programm mit Klassendefinition und Implementierung der Klasse `Konto`:

Programm: (Konto.cc)

```
#include "fcpp.hh"

class Konto
{
public:
   Konto( int start );      // Konstruktor
   ~Konto();                // Destruktor
   int kontostand();
   int abheben( int betrag );
private:
   int bilanz;
};

Konto::Konto( int startkapital )
{
```

```
    bilanz = startkapital;
    print( "Konto mit ", bilanz, " eingerichtet", 0 );
}

Konto::~Konto()
{
    print( "Konto mit ", bilanz, " aufgelöst", 0 );
}

int Konto::kontostand()
{
    return bilanz;
}

int Konto::abheben( int betrag )
{
    bilanz = bilanz - betrag;
```

```
    return bilanz;
}

int main()
{
    Konto k1( 100 ), k2( 200 );

    k1.abheben( 50 );
    k2.abheben( 300 );
}
```

Bemerkung:

- Die Definitionen der Klassenmethoden sind normale Funktionsdefinitionen, nur der Funktionsname lautet

  <Klassenname>::<Methodenname>

- Klassen bilden einen eigenen $Namensraum$. So ist `abheben` keine global sichtbare Funktion. Der Name `abheben` ist nur innerhalb der Definition von `Konto` sichtbar.

- Außerhalb der Klasse ist der Name erreichbar, wenn ihm der Klassenname gefolgt von zwei Doppelpunkten ($scope\ resolution\ operator$) vorangestellt wird.

Klassen im Umgebungsmodell

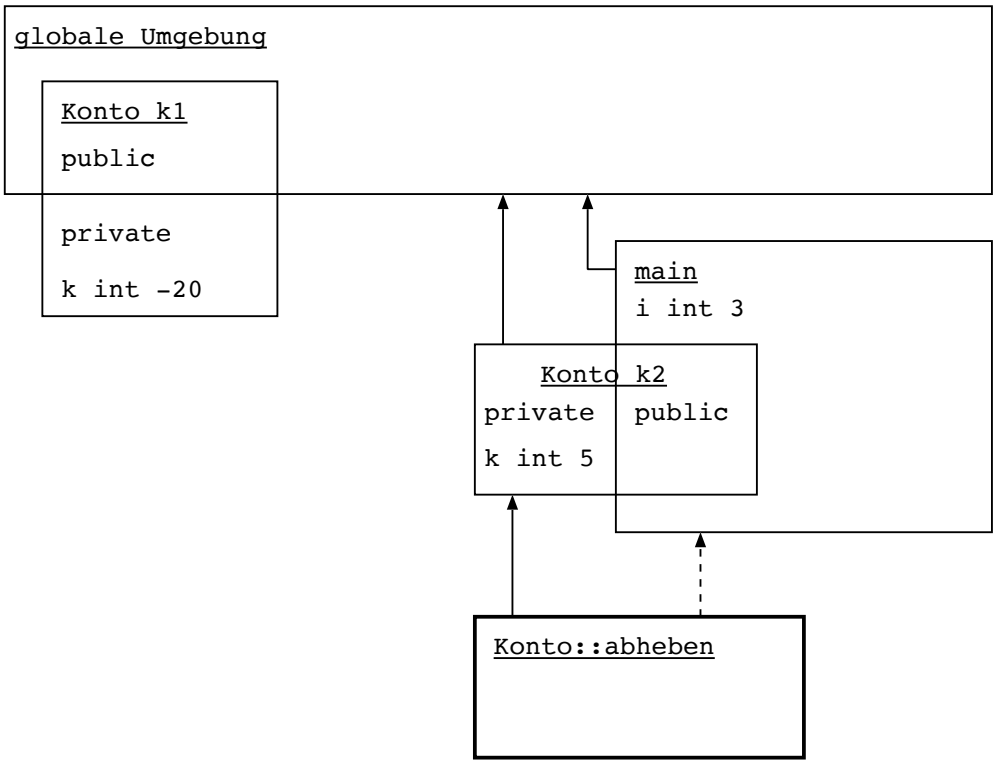
```
class Konto; // wie oben

Konto k1( 0 );

void main()
{
  int i = 3;
  Konto k2( 0 );

  k1.abheben( 20 );
  k2.abheben( -5 );
}
```

In `k2.abheben(-5)`



Bemerkung:

- Jedes Objekt definiert eine eigene Umgebung.
- Die öffentlichen Daten einer Objektumgebung überlappen mit der Umgebung, in der das Objekt definiert ist, und sind dort auch sichtbar.
- Der Methodenaufruf erzeugt eine neue Umgebung unterhalb der Umgebung des zugehörigen Objektes

Folgerung:

- Öffentliche Daten von `k1` sind global sichtbar.
- Öffentliche Daten von `k2` sind in `main` sichtbar.

- Private Daten von `k1` und `k2` sind von Methoden der Klasse `Konto` zugreifbar (jede Methode eines Objektes hat Zugriff auf die Mitglieder aller Objekte dieser Klasse, sofern bekannt).

Bemerkung: Die Lebensdauer von Objekten (bzw. Objektvariablen) ist genauso geregelt wie die von anderen Variablen.

Beispiel: Monte-Carlo objektorientiert

Wir betrachten nochmal das Beispiel der Bestimmung von π mit Hilfe von Zufallszahlen.

Bestandteile:

- Zufallsgenerator: Liefert bei Aufruf eine Zufallszahl.
- Experiment: Führt das Experiment einmal durch und liefert im Erfolgsfall 1, sonst 0.
- Monte-Carlo: Führt Experiment N mal durch und berechnet relative Häufigkeit.

Zufallsgenerator

Programm: Der Zufallsgenerator lässt sich hervorragend als Klasse formulieren. Er kapselt die aktuelle Zufallszahl als internen Zustand.

```
class Zufall
{
public:
  Zufall( unsigned int anfang );
  unsigned int ziehe_zahl();
private:
  int x;
};

Zufall::Zufall( unsigned int anfang )
{
  x = anfang;
```

```
}

// Implementierung ohne lange Arithmetik
// siehe Numerical Recipes, Kap. 7.
unsigned int Zufall::ziehe_zahl()
{
  // a = 7^5, m = 2^31-1
  const int ia = 16807,  im = 2147483647;
  const int iq = 127773, ir = 2836;
  const int k = x / iq;
  x = ia*(x-k*iq) - ir*k;
  if ( x < 0 ) x = x + im;
  return x;
}
```

Vorteile:

- Durch die Angabe des Konstruktors ist sichergestellt, dass der Zufallsgenerator initialisiert werden muss. Beachte: Wenn ein Konstruktor angegeben ist, so gibt es keinen Default-Konstruktor!

- Die Realisierung des Zufallsgenerators ist nach außen nicht sichtbar (`x` ist **private**). Beispielsweise könnte man nun problemlos die Implementation so abändern, dass man intern mit längeren Zahlen arbeitet.

Klasse für das Experiment

Programm:

```
class Experiment
{
public:
  Experiment( Zufall& z ); // Konstruktor
  int durchfuehren();      // einmal ausfuehren
private:
  Zufall& zg; // Merke Zufallsgenerator
  unsigned int ggT( unsigned int a, unsigned int b );
};

Experiment::Experiment( Zufall& z ) : zg( z ) {}

unsigned int Experiment::ggT( unsigned int a,
```

```
                                        unsigned int b )
{
  if ( b == 0) return a;
  else         return ggT( b, a % b );
}

int Experiment::durchfuehren()
{
  unsigned int x1 = zg.ziehe_zahl();
  unsigned int x2 = zg.ziehe_zahl();
  if ( ggT( x1, x2 ) == 1 )
    return 1;
  else
    return 0;
}
```

Bemerkung: Die Klasse `Experiment` enthält (eine Referenz auf) ein Objekt einer Klasse als Unterobjekt. Für diesen Fall gibt es eine spezielle Form des Konstruktors, die weiter unten erläutert wird.

Monte-Carlo-Funktion und Hauptprogramm

Programm:

```
#include "fcpp.hh"          // fuer print
#include "Zufall.cc"        // Code fuer die beiden
#include "Experiment.cc"    // Klassen hereinziehen

double montecarlo( Experiment& e, int N )
{
  int erfolgreich = 0;

  for ( int i=0; i<N; i=i+1 )
    erfolgreich = erfolgreich + e.durchfuehren();

  return ((double) erfolgreich) / ((double) N);
}
```

```
int main( int argc, char *argv[] )
{
  Zufall z( 93267 ); // ein Zufallsgenerator
  Experiment e( z ); // ein Experiment

  print( sqrt( 6.0/montecarlo( e,
                               readarg_int( argc,
                                            argv,
                                            1 ) ) ) );
}
```

Diskussion:

- Es gibt keine globale Variable mehr! `Zufall` kapselt den Zustand intern.
- Wir könnten auch mehrere unabhängige Zufallsgeneratoren haben.
- Die Funktion `montecarlo` kann nun mit dem Experiment parametrisiert werden. Dadurch kann man das Experiment leicht austauschen: beispielsweise erhält man π auch, indem man Punkte in $(-1, 1)^2$ würfelt und misst, wie oft sie im Einheitskreis landen.

Initialisierung von Unterobjekten

Ein Objekt kann Objekte anderer Klassen als Unterobjekte enthalten. Um in diesem Fall die ordnungsgemäße Initialisierung des Gesamtobjekts sicherzustellen, gibt es eine erweiterte Form des Konstruktors selbst.

Syntax: (Erweiterter Konstruktor) Ein Konstruktor für eine Klasse mit Unterobjekten hat folgende allgemeine Form:

<Konstruktor> ::= <Klassenname>::<Klassenname> (<ArgListe>) :
<UnterObjekt> (<ArgListe>)
{, <UnterObjekt> (<ArgListe>)}
{ <Rumpf> }

Die Aufrufe nach dem : sind Konstruktoraufrufe für die Unterobjekte. Deren Argumente sind Ausdrücke, die die formalen Parameter des Konstruktors des Gesamtobjektes enthalten können.

Eigenschaften:

- Bei der Ausführung jedes Konstruktors (egal ob einfacher, erweiterter oder default) werden *erst* die Konstruktoren der Unterobjekte ausgeführt und dann der Rumpf des Konstruktors.

- Wird der Konstruktoraufruf eines Unterobjektes im erweiterten Konstruktor weggelassen, so wird dessen argumentloser Konstruktor aufgerufen. Gibt es keinen solchen, wird ein Fehler gemeldet.

- Beim Destruktor wird erst der Rumpf abgearbeitet, dann werden die Destruktoren der Unterobjekte aufgerufen. Falls man keinen programmiert hat, wird die Default-Version verwendet.

- Dies nennt man hierarchische Konstruktion/Destruktion.

Erinnerung: Eingebaute Datentypen und Zeiger haben keine Konstruktoren und werden nicht initialisiert (es sei denn man initialisiert sie explizit).

Anwendung: `Experiment` enthält eine Referenz als Unterobjekt. Mit einer Instanz der Klasse `Experiment` wird auch diese Referenz erzeugt. Referenzen müssen aber *immer* initialisiert werden, daher muss die erweiterte Form des Konstruktors benutzt werden.

Es ist in diesem Fall nicht möglich, die Referenz im Rumpf des Konstruktors zu initialisieren.

Frage: Was würde sich ändern, wenn man ein `Zufall`-Objekt statt der Referenz speichern würde?

Selbstreferenz

Innerhalb jeder Methode einer Klasse `T` ist ein Zeiger **this** vom Typ `T*` definiert, der auf das Objekt zeigt, dessen Methode aufgerufen wurde.

Beispiel: Folgendes Programmfragment zeigt eine gleichwertige Implementierung von `abheben`:

```
int Konto::abheben( int betrag )
{
  this->k = this->k - betrag;
  return this->k; // neuer Kontostand
}
```

Bemerkung: Anders ausgedrückt, ist die alte Form von `abheben` syntaktischer Zucker für die Form mit **this**. Der Nutzen von **this** wird sich später zeigen (Verkettung von Operationen).

Überladen von Funktionen und Methoden

C++ erlaubt es, mehrere Funktionen *gleichen Namens* aber mit unterschiedlicher Signatur (Zahl und Typ der Argumente) zu definieren.

Beispiel:

```
int summe()                                  { return 0; }
int summe( int i )                           { return i; }
int summe( int i, int j )                    { return i + j; }
double summe( double a, double b )           { return a + b; }

int main()
{
  int i[2];
  double x[2];
  short c;
```

Klassen

- Motivation, Klassendefinition, Objektdefinition
- Kapselung
- Konstruktoren und Destruktoren
- Implementierung der Klassenmethoden
- Klassen im Umgebungsmodell
- Beispiel: Monte-Carlo objektorientiert
- Initialisierung von Unterobjekten
- Selbstreferenz
- Überladen von Funktionen und Methoden
- Objektorientierte und funktionale Programmierung
- Operatoren
- Anwendung: rationale Zahlen objektorientiert
- Beispiel: Turingmaschine
- Abstrakter Datentyp

Überladen von Funktionen und Methoden

C++ erlaubt es, mehrere Funktionen *gleichen Namens* aber mit unterschiedlicher Signatur (Zahl und Typ der Argumente) zu definieren.

Beispiel:

```
int summe()                                  { return 0; }
int summe( int i )                           { return i; }
int summe( int i, int j )                    { return i + j; }
double summe( double a, double b )           { return a + b; }

int main()
{
```

```
int summe()
int summe( int i )
int summe( int i, int j )
double summe( double a, double b )
```

```
    int i[2];
    double x[2];
    short c;
    i[1] = summe();                    // erste Version
    i[1] = summe( 3 );                 // zweite Version
    i[0] = summe( i[0], i[1] );        // dritte Version
    x[0] = summe( x[0], x[1] );        // vierte Version
    i[0] = summe( i[0], c );           // dritte Version
    i[0] = summe( x[0], i[1] );        // Fehler, mehrdeutig
}
```

Dabei bestimmt der Übersetzer anhand der Zahl und Typen der Argumente, welche Funktion aufgerufen wird. Der Rückgabewert ist dabei unerheblich.

Bezeichnung: Diesen Mechanismus nennt man Überladen von Funktionen.

Automatische Konversion

Schwierigkeiten entstehen durch automatische Konversion eingebauter numerischer Typen. Der Übersetzer geht nämlich in folgenden Stufen vor:

1. Versuche passende Funktion ohne Konversion oder mit trivialen Konversionen (z. B. Feldname nach Zeiger) zu finden. Man spricht von exakter Übereinstimmung. Dies sind die ersten vier Versionen oben.

2. Versuche innerhalb einer Familie von Typen ohne Informationsverlust zu konvertieren und so eine passende Funktion zu finden. Z. B. ist erlaubt, **bool** nach **int**, **short** nach **int**, **int** nach **long**, **float** nach **double**, etc. Im obigen Beispiel wird c in Version 5 nach **int** konvertiert.

3. Versuche Standardkonversionen (Informationsverlust!) anzuwenden: **int** nach **double**, **double** nach **int** usw.

4. Gibt es verschiedene Möglichkeiten auf *einer* der vorigen Stufen, so wird ein Fehler gemeldet.

Tip: Verwende Überladen möglichst nur so, dass die Argumente mit einer der definierten Signaturen exakt übereinstimmen!

Überladen von Methoden

Auch Methoden einer Klasse können überladen werden. Dies benutzt man gerne für den Konstruktor, um mehrere Möglichkeiten der Initialisierung eines Objektes zu ermöglichen:

```
class Konto
{
public:
  Konto();                  // Konstruktor 1
  Konto( int start );       // Konstruktor 2
  int konto_stand();
  int abheben( int betrag );
private:
  int k;                    // Zustand
};
```

```
Konto::Konto() { k = 0; }
Konto::Konto( int start ) { k = start; }
```

Jetzt können wir ein Konto auf zwei Arten erzeugen:

```
Konto k1;        // Hat Wert 0
Konto k2(100); // Hundert Euro
```

Bemerkung:

- Eine Klasse muss einen Konstruktor ohne Argumente haben, wenn man Felder dieses Typs erzeugen will.
- Ein Default-Konstruktor wird nur erzeugt, wenn kein Konstruktor explizit programmiert wird.

Das Überladen von Funktionen ist eine Form von Polymorphismus womit man meint:

Eine Schnittstelle, viele Methoden.

Aber: Es ist sehr verwirrend, wenn überladene Funktionen sehr verschiedene Bedeutung haben. Dies sollte man vermeiden.

Klassen

- Motivation, Klassendefinition, Objektdefinition
- Kapselung
- Konstruktoren und Destruktoren
- Implementierung der Klassenmethoden
- Klassen im Umgebungsmodell
- Beispiel: Monte-Carlo objektorientiert
- Initialisierung von Unterobjekten
- Selbstreferenz
- Überladen von Funktionen und Methoden
- Objektorientierte und funktionale Programmierung
- Operatoren
- Anwendung: rationale Zahlen objektorientiert
- Beispiel: Turingmaschine
- Abstrakter Datentyp

Programm: (Inkrementierer.cc)

```
#include "fcpp.hh"          // fuer print

class Inkrementierer
{
public:
  Inkrementierer( int n ) { inkrement = n; }
  int eval( int n ) { return n + inkrement; }
private:
  int inkrement;
};

void schleife( Inkrementierer& ink )
{
  for ( int i=1; i<10; i++ )
    print( ink.eval( i ) );
```

```
}

int main()
{
  Inkrementierer ink( 10 );
  schleife( ink );
}
```

Bemerkung:

- Man beachte die Definition der Methoden innerhalb der Klasse. Dies ist zwar kürzer, legt aber die Implementation der Schnittstelle offen.
- Die innerhalb einer Klasse definierten Methoden werden „inline“ (d. h. ohne Funktionsaufruf) übersetzt. Bei Änderungen solcher Methoden muss daher aufrufender Code neu übersetzt werden!
- Man sollte dieses Feature daher nur mit Vorsicht verwenden (z. B. bei nur lokal verwendeten Klassen oder wenn das Inlining gewünscht wird).
- Eine erweiterte Schnittstelle zur Simulation funktionaler Programme erhält man in der STL (*Standard Template Library*) mit **#include** <functional>.

Klassen

- Motivation, Klassendefinition, Objektdefinition
- Kapselung
- Konstruktoren und Destruktoren
- Implementierung der Klassenmethoden
- Klassen im Umgebungsmodell
- Beispiel: Monte-Carlo objektorientiert
- Initialisierung von Unterobjekten
- Selbstreferenz
- Überladen von Funktionen und Methoden
- Objektorientierte und funktionale Programmierung
- Operatoren
- Anwendung: rationale Zahlen objektorientiert
- Beispiel: Turingmaschine
- Abstrakter Datentyp

Operatoren

In C++ hat man auch bei selbstgeschriebenen Klassen die Möglichkeit, einem Ausdruck wie `a+b` eine Bedeutung zu geben:

Idee: Interpretiere den Ausdruck `a+b` als `a.operator+(b)`, d. h. die Methode `operator+` des Objektes `a` (des linken Operanden) wird mit dem Argument `b` (rechter Operand) aufgerufen:

```
class X
{
public:
  X operator+( X b );
};

X X::operator+( X b ) { .... }
X a, b, c;
c = a + b;
```

Bemerkung:

- `operator+` ist also ein ganz normaler Methodenname, nur die Methode wird aus der Infixschreibweise heraus aufgerufen.

- Diese Technik ist insbesondere bei Klassen sinnvoll, die mathematische Konzepte realisieren, wie etwa rationale Zahlen, Vektoren, Polynome, Matrizen, gemischtzahlige Artihmetik, Arithmetik beliebiger Genauigkeit.

- Man sollte diese Technik zurückhaltend verwenden. Zum Beispiel sollte man + nur überladen, wenn die Operation wirklich eine Addition im mathematischen Sinn ist.

- Auch eckige Klammern `[]`, Dereferenzierung `->`, Vergleichsoperatoren `<`, `>`, `==` und sogar die Zuweisung `=` können (um-)definiert werden. `<<`, `>>` spielt bei Ein-/Ausgabe eine Rolle.

Klassen

- Motivation, Klassendefinition, Objektdefinition
- Kapselung
- Konstruktoren und Destruktoren
- Implementierung der Klassenmethoden
- Klassen im Umgebungsmodell
- Beispiel: Monte-Carlo objektorientiert
- Initialisierung von Unterobjekten
- Selbstreferenz
- Überladen von Funktionen und Methoden
- Objektorientierte und funktionale Programmierung
- Operatoren
- Anwendung: rationale Zahlen objektorientiert
- Beispiel: Turingmaschine
- Abstrakter Datentyp

Anwendung: rationale Zahlen objektorientiert

Definition der Klasse (`Rational.hh`):

```
class Rational
{
private:
  int n, d;
  int ggT( int a, int b );
public:
  // (lesender) Zugriff auf Zaehler und Nenner
  int numerator();
  int denominator();

  // Konstruktoren
  Rational( int num, int denom ); // rational
  Rational( int num );            // ganz
```

```
    Rational();                                  // Null

    // Ausgabe
    void print();

    // Operatoren
    Rational operator+( Rational q );
    Rational operator-( Rational q );
    Rational operator*( Rational q );
    Rational operator/( Rational q );
};
```

Programm: Implementierung der Methoden (`Rational.cc`):

```
int Rational::numerator()
{
  return n;
}

int Rational::denominator()
{
  return d;
}

void Rational::print()
{
  ::print( n, "/", d, 0 );
}
```

```
// ggT zum kuerzen
int Rational::ggT( int a, int b )
{
  return ( b == 0 ) ? a : ggT( b, a % b );
}

// Konstruktoren
Rational::Rational( int num, int denom )
{
  int t = ggT( num, denom );
  if ( t != 0 )
  {
    n = num / t;
    d = denom / t;
  }
  else
  {
```

```
        n = num;
        d = denom;
    }
}

Rational::Rational( int num )
{
    n = num;
    d = 1;
}

Rational::Rational()
{
    n = 0;
    d = 1;
}
```

```
// Operatoren
Rational Rational::operator+( Rational q )
{
  return Rational( n*q.d + q.n*d, d*q.d );
}

Rational Rational::operator-( Rational q )
{
  return Rational( n*q.d - q.n*d, d*q.d );
}

Rational Rational::operator*( Rational q )
{
  return Rational( n*q.n, d*q.d );
}

Rational Rational::operator/( Rational q )
```

```
{
  return Rational( n*q.d, d*q.n );
}
```

Programm: Lauffähiges Beispiel (`UseRational.cc`):

```
#include "fcpp.hh"          // fuer print
#include "Rational.hh"
#include "Rational.cc"

int main()
{
  Rational p( 3, 4 ), q( 5, 3 ), r;

  p.print(); q.print();
  r = ( p + q*p ) * p*p;
  r.print();

  return 0;
}
```

Bemerkung:

- Es ist eine gute Idee die Definition der Klasse (Schnittstelle) und die Implementierung der Methoden in getrennte Dateien zu schreiben. Dafür haben sich in C++ die Dateiendungen .hh („Headerdatei“) und .cc („Quelldatei“) eingebürgert. (Auch: .hpp, .hxx, .h, .cpp, .cxx).

- Später wird dies die sog. „getrennte Übersetzung“ ermöglichen.

- Wie schon früher erwähnt, ist die Implementierung einer leistungsfähigen gemischtzahligen Arithmetik eine hochkomplexe Aufgabe, für welche die Klasse `Rational` nur ein erster Ansatz sein kann.

- Sehr notwendig wäre auf jeden Fall die Verwendung von Ganzzahlen beliebiger Länge anstatt von **int** als Bausteine für `Rational`.

Wiederholung

Monte-Carlo Beispiel objektorientiert → Klassen helfen globale Variablen zu vermeiden.

Hierarchische Konstruktion/Destruktion von Unterobjekten. Wichtig z. B. bei Initialisierung von Referenzen.

Selbstreferenz → jede Klasse hat einen **this** Zeiger. Ist später wichtig bei Vererbung.

Überladen von Funktionen und Methoden. Nützlich z. B. bei Konstruktoren oder gemischter Arithmetik.

Operatoren erlauben infix Schreibweise.

Klassen erlauben die Übergabe von Funktionen als Parameter (später Funktoren).

Trennung von Schnittstelle und Implementierung in separaten Dateien.

Klassen

- Motivation, Klassendefinition, Objektdefinition
- Kapselung
- Konstruktoren und Destruktoren
- Implementierung der Klassenmethoden
- Klassen im Umgebungsmodell
- Beispiel: Monte-Carlo objektorientiert
- Initialisierung von Unterobjekten
- Selbstreferenz
- Überladen von Funktionen und Methoden
- Objektorientierte und funktionale Programmierung
- Operatoren
- Anwendung: rationale Zahlen objektorientiert
- Beispiel: Turingmaschine
- Abstrakter Datentyp

Beispiel: Turingmaschine

Ein großer Vorteil der objektorientierten Programmierung ist, dass man seine Programme sehr „problemnah“ formulieren kann. Als Beispiel zeigen wir, wie man eine Turingmaschine realisieren könnte. Diese besteht aus den drei Komponenten

- Band
- Programm
- eigentliche Turingmaschine

Es bietet sich daher an, diese Einheiten als Klassen zu definieren.

Band

Programm: **(Band.hh)**

```
// Klasse fuer ein linksseitig begrenztes Band
// einer Turingmaschine.
// Das Band wird durch eine Zeichenkette aus
// Elementen des Typs char realisiert
class Band
{
public:
  // Initialisiere Band mit s, fuelle Rest
  // mit dem Zeichen init auf.
  // Setze aktuelle Bandposition auf linkes Ende.
  Band( std::string s, char init );

  // Lese Symbol unter dem Lesekopf
  char lese();
```

```
  // Schreibe und gehe links
  void schreibe_links( char symbol );

  // Schreibe und gehe rechts
  void schreibe_rechts( char symbol );

  // Drucke aktuellen Bandinhalt bis zur
  // maximal benutzten Position
  void drucke();
private:
  enum { N = 100000 };   // maximal nutzbare Groesse
  char band[N];          // das Band
  int pos;               // aktuelle Position
  int benutzt;           // bisher beschriebener Teil
};
```

TM-Programm

Programm: **(Programm.hh)**

```
// Eine Klasse, die das Programm einer
// Turingmaschine realisiert.
// Zustaende sind vom Typ int
// Bandalphabet ist der Typ char
// Anfangszustand ist Zustand in der ersten Zeile
// Endzustand ist Zustand in der letzten Zeile
class Programm
{
public:
  // Symbole fuer links/rechts
  enum R { links, rechts };

  // Erzeuge leeres Programm
```

```
Programm();

// definiere Zustandsuebergaenge
// Mit Angabe des Endzustandes ist die
// Programmierphase beendet
void zeile( int q_ein, char s_ein,
            char s_aus, R richt, int q_aus );
void zeile( int endzustand );

// lese Zustandsuebergang in Abhaengigkeit
// von akt. Zustand und gelesenem Symbol
char Ausgabe( int zustand, char symbol );
R Richtung( int zustand, char symbol );
int Folgezustand( int zustand, char symbol );

// Welcher Zustand ist Anfangszustand
int Anfangszustand();
```

```
  // Welcher Zustand ist Endzustand
  int Endzustand();

private:
  // Finde die Zeile zu geg. Zustand/Symbol
  // Liefere true, falls so eine Zeile gefunden
  // wird, sonst false
  bool FindeZeile( int zustand, char symbol );

  enum { N = 1000 };     // maximale Anzahl Uebergaenge
  int zeilen;            // Anzahl Zeilen in Tabelle
  bool fertig;           // Programmierphase beendet
  int Qaktuell[N];       // Eingabezustand
  char eingabe[N];       // Eingabesymbol
  char ausgabe[N];       // Ausgabesymbol
  R richtung[N];         // Ausgaberichtung
```

```
    int Qfolge[N];          // Folgezustand
    int letztesQ;           // Merke Zustand, Eingabe
    char letzteEingabe;     // und Zeilennummer des
    int letzteZeile;        // letzten Zugriffes.
};
```

Bemerkung: Man beachte die Definition des lokalen Datentyps `R` durch `enum`. Andererseits wird eine Form von `enum`, bei der den Konstanten gleich Zahlen zugewiesen werden, verwendet, um die Konstante `N` innerhalb der Klasse `Programm` zur Verfügung zu stellen.

Turingmaschine

Programm: **(TM.hh)**

```
// Klasse, die eine Turingmaschine realisiert
class TM
{
public:
  // Konstruiere Maschine mit Programm
  // und Band
  TM( Programm& p, Band& b );

  // Mache einen Schritt
  void Schritt();

  // Liefere true falls sich Maschine im
  // Endzustand befindet
```

```
    bool Endzustand();

  private:
    Programm& prog; // Merke Programm
    Band& band;     // Merke Band
    int q;          // Merke akt. Zustand
};
```

Programm: (TM.cc)

```
// Konstruiere die TM mit Programm und Band
TM::TM( Programm& p, Band& b ) : prog( p ), band( b )
{
  q = p.Anfangszustand();
}

// einen Schritt machen
```

```
void TM::Schritt()
{
  // lese Bandsymbol
  char s = band.lese();

  // schreibe Band
  if ( prog.Richtung( q, s ) == Programm::links )
    band.schreibe_links( prog.Ausgabe( q, s ) );
  else
    band.schreibe_rechts( prog.Ausgabe( q, s ) );

  // bestimme Folgezustand
  q = prog.Folgezustand( q, s );
}

// Ist Endzustand erreicht?
bool TM::Endzustand()
```

```
{
  if ( q == prog.Endzustand() )
    return true;
  else
    return false;
}
```

Turingmaschinen-Hauptprogramm

Programm: **(Turingmaschine.cc)**

```
#include "fcpp.hh" // fuer print

#include "Band.hh"   // Inkludiere Quelldateien
#include "Band.cc"
#include "Programm.hh"
#include "Programm.cc"
#include "TM.hh"
#include "TM.cc"

int main( int argc, char *argv[] )
{
  // Initialisiere ein Band
  Band b( "1111", '0' );
```

```
b.drucke();

// Initialisiere ein Programm
Programm p;
p.zeile( 1, '1', 'X', Programm::rechts, 2 );
p.zeile( 2, '1', '1', Programm::rechts, 2 );
p.zeile( 2, '0', 'Y', Programm::links, 3 );
p.zeile( 3, '1', '1', Programm::links, 3 );
p.zeile( 3, 'X', '1', Programm::rechts, 4 );
p.zeile( 4, 'Y', '1', Programm::rechts, 8 );
p.zeile( 4, '1', 'X', Programm::rechts, 5 );
p.zeile( 5, '1', '1', Programm::rechts, 5 );
p.zeile( 5, 'Y', 'Y', Programm::rechts, 6 );
p.zeile( 6, '1', '1', Programm::rechts, 6 );
p.zeile( 6, '0', '1', Programm::links, 7 );
p.zeile( 7, '1', '1', Programm::links, 7 );
p.zeile( 7, 'Y', 'Y', Programm::links, 3 );
```

```
    p.zeile( 8 );

    // Baue eine Turingmaschine
    TM tm( p, b );

    // Simuliere Turingmaschine
    while ( !tm.Endzustand() )
    { // Solange nicht Endzustand
      tm.Schritt() ;              // mache einen Schritt
      b.drucke();                 // und drucke Band
    }

    return 0;                          // fertig.
}
```

Die TM realisiert das Programm „Verdoppeln einer Einserkette”

Experiment: Ausgabe des oben angegebenen Programms:

```
4  Symbole auf Band initialisiert
[1]111
Programm mit  14  Zeilen definiert
Anfangszustand  1
Endzustand  8
X[1]11
X1[1]1
X11[1]
X111[0]
X11[1]Y
X1[1]1Y
X[1]11Y
[X]111Y
1[1]11Y
1X[1]1Y
1X1[1]Y
```

```
1X11[Y]
1X11Y[0]
1X11[Y]1
1X1[1]Y1
1X[1]1Y1
1[X]11Y1
11[1]1Y1
11X[1]Y1
11X1[Y]1
11X1Y[1]
11X1Y1[0]
11X1Y[1]1
11X1[Y]11
11X[1]Y11
11[X]1Y11
111[1]Y11
111X[Y]11
```

```
111XY[1]1
111XY1[1]
111XY11[0]
111XY1[1]1
111XY[1]11
111X[Y]111
111[X]Y111
1111[Y]111
11111[1]11
```

Kritik:

- Das Band könnte seine Größe dynamisch verändern.
- Statt eines einseitig unendlichen Bandes könnten wir auch ein zweiseitig unendliches Band realisieren.
- Das Finden einer Tabellenzeile könnte durch bessere Datenstrukturen beschleunigt werden.
- Bei Fehlerzuständen bricht das Programm nicht ab. Fehlerbehandlung ist keine triviale Sache.

Aber: Diese Änderungen betreffen jeweils nur die Implementierung einer einzelnen Klasse (`Band` oder `Programm`) und beeinflussen die Implementierung anderer Klassen nicht!

Klassen

- Motivation, Klassendefinition, Objektdefinition
- Kapselung
- Konstruktoren und Destruktoren
- Implementierung der Klassenmethoden
- Klassen im Umgebungsmodell
- Beispiel: Monte-Carlo objektorientiert
- Initialisierung von Unterobjekten
- Selbstreferenz
- Überladen von Funktionen und Methoden
- Objektorientierte und funktionale Programmierung
- Operatoren
- Anwendung: rationale Zahlen objektorientiert
- Beispiel: Turingmaschine
- Abstrakter Datentyp

Abstrakter Datentyp

Eng verknüpft mit dem Begriff der Schnittstelle ist das Konzept des abstrakten Datentyps (ADT). Ein ADT besteht aus

- einer Menge von Objekten, und
- einem Satz von Operationen auf dieser Menge, sowie
- einer genauen Beschreibung der Semantik der Operationen.

Bemerkung:

- Das Konzept des ADT ist unabhängig von einer Programmiersprache, die Beschreibung kann in natürlicher (oder mathematischer) Sprache abgefasst werden.
- Der ADT beschreibt, was die Operationen tun, aber nicht, wie sie das tun. Die Realisierung ist also nicht Teil des ADT!
- Die Klasse ist der Mechanismus zur Konstruktion von abstrakten Datentypen in C++. Allerdings fehlt dort die Beschreibung der Semantik der Operationen! Diese kann man als Kommentar über die Methoden schreiben.
- In manchen Sprachen (z. B. Eiffel, PLT Scheme) ist es möglich, die Semantik teilweise zu berücksichtigen (Design by Contract: zur Funktionsdefinition kann man Vorbedingungen und Nachbedingungen angeben).

Beispiel 1: Positive m-Bit-Zahlen im Computer

Der ADT „Positive m-Bit-Zahl“ besteht aus

- Der Teilmenge $P_m = \{0, 1, \ldots, 2^m - 1\}$ der natürlichen Zahlen.
- Der Operation $+_m$ so dass für $a, b \in P_m$: $a +_m b = (a + b) \bmod 2^m$.
- Der Operation $-_m$ so dass für $a, b \in P_m$: $a -_m b = ((a - b) + 2^m) \bmod 2^m$.
- Der Operation $*_m$ so dass für $a, b \in P_m$: $a *_m b = (a * b) \bmod 2^m$.
- Der Operation $/_m$ so dass für $a, b \in P_m$: $a/_m b = q$, q die größte Zahl in P_m so dass $q *_m b \leq a$.

Bemerkung:

- Die Definition dieses ADT stützt sich auf die Mathematik (natürliche Zahlen und Operationen darauf).

- In C++ (auf einer 32-Bit Maschine) entsprechen **unsigned char**, **unsigned short**, **unsigned int** den Werten $m = 8, 16, 32$.

Beispiel 2: ADT Stack

- Ein Stack S über X besteht aus einer geordneten Folge von n Elementen aus X: $S = \{s_1, s_2, \ldots, s_n\}$, $s_i \in X$. Die Menge aller Stacks $\mathcal{S}$ besteht aus $allen$ $möglichen$ $Folgen$ der $Länge$ $n \geq 0$.
- Operation $new : \emptyset \to \mathcal{S}$, die einen leeren Stack erzeugt.
- Operation $empty : \mathcal{S} \to \{w, f\}$, die prüft ob der Stack leer ist.
- Operation $push : \mathcal{S} \times X \to \mathcal{S}$ zum Einfügen von Elementen.
- Operation $pop : \mathcal{S} \to \mathcal{S}$ zum Entfernen von Elementen.
- Operation $top : \mathcal{S} \to X$ zum Lesen des obersten Elementes.

- Die Operationen erfüllen folgende Regeln:

1. $empty(new()) = w$
2. $empty(push(S, x)) = f$
3. $top(push(S, x)) = x$
4. $pop(push(S, x)) = S$

Bemerkung:

- Die einzige Möglichkeit einen Stack zu erzeugen ist die Operation new.
- Die Regeln erlauben uns formal zu zeigen, welches Element nach einer beliebigen Folge von $push$ und pop Operationen zuoberst im Stack ist:

$$top(pop(push(push(push(new(), x_1), x_2), x_3))) =$$

$$top(push(push(new(), x_1), x_2)) = x_2$$

- Auch nicht gültige Folgen lassen sich erkennen:

$$pop(pop(push(new(), x_1))) = pop(new())$$

und dafür gibt es keine Regel!

Bemerkung: Abstrakte Datentypen, wie Stack, die Elemente einer Menge X aufnehmen, heißen auch Container. Wir werden noch eine Reihe von Containern kennenlernen: Feld, Liste (in Varianten), Queue, usw.

Beispiel 3: Das Feld

Wie beim Stack wird das Feld über einer Grundmenge X erklärt. Auch das Feld ist ein Container.

Das charakteristische an einem Feld ist der indizierte Zugriff. Wir können das Feld daher als eine Abbildung einer Indexmenge $I \subset \mathbb{N}$ in die Grundmenge X auffassen.

Die $Indexmenge$ $I \subseteq \mathbb{N}$ sei beliebig, aber im folgenden fest gewählt. Zur Abfrage der Indexmenge gebe es folgende Operationen:

- Operation min liefert kleinsten Index in I.
- Operation max liefert größten Index in I.
- Operation $isMember : \mathbb{N} \to \{w, f\}$. $isMember(i)$ liefert wahr falls $i \in I$, ansonsten falsch.

Den ADT Feld definieren wir folgendermaßen:

- Ein Feld f ist eine Abbildung der Indexmenge I in die Menge der möglichen Werte X, d. h. $f : I \to X$. Die Menge aller Felder $\mathcal{F}$ ist die Menge aller solcher Abbildungen.
- Operation $new : X \to \mathcal{F}$. $new(x)$ erzeugt neues Feld mit Indexmenge I (und initialisiert mit x, siehe unten).
- Operation $read : \mathcal{F} \times I \to X$ zum Auswerten der Abbildung.
- Operation $write : \mathcal{F} \times I \times X \to \mathcal{F}$ zum Manipulieren der Abbildung.
- Die Operationen erfüllen folgende Regeln:

1. $read(new(x), i) = x$ für alle $i \in I$.
2. $read(write(f, i, x), i) = x$.

3. $read(write(f, i, x), j) = read(f, j)$ für $i \neq j$.

Bemerkung:

- In unserer Definition darf $I \subset \mathbb{N}$ beliebig aber fest gewählt werden. Es sind also auch nichtzusammenhängende Indexmengen erlaubt.

- Als Variante könnte man die Manipulation der Indexmenge erlauben (die Indexmenge sollte dann als weiterer ADT definiert werden).