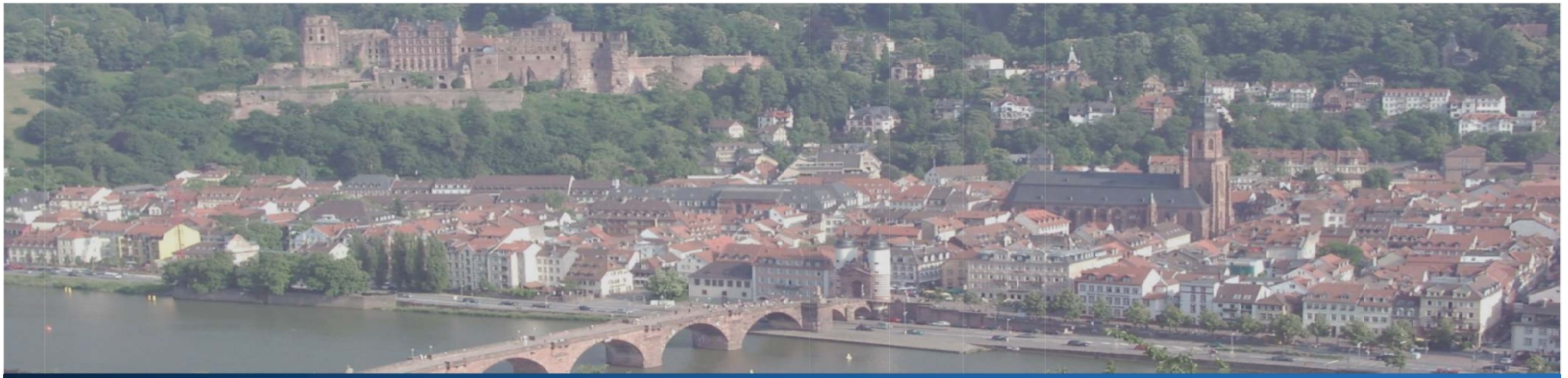
Einführung in die Praktische Informatik

Prof. Björn Ommer **HCI, IWR**

Computer Vision Group



Klassen und dynamischer Speicher

- Klassendefinition
- Konstruktor
- Indizierter Zugriff
- Copy-Konstruktor
- Zuweisungsoperator
- Hauptprogramm
- Default-Methoden
- C++ Ein- und Ausgabe

Klassen und dynamische Speicherverwaltung

Erinnerung: Nachteile von eingebauten Feldern in C/C++:

- Ein eingebautes Feld kennt seine Größe nicht, diese muss immer extra mitgeführt werden, was ein Konsistenzproblem mit sich bringt.
- Bei dynamischen Feldern ist der Programmierer für die Freigabe des Speicherplatzes verantwortlich.
- Eingebaute Felder sind äquivalent zu Zeigern und können daher nur *by reference* übergeben werden.
- Eingebaute Felder prüfen nicht, ob der Index im erlaubten Bereich liegt.
- Manchmal bräuchte man Verallgemeinerungen, z. B. andere Indexmengen.

Klassendefinition

Unsere Feldklasse soll Elemente des Grundtyps **float** aufnehmen. Hier ist die Klassendefinition:

Programm: **(SimpleFloatArray.hh)**

```
class SimpleFloatArray
{
public:
  // Neues Feld mit s Elementen, I=[0,s-1]
  SimpleFloatArray( int s, float f );

  // Copy-Konstruktor
  SimpleFloatArray( const SimpleFloatArray& );

  // Zuweisung von Feldern
  SimpleFloatArray& operator=(const SimpleFloatArray&);
```

```
// Destruktor: Gebe Speicher frei
~SimpleFloatArray();

// Indizierter Zugriff auf Feldelemente
// keine Ueberpruefung ob Index erlaubt
float& operator[]( int i );

// Anzahl der Indizes in der Indexmenge
int numIndices();

// kleinster Index
int minIndex();

// größter Index
int maxIndex();
```

```
    // Ist der Index in der Indexmenge?
    bool isMember( int i );

  private:
    int n;          // Anzahl Elemente
    float* p;       // Zeiger auf built-in array
};
```

Bemerkung: Man beachte, dass diese Implementierung das eingebaute Feld nutzt.

Konstruktor

Programm: **(SimpleFloatArrayImp.cc)**

```
SimpleFloatArray::SimpleFloatArray( int s, float v )
{
  n = s;
  try
  {
    p = new float[n];
  }
  catch ( std::bad_alloc )
  {
    n = 0;
    throw;
  }
  for ( int i=0; i<n; i=i+1 ) p[i] = v;
```

```
}

SimpleFloatArray::~SimpleFloatArray() { delete[] p; }

int SimpleFloatArray::numIndices() { return n; }

int SimpleFloatArray::minIndex() { return 0; }

int SimpleFloatArray::maxIndex() { return n - 1; }

bool SimpleFloatArray::isMember( int i )
{
  return ( i>=0 && i<n );
}
```

Ausnahmen

Bemerkung:

- Oben kann in der Operation **new** das Ereignis eintreten, dass nicht genug Speicher vorhanden ist. Dann setzt diese Operation eine sogenannte Ausnahme (*exception*), die in der **catch**-Anweisung abgefangen wird.

- Gute Fehlerbehandlung (Reaktionen auf Ausnahmen) ist in einem großen, professionellen Programm sehr wichtig!

- Die Schwierigkeit ist, dass man oft an der Stelle des Erkennens des Ereignisses nicht weiss, wie man darauf reagieren soll.

- Hier wird in dem Objekt vermerkt, dass das Feld die Größe 0 hat und auf eine Fehlerbehandlung an anderer Stelle verwiesen.

Indizierter Zugriff

Erinnerung: Die Operationen $read$ und $write$ des ADT Feld werden bei eingebauten Feldern durch den Operator `[]` und die Zuweisung realisiert:

```
x = 3 * a[i] + 17.5;
a[i] = 3 * x + 17.5;
```

Unsere neue Klasse soll sich in dieser Beziehung wie ein eingebautes Feld verhalten. Dies gelingt durch die Definition eines Operators `operator[]`:

Programm: **(SimpleFloatArrayIndex.cc)**

```
float& SimpleFloatArray::operator[](  int  i  )
{
  return p[i];
}
```

Bemerkung:

- `a[i]` bedeutet, dass der `operator[]` von `a` mit dem Argument `i` aufgerufen wird.

- Der Rückgabewert von `operator[]` muss eine Referenz sein, damit `a[i]` auf der linken Seite der Zuweisung stehen kann. Wir wollen ja das i-te Element des Feldes verändern und keine Kopie davon.

Copy–Konstruktor

Schließlich ist zu klären, was beim Kopieren von Feldern passieren soll. Hier sind zwei Situationen zu unterscheiden:

1. Es wird ein *neues* Objekt erzeugt welches mit einem existierenden Objekt initialisiert wird. Dies ist der Fall bei
   - Funktionsaufruf mit call by value: der aktuelle Parameter wird auf den formalen Parameter kopiert.
   - Objekt wird als Funktionswert zurückgegeben: Ein Objekt wird in eine temporäre Variable im Stack des Aufrufers kopiert.
   - Initialisierung von Objekten mit existierenden Objekten bei der Definition, also `SimpleFloatArray a(b); SimpleFloatArray a = b;`

2. Kopieren eines Objektes *auf ein bereits existierendes Objekt*, das ist die Zuweisung.

Im ersten Fall wird von C++ der sogenannte Copy-Konstruktor aufgerufen. Ein Copy-Konstruktor ist ein Konstruktor der Gestalt

<Klassenname> (const <Klassenname> &);

Als Argument wird also eine Referenz auf ein Objekt desselben Typs übergeben. Dabei bedeutet **const**, dass das Argumentobjekt nicht manipuliert werden darf.

Programm: **(SimpleFloatArrayCopyCons.cc)**

```
SimpleFloatArray::SimpleFloatArray( const SimpleFloatArray& a )
{
  n = a.n;
  p = new float[n];
  for ( int i=0; i<n; i=i+1 )
    p[i] = a.p[i];
}
```

Bemerkung:

- Unser Copy-Konstruktor allokiert ein neues Feld und kopiert alle Elemente des Argumentfeldes.

- Damit gibt es *immer nur jeweils einen Zeiger auf ein dynamisch erzeugtes, eingebautes Feld*. Der Destruktor kann dieses eingebaute Feld gefahrlos löschen!

Beispiel:

```
int f()
{
  SimpleFloatArray a( 100, 0.0 ); // Feld mit 100 Elem.
  SimpleFloatArray b = a;         // Aufruf Copy-Konstr.
  ... // mach etwas schlaues
} // Destruktoren rufen delete[] ihres eingeb. Feldes auf
```

Bemerkung:

- Hier hat man mit der dynamischen Speicherverwaltung der eingebauten Felder nichts mehr zu tun, und es können auch keine Fehler passieren.

- Dieses Verhalten des Copy-Konstruktors nennt man *deep copy*.

- Alternativ könnte der Copy-Konstruktor nur den Zeiger in das neue Objekt kopieren (*shallow copy*). Hier dürfte der Destruktor das Feld aber nicht einfach freigeben, weil noch Referenzen bestehen könnten! (Abhilfen: *reference counting*, *garbage collection*)

Zuweisungsoperator

Bei einer Zuweisung `a = b` soll das Objekt rechts des =-Zeichens auf das *bereits initialisierte* Objekt links des =-Zeichens kopiert werden. In diesem Fall ruft C++ den `operator=` des links stehenden Objektes mit dem rechts stehenden Objekt als Argument auf.

Programm: (SimpleFloatArrayAssign.cc)

```
SimpleFloatArray& SimpleFloatArray::operator=
  ( const SimpleFloatArray& a )
{
  // nur bei verschiedenen Objekten ist was tun
  if ( &a != this )
  {
    if ( n != a.n )
    {
      // allokiere fuer this ein
```

```
        // Feld der Groesse a.n
        delete[] p; // altes Feld loeschen
        n = a.n;
        p = new float[n]; // keine Fehlerbeh.
      }
      for ( int i=0; i<n; i=i+1 ) p[i] = a.p[i];
    }

    // Gebe Referenz zurueck damit a = b = c klappt
    return *this;
  }
```

Bemerkung:

- Haben beide Felder unterschiedliche Größe, so wird für das Feld links vom Zuweisungszeichen ein neues eingebautes Feld der korrekten Größe erzeugt.

- Der Zuweisungsoperator ist in C/C++ so definiert, dass er gleichzeitig den zugewiesenen Wert hat. Somit werden Ausdrücke wie `a = b = 0` oder `return tabelle[i] = n` möglich.

Hauptprogramm

Programm: **(UseSimpleFloatArray.cc)**

```
#include <iostream>
#include "SimpleFloatArray.hh"
#include "SimpleFloatArrayImp.cc"
#include "SimpleFloatArrayIndex.cc"
#include "SimpleFloatArrayCopyCons.cc"
#include "SimpleFloatArrayAssign.cc"

void show( SimpleFloatArray f )
{
  std::cout << "#( ";
  for ( int i=f.minIndex(); i<=f.maxIndex(); i++ )
    std::cout << f[i] << " ";
  std::cout << ")" << std::endl;
```

```
}

int main()
{
  SimpleFloatArray a( 10, 0.0 ); // erzeuge Felder
  SimpleFloatArray b( 5, 5.0 );

  for ( int i=a.minIndex(); i<=a.maxIndex(); i++ )
    a[i] = i;

  show( a ); // call by value, ruft Copy-Konstruktor
  b = a;     // ruft operator= von b
  show( b );

  // hier wird der Destruktor beider Objekte gerufen
}
```

Bemerkung:

- Jeder Aufruf der Funktion `show` kopiert das Argument mittels des Copy-Konstruktors. (Für Demonstrationszwecke: eigentlich sollte man in `show` eine Referenz verwenden!)

- Entscheidend ist, dass der Benutzer gar nicht mehr mit dynamischer Speicherverwaltung konfrontiert wird.

- Hier wird erstmals „richtige“ C++ Ausgabe verwendet. Das werden wir noch behandeln.

Default-Methoden

Für folgende Methoden einer Klasse `T` erzeugt der Übersetzer automatisch Default-Methoden, sofern man keine eigenen definiert:

- Argumentloser Konstruktor `T();`
  Dieser wird erzeugt, wenn man keinen anderen Konstruktor außer dem Copy-Konstruktor angibt. Hierarchische Konstruktion von Unterobjekten.

- Copy-Konstruktor `T( const T& );`
  Kopiert alle Mitglieder in das neue Objekt (*memberwise copy*) unter Benutzung von deren Copy-Konstruktor.

- Destruktor `~T();` Hierarchische Destruktion von Unterobjekten.

- Zuweisungsoperator `T& operator=( const T& );`
  Kopiert alle Mitglieder des Quellobjektes auf das Zielobjekt unter Nutzung der jeweiligen Zuweisungsoperatoren.

- Adress-of-Operator (`&`) mit Standardbedeutung.

Bemerkung:

- Der Konstruktor (ob default oder selbstdefiniert) ruft rekursiv die Konstruktoren von selbstdefinierten Unterobjekten auf.

- Ebenso der Destruktor.

- Enthält ein Objekt Zeiger auf andere Objekte und ist für deren Speicherverwaltung verantwortlich, so wird man wahrscheinlich alle oben genannten Methoden speziell schreiben müssen (außer dem &-Operator). Die Klasse `SimpleFloatArray` illustriert dies.

C++ Ein- und Ausgabe

Eingabe von Daten in ein Programm sowie deren Ausgabe ist ein elementarer Aspekt von Programmen.

Wir haben diesen Aspekt bis jetzt verschoben nicht weil er unwichtig wäre, sondern weil sich die entsprechenden Konstrukte in C++ nur im Kontext von Klassen und Operatoren verstehen lassen.

Jedoch werden wir hier nur die Ein- und Ausgabe von Zeichen, insbesondere auf eine Konsole, betrachten. Dies lässt sich leicht auf Dateien erweitern.

Graphische Ein- und Ausgabe werden wir aus Zeitgründen nicht betrachten. Allerdings wurde die Programmierung von graphischen Benutzerschnittstellen durch objektorientierte Programmierung revolutioniert und C++ ist dafür gut geeignet.

Namensbereiche

Zuvor benötigen wir noch das für große Programme wichtige Konstrukt der Namensbereiche welches auch in der Standardbibliothek verwendet wird.

In der globalen Umgebung darf jeder Name höchstens einmal vorkommen. Dabei ist egal, ob es sich um Namen für Variablen, Klassen oder Funktionen handelt.

Damit ergibt sich insbesondere ein Problem, wenn zwei Bibliotheken die gleichen Namen verwenden.

Eine Bibliothek ist eine Sammlung von Klassen und/oder Funktionen, die einem bestimmten Zweck dienen und von einem Programmierer zur Verfügung gestellt werden. Eine Bibliothek enthält keine main-Funktion!

Mittels Namensbereichen lässt sich dieses Problem lösen.

```
namespace A
{
   int n = 1;
   int m = n;
}

namespace B
{
   int n = 2;
   class X {};
}

int main()
{
   A::n = B::n + 3;
   return A::n;
}
```

Mittels **namespace** werden ähnlich einem Block Unterumgebungen innerhalb der globalen Umgebung geschaffen. Allerdings existieren diese Umgebungen gleichzeitig und sind auch nur innerhalb der globalen Umgebung möglich.

Namen innerhalb eines Namensbereiches werden von ausserhalb durch Voranstellen des Namens des Namensraumes und dem :: angesprochen (Qualifizierung).

Innerhalb des Namensraumes ist keine Qualifizierung erforderlich. Mittels **using** kann man sich die Qualifizierung innerhalb eines Blockes sparen.

Namensräume können wieder Namensräume enthalten.

Elemente eines Namensraumes können an verschiedenen Stellen, sogar in verschiedenen Dateien definiert werden. Ein Name darf aber innerhalb eines Namensraumes nur einmal vorkommen.

```
namespace C
{
  double x;
  int f( double x ) { return x; } // eine Funktion
  namespace D
  {
    double x, y; // x verdeckt das x in C
  }
}

namespace C
{ // fuege weitere Namen hinzu
  double y;
}

int main()
{
```

```
    C::x = 0.0; C::y = 1.0; C::D::y = 2.0;
    C::f( 2.0 );
    return 0;
}
```

Ein- und Ausgabe mit Streams

Für die Ein- und Ausgabe stellt C++ eine Reihe von Klassen und globale Variablen in der Standardbibliothek zur Verfügung. Ein- und Ausgabe ist also kein Teil der Sprache C++ selbst.

Alle Variablen, Funktionen und Klassen der C++-Standardbibliothek sind innerhalb des Namensraumes `std` definiert.

Grundlegend für die Ein- und Ausgabe in C++ ist die Idee eines Datenstromes. Dabei unterscheidet man Eingabe- und Ausgabeströme:

- Ein Ausgabestrom ist ein Objekt in welches man Datenelemente hineinsteckt. Diese werden dann an den gewünschten Ort weitergeleitet, etwa den Bildschirm oder eine Datei.

- Ein Eingabestrom ist ein Objekt aus welchem man Datenelemente herausholen kann. Diese kommen von einem gewünschten Ort, etwa der Tastatur oder einer

Datei.

Datenströme werden mittels Klassen realisiert:

- `std::istream` realisiert Eingabeströme.
- `std::ostream` realisiert Ausgabeströme.

Um diese zu verwenden, muss der Header `iostream` eingebunden werden.

Die Ein-/Ausgabe wird mittels überladener Methoden realisiert:

- **operator**>> für die Eingabe.
- **operator**<< für die Ausgabe.

Zur Ein- und Ausgabe auf der Konsole sind globale Variablen vordefiniert:

- `std::cin` vom Typ `std::istream` für die Eingabe.
- `std::cout` vom Typ `std::ostream` für die reguläre Ausgabe eines Programmes.
- `std::cerr` vom Typ `std::ostream` für die Ausgabe von Fehlern.

Damit sind wir bereit für ein Beispiel.

Programm: **(iostreamexample.cc)**

```
#include <iostream>

int main()
{
  int n;
  std::cin >> n;   // d.h. cin.operator>>(n);
  double x;
  std::cin >> x;   // d.h. cin.operator>>(x);
  std::cout << n; // d.h. cout.operator<<(n);
  std::cout << " ";
  std::cout << x;
  std::cout << std::endl; // neue Zeile
  std::cout << n << " " << x << std::endl;
  return 0;
}
```

Die Ausgabe mehrerer Objekte innerhalb einer Anweisung gelingt dadurch, dass die Methode **operator**<< den Stream, also sich selbst, als Ergebnis zurückliefert:

```
std::cout << n << std::endl;
```

ist dasselbe wie

```
( std::cout.operator<<( n ) ).operator<<( std::endl );
```

Die Methoden **operator**>> und **operator**<< sind für alle eingebauten Datentypen wie **int** oder **double** überladen.

Durch Überladen der Funktion

std :: ostream& **operator**<<(std::ostream&, <Typ>);

kann man obige Form der Ausgabe für selbstgeschriebene Klassen ermöglichen.

Als Beispiel betrachten wir eine Ausgabefunktion für die Klasse Rational:

Programm: (RationalOutput.cc)

```
std::ostream& operator<<( std::ostream& s, Rational q )
{
  s << q.numerator() << "/" << q.denominator();
  return s;
}
```

Beachte, dass das Streamargument zurückgegeben wird, um die Hintereinanderausführung zu ermöglichen.

In einer „richtigen“ Version würde man die rationale Zahl als **const** Rational& q übergeben um die Kopie zu sparen. Dies würde allerdings erfordern, die Methoden numerator und denominator als **const** zu deklarieren.

Schließlich können wir damit schreiben:

Programm: **(UseRationalOutput.cc)**

```
#include <iostream>
#include "fcpp.hh"          // fuer print
#include "Rational.hh"
#include "Rational.cc"
#include "RationalOutput.cc"

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

  std::cout << p << " " << q << std::endl;
  std::cout << (p + q*p) * p*p << std::endl;
  return 0;
}
```