Von Java nach C++

I. Von Java nach C++

Inhalt

1. Daten
   - Datentypen
   - Ströme
   - Referenztypen
   - Speicherklassen
   - Ausnahmebehandlung
2. Funktionen
   - Werte- und Referenzsemantik
   - Operatorüberladung
   - Überladungsauflösung
3. Klassen
   - Klassen in C++
   - Operatormethoden
   - benutzerdefinierte Typumwandlungen
   - Funktionsobjekte und Prädikate
4. sequentielle Container

I. Von Java nach C++

Quellen

U. Breymann
Der C++ Programmierer
Hanser

I. Hello World

In C++ braucht man keine Klassen!

☞ ERSTES_PROGRAMM

Java

```
public class HelloWorld {
 // Jedes Java-Programm besitzt eine Klasse
 static public void main(String[] args) {
  System.out.println("Hello World!");
 }
}
```

C++

```
#include <iostream> // Bibliothek fuer Ein-/Ausgabefunktionen
using namespace std; // Namensraum fuer C++ Bibliotheksfunktionen

int main() {
 cout << "Hello World!" << endl;
 return 0;
}
```

I. Programmieren ohne Klassen

☞ ADDIERE

- In einem C++ Programm gibt es genau eine Funktion **int** main(), die bei Programmstart aufgerufen wird
- Methoden, die keiner Klasse zugeordnet sind, heißen Funktionen.
- Beim Aufruf einer Funktion muss deren Signatur (Funktionsname+Parameterliste) bekannt sein. Dies erfolgt entweder durch Implementation der Funktion _vor_ dem Aufruf oder durch die Funktionsdeklaration am Anfang

```
int addiere(int, int); // Funktionsdeklaration
                       // (kann entfallen, wenn die Funktion
                       // direkt hier implementiert wird)
int main() {
 int a {4}, b {5}, c;
 c = addiere(a,b);     // Die Werte von a,b werden als Kopie
                       // an die Funktion uebergeben
 return 0;             // Rueckgabewert bei normaler
}                      // Programmausfuehrung

int addiere(int a, int b) { // Funktionsimplementation
 return a+b;
}
```

1. Daten
   - Datentypen
   - Ströme
   - Referenztypen
   - Speicherklassen
   - Ausnahmebehandlung
2. Funktionen
   - Werte- und Referenzsemantik
   - Operatorüberladung
   - Überladungsauflösung
3. Klassen
   - Klassen in C++
   - Operatormethoden
   - benutzerdefinierte Typumwandlungen
   - Funktionsobjekte und Prädikate
4. sequentielle Container

I.1a Von Java nach C++

Inhalt

1. Daten
   - Datentypen
   - Ströme
   - Referenztypen
   - Speicherklassen
   - Ausnahmebehandlung
2. Funktionen
   - Werte- und Referenzsemantik
   - Operatorüberladung
   - Überladungsauflösung
3. Klassen
   - Klassen in C++
   - Operatormethoden
   - benutzerdefinierte Typumwandlungen
   - Funktionsobjekte und Prädikate
4. sequentielle Container

I.1a Vergleich der elementaren Datentypen

Bez. Java	Größe	Min	Max	Bez. C++	Größe	Min	Max
boolean	1 bit	–	–	bool	8 Bit	0 ($false$)	1 ($true$)
char	16 Bit	Unicode 0	Unicode $2^{16}+1$	char	8 Bit	-128	$+127$
-	-	–	–	unsigned char	8 Bit	0	255
byte	8 Bit	-128	127	-	-	–	–
short	16 Bit	-2^{15}	$2^{15}-1$	short	16 Bit	-2^{15}	$2^{15}-1$
-	-	–	–	unsigned short	16 Bit	0	$2^{16}-1$
int	32 Bit	-2^{31}	$2^{31}-1$	int	32 Bit	-2^{31}	$2^{31}-1$
-	-	–	–	unsigned int	32	0	$2^{32}-1$
long	64 Bit	-2^{63}	$2^{63}-1$	long	64 Bit	-2^{63}	$2^{63}-1$
-	-	–	–	unsigned long	64 Bit	0	$2^{64}-1$
float	32 Bit	$1{,}4e-45$	$34e+38$	float	32 Bit	$1{,}1e-38$	$3{,}4e+38$
double	64 Bit	$4{,}9e-324$	$1{,}7e+308$	double	64 Bit	$2{,}2e-308$	$1{,}7e+308$
-	-	–	–	long double	128 Bit	$3{,}3e-4932$	$1{,}1e+4932$

Größen sind abhängig von der Rechnerarchitektur

I.1a logische Werte

Typ	Speicherbedarf	Wertebereich
`bool`	1 Byte	0 oder 1

Manipulatoren für Wahrheitswerte	
`(no)boolalpha`	Ausgabe von `true`/`false` anstatt 0/1

```
#include<iostream>
#include<iomanip>                    // Manipulatoren
using namespace std;

int main() {
 bool wahrheitswert {true};// Initialisierung
 cout << wahrheitswert << endl; // 1
 cin >> wahrheitswert;          // Eingabeaufforderung
 cout << boolalpha << wahrheitswert << endl;
                                // true, falls Eingabe != 0
 return 0;
}
```

I.1a Zeichen und Zeichenketten

char-Werte

...werden in einfache Hochkommata eingeschlossen: 'x', 'F', '$', '0', '\0'

'\n': Neue Zeile, '\r': Zeilenanfang, '\t': Tabulator
'\'': Hochkomma, '\"': Anführungszeichen, '\\': Backslash

Zeichenkettenkonstanten (string-literale)

...werden in Anführungszeichen eingeschlossen: "Hello World\n"

「Zeichenkettenkonstanten sind keine String-Objekte!」

Typ	Speicherbedarf	Bemerkung
`char`	1 Byte	als ASCII-Code repräsentiert
`signed char`	1 Byte	-128..127
`unsigned char`	1 Byte	0..255

I.1a Zeichen und Zeichenketten

Ein-/Ausgabe

☞ ZEICHEN_IO

Manipulatoren und spezielle Zeichen	
`(no)skipws`	Whitespace überspringen
`left, right`	Ausrichtung
`setfill(char)`	Füllzeichen angeben
`setw(int)`	Feldbreite angeben
`endl`	'\n' einfügen
`ends`	'\0' einfügen
`'\r'`	Zeilenrücklauf
`flush`	Ausgabepuffer leeren

```
char buchstabe;
cin >> buchstabe;
cout << ( (buchstabe>='A') && (buchstabe<='Z') );
  // true, falls Grossbuchstabe

cout << setw(10) << setfill('.') << "Hallo" << endl;
  // .....Hallo
cout << left << setw(10) << setfill('_') << "Hallo" << endl;
  // Hallo_____
```

I.1a ganze Zahlen

Typ	Speicherbedarf
`(unsigned) short`	2 Byte
`(unsigned) int`	4 Byte
`(unsigned) long`	4 oder 8 Byte
`(unsigned) long long`	4 oder 8

Die Bibliothek climits zeigt die tatsächlichen Größen und Zahlgrenzen an.

Integerkonstanten

Dezimalzahlen	: -23, 186, 186u, 189l, 4243434uL, 0
Oktalzahlen	: 04, 047u; nicht 09
Hexadezimalzahlen	: -0xF0, 0XaaL

I.1a ganze Zahlen

Ein-/Ausgabe

☞ GANZE_ZAHLEN

Manipulatoren für ganze Zahlen	
`dec, oct, hex`	Zahlenformat
`(no)showbase`	Ausgabe der Zahlenbasis
`(no)showpos`	"+" mit ausgeben
`(no)uppercase`	Großbuchstaben für Hexzeichen
`setbase(int)`	Zahlenbasis festlegen
`left, right, internal`	Ausrichtung (internal: VZ links, Zahlen rechts)
`setw(int)`	Feldbreite angeben
`setfill(char)`	Füllzeichen angeben

```
int hexZahl;
cin >> hex >> hexZahl;
    // ^-- naechste Eingabe wird hexadezimal interpretiert!
    //     moegliche Ziffern: 0-9,A-F bzw. a-f
    //     Beispieleingabe: c
cout << hexZahl
     << " (hex: " << hex << hexZahl << ")" << endl;
    // 12 (hex: c)
```

I.1a ganze Zahlen

Ein-/Ausgabe

☞ INT_IO

Manipulatoren für ganze Zahlen	
dec, oct, hex	Zahlenformat
(no)showbase	Ausgabe der Zahlenbasis
(no)showpos	"+" mit ausgeben
(no)uppercase	Großbuchstaben für Hexzeichen
setbase(int)	Zahlenbasis festlegen
left, right, internal	Ausrichtung (internal: VZ links, Zahlen rechts)
setw(int)	Feldbreite angeben
setfill(char)	Füllzeichen angeben

```
int dezZahl {12}; // Initialisierung mit geschweifter Klammer
cout << showbase; // Zahlenbasis bei Ausgabe anzeigen
cout << "oct: " << oct<< setw(4)<< dezZahl<< ", ";
cout << "hex: " << hex<< uppercase<< setw(4)<< dezZahl<< ", ";
cout << "dec: " << dec<< setw(4)<< setfill('_')<< showpos
                << dezZahl<< endl;
cout << noshowbase;  // auf Standard zuruecksetzen
    // oct:  014, hex:  0XC, dec:  +12
```

I.1a gebrochen rationale Zahlen

Typ	Speicherbedarf (implementationsabhängig)
`float`	4 Byte
`double`	8 Byte
`long double`	12 Byte

Die Bibliothek limits zeigt die tatsächlichen Größen und Zahlgrenzen an.

spezifische Manipulatoren für gebrochene Zahlen	
`(no)showpoint`	angehängten Dezimalpunkt/Nullen anzeigen
`fixed, scientific`	Fest-/Gleitpunktdarstellung
`setprecision(int)`	Genauigkeit angeben
`left, right, internal`	Ausrichtung (internal: VZ links, Zahlen rechts)
`setw(int)`	Feldbreite angeben
`setfill(char)`	Füllzeichen angeben

Die Bibliothek cmath enthält elementare mathematische Funktionen für alle Zahltypen.

implizite Anpassungen

```
double a= -3.7 + 5; // 5 --> 5.0 vor Addition
double x=5+8; // 13 --> 13.0 vor Zuweisung
cout << (-3.7 || 0); // -3.7 --> true , 0 --> false
```

Typkonvertierungen

1. Promotion (werterhaltende Konversion)
   - ganzzahlig: `bool`, `char`, `short` $\longrightarrow$ `int`
   - Fließkommatypen: float$\longrightarrow$double
2. Standardkonvertierungen (i.allg. nicht werterhaltend)
   - `int`$\longleftrightarrow$`long`
   - `int`$\longleftrightarrow$`float/double`

I.1a Operatorprioritäten

Prio	Operator	Ass.	Bemerkung
0	::		Bereichsauflösung
1	++ −− () [] −> . x_cast<Typ>() x=static, const, reinterpret, dynamic	l	postfix, unär
2	++ −− ! ~ + − * & sizeof() new delete	r	präfix, unär
3	.* −>*	l	Elementselektion
4	* / \%	l	binär
5	+ −	l	arithmetisch
6	<< >>	l	shift
7	< <= > >=	l	relational
8	== !=	l	
9	&	l	
10	\^	l	bit
11	|	l	
12	\&\&	l	logisch
13	||	l	
14	?	r	ternär
15	= += −= *= /= %= &= ^= |= <<= >>=	r	Zuweisung
16	throw		Ausnahme werfen
17	,	l	Sequenz

I.1a Strings

Java

```java
public class Klasse {
 public static void main(String[] args) {
  String str = "String";
  // inhaltlicher Vergleich + Elementzugriff; Ausgabe: r
  if(str.equals("String")) System.out.println(str.charAt(2));
  // Konkatenation + Zuweisung
  str+= " in Java";
  System.out.println(str.length() + " " + str);
 } // Ausgabe: 14 String in Java
}
```

C++

```cpp
#include <iostream>
#include <string>          //muss eingebunden werden
using namespace std;
int main(){
 string str  {"String"}; // Initialisierung
 // inhaltlicher Vergleich + Elementzugriff; Ausgabe: r
 if(str == "String") cout << str[2] << endl;
 // Konkatenation + Zuweisung
 str += " in c++";
 cout << str.size() << " " << str << endl;
 return 0;
}  // Ausgabe: 13 String in C++
```

I.1a Strings

Operationen	
Deklaration und Initialisierung	`string s {"Hallo"};` `string t="du da";`
Verkettung	`string u=s+t;`
Vergleichsoperatoren	`==, !=, <, >, <=, >=`

Manipulationen	
Leeren	`s.clear()`
t ab Position pos einfügen	`s.insert(pos,t)`
anz Zeichen ab pos löschen	`s.erase(pos,anz)`
anz Zeichen ab pos durch t ersetzen	`s.replace(pos, anz, t)`

Abfragen	
Länge	`s.length()`
Kürzen/Verlängern (auffüllen mit 0)	`s.resize(len)`
anz Zeichen ab pos zurückgeben	`s.substr(pos,anz)`
Position von t in s	`s.find(t, [start_pos])`
—"— Suche von rechts nach links	`s.rfind(t, [start_pos])`

Vom Eingabestrom in einen String s einlesen bis zum '\n'

```
getline(cin, s, '\n');
```

I.1a Vektoren

Vektoren sind Typschablonen: Der Elementtyp wird in spitzen Klammern angegeben und gehört mit zum Typnamen! ⌜primitive Elementtypen sind erlaubt!⌟

```
vector<double> vec {1.5, -2.3};  // Vektor mit 2 double-Werten initialisiert
vector<double> vec_kopie(vec);   // Inhaltskopie von vec
vector<int>    quadrate(10);     // Vektor mit Platz fuer 10 int-Werte

// Elemente anfuegen/entfernen
vec.push_back(3.7);              // 1.5, -2.3, 3.7
vec.push_back(5.4);              // 1.5, -2.3, 3.7, 5.4
vec.pop_back();                  // 1.5, -2.3, 3.7

// Elementzugriff
cout << vec.front() << " ";      // 1.5
cout << vec.back() << endl;      // 3.7

// wahlfreier Zugriff
for (int i=0; i<quadrate.size(); i++) quadrate[i] = i*i;
// 0 1 4 9 16 25 36, 49 64 81
for (int i=10; i<15; i++) quadrate.push_back(i*i);
// 0 1 4 9 16 25 36 49 64 81 100 121 144 169 196

// Globalzugriff
quadrate.resize(10);             // schneidet Vektor ab
// 0 1 4 9 16 25 36 49 64 81
quadrate.resize(15);             // verlaengert Vektor
// 0 1 4 9 16 25 36 49 64 81 0 0 0 0 0

vec_kopie = vec; vec_kopie[2]=3; // Zuweisung kopiert Inhalt!
cout << boolalpha << (vec_kopie<vec) << endl;
// lexikographischer Vergleich: true
```

I.1a Vektoren

logisches Speicherabbild

Konstruktion (Elementtyp=T)	「T steht für einen existierenden Datentypnamen」
leerer Vektor	`vector<T> v;`
Vektor mit n Elementen	`vector<T> v(n);`
Vektor mit n Elementen und Vorgabewert `x`	`vector<T> v(n,x);`
Vector `v` anlegen und initialisieren	`vector<T> v {t1, t2, t3};`
Kopie von `v` anlegen	`vector<T> w(v);`

Anfügen/Entfernen	
`x` am Ende anfuegen	`v.push_back(x);`
Element am Ende entfernen	`v.pop_back();`

Elementzugriff	
Element am Anfang	`v.front();`
Element am Ende	`v.back();`
i-tes Element	`v[i];`
i-tes Element mit Bereichsüberprüfung	`v.at(i);`

Globalzugriff	
Anzahl Elemente	`v.size();`
Vektor abschneiden/auffüllen	`v.resize(n);`
Elemente löschen	`v.clear();`
Vektor leer?	`v.empty();`
Zuweisung	`=`
lexikographische Vergleiche	`==, !=, <, <=, >, >=`

I.1a Aufzählungen

☞ ENUM

```
// Neuer Aufzaehlungstyp Ampel
enum Ampel { rot, gelb, gruen };

// eine Ampelvariable
Ampel ampel_ecke_lessingstrasse=rot;

//
if (ampel_ecke_lessingstrasse==rot) { /* warten */ }

// Ampel weiterschalten
ampel_ecke_lessingstrasse = gruen;
int interneNr {ampel_ecke_lessingstrasse};
cout << interneNr << endl; // 2
cout << gruen << endl;     // 2
```

I.1a Strukturen

Zusammenfassen von Daten unterschiedlichen Typs

☞ STRUCT

```
struct Person {  // Person ist jetzt ein neuer Typname
 string name;
 int    alter;
 string adresse;
};          //! abschliessendes Semikolon nicht vergessen
Person person;   // Vereinbarung einer struct-Variable

// Komponentenzugriff
person.name = "Max Muster";
person.alter = 30;
person.adresse = "Hauptstrasse 15";

// Zuweisung kopiert Inhalte!
Person neu {"Hans Wurst", 25, "Bierstrasse 13"}
neu=person;      // ueberschreibt die Komponeten von neu
                 // durch die Komponenten von person
```

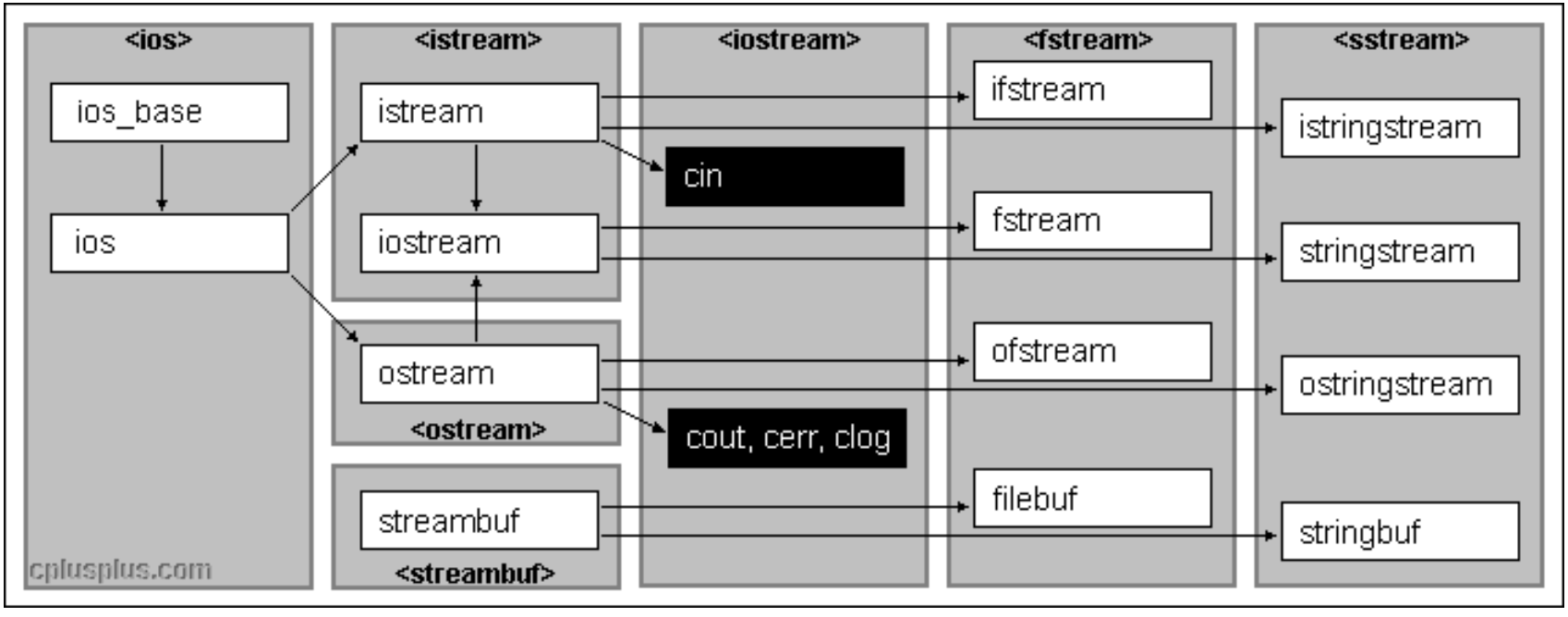
I.1b Von Java nach C++

Inhalt

1. Daten
   - Datentypen
   - Ströme
   - Referenztypen
   - Speicherklassen
   - Ausnahmebehandlung
2. Funktionen
   - Werte- und Referenzsemantik
   - Operatorüberladung
   - Überladungsauflösung
3. Klassen
   - Klassen in C++
   - Operatormethoden
   - benutzerdefinierte Typumwandlungen
   - Funktionsobjekte und Prädikate
4. sequentielle Container

I.1b Das Stromkonzept

In C++ ist die Ein-/Ausgabe über sogenannte *streams* realisiert. Die iostream-Bibliothek stellt Klassen, sowie Eingabe-/Ausgabemethoden bzw. -funktionen für jeden eingebauten Datentyp zur Verfügung.
für benutzerdefinierte Datentypen ist es möglich, eigene Ein-/Ausgabefunktionen hinzuzufügen.



I.1b Das Stromkonzept

Die Standardausgabe

- Die Standardausgabe erfolgt über das Stream-Objekt cout, das eine formatierte Ausgabe der C++ Datentypen auf den Bildschirm ermöglicht.
- cout ist ein globales Objekt der Klasse ostream und wird per **#include** <iostream> in eigene Programme eingebunden.
- In Verbindung mit cout ist << ein Operator, der Werte von Variablen oder Konstanten ausgibt und eine Referenz auf den Strom zurückliefert.
- Auf diese Weise sind in der Abarbeitung von links nach rechts Verkettungen möglich: cout << "Werte:␣" << i << i*i << endl;
- Mithilfe von Manipulatoren (**#include**<iomanip>) kann die Ausgabe formatiert werden.
- Verwendet man anstelle von cout den Fehlerstrom cerr wird die Ausgabe in die Standard-Fehlerausgabe (stderr) umgeleitet (typischerweise ebenfalls die Konsole)

I.1b Das Stromkonzept

Die Standardeingabe

- Die Standardeingabe erfolgt über das Stream-Objekt cin, das ein Einlesen der C++ Datentypen von der Tastatur ermöglicht.
- cin ist ein globales Objekt der Klasse istream und wird per **#include**<iostream> in eigene Programme eingebunden.
- In Verbindung mit cin ist >> ein Operator, der Werte von der Tastatur in Variablen einschreibt und eine Referenz auf den Strom zurückliefert.
- Auf diese Weise sind in der Abarbeitung von links nach rechts Verkettungen möglich: cin >> intVar >> doubleVar;
- Mithilfe von Manipulatoren (**#include**<iomanip>) kann die Eingabe manipuliert werden.

I.1b Das Stromkonzept

String-Ströme

☞ STRINGSTREAM

stringstream-Objekte (**#include**<sstream>) ermöglichen die Manipulation von string-Objekten so, als wären es Ströme. Damit lassen sich komfortabel Datentypen konvertieren

```
string str= "12.345 45";
float doubleVar = 0; int intVar = 0;

//Konvertierung eines strings nach double und int
stringstream strom(str); // Stromobjekt initialisieren
if(!(strom >> doubleVar >> intVar)) // Probieren der Zuweisung
 cout << "error: could not convert" << endl; // nicht geklappt
else { // hat geklappt - Ausgabe: 12.345 - 45
 cout << doubleVar << " - " << intVar << endl;
}

//Konvertierung von double und int in einen string
 stringstream neuerStrom;
 neuerStrom << doubleVar << " : " << intVar; // Konvertierung
 cout << neuerStrom.str() << endl; // Umwandlung in string-Wert
                                   // Ausgabe: 12.345 : 45
```

I.1b Das Stromkonzept

Datei-Ströme

☞ FSTREAM

fstream-Objekte (**#include**<fstream>) ermöglichen die Manipulation von Dateien so, als wären es Ströme.

```
//Eingbestrom anlegen und mit Datei "eingabe.txt" verbinden
ifstream ein("eingabe.txt");
//Ausgabestrom anlegen und mit Datei "ausgabe.txt" verbinden
ofstream aus("ausgabe.txt");
if(!ein || !aus){ //Pruefen ob die Datein geoeffnet wurden
 cout << "Fehler beim Oeffnen der Dateien" << endl; exit(1);
}
//Variante - Dieselbe Datei fuer Ein- und Ausgabe:
//fstream strom("datei.txt");

// Datei kopieren und dabei Werte verdreifachen
double wert;
while(!(ein >> wert).eof()) //Einlesen bis End of File
 aus << "Wert=" << setw(6) << 3*wert << endl;
ein.close(); aus.close();
```

```
  -2.3 5.7                                  Wert=  -6.9
                                            Wert=  17.1
  4.1                                       Wert=  12.3
                                            Wert=    27
  9                                         Wert=    30

  10.0
```

I.1b Das Stromkonzept

Abfragen von Stromzuständen

☞ SICHEREEINGABE

`strom.good()`	letzte Operation war fehlerfrei
`strom.fail()`	letzte Operation war fehlerhaft
`strom.bad()`	wie fail(), aber strom jetzt kaputt
`strom.eof()`	Dateiende ist erreicht worden
`strom.clear()`	macht den Strom wieder aufnahmebereit

```
int zahl;                       // einzulesende Werte
int ungueltigeZeichen {0};    // Zeichen, die keine Ziffern sind
char falschesZeichen  {'␣'}; // nimmt die Falscheingaben auf

cout << "Bitte␣eine␣Zeile,␣die␣Zahlen␣enthaelt,␣eingeben" << endl;
do { // Eingabe wird Zeichen fuer Zeichen durchgegangen
 cin >> noskipws >> zahl;// Einlesen der Zahl
 //      ^-- Whitespace-Charaktere werden als als normale Zeichen eingelesen
 if (cin.fail()) {         // Abfrage des Eingabestrom-Zustands:
  cin.clear();             // macht den Strom wieder aufnahmebereit
  cin >> noskipws >> falschesZeichen; // Aufnahme des ungueltigen Zeichens
                                      // in den dafuer vorgesehenen Datentyp
  ungueltigeZeichen++;    // ungueltige Zeichen zaehlen
 } else cout << "akzeptierter␣Wert:␣" << zahl << endl;
} while(falschesZeichen != '\n');  // Pruefung, ob die Schleifen erneut
                                   // durchlaufen werden soll
ungueltigeZeichen--;       // Das letzte '\n' war nicht ungueltig
```

I.1c Von Java nach C++

Inhalt

1. Daten
   - Datentypen
   - Ströme
   - Referenztypen
   - Speicherklassen
   - Ausnahmebehandlung
2. Funktionen
   - Werte- und Referenzsemantik
   - Operatorüberladung
   - Überladungsauflösung
3. Klassen
   - Klassen in C++
   - Operatormethoden
   - benutzerdefinierte Typumwandlungen
   - Funktionsobjekte und Prädikate
4. sequentielle Container

I.1c Referenztypen

Eine Referenz in C++ ist eine unlösbare Verknüpfung / ein Alias für eine bestehende Variable.

Referenztypen entstehen durch Anhängen von & an den Typnamen

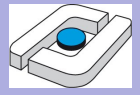
```
int  i {1};
int& i_ref {i};  // i_ref ist Referenz auf i
int  j {i};      // Initialisierung von j mit dem Wert von i

j     = i_ref;   // Dasselbe wie j = i;
i_ref = 2;       // Dasselbe wie i = 2;
j     = 3;
i     = j;
cout << i << " " << i_ref; // Ausgabe: 3 3
```

L- und R-Werte: konstante Referenzen

Die Verknüpfungsziele für Referenzen nennt man auch L-Werte. Sie besitzen einen Namen und eine damit verknüpfte Speicherdresse. Sie können auf der linken Seite einer Zuweisung stehen.

Im Gegensatz dazu haben Konstanten und temporäre Ausdrücke (z.B. a+b) nicht notwendig einen Namen und können nur auf der rechten Seite einer Zuweisung stehen (R-Werte).

const T& t_ref {wert}; vereinbart eine Referenz auf den L- oder R-Wert wert und erlaubt nur lesenden Zugriff.



I.1c Referenztypen

Vergleich von Referenztypen in Java und C++

Java	C++
Referenzen zeigen auf Objekte oder Arrays. Auf die Standarddatentypen kann nicht referenziert werden	Referenzen können für jeden Datentyp angelegt werden, dies wird durch Anhägen von & kenntlich gemacht
Referenzen können auf NULL verweisen	Referenzen können nur an existierende Variablen gebunden werden
Referenzen können jederzeit auf ein anderes Ziel verweisen	Eine einmal initialisierte Referenz ist unlösbar mit ihrem Ziel verbunden
Änderungen an der Referenz wirken sich auf das Original aus und umgekehrt	
Es können beliebig viele Referenzen für ein Original angelegt werden	
== vergleicht Identität, equals vergleicht Inhalte	== vergleicht Inhalte
= weist Speicheradressen zu	= weist Inhalte zu

I.1d Von Java nach C++

Inhalt

1. Daten
   - Datentypen
   - Ströme
   - Referenztypen
   - Speicherklassen
   - Ausnahmebehandlung
2. Funktionen
   - Werte- und Referenzsemantik
   - Operatorüberladung
   - Überladungsauflösung
3. Klassen
   - Klassen in C++
   - Operatormethoden
   - benutzerdefinierte Typumwandlungen
   - Funktionsobjekte und Prädikate
4. sequentielle Container

I.1d Speicherklassen

Beim Start eines Programms wird Speicherplatz für folgende Segmente bereit gestellt

- Code-Segment: Enthält das Programm, sowie Konstanten
- Stack-Segment: Enthält automatische Variable
- Heap-Segment: Enthält dynamische Variable
- Daten-Segment: Enthält globale und statische Variable

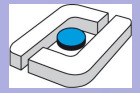
- Code- und Datensegment — statisch
  werden statisch zum Programmstart erzeugt
- Stack und Heap — dynamisch
  werden dynamisch zur Laufzeit belegt und wieder freigegeben.
  Dabei wird der Stack vom System verwaltet, während der Heap vom Benutzer verwaltet wird

I.1d Stack — automatische Variablen

- Auf dem Stack werden lokale Variablen einer Funktion oder Verbundanweisung abgelegt
- Sie besitzen nach der Deklaration einen undefinierten Wert
- und werden wieder vom Stack gelöscht, sobald der umgebende Block (Funktion, Verbundanweisung) verlassen wird ("automatisch")
- Bei jedem Aufruf einer Funktion wird auf dem Stack Speicher für die Parameter und die lokalen Variablen belegt
- Die Speicherung auf dem Stack geschieht nach dem LIFO (Last In First Out) Prinzip

I.1d Heap — Dynamische Variablen

- Dynamische Variablen werden durch den Benutzer auf dem Heap eingerichtet und verwaltet
- Die Lebensdauer bestimmt der Benutzer durch explizite (Bedarfs-)Anforderung und Freigabe von Speicherplatz ("dynamisch")
- Sie besitzen keinen eigenen Variablennamen, sondern werden über zugeordnete Zeiger angesprochen
- Sie werden im späteren Verlauf der Vorlesung thematisiert



I.1d Variablen im Datensegment

lokale statische Variablen

Lokale statische Variablen

- werden am Beginn eines Funktionsrumpfes oder einer Verbundanweisung deklariert und durch das Schlüsselwort **static** gekennzeichnet
- werden im (statischen) Datensegment abgelegt und besitzen daher eine feste Speicheradresse
- daher bleibt ein einmal zugewiesener Wert während der gesamten Programmlaufzeit erhalten, auch wenn der zugehörige Programmblock bereits verlassen wurde
- auf sie kann nur innerhalb des zugehörigen Programmblocks zugegriffen werden (lokal statisch).
- Statische Variablen werden beim Programmstart einmalig(!) automatisch mit 0 initialisiert falls keine Initialisierung vom Benutzer vorgenommen wird

I.1d Variablen im Datensegment

lokale statische Variablen

☞ SPEICHERKLASSEN_LOKAL_STATISCH

```
#include <iostream>
using namespace std;
int test(){
 static int i=1; // wird beim Programmstart
                 // EINMALIG initialisiert
 return i++;
} // i ueberlebt das Ende der Verbundanweisung

int main(){
 cout << test() << endl; // 1
 cout << test() << endl; // 2
 cout << test() << endl; // 3
 //cout << i << endl; // i hier unbekannt
 return 0;
}
```

I.1d Variablen im Datensegment

Globale Variablen

- werden außerhalb von Funktionen deklariert und im Datensegment abgelegt, so dass sie wie statische Variablen während der gesamten Laufzeit des Programms existieren
- Globale Variablen werden ebenfalls (einmalig) mit 0 initialisiert, falls keine Initialisierung vom Benutzer vorgenommen wird
- modulweite Sichtbarkeit:
  Deklaration durch das Schlüsselwort **static** außerhalb von Funktionen macht die Variable sichtbar für das gesamte Modul (=Datei/Übersetzungseinheit)
- programmweite Sichtbarkeit:
  Deklaration ohne Schlüsselwort außerhalb von Funktionen macht die Variable programmweit global. Anderen Modulen wird die externe Variablendefinition durch das Schlüsselwort **extern** bekannt gemacht. Es führt lediglich zu einer Deklaration, aber keiner Definition der Variablen.

I.1d Variablen im Datensegment

globale Variablen

☞ SPEICHERKLASSEN_GLOBAL_TEST1

test.cpp

```
extern int global;       // extern definierte Variable
static int statisch=1; // modulweit sichtbare Variable
void test(int n){

 global=statisch;       // beide Variablen hier bekannt
 statisch=n;
}
```

main.cpp

```
#include <iostream>
using namespace std;
void test(int);
int global=10;           // programmweit globale Variable
int main(){
            cout << global << endl; // 10
 test(3); cout << global << endl; // 1
 test(7); cout << global << endl; // 3
 test(0); cout << global << endl; // 7
 return 0;
```

I.1d Variablen im Datensegment

globale Variablen

☞ SPEICHERKLASSEN_GLOBAL_TEST2

test.cpp

```
extern int global;       // extern definierte Variable
static int statisch=1; // modulweit sichtbare Variable
void test(int n){
 int statisch=1;        // statisch jetzt lokal und
 global=statisch;       // ueberdeckt den aeusseren Wert
 statisch=n;
}
```

main.cpp

```
#include <iostream>
using namespace std;
void test(int);
int global=10;          // programmweit globale Variable
int main(){
           cout << global << endl; // 10
 test(3); cout << global << endl; // 1
 test(7); cout << global << endl; // 1
 test(0); cout << global << endl; // 1
 return 0;
```

I.1d Zusammenfassung

Speicher-klasse	**Schlüssel-wort**	**Segement**	**Initiali-sierung**	**Gültig-keit**	**Lebens-dauer**
automatisch	-	Stack	nein	Block	Block
lokal,statisch	`static`	Daten	ja	Block	Programm
global	`static`	Daten	ja	Modul	Programm
	-			Programm	
	`extern`		-		
dynamisch	-	Heap	nein	-	Benutzer

I.1d Ergänzung

Modifizierer, Qualifizierer, Spezifizierer

Typ-Modifizierer — Fügt dem Datentyp eine Modifikation hinzu	
`signed`	vorzeichenbehaftet
`unsigned`	vorzeichenlos
`short`	kürzer
`long`	länger

Typ-Qualifizierer — Fügt den Variablen neue Eigenschaften hinzu	
`const`	Variable kann nach der Initialisierung nicht mehr verändert werden
`volatile`	Wert der Variablen kann durch Prozesse außerhalb des Programms verändert worden sein
`mutable`	Wert der Variablen kann durch ☞ KONSTANTE KLASSENMETHODEN verändert werden

Speicherklassen-Spezifizierer — Lebensdauer und Sichtbarkeit	
`auto`	Lebensdauer und Sichtbarkeit: aktueller Block
`static`	Lebensdauer: Programm; Sichtbarkeit: Block der Definition
`extern`	Lebensdauer und Sichtbarkeit: Programm

I.1e Von Java nach C++

Inhalt

1. Daten
   - Datentypen
   - Ströme
   - Referenztypen
   - Speicherklassen
   - Ausnahmebehandlung
2. Funktionen
   - Werte- und Referenzsemantik
   - Operatorüberladung
   - Überladungsauflösung
3. Klassen
   - Klassen in C++
   - Operatormethoden
   - benutzerdefinierte Typumwandlungen
   - Funktionsobjekte und Prädikate
4. sequentielle Container

I.1e Ausnahmebehandlung

Die Ausnahmebehandlung in C++ entspricht weitgehend der in Java. Sie dient der Behandlung von durch äußere Einflüsse verursachten Fehlern im Programmablauf

Stationen der Ausnahmebehandlung

- Fehlererkennung → "Werfen" eines Ausnahmeobjektes (**throw**)
- Code für Normalfall → Markiert durch **try**-Block
- Code für Ausnahmefall → "Fangen" von Ausnahmen im **catch**-Block

Beim Werfen einer Ausnahme wird der entsprechende Programmblock sofort verlassen, der zugehörige Programmstack abgewickelt und in die nächst höhere Programmebene gesprungen.
Dieser Vorgang wiederholt sich, bis:

- ein **try**-Block erreicht wird, der eine passende Ausnahme bereitstellt (d.h. **catch** fängt den passenden Ausnahmetyp)
- bis zur obersten Programmebene, wo das Programm verlassen wird

I.1e Ausnahmebehandlung

try-catch Mechanismus


```
double kehrwert(int n) {
  if (n==0) throw string("Division durch Null");
  return 1./n;
}
int main() {
  int zahl;
  double inverse;
  do {
    cout << "Bitte eine ganze Zahl eingeben (-1=Ende): ";
    cin >> zahl;
    try {
      inverse=kehrwert(zahl);
      cout << inverse << endl;
    }
    catch(const string& s) {
      cout << s << endl << "Bitte neuen Wert eingeben:";
    }
  } while(zahl!=-1);
  return 0;
}
```

Methode kehrwert wirft ggf. ein Ausnahmeobjekt vom Typ string

I.1e Ausnahmebehandlung

try-catch Mechanismus


```
double kehrwert(int n) {
  if (n==0) throw string("Division durch Null");
  return 1./n;
}
int main() {
  int zahl;
  double inverse;
  do {
    cout << "Bitte eine ganze Zahl eingeben (-1=Ende): ";
    cin >> zahl;
    try {
      inverse=kehrwert(zahl);
      cout << inverse << endl;
    }
    catch(const string& s) {
      cout << s << endl << "Bitte neuen Wert eingeben:";
    }
  } while(zahl!=-1);
  return 0;
}
```

Schlüsselwort **try** markiert, dass der folgende Block Ausnahmen werfen kann,...

```
double kehrwert(int n) {
  if (n==0) throw string("Division durch Null");
  return 1./n;
}
int main() {
  int zahl;
  double inverse;
  do {
    cout << "Bitte eine ganze Zahl eingeben (-1=Ende): ";
    cin >> zahl;
    try {
      inverse=kehrwert(zahl);
      cout << inverse << endl;
    }
    catch(const string& s) {
      cout << s << endl << "Bitte neuen Wert eingeben:";
    }
  } while(zahl!=-1);
  return 0;
}
```

...die im nächsten, zur Ausnahme passenden **catch**-Block (der in den Klammern stehende Typ muss dem Ausnahmetyp entsprechen) gefangen und behandelt oder weitergeworfen werden muss.

I.1e Ausnahmebehandlung

try-catch Mechanismus


```
double kehrwert(int n) {
  if (n==0) throw string("Division durch Null");
  return 1./n;
}
int main() {
  int zahl;
  double inverse;
  do {
    cout << "Bitte eine ganze Zahl eingeben (-1=Ende): ";
    cin >> zahl;
    try {
      inverse=kehrwert(zahl);
      cout << inverse << endl;
    }
    catch(const string& s) {
      cout << s << endl << "Bitte neuen Wert eingeben:";
    }
  } while(zahl!=-1);
  return 0;
}
```

Der Benutzer gibt eine Zahl ein, dessen Inverse berechnet werden soll. Anschließend wird versucht, den Kehrwert hierzu zu berechnen.

I.1e Ausnahmebehandlung

try-catch Mechanismus


```
double kehrwert(int n) {
  if (n==0) throw string("Division durch Null");
  return 1./n;
}
int main() {
  int zahl;
  double inverse;
  do {
    cout << "Bitte eine ganze Zahl eingeben (-1=Ende): ";
    cin >> zahl;
    try {
      inverse=kehrwert(zahl);
      cout << inverse << endl;
    }
    catch(const string& s) {
      cout << s << endl << "Bitte neuen Wert eingeben:";
    }
  } while(zahl!=-1);
  return 0;
}
```

Bei Aufruf der Funktion kehrwert wird eine Ausnahme vom Typ string geworfen, wenn die übergebene Zahl mit 0 übereinstimmt.

I.1e Ausnahmebehandlung

try-catch Mechanismus


```
double kehrwert(int n) {
  if (n==0) throw string("Division durch Null");
  return 1./n;
}
int main() {
  int zahl;
  double inverse;
  do {
    cout << "Bitte eine ganze Zahl eingeben (-1=Ende): ";
    cin >> zahl;
    try {
      inverse=kehrwert(zahl);
      cout << inverse << endl;
    }
    catch(const string& s) {
      cout << s << endl << "Bitte neuen Wert eingeben:";
    }
  } while(zahl!=-1);
  return 0;
}
```

In diesem Fall wird die Ausführung des Codes bis zum nächsten, passenden **catch**-Block übersprungen.

I.1e Ausnahmebehandlung

try-catch Mechanismus


```
double kehrwert(int n) {
  if (n==0) throw string("Division durch Null");
  return 1./n;
}
int main() {
  int zahl;
  double inverse;
  do {
    cout << "Bitte eine ganze Zahl eingeben (-1=Ende): ";
    cin >> zahl;
    try {
      inverse=kehrwert(zahl);
      cout << inverse << endl;
    }
    catch(const string& s) {
      cout << s << endl << "Bitte neuen Wert eingeben:";
    }
  } while(zahl!=-1);
  return 0;
}
```

Abschließend wird der Code des entsprechenden **catch**-Blockes ausgeführt und das Programm nach diesem Block regulär ausgeführt.

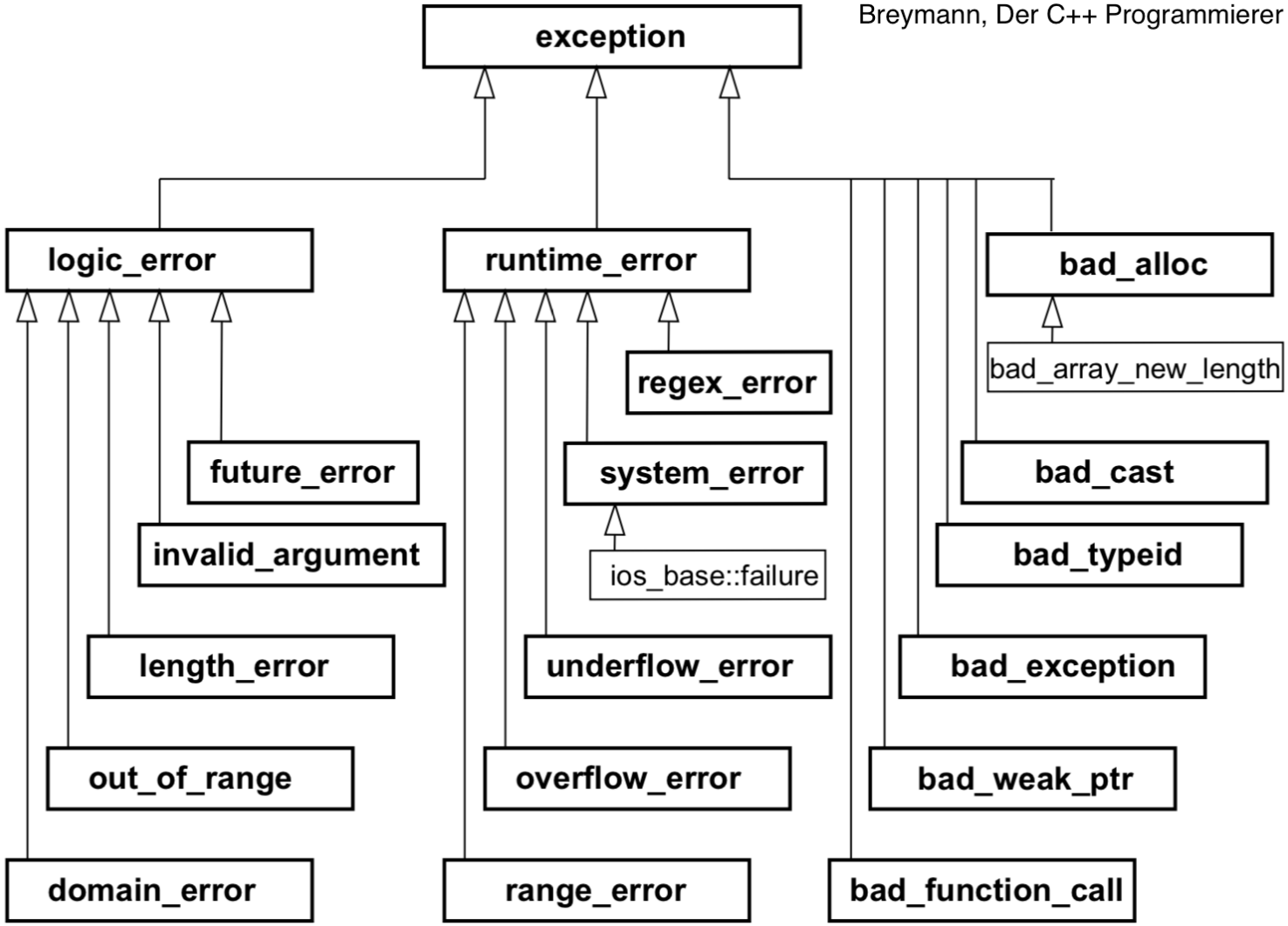
I.1e Ausnahmen

Anforderungen

- Catch-Block sollte nach der Behandlung der Ausnahme wieder den ursprünglichen Programmzustand wiederherstellen (Exception-Sicherheit)
- In Java wird dies durch die finally -Klausel erreicht
- In C++ müssen dynamische Ressourcen so erzeugt/gelöscht werden, dass beim Auftreten von Ausnahmen keine Speicherlecks oder verwitwete Objekte entstehen ☞ SmartPointer⌜Programmierung 3⌟

I.1e Ausnahmen

Ausnahmeklassen in C++



Den erbenden Klassen kann im Konstruktor ein string-Objekt (Fehlermeldung) übergeben werden. Die geerbte Methode **const char*** what() liefert dieses als Zeichenkettenkonstante zurück

I.1e Ausnahmen

Ausnahmeklassen in C++

Tabelle 7.1: Bedeutung der Exception-Klassen — Breymann, Der C++ Programmierer

Klasse	Bedeutung	Header
`exception`	Basisklasse	`<exception>`
`logic_error`	theoretisch vermeidbare Fehler, zum Beispiel Verletzung von logischen Vorbedingungen	`<stdexcept>`
`invalid_argument`	ungültiges Argument bei Funktionen	`<stdexcept>`
`length_error`	Fehler in Funktionen der Standard-C++-Bibliothek, wenn ein Objekt erzeugt werden soll, das die maximal erlaubte Größe für dieses Objekt überschreitet	`<stdexcept>`
`out_of_range`	Bereichsüberschreitungsfehler	`<stdexcept>`
`domain_error`	anderer Fehler des Anwendungsbereichs	`<stdexcept>`
`future_error`	für asynchrone System-Aufrufe	`<future>`
`runtime_error`	nicht vorhersehbare Fehler, z.B. datenabhängige Fehler	`<stdexcept>`
`regex_error`	Fehler bei regulären Ausdrücken	`<regex>`
`system_error`	Fehlermeldung des Betriebssystems	`<system_error>`
`range_error`	Bereichsüberschreitung	`<stdexcept>`
`overflow_error`	arithmetischer Überlauf	`<stdexcept>`
`underflow_error`	arithmetischer Unterlauf	`<stdexcept>`
`bad_alloc`	Speicherzuweisungsfehler (Details siehe Abschnitt 7.2)	`<new>`
`bad_typeid`	falscher Objekttyp (vgl. Abschnitt 6.9)	`<typeinfo>`
`bad_cast`	Typumwandlungsfehler (vgl. Abschnitt 6.8)	`<typeinfo>`
`bad_weak_ptr`	kann vom `shared_ptr`-Konstruktor geworfen werden	`<memory>`
`bad_function_call`	wird ggf. von `function::operator()()` geworfen	`<functional>`

I.2 Von Java nach C++

Inhalt

1. Daten
   - Datentypen
   - Ströme
   - Referenztypen
   - Speicherklassen
   - Ausnahmebehandlung
2. Funktionen
   - Werte- und Referenzsemantik
   - Operatorüberladung
   - Überladungsauflösung
3. Klassen
   - Klassen in C++
   - Operatormethoden
   - benutzerdefinierte Typumwandlungen
   - Funktionsobjekte und Prädikate
4. sequentielle Container

1 Daten
- Datentypen
- Ströme
- Referenztypen
- Speicherklassen
- Ausnahmebehandlung

2 Funktionen
- Werte- und Referenzsemantik
- Operatorüberladung
- Überladungsauflösung

3 Klassen
- Klassen in C++
- Operatormethoden
- benutzerdefinierte Typumwandlungen
- Funktionsobjekte und Prädikate

4 sequentielle Container

I.2a Wert- und Referenzparameter

- In Java können primitive Datentypen nur per Wert und Arrays/Objekte nur per Referenz an Methoden übergeben werden
- In C++ wird jeder primitive Datentyp und jedes Objekt stets per Wert übergeben. Für die Referenzübergabe müssen die Formalparameter einer Funktion/Methode Referenztypen sein.

```
// T bezeichne einen beliebigen C++ Typ oder Klasse

void f(T t); // akzeptiert jeden nach T konvertierbaren
             // _Wert_. Dieser wird als Kopie in t
             // eingeschrieben. Aenderungen an t wirken
             // sich nicht auf das Original aus

void f(T& t); // akzeptiert jede nach T konvertierbare
              // Variable (L-Wert). t ist eine Referenz
              // auf den uebergebenen Wert. Aenderungen
              // des Wertes wirken sich auf das Original
              // aus

void f(const T& t); // akzeptiert jeden nach T
                    // konvertierbaren L- oder R-Wert
                    // und besitzt kein Schreibrecht
                    // darauf
```

Java — call by value

```
public class K {
 static public // call by value fuer primitive Datentypen:
 void swap(int x, int y) {     // x,y sind Kopien von a,b
  int hilf=x; x=y; y=hilf;     // Kopien werden vertauscht
 }                             // ...und geloescht

 static public void main(String[] args) {
  int a=1, b=2; // a==1, b==2
  swap(a,b);    // a==1, b==2 --> call by value
 }
}
```

C++ — call by reference

```
void swap(int& x, int& y) {  // x,y sind Verweise auf a,b
 int hilf {x}; x=y; y=hilf;  // Inhalte werden vertauscht
}                            // Verweise werden wieder geloescht

int main() {
 int a {1}, b {2}; // a==1, b==2
 swap(a,b);        // a==2, b==1 --> call by reference
} // ohne &-Zeichen oben: Dasselbe Ergebnis wie im Java-Beispiel
```

Es sei obj ein Objekt einer Klasse Obj mit Attribut **int** i;

Java — call by reference

```
public class K {
 static public // call by reference fuer Referenztypen
 void set(Obj obj, int _i) { // obj ist Referenztyp
  obj.setze_i(_i);           // Wrapper fuer Methode Obj.setze_i
 }
 static public void main(String[] args) {
  Obj obj = new Obj();
  set(obj, 10);              // obj.i == 10 --> call by reference
 }
}
```

C++ — call by reference

```
void set(Obj& obj, int _i) { //!obj als Referenz uebergeben!
 obj.setze_i(_i);           // Wrapper fuer Methode Obj::setze_i
}
int main() {
 Obj obj;                   // Wertesemantik, keine Referenz!
 set(obj, 10);              // obj.i == 10 --> call by reference
} // ohne &-Zeichen oben: Attribut i in Objektkopie gesetzt!
```

I.2b Von Java nach C++

Inhalt

1. Daten
   - Datentypen
   - Ströme
   - Referenztypen
   - Speicherklassen
   - Ausnahmebehandlung
2. Funktionen
   - Werte- und Referenzsemantik
   - Operatorüberladung
   - Überladungsauflösung
3. Klassen
   - Klassen in C++
   - Operatormethoden
   - benutzerdefinierte Typumwandlungen
   - Funktionsobjekte und Prädikate
4. sequentielle Container

I.2b Operatorüberladung

Alle Operatoren in C++ sind als Operatorfunktionen implementiert

Operatorüberladung

Bis auf ::, ., .*, ?:, **sizeof**, x_cast und **typeid** lassen sich alle Operatoren überladen.

<Rückgabetyp> **operator**+ (Parameterliste der Operanden);
↑ beliebiges Operatorzeichen

Regeln beim Überladen von Operatoren

- Operator-Prioritäten werden beim Überladen nicht verändert
- zusammengesetzte Operatoren (z.B. +=) passen sich nicht automatisch der neuen Bedeutung von + an, sie müssten dazu explizit überladen werden
- binäre Operatoren (z.B. +, −, ∗, /) müssen in der Regel einen neuen Ergebniswert erzeugen und zurückgeben

Demonstration siehe Beispielprogramm ☞ RGB_Pixel

1 Daten
- Datentypen
- Ströme
- Referenztypen
- Speicherklassen
- Ausnahmebehandlung

2 Funktionen
- Werte- und Referenzsemantik
- Operatorüberladung
- Überladungsauflösung

3 Klassen
- Klassen in C++
- Operatormethoden
- benutzerdefinierte Typumwandlungen
- Funktionsobjekte und Prädikate

4 sequentielle Container

I.2c Uberladungsauflösung

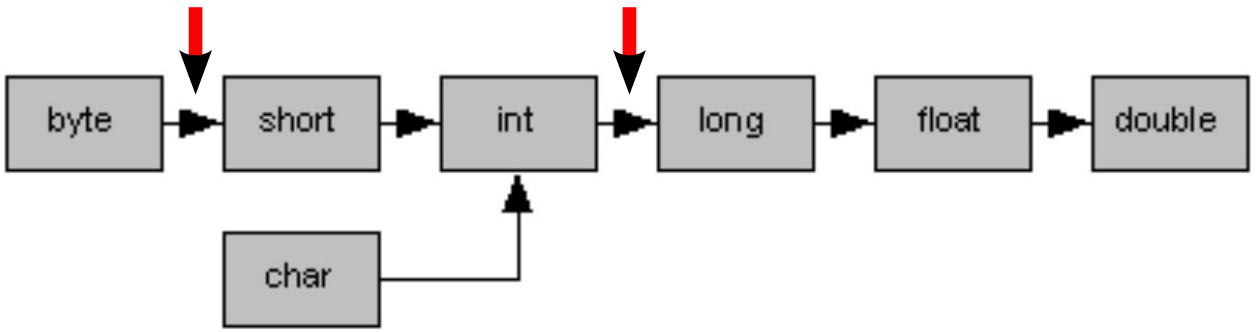
automatische Typumwandlungen in C++ und Java

Erinnerung: Promotion und Standardkonvertierungen in C++

TYPKONVERTIERUNGEN

1. Promotion (werterhaltende Konversion)
   - ganzzahlig: `bool`, `char`, `short` $\longrightarrow$ `int`
   - Fließkommatypen: float$\longrightarrow$double
2. Standardkonvertierungen (i.allg. nicht werterhaltend)
   - `int`$\longleftrightarrow$`long`
   - `int`$\longleftrightarrow$`float/double`

Zum Vergleich: In Java werden bei Bedarf die folgenden Konvertierungen durchgeführt:



Quelle: G. Krüger, H. Hansen, Handbuch der Java Programmierung

I.2c Überladungsauflösung

Auflösung von Mehrdeutigkeiten

Bei jedem Aufruf einer überladenen Funktion wird jeder Parameter des Aufrufs nach den folgenden Kriterien der Überladungsauflösung beurteilt

Krit.	Bezeichnung	Beschreibung	Beispiel
1	Übereinstimmung	Der Typ entspricht dem verlangten (bis auf triviale Umwandlungen, z.B. T $\rightarrow$ const T; Zuordnung einen Schablonentyps)	int $\rightarrow$ int int $\rightarrow$ const int
2	Promotion	Werterhaltende Konversion	float $\rightarrow$ double
3	Standardkonvertierung	Im allgemeinen nicht werterhaltende Konversion	int $\leftrightarrow$ float
4	Benutzerdefinierte Konvertierung	Typumwandlungsoperator	
5	Übereinstimmung mit variabler Parameterliste	Parameter lässt sich in variable Parameterliste übergeben	

I.2c Überladungsauflösung

Auswahlverfahren

Nach Beurteilung der Parameter nach den Kriterien der Überladungsauflösung wählt der Compiler den passendsten Kandidaten nach folgendem Auswahlverfahren:

Auswahlverfahren

Eine Funktion wird aufgerufen, falls

- für jeden Parameter das Zuordnungskriterium minimal ist
- die so gefundene Signatur eindeutig ist

Andernfalls wird der Aufruf abgelehnt.

Warnung | Ausnahme bei Promotion

Von **int** nach **long** findet keine Promotion statt!

I.2c Uberladungsauflösung: Beispiel

Bestimmen Sie, welcher Kandidat bei welchem Aufruf gewählt wird, indem Sie die einzelnen Zuordnungskriterien der Überladungsauflösung in der Tabelle eintragen und so die Auswahl treffen.

Welcher Kandidat wird gewählt?

Aufruf: `f(d, i)` mit `d: double, i: int`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`		
☐	`f(float, int)`		
☐	`f(float, long)`		
☐	keine Auswahl		

I.2c Uberladungsauflösung: Beispiel

Bestimmen Sie, welcher Kandidat bei welchem Aufruf gewählt wird, indem Sie die einzelnen Zuordnungskriterien der Überladungsauflösung in der Tabelle eintragen und so die Auswahl treffen.

Welcher Kandidat wird gewählt?

Aufruf: `f(d, i)` mit `d: double, i: int`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`	1	
☐	`f(float, int)`		
☐	`f(float, long)`		
☐	keine Auswahl		

I.2c Uberladungsauflösung: Beispiel

Bestimmen Sie, welcher Kandidat bei welchem Aufruf gewählt wird, indem Sie die einzelnen Zuordnungskriterien der Überladungsauflösung in der Tabelle eintragen und so die Auswahl treffen.

Welcher Kandidat wird gewählt?

Aufruf: `f(d, i)` mit `d: double, i: int`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`	1	
☐	`f(float, int)`	3	
☐	`f(float, long)`		
☐	keine Auswahl		

I.2c Uberladungsauflösung: Beispiel

Bestimmen Sie, welcher Kandidat bei welchem Aufruf gewählt wird, indem Sie die einzelnen Zuordnungskriterien der Überladungsauflösung in der Tabelle eintragen und so die Auswahl treffen.

Welcher Kandidat wird gewählt?

Aufruf: `f(d, i)` mit `d: double, i: int`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`	1	
☐	`f(float, int)`	3	
☐	`f(float, long)`	3	
☐	keine Auswahl		

I.2c Uberladungsauflösung: Beispiel

Bestimmen Sie, welcher Kandidat bei welchem Aufruf gewählt wird, indem Sie die einzelnen Zuordnungskriterien der Überladungsauflösung in der Tabelle eintragen und so die Auswahl treffen.

Welcher Kandidat wird gewählt?

Aufruf: `f(d, i)` mit `d: double, i: int`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`	1	3
☐	`f(float, int)`	3	
☐	`f(float, long)`	3	
☐	keine Auswahl		

I.2c Uberladungsauflösung: Beispiel

Bestimmen Sie, welcher Kandidat bei welchem Aufruf gewählt wird, indem Sie die einzelnen Zuordnungskriterien der Überladungsauflösung in der Tabelle eintragen und so die Auswahl treffen.

Welcher Kandidat wird gewählt?

Aufruf: `f(d, i)` mit `d: double, i: int`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`	1	3
☐	`f(float, int)`	3	1
☐	`f(float, long)`	3	
☐	keine Auswahl		

I.2c Uberladungsauflösung: Beispiel

Bestimmen Sie, welcher Kandidat bei welchem Aufruf gewählt wird, indem Sie die einzelnen Zuordnungskriterien der Überladungsauflösung in der Tabelle eintragen und so die Auswahl treffen.

Welcher Kandidat wird gewählt?

Aufruf: `f(d, i)` mit `d: double, i: int`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`	1	3
☐	`f(float, int)`	3	1
☐	`f(float, long)`	3	3
☐	keine Auswahl		

I.2c Überladungsauflösung: Beispiel

Bestimmen Sie, welcher Kandidat bei welchem Aufruf gewählt wird, indem Sie die einzelnen Zuordnungskriterien der Überladungsauflösung in der Tabelle eintragen und so die Auswahl treffen.

Welcher Kandidat wird gewählt?

Aufruf: `f(d, i)` mit `d: double, i: int`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`	1	3
☐	`f(float, int)`	3	1
☐	`f(float, long)`	3	3
☑	keine Auswahl		

I.2c Überladungsauflösung: Beispiel

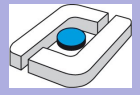
Bestimmen Sie, welcher Kandidat bei welchem Aufruf gewählt wird, indem Sie die einzelnen Zuordnungskriterien der Überladungsauflösung in der Tabelle eintragen und so die Auswahl treffen.

Welcher Kandidat wird gewählt?

Aufruf: `f(d, i)` mit `d: double, i: int`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`	1	3
☐	`f(float, int)`	3	1
☐	`f(float, long)`	3	3
☑	keine Auswahl		

In Java wäre Kandidat 1 aufgerufen worden!



I.2c Uberladungsauflösung: Beispiel

Fortsetzung

Welcher Kandidat wird gewählt?

Aufruf: `f(i, l)` mit `i: int, l: long`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`		
☐	`f(float, int)`		
☐	`f(float, long)`		
☐	keine Auswahl		

Aufruf: `f(i, s)` mit `i: int, s: short`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`		
☐	`f(float, int)`		
☐	`f(float, long)`		
☐	keine Auswahl		

I.2c Uberladungsauflösung: Beispiel

Fortsetzung

Welcher Kandidat wird gewählt?

Aufruf: `f(i, l)` mit `i: int, l: long`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`	3	
☐	`f(float, int)`		
☐	`f(float, long)`		
☐	keine Auswahl		

Aufruf: `f(i, s)` mit `i: int, s: short`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`		
☐	`f(float, int)`		
☐	`f(float, long)`		
☐	keine Auswahl		

I.2c Uberladungsauflösung: Beispiel

Fortsetzung

Welcher Kandidat wird gewählt?

Aufruf: `f(i, l)` mit `i: int, l: long`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`	3	
☐	`f(float, int)`	3	
☐	`f(float, long)`		
☐	keine Auswahl		

Aufruf: `f(i, s)` mit `i: int, s: short`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`		
☐	`f(float, int)`		
☐	`f(float, long)`		
☐	keine Auswahl		

I.2c Uberladungsauflösung: Beispiel

Fortsetzung

Welcher Kandidat wird gewählt?

Aufruf: `f(i, l)` mit `i: int, l: long`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`	3	
☐	`f(float, int)`	3	
☐	`f(float, long)`	3	
☐	keine Auswahl		

Aufruf: `f(i, s)` mit `i: int, s: short`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`		
☐	`f(float, int)`		
☐	`f(float, long)`		
☐	keine Auswahl		

I.2c Uberladungsauflösung: Beispiel

Fortsetzung

Welcher Kandidat wird gewählt?

Aufruf: `f(i, l)` mit `i: int, l: long`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`	3	1
☐	`f(float, int)`	3	
☐	`f(float, long)`	3	
☐	keine Auswahl		

Aufruf: `f(i, s)` mit `i: int, s: short`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`		
☐	`f(float, int)`		
☐	`f(float, long)`		
☐	keine Auswahl		

I.2c Uberladungsauflösung: Beispiel

Fortsetzung

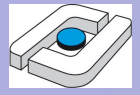
Welcher Kandidat wird gewählt?

Aufruf: `f(i, l)` mit `i: int, l: long`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`	3	1
☐	`f(float, int)`	3	3
☐	`f(float, long)`	3	
☐	keine Auswahl		

Aufruf: `f(i, s)` mit `i: int, s: short`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`		
☐	`f(float, int)`		
☐	`f(float, long)`		
☐	keine Auswahl		



I.2c Uberladungsauflösung: Beispiel

Fortsetzung

Welcher Kandidat wird gewählt?

Aufruf: `f(i, l)` mit `i: int, l: long`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`	3	1
☐	`f(float, int)`	3	3
☐	`f(float, long)`	3	1
☐	keine Auswahl		

Aufruf: `f(i, s)` mit `i: int, s: short`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`		
☐	`f(float, int)`		
☐	`f(float, long)`		
☐	keine Auswahl		

I.2c Uberladungsauflösung: Beispiel

Fortsetzung

Welcher Kandidat wird gewählt?

Aufruf: `f(i, l)` mit `i: int, l: long`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`	3	1
☐	`f(float, int)`	3	3
☐	`f(float, long)`	3	1
☑	keine Auswahl		

Aufruf: `f(i, s)` mit `i: int, s: short`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`		
☐	`f(float, int)`		
☐	`f(float, long)`		
☐	keine Auswahl		

I.2c Uberladungsauflösung: Beispiel

Fortsetzung

Welcher Kandidat wird gewählt?

Aufruf: `f(i, l)` mit `i: int, l: long`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`	3	1
☐	`f(float, int)`	3	3
☐	`f(float, long)`	3	1
☑	keine Auswahl		

Aufruf: `f(i, s)` mit `i: int, s: short`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`	3	
☐	`f(float, int)`		
☐	`f(float, long)`		
☐	keine Auswahl		

I.2c Uberladungsauflösung: Beispiel

Fortsetzung

Welcher Kandidat wird gewählt?

Aufruf: `f(i, l)` mit `i: int, l: long`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`	3	1
☐	`f(float, int)`	3	3
☐	`f(float, long)`	3	1
☑	keine Auswahl		

Aufruf: `f(i, s)` mit `i: int, s: short`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`	3	
☐	`f(float, int)`	3	
☐	`f(float, long)`		
☐	keine Auswahl		

I.2c Uberladungsauflösung: Beispiel

Fortsetzung

Welcher Kandidat wird gewählt?

Aufruf: `f(i, l)` mit `i: int, l: long`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`	3	1
☐	`f(float, int)`	3	3
☐	`f(float, long)`	3	1
☑	keine Auswahl		

Aufruf: `f(i, s)` mit `i: int, s: short`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`	3	
☐	`f(float, int)`	3	
☐	`f(float, long)`	3	
☐	keine Auswahl		

I.2c Uberladungsauflösung: Beispiel

Fortsetzung

Welcher Kandidat wird gewählt?

Aufruf: `f(i, l)` mit `i: int, l: long`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`	3	1
☐	`f(float, int)`	3	3
☐	`f(float, long)`	3	1
☑	keine Auswahl		

Aufruf: `f(i, s)` mit `i: int, s: short`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`	3	3
☐	`f(float, int)`	3	
☐	`f(float, long)`	3	
☐	keine Auswahl		

I.2c Uberladungsauflösung: Beispiel

Fortsetzung

Welcher Kandidat wird gewählt?

Aufruf: `f(i, l)` mit `i: int, l: long`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`	3	1
☐	`f(float, int)`	3	3
☐	`f(float, long)`	3	1
☑	keine Auswahl		

Aufruf: `f(i, s)` mit `i: int, s: short`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`	3	3
☐	`f(float, int)`	3	2
☐	`f(float, long)`	3	
☐	keine Auswahl		

I.2c Uberladungsauflösung: Beispiel

Fortsetzung

Welcher Kandidat wird gewählt?

Aufruf: `f(i, l)` mit `i: int, l: long`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`	3	1
☐	`f(float, int)`	3	3
☐	`f(float, long)`	3	1
☑	keine Auswahl		

Aufruf: `f(i, s)` mit `i: int, s: short`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`	3	3
☐	`f(float, int)`	3	2
☐	`f(float, long)`	3	3
☐	keine Auswahl		

I.2c Uberladungsauflösung: Beispiel

Fortsetzung

Welcher Kandidat wird gewählt?

Aufruf: `f(i, l)` mit `i: int, l: long`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`	3	1
☐	`f(float, int)`	3	3
☐	`f(float, long)`	3	1
☑	keine Auswahl		

Aufruf: `f(i, s)` mit `i: int, s: short`

	Kandidat	Parameter 1	Parameter 2
☐	`f(double, long)`	3	3
☑	`f(float, int)`	3	2
☐	`f(float, long)`	3	3
☐	keine Auswahl		

Nur der zweite Aufruf hätte ebenfalls so in Java stattgefunden!

1. Daten
   - Datentypen
   - Ströme
   - Referenztypen
   - Speicherklassen
   - Ausnahmebehandlung
2. Funktionen
   - Werte- und Referenzsemantik
   - Operatorüberladung
   - Überladungsauflösung
3. Klassen
   - Klassen in C++
   - Operatormethoden
   - benutzerdefinierte Typumwandlungen
   - Funktionsobjekte und Prädikate
4. sequentielle Container

I.3a Von Java nach C++

Inhalt

1. Daten
   - Datentypen
   - Ströme
   - Referenztypen
   - Speicherklassen
   - Ausnahmebehandlung
2. Funktionen
   - Werte- und Referenzsemantik
   - Operatorüberladung
   - Überladungsauflösung
3. Klassen
   - Klassen in C++
   - Operatormethoden
   - benutzerdefinierte Typumwandlungen
   - Funktionsobjekte und Prädikate
4. sequentielle Container

In C++ wird der Quellcode einer Klasse auf 2 Dateien aufgeteilt

- Die Headerdatei ist die öffentliche Schnittstelle, die im wesentlichen nur Vereinbarungen von Klassen und Methoden enthält, sowie von Funktionen außerhalb von Klassen
- Die Implementationsdatei enthält alle Implementationen. Zur korrekten Zuordnung einer Implementation muss mittels des Bereichsauflösungsoperators :: die zugehörige Klasse dem Methodennamen vorangestellt werden.
- Durch diese Aufteilung kann auch nachträglich die Implementation von Methoden verändert werden, ohne die Schnittstelle zu verändern

Test.h

```
void f(int);
class Test {
 int i {0};   // Init
public:
 int gib_i();
};
```

Test.cpp

```
f(int i) {
 cout << "f::i";
}
int Test::gib_i() {
 return i;
}
```

main.cpp

```
int main() {
 f(5);
 Test test;
 test.gib_i();
}
```

I.3a Trennung von Schnittstelle und Implementation

☞ TEST_MAIN

- Im Header werden in der Regel keine Implementationen vorgenommen. Falls doch, werden Methoden als **inline** interpretiert, d.h. der Compiler ersetzt jeden Aufruf direkt durch die Implementation 「Vorsicht: Codeaufblähung」
- Die Implementationsdatei und Module, die den Code verwenden, müssen die zugehörige Headerdatei mittels **#include** einbinden. Dadurch kann der Compiler syntaktisch die korrekte Verwendung der eingebundenen Funktionen/Methoden prüfen
- Um Mehrfacheinbindungen zu verhindern, enthält die Headerdatei einen sogenannten Wächter, der vom Präprozesser ausgewertet wird und das Modul bedingungsabhängig dem Compiler zuführt

Test.h

```
#ifndef Test_h
#define Test_h

// hier weiter

#endif
```

Test.cpp

```
#include "Test.h"

// hier weiter
```

main.cpp

```
#include "Test.h"

// hier weiter
```

I.3a Sichtbarkeiten in Klassen

In C++ werden Sichtbarkeiten nicht für jede Methode/Attribut einzeln gesetzt, sondern gruppiert in Abschnitten hinter z.B. **public**:

Sichtbarkeit	Java	C++
public	öffentlicher Zugriff	
protected	nur erbende Klassen, sowie Klassen im selben Paket haben Zugriff	nur erbende Klassen haben Zugriff
package	(Standard) — nur Klassen im selben Paket haben Zugriff	gibt es in C++ nicht!
private	nur die eigene Klasse kann zugreifen	(Standard) — nur die eigene Klasse kann zugreifen

Sonderfall struct

Strukturen in C++ sind Klassen, bei denen die Sichtbarkeit **public** als Standard voreingestellt ist

I.3a Sichtbarkeiten in Klassen

In C++ werden Sichtbarkeiten nicht für jede Methode/Attribut einzeln gesetzt, sondern gruppiert in Abschnitten hinter z.B. **public**:

Java

```
public class K {
 private int i;
 protected int j;
 public K() {}
 public void methode(){}
}
```

C++

```
class K {   // immer public
 int i {0};// Voreinstellung private
protected: // ab hier: protected
 int j {0};
public:    // ab hier: public
 K() {}
 void methode() {}
}; // Semikolon nicht vergessen!
```

Die Sichtbarkeitsabschnitte können beliebig oft und in beliebiger Reihenfolge aufgeführt werden

I.3a Konstante Objekte und Methoden

Der Qualifizierer **const** ist eine Zusicherung an den Compiler:

```
K K::methode(const int& i) const {}
```

- Das erste **const** sichert zu, dass das zurückgegebene Objekt nicht verändert werden kann
- Das zweite **const** sichert zu, dass der Übergabeparameter eine konstante Referenz ist (d.h. nur mit Leserecht)
- Das dritte **const** bezieht sich auf das Objekt, an dem die Methode aufgerufen wird. Es sichert zu, dass die Methode keine Objektvariablen verändert.
- Ausnahme: Der Qualifizierer **mutable** vor einer Objektvariablen, lässt Änderungen von konstanten Methoden zu.
- Konstante Objekte können (wie andere Variablen auch) im Programm vereinbart werden. Für konstante Objekte stehen nur als konstant markierte Methoden zur Verfügung.
- Methoden können sowohl in einer konstanten und einer nicht-konstanten Version vorliegen. Daher muss der Qualifizierer **const** sowohl bei der Vereinbarung, als auch bei der Implementation angefügt werden.

Der Spezifizierer **static** kennzeichnet Klassenvariablen und Klassenmethoden

- Klassenvariablen und -konstanten dürfen nur einmal außerhalb der Klasse initialisiert werden, oder **inline** innerhalb der Klassenvereinbarung
- Außerhalb der Klasse geschieht der Zugriff über den Bereichsauflösungsoperator ::
- statische Methoden dürfen nur Klassenvariablen und keine Objektvariablen verwenden

Test.h

```
class K {
 static int i;
 static inline int j {1};
 static const int k;
 static const int l {1};
 static int feld[l];
public:
 static void methode();
}
```

Test.cpp

```
int K::i {1};
const int K::k {1};
void K::methode() {}
```

I.3a Standardmethoden

In C++ gibt es keine Wurzelklasse `Object` aber Methoden, die automatisch angelegt werden

- Standardkonstruktor (SK)
- Kopierkonstruktor (KK) bei Wertüber-/rückgabe stets aufgerufen!
- Zuweisungsoperator für Komponentenzuweisungen(!)
- Destruktor (D): "letzter Wille des Objekts vor seiner Zerstörung"

「Java: public void finalize(); wichtig in Kapitel ☞ ZEIGER」

```
class K {
 size_t anz {0};
 vector<int> vec;
public:
 K() = default;              // Standardkonstruktor behalten
 K(size_t);                  // Ueberladung
 K(const K&);                // Kopierkonstruktor
 K& operator=(const K&);     // Zuweisungsoperator
 ~K() = default;             // Standarddestruktor verwenden
};
```

- =**default** übernimmt die Standardimplementation des Systems
- =**delete** ermöglicht das Sperren eines Aufrufs

Es gibt standardmäßig KEINE der Methoden clone, equals, toString

I.3a Standardmethoden


```
//                    v-- leitet Initialisierungsliste ein
K::K(size_t _anz) : anz(_anz), vec(_anz) {}
// Vektor mit vorgegebener Laenge --^

K::K(const K& orig)
: anz(orig.anz), vec(orig.vec) {}

K& K::operator=(const K& rhs) {
 anz = rhs.anz;
 vec = rhs.vec;
 return *this;
}
```

- Initialisierungslisten initialisieren Objektvariablen bereits vor Betreten des Anweisungsblocks
- Die Initialisierungsreihenfolge ist durch die Reihenfolge der Attribute in der Klassendeklaration festgelegt!
- Initialisierung ist bei Objekten anderer Klassen ein Konstruktoraufruf
  「einzige Möglichkeit, überladene Konstruktoren aufzurufen」
- Zuweisungsoperator nimmt explizite Komponentenzuweisungen vor

I.3a Anlegen von Objekten

Objekte werden, wie jede andere Variable auch, auf dem Stack erzeugt und nach verlassen des Programmblocks wieder zerstört

```
K f(K k) { return k; }

int main() {
 K k1;       // Anlegen mit SK
 K k2(1);    // Anlegen mit ueberladenem Konstruktor
 K k3 {1}; // Initialisierung: Auswahl passender Konstruktor
             //                    ( K(int) )

 k1 = k3;   // kopiert Inhalte
 k2 = f(k3); // 2*KK + 2*D ! --> besser: Referenzuebergabe
}            // Destruktoraufruf fuer k1, k2, k3
```

Warnung

- Der Ausdruck K k(); erzeugt in C++ kein K-Objekt, sondern wird als Vereinbarung einer parameterlosen Funktion k() interpretiert, die ein K-Objekt zurückliefert. Solange diese "Funktion" nicht verwendet wird, gibt es keine Fehlermeldung des Compilers!
- Destruktoren sollten niemals Ausnahmen werfen 「Warum?」

Der Sequenzkonstruktor erlaubt variable Initialisierungslisten:
Vereinbart man zusätzlich die Methode ⌜**#include**< initializer_list >⌟:

```
K::K(initializer_list<int> liste): anz(liste.size()) {
 for (int wert: liste) vec.push_back(wert);
}
```

```
int main() {
 K k1;       // Anlegen mit SK
 K k2(1);    // Anlegen mit ueberladenem Konstruktor
 K k3 {1}; // Auswahl passender Konstruktor
             // Sequenzkonstruktor hat Vorrang
 K k4 {1, 2, 3, 4, 5, 6};
}
```

erhält man die Belegungen

```
k1: 0|
k2: 1| 0
k3: 1| 1
k4: 6| 1 2 3 4 5 6
```

Bestimmen Sie die Ausgabe

```
#include<iostream>
#include<string>
using namespace std;

class Person {
 string name, adresse;
public:
 Person()           { cout << "Person" << endl; }
 Person(Person& p) : name(p.name), adresse(p.adresse)
                    { cout <<"Person-Init"<<endl; }
 ~Person()          { cout << "~Person" << endl; }
};

class Student {
 Person person;
 string matrikelnummer, semester;
public:
 Student()          { cout << "Student" << endl; }
 Student(Student& t)
  : person(t.person), matrikelnummer(t.matrikelnummer), semester(t.semester)
                    { cout << "Student-Init" << endl; }
 ~Student()         { cout << "~Student" << endl; }
};

Student returnStudent(Student t) {return t;}

int main() {
 Student s;
 returnStudent(s);
}
```

OK oder nicht OK?

```
1  #include <iostream>
2  using namespace std;
3
4  class Test {
5   int i;
6   // weitere Komponenten
7  public:
8   Test() : i(0) {}                          // Standardkonstruktor
9   Test(const Test& t) { *this=t; }          // Kopierkonstruktor
10  Test operator= (const Test& t)            // Zuweisungsoperator
11    { i=t.i; return *this; }
12  void put(const int v) { i=v; }            // Weitere Methoden
13  int get() { return i; }
14 };
15
16 int main() {
17  Test t;
18  Test tt(t);
19  cout << t.get() << " " << tt.get() << endl;
20  return 0;
21 }
```

I.3b Von Java nach C++

Inhalt

1. Daten
   - Datentypen
   - Ströme
   - Referenztypen
   - Speicherklassen
   - Ausnahmebehandlung
2. Funktionen
   - Werte- und Referenzsemantik
   - Operatorüberladung
   - Überladungsauflösung
3. Klassen
   - Klassen in C++
   - Operatormethoden
   - benutzerdefinierte Typumwandlungen
   - Funktionsobjekte und Prädikate
4. sequentielle Container

I.3b Operatormethoden

Operatormethoden sind überladene Operatoren in Klassen, der erste Operand wird dabei automatisch durch **this** vorbelegt.

I.3b Operatormethoden

I Ops, die einen elementaren Datentyp zurückliefern

☞ OP_ELEMENTARER_DATENTYP

```
class K {
 vector<double> werte
public:
 //!
 //! typisches Beispiel I: Vergleichsoperatoren
 // in der Regel implementiert man nur
 bool operator==(const K&) const;
 bool operator< (const K&) const;
 // alle anderen Vergleichsoperatoren
 // sind davon abgeleitet
 bool operator!=(const K& rhs) { return !operator==(rhs); }
 bool operator<=(const K& rhs)
 { return ( operator==(rhs) || operator<(rhs) ); }
 bool operator> (const K& rhs) { return !operator<=(rhs); }
 bool operator>=(const K& rhs) { return !operator< (rhs); }
 //!
 //! typisches Beispiel II: Indexoperator
 // Variante mit Schreibrecht auf den Vektorinhalten
 double&       operator[](int i)       { return werte[i]; }
 // Variante ohne Schreibrecht auf den Vektorinhalten
 const double& operator[](int i) const { return werte[i]; }
 // normalerweise beeinhaltet die Implementation auch
 // noch eine Gueltigkeitspruefung des uebergebenen Index i
};
```

I.3b Operatormethoden

II Operationen, die das aktuelle Objekt verändern

☞ OP_AKTUELLES_OBJEKT

```
class K {
 public:
 // Praefix-Operatoren
 K& operator++() { /* Implementation */ return *this; }
 K& operator--() { /* Implementation */ return *this; }

// += und -= (ggf. auf ++/-- zurueckfuehren)
K& operator+=(Param) { /*Implementation*/ return *this; }
K& operator-=(Param) { /*Implementation*/ return *this; }

// Weitere, z.B. *= und /= (ggf. auf +=/-= zurueckfueren)...
};
```

```
class K {
 public:
 // Praefix-Inkrement
 K& operator++() { /* Implementation */ return *this; }

// += und (ggf. auf ++ zurueckfuehren)
K& operator+=(Param) { /*Implementation*/ return *this; }
};

//!
//!ausserhalb der Klasse
// Postfix Inkrement
K operator++(K& k, int) {
 K temp(k);
 ++k;
 return temp;
}

// Addition mit neuem Ergebnis
K operator+(const K& lhs, const K& rhs) {
 K ergebnis(lhs);
 return ergebnis+=rhs;
}
```

I.3b Operatormethoden

III Operatoren, die ein neues Objekt erzeugen

- Operatoren wie $+$, $-$, $*$, $/$,... müssen ein neues Ergebnisobjekt zurückliefern: Die rechte Seite der Anweisung c=a+b wird zunächst als temporäres Objekt erzeugt, bevor es der linken Seite zugewiesen wird
- Andererseits ist die Funktionalität der Operatoren oftmals bereits implementiert in den korrespondierenden Operatoren auf dem aktuellen Objekt (wie $+=$, -=, *=, $/=$, ...)
- allgemein gilt außerdem für symmetrische, binäre Operatoren wie ($+$, $-$, $*$, $/$, ...), dass sie *nicht* als Methode, sondern als globale Operatorfunktion implementiert werden sollen, um Typkonvertierungen in beiden Operanden gleichberechtigt zuzulassen
- damit verlieren sie den Zugriff auf alle privaten Daten der Klasse und müssen mithilfe öffentlicher Methoden (Konstruktoren, Operatormethoden) implementiert werden
- nach den Regeln der Überladungsauflösung sind dann auch Aufrufe für beide Operanden gleichberechtigt möglich

I.3b Operatormethoden

Warnung

- Wenn Operatoren überladen werden, behalten Sie ihre Assoziativität und Priorität
- Die Operatoren $+=$, $-=$, $*=$, $/=$, $\%=$, $\&=$, $\hat{}=$, $|=$, $<<=$, $>>=$ passen sich NICHT an, wenn die zugehörigen Operatoren $+$, $-$, $*$, $/$, $\%$, $\&$, $\hat{}$, $|$, $<<$, $>>$ überladen wurden
- Die Operatoren $=$, $[]$, $()$ und $->$ können nur als nicht-statische Operatoren überladen werden, da sie zwingend an ein Objekt gebunden sind.
- Die Stromoperatoren $<<$ und $>>$ benötigen als ersten Operanden ein Objekt der Stromklassen und können daher nicht innerhalb eigener Klassen überladen werden. Sie müssen stets außerhalb der Klasse als Operatorfunktionen überladen werden.

1 Daten
- Datentypen
- Ströme
- Referenztypen
- Speicherklassen
- Ausnahmebehandlung

2 Funktionen
- Werte- und Referenzsemantik
- Operatorüberladung
- Überladungsauflösung

3 Klassen
- Klassen in C++
- Operatormethoden
- benutzerdefinierte Typumwandlungen
- Funktionsobjekte und Prädikate

4 sequentielle Container

I.3c Benutzerdefinierte Typumwandlungen

Typwandlungsoperatoren

☞ TYPUMWANDLUNG

gegeben:	Typ T
	Klasse K
gesucht:	Umwandlung T $\leftrightarrow$ K
Lösung:	T $\rightarrow$ K K(**const** T&); (Konstruktor)
	K $\rightarrow$ T **operator** T(); (Typumwandlungsoperator)

```
class Zahl {
private:
 int inhalt;
public:
 Zahl(int _inhalt) : inhalt(_inhalt) {}
 operator int() { return inhalt; }
 // Der Rueckgabetyp des Typumwandlungsoperators ist
 // sein Name (keine explizite Angabe!)
;
```

I.3c Benutzerdefinierte Typumwandlungen

Gefahren I — unbeabsichtigte Konvertierungen mittels **explicit** unterbinden

☞ TYPUMWANDLUNG_GEFAHREN1

```
#include <iostream>
using namespace std;

class X { // Klasse mit Typumwandlung durch den Konstruktor
public:
 /*explicit*/                  // verhindert:
 X(int)                        // Benutzerdefinierte Konversion
 { cout << "XK" << endl; } // von int nach X durch Konstruktor
 ~X() { cout << "XD" << endl; }
};

void f(X) { // Eine Funktion auf Objekten der Klasse X
 cout << "f aufgerufen" << endl;
}

int main() { // Demo

 f(11);                        // Konvertierung int --> X
 f(static_cast<X>(11)); // mittels X(int)

 f(0.5);                       // Konvertierung double --> int --> X
 f(static_cast<X>(0.5));
}
```

I.3c Benutzerdefinierte Typumwandlungen

Gefahren II — mehrdeutige Konvertierungen vermeiden

☞ TYPUMWANDLUNG_GEFAHREN2

```
#include <iostream>
using namespace std;

class B; // Vorwaertsdeklaration! --> macht Typ B bekannt

class A { // Klasse A, die B verwendet
 int wert;
public:
 A(int x): wert(x) {}
 A(const B&)          { wert=1; }      // Typwandlung B-->A
 int get_v() const { return wert; }
};

class B { // Definition von B, die A verwendet
public:
 operator A() { return A(2); }     // Typwandlung B-->A
};

int main() { // Demo
 B b;                          // Welche Typwandlung findet statt?
 cout << static_cast<A>(b).get_v() // --> Compilerabhaengig
      << endl;
}
```

1 Daten
- Datentypen
- Ströme
- Referenztypen
- Speicherklassen
- Ausnahmebehandlung

2 Funktionen
- Werte- und Referenzsemantik
- Operatorüberladung
- Überladungsauflösung

3 Klassen
- Klassen in C++
- Operatormethoden
- benutzerdefinierte Typumwandlungen
- Funktionsobjekte und Prädikate

4 sequentielle Container

Ein Funktionsobjekt entsteht durch Überladung des Klammer-Operators innerhalb einer Klasse:

```
Rückgabetyp operator() (Parameterliste);
```

- Der Aufruf des Operators ähnelt dem Funktionsaufruf, der mit dem aufrufenden Objekt verbunden ist
- übernimmt in C++ die Rolle einfacher Interfaces, die genau eine Methode bereitstellen
- Eine denkbare Anwendung sind "Funktion mit Gedächtnis", d.h. Klassen mit überladenem Klammeroperator und Attributen, die im Konstruktor initialisiert und zur "Berechnung eines Funktionswertes" herangezogen werden

```
class Random {
 long letzteZufallszahl;  // "Gedaechtnis"
 // Klassen-Konstanten aus Park/Miller (1988),
 // Random Number Generators, Good Ones are Hard to Find
 static const long a=16807, m=2147483647, q=m/a, r=m%a;
public:
 Random(long seed=314159): letzteZufallszahl(seed) {} //Init

 long operator()() {                  // erzeugt neue Zufallszahl
  long gamma;
  gamma = a * (letzteZufallszahl % q)
        - r * (letzteZufallszahl / q);
  if (gamma > 0) letzteZufallszahl = gamma;
  else letzteZufallszahl = gamma + m;
  return letzteZufallszahl;
 }
                                      //    Weitere Ueberladungen
 long operator()(long min, long max) //Zufallszahl innerhalb
  { return min+(operator()())%(max-min); }       // [min,max)

 double operator() (double min, double max)    // fuer double
  { return min+static_cast<double>(operator()())/m*(max-min); }
};
```

```
int main() {
 Random rd(1234);      // Initialisierung Zufallszahlen-Objekt
 for (int i=0; i<10; i++)          // Aufruf von operator()()
  cout << rd()%1000 << " ";
 cout << endl;
 for (int i=0; i<10; i++)     // Aufruf operator()(long,long)
  cout << rd(50L,100L) << " ";
 cout << endl;
 for (int i=0; i<10; i++)// Aufruf operator()(double,double)
  cout << setprecision(2) << rd(0.,1.) << " ";
 cout << endl;
}
```

Ausgabe:

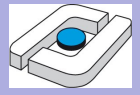
```
838 452 78 417 541 958 770 913 796 915
79 80 53 54 53 61 52 63 79 54
0.18 0.48 0.8 0.16 0.17 0.23 0.48 0.94 0.34 0.27
```

Ein Prädikat ist ein Funktionsobjekt mit spezieller Signatur:

```
bool operator() (Parameterliste) const;
```

Sie können zur Abfrage von Eigenschaften genutzt werden

```
class IstVielfachesVon {                                  // Praedikat
 int teiler;                  // hiervon werden Vielfache gesucht
public:
 IstVielfachesVon(int n): teiler(n) {}//Initialisiert teiler
 bool operator()(int zahl) const {          // prueft, ob zahl
  return !(zahl%teiler);     // ein Vielfaches von teiler ist
 }
};

// count_if zaehlt in einem Vektor alle Vielfachen von
// IstVielfachesVon::teiler
size_t count_if(const vector<int>& vec,
                const IstVielfachesVon& vielfach_n) {
 size_t anzahl {0}; // bereits initialisiert --^ (!)
 for (size_t i=0; i<vec.size(); i++)
  if (vielfach_n(vec[i])) anzahl++;
    //       ^-- Aufruf von IstVielfachesVon::operator()(int)
 return anzahl;
}

int main() {
 vector<int> vec { 7, -6, 13, 12, 8, 9, 81, -243, 2, 4};
 cout << count_if(vec, IstVielfachesVon(3)); // --> 5
 return 0;
}
```



I.4 Von Java nach C++

Inhalt

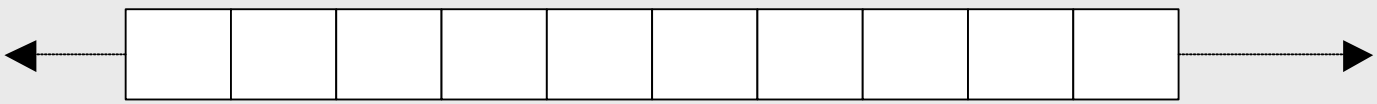
1. Daten
   - Datentypen
   - Ströme
   - Referenztypen
   - Speicherklassen
   - Ausnahmebehandlung
2. Funktionen
   - Werte- und Referenzsemantik
   - Operatorüberladung
   - Überladungsauflösung
3. Klassen
   - Klassen in C++
   - Operatormethoden
   - benutzerdefinierte Typumwandlungen
   - Funktionsobjekte und Prädikate
4. sequentielle Container

Container der STL - Sequentielle Container

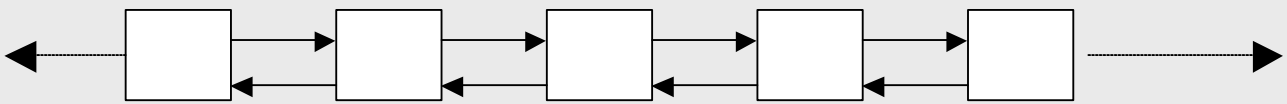
Sequentielle Container sind geordnete Container, in denen jedes Element eine bestimmte Position besitzt, der durch den Zeitpunkt und den Ort des Einfügens bestimmt ist (Josuttis [96, S.45ff]).

Vektor:

Deque:



Liste:



Erläuterung: Container

Abstrakte Datentypen zur Speicherung von Daten gleichen Typs

- realisiert als Klassenschablone (z.B.: vector<**double**>)
- andere Container erlauben z.B.
  - Schnelles Einfügen von Elementen in der Mitte
  - Schnelles Suchen nach Elementen
  - ...
- Klassenbibliothek nennt sich

STL - Standard Template Library

☞ Quellen: `www.cplusplus.com`
`www.cppreference.com`
`www.sgi.com/tech/st`

I.4 sequentielle Container

Bibliothek	`vector`	`deque`	`list`
Beschreibung	dynamisches Feld	doppelseitige Schlange	Liste
logisches Speicherabbild	front | | | | back	front | | | | back	front | | | | back

Konstruktion	`vector<T> c;`
	`vector<T> c(n);`
	`vector<T> c(n,v);`
	`vector<T> c {...};`
	`vector<T> c(d);`
		`deque<T>` ~	`list<T>` ~

Anfügen/Entfernen	`c.push_back(v);` `c.pop_back();`	+`c.push_front(v);` +`c.pop_front();`	~

Elementzugriff	`v=c.front();`
	`v=c.back();`
	`v=c[i];`
	`v=c.at(i);`
		~	−`c[i];` −`c.at(i);`

Globalzugriff	`n=c.size();`
	`c.resize(n);`
	`c.clear();`
	`b=c.empty();`
	`=`
	`==, <` (lex)
	↪`!=, <=, >, >=`
		~	+`c.unique([bp]);`
			+`c.remove(v);`
			+`c.remove_if(p);`
			+`c.sort([cmp]);`
			+`c.reverse();`
			+`c.merge(d,[cmp]);`

Legende:
`int n; T v; initializer_list<T> {...}; bool b; C<T> d;` wobei C den jeweiligen Containertyp bezeichnet
Optionale Parameter: Prädikate: `p` (unär), `bp`,`cmp` (binär) für Argumente vom Typ `T`
~: Dasselbe wie links; +: zusätzlich; −: ohne
↪: abgeleitet, Anforderung: Typ T muss `=`,`==`,`<` unterstützen; lexikographischer Vergleich gemäß logischem Speicherabbild

I.4 sequentielle Container

vector

```
vector<int> v,vv;
v.push_back(1);
v.push_back(2);
v.push_back(3);
v.push_back(0);
// 4| 1 2 3 0 |
// front: 1
// back:  0

vv=v;
cout << (vv==v);
// true

vv.pop_back();
// 3| 1 2 3 |
// front: 1
// back:  3
cout << (v<vv);
// false
```

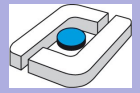
deque

```
deque<int> dq,dqq;
dq.push_back(1);
dq.push_back(2);
dq.push_back(3);
dq.push_front(0);
// 4| 0 1 2 3 |
// front: 0
// back:  3

dqq=dq;
cout << (dqq==dq);
// true

dqq.pop_front();
// 3| 1 2 3 |
// front: 1
// back:  3
cout << (dq<dqq);
// true
```

list

```
deque<int> l,ll;
l.push_back(1);
l.push_back(2);
l.push_back(3);
l.push_front(0);
// 4| 0 1 2 3 |
// front: 0
// back:  3

ll=l;
cout << (ll==l);
// true

ll.pop_front();
// 3| 1 2 3 |
// front: 1
// back:  3
cout << (l<ll);
// true
```

```
/*
 * (unaeres) Praedikat fuer Listenelemente
 * - wahr, wenn Wert des Listenelementes gleich 1 ist
 * - falsch, sonst
 */
class IstEins {
public:
 bool operator()(int n) { return (n==1); }
};

/*
 * binaeres Praedikat zum Abgleich von Listenelementen
 */
class IstBetragsGleich {
public:
 bool operator()(int i, int j) { return (abs(i)==abs(j)); }
};

/*
 * binaeres Praedikat zum Vergleich von Listenelementen
 */
class BetragKleiner {
public:
 bool operator()(int i, int j) { return (abs(i)<abs(j)); }
};
```



I.4 Listenmanipulation

...auf der Liste l={1,2,3,3,3,2,4,-4}

```
// unique() entfernt benachbarte Dubletten
l.unique();                          // 6| 1 2 3 2 4 -4 |
l.unique(IstBetragsGleich());  // 5| 1 2 3 2 4 |

// remove(v) entfernt alle Vorkommen von v
// remove_if(p) entfernt alle Elemente, fuer die das
//                Praedikat p true liefert
l.remove(2);                        // 6| 1 3 3 3 4 -4 |
l.remove_if(IstEins());    // 5| 3 3 3 4 -4 |

// sort() sortiert bzgl. operator<
// sort(cmp) sortiert bzgl. binärem Praedikat cmp
l.sort();   // 8| -4 1 2 2 3 3 3 4 |
l.sort(BetragKleiner()); // 8| 1 2 2 3 3 3 -4 4 |

// merge()
// l.merge(zweiteListe,cmp) fuegt aktuelles Element el2
// von zweiteListe vor aktuellem Element el1 von l ein,
// falls cmp(el2,el1) wahr ist.
// zweiteListe ist danach leer
list<int> zweiteListe = {2,1,-3,5};
l.merge(zweiteListe); // 12| 1 2 2 1 -3 3 3 3 2 4 -4 5 |
// bzw. mit binaerem Praedikat ergaebe sich:
l.merge(zweiteListe, BetragKleiner());
                      // 12| 1 2 2 1 3 3 3 2 -3 4 -4 5 |
```

I.4 Iteratoren

Operationen und Iteratorenkategorien

Jeder Container stellt in seinem Namensbereich eine Iteratorklasse bereit, um Containerelemente zu selektieren

`C::iterator it;`		
Container	alle	`vector, deque, string`
Kategorie	bidirektional	Random-Access

Positionierung	`c.begin();c.end();`
	`it++; it--;`
	`advance(it, n);`
	`n=distance(first, last);`
		+`it+=n; it-=n;`
		+`it=it+n; it=it-n;`

Elementzugriff	`*it; it->`*	+`it[n];`**
Vergleich	`==, !=`	+`<, <=, >, >=`

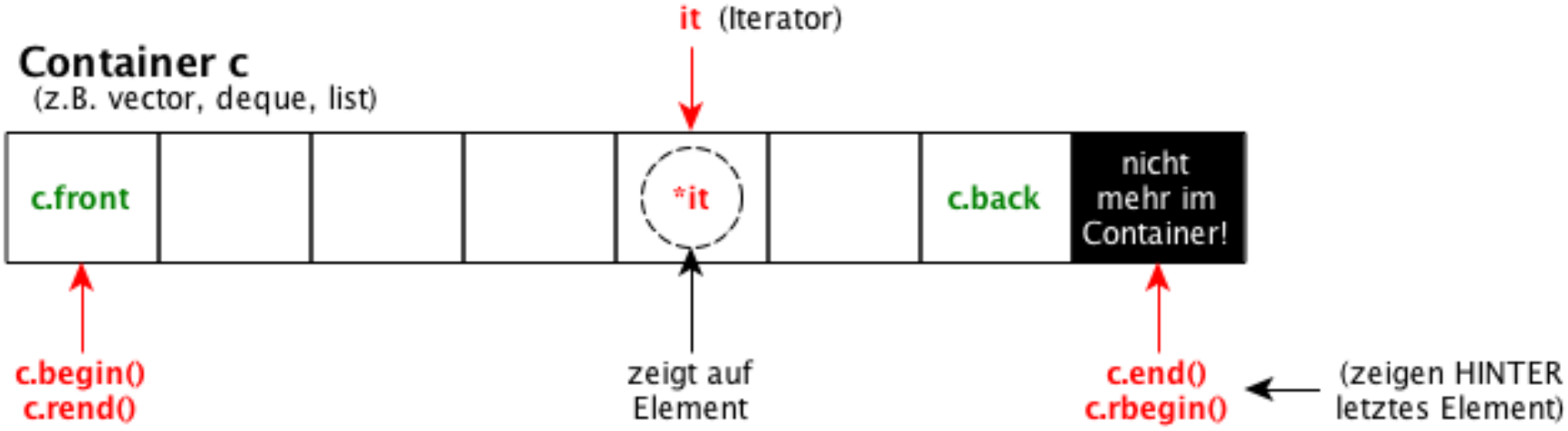
Legende:
`int n;` C: Containertyp (kein Adapter!);
+: zusätzlich; −: ohne
*: `it-> == (*it).` selektiert Klassen- oder Strukturkomponente im Elementtyp, falls zutreffend
**: `it[n] == *(it+=n);`

Weitere Kategorien/Differenzierungen:

- `Vorwärts-Iteratoren` unterstützen nur die Inkrement-Positionierung
- `Ein-/Ausgabe-Iteratoren` ermöglichen nur lesenden (`e=*it`) bzw. schreibenden (`*it=e`) Zugriff auf Containerelemente `e`

I.4 Iteratoren

Positionierung



Warnung | Referenzierung beim reverse_iterator

reverse_iterator geht von hinten nach vorne.

→ Referenziert immer auf vorausgehendes Objekt, da `c.rbegin()` nicht auf `c.back` zeigt, aber trotzdem referenziert werden kann!

```
// Initialisierung mit Sequenzkonstruktor
vector<int> v = {1,2,3,3,3,2,4,-4};
// Initialisierung mit Iteratorsequenz
list<int> l(v.begin(), v.end());

// Bidirektionale Iteratoren fuer Listen
// Positionierung und Elementzugriff
list<int>::iterator it;
list<int>::reverse_iterator r_it;

for (it=l.begin(); it!=l.end(); it++)
 cout << *it << " ";     // 1 2 3 3 3 2 4 -4
for (r_it=l.rbegin(); r_it!=l.rend(); r_it++)
 cout << *r_it << " ";   // -4 4 2 3 3 3 2 1

// Random-Access Iteratoren fuer Vektoren
vector<int>::iterator v_iter=v.begin();
v_iter+=5;
cout << *v_iter << " " << v_iter[1] << " ";     // 2 4
cout << boolalpha << (v_iter<v.end()) << endl; // true
```

```
// Eine Klasse, die nichts macht
class EgalWas {
public:
 int x;
 int y;
 EgalWas(int _x, int _y): x(_x), y(_y) {}
};
```

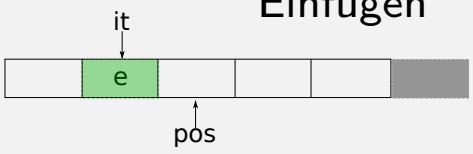
```
// Zugriff auf Klassenkomponenten
EgalWas egal(1,2), ganzEgal(3,4), totalEgal(5,6);
vector<EgalWas> vec_egal = {egal, ganzEgal, totalEgal};

vector<EgalWas>::iterator it_egal=vec_egal.begin();
for (size_t i=0; i<vec_egal.size(); i++)
 cout << it_egal[i].x << "," << (it_egal+i)->y << "␣";
 //              ^-- greift direkt auf   ^-- zeigt auf das
 //                  i-tes Element zu        i-te Element
 //                                          greift darauf zu
 //                   und selektiert die x/y-Komponente
 // Ausgabe: 1,2 3,4 5,6
```

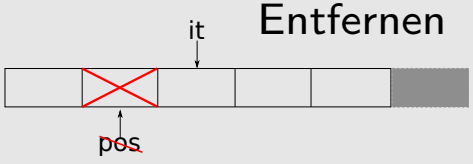
I.4 Iteratoren

Verwendung in Containermethoden

Funktionalität	Varianten	Beschreibung

Positionierung	`it=c.begin();`
	`it=c.end();`
	`r_it=c.rbegin();`
	`r_it=c.rend();`*
		

Einfügen 	`it=c.insert(pos,e);`
	`c.insert(pos,n,e);`
	`c.insert(pos,first,last);`
		fügt (n) Kopie(n) von `e` vor `pos` ein, bzw. die Sequenz** `[first,last)`***.
		`it` zeigt auf eingefügte Kopie

Entfernen 	`it=c.erase(pos);` `it=c.erase(first,last);`	löscht Element `*pos` bzw. Sequenz `[first,last)`. `it` zeigt auf nachfolgende Position bzw. `end()`

Einsetzen	`c.splice(pos,liste);`****
	`c.splice(pos,liste, it);`
	`c.splice(pos,liste,`
	`first, last);`
		setzt die `liste`, bzw. nur `*it` oder Listensequenz `[first,last)` vor `pos` ein. Eingesetzte Elemente werden `liste` entnommen

Legende:
`C c`: Container (kein Adapter!); Container-Element: `T e; int n;`
`C::iterator it, pos, first, last; C::reverse_iterator r_it;`
*Um die Reverse-Iteratoren `rbegin, rend` funktionell an `begin, end` anzugleichen, wirkt die Dereferenzierung auf die logisch vorangehende Position.
**: eines anderen Containers
***: einschließlich `first` und ausschließlich `last` von passenden Containerelementen
****: Nur, falls `c` Listencontainer

Einfügen

```
list<int> l={1,2,3,4}, list<int> ll={7,8,9};
list<int>::iterator pos, it;                          // Iteratoren
pos=l.begin();pos++;pos++;              // Startposition festlegen

it=l.insert(pos,6); l.insert(it,5); // von rechts nach links
// 6| 1 2 5 6 3 4 |

l.insert(pos,7); l.insert(pos,8);          // links nach rechts
// 8| 1 2 5 6 7 8 3 4 |
l.insert(pos,3,-6);                       // mehrfaches Einfuegen
// 11| 1 2 5 6 7 8 -6 -6 -6 3 4 |

pos=l.end();                        // Einfuegen einer Liste am Ende
l.insert(pos,ll.begin(),ll.end());
// 14| 1 2 5 6 7 8 -6 -6 -6 3 4 7 8 9 |
```

Entfernen

```
list<int> l={1,2,3,4};
list<int>::iterator pos, it;                    // Iteratoren
pos=l.begin();pos++;                  // Startposition festlegen

it=l.erase(pos);                                // pos jetzt ungueltig
// 3| 1 3 4 |

it=l.erase(it,l.end());             // Loeschen bis zum Listenende
// 1| 1 |

cout << boolalpha << (it==l.end()) << endl;
// true
```

Einsetzen

```
list<int> l={1,2,3,4}, list<int> ll={7,8,9};
list<int>::iterator pos, it;                    // Iteratoren
pos=l.begin();pos++;pos++;            // Startposition festlegen

l.splice(pos,ll);                               // ll vor pos einsetzen
// l:  7| 1 2 7 8 9 3 4 |
// ll: 0| |
```

Die Anforderungen beim Elementzugriff bestimmen die Wahl des Containers

Anforderung	Container
indizierter Zugriff	`vector, deque`
Einfügen/Löschen nur am Anfang oder Ende	`deque`
Einfügen/Löschen überall	`list`

hybride Anforderungen:

- anfängliche umfangreiche Dateneingabe an vorgegebenen Positionen
- später indizierter Zugriff notwendig

Lösung:

- Daten in `list`-Container einschreiben
- relevanter Iteratorbereich `[first,last)` umkopieren in `vector`-Container mittels Konstruktor `c(first,last);`

```
list<int> dieListe;
// Liste befuellen...

vector<int> derVektor(dieListe.begin(), dieListe.end());
// Jetzt befinden sich die Daten im Vektor
```