



- 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



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;  
}
```

- 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;
}
```



I.1a Vergleich der elementaren Datentypen

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

Größen sind abhängig von der Rechnerarchitektur

Typ	Speicherbedarf	Wertebereich
bool	1 Byte	0 oder 1

Manipulatoren für Wahrheitswerte	
(<u>no</u>)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;
}
```



Manipulatoren und spezielle Zeichen	
(no) <u>skipws</u>	Whitespace überspringen
left, <u>right</u>	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_____

```



Manipulatoren für ganze Zahlen	
<u>dec</u> , oct, hex	Zahlenformat
(<u>no</u>)showbase	Ausgabe der Zahlenbasis
(<u>no</u>)showpos	"+" mit ausgeben
(<u>no</u>)uppercase	Großbuchstaben für Hexzeichen
setbase(int)	Zahlenbasis festlegen
<u>left</u> , 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	
(<u>no</u>)showpoint	angehängten Dezimalpunkt/Nullen anzeigen
fixed, scientific	Fest-/Gleitpunktdarstellung
setprecision(int)	Genauigkeit angeben
<u>left</u> , 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.



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	bit
10	\^	l	
11		l	
12	\&\&	l	logisch
13		l	
14	?	r	ternär
15	= += -= *= /= %= &= ^= = <<= >>=	r	Zuweisung
16	throw		Ausnahme werfen
17	,	l	Sequenz



I.1a Strings

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

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

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

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

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



logisches Speicherabbild

front				back
-------	--	--	--	------

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

Anfügen/Entfernen	
x am Ende anfügen	<code>v.push_back(x);</code>
Element am Ende entfernen	<code>v.pop_back();</code>

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

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



```
// 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
```



```

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 Komponenten von neu
            // durch die Komponenten von person

```



`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
```



<code>strom.good()</code>	letzte Operation war fehlerfrei
<code>strom.fail()</code>	letzte Operation war fehlerhaft
<code>strom.bad()</code>	wie <code>fail()</code> , aber strom jetzt kaputt
<code>strom.eof()</code>	Dateiende ist erreicht worden
<code>strom.clear()</code>	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 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 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 Anhängen 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




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



Typ-Modifizierer — Fügt dem Datentyp eine Modifikation hinzu

<code>signed</code>	vorzeichenbehaftet
<code>unsigned</code>	vorzeichenlos
<code>short</code>	kürzer
<code>long</code>	länger

Typ-Qualifizierer — Fügt den Variablen neue Eigenschaften hinzu

<code>const</code>	Variable kann nach der Initialisierung nicht mehr verändert werden
<code>volatile</code>	Wert der Variablen kann durch Prozesse außerhalb des Programms verändert worden sein
<code>mutable</code>	Wert der Variablen kann durch  KONSTANTE KLASSENMETHODEN verändert werden

Speicherklassen-Spezifizierer — Lebensdauer und Sichtbarkeit

<code>auto</code>	Lebensdauer und Sichtbarkeit: aktueller Block
<code>static</code>	Lebensdauer: Programm; Sichtbarkeit: Block der Definition
<code>extern</code>	Lebensdauer und Sichtbarkeit: Programm



```
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 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.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. Änderungen 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. Änderungen  
// 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
```

Alle Operatoren in C++ sind als Operatorfunktionen implementiert

OPERATORÜBERLADUNG

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

```
<Rückgabety> 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`



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. $T \rightarrow \text{const } T$; Zuordnung eines Schablonentyps)	$\text{int} \rightarrow \text{int}$ $\text{int} \rightarrow \text{const int}$
2	Promotion	Werterhaltende Konversion	$\text{float} \rightarrow \text{double}$
3	Standardkonvertierung	Im allgemeinen nicht werterhaltende Konversion	$\text{int} \leftrightarrow \text{float}$
4	Benutzerdefinierte Konvertierung	Typumwandlungsoperator	
5	Übereinstimmung mit variabler Parameterliste	Parameter lässt sich in variable Parameterliste übergeben	

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
<input type="checkbox"/>	<code>f(double, long)</code>	1	3
<input type="checkbox"/>	<code>f(float, int)</code>	3	1
<input type="checkbox"/>	<code>f(float, long)</code>	3	3
<input checked="" type="checkbox"/>	keine Auswahl		

In Java wäre Kandidat 1 aufgerufen worden!



WELCHER KANDIDAT WIRD GEWÄHLT?

Aufruf: `f(i, 1)` mit `i: int`, `1: long`

	Kandidat	Parameter 1	Parameter 2
<input type="checkbox"/>	<code>f(double, long)</code>	3	1
<input type="checkbox"/>	<code>f(float, int)</code>	3	3
<input type="checkbox"/>	<code>f(float, long)</code>	3	1
<input checked="" type="checkbox"/>	keine Auswahl		

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

	Kandidat	Parameter 1	Parameter 2
<input type="checkbox"/>	<code>f(double, long)</code>	3	3
<input checked="" type="checkbox"/>	<code>f(float, int)</code>	3	2
<input type="checkbox"/>	<code>f(float, long)</code>	3	3
<input type="checkbox"/>	keine Auswahl		

Nur der zweite Aufruf hätte ebenfalls so in Java stattgefunden!

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 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



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 field[l];  
public:  
    static void methode();  
}
```

Test.cpp

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

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`

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?」



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 beinhaltet die Implementation auch
    // noch eine Gueltigkeitspruefung des uebergebenen Index i
};
```



```
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;
}
```



WARNUNG |

- Wenn Operatoren überladen werden, behalten Sie ihre Assoziativität und Priorität
- Die Operatoren `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `<<=`, `>>=` passen sich NICHT an, wenn die zugehörigen Operatoren `+`, `-`, `*`, `/`, `%`, `&`, `^`, `|`, `<<`, `>>` 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.



gegeben: Typ T

Klasse K

gesucht: Umwandlung $T \leftrightarrow K$

Lösung: $T \rightarrow K$ $K(\text{const } T\&);$ (Konstruktor)

$K \rightarrow T$ **operator** $T();$ (Typumwandlungsoperator)

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



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;
}
```



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

```
Rückgabotyp 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 überladenen 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); }
};

```



```

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;
}

```



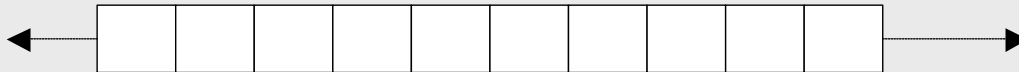

Container der STL - Sequentielle Container

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:





I.4 sequentielle Container

Bibliothek	vector	deque	list												
Beschreibung	dynamisches Feld	doppelseitige Schlange	Liste												
logisches Speicherabbild	front <table border="1"><tr><td></td><td></td><td></td><td></td></tr></table> back					front <table border="1"><tr><td></td><td></td><td></td><td></td></tr></table> back					front <table border="1"><tr><td></td><td></td><td></td><td></td></tr></table> back				
Konstruktion	<pre>vector<T> c; vector<T> c(n); vector<T> c(n,v); vector<T> c {...}; vector<T> c(d);</pre>	<code>deque<T> ~</code>	<code>list<T> ~</code>												
Anfügen/Entfernen	<pre>c.push_back(v); c.pop_back();</pre>	<pre>+c.push_front(v); +c.pop_front();</pre>	<code>~</code>												
Elementzugriff	<pre>v=c.front(); v=c.back(); v=c[i]; v=c.at(i);</pre>	<code>~</code>	<code>-c[i]; -c.at(i);</code>												
Globalzugriff	<pre>n=c.size(); c.resize(n); c.clear(); b=c.empty(); = ==, < (lex) ↪ !=, <=, >, >=</pre>	<code>~</code>	<pre>+c.unique([bp]); +c.remove(v); +c.remove_if(p); +c.sort([cmp]); +c.reverse(); +c.merge(d,[cmp]);</pre>												

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



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

```
list<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 |
```



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!);

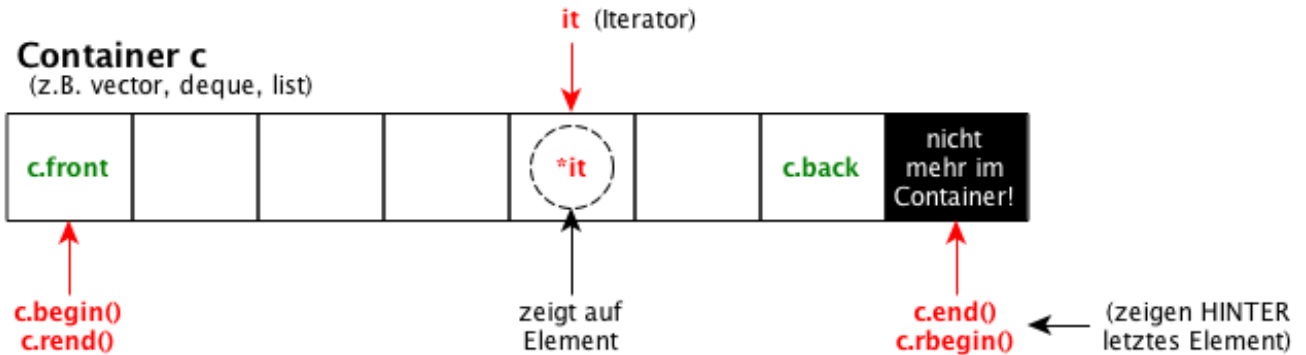
+ : zusätzlich; - : ohne

* : 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



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!



I.4 Iteratoren

Verwendung in Containermethoden

Funktionalität	Varianten	Beschreibung
Positionierung	<code>it=c.begin();</code> <code>it=c.end();</code> <code>r_it=c.rbegin();</code> <code>r_it=c.rend();*</code>	
	<code>it=c.insert(pos,e);</code> <code>c.insert(pos,n,e);</code> <code>c.insert(pos,first,last);</code>	fügt (n) Kopie(n) von e vor pos ein, bzw. die Sequenz** [first,last)***. it zeigt auf eingefügte Kopie
	<code>it=c.erase(pos);</code> <code>it=c.erase(first,last);</code>	löscht Element *pos bzw. Sequenz [first,last). it zeigt auf nachfolgende Position bzw. end()
Einsetzen	<code>c.splice(pos,liste);****</code> <code>c.splice(pos,liste, it);</code> <code>c.splice(pos,liste, first, last);</code>	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 ungültig
// 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 befüllen...  
  
vector<int> derVektor(dieListe.begin(), dieListe.end());  
// Jetzt befinden sich die Daten im Vektor
```



1 Zeiger und Felder

- Zeigervariablen
- Felder
- Zeigerarithmetik
- Vergleich: Zeiger vs Felder & Referenzen
- Anwendung: Online-Shop

2 dynamische Speicherverwaltung

- Speicheranforderung und Speicherfreigabe
- flache und tiefe Kopie
- Anwendung: Favoritenliste

3 Mehrdimensionale Felder



```

1 // Deklaration eines (nicht initialisierten!)
2 // double-Zeigers und einer double-Variablen
3 double *a, b;
4
5 a=&b;    // a zeigt jetzt auf b
6 *a=5;    // b=5
7 (*a)++;  // b=6
8
9 // Ein Zeiger auf Zeiger
10 double **c = &a;
11 cout << *c << " " << **c;
12 // gibt die Adresse von b gefolgt von dem
13 // aktuellen Wert von b aus
14
15 //! Nicht initialisierte Zeiger sind gefaehrlich
16 // Im folgenden Beispiel wird ein zufaelliger
17 // Speicherbereich durch den int-Wert 5 ueberschrieben.
18 int *d; //erzeugt Speicher fuer die Zeigervariable d,
19 *d=5;   //aber nicht fuer die Adresse, auf die d zeigt!
20 //! Der weitere Verlauf dieses Programme ist jetzt
21 //! unvorhersehbar!

```



Zur direkten byteweisen Speicher manipulation stehen typenlose Zeiger zur Verfügung. Der Grunddatentyp heißt **void** (leer) und entspricht einer Speicherstelle von 1 Byte Größe

void-Zeiger sind mit allen Datentypen kompatibel!

```
void *zeiger;  
double d;  
long int l;  
zeiger=&d;    // geht  
zeiger=&l;    // geht auch
```



Es gibt auch Zeiger, die wie Referenzen unverrückbar an ihre Zieladresse gebunden sind

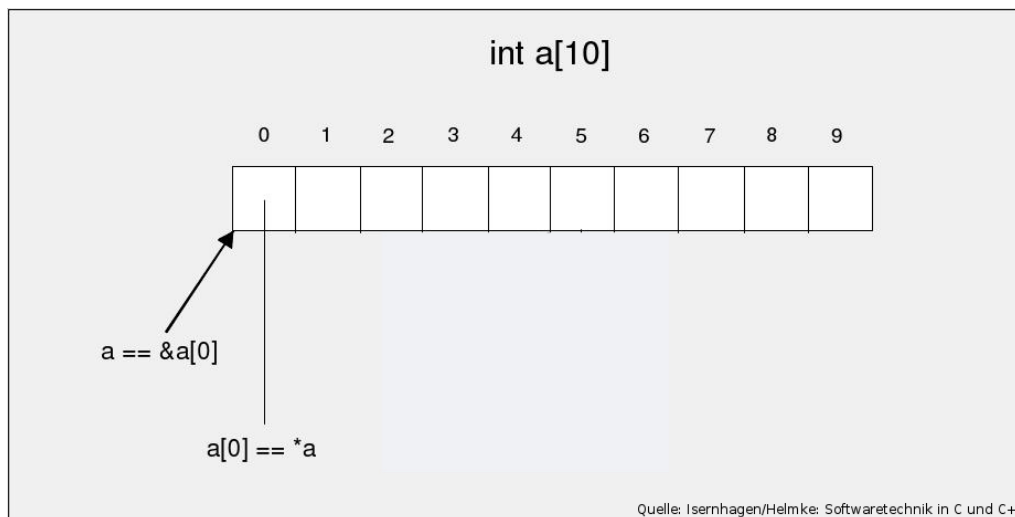
```
class K {};  
K k;  
K* const pK = &k; // pk wird unwideruflich  
                  // an das Objekt k gebunden  
//! Beachte: Das Schluesselfwort const steht  
//! hinter K*
```



II.1b Felder (Arrays)

Ein (sehr einfacher) Container

FELDER_ALSZEIGER



Felder

```
int a[10];           // reserviert einen zusammenhaengenden
                    // Speicherbereich fuer 10 integer-Werte

// a ist ein (konstanter) Zeiger auf das erste Feldelement
// syntaktisch aequivalenter Typ:
int* const b {a}; // hier erfolgt keine Speicherreservierung!

// Zugriff auf Feldelemente mittels des Indexoperators:
a[3] = 5; b[2] = 7;
```




- Felder lassen sich direkt bei der Vereinbarung mit Werten füllen
- Fehlende Initialwerte werden mit 0 vorbelegt
- Eine fehlende Dimensionierung wird aus der Initialisierungsliste festgelegt

```
/* vollstaendige Initialisierung */
```

```
int tage[6]={31,28,31,30,27,31};
```

```
/* unvollstaendige Initialisierung */
```

```
int n[5]={1,2,3}; /* n[3]=0, n[4]=0 */
```

```
/* Dimension durch Initialisierung festgelegt */
```

```
int tage[]={31,28,31,30,27,31};
```



Zeichenkettenkonstanten sind Felder vom Grundtyp **char**, die stets mit `'\0'` abschließen.

```
char text[8] = {'S', 't', 'r', 'i', 'n', 'g', '!', '\0'};  
char text[] = "String!"; //Dasselbe wie  
// char* text= "String!"; //dieses hier  
//! Beachte: text besteht wegen des abschließenden  
//! '\0' aus 8 Zeichen  
  
// "Hallo"[1] ist das Zeichen 'a'  
// "Hallo"[5] ist das Element '\0'
```

Lesehilfe

Deklarationen beim Variablennamen beginnend
von rechts nach links lesen

```
// konstante Zeiger
char s[20];           // s ist konstanter Zeiger
char* const t = s;    // t ist konstanter Zeiger
s++; t++;             // geht beides nicht

// konstanter Inhalt
const char* u = "Halli";
u[3]=s[3];            // geht nicht
u = s;                // geht!

// konstanter Zeiger auf konstanten Inhalt
const char* const v = "Hallo";
v[3]=s[3];            // geht nicht
v = s;                // geht auch nicht mehr
```



Auf die Klammern kommt es an

```
const int n=10;  
T *p;           // Zeiger auf T-Objekte  
T (*p)[n];      // Zeiger auf ein Feld von n T-Objekten  
T *p[n];        // Feld von n Zeigern auf T-Objekten
```

ALGORITHMUS ZUR ENTSCHLÜSSELUNG VON ZEIGERDEKLARATIONEN

- ① Beginn ist der Variablenname
- ② Interpretation von links nach rechts
- ③ Bei Erreichen einer schließenden runden Klammer wird von rechts nach links interpretiert
- ④ Bei Erreichen einer öffnenden runden Klammer „von rechts“ wird wieder von links nach rechts interpretiert, beginnend nach der zugehörigen geschlossenen runden Klammer
- ⑤ Ist das Ende des Ausdrucks erreicht, werden die noch verbliebenen Teile von rechts nach links interpretiert



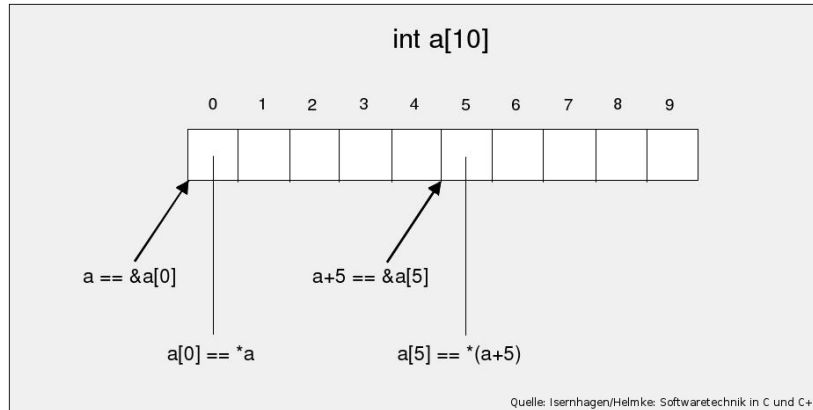
Zeigerarithmetik dient der typgerechten Verschiebung von Zeigerwerten und stellt ein effizientes Werkzeug zur Datenmanipulation im Speicher dar

```
T *p; int n;
```

- $p++$, $++p$, $p--$, $--p$ erhöht/erniedrigt die Adresse, auf die p zeigt, um $\text{sizeof}(T)$ Bytes.
- $p+n$, $p-n$ erhöht/erniedrigt die Adresse, auf die p zeigt, um $n \cdot \text{sizeof}(T)$ Bytes.
- Ist q ein weiterer Zeiger vom selben Grundtyp (T) wie p , so gilt:
 $p-q$ ist die Anzahl von Speicherplätzen vom Typ T , die zwischen p und q liegen. Der Abstand wird also in Einheiten von $\text{sizeof}(T)$ gemessen.



Feldnamen sind ja selbst Zeiger...



```
int a[10]; int *pa=a;
```

```
// Feld als Zeiger
```

```
a[i];    // ist dasselbe wie *(a+i); oder *(pa+i);
```

```
// Zeiger als Feld
```

```
pa[i];    // ist dasselbe wie a[i]
```

ACHTUNG

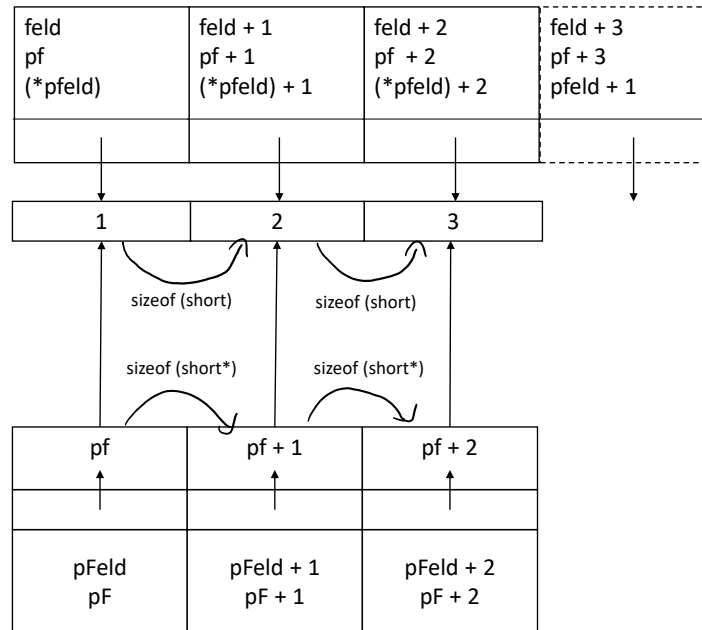
In Ausdrücken sind **a** und **pa** austauschbar. Tatsächlich ist **a** jedoch ein konstanter Zeiger, d.h. unlösbar mit der Adresse &a[0] verbunden.



```

const int n=3;
//
short feld[n] = {1,2,3};    // Feld von n short-Werten
short *pf = feld;           // Zeiger auf ersten short-Wert
short (*pfeld)[n] = &feld;  // Zeiger auf feld
//
short *pFeld[n] = {pf, pf+1, pf+2}; // Feld von short-Zeigern
short **pF = pFeld;          // Zeiger auf ersten short-Zeiger

```





ZEIGERVERSCHIEBUNG UND DEREFERENZIERUNG

Prio	Operator	Ass.	Bemerkung
1	++ -- () [] -> .	l	postfix
2	++ -- ! ~ + - * & sizeof (Typ)	r	präfix, unär

Wenn **p** eine Zeigervariable bezeichnet, dann ist...

- ***p++** = ***(p++)**, d.h. p wird inkrementiert, **nachdem (postfix!)** der dereferenzierte Inhalt von p geliefert wurde.
- **(*p)++** der Inhalt von p wird ausgelesen und inkrementiert. Der Zeiger p bleibt unverändert.
- ***++p** = ***(++p)** (rechts-assoziativ), d.h. der Zeiger p wird inkrementiert und der Inhalt der neuen Adresse wird ausgelesen.
- **++*p** = **++(*p)** (rechts-assoziativ), d.h. der Inhalt von p wird ausgelesen und inkrementiert. Der Zeiger p bleibt unverändert.

Fazit: Klammern setzen



- Felder sind konstante Zeiger auf das erste Feldelement und der Compiler reserviert bei der Vereinbarung automatisch einen zusammenhängenden Speicherbereich für das gesamte Feld
- Der Zugriff auf Feldelemente erfolgt entweder über den []-Operator oder nach den Regeln der Zeigerarithmetik (und anschließender Dereferenzierung)
- Die Zeigerarithmetik ist syntaktisch identisch mit der Nutzung von Random-Access Iteratoren für Container
- Bei der Übergabe von Feldern an Funktionen, ist der Formalparameter i.d.R. ein nicht-konstanter Zeiger vom kompatiblen Grundtyp. Damit kann mittels Zeigerarithmetik an beliebige Stellen im Feld gesprungen werden.
Die Information über die Länge geht dabei verloren
(und muss ggf. als weiterer Parameter übergeben werden)



- Eine Referenz ist ein Zeiger, der an einen existierenden Wert gebunden ist und diese Verbindung ist nicht lösbar
- Zeiger können überall hin zeigen und sind nicht an einen festen Wert gebunden (Iteratorenfunktionalität durch Zeigerarithmetik)
- Zeiger sind flexibler als Referenzen, aber auch fehleranfälliger, da der Programmierer verantwortlich dafür ist, dass die Zeigervariable auf einen definierten Wert zeigt
- Referenzen sind sicherer als Zeiger und syntaktisch angenehmer, da eine Referenz automatisch auf ihren Wert dereferenziert wird (kein Voranstellen von * notwendig)

Referenzen sind Zeigern stets vorzuziehen, wenn die zusätzliche Funktionalität der Zeigern nicht benötigt wird

11.2a new und delete

Wie bei Feldern kann man für beliebige Datentypen in einem Schritt Speicher anfordern, initialisieren und einen Zeiger auf die Anfangsadresse erhalten

Anforderung von Speicher (statt <code>int</code> kann ein beliebiger Datentyp eingesetzt werden)	
<code>int* pi;</code> <code>pi=new int;</code>	legt Speicher für eine <code>int</code> -Variable an und liefert dessen Speicheradresse zurück
<code>int* pi;</code> <code>pi=new int(5);</code>	wie oben; Wert wird mit 5 initialisiert / Konstruktoraufruf
<code>int* pfeld;</code> <code>pfeld=new int[10];</code>	zusammenhängender Speicher für 10 <code>int</code> -Variablen; 10 Standardkonstruktoraufufe; liefert Anfangsadresse zurück

Freigabe von <i>zuvor reserviertem</i> Speicher	
<code>delete pi;</code>	Destruktoraufruf für <code>*pi</code>
<code>delete[] pfeld;</code>	Destruktoraufruf für alle Feldelemente

- Ohne die (korrekte) Freigabe entstehen Speicherlecks!
- Die Lebensdauer von mit `new` erzeugten Objekten besteht bis zur Speicherfreigabe mit `delete`. Es ist darauf zu achten, dass der zugehörige Zeiger noch bis zu diesem Zeitpunkt 'lebt'
- `delete` auf einen Nullzeiger bewirkt nichts

```

class K {
    int n;
public:
    K() : n(0) { cout << "K_"; }
    K(int _n) : n(_n) { cout << "K(" << n << ")_"; }
    K(const K& k) : n(k.n) { cout << "KK(" << n << ")_"; }
    ~K() { cout << "D(" << n << ")_"; }
};

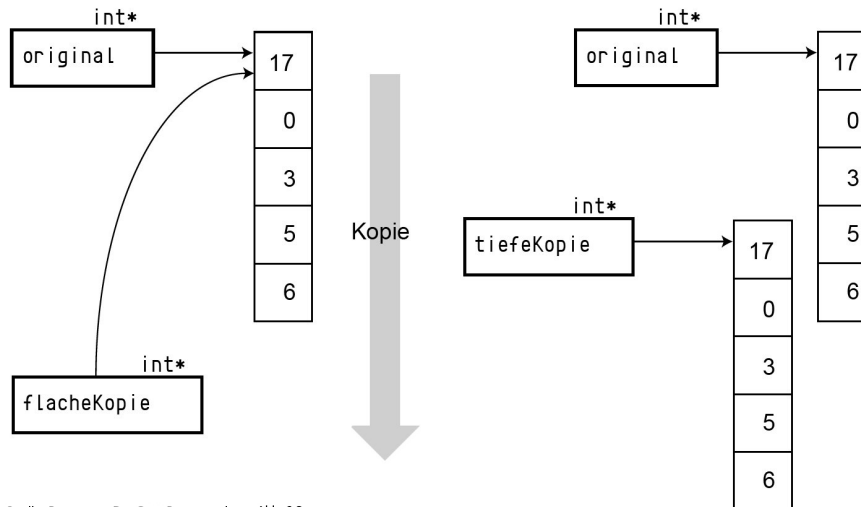
{
    // Speicher anfordern...
    K* pK_standard=new K; // —>K
    {
        K* pK_anweisungsblock=new K; // —>K
    }
    K* pK_zwei=new K(2); // —>K(2)
    K* pK_feld=new K[5]; // —>K K K K K
    vector<K> vK(5); // —>K K K K K

    // Speicher freigeben
    delete pK_standard; // —>D(0)
    //delete pK_anweisungsblock; // —>Speicherleck!!!
    delete pK_zwei; // —>D(2)
    delete [] pK_feld; // —>D(0) D(0) D(0) D(0) D(0)
    // ^— !!!

} // —>D(0) D(0) D(0) D(0) D(0)

```

- Eine **flache Kopie** kopiert sämtliche Datenkomponenten einer Klasse. Enthält die Klasse Zeiger, so werden lediglich der Zeiger kopiert
- Eine **tiefe Kopie** legt eine echte Inhaltskopie an, d.h. bei Bedarf wird passender Speicher angefordert und auch die Zeigerziele werden kopiert



Quelle: Breymann, Der C++ Programmierer, Abb. 6.2.

Beispiel: flache und tiefe Kopie eines durch einen Zeiger referenzierten Feldes



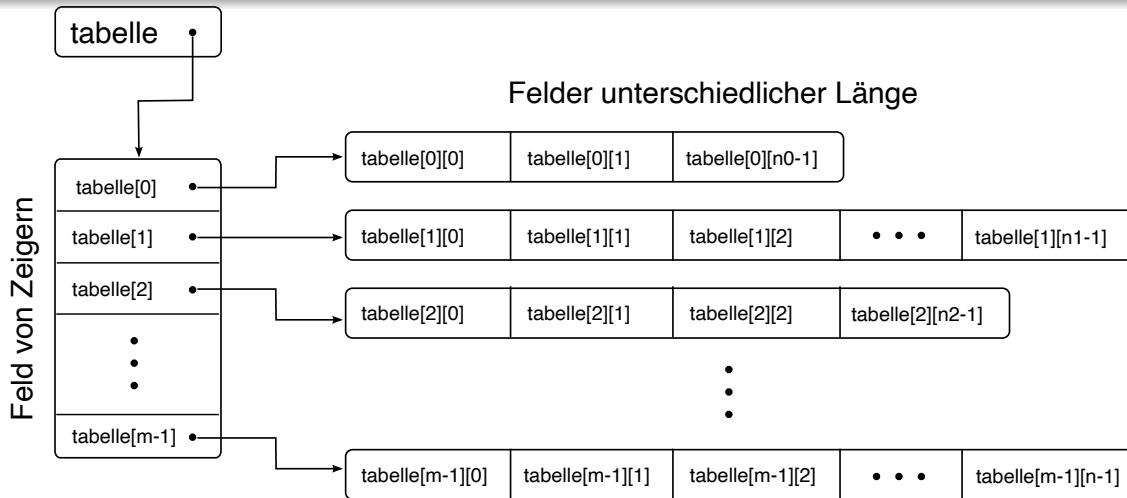
II.3 (mehrdimensionale) Felder und Zeiger

	Typ	syntaktisch äquivalenter Zeiger	Parameterliste in Funktionen
1D	T zeile[n]	T * const pZeile T pZeile[]	(T *pZeile, int n) (T pZeile[], int n)
1D-Felder sind konstante Zeiger auf das erste Element			
2D	T matrix[m][n]	T (* const pMatrix)[n] T pMatrix[][n]	(T (*pMatrix)[n], int m) (T pMatrix[][n], int m)
2D-Felder sind konstante Zeiger auf 1D-Felder vorgegebener Länge n			
3D	T quader[k][m][n]	T (* const pQuader)[m][n] T pQuader[][m][n]	(T (*pQuader)[m][n], int k) (T pQuader[][m][n], int k)
3D-Felder sind konstante Zeiger auf $m \times n$ 2D-Felder			

Erinnerung: Klammersetzung

T *p	ist ein Zeiger auf T-Objekte	Inkrement: sizeof(T)
T (*p)[n]	ist ein Zeiger auf ein Feld von n T-Objekten	Inkrement: n*sizeof(T)
T *p[n]	ist ein Feld von n Zeigern auf T-Objekten	Inkrement: sizeof(T*)

II.3 2D-Felder: variable Zeilenlänge



Typ	Parameterliste in Funktionen
<code>T **tabelle</code>	<code>(T** tabelle, int m, int *n)</code>

BEACHTEN

Die Zeiger `tabelle[i]` existieren physikalisch im Speicher. Sie enthalten die Startadressen, für die einzelnen Tabellenzeilen. Aufeinanderfolgende Tabellenzeilen müssen nicht notwendig hintereinander im Speicher abgelegt sein.



Obwohl bei Feldern manche "Zeiger" nur logisch existieren, unterstützen sie alle die gleiche Zeigerarithmetik, wie für Tabellen.

Lediglich die Adressen der nur logisch vorhandenen Zeiger stimmen in diesem Fall alle mit der Adresse des ersten Matrixeintrages überein

Zeiger	Grundtyp	Bedeutung des Grundtyps
$T **A$	T^*	Feld von Zeigern
$T *A[i]$	T	i-tes Feld von T-Werten

Ausdruck	Wert	Beschreibung
A	$\&A[0]$	Zeiger auf den Anfang des Zeigerfeldes ($A=\text{matrix}, p\text{Matrix}$: Zeiger auf Anfangselement)
$A+i$	$\&A[i]$	Zeiger auf den i-ten Zeiger im Zeigerfeld ($A=\text{matrix}, p\text{Matrix}$: Zeiger auf Anf.elem. i-te Zeile)
$*(A+i)$	$A[i]=\&A[i][0]$	Zeiger auf den Anfang der i-ten Tabellenzeile
$*(A+i)+j$	$A[i]+j=\&A[i][j]$	Zeiger auf Element j der i-ten Tabellenzeile
$((*A+i)+j)$	$A[i][j]$	j-tes Element der i-ten Tabellenzeile



1 Objekte in Beziehung

2 Vererbung

- Umsetzung in C++
- Wertesemantik: Auflösung von Namenskonflikten
- Zugriffsrechte
- Standardmethoden

3 Polymorphismus

- Umsetzung in C++
- Auswahlregeln bei virtuellen Methoden
- Standardmethoden und Polymorphie

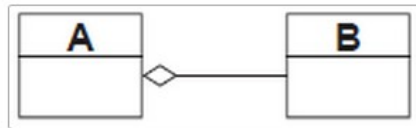
4 Abstrakte Klassen und Interfaces

- Abstrakte Klassen
- Interfaces
- Mehrfachvererbung

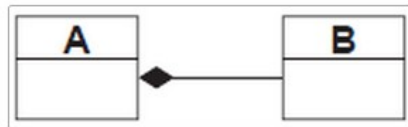
UML - Klassendiagramme



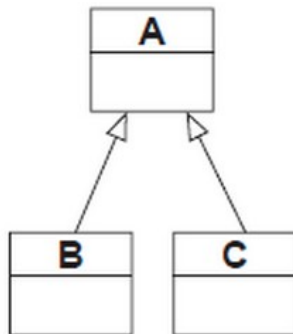
Assoziation



Aggregation



Komposition



Vererbung

Assoziation

Objekte, die miteinander in Beziehung stehen



Beziehung der Art

"benutzt ein/e"

"ist zugeordnet zu"

"hat eine Beziehung zu"

Beispiele A/B:

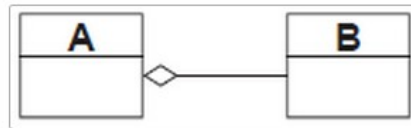
Mann/Frau

Person/Computer

Tafel/Kreide

Aggregation

Stärkere Beziehung als Assoziation, assoziiert Besitz



Beziehung der Art

"besitzt ein/e"

Beispiele A/B:

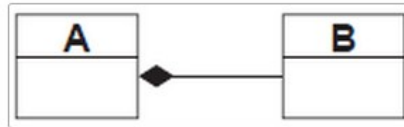
Auto/Fahrer

Restaurant/Kunde

Mannschaft/Spieler

Komposition

Sehr starke Beziehung, gleiche Lebenszeit



Beziehung der Art

"ist ein Teil von"

"besteht aus"

Beispiele A/B:

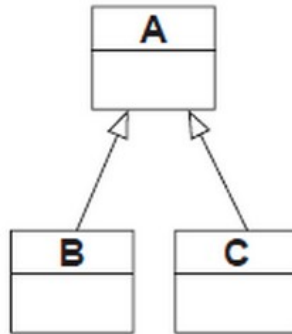
Mensch/Herz

Buch/Kapitel

Gebäude/Raum

Vererbung

Generalisierung, Spezialisierung



Beziehung der Art

"ist ein"

Beispiele A/B,C,...:

Fahrzeug/Auto, Bus, Bahn, ...

Beruf/Politiker, Professor, Maurer, ...

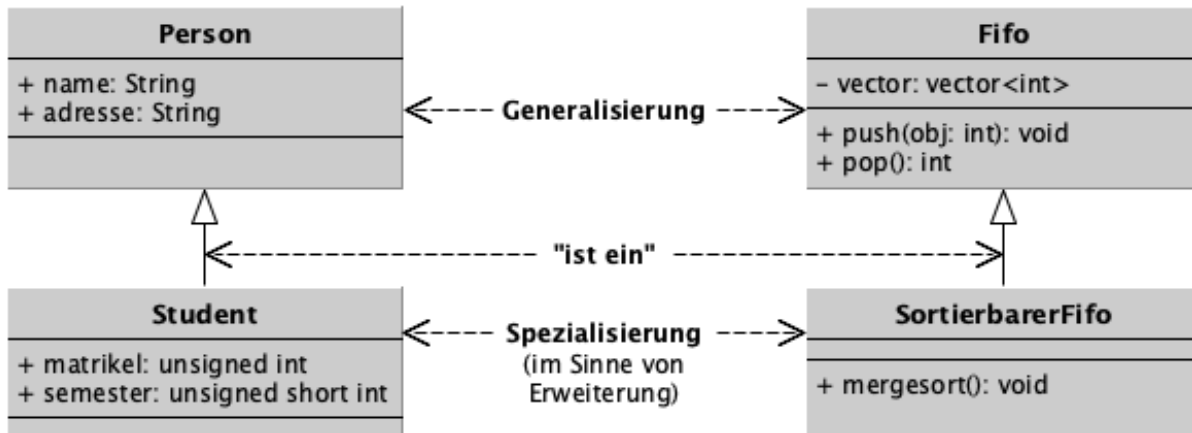
Tier/Vogel, Säugetier, Fisch...



III.1 Objekte in Beziehung

Vererbung: Sprechweisen

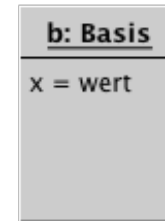
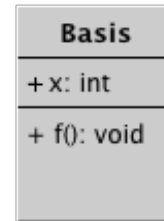
- Die abgeleitete Klasse **erbt** alle Daten/Methoden der Basisklasse
- Die abgeleitete Klasse **erweitert** die Basisklasse durch weitere Daten/Methoden
- Die Basisklasse hat eine kleinere Schnittstelle als die abgeleitete Klasse. Jedes Objekt der abgeleiteten Klasse realisiert *mindestens* diese Schnittstelle und kann daher überall dort stehen, wo auch ein Objekt der Basisklasse stehen könnte („**ist ein**“-Beziehung)



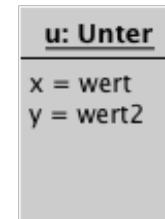
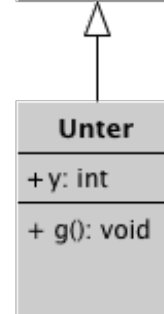


```
class Basis {
public:
    int x;
    void f() {}
};

class Unter: public Basis {
public:
    int y;
    void g() {}
};
```

SpeicherabbildMethoden

b.f();

u.f();
u.g();

```
void funcB(Basis b) { b.x=5; cout << "funcB(Basis)" << endl; }
void funcU(Unter u) { u.x=5; u.y=6; cout << "funcU(Unter)" << endl; }
```

```
funcB(b);      // OK
funcB(u);      // OK, denn "u ist auch ein b" (Upcast Unter → Basis)
//funcU(b);    // nicht OK, da keine Typkonvertierung Basis → Unter vorhanden
funcU(u);      // OK
//
//
```



Die Vererbungshierarchie wird beginnend bei der Klasse des aktuellen Objektes **von unten nach oben** nach dem Komponentennamen abgesucht und die erste gefundene Komponente wird gewählt.

Bei Methoden kommt es dabei zur **Überdeckung** aller gleichnamigen Bezeichner in höheren Ebenen unabhängig(!) von der Signatur

```
class Basis {
public:
    int x;
    void f() {}
    void g() {}
    void g(int) {}
    void g(double) {}
};

class Unter: public Basis {
public:
    int y;
    void g(float) {}
    void h() {}
};
```

```
Basis b;
b.x=1;
b.f();
b.g();
b.g(1);
b.g(1.5);

Unter u;
u.x=2;    // geerbt
u.y=3;
u.f();    // geerbt
//u.g();  // verdeckt!
u.g(1.5); // dasselbe wie:
u.g(static_cast<float>(1.5));
u.h();
```



Die Vererbungshierarchie wird beginnend bei der Klasse des aktuellen Objektes **von unten nach oben** nach dem Komponentennamen abgesucht und die erste gefundene Komponente wird gewählt.

Zugriff über die Vererbungshierarchie hinweg erfolgt durch explizite
Namensbereichauflösung

```
class Basis {
public:
    int x;
    void f() {}
    void g() {}
    void g(int) {}
    void g(double) {}
};

class Unter: public Basis {
public:
    int y;
    void g(float) {}
    void h() {}
};
```

```
Basis b;
b.x=1;
b.f();
b.g();
b.g(1);
b.g(1.5);

Unter u;
u.x=2;    // geerbt
u.y=3;
u.f();    // geerbt
u.Basis::g(); //!! "super.g()"
u.g(1.5); // dasselbe wie:
u.g(static_cast<float>(1.5));
u.h();
```



Die Vererbungshierarchie wird beginnend bei der Klasse des aktuellen Objektes **von unten nach oben** nach dem Komponentennamen abgesucht und die erste gefundene Komponente wird gewählt.

Überladung über mehrere Ebenen der Vererbungshierarchie hinweg wird durch die **using**-Direktive ermöglicht, die Bezeichner bekannt macht

```
class Basis {
public:
    int x;
    void f() {}
    void g() {}
    void g(int) {}
    void g(double) {}
};

class Unter: public Basis {
public:
    int y;
    using Basis::g; ///
    void g(float) {}
    void h() {}
};
```

```
Basis b;
b.x=1;
b.f();
b.g();
b.g(1);
b.g(1.5);

Unter v;
v.g();          /// jetzt moeglich
v.g(1.5);       /// Basis::g(double)
v.g(static_cast<float>(1.5));
                /// Unter::g(float)
```



Da es in C++ keine Paket-Sichtbarkeit gibt, sind Objektvariablen in der Regel durch **private** geschützt. Abgeleitete Klassen haben Zugriff auf **protected**-Objektvariablen

```
class Basis {  
private:  
    int privat;  
protected:  
    int geschuetzt;  
public:  
    int offen;  
  
    int gibPrivat() { return privat; }  
    int gibGeschuetzt() { return geschuetzt; }  
};  
  
/// Zugriff von "aussen"  
int i;  
Basis b;  
i = b.offen; i = b.gibGeschuetzt(); i = b.gibPrivat();
```



C++ erlaubt öffentliche, geschützte und private Vererbung mittels:

```
class Unter: public Basis { ... }; //öffentliche Vererbung
class Unter: protected Basis { ... }; //unueblich
class Unter: private Basis { ... }; //Implementationsvererbung
```

Der Zugriff von aussen auf die geerbten Komponenten wird auf höchstens **public** / **protected** / **private** eingeschränkt.

Bei nicht-öffentlicher Vererbung gilt:

Nur die Klasse Unter kann auf die öffentlichen/geschützten Komponenten von Basis zugreifen, erbt also insbesondere die **Implementationen** der Methoden

Man sagt: Unter "hat ein" oder "ist implementiert mit" Basis
Alternative: Delegation statt privater Vererbung stets möglich

WARNUNG | Aufpassen bei der Typkonvertierung

Der Compiler kennt keine Typkonvertierung Unter → Basis bei privater Vererbung (da Basis in Unter privat ist)



Delegation

Implementationsvererbung

```
class Warteschlange
: private list<int> {
public:
    // Abfragen
    using list<int>::empty;
    using list<int>::size;

    // am Ende einfuegen:
    void push(const int& x)
    { list<int>::push_back(x); }

    // am Anfang entnehmen:
    void pop()
    { list<int>::pop_front(); }

    // am Anfang bzw. Ende lesen
    using list<int>::front;
    using list<int>::back;
};
```

```
class Warteschlange {
private:
    list<int> liste;
public:
    // Abfragen
    bool empty()
    { return liste.empty(); }
    size_t size()
    { return liste.size(); }

    // am Ende einfuegen:
    void push(const int& x)
    { liste.push_back(x); }

    // am Anfang entnehmen:
    void pop()
    { liste.pop_front(); }

    // am Anfang bzw. Ende lesen
    const int& front()
    { return liste.front(); }
    const int& back()
    { return liste.back(); }
};
```



Default-Initialisierung

```

class Basis {
public:
    Basis() {}
    Basis(int) {}
    Basis(Basis&) {}
    ~Basis() {}
};

class Unter: public Basis {
public:
    Unter() {}
    Unter(int) {}
    Unter(Unter&) {}
    ~Unter() {}
};

Unter eins, zwei(0), drei(zwei);
//
// Aufrufe:
// Basis(), Unter()
// Basis(), Unter(int)
// Basis(), Unter(Unter&)

```

passende Initialisierung

```

class Basis {
public:
    Basis() {}
    Basis(int) {}
    Basis(Basis&) {}
    ~Basis() {}
};

class Unter: public Basis {
public:
    Unter() : Basis() {}
    Unter(int i) : Basis(i) {}
    Unter(Unter& u) : Basis(u) {}
    ~Unter() {}
};

Unter eins, zwei(0), drei(zwei);
//
// Aufrufe:
// Basis(), Unter()
// Basis(int), Unter(int)
// Basis(Basis&), Unter(Unter&)

```




Der Compiler kennt nur den statischen Typ von **pB** und die im statischen Typ vorhandenen Daten/Methoden!

```
class B {
public:
    int x;
};

class U : public B {
public:
    int y;
};

void f(B*) {}
void g(U*) {}

// statischer Typ: B
B* pB;

// statischer Typ: U
U* pU;
```

```
/// statischer Typ: B
pB = new B; // Zeigerziel: B-Objekt
pB->x=1;    // OK
f(pB);      // OK
//g(pB)     // nicht OK

pB = new U; // Zeigerziel: U-Objekt
pB->x=2;    // OK
//!pB->y=3; // nicht OK
f(pB);      // OK
//g(pB)     // nicht OK

/// statischer Typ: U
pU = new U; // Zeigerziel: U-Objekt
pU->x=4;    // OK: x geerbt
pU->y=5;    // OK
f(pU);      // OK: U* "ist ein" B*
g(pU);      // OK
```



Damit eine Methode in den abgeleiteten Klassen polymorph überschrieben werden kann, muss sie in der Basisklasse als **virtual** markiert werden

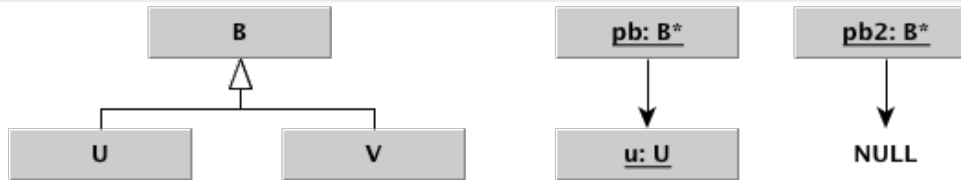
```
class Bank {  
    vector<Konto*> konto;    // heterogener Vektor-Container  
};                          // (Basisklassenzeiger)  
class Konto {  
    virtual void abheben(); // Methode polymorph markiert  
}
```

- `konto[347]—>abheben()` sucht eine passende Implementation der Methode `abheben()` im dynamischen Typ von `konto[347]`
- Polymorphie benötigt also sowohl Referenzsemantik (Methodenaufruf über (Basisklassen-)Zeiger oder Referenzen), als auch (in der Basisklasse vereinbarte) virtuelle Methoden
- Bei Wertesemantik haben virtuelle Methoden bei konventionellen Methodenaufrufen aus einem Objekt heraus (anstelle über einen Zeiger/Referenz) **keine** Bedeutung (—>Überdeckung)



III.3c Standardmethoden und Polymorphie

Kopie und Zuweisung — Kopie durch Erzeugung



ZIEL

Es soll eine Kopie von ***pb** typgerecht angelegt werden und **pb2** soll darauf zeigen

Implementation in **jeder Klasse K** (hier: **B**, **U**, **V**):

```

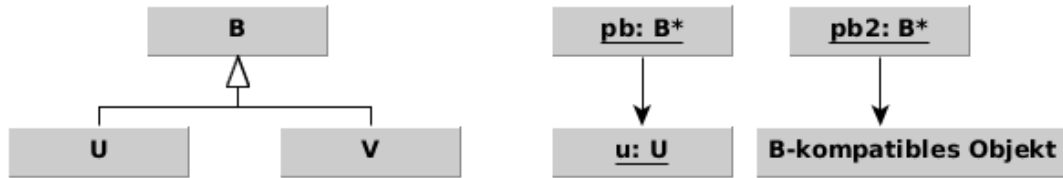
virtual K* clone() const { // Aufruf, wenn K=dynamischer Typ
    return new K(*this); // Kopierkonstruktor der Klasse K
} // (stets vorhanden, auch wenn nicht selbst geschrieben)
  
```

Aufruf durch: **pb2 = pb->clone();**



III.3c Standardmethoden und Polymorphie

Kopie und Zuweisung — Kopie durch Zuweisung



ZIEL

Es soll eine Kopie von **pb* typgerecht angelegt werden und *pb2* soll darauf zeigen

Implementation in *jeder Klasse K* (hier: *B*, *U*, *V*): 「Aufruf durch: **pb2* = **pb*」

```

K& operator=(const K& rhs) {
    if (this==&rhs) return *this;
    return assign(rhs); // polymorpher Aufruf
}

virtual K& assign(const B& rhs) { // feste Signatur!
    const K* pK = dynamic_cast<const K*>(&rhs);
    if (!pK) /* Ausnahme behandeln */ return *this;

    B::assign(rhs); // Aufruf direkte Elternklasse
    /* lokale Komponentenzuweisungen */
    return *this;
}
  
```



```

class B {
    int x;
public:
    B(int _x): x(_x) {}
    virtual ~B() {}
    virtual B* clone()
    { return new B(*this); }
    B& operator=(const B& rhs) {
        if (this==&rhs) return *this;
        return assign(rhs);
    }
    virtual
    B& assign(const B& rhs) {
        x=rhs.x;
        return *this;
    }
};

```

```

class U : public B {
    int y;
public:
    U(int _x, int _y)
    : B(_x), y(_y) {}

    virtual ~U() {}
    virtual U* clone()
    { return new U(*this); }
    U& operator=(const U& rhs) {
        if (this==&rhs) return *this;
        return assign(rhs);
    }
    virtual
    U& assign(const B& rhs) {
        const U* pU=
            dynamic_cast<const U*>(&rhs);
        if (!pU) return *this;
        B::assign(rhs);
        y = pU->y;
        return *this;
    }
};

```



Abstrakte Klassen sind Klassen, von denen keine Objekte angelegt werden können. Sie treten ausschließlich als Basisklassen auf.

BEISPIEL | Klasse Konto

Die Klasse Konto sollte keine Objekte haben, da es nur spezielle Ausprägungen (Sparkonto, Girokonto etc.) gibt.

MERKMALE ABSTRAKTER KLASSEN

- fassen allgemeine Eigenschaften ihrer abgeleiteten Klassen zusammen
- können als Zieltypen (statischer Typ) in Feldern/Vektoren von Basisklassenzeigern vorkommen und über den dynamischen Typ polymorph Ausprägungen der abgeleiteten Klassen repräsentieren
- stellen sicher, dass die abgeleiteten Klassen bestimmte Methoden anbieten (Schnittstellenanforderung)



```

// abstrakte Klasse
class B {
public:
    virtual void f()=0; // abstrakt durch rein virtuelle Methode
};                      // (kann, muss aber nicht implementiert werden)

// konkrete Klasse // muss die rein virtuellen Methoden
class U : public B { // implementieren
public:
    void f() { cout << "Durch_diese_Implementation"
                  << "_bin_ich_eine_konkrete_Klasse"
                  << endl;
    }
};

// B b;           // geht nicht, da B abstrakt
B *pb;           // geht

// pb = new B;    // geht nicht, da B abstrakt
pb = new U;       // geht

pb->f();           // Aufruf von f im zugehoerigen
                  // dynamischen Typ U

```



III.4a Abstrakte Klassen

Implementation in C++

KONTO_ABSTRAKT.CPP

```
class Konto { // abstrakte Klasse
public:
    virtual void abheben(int) = 0; // rein virtuelle Methode
    virtual ~Konto() {} // virtueller Destruktor
};

void Konto::abheben(int) { // Standardimplementation
    cout << "Konto_abheben_"; // (muss es nicht geben)
}

class Sparkonto : public Konto { // konkrete Klasse
public:
    void abheben(int); // muss es geben, wenn Objekte
                        // angelegt werden sollen
    void Sparkonto::abheben(int n) { // konkrete Implementation
        Konto::abheben(n);
        cout << "vom_Sparkonto" << endl;
    }

class Girokonto : public Konto { // konkrete Klasse
public:
    void abheben(int); // muss es geben, wenn Objekte
                        // angelegt werden sollen
    void Girokonto::abheben(int n) { // konkrete Implementation
        Konto::abheben(n);
        cout << "vom_Girokonto" << endl;
    }
}
```




Interfaces sind Schnittstellen-Anforderungsklassen, deren Aufgabe darin besteht, sicherzustellen, dass die abgeleiteten Klassen die vorgegebene Schnittstelle implementieren. Ein virtueller Destruktor ist auch hier nötig, da ein Interface auch als statischer Typ eines Zeigers genutzt werden kann.

IMPLEMENTATION IN C++

Abstrakte Klassen mit ausschließlich öffentlichen, rein virtuellen Methoden und ohne Daten

```
// Interface — abstrakte Klasse mit 100% rein virtuellen
class IstVergleichbar { // Methoden ohne Daten
public:
    virtual bool operator<(const IstVergleichbar&) const = 0;
    virtual ~IstVergleichbar() = 0;
};
```

```
// Implementierende Klasse
class MeineDaten : public IstVergleichbar {
    /* private Daten und Methoden */
public:
    bool operator<(const IstVergleichbar& rhs) const {
        const MeineDaten* pMeineDaten=
            dynamic_cast<const MeineDaten*>(&rhs);
        if (!pMeineDaten) { /* Werfe Ausnahme */ }
        /* Jetzt den Vergleich implementieren */
    }
};
```

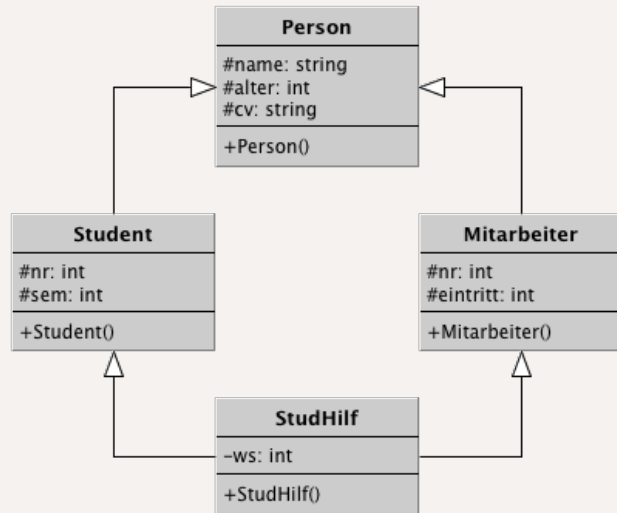
C++ erlaubt grundsätzlich das Erben von mehreren Basisklassen:

```
class U: public B1, public B2 { ... };
```

WARNUNG | Technische Schwierigkeiten!

Die Klasse **StudHilf** erbt

- Die Attribute **Student::nr** und **Mitarbeiter::nr**
- Zweimal (!) die Attribute der Klasse **Person**:
 - Student::name**,
Student::alter,
Student::CV
 - Mitarbeiter::name**,
Mitarbeiter::alter,
Mitarbeiter::CV



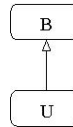
⇒ doppelte Datenhaltung!

「Workaround: „virtuelle“ Vererbung vererbt nur Zeiger auf Daten (unüblich)」

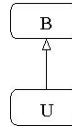


III.5 Handreichung: Vererbung

öffentliche Vererbung, private Vererbung und Delegation



- öffentliche Vererbung als "ist ein"-Beziehung
 - Jedes Objekt vom Typ U darf auch für ein Objekt vom Typ B stehen (insbesondere bei Parameterübergabe an Funktionen)
 - Die Unterklasse U erbt die Schnittstelle der Basisklasse und *erweitert* diese
- private Vererbung als "ist implementiert mit"-Beziehung
 - U kann nicht überall dort stehen, wo B erwartet wird
 - U erbt jedoch die Implementation der B-Methoden (Verwendung mittels using-Direktive oder Delegation)
- Delegation als "benutzt"-Beziehung
 - modelliert eine Klassenbeziehung, die nicht als öffentliche Vererbung im Sinne von Schnittstellenerweiterung besteht
 - Alternative zu privater Vererbung (durch "Benutzung" der benötigten Methoden)

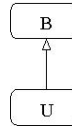


- Objektzugriff

- Komponenten*name* wird, beginnend beim aktuellen Objekt, von unten nach oben gesucht
- Überdeckung aller gleichnamigen Komponenten der in der Hierarchie "darüberliegenden" Klassen, unabhängig von der Signatur
- Überladung findet nur zwischen den gleichnamigen Methoden statt, die auf der "Ebene" liegen, in der die erste Namensübereinstimmung gefunden wurde
- "Ebenenübergreifende" Überladung kann mittels der using-Direktive realisiert werden

- Zeiger-/Referenzzugriff

- ermöglicht *generischen* Zugriff auf alle Objektinstanzen von B und seiner Unterklassen
- Komponentenauswahl gemäß statischen Objekttyp
- Auswahl der Methoden*implementation* gemäß dynamischen Objekttyp, wenn die Methode virtuell vereinbart wurde

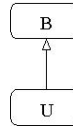


- gewöhnliche Methoden in B definieren eine *obligatorische Implementation*, die in der Unterklasse nicht überschrieben werden soll
- virtuelle Methoden definieren eine *Schnittstelle und eine Standardimplementation*, die die Unterklasse erbt und überschrieben werden *kann*
- rein virtuelle Methoden definieren *Schnittstelle* (ggf. eine Teil-Implementation), die in der Unterklasse implementiert werden *muss*
- abstrakte Klassen sind Klassen, die mindestens eine rein virtuelle Methode haben. Von Ihnen können keine Objekte angelegt werden, sie dienen als Blaupause für davon abgeleitete konkrete Klassen, die die Implementation zur Verfügung stellen.
Abstrakte Klassen sind allgemeine Konzepte, die jede andere (abgeleitete) Klasse der Klassenhierarchie konkretisiert
- Interfaces sind Klassen mit ausschließlich rein virtuellen Methoden ohne Daten. Sie fungieren als Anforderungen, Schnittstellen oder Eigenschaften für die von Ihnen abgeleiteten konkreten Klassen.
Mehrfachvererbung von Schnittstellen ist kein Problem



III.5 Handreichung: Vererbung

Polymorphie der Standardmethoden



- Konstruktoren sind nie virtuell, da sie durch den Klassennamen stets eindeutig sind
- Destruktoren können (als Ausnahme von der Namensregel) und sollten virtuell sein, damit Objekte der Unterklassen korrekt zerstört werden können
- Kopierkonstruktoren sind nach Namensregel nicht virtuell. Eine polymorphe Objektkopie kann durch eine einheitliche virtuelle Methode (z.B. clone()) erreicht werden
- Der Zuweisungsoperator kann nicht vererbt werden, stattdessen überdeckt jeder Zuweisungsoperator der Unterklassen die "darüberliegenden" Versionen. Polymorphie wird ermöglicht durch Aufruf einer einheitlichen virtuellen Methode
virtual U& assign(const B&) aus dem Standard-Zuweisungsoperator heraus und Nutzung von **dynamic_cast** zur Typprüfung des Operanden.