Geschichtliches

zu C

- Programmiersprache für allgemeine Anwendungen,
- $\approx$ 1972 entworfen und implementiert von Dennis Ritchie (Bell–Labs),
- anfänglich eng mit BS UNIX verbunden,
- 1978 erstes Manual: "The C–Programming–Language" (Kernighan, Ritchie),
- eine Weile "inoffizielle" Weiterentwicklungen,
- 1983 Einsetzen eines ANSI–Komitees zur Festlegung eines Standards, fertig 1989 (ANSI X3.159-1989),
- 1990 Übernahem des ANSI–Standards (mit kleinen Änderungen) als ISO–Standard (ISO/IEC 9899:1990),
- 1990 überarbeitete Version des Manuals "The C–Programming–Language" (Kernighan, Ritchie),
- relativ stabiler Standard,
- 1999 Überarbeitung und Modernisierung des Standards (ISO 9899:1999),
- 2000 Übernahme des neuen ISO–Standards als ANSI–Standard,
- neuer Standard wird nicht von "Allen" beachtet,
- de facto ist der 1989 Standard nach wie vor "der" Standard.

Geschichtliches

zu C++

- $\approx$ 1980 anfängliche Entwicklung durch Bjarne Stroustrup (Name: "C with classes"), Quelltexte werden in C–Quelltexte umgesetzt und anschließend mit C–Compiler übersetzt,
- 1982 eigentliche Geburtsstunde von C++ (Name)
- $\approx$ 1985 erste Compiler, erstes Buch "C++–Programming Language" von Bjarne Stroustrup
- 1989 Manual: "The Annotated C++–Reference Manual" (Ellis, Stroustrup)
- 1998 (nach jahrelanger Arbeit) ANSI/ISO–Standard (ISO/IEC 14882:1998)
- seither: Compilerhersteller versuchen, diesem Standard möglichst nahe zu kommen.

Literatur

[Eck 98] B. Eckel, *In C++ denken*, Prentice Hall, 1998

[Han 00] RRZN Hannover, *Die Programmiersprache C++ für C–Programmierer*, 11–Te Auflage, Regionales Rechenzentrum für Niedersachsen/ Universität Hannover, 2000

[ISO 98] ISO/IEC JTC1/SC22 Sekretariat, CD 14882: *Programming Language C++*, Veröffentlichung der Intarnational–Standards–Orgrnisation, 1998

[Jos 94] N. Josuttis, *Objektorientiertes Programmieren in C++*, Addison–Wesley, 1994

[Jos 96] N. Josuttis, *Die C++–Standardbibliothek*, Addison–Wesley, 1996

[Ker 90] B. W. Kernighan und D. Ritchie, *Programmieren in C*, 2–te Auflage, Carl Hanser, 1990

[Mey 92] S. Meyers, *Effective C++*, Addison–Wesley 1992

[Mey 96] S. Meyers, *More Effective C++*, Addison–Wesley 1996

[Pri 98] P. Prinz, U. Kirch–Prinz, *Objektorientiertes Programmieren in ANSI–C++*, Prentice Hall, 1998

[Str 94] B. Stroustrup, *The Design and Evolution of C++*, Addison–Wesley, 1994

[Str 98] B. Stroustrup, *Die C++ Programmiersprache*, 3–te Auflage, Addison–Wesley, 1998

Erstes C++–Programm

```
#include <iostream>

int main(void)
{
  std::cout << "hello, world" << std::endl;
  return 0;
}
```

– `#include <iostream>`
Information über Standard Ein–Ausgabe wird zur Verfügung gestellt.

– "Hauptfunktion":

```
int main(void)
{ ...
  return 0;
}
```

beim Programmaufruf wird diese Funktion ausgeführt.

Bedeutung:

`int`, `return 0;`: "Aufrufer" (BS) bekommt ganzzahliges Ergebnis (hier Wert 0),
`main`: Name der Hauptfunktion (immer),
`(void)`: (Haupt–)Funktion bekommt beim Aufruf vom Aufrufer (BS) keine Argumente
`{...}`: Anweisungsteil der (Haupt–)Funktion

– `std::` Zugriff auf im Standard definierte "Dinge":

`cout`: Standardausgabekanal (Bildschirm)
`endl`: Zeilenvorschub

– `<<`: Ausgabeoperator (geschachtelt aufrufbar)

– *Anweisungen* werden durch `';'` abgeschlossen!

Übersetzen von C++–Programmen

– Schreiben eines C++–Quelltextes (mit bel. Editor),
übliche Endungen von C++–Quelltexten: `.cc`, `.CC`, `cpp`, ...

– Übersetzten (Compilieren), auf LINUX–Systemen:

`g++ hello.cc`<cr>

(`hello.cc` sei der Name des Quelltextes)
wenn alles klappt, entsteht eine Datei mit dem Namen `a.out`

– `a.out` ist ein lauffähiges Programm (Maschinenbefehle der Maschine, für die übersetzt), kann direkt aufgerufen werden:

`a.out`<cr> (oder `./a.out`<cr>)

(keine "virtuelle" Maschine erforderlich, `a.out` ist allerdings maschinenabhängig!)

– ausführbare Datei kann umbenannt werden, etwa auf LINUX–Systemen:

`mv a.out hello`<cr> oder `mv a.out hello.exe`

und umbenannte Programme können ausgeführt werden:

`hello`<cr> bzw. `hello.exe`<cr>

– Übersetzen und Umbennenn in einem:

`g++ -o hello hello.cc`<cr>

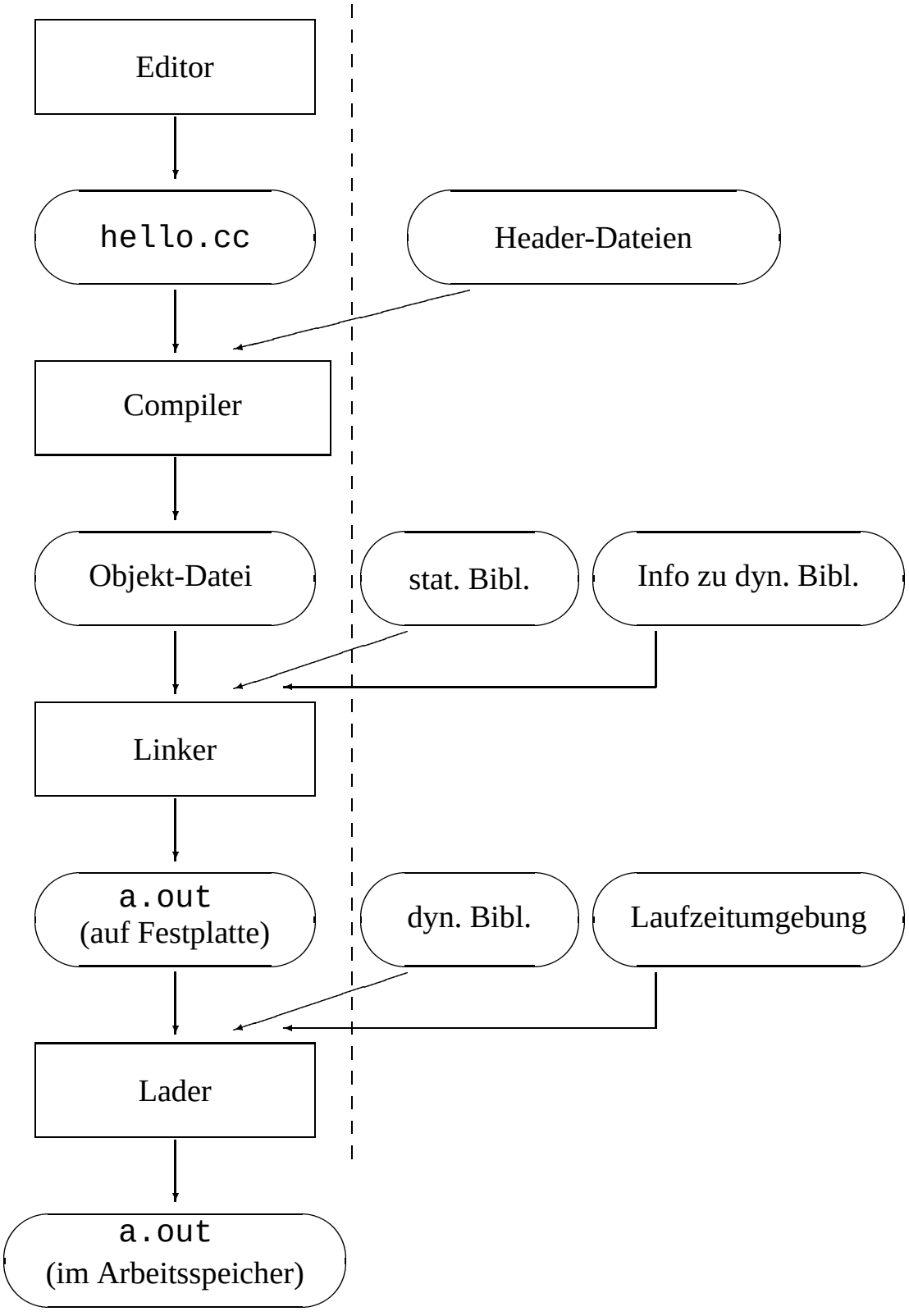
(zusätzliche Option: `-o` *Progrmmname*).

Vom Quelltext zum ablaufenden Programm:

Editieren:
`vi hello.cc`

Compilieren:
`g++ hello.cc`

Aufruf:
`a.out`<cr>



Präprozessor

Textersetzung vor dem eigentlichen Compilieren.

Präprozessoranweisung:

besteht aus einer ganzen Zeile, die mit dem #–Zeichen beginnt.

- **Dateien einbinden**:

  `#include <datei>` bzw. `#include "Datei"`

  Zeile wird durch Inhalt der angeg. Datei (*Header–Datei*) ersetzt.

  – `< ... >` : System–Header–Datei, steht in speziellen Systemverzeichnissen auf dem Rechner,
  – `" ... "` : eigene Header–Datei, steht im aktuellen Verzeichnis.

- **Makros**:

  `#define NAME Ersetzungstext`

  im Rest der Datei wird `NAME` jedesmal, wenn es als eigenständiges Wort auftaucht, durch den *Ersetzungestext* ersetzt.

  `#undef NAME`

  Makrodefinition für den Rest der Datei aufheben.

- **Bedingte Übersetzung**:

```
#if const_Ausdruck
...
#endif
```

  oder

```
#if const_Ausdruck
...
#else
...
#endif
```

etwa: **Auskommentieren eines (kommentierten) Quellcode–Teiles**:

```
#if 0
  ...
#endif
```

Häufige Bedingung:

`#if defined NAME` gleichbedeutend: `#ifdef NAME`

`#if !defined NAME` gleichbedeutend: `#ifndef NAME`

Anwendungen:

- mehrmaliges Einbinden ein– und derselben Header–Datei in **einem** Quelltext verhindern:

```
#ifndef _STRAFE_H_
#define _STRAFE_H_
  ...
  ...  // eigentlicher Inhalt der Header-Datei
  ...
#endif
```

- Systemabhängigkeiten “kapseln“:

```
#define LINUX
// #define WINDOWS
...
#ifdef LINUX
  ...  // Linux-spezifische Vereinbarungen
  ...  // und Einstellungen
#elif defined WINDOWS
  ...  // Windows-spezifische Vereinbarungen
  ...  // und Einstellungen
#else
  #error Kein System spezifiziert
#endif
```

Kommentare

```
#include <iostream>

/*************************************************
 * mein erstes C++-Programm                      *
 *                                               *
 * Wilhelm Hanrath      Datum: 5.3.2004          *
 *                                               *
 *************************************************/

int main(void)  // Kopfzeile der Hauptfunktion
{
  // Ausgabe der Begruessung auf dem Bildschirm:
  std::cout << "hello, world" << std::endl;

  // 0 ans Betriebssystem zurueckgeben:
  return 0;
}
```

- (mehrzeiliger) Kommentar: `/* ... */`
- Zeilenendekommentar: `// ...`

Variablen, Funktionen, Schleifen

```
#include <iostream>
#include <string>

/*********************************************
 * Variablen, eigene Funktion, Schleife      *
 *                                           *
 * Wilhelm Hanrath      Datum: 5.3.2004      *
 *********************************************/

// Bekanntmachung der eigenen Funktion:
void strafarbeit( int , std::string);

int main(void)
{ // Definition einer ganzzahligen Variablen:
  int anzahl;
  // Eingabeaufforderung:
  std::cout << "ganze Zahl eingeben: ";
  // ganzzahligen Wert einlesen:
  std::cin >> anzahl;

  // Definition einer Variablen vom Typ String:
  std::string veranstaltung;
  // Eingabeaufforderung:
  std::cout << "Name der Veranstaltung eingeben: ";
  // Namen lesen:
  std::cin >> veranstaltung;

  // Aufruf der eigenen Funktion:
  strafarbeit(anzahl, veranstaltung);

  return 0;
}

// Definition der eigenen Funktion:
void strafarbeit(int n, std::string name)
{ // Schleife: Anweisungsteil n mal durchfuehren:
  for ( int i = 0; i < n; ++i)
    std::cout << "Ich soll waehrend der " << name
```

```
                << " aufpassen!" << std::endl;

  return;
}

/* Ergebnis des Aufrufs:

ganze Zahl eingeben: 10
Name der Veranstaltung eingeben: C++-Vorlesung
Ich soll waehrend der C++-Vorlesung aufpassen!
Ich soll waehrend der C++-Vorlesung aufpassen!
Ich soll waehrend der C++-Vorlesung aufpassen!
Ich soll waehrend der C++-Vorlesung aufpassen!
Ich soll waehrend der C++-Vorlesung aufpassen!
Ich soll waehrend der C++-Vorlesung aufpassen!
Ich soll waehrend der C++-Vorlesung aufpassen!
Ich soll waehrend der C++-Vorlesung aufpassen!
Ich soll waehrend der C++-Vorlesung aufpassen!
Ich soll waehrend der C++-Vorlesung aufpassen!

*/
```

mehrere Quelltexte

1. **eigene Header–Datei** “`strafe.h`“:

```
#include <string>

// Bekanntmachung der Funktion:
void strafarbeit( int , std::string);
```

2. **Definitionsdatei für Funktion** (“`strafe.cc`“):

```
#include <iostream>
#include <string>

// Definition der eigenen Funktion:
void strafarbeit(int n, std::string name)
{ // Schleife: Anweisungsteil n mal durchfuehren:
  for ( int i = 0; i < n; ++i)
    std::cout << "Ich soll waehrend der " << name
              << " aufpassen!" << std::endl;

  return;
}
```

3. **Definitionsdatei für Anwendung** (“haupt.cc“):

```
#include <iostream>
#include <string>

// zur Bekanntmachung Header-Datei includen:
#include "strafe.h"

int main(void)
{ // Definition einer ganzzahligen Variablen:
  int anzahl;

  // Eingabeaufforderung:
  std::cout << "ganze Zahl eingeben: ";
  // ganzzahligen Wert einlesen:
  std::cin >> anzahl;

  // Definition einer Variablen vom Typ String:
  std::string veranstaltung;

  // Eingabeaufforderung:
  std::cout << "Name der Veranstaltung eingeben: ";
  // Namen lesen:
  std::cin >> veranstaltung;

  // Aufruf der eigenen Funktion:
  strafarbeit(anzahl, veranstaltung);

  return 0;
}
```

Übersetzungsmöglichkeiten

1. **alles zusammen**:

   `g++ haupt.cc strafe.cc`<cr>

   (Ergebnis: ausführbare Programmdatei `a.out`) oder

   `g++ -o anwendung haupt.cc strafe.cc`<cr>

   (Ergebnis: ausführbare Programmdatei `anwendung`),

2. **getrennt übersetzen und anschließend zusammenlinken**:

   (a) Übersetzen von `strafe.cc`:

   `g++ -c strafe.cc`<cr>
   es entsteht die *Objektdatei* `strafe.o`,

   (b) Übersetzen von `haupt.cc`:

   `g++ -c haupt.cc`<cr>
   es entsteht die *Objektdatei* `haupt.o`,

   (c) Zusammenbinden der Objektdateien:

   `g++ haupt.o strafe.o`<cr>

   (Ergebnis: ausführbare Programmdatei `a.out`) oder

   `g++ -o anwendung haupt.o strafe.o`<cr>

   (Ergebnis: ausführbare Programmdatei `anwendung`),

3. **oder gemischt** (in der Entwicklungsphase):

   (a) Quelltext ändern (etwa: `haupt.cc`),

   (b) geänderten Quelltext neu übersetzen:

   `g++ -c haupt.cc`<cr>

   (c) erneutes Zusammenbinden der Objektdateien, etwa:

   `g++ -o anwendung haupt.o strafe.o`<cr>

   (d) Alternative zu 2b) und 2c): Quelltext übersetzen und linken in einem:

   `g++ -o anwendung haupt.cc strafe.o`<cr>

Namensbereiche

using–Direktive

```
#include <iostream>

/***********************************************
 * Namespaces,                                 *
 * using-Direktive                             *
 *                                             *
 * Wilhelm Hanrath      Datum: 5.3.2004        *
 ***********************************************/

// using Direktive: ganzen Namensraum bekanntmachen:
using namespace std;

int main(void)
{
  // Ausgabe der Begruessung auf dem Bildschirm:
  cout << "hello, world" << endl;

  // 0 ans Betriebssystem zurueckgeben:
  return 0;
}
```

- alle Namen des Namensraumes `std` werden bekannt gemacht,
- keine explizite Qualifikation des Namensraumes mehr erforderlich,
- man kann mehrere Namensräume bekannt machen:

  ```
  using namespace A;
  using namespace B;
  ```

  (bei Konflikten explizite Qualifikation erforderlich)

eigener Namensbereich

1. **Namensbereich und zugehörige “Dinge“ deklarieren**
   (Datei: `strafe.h`):

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

// Definition des eigenen Namensbereiches:
namespace wh {
// Deklaration der Funktion:
void strafarbeit( int , string);

// weitere Deklarationen
}
```

2. **Definition der Dinge aus dem Namenbereich**:
   (etwa in Datei `strafe.cc`):

```
#include <iostream>
#include <string>

#include "strafe.h"
using namespace std;

// Definition der Funktion "strafarbeit"
// aus dem Namensbereich "wh"

void wh::strafarbeit(int n, string name)
{ for ( int i = 0; i < n; ++i)
    cout << "Ich soll waehrend der " << name
         << " aufpassen!" << endl;

  return;
}
```

3. **Anwendung**:

   (a) explizite Qualifikation des Namensbereiches:

```
...
wh::strafarbeit(10,"C++-Vorlesung");
...
```

   (b) `using`–Direktive zum ganzen Namensbereich:

```
...
using namesspace wh;
...
strafarbeit(10,"C++-Vorlesung");
...
```

   (c) `using`–Deklaration eines einzelnen “Dings“ aus dem Namensbereich:

```
...
using wh::strafarbeit;
...
strafarbeit(10,"C++-Vorlesung");
...
```

“eingebaute“ Typen

1. `bool` Datentyp zur Darstellung von Wahrheitswerten,

   auf unseren Systemen: 1 Byte

   typische Konstanten: `true`, `false` (Schlüsselworte)

   bel. Zahlwerte werden ggf. auch als Wahrheitswert interpretiert: $\neq 0$ entspricht *wahr* und $= 0$ entspricht *falsch*

2. `char` Datentyp zur Darstellung (einfacher) Zeichen (ASCII–Zeichensatz),

   auf unseren Systemen: 1 Byte

   `char`–Werte werden als kleine ganze Zahlen interpretiert (Nummer des Zeichens im Maschinenzeichensatz)

   zwei Ausprägungen:
   `unsigend char`: Werte von 0 bis 255
   `signed char`: Werte von -127 bis 127
   (`char` ist eine von beiden, compilerabhängig)

   typische Konstanten: `'A'`, oktal: `'\101'`, hexadezimal: `'\x41'`

3. `wchar_t` Datentyp zur Darstellung von Unicode–Zeichen.

   auf unseren Systemen: 4 Byte

   `wchar_t`–Werte werden als kleine ganze Zahlen interpretiert (Nummer des Zeichens im Zeichensatz)

   typische Konstanten: `L'A'`, oktal: `L'\101'`, hexadezimal: `L'\x41'` oder aber `L'ab'`

4. `short` Datentyp zur Darstellung kleiner ganzzahliger Werte

   auf unseren Systemem: 2 Byte

   zwei Ausprägungen:
   `unsigned short`: Werte von 0 bis 65 535
   `signed short`: Werte von $-32\,767$ bis 32 767
   (`short` entspricht `signed short`)

   `short`–Konstanten gibt es nicht!

5. `int` Datentyp zur Darstellung “normaler“ ganzzahliger Werte

   auf unseren Systemem: 4 Byte

   zwei Ausprägungen:
   `unsigned int`: Werte von 0 bis 4 294 967 295

`signed int`: Werte von $-2\,147\,483\,647$ bis $2\,147\,483\,647$
(`int` entspricht `signed int`)

typische `int`–Konstanten: `1234`, `1234u`

ist eine der Ausprägungen `signed` oder `unsigned` angegeben, kann das Schlüsselwort `int` fortgelassen werden!

6. `long` Datentyp zur Darstellung "großer" ganzzahliger Werte

   auf unseren Systemem: 4 Byte (wie `int`!)

   zwei Ausprägungen:
   `unsigned long`: Werte von 0 bis $4\,294\,967\,295$
   `signed long`: Werte von $-2\,147\,483\,647$ bis $2\,147\,483\,647$
   (`long` entspricht `signed long`)

   typische `long`–Konstanten: `1234l`, `1234ul`

7. `float` Datentyp zur Darstellung von Gleitkommawerten mit "geringer" Genauigkeit

   Speicher (bei uns):

insges	VZ.	Exponent	Mantisse
4 Byte	1 Bit	8 Bit	23 (+1) Bit

   Kenngrößen:

Exponent (dual):	$-126 \ldots 127$
`FLT_EPSILON`	$1.19209290 \cdot 10^{-7}$
`FLT_MIN`	$1.17549435 \cdot 10^{-38}$
`FLT_MAX`	$3.40282347 \cdot 10^{38}$

   typische `float`–Konstanten: `1.234f`, `-1e25f`

8. `double`: Datentyp zur Darstellung von Gleitkommawerten mit "normaler" Genauigkeit

   Speicher (bei uns):

insges	VZ.	Exponent	Mantisse
8 Byte	1 Bit	11 Bit	52 (+1) Bit

   Kenngrößen:

Exponent (dual):	$-1022 \ldots 1023$
`DBL_EPSILON`	$2.2204460492503131 \cdot 10^{-16}$
`DBL_MIN`	$2.2250738585072014 \cdot 10^{-308}$
`DBL_MAX`	$1.7976931348623157 \cdot 10^{308}$

   typische `double`–Konstanten: `1.234`, `-1e25`

9. `long double`: Datentyp zur Darstellung von Gleitkommawerten mit "hoher" Genauigkeit

   Speicher (bei uns):

insges	VZ.	Exponent	Mantisse
12 Byte	1 Bit	15 Bit	80 (+1) Bit

   Kenngrößen:

Exponent (dual):	$-16382 \ldots 16383$
`LDBL_EPSILON`	$1.0842021724855044340 \cdot 10^{-19}$
`LDBL_MIN`	$3.3621031431120935063 \cdot 10^{-4932}$
`LDBL_MAX`	$1.1897314953572317650 \cdot 10^{4932}$

   typische `long double`–Konstanten: `1.234L`, `-1e25L`

Typen 1 bis 6 heißen **integrale Typen**,

Typen 1 bis 9 heißen **arithmetische Typen**.

Der ASCII–Zeichensatz:

b7 b6 b5	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
Bits b4 b3 b2 b1	Steuer-zeichen		Symbole Ziffern		Groß-buchstaben		Klein-buchstaben	
0 0 0 0	0 NUL 0	16 DLE 10	32 SP 20	48 0 30	64 @ 40	80 P 50	96 ` 60	112 p 70
0 0 0 1	1 SOH 1	17 DC1 11	33 ! 21	49 1 31	65 A 41	81 Q 51	97 a 61	113 q 71
0 0 1 0	2 STX 2	18 DC2 12	34 ” 22	50 2 32	66 B 42	82 R 52	98 b 62	114 r 72
0 0 1 1	3 ETX 3	19 DC3 13	35 # 23	51 3 33	67 C 43	83 S 53	99 c 63	115 s 73
0 1 0 0	4 EOT 4	20 DC4 14	36 $ 24	52 4 34	68 D 44	84 T 54	100 d 64	116 t 74
0 1 0 1	5 ENQ 5	21 NAK 15	37 % 25	53 5 35	69 E 45	85 U 55	101 e 65	117 u 75
0 1 1 0	6 ACK 6	22 SYN 16	38 & 26	54 6 36	70 F 46	86 V 56	102 f 66	118 v 76
0 1 1 1	7 BEL 7	23 ETB 17	39 ’ 27	55 7 37	71 G 47	87 W 57	103 g 67	119 w 77
1 0 0 0	8 BS 8	24 CAN 18	40 (28	56 8 38	72 H 48	88 X 58	104 h 68	120 x 78
1 0 0 1	9 HT 9	25 EM 19	41) 29	57 9 39	73 I 49	89 Y 59	105 i 69	121 y 79
1 0 1 0	10 LF A	26 SUB 1A	42 * 2A	58 : 3A	74 J 4A	90 Z 5A	106 j 6A	122 z 7A
1 0 1 1	11 VT B	27 ESC 1B	43 + 2B	59 ; 3B	75 K 4B	91 [5B	107 k 6B	123 { 7B
1 1 0 0	12 FF C	28 FS 1C	44 , 2C	60 < 3C	76 L 4C	92 \ 5C	108 l 6C	124 | 7C
1 1 0 1	13 CR D	29 GS 1D	45 – 2D	61 = 3D	77 M 4D	93] 5D	109 m 6D	125 } 7D
1 1 1 0	14 SO E	30 RS 1E	46 . 2E	62 > 3E	78 N 4E	94 ^ 5E	110 n 6E	126 ~ 7E
1 1 1 1	15 SI F	31 US 1F	47 / 2F	63 ? 3F	79 O 4F	95 - 5F	111 o 6F	127 DEL 7F

Variablendefinition

- *Typname, Liste von Variablennamen (Trennzeichen: `','`), Strichpunkt*
- hinter jedem Variablenname kann `=` und ein initialisierender Ausdruck stehen

Ausdruck

Verknüpfung von **Operanden** (Variablen, Konstanten oder andere Ausdrücke) durch **Operatoren**.
Ein Ausdruck hat im allgemeinen einen **Typ** und einen **Wert**.
Es können Operanden von unterschiedlichem Typ verknüpft werden, → ganzzahlige Aufwertung, Typumwandlung von "klein" nach "groß".

C–Operatoren (in C++ gibt es noch ein paar mehr!)

Operator	Ass.
`()  []  ->  .`	lr
`!  ~  ++  --  +  -  *  &  (type)  sizeof`	rl
`*  /  %`	lr
`+  -`	lr
`<<  >>`	lr
`<  <=  >  >=`	lr
`==  !=`	lr
`&`	lr
`^`	lr
`|`	lr
`&&`	lr
`||`	lr
`?:`	rl
`=  +=  -=  *=  /=  %=  &=  ^=  |=  <<=  >>=`	rl
`,`	lr

Speicherklassen von Variablen

1. Speicherklasse `auto`

- Definition der Variablen innerhalb von `{ ... }` (Funktion oder Verbundanweisung),
- Variable wird **jedesmal neu erzeugt**, wenn `{ ... }` abgearbeitet wird (Variable ggf. mehrfach vorhanden → Rekursion),
- Variable wird am Ende von `{ ... }` **jedesmal zerstört**,
- Variable ist dem Compiler nur innerhalb von `{ ... }` bekannt (*Scope*),
- falls die Variable nicht explizit initialisiert wird, hat sie bei **jeder** Erzeugung einen "zufälligen" Wert,
- falls explizit initialisiert, darf der initialisierende Ausdruck beliebig sein, explizite Initialisierung wird bei **jeder** Erzeugung erneut durchgeführt.

Funktionsparameter verhalten sich (bis auf ihre Initialisierung) wie automatische Variablen, werden aber durch zug. Funktionsargumente initialisiert!

2. Speicherklasse `register`

wie automatische Variablen oder Funktionsparameter, Compiler wird jedoch "gebeten", die Variable in einem Maschinenregister der CPU zu halten, nicht im Arbeitsspeicher (ggf. Geschwindigkeitsvorteil).

- nur für wenige, kleine Variablen möglich,
- Compiler muss der Bitte nicht nachkommen,
- register–Variablen haben **keine Adresse** (Adress–Operator nicht anwendbar!) (unabhängig davon, ob Compiler der Bitte nachkommt oder nicht!)

3. Speicherklasse `extern`

- Definition der Variablen außerhalb jeder Verbundanweisung (`{ ... }`) (außerhalb jeder Funktionsdefinition),
- Variable wird beim Start des Programms erzeugt,
- Variable wird erst beim Programmende zerstört,
- Variable ist bekannt von ihrer Definition
  - **bis zum Ende des Quelltextes** (in allen Funktionen)
  - zusätzlich überall dort, wo sie nochmals **deklariert** wird (auch in anderen Quelltexten!)
    Deklaration sieht wie Definition aus, zusätzlich jedoch Schlüsselwort `extern`, bei Deklaration ist kein initialisierender Ausdruck möglich!
    Deklaration kann lokal oder global sein, auch in Header–Dateien möglich!
    (Definition in Header–Dateien unsinnig!)
- wenn nicht explizit initialisiert, sind alle Bytes mit 0 initialisiert,
- falls explizit initialisiert, muss der initialisierende Ausdruck ein **konstanter Ausdruck** sein.

externe Variablen: global für alle Funktionen, auf die Variable können alle Funktionen zugreifen (gefährlich in großen Programmpaketen!).

eine gleichnamige automatische Variable "überdeckt" die externe Variable!

Gültigkeitsbereichsauflösungsoperator: `::`

Zugrif auf **globale** Variable, falls gleichnamige lokale Variable vorhanden ist.

4. statische lokale Variablen

- wie automatische Variablen innerhalb von `{ ... }` definiert, zusätzlich mit Schlüsselwort `static`,
- Variable wird beim Start des Programms erzeugt,
- Variable wird erst beim Programmende zerstört,
- Variable ist jedoch **nur innerhalb** von `{ ... }` bekannt,
- wenn nicht explizit initialisiert, sind alle Bytes mit 0 initialisiert,
- falls explizit initialisiert, muss der initialisierende Ausdruck ein **konstanter Ausdruck** sein.
- Initialisierung wird **einmal** beim Programmstart vorgenommen,
- Variable und ihr Wert bleibt von Aufruf zu Aufruf der Funktion erhalten.

5. statische globale Variablen

- wie externe Variablen, zusätzlich aber bei der Definition Schlüsselwort `static`,
- verhält sich bzgl. Erzeugung, Zerstörung und Initialisierung wir externe Variablen,
- Bekanntheit (*Scope*) der Variablen ist jedoch auf den einen Quelltext (ihrer Definition) eingeschränkt, kann in anderen Quelltexten nicht deklariert werden!
- kann den gleichen Namen wie eine externe Variable eines anderen Quelltextes (überdeckt diese externe Variable!).

gewöhnliche Verwendung von Namensbereichen:

1. Definition des Namensbereiches und **Deklaration** der enthaltenen “Dinge“: (üblicherweise in einer Header–Datei, etwa `"strafe.h"`)

```
...
namespace wh {
void strafarbeit( int , string);
extern int mincount;
}
```

2. **Definition** der im Namensbereich enthaltenen Dinge: (nur einmal in einer Definitionsdatei, etwa `"strafe.cc"`)

```
...
#include "strafe.h"

int wh::mincount = 5;

void wh::strafarbeit(int n, string name)
{ if ( n <= wh::mincount )
    n = wh::mincount;

  for ( int i = 0; i < n; ++i)
    cout << "Ich soll waehrend der " << name
         << " aufpassen!" << endl;
  return;
}
```

3. Anwendung:

```
...
using namespace wh;
...
strafarbeit(10, "C++-Vorlesung");
...
```

Unbenannte Namensbereiche

Definition eines Namensbereiches **ohne** Namen und gleichzeitig **Definition** der enthaltenen “Dinge“ im Namensbereich.
Definierten “Dinge“ sind

- im **Quelltext** durch einfachen Namen ansprechbar,
- **nur in diesem Quelltext** bekannt!

```
...
namespace {
  int mincount = 5;

  void strafarbeit(int n, string name)
  { if ( n <= mincount )
      n = mincount;

    for ( int i = 0; i < n; ++i)
      cout << "Ich soll waehrend der " << name
           << " aufpassen!" << endl;
    return;
  }
} // Ende der Definition des Namensbereiches

// Anwendung

int main(void)
{
  ...
  strafarbeit(10, "C++-Vorlesung");
  ...
}
...
```

Weitere Techniken zu Namensbereichen

1. Schachteln von Namensbereichen:

```
namespace A {
  namespace B {
    ...
    int fkt(int);
    ...
  }
  ...
}
// Definition:
int A::B::fkt(int n) { ... }
...
```

2. Alias–Namen für (lange) Namen eines Namensbereiches:

```
namespace DV_Labor_FB_8_FH_Aachen_Prj_02_04 {
...
}

namespace FH_AC=DV_Labor_FB_8_FH_Aachen_Prj_02_04;

using FH_AC::...;
...
```

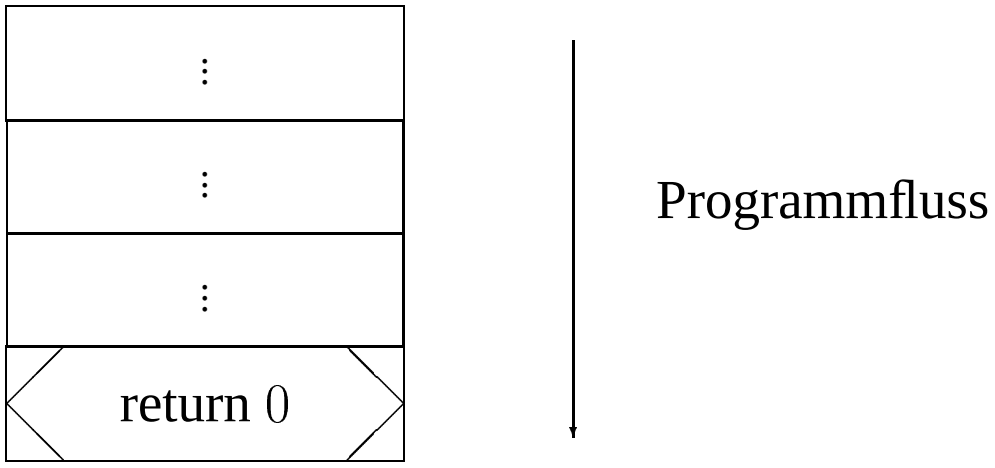
3. Schnittstellen zusammenstellen:

```
namespace Schnittstelle {
  using namespace A;
  using namespace B;
  using C::...;
  using D::...;
}

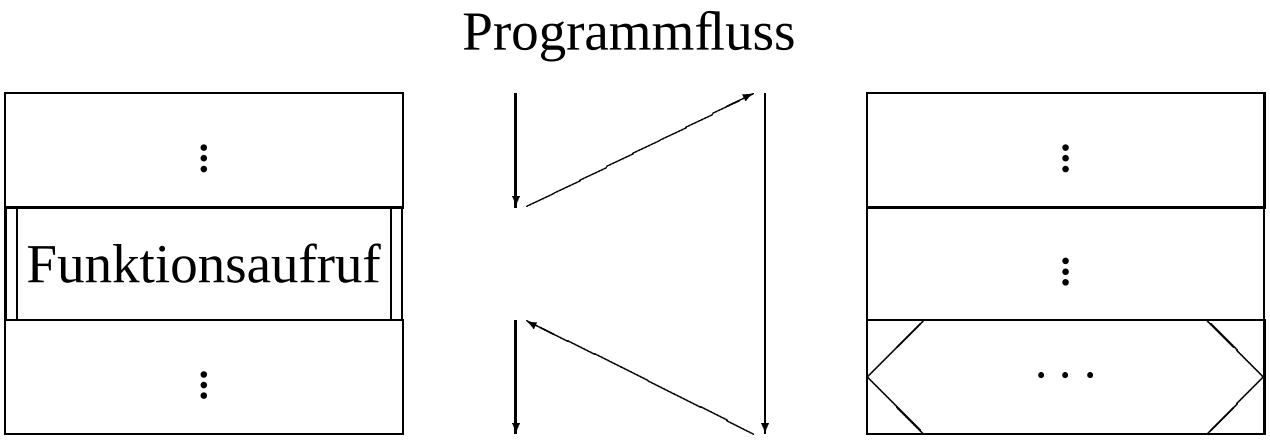
using namespace Schnittstelle;
...
```

Ablauf

1. Programmaufruf: Abarbeitung der Hauptfunktion `main`
   - Anweisungen werden der Reihe nach ausgeführt:



   - falls Funktion aufgerufen wird, werden deren Anweisungen der Reihe nach abgearbeitet, anschließend in aufrufender Funktion weiter:



2. Auswahlanweisungen

- `if`–Anweisung:

  `if(`*Bedingung*`)`
  Anweisung

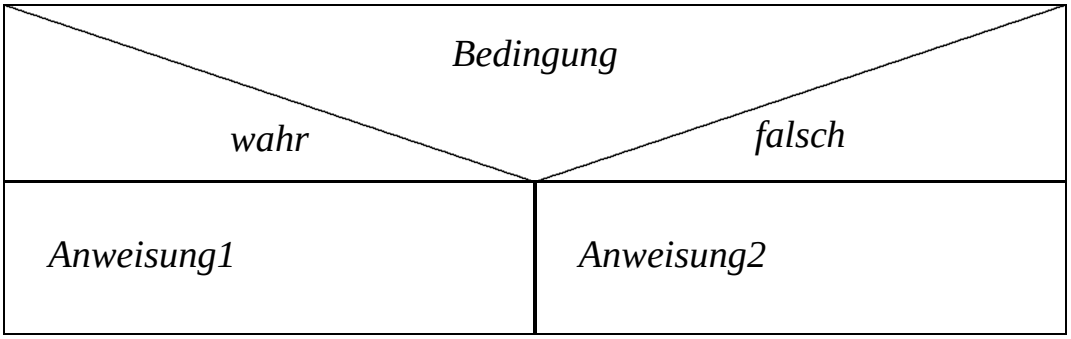
  Bedinung: etwas, was als *wahr* oder *falsch* interpretiert werden kann (auch Zahlen!)

  Anweisung: einfache Anweisung, aber auch Verbundanweisung `{ ... }` (oder auch andere Kontrollanweisung)

- `if-else`–Anweisung:

  `if(`*Bedingung*`)`
  Anweisung1
  `else`
  Anweisung2

Struktogrammelement:



3. Wiederholungsanweisungen (Schleifen)

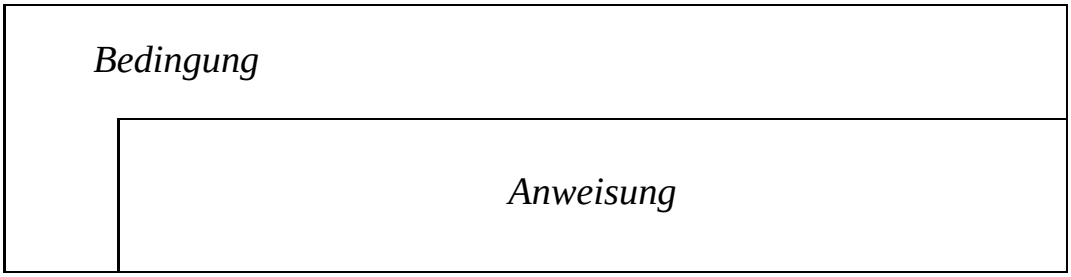
- kopfgesteuerte Schleifen:
  - `for`–Schleife:

    `for(`*Initialisierung*`;`*Bedinung*`;`*Inkrementierung*`)`
    Anweisung

  - `while`–Schleife:

    `while(`*Bedingung*`)`
    Anweisung

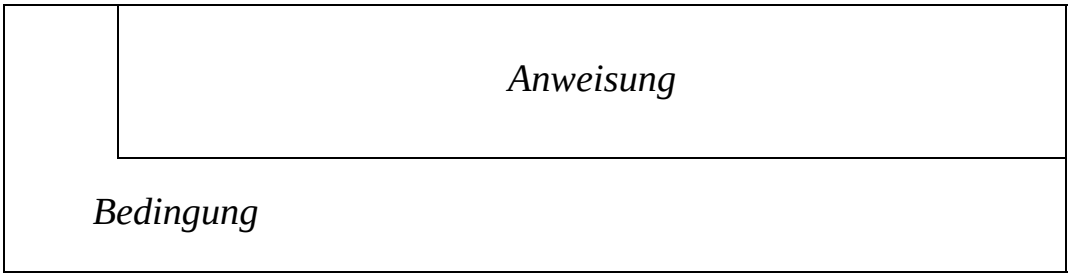
  Struktogrammelement:



- fußgesteuerte Schleifen:
  - `do-while`–Schleife:

    `do {`
    Anweisung
    `} while(`*Bedingung*`);`

    Struktogrammelement:



- Sonderanweisungen `break` und `continue`

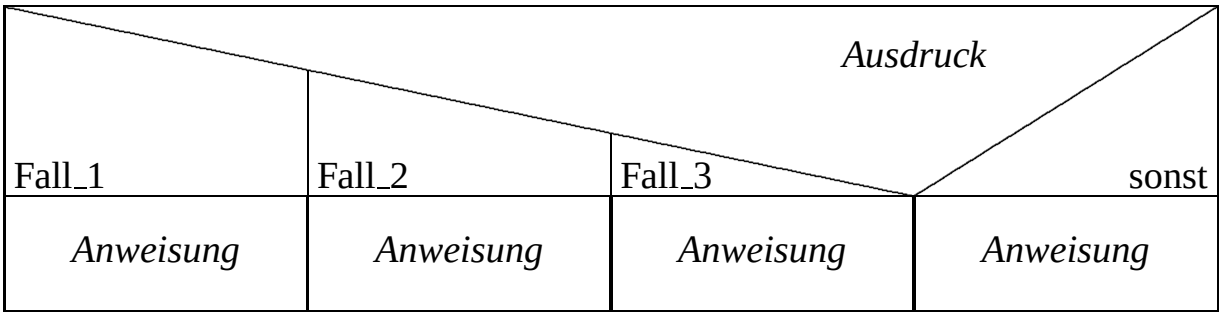
4. `switch`–Anweisung

```
switch ( ausdruck )
{ case k_ausdr1: Anweisung;
                 Anweisung;
  case k_ausdr2: Anweisung;
                 Anweisung;
  case k_ausdr3:
  case k_ausdr4: Anweisung;
                 Anweisung;
        default: Anweisung; // optional
                 Anweisung;
}
```

mit `break` (Fallunterscheidung zwischen mehreren Fällen):

```
switch ( ausdruck )
{ case Fall_1: Anweisung;
               break;
  case Fall_2: Anweisung;
               break;
  case Fall_3: Anweisung;
               break;
      default: Anweisung; // optional
}
```

Struktogrammelement:



Felder

- **Definition:**

  Typ name`[`*Feldlänge*`];`

  oder mit expliziter Initialisierung:

  Typ name`[`*Feldlänge*`] = {`*Wert_1, Wert_2,...*`};`

  - *Feldlänge* muss ein konstanter Ausdruck sein (Wert muss zur Compilierzeit festliegen!)
  - Indizierung beginnt mit "0", endet bei "*Feldlänge* − 1"
  - Zugriff auf Feldelemente ist ungeprüft!!!
  - lokal, global, statisch möglich (hat Einfluß auf Erzeugung, Initialisierung, Sichtbarkeit, Zerstörung!)
  - Arbeiten mit Feldern: nur mittels Elementzugriff
    (Zuweisungsoperator = für Felder nicht definiert!)

- **Deklaration** eines globalen Feldes:

  `extern` *Typ name*`[];`

  (bei Deklaration **keine** Feldlänge, **keine** Initialisierung)

C–Strings

Felder vom Typ `char` mit abschließendem *Stringendezeichen* `'\0'`, etwa:

```
char Wort[16] = {'h', 'a', 'l', 'l', 'o', '\0' };
```

'h'	'a'	'l'	'l'	'o'	'\0'										

dient zum Abspeichern von Zeichenketten, Zeichenkette kann kürzer sein als Feldlänge!

Alternative Definition und Initialisierung:

```
char Wort[16] = "hallo";
```

Zeiger

Jede Variable hat:

- **Namen**: mit dem sie in ihrem Geltungsbereich angesprochen werden kann!
- **Typ**: legt fest,
  - wieviel Speicher für die Variable im Arbeitsspeicher reserviert wird,
  - welche Oparationen mit der Variable möglich sind.
- **Wert**: im Arbeitsspeicher abgespeichertes Bitmuster, interepretiert im zugrundeliegenden Typen.
- **Bereich im Arbeitsspeicher**

  Der Arbeitsspeicher kann als riesige, durchnummerierte Folge einzelner Bytes (8 Bits) angesehen werden!
- Jede Variable hat eine **Adresse**, das ist die Nummer des ersten zur Variablen gehörenden Byte im Arbeitsspeicher!

Adressoperator &

liefert zu einer Variablen (außer `register`–Variablen) deren Adresse (Nummer des ersten Bytes der Variablen, also einen ziemlich großen ganzzahligen Wert)
Adressen sind **typgebunden**

```
...
int i;
double x;
...
...&i...;     // Adresse einer int-Variablen
...&x...;     // Adresse einer double-Variablen
...
```

Adressvariablen (Zeiger)

- Variablen, die als Wert die Adresse einer anderen Variablen (eines ganz bestimmten Types) aufnehmen können.
- bei Definition der Variablen ist dem Namen ein * voranzustellen

```
int i, *ip;         // i  ist int-Variable,
                    // ip ist int-Adress-Variable
double x, y, *dp;   // x und y sind double-Variablen,
                    // dp ist double-Adress-Variable
ip = &i;            // ok, ip bekommnt als Wert die
                    // Adresse der int-Variablen i
dp = &x;            // ok, dp bekommnt als Wert die
                    // Adresse der double-Variablen x

ip = &y;            // FEHLER:
                    // ip ist int-Adress-Variable
                    // und &y ist double-Adresse
                    // FALSCHER Typ!

dp = &y;            // ok, dp bekommt neuen Wert
```

typlose Adressvariablen

```
void *p;
```

kann beliebige Adresse aufnehmen (allerdings wird der Typ dessen vergessen, "auf was gezeigt wird")

```
int i;
double x;

p = &i;     // ok, allerdings reine Adresse
p = &x;     // ok, allerdings reine Adresse
```

Verweisoperator *

- anwendbar auf Adressen (**außer**: typlosen Adressen `void *`)
- liefert die Variable des zugrundeliegenden Types, welche an der entsprechenden Adresse im Arbeitsspeicher liegt!

```
...
double x, y, *dp;  // zwei double, ein double-Zeiger
int i, *ip1, *ip2; // ein int, zwei int-Zeiger
void *p;           // typloser Zeiger
...
dp = &x;         // dp zeigt auf x
cin >> *dp;      // *dp "ist" x; x wird gelesen
...
dp = &y;         // dp zeigt jetzt auf y
*dp = 3.14;      // *dp "ist" y; y bekommt Wert 3.14
...
ip1 = &i;        // ip1 zeigt auf i
cout << *ip1;    // *ip1 "ist" i, i wird ausgegeben
...
ip2 = ip1;       // ip2 zeigt jetzt auch auf i
*ip2 = 27;       // *ip2 "ist" i, i bekommt Wert 27
...

p = &i;          // ok, p bekommt Adresse von i
                 // *p NICHT ERLAUBT!!
p = &x;          // ok, p bekommt Adresse von x
                 // *p NICHT ERLAUBT!!
...
```

ungültige Adresse

– in C: symbolische Konstante `NULL`,

– in C++: besser: Konstante `0`

Zeiger und Felder, Rechnen mit Adressen

- `T` irgendein Typ
- `T *p;` Adress–Variable dieses Types

Hat `p` einen Wert, so geht das System geht davon aus, dass `p` auf Element eines Feldes vom Typ `T` zeigt!
Dann gilt etwa:

- `p+1` zeigt auf das folgende Feldelement,
  `*(p+1)` "ist" das nächste Feldelement
- `p-1` zeigt auf das vorhergehende Feldelement,
  `*(p-1)` "ist" das vorhergehende Feldelement
- `p+2` zeigt auf das übernächste Feldelement,
  `*(p+2)` "ist" das übernächste Feldelement
- . . .

Erlaubte Operationen:

1. Addition/Subtraktion eines ganzzahligen Wertes `n` auf/von `p`, etwa `p+n` , `p = p-n;` , `p += n;` , `++p` , `p--` , . . .
2. Vergleiche (`q` sei eine weitere Adressvariable vom Typ `T`)
   - `p == q` , `p != q` , `p == 0` , `p != 0`
   - `p < q` , `p <= q`
3. Subtraktion zweier Adress–Werte voneinander (`q` sei eine weitere Adressvariable vom Typ `T`):
   - `p - q` : **ganzzahliger** *Abstand* der Feldelemente

Operationen machen nur dann Sinn, wenn `p` und `q` auf Elemnte **ein– und desselben** Feldes (vom Typ `T`) zeigen!

Zeiger wie Feldnamen verwenden

zeigt der Zeiger `p` auf ein Element eines Feldes vom Typ `T`, so gilt:

- `p[0]` entspricht `*p`,
- `p[1]` entspricht `*(p+1)`,
- `p[7]` entspricht `*(p+7)`,
- allgemein: `p[n]` entspricht `*(p+n)` (`n` ganzzahlig).

Fazit: eine Adressvariable kann wie ein Feldname verwendet werden!

Beachte:

- macht nur dann Sinn, wenn `p` tatsächlich auf ein Element eines (anderweitig reservierten) Feldes zeigt!
- Feldüberlauf möglich!
- eine Adressvariable **ist kein** Feld!

Feldnamen als Zeiger verwenden

Der Name eines Feldes hat einen

- **Wert**: Adresse des ersten Feldelementes
- **Typ**: Adresstyp vom vom zugrundeliegenden Typ des Feldes

Fazit: ein Feldname kann wie ein Zeiger verwendet werden (**ist aber keiner!**)

Auch Zeichenkettenliterale (Felder von konstanten `char`'s, etwa `"hallo"`) haben als "Wert" die Adresse des ersten Zeichens.

Anwendung: dynamische Felder

(wenn man mit Feldern arbeiten möchte, deren Länge erst zur Laufzeit feststeht)

Vorgehensweise:

1. Adressvariable des gewünschten Types vereinbaren, etwa:

   ```
   double *a;
   ```

2. wenn die gewünschte Feldlänge feststeht, Feld dynamisch vereinbaren:

   ```
   int n;
   cin >> n;
   a = new double[n];
   ```

   Feld ist nicht explizit initialisiert, man kann initialisierenden Ausdruck angeben:

   ```
   a = new double[n](1.0);    // laut Standard!!
   ```

   (alle `n` Feldelemente bekommen den Wert `1.0`)

3. Adressvariable `a` wir einen Feldnamen verwenden:

   ```
   for ( int i = 0; i < n; ++i)
   { // mach was mit a[i]
     ...
   }
   ```

4. Freigabe, wenn das dynamisch allokierte Feld nicht mehr benötigt wird (geschieht **nicht** automatisch!)

   ```
   delete[] a;
   ```

Funktionen

werden

- deklariert (bekannt gemacht, in jedem Quelltext erforderlich, wo die Funktion aufgerufen wird)
- aufgerufen
- definiert (genau einmal, ggf. in separatem Quelltext!)

Funktion hat

- Namen
- Rückgabetyp
- Signatur (Typ, Anzahl und Reihenfolge der Parameter)

Erläuterungen:

- bei der Funktionsdeklaration werden Name, Rückgabetyp und Signatur bekanntgemacht (Compiler kann damit richtigen Aufruf überprüfen!, Parameter: zumindest die Typen, Namen nicht erforderlich!)
- beim Aufruf (Funktionsname gefolgt von runder Klammer auf): für jeden Funktionsparameter steht ein entsprechendes Funktionsargument
- bei der Definition
  - haben Parameter einen Namen
  - steht der **Anweisungsteil**
  - Funktionsparameter sind (i. Allg) lokale Variablen der Funktion

Beispiele für Funktionen:

1. Funktion mit einem `int` und einem `double`–Argument und `int`–Rückgabe:
   - Deklaration: `int fkt(int, double);`
   - Aufrufe:

```
int i,j,k;
double x;
...
i = fkt(j,x);   // ok!
fkt(j,x);       // ok, Ergebnis nicht benoetigt!
i = fkt(j,k);   // ok, Typumwandlung
...
```

2. Funktion ohne Argumente:
   - Deklaration: `int fkt(void);`
   - Aufruf:

```
int i;
...
i = fkt();
...
```

3. Funktion ohne Rückgabe:
   - Deklaration: `void fkt(int);`
   - Aufruf:

```
int i;
...
fkt(i);
...
```

Beispiele für Funktionen:

4. Adresse als Argument:
   - Deklaration: `void fkt(int*);`
   - Aufrufe:

```
int i, a[100];
...
fkt( &i);  // ok, Adresse von i wird uebergeben
fkt( a);   // ok, Anfangsadresse des Feldes
...
```

5. Adresse als Funktionsergebnis:
   - etwa `int`–Adresse, Definition:

```
int * fkt(int n)
{ int *p = new int[n];
  ...
  return p;
}
```

     (Vorsicht: Rückgabe der Adresse einer lokalen Variablen vermeiden!)

   - etwa reine (typlose) Adresse, Deklaration:

```
void *fkt ( int);
```

6. Felder als Argument oder Ergebnis einer Funktion:

   geht eigentlich nicht, **nur Anfangsadresse**

Call by Value:

Funktionsparameter sind i. Allg. **lokale** Variablen der Funktion, die beim Funktionsaufruf **den Wert** der korrespondierenden Funktionsargumente erhalten.

Folgende Funktion zum Vertauschen zweier Variable **funktioniert nicht**!

```
void swap(int a, int b)
{ int tmp = a;
  a = b;
  b = tmp;
}

// Aufruf
...
int i, j;
...
swap(i,j);
...
```

Abhilfe: mit Adressen arbeiten (einzige Möglichkeit in C)

```
void swap(int *a, int *b)
{ int tmp = *a;
  *a = *b;
  *b = tmp;
}

// Aufruf
...
int i, j;
...
swap(&i,&j);
...
```

Grundlegendes Prinzip in C:

Will m an in C durch eine Funktion den Inhalt einer Variablen abändern, so muss der Funktion **die Adresse der zu ändernden Variablen** *übergeben werden!*

Da bei der Übergabe eines Feldes an eine Funktion in der Funktion nur **die Adresse** des Feldanfangs ankommt (und in einer Adressvariablen abgespeichert wird), kann ein an eine Funktion übergebenes Feld von der Funktion abgeändert werden!

Alternative in C++: Referenzen:

eine Referenz ist ein anderer Name für eine (bei der Erzeugung der Refernez) bereits vorhandene Variable!

```
void swap( int &a, int &b)
{ int tmp = a;
  a = b;
  b = tmp;
}

// Aufruf
...
int i, j;
...
swap(i,j);
...
```

Bei diesem Aufruf ist der Name `a` in swap nur ein anderer Name für die Variable `i` und `b` für `j`.

Referenzen sind auch als Funktionsergebnisse möglich:

```
int& fkt(int, int);
```

(Keine Referenzen auf **lokale Variablen** zurückgeben!)

Zeiger/Referenzen und `const`:

1. Wird eine Adresse an eine Funktion übergeben und wird das, worauf diese Adresse zeigt, innerhalb der Funktion nicht geändert, so sollte der entsprechende Parameter ein Zeiger auf `const` sein!

   ```
   int fkt(const double *param);
   ```

   Damit wird die Funktion auch für Konstanten als Argument aufrufbar (aber auch für Variablen)!

2. Wird eine Variable per Referenz an eine Funktion übergeben und wird die Variable innerhalb der Funktion nicht geändert, so sollte der entsprechende Parameter eine Referenz auf `const` sein!

   ```
   int fkt(const double &param);
   ```

   Damit wird die Funktion auch für Konstanten als Argument aufrufbar (aber auch für Variablen)!

3. Liefert eine Funktion als Ergebnis eine Adresse und soll das, was an dieser Adresse steht, im Folgenden nicht abgeändert werden, so sollte die Funktion eine Adresse auf `const` zurückgeben:

   ```
   const int * fkt(...);
   ```

4. Liefert eine Funktion als Ergebnis eine Referenz auf eine Variable und soll diese Variable im Folgenden nicht abgeändert werden, so sollte die Funktion eine Referenz auf `const` zurückgeben:

   ```
   const int & fkt(...);
   ```

Konstantheit von Parametern und von Funktionsergebnissen (jeweils vom Adress– oder Referenztyp) muss bei **Deklaration** und **Definition** der Funktion angegeben sein!

Standardargumente von Funktionen:

- können bei der **Deklaration** einer Funktion angegeben werden,
- Parametern können (von hinten beginnend, der Reihe nach) Werte zugewiesen werden,
- beim Aufruf können (von hinten beginnend, der Reihe nach) Argumente weggelassen werden (Parameter erhalten dann entsprechende Standardwerte)

Beispiel:
Funktionsdeklaration:

```
void fkt( int a, double b = 2.7,
          int c = 4, double d = 3.1);
```

Aufrufe:

```
int i,j;
double x,y;
...
fkt(i,x,j,y);   // a bekommt Wert von i,
                // b bekommt Wert von x,
                // c bekommt Wert von j,
                // d bekommt Wert von y.
fkt(i,x,j);     // a bekommt Wert von i,
                // b bekommt Wert von x,
                // c bekommt Wert von j,
                // d bekommt Defaultwert 3.1
fkt(i,x);       // a bekommt Wert von i,
                // b bekommt Wert von x,
                // c bekommt Defaultwert 4,
                // d bekommt Defaultwert 3.1
fkt(i);         // a bekommt Wert von i,
                // b bekommt Defaultwert 2.7
                // c bekommt Defaultwert 4,
                // d bekommt Defaultwert 3.1
```

`inline`–Funktionen:

- bei **Definition** ist zusätzlich das Schlüsselwort `inline` angegeben,
- Definition muss **vor** dem Aufruf im selben Quelltext erfolgen,
- wird **nicht** deklariert,
- Definition einer `inline`–Funktion kann in einer Headerdatei stehen (nicht mehrfach einbinden!),
- Compiler wird "gebeten", Code für die Funktion an Ort und Stelle einzubinden (kein Funktionssprung),
- Compiler muss dieser Bitte nicht nachkommen (insbes. bei großen oder rekutsiven Funktionen),
- möglicherweise (geringfügiger) Perforemancegewinn,
- Typüberprüfung und Umwandlung wie bei richtigen Funktionen.

Beispiel:

```
inline int max( int a, int b)
{
  return ( a > b ? a : b );
}
```

Funktionsüberladung:

- unterschiedliche Funktionen mit **gleichem Namen** aber **(wesentlich) unterschiedlicher Signatur**.
- Compiler entscheidet anhand der angegebenen Argumente, welche der Funktionen er zu nehmen hat!
- Argumenttypen müssen genau zu Parametertypen passen oder (mehr oder wendige einduetig) in passende Parametertypen umwandelbar sein!

Beispiel:

```
void swap ( int &a, int &b)
{ int tmp = a;
  a = b;
  b = tmp;
}

void swap ( double &a, double &b)
{ double tmp = a;
  a = b;
  b = tmp;
}
...
int i,j;
double x,y;

swap (i,j);   // ok: swap (int&, int&)
swap (x,y);   // ok: swap(double&,double&)
swap (x,i);   // FEHLER: zwei passende
              // Umwandlungsmoeglichkeiten
```

Funktionsüberladung und Gültigkeitsbereich:

Funktionsüberladung ist nur im gleichen Gültigkeitsbereich möglich, ansonsten: "Funktionsüberdeckung"

```
void f(int);     // global
void g()
{
  void f(double);  // lokal, globales f(int) hat
                   // andereren Gueltigkeitsbereich
  f(1);         // lokales f(double) wird aufgerufen,
                // hierbei wird das Argument von int
                // nach double umgewandelt
  ::f(1);       // Gueltigkeitsbereichsaufloesung,
                // globales f(int) wird aufgerufen
  ...
}
```

Wesentlicher Unterschied in Signatur erforderlich:

Unterschied muss vom Compiler erkannt werden können!

```
void fkt( int );     // Parameter: int
void fkt( int&);     // Parameter: Referenz auf int
int i;
fkt(i);      // Fehler: fkt(int) oder fkt(int&) ???
```

Konstantheit ist wesentlicher Bestandteil der Signatur:

```
void fkt( char *);         // Typ char *
void fkt( char const *);  // Typ char const *
...
char w[100];
fkt(w);          //  1. Version:  fkt(char *);
fkt("hallo");    //  2. Version:  fkt(char const *);
```

Fehlerbehandlung

```
try {
  ...
  if ( Sonderfall1 )
    throw ausdruck1;
  ...

  if ( Sonderfall2 )
    throw ausdruck2;
  ...

  fkt(...);
  ...
}
```

Algorithmus, in dem ggf. Fehler auftreten und erkannt werden

```
catch ( typ1 name)
{
  ...
}
catch ( typ2 name)
{
  ...
}
.
.
.
```

Bereich, in dem auf ggf. aufgetretene Fehler reagiert wird

```
void fkt(...)
{
  ...
  if ( Sonderfall3 )
    throw ausdruck3;
  ...
}
```

Fehlererkennung, auch in Funktionen möglich

Ablauf der Fehlerbehandlung

- `try`–Block wird abgearbeitet, tritt kein Sonderfall auf,
  - Block wird bis zum Ende abgearbeitet,
  - anschließende `catch`'s werden ignoriert,
- tritt jedoch im `try`–Block ein Sonderfall ein:
  - `throw`–Anweisung wird ausgeführt, dabei "Fehlerobjekt" ausgeworfen,
  - `try`–Block wird sofort verlassen,
  - alle innerhalb des `try`–Blocks definierten (lokalen) Variablen werden zerstört,
  - einzige überbleibende Wert ist das "Fehlerobjekt" hinter `throw`,
  - Typ des Fehlerobjektes wird der Reihe nach mit den Typen der `catch`'s verglichen:
    - bei erster Übereinstimmung wird der zugehörige Anweisungsteil abgearbeitet, wobei der "Parameter" den Wert des Fehlerobjektes erhält,
    - übrige `catch`'s werden ignoriert,
    - anschließend geht es hinter dem letzten `catch` "normal" weiter.
    - gibt es kein `catch` mit passendem Typen, so wird wird der Fehler ggf. als zu einem übergeordneten `try`–Block gehörig eingestuft und dort behandelt (geschachtelte `try`–Blöcke)
    - eine nicht abgefangene Ausnahme führt zum Programmabbruch!

geschachtelte `try`–Blöcke

```
...
try {   // Anfang try-Block1
  .
  .
  try {   // Anfang try-Block2
    .
    .
  }       // Ende try-Block2
  // Abfangen von Fehlern aus Block2
  catch (int i) { ... }
  catch (char c) { ... }
  .
  .
}      // Ende try-Block1
// Abfangen von Fehlern aus Block1
catch ( float f) { ... }
catch ( double *dp) { ...}
...
```

Abfangen beliebiger Fehlerobjekte

```
...
try {
  ...
}
...
catch ( ... )   // hier wirklich drei Punkte
{
 ...
}
```

Weiterreichen von Fehlerobjekten

```
...
try {           // Anfang try-Block1
  .
  .
  .
  try {         // Anfang try-Block2
    ...
    if (sonderfall)
      throw 7;  // int-Fehler auswerfen
    ...
  }       // Ende try-Block2
  catch (int i) // Abfangen von Fehlern aus Block2
  {
    .           // gewisse Aufraeumungsarbeiten
    .           // bei int-Fehler durchfuehren
    .
    throw;      // Fehlerobjekt erneut auswerfen,
                // weiterreichen
  }
  .
  .
  .
}               // Ende try-Block1
catch ( int f)  // Abfangen von Fehlern aus Block1
{
  ...           // weitere Aufraeumungsarbeiten fuer
                // gleichen Fehler durchfuehren
}
...
```

Unterscheidung von Ausnahmen

anhand des Wertes des Fehlerobjektes:

```
// Fehlerfaelle anhand des Wertes des
// Fehlerobjektes unterscheiden
...
try {
  ...
  if ( dies ) throw 7;
  ...
  if ( jenes ) throw 12;
  ...
  if ( sonstwas ) throw 25;
  ...
}
catch (int i)
{ switch (i)
  {
    ...
    case 7: ...; break;    // dies
    ...
    case 12: ...; break;    // jenes
    ...
    case 25: ...; break;    // sonstwas
    ...
  }
}
```

Unterscheidung von Ausnahmen

anhand des Types des Fehlerobjektes:

```
...
struct Lesefehler {  // Fehlertyp, der einen Fehler
  ...                // beim Lesen anzeigt!
};

struct Speichermangel { // Fehlertyp, der
  ...                   // Speichermangel anzeigt!
};
...

try {
  int zahl, *intfeld;
  ...
  if ( (!(cin >> zahl) )
  { Lesefehler lesefehler; // Fehlerobjekt erzeugen
    throw lesefehler;      // und auswerfen
  }
  ...
  if ( (intfeld = new(nothrow) int[zahl]) == 0)
  { Speichermangel speichermangel; // Fehlerobjekt
    throw speichermangel;          // erzeugen und
  }                                // auswerfen
  ...
}
catch ( Lesefehler err)
{ ... }
catch ( Speichermangel err)
{ ... }
...
```

Ausnahmespezifikation in Funktionsschnittstellen

für Anwender einer Funktion wichtig zu wissen, ob diese ggf. Ausnahmen (von welchem Typ?) auswirft.

- gewöhnliche Deklaration und Definition von Funktionen:

  keine Aussage über Ausnahmen (Funktion könnte Ausnahme beliebigen Types auswerfen!)

- Ausnahmespezifikation kann bei **Deklaration und Definition** der Funktion angegeben werden:

  Beispiele:

  1. Deklaration:

     ```
     int fkt(double, int*)
           throw(Lesefehler, Speichermangel);
     ```

     Definition:

     ```
     int fkt(double x, int *p)
     throw(Lesefehler, Speichermangel)
     {
       ...
     }
     ```

  2. Funktion, welche definitiv **keine** Ausnahmen auswirft:

     Deklaration:

     ```
     int fkt(double, int*) throw();
     ```

     (Definition müsste analog lauten)

  Achtung:

  wirft eine Funktion eine Ausnahme eines Types aus, der **nicht** in der Ausnahmespezifikation aufgeführt ist, so führt das i. Allg. zum Programmabbruch!

Speichermangel, Standardeinstellung:

```
// Standardeinstellung:
// Ausnahme vom Typ: std::bad_alloc auswerfen
#include <new>
using namespace std;
...
try
{ ...
  // Speicher reservieren, ohne Ueberpruefung,
  // ob es geklappt hat:
  double *dp = new double[1000];
  char *cp = new char[10000];
  ...
}
catch( bad_alloc fehler)
{ ...  // auf Speichermangel reagieren
}
...
```

Speichermangel, keine Ausnahmen auswerfen lassen:

```
...
using namespace std;
double *dp1, *dp2;

if ( (dp1 = new(nothrow) double) == 0)
{ // nicht geklappt
}
...
if ( (dp2 = new(nothrow) double[100]) == 0)
{ // nicht geklappt
}
...
```

Speichermangel, Reservespeicher anlegen und verwenden:

```
#include <new>
// Zeiger auf Reservespeicher
char *reserve;

// Dekl. des eigenen New-Handlers
void speichermangel(void);
...
int main()
{
  // Reservespeicher anlegen,
  // duerfte hier noch klappen:
  reserve = new char[10000];

  // eigenen New-Handler installieren:
  set_new_handler( speichermagel);
  ...
}

// Definition des New-Handlers:
void speichermangel()
{
  // zunaechst Reservespeicher freigeben
  delete[] reserve;

  // Aufraeumungsarbeiten durchfuehren
  .
  .
  .
  // Programm beenden
  exit(1);
}
...
```

Ein–/Ausgabe

- wird durch Klassen/Objekte und (objektorientierte) C++–Techniken zur Verfügung gestellt
- Headerdatei `iostream` includen
- zentrale Klassen (Typen) und Objekte (Variablen):
  - Klasse `ostream` (“*Ausgabestrom*“)
    mit vordefinierten Objekten
    - `cout` (*Standardausgabekanal*, → Bildschirm)
    - `cerr` (*Standardfehlerkanal*, → Bildschirm)
  - Klasse `istream` (“*Eingabestrom*“)
    mit vordefiniertem Objekt:
    - `cin` (*Standardeingabekanal*, ← Tastatur)
- Ausgabeoperator `<<` (Überladener *Bit–Shift–Operator*),
  für jeden Standardtypen vordefiniert:

```
int i;
double x;
char c, wort[] = "Hello";
...
cout << "Hallo"; // Ausgabe: String-Literal
cout << i;       // Ausgabe: int-Wert
cout << 7;       // Ausgabe: int-Konstante
cout << x;       // Ausgabe: double-Werte
cout << 3.141;   // Ausgabe: double-Konstante
cout << c;       // Ausgabe: Zeichen
cout << "\n";    // Ausgabe: konstantes Zeichen,
                 // Zeilenvorschubzeichen
cout << wort;    // Ausgabe: Zeichenkette
...
cerr << "Kein Speicher mehr!\n"; // Ausgabe einer
         // Meldung auf die Standardfehlerausgabe
```

- Ausgabeoperatoren `<<` "aneinanderhängen":

  (formal möglich, da das Ergebnis des Ausgabeoperators wieder der Ausgabestrom ist!)

```
...
int i;
double x;
...
cout << i << " * " << x << " = " << i*x << '\n';
...
```

- "wirklicher" Bit–Shift:

```
int i,j
...
cout << (i << j) ;
```

  Bit–Shift

- Manipulatoren für Ausgabe:
  - `endl` für: Zeilenvorschub und Ausgabepuffer leeren,
  - `ends` für: `'\0'` ausgeben und Ausgabepuffer leeren,
  - `flush` für: Ausgabepuffer leeren.

- Eingabeoperator >> (Überladener *Bit–Shift–Operator*),
  für jeden Standardtypen vordefiniert:

```
int    i;
char   c, w[100];
short  s;
long   l;
double x;
float  f;
...
cin >> i;   // Einlesen eines int-Wertes,
            // Abspeichern in Variable i
cin >> c;   // Einlesen eines Zeichens,
            // Abspeichern in Variable c
cin >> w;   // Einlesen eines Strings,
            // Abspeichern in char-Feld w
cin >> s;   // Einlesen eines short-Wertes,
            // Abspeichern in Variable s
cin >> l;   // Einlesen eines long-Wertes,
            // Abspeichern in Variable l
cin >> x;   // Einlesen eines double-Wertes,
            // Abspeichern in Variable x
cin >> f;   // Einlesen eines float-Wertes,
            // Abspeichern in Variable f
...
```

- Eingabeoperatoren >> "aneinanderhängen":

  (formal möglich, da das Ergebnis des Eingabeoperators wieder der Eingabestrom ist!)

```
cin >> i >> c >> w >> s >> l >> x >> f;
```

- Manipulator für Eingabe:

  – ws für: explizites Überlesen von Zwischenraumzeichen

Einschub: Bit–Operatoren

- in C bereits vorhanden,
- eingebaute–Typen: nur für integrale Typen anwendbar,
- Operator: `~`: unär, Bit–Komplement (aus Bit `'0'` wird `'1'` und umgekehrt),
- Operator: &: binär, bitweise UND–Operation (AND):

AND	`'0'`	`'1'`
`'0'`	`'0'`	`'0'`
`'1'`	`'0'`	`'1'`

- Operator: `|`: binär, bitweise (inklusive) ODER–Operation (OR):

OR	`'0'`	`'1'`
`'0'`	`'0'`	`'1'`
`'1'`	`'1'`	`'1'`

- Operator: `^`: binär, bitweise (exklusive) ODER–Operation (XOR):

XOR	`'0'`	`'1'`
`'0'`	`'0'`	`'1'`
`'1'`	`'1'`	`'0'`

- Operator <<: binär, Linksshift: `i<<j`

  Bitmuster von `i` wird um `j` Positionen nach links verschoben, rechts mit `'0'`–Bits aufgefüllt,

- Operator >>: binär, Rechtssshift: `i>>j`

  Bitmuster von `i` wird um `j` Positionen nach rechts verschoben, links mit `'0'`–Bits (oder Vorzeichenbit) aufgefüllt,

- Operatoren können für beliebige "selbstdefinierte" Typen mit beliebiger Bedeutung "überladen" werden,

Fehler in Ein–/Ausgabeströmen

- Fehlerzustand ist im Objekt (etwa `cin` oder `cout`) selbst abgespeichert,
- es gibt implizite “Typumwandlung“ eines Stromes in einen Wahrheitswert:
  - Ergebnis `true`, falls Strom in Ordnung ist,
  - Ergebnis `false`, falls Strom nicht in Ordnung ist.

Beispiele:

```
...
int i;
...
cin >> i;
if ( cin )
{ // Lesen von i hat geklappt!
  ...
}
...
double x;
...
cin >> x;
if ( !cin )
{ // Lesen hat nicht geklappt!
  ...
}
...
while ( cin >> i ) // solange Lesen
                   // eines int’s klappt
{
  ...  // mach was mit i
}
...
```

Dateibehandlung

- Headerdatei `fstream` includen,
- in dieser Datei sind (u.a.) die zwei Klassen:
  - `ifstream` (für *input–file–stream*)
  - `ofstream` (für *output–file–stream*)
- bei der Erzeugung eines entsprechenden Objektes (Variablen), kann (in runden Klammern) der Name der zu öffnenden Datei angegeben werden, etwa:

  ```
  ofstream ausgabe("Ausgabe.txt");
  ```

  Das Objekt `ausgabe` ist eine Variable vom Typ `ofstream` und mit der Datei `Ausgabe.txt` des Systems verknüpft. Diese Datei wird zum Schreiben geöffnet.

  ```
  ifstream eingabe("Eingabe.txt");
  ```

  Das Objekt `eingabe` ist eine Variable vom Typ `ifstream` und mit der Datei `Eingabe.txt` des Systems verknüpft. Diese Datei wird zum Lesen geöffnet.
- Das Öffnen einer Datei sollte immer überprüft werden (entweder direkt drauf reagieren oder Ausnahme auswerfen), etwa:

  ```
  struct Dateifehler {...}; // Fehlerklasse
  ...
  try
  { ...
    ifstream eingabe("Eingabe.txt");
    if ( !eingabe) throw Dateifehler();
    ...
  }
  catch (Dateifehler dateifehler)
  { ... }
  ...
  ```

- nach erfolgreichem Öffnen einer Eingabedatei kann von dem `ifstream`–Objekt wie von `cin` gelesen werden:

```
ifstream eingabe("Eingabe.txt");
...
int    i;
double x;
...
eingabe >> i >> x;
...
```

- nach erfolgreichem Öffnen einer Ausgabedatei kann auf das `ofstream`–Objekt wie auf `cout` geschrieben werden:

```
ofstream ausgabe("Ausgabe.txt");
...
int    i;
double x;
...
ausgabe << i << x << endl;
...
```

- am Ende der Lebenszeit eines `ifstream`– oder `ofstream`–Objektes wird die zugehörige Datei automatisch geschlossen:

```
...
int f(void)
{ ofstream aus("Ausgabe.txt"); // Lebensanfang,
    // Variable aus wird erzeugt und mit der
    // hierbei geoeffneten Datei verknuepft
  ...
  return 0;
}   // Lebensende der automatischen Variablen aus,
    // Datei wird hierbei vom System geschlossen!
...
```

C++–Strings

Notation:

- **C-String**: Feld vom Typ `char` mit abschließendem `'\0'`.

  (auch C–String–Literale wie: `"hallo"`)

- **C++–String**: Objekt (Variable) der Klasse `string`

Enger Zusammenhang zwischen C++–Strings und C–Strings.

Verwendung von C++–Strings:

1. Headerdatei `string` includen
2. Erzeugung und Initialisierung von C++–Strings:

```
string s1;           // erzeugt leeren String
...
string s2("hallo");  // Init. mit C-String-Literal
string s3 = "hallo"; // Init. mit C-String-Literal
...
char w[100] = ...;   // char-Feld,
                     // muss C-String enthalten
string s4(w);        // Init. mit char-Feld
string s5 = w;       // Init. mit char-Feld
...
char *p=...;         // char-Zeiger,
                     // muss auf C-String zeigen
string s6(p);        // Init. mit char-Zeiger
string s7 = p;       // Init. mit char-Zeiger
...
string s8(s2);       // Init. mit C++-String
string s9 = s2;      // Init. mit C++-String
...
```

   (Inhalt des erzeugten Strings jeweils **Kopie** des initialisierenden Ausdrucks!)

3. Konstante Strings (Inhalt kann nach ihrer Erzeugung nicht mehr verändert werden):

   ```
   const string s = ...;  bzw.
   const string s(...);
   ```

   (müssen bei ihrer Erzeugung initialisiert werden!)

4. Ein–/Ausgabe von C++–Strings:

   - Ausgabe: mit <<:

     ```
     string s;
     ...
     // Ausgabe von s und Zeilenvorschub
     cout << s << endl;
     ...
     ```

   - Eingabe mit >> (für nicht konstante Strings):

     ```
     string s;
     ...
     cin >> s;
     ...
     ```

     Ablauf:

     – führender Zwischenraum wird überlesen,
     – alle folgenden Nichtzwischenraumzeichen werden gelesen und der Reihe nach in `s` abgespeichert (bisheriger Inhalt von `s` geht verloren),
     – nächstes Zwischenraumzeichen beendet das Lesen (wird formal nicht mitgelesen!)

   - alternative Funktionen zum Lesen eines C++–Strings:

     – Zeile lesen (Zeilenende wird gelesen, nicht abgespeichert!)

       ```
       istream& getline(istream& strm, string& s);
       ```

     – Alternative zum "*Zeilenende*":

       ```
       istream& getline(istream& strm, string& s,
                        char ende);
       ```

5. Zugriff auf einzelne Zeichen eines C++–Strings:

   Index–Operator `[]`:

```
string s = "hallo";
s[0] = 'H';
cout << s << endl;   /// Ausgabe: Hallo
```

   (Keine Indexprüfung, bei konstanten Strings kann das gelieferte Zeichen nicht geändert werden!)

6. (lexikographischer) Vergleich von Strings:

`==`	Test auf Gleichheit
`!=`	Test auf Ungleichheit
`<`	Test auf lexikographisch kleiner
`>`	Test auf lexikographisch größer
`<=`	Test auf lexikographisch kleiner oder gleich
`>=`	Test auf lexikographisch größer oder gleich

   Beispiel:

```
string s1 = ..., s2 = ...;
...
if ( s1 == s2 )  // Inhalte gleich?
{ ... }
...
if ( s1 < s2 )   // s1 lexikographisch
{ ... }          // kleiner als s2?
...
```

7. Auch Vergleiche mit C–Strings:

```
string s = ...;
char w[100] = ...;  // muss C-String enthalten
char *p = ...;      // muss auf C-String zeigen
...
if ( s == "hallo" )   // Vergleich mit
{ ... }               // Zeichenkettenliteral
if ( "hallo" == s )   // auch so herum moeglich
{ ... }
if ( w < s ) { ... }  // Vergleich mit char-Feld
if ( s > w) { ... }   // auch so herum moeglich
if ( s >= p) { ... }  // Vergleich mit char-Zeiger
if ( p <= s) { ... }  // auch so herum moeglich
```

(mindestens ein C++–String beteiligt!)

8. Zuweisen an C++–Strings:

zugewiesen werden kann C++–String, C–String (Feld, Zeiger, Literal), **ein** Zeichen

```
string s1 = ..., s2 = ...;
char w[100] = ...;   // muss C-String enthalten
char *p = ...;       // muss auf C-String zeigen
...
// Zuweisungen
s1 = s2;        // String an String
s1 = w;         // char-Feld an String
s1 = p;         // char-Zeiger an String
s1 = "hallo";   // Zeichenkettenliteral an String
s1 = 'c';       // Zeichen an String,
...             // entspricht: s1 = "c";
```

9. Verketten, Anhängen:

```
string s1 = ..., s2 = ...;
char w[100] = ...;   // muss C-String enthalten
char *p = ...;       // muss auf C-String zeigen
...

// Verkettungen
s1 + s2;       // String mit String
s1 +  w;       // String mit char-Feld
w + s1;        // char-Feld mit String
s1 +  p;       // String mit char-Zeiger
p + s1;        // char-Zeiger mit String
s1 + "hallo";  // String mit Zeichenkettenliteral
"hallo" + s1;  // Zeichenkettenliteral mit String
s1 + 'c';      // String mit Zeichen,
               // entspricht:  s1 + "c";
'c' + s1;      // Zeichen mit String,
               // entspricht:  "c" + s1;

// Anhaengen
s1 += s2;      // String s2 an String
s1 +=  w;      // char-Feld w an String
s1 +=  p;      // char-Zeiger an String
s1 += "hallo"; // Zeichenkettenliteral an String
s1 += 'c';     // Zeichen an String s1,
               // entspricht: s1 += "c";
```

10. Für C++–Strings existiert eine implementationsabhängige Kapazitätsgrenze.

    Wird diese (bei Erzeugung, Verkettung oder durch Anhängen) überschritten, wird eine Ausnahme vom Typ `length_error` ausgelöst!

Strukturen

Zusammenfassung unterschiedlicher Variablen unterschiedlichen Types zu einer Einheit:

1. Typ–Vereinbarung:

```
struct anschrift
{ string strasse;
  int hausnr;
  int plz;
  string ort;
};
```

2. Definition einer Variablen dieses Types:

```
anschrift myanschr;
```

3. Zugriff auf “Komponenten“:

```
myanschr.strasse = "Goethestr.";
myanschr.hausnr  = 1;
myanschr.plz     = 52064;
myanschr.ort     = "Aachen";
...
cout << "Meine Adresse:" << endl;
cout << myanschr.strasse << " ";
cout << myanschr.hausnr  << endl;
cout << myanschr.plz << " " ;
cout << myanschr.ort << endl;
```

Strukturen und Funktionen:

```
struct anschrift { ... }; /* Def. wie oben */

/* Ausgabefunktion: */
void anschrift_ausgabe(const anschrift &a)
{ cout << a.strasse << " ";
  cout << a.hausnr  << endl;
  cout << a.plz << " ;
  cout << a.ort << endl;
}

/* Lesefunktion: Funktionsergebnis */
anschrift anschrift_lesen(void)
{ anschrift tmp;
  cout << "Strasse eingeben: ";
  cin  >> tmp.strasse;
  cout << "Hausnummer eingeben: ";
  cin  >> tmp.hausnr;
  cout << "Postleitzahl eingeben: ";
  cin  >> tmp.plz;
  cout << "Ort eingeben: ";
  cin  >> tmp.ort;

  return tmp;
}

/* Anwendung: */
anschrift myanschr;

myanschr = anschrift_lesen();
anschrift_ausgabe(myanschr);
```

Strukturen als Komponenten anderer Strukturen:

```
struct anschrift { ... };   /* Def. wie oben */

struct person
{ string vname;
  string nname;
  anschrift anschr;
  int personalnummer;
};

/* Verwendung: */
person chef;

chef.personalnummer = 123456;
chef.anschr.hausnr = 1;
...
cout << chef.nname << " "
     << chef.anschr.ort << endl;
```

Felder von Strukturen:

```
person mitarbeiter[100];
...
for ( i = 0; i < n; ++i)
{ cin >> mitarbeiter[i].vname;
  cin >> mitarbeiter[i].nname;
  mitarbeiter[i].anschr = anschrift_lesen();
  cin >> mitarbeiter[i].personalnummer;
}
```

Adressen von Strukturen:

```
void person_lesen(person *p)
{ cout << "Vorname: ";
  cin  >> (*p).vname;
  cout << "Name: ";
  cin  >> (*p).nname;
  (*p).anschrift = anschrift_lesen();

  cout << "Personalnummer: ";
  cin  >> (*p).personalnummer;
}
```

alternativer Zugriff über Adressen: Operator `->`

```
void person_lesen(person *p)
{ cout << "Vorname: ";
  cin  >> p->vname;
  cout << "Name: ";
  cin  >> p->nname;
  p->anschrift = anschrift_lesen();

  cout << "Personalnummer: ");
  cin  >> p->personalnummer;
}
```

Strukturen dynamisch vereinbaren:

```
struct anschrift { ... };   // wie oben
struct person { ... };      // wie oben

/* Zeigervariable: */
person *p;
...
/* Speicher dynamisch reservieren: */
p = new person;

...
/* Struktur verwenden: */
p->vname = "Wilhelm";
p->nname = "Hanrath";
p->anschrift = anschrift_lesen();
p->personalnummer = 123456;
...
/* Speicher freigeben: */
delete p;
```

kleine Strukturen

```
struct bildschirmpunkt
{ short x;
  short y;
};

bildschirmpunkt a, b;

a.x = 0; a.y = 0;
b.x = 128; b.y = 64;
```

rekursive Strukturen

Strukturen, die eine Adress–Variable vom eigenen Typ als Komponente haben:

1. Definition des rekursiven Strukturtypes:

```
struct baumel
{ string wort;
  int count;
  baumel *left;
  baumel *right;
};
```

2. Einfüge–Operation:

```
baumel * baum_insert(baumel *p, string W)
{ if ( p == 0 )
  { baumel *tmp = new baumel;
    tmp->wort = W;
    tmp->count = 1;
    tmp->left = 0;
    tmp->right = 0;
    return tmp;
  }

  if ( p -> wort == W)
   ++(p->count);
  else if ( W < p->wort)
    p->left = baum_insert(p->left,W);
  else
    p->right = baum_insert(p->right,W);

  return p;
}
```

3. Ausgabe– und Lösch–Funktion:

```
void baum_ausgabe(baumel *p)
{
  if ( p != 0)
```

```
  { baum_ausgabe(p->left);
    cout << p->wort << ": " << p->count << endl;
    baum_ausgabe(p->right);
  }
  return;
}

void baum_loeschen(baumel *p)
{
  if (p != 0)
  { baum_loeschen(p->left);
    baum_loeschen(p->right);
    delete p;
  }
}
```

4. Anwendung:

```
int main(void)
{ string Wort;
  baumel *anker = 0;

  while ( cin >> Wort)
    anker = baum_insert(anker,Wort);

  baum_ausgabe(anker);
  baum_loeschen(anker);

  return 0;
}
```

Kellerspeicher als Abstrakter Datentyp `Stack`:

1. Header–Datei:

```
#ifndef _Stack_h
#define _Stack_h

struct Stack { // neuer Datentyp:  Stack

  protected: // Impl.-Details, nicht oeffentlich

    int keller[100];
    int sp;

  public:         // oeffentliche Schnittstelle

    Stack();        // Konstruktor
    void push(int); // Funktion zum Einkellern
    int pop(void);  // Funktion zum Auskellern
};

#endif
```

2. Implementationsdatei:

```
// Stack.cc: Impl. des Datentypes  Stack
// Eigene Headerdatei einbinden
#include "Stack.h"
#include <iostream>   // wegen Ein-/Ausgabe
#include <cstdlib>    // wegen exit()

using namespace std;
```

```
/* Definition der zum Datentyp Stack
   gehoerenden Funktion push:         */
void Stack::push(int wert)
{ if ( sp >= 100)    // Keller voll?
  { cerr << "Keller voll" << endl;
    exit(-1);
  }
  else
    keller[sp++] = wert;
}

/* Definition der zum Datentyp Stack
   gehoerenden Funktion pop:         */
int Stack::pop(void)
{ if ( sp == 0)     // Keller leer?
  { cerr << "Keller leer" << endl;
    exit(-1);
  }
  else
    return keller[ --sp];
}

/* Definition des zum Datentyp Stack gehoerenden
   Konstruktors Stack:
   notwendig, damit bei Erzeugung eines Stacks
   der Stack-Pointer sp den Wert 0 erhaelt!  */
Stack::Stack()
{ sp = 0;
}
```

Anwendung dieses Datentypes:

```
// Anwendung, welche Stacks benoetigt:

#include "Stack.h"

int main()
{ int i,j;    /* zwei int-Variablen */
  Stack a, b; /* Erzeuge zwei Stacks, der eine
                 heisst a, der andere b        */
  ...
  a.push(7);  /* Einkellern des Wertes 7 in Stack
                 a, es wird die Member-Funktion push
                 fuer den Stack a aufgerufen     */
  b.push(i);  /* Einkellern des Wertes von i in
                 Stack b */
  j = i+b.pop(); /* b.pop(): hole oberstes Element
                    des Stacks b heraus, mit diesem
                    wird der Ausdruck i+...
                    ausgewertet und das Ergebnis
                    der Variablen j zugewiesen! */
  ...
}

// KEIN Zugriff auf private oder
// protected Komponenten:
...
Stack a, b;
...
a.sp = 10000;         // FEHLER: sp nicht public
...
cout << b.keller[0];  // FEHLER: keller nicht public
...
```

mehrfache Nennung eines Zugriffsabschnittes

```
struct A {
  public:
    ...   // oeffentliche Komponenten
  private:
    ...   // private Implementationsdetails
  public:
    ...   // wiederum oeffentlich
};
```

neues, wie struct zu verwendendes Schluesselwort: `class`

```
struct A {
    int f(void);      // implizit public
    char b;           // implizit public
  private:
    int sp;           // explizit private
    void g(int);      // explizit private
  public:
    void h(int);      // explizit public
  ...
};
class B {
    int f(void);      // implizit private
    char b;           // implizit private
  private:
    int sp;           // explizit private
    void g(int);      // explizit private
  public:
    void h(int);      // explizit public
  ...
};
```

Konstruktoraufrufe:

```
...
#include "Stack.h"
...
Stack a, b, c;          // 3 Konstruktoraufrufe, je
...                     // einer fuer a, b und c
Stack stackfeld[100];   // 100 Konstruktoraufrufe, je
...                     // einer fuer jedes
...                     // Feldelement
Stack *p;               // KEIN Konstruktoraufruf
...                     // (Zeiger)
...
p = new Stack;          // ein Konstruktoraufruf fuer
                        // dynamisch erzeugten Stack
...
p = new Stack[100];     // 100 Konstruktoraufrufe,
                        // je einer fuer jedes dyn.
                        // erzeugte Feldelement
...
// VORSICHT: KEIN Konstruktoraufruf
// bei malloc etc.!!!
// (dynamische Speicherreservierung in C!)
p = ( Stack *) malloc( sizeof( Stack) );
p = ( Stack *) malloc( 100 * sizeof( Stack) );
...
```

Alternative Definition:
alles im Klassenrumpf definieren, Funktionen implizit `inline`:

```
#ifndef _Stack_h
#define _Stack_h

#include <iostream>
#include <cstdlib>

struct Stack { // neuer Datentyp hat Namen:  Stack

  protected:   // Impl.-Details, nicht oeffentlich
  int keller[100];
  int sp;

  public:         // oeffentliche Schnittstelle

  Stack()        // Konstruktor,
  { ... }        // gleich definiert
  void push(int i) // Funktion zum Einkellern
  { ... }          // gleich definiert
  int pop(void)  // Funktion zum Auskellern
  { ... }        // gleich definiert
};

#endif
```

Alternative Definition:
alles in Headerdatei definieren, Funktionen explizit `inline`:

```
#ifndef _Stack_h
#define _Stack_h

#include <iostream>
#include <cstdlib>

struct Stack { // neuer Datentyp hat Namen:  Stack

  protected:   // Impl.-Details, nicht oeffentlich
  int keller[100];
  int sp;

  public:          // oeffentliche Schnittstelle

  Stack();         // Konstruktor,
  void push(int);  // Funktion zum Einkellern
  int pop(void);   // Funktion zum Auskellern
};

// Definition der Memberfunktionen
inline Stack::Stack()
{ ... }

inline void Stack::push(int i)
{ ... }

inline int Stack::pop(void)
{ ... }
#endif
```

Kellerspeicher als Lineare Liste realisiert:

1. Header–Datei:

```
#ifndef _Stack_h
#define _Stack_h

class Stack { // neuer Datentyp hat Namen:  Stack

  protected: // Impl.-Details, nicht oeffentlich
    struct listel {  // eingebetteter Typ:
       int eintrag;  // Listenelement
       listel *next;
    } *p;            // Zeiger auf Listenanfang

  public:          // oeffentliche Schnittstelle
                   // wie bei Feld-Realisierung
  Stack();         // Konstruktor
  void push(int); // Funktion zum Einkellern
  int pop(void);  // Funktion zum Auskellern
};

#endif
```

2. Implementationsdatei:

```
// Stack.cc: Listen-Implementierung des Stack's
// Eigene Headerdatei einbinden
#include "Stack.h"
#include <iostream>   // wegen Ein-/Ausgabe
#include <cstdlib>    // wegen exit()

using namespace std;
```

```
void Stack::push(int wert)
{
  listel * tmp = new listel;  // neues
     // Listenelement dynamisch anfordern

  tmp -> eintrag = wert;  // Wert uebernehmen
  tmp -> next    = p;     // Listenelement vorne
  p              = tmp;   // in Liste einfuegen
}

int Stack::pop(void)
{
  if ( p == 0) // Liste leer?
  { cerr << "Keller leer" << endl;
    exit(-1);
  }

  int i = p -> eintrag; // Wert zwischenspeichern

  listel *tmp = p;      // erstes Listenelement
  p = p -> next;        // loeschen
  delete tmp;

  return i;  // zwischengespeicherten Wert
}            // zurueckgeben

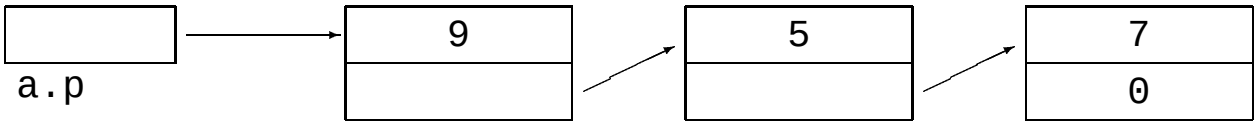
Stack::Stack()
{ p = 0;   // Liste zunaechst leer
}
```

Problem: lokale Benutzung dieses, über eine Lineare Liste realisierten Stacks:

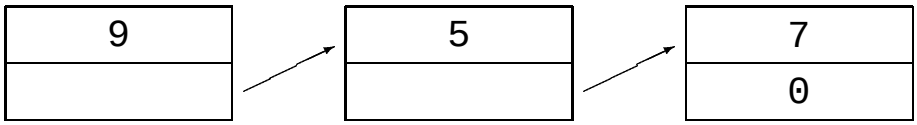
```
...
#include "Stack.h"
...
void fkt(void)
{
  Stack a;
  a.push(7);
  a.push(5);
  a.push(9);

  return;
}
```

Die Situation vor dem Ende der Funktion (vor dem `return`) sieht also wie folgt aus:



Die Situation nach dem Funktionsende ist also wie folgt aus:



Kellerspeicher als Lineare Liste realisiert:
(mit Destruktor)

1. Header–Datei:

```
#ifndef _Stack_h
#define _Stack_h

class Stack {
  protected:
  ...  // wie oben
  public:
  ...   // wie oben
  // zusaetzlich: Dekl. des Destruktors
  ~Stack();
};
#endif
```

2. Implementationsdatei:

```
... // wie oben

// zusaetzlich: Definition des Destruktors
Stack::~Stack()
{ listel *tmp;

  while ( (tmp = p ) != 0)
  { p = p -> next;
    delete tmp;
  }
}
```

Konstruktor/Destruktor–Aufrufe:

```
#include "Stack.h"

void fkt(void)
{
  Stack a, b, c;          // 3 Konstruktoraufrufe
  Stack stackfeld[100];   // 100 Konstruktoraufrufe
  Stack *p, *q, *p1;

  p = new Stack;          // ein Konstruktoraufruf
  q = new Stack[20];      // 20 Konstruktoraufrufe

  // KEIN Konstruktoraufruf:
  p1= (Stack *) malloc(sizeof(Stack));
  ...
  delete p;               // ein Destruktoraufruf
  delete[] q;             // 20 Destruktoraufrufe

  // KEIN Destruktoraufruf:
  free(p1);
  ...
  return;    // 103 Destruktoraufrufe, je einen fuer
             // a, b und c  und je einen fuer jedes
             // Feldelement von stackfeld!
}
```

Korrektur der Schnittstelle für Feld-Realisierung:

```
#ifndef _Stack_h
#define _Stack_h

struct Stack { // neuer Datentyp hat Namen:  Stack

  protected:   // Impl.-Details, nicht oeffentlich
  int keller[100];
  int sp;

  public:         // oeffentliche Schnittstelle

  Stack();        // benoetigt, um Stack zu erzeugen
  void push(int); // Funktion zum Einkellern
  int pop(void);  // Funktion zum Auskellern

  // leerer Destruktor, gleich ganz definiert:
  ~Stack()
  { }
};

#endif
```

aktuelles Objekt in einer Member–Funktion

```
class A {
  public:
    void f1(int);
    int f2(void);
    ...
  private:
    int i_komp;
    ...
};
...
void A::f1( int i)
{ ...
   i_komp = 3*i+7;  // Zugriff auf Komponente
   ...              // i_komp des akt. Objektes
   i = f2();        // Aufruf der Member-Fkt. f2
   ...              // fuer aktuelles Objekt
}
...
int main(void)
{ A a,b;
  ...
  a.f1(7);   // Aufruf von f1 fuer a,
             // aktuelles Objekt ist a
  ...
  b.f1(5);   // Aufruf von f1 fuer b,
             // aktuelles Objekt ist b
  ...
}
```

Zugriff auf Komponenten über den `this`–Zeiger

```
class A {
  public:
    void f1(int);
    int f2(void);
    ...
  private:
    int i_komp;
    ...
};
...
void A::f1( int i)
{ ...
   this->i_komp = 3 * i + 7;  // Zugriff auf Komp.
   ...                        // i_komp des aktuellen Objektes
   i = this->f2();  // Aufruf der Member-Funktion f2
   ...              // fuer aktuelles Objekt
}
...
int main(void)
{ A a,b;
  ...
  a.f1(7);   // Aufruf von f1 fuer a,
             // aktuelles Objekt ist a
  ...
  b.f1(5);   // Aufruf von f1 fuer b,
             // aktuelles Objekt ist b
  ...
}
```

Zugriff auf das ganze aktuelle Objekt:

```
class A {
  public:
    A& f1(int);
    A* f2(int);
    ...
  private:
    int i_komp;
    ...
};
// normale Funktion mit A-Adress-Parameter
void glob_fkt1(A *);
// normale Funktion mit A-Referenz-Parameter
void glob_fkt2(A &);
...
A& A::f1( int i)
{ ...
  // aktuelles Objekt (als Referenz) zurueckgeben:
  return *this;
}
...
A* A::f2( void)
{ ...
  // Adresse des aktuellen Objektes als Argument:
  glob_fkt1(this);
  // aktuelles Objektes als Refernz-Argument:
  glob_fkt2(*this);
  ...
  // Rueckgabe der Adresse des aktuellen Objektes:
  return this;
}
```

Konstante Member–Funktionen:
`const` gehört bei Zeiger- oder Referenz-Parametern zur Signatur:

```
void fkt1( T &a)         // a: Referenz auf T
{ ... }
void fkt2( const T &b)   // b: Refernez auf const T
{ ... }
void fkt3( T *a)         // a: Zeiger auf T
{ ... }
void fkt4( const T *b)   // b: Zeiger auf const T
{ ... }
```

Auch bei Member-Funktion für (Referenz auf) aktuelles Objekt:

```
class A {
  private:
    int a; // irgendeine Komponente:
    ...
  public: // Deklaration:
    int   fkt(void);       // normale Funktion
    int c_fkt(void) const; // konst. Member-Fkt.
    ...
};
...
// Definition der Member-Funktionen:
int   A::fkt( void )
{ ...
  a = ...;   // OK: Komponente a kann
  ...        // geaendert werden!
}
int   A::c_fkt( void ) const
{ ...
  a = ...;     // FEHLER: Komponente a kann nicht
  ...          //         geaendert werden!
}
```

`const` des aktuellen Objektes gehört zur Signatur:

```
A glob_fkt(void);     // Funktion mit A-Ergebnis
...
A a;          // variables Objekt
```

```
A const b;    // konstantes Objekt
...
a.fkt();      // OK
b.fkt();      // FEHLER: b const, fkt nicht!
...
b.c_fkt();    // OK: b const, c_fkt auch!
a.c_fkt();    // auch OK: a zwar nicht const, wird
              // aber durch c_fkt nicht geaendert
...
glob_fkt().fkt(); // FEHLER: kann auf Ergebnis von
                  // glob_fkt() nicht fkt anwenden!
glob_fkt().c_fkt(); // OK: Ergebnis von glob_fkt
                    // ist vom  Typ const A, wende
                    // hierauf c_fkt an!
...
```

mutable–Komponenten:

```
#include <iostream>
#include <ctime>     // wegen time, time_t
#include <cstring>   // wegen strcpy

class Datum {
  private:
    // intern: Zahl der Sek. seit 1.1.1970 0 Uhr:
    time_t internes_Datum;

    // bei Bedarf: Datum als String:
    mutable char * Datum_als_String;
    ...
  public:
    Datum(int);
    long Datum_intern(void) const;
    const char * Datum_String(void) const;
    ...
};

// Konstruktor
Datum::Datum(int i)
{ if ( i < 0)  // heutiges Datum ermitteln:
    internes_Datum = time(NULL);
  else
    internes_Datum = i;

  // String-Repraesentation vorerst leer:
  Datum_als_String = 0;
}
```

```
// internes Datum zurueckgeben
long Datum::Datum_intern(void) const
{ return static_cast<long> (internes_Datum); }

const char * Datum::Datum_String(void) const
{ // Falls String-Repr. bereits vorhanden,
  // gib diese zurueck:
  if ( Datum_als_String != 0 )
    return Datum_als_String;

 // ansonsten: String-Repraesentierung ermitteln:
 Datum_als_String = new char[64];
 // hier wurde mutable-Komponente veraendert!

 strcpy(Datum_als_String, ctime(&internes_Datum));

 return Datum_als_String;
}
...

// Anwendung:
int main(void)
{ Datum const heute(-1);
  ...
  cout << "Heutiges Datum: "
       << heute.Datum_String() << endl;
  ...
}
```

Ausgabe der Anwendung:

```
Heutiges Datum: Thu May  3 10:52:22 2001
```

Verwenden von Klassen

1. In einer Anwendung mehrere Objekte der Klasse vereinbaren:

```
A a,b,c;           // 3 variable A-Objekte
const A d, e, f;   // 3 konstante A-Objekte
```

2. Felder definieren:

```
A A_Feld[100];    // 100 variable A-Objekte
```

3. Referenzen auf Objekte vereinbaren:

```
A a, &b = a;   // b ist Referenz auf a
```

4. Adressvariablen vom Klassentyp verwenden:

```
A *ap;          // ap ist Zeiger auf ein A
```

5. Objekte/Felder dynamisch reservieren:

```
A *ap1 = new A;       // dynamisches A-Objekt
A *ap2 = new A[100];  // dynamisches A-Feld
...
delete ap1;
delete[] ap2;
```

6. Objekte per Wert, Referenz oder Adresse an Funktionen übergeben:

```
void fkt1( A);         // Uebergabe per Wert
void fkt2 ( A&);       // Ubergabe per Referenz
void fkt3( const A&);  // Referenz auf const
void fkt4( A*);        // Zeiger auf A
void fkt5(const A*);   // Zeiger auf const A
```

7. Funktionen mit einem Objekt als Funktionsergebnis (als Wert, Referenz oder Adresse) definieren:

```
A        fkt1(void);  // A-Wert
A&       fkt2(void);  // Ref. auf A-Obj.
const A& fkt3(void);  // Ref. auf const A-Obj.
A*       fkt4(void);  // Adr. eines A-Obj.
const A* fkt5(void);  // Adr. eines const A-Obj.
```

8. Objekte einer Klasse als Komp. in einer anderen Klasse verwenden:

```
class B {  // neue Klasse
  private:
    ...
    A a_Komponente;
    ...
  public:
    ...
};
```

   Keine Rekursion möglich!

   Eine Komponente vom eigenen Adresstyp ist jedoch möglich:

```
class B {  // neue Klasse
  private:
    ...
    B * b_zeiger;
    ...
  public:
    ...
};
```

9. (a) Verwendete Klasse global vereinbaren:

```
class A { ... };

class B {
  private:
    ...
    A a-Komponente;
    ...
  public:
    ...
};
```

(b) Verwendete Klasse lokal als Hilfsklasse vereinbaren:

```
class B {
  private:
    ...
    class A {
      ...
      // Dekl. einer Member-Fkt. der Hilfskl.
      int fkt(void);
      ...
    } a_Komponente;
    ...
  public:
    ...
};
...
// Definition der Member-Fkt. der Hilfsklasse:
int B::A::fkt(void) // fkt ist Member-Fkt. der
{ ... }             // Hilfskl. A der Klasse B
```

In beiden Fällen verwendet die Klasse `B` die Klasse `A`, Zugriff nur auf die Schnittstelle von `A`.

10. Eine dritte Klasse C verwendet in ihren Member-Funktionen lokale Variablen (Objekte) oder auch Parameter vom Typ A:

```
class C { // neue Klasse
  public:
    ...
    void f( A );    // C-Methode mit A-Parameter
    void g(void);   // weitere C-Methode
    ...
  private:
    ...
};

void C::g(void)
{
  A tmp;     // lokales A-Objekt
  ...
}
```

    Zugriff nur auf die Schnittstelle von A.

11. Member–Funktionen einer Klasse C können natürlich auch Parameter oder lokale Variablen vom eigenen Typ, also vom Typ C haben:

```
class C { // neue Klasse
  public:
    void f( C );    // C-Methode mit C-Parameter
    void g(void);   // weitere C-Methode
    ...
  private:
    int i_komp;
    ...
};
```

```
void C::f(C c)
{ // in dieser Funktion sind (mind.)
  // zwei C-Objekte  bekannt:  das
  // aktuelle Objekt (*this) und c!
  ...
  i_komp = ...; // Komponente des akt. Objektes
  c.i_komp = ... ; // Komponente des Objektes c
  ...
}

void C::g(void)
{
  C tmp;     // lokales C-Objekt
             // in dieser Funktion sind (mind.)
             // zwei C-Objekte  bekannt:  das
             // aktuelle Objekt (*this) und tmp!
  ...
  i_komp = ...; // Komp. des aktuellen Objektes
  tmp.i_komp = ... ; // Komp. des Objektes tmp
  ...
}
```

In diesem Fall haben die Member–Funktionen Zugriff auf alle Komponenten der beteiligten `C`–Objekte (Parameter oder lokale Objekte)!

Standard–Parameterloser–Konstruktor

Typ: `A::A();`

Aufrufe:

```
A a,b,c;       //   3-mal parameterloser Konstruktor
A A_Feld[100]; // 100-mal parameterloser Konstruktor

A *p = new A;  //   1-mal parameterloser Konstruktor
A *q = new A[20]; // 20-mal param.-loser Konstruktor
...
// VORSICHT: Funktionsdeklaration!!!:
A a();   // a ist Fkt. ohne Argumente und A-Resultat
...
// temp. Objekt mit Standardkonstruktor erzeugen:
// Funktion mit A-Parameter
void fkt( A );
...
// Aufruf der Funktion: Standard-A-Objekt als Arg.:
fkt( A() );
...
// als Fehlerobjekt:
try { ...
  if ( murks )
    throw A();    // Standard-A-Objekt auswerfen
  ...
  // auch fuer Standardtypen:
  if ( murks )
    throw int();    // Standard-int-Objekt auswerfen
  ...
}
catch( ... )
...
```

Standard–Copy–Konstruktor

Typ: `A::A( const A &);`

Aufrufe:

```
...
A a;           // parameterloser Konstruktor
A b = a;       // b als Kopie von a erzeugen
A c(a);        // c als Kopie von a erzeugen
...
// Funktion mit A-Parameter
void fkt1( A a_param)
{ ... }

// Aufruf dieser Funktion:
fkt1(a);  // bei diesem Aufruf wird der Parameter
          // a_param mit dem Copy-Konstruktor als
          // Kopie des Argumentes a erzeugt

// Funktion mit A-Ergebnis:
A fkt2(void)
{ A tmp;
  ...         // beim return kommt im aufrufenden
  return tmp; // Programmteil eine mit Copy-Konstr.
              // erzeugte Kopie von tmp an!
}

// dynamische Erzeugung mit Copy-Konstruktor
A a;                    // parameterloser Konstruktor
A *p = new A (a);       // Copy-Konstruktor
A *q = new A[100](a);  // 100 x Copy-Konstruktor
```

Selbstgeschriebene Konstruktoren:

```
class Bruch {
  private:
    int zaehler;
    int nenner;
  public:
    // Konstruktor mit zwei int-Parametern
    Bruch(int z, int n)
    { zaehler = z;
      nenner  = n;
    }
  ...
};
...
Bruch a(7,10);   // sieben Zehntel
Bruch b(1,3);    // ein Drittel
Bruch c(4,2);    // vier Zweite
Bruch d(1,0);    // PROBLEM, kein Fehler
...
// jetzt kein Standard-Parameterloser-Konstruktor
// mehr verfuegbar!!!
...
Bruch a;                    // FEHLER!
Bruch feld[10];             // FEHLER!
Bruch *p = new Bruch;       // FEHLER!
Bruch *q = new Bruch[100];  // FEHLER!
...
```

Abhilfe: (unterchiedliche Möglichkeiten!)

1. parameterlosen Konstruktor selbst definieren:

```
class Bruch {
  ...
  public:
    Bruch()    // parameterloser Konstruktor
    { }        // leerer Anweisungsteil ist OK!
    ...
};
```

2. Default–Argumente für parameterbehafteten Konstruktor:

```
class Bruch {
  ...
  public:
    Bruch(int z = 0, int n = 1)
    { zaehler = z; nenner  = n; }
    ...
};
...
Bruch c(3,4);    // Zaehler 3, Nenner 4
Bruch b(7);      // Zaehler 7, Nenner 1
Bruch a;         // Zaehler 0, Nenner 1
// Konstruktoraufrufe mit einem Argument:
Bruch *p=new Bruch(7);    // Zaehler 7, Nenner 1
Bruch *q=new Bruch[20](7);// jew. Zae. 7, Nen. 1

// Konstruktoraufrufe ohne Argument:
Bruch *r=new Bruch;      // Zaehler 0, Nenner 1
Bruch *s=new Bruch[100]; // jew. Zaehl. 0, Nen. 1
Bruch feld[100];         // jew. Zaehl. 0, Nen. 1
```

Initialisierungslisten:

```
class A { ...
  public:
    A ( int);      // einziger Konstruktor fuer A!
    ...
};

class B {
  private:
    A a_komp1;       // erste A-Komponente
    A a_komp2;       // zweite A-Komponente
    ...
  public:
    B( int i ) // hier wird versucht, fuer die
    // A-Komponenten jeweils den parameterlosen
    // A-Konstruktor aufzurufen. Da es diesen nicht
    // gibt, meldet der Compiler einen  FEHLER!!!!
    { ...  }
  ...
};

// Abhilfe: Initialisierungsliste (bei Definition!):
    B( int i ) : a_komp1( i ), a_komp2( 2*i+5 )
    { ... }
// auch fuer Standard-Komponenten:
class Bruch {
  private:
    int zaehler; int nenner;
  public:
    Bruch(int z = 0,int n = 1):nenner(n), zaehler(z)
    { }
  ...
};
```

Copy–Konstruktor und dynamische Komponenten:

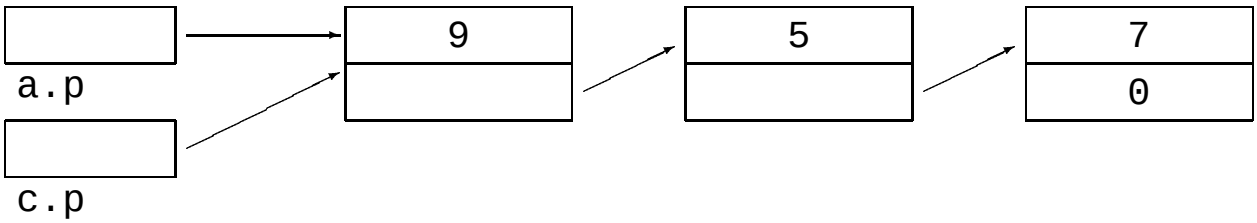
```
class Stack { // neuer Datentyp hat Namen:  Stack
  protected:   // Impl.-Details, nicht oeffentlich
    struct listel {     // eingebetteter Typ:
       int eintrag;     // Listenelement
       listel *next;
    } *p;               // Zeiger auf Listenanfang

  public:          // oeffentliche Schnittstelle
    Stack();       // Konstruktor, init. leere Liste
    void push(int); // Funktion zum Einkellern
    int pop(void);  // Funktion zum Auskellern
    ~Stack();       // Destruktor, Freigabe Liste
};
...
void fkt(void)
{ Stack a;
  a.push(7);
  a.push(5);
  a.push(9);

  Stack c(a);   // c als Kopie von a erzeugt

  return;
}
```

Situation am Funktionsende:



Abhilfe:

1. Copy–Konstruktor verbieten:

```
class Stack {
  protected:
    ...
    // Copy--Konstruktor privat deklarieren und
    // somit fuer den Anwender verbieten:
    Stack ( const Stack &);

  public:
    ....
};
// braucht nicht definiert zu werden!
```

2. Copy–Konstruktor neu definieren:

```
class Stack {
  protected:
    ...

  public:
    Stack(const Stack &); // Copy-Konstruktor
    ...
};
```

Definition eines geeigneten Copy–Konstruktors:

```
Stack::Stack( const Stack &alt)
{
  p = 0;   // neue Liste zunaechst leer

  // zwei Hilfszeiger
  listel *tmp_alt;
  listel *tmp_neu;

  if ((tmp_alt = alt.p) != 0) // falls alte Liste
  {                           // nicht leer
    // erstes Element kopieren
    p = new listel;
    p->eintrag = tmp_alt->eintrag;
    p->next   = 0;

    tmp_neu = p;
    tmp_alt = tmp_alt->next;

    // ggf. alle weiteren Elemente kopieren
    while ( tmp_alt != 0 )
    {
      tmp_neu->next = new listel;
      tmp_neu->next->eintrag = tmp_alt->eintrag;
      tmp_neu->next->next = 0;

      tmp_alt = tmp_alt->next;
      tmp_neu = tmp_neu->next;
    }
  }
}
```

`explicit`–**Konstruktoren:**

1. Konstruktor für Klasse `A` mit Parameter vom Typ `T` definiert Typumwandlung von `T` nach `A`:

```
class Bruch { ...
  public:
    Bruch (int = 0, int = 1);
    ...
};

void fkt(Bruch);  // Fkt. mit Bruch-Parameter,
// keine gleichnamige Fkt. fuer int-Parameter!

// Aufruf:
fkt(7);    // aus int 7 wird temp. Bruch 7/1
```

2. implizite Typumwandlung verhindern:

```
class Bruch { ...
  public:
  // explicit-Deklaration des Konstruktors
  explicit Bruch( int z, int n);
  ...
};
...          // Compiler-FEHLER-Meldung:
fkt( 7 );    // keine passende Funktion da keine
             // implizite Typumwandlung vorhanden!
...
fkt( Bruch(7) ); // OK, expl. Konstruktoraufruf

fkt( (Bruch) 7); // OK, explizite Typumwandlung

fkt( static_cast<Bruch> (7) );  // OK, explizite
                    // Typumwandlung
```

Ausnahmen in Konstruktoren

```
class A {
    ...
  public:
    // eingebettete Fehlerklasse
    struct A_Konstruktion_Fehler {};

    // Konstruktor mit Ausnahmespezifikation
    A(int i) throw(A_Konstruktions_Fehler);
    ...
};

class B  {
    A a;
    ...
  public:
    // Deklaration des Konstruktors,
    // keine (oder leere) Ausnahmespezifikation:
    B(int);
    ...
};

// Definition des B-Konstruktors
B::B(int i)
try
  : a(i)    // Konstruktoraufruf fuer A-Komponente
  {
    ... // Anweisungsteil des B-Konstruktors
  }
  catch ( A::A_Konstruktions_Fehler err)
  {
    ...  // Behandle die A-Ausnahme
  }
```

Ausnahmen in Destruktoren:

```
class A { ...
  public:
    // eingebettete Fehlerklasse
    struct A_Destruktions_Fehler {};
    // Destruktor mit Ausnahmespezifikation
    ~A() throw(A_Destruktions_Fehler);
    ...
};
A::~A() throw(A_Destruktions_Fehler)
{ ...
  if ( murks )
    if (!uncaught_exception() )
      throw A_Destruktions_Fehler();
    else
    { /* mach andere Fehlerbehandlung */ }
  ...
}
...
class B  { A a;
    ...
  public:
    ~B();
    ...
};

B::~B()
try {
...   // Anweisungsteil des Destruktors
}
catch ( ... )
{ ... }
...
```

Statische–Klassenkomponenten: Daten

```
// Klassenrumpf, etwa in Header-Datei:
class A {
  private:
    // gewoehnliche Member-Daten
    int komp1;
    int komp2;
    ...
  public:
    // statisches Member-Datum,
    static int st_kom;  // hier ausnahmsweise public

    // Member-Funktionen
    A (int);
    int f(void);
  ...
};
...
// Implementierung, etwa in *.cc-Datei:
// Definition des Konstruktors
A::A(int i)
{ ... }
// Definition weiterer Member-Funktionen
int A::f(void)
{ ... }

// Definition der statischen Klassenkomponente,
// ggf. mit Initialisierung:
int A::st_komp = 12;
...
```

```
// Zugriff
int main(void)
{
  A a,b;

  a.st_komp = 5;   // Zugriff ueber Objekt a
  b.st_komp = 7;   // Zugriff ueber Objekt b
  cout << a.st_komp << endl; // Zugriff ueber
           // Objekt a, Ausgabe des Wertes 7!!
  // Zugriff ueber explizite Qualifikation,
  // nicht ueber Objekt
  A::st_komp = 25;
  ...
}

// statische Komponente vom eigenen Typ moeglich,
// KEINE Rekursion:
class Datum {
  private:
    // normale Komonenten
    int t, m, j;
    // statische Komponente vom eigenen Typ
    static Datum standardDatum;
  ...
};
```

Statische–Klassenkomponenten: Funktionen

- haben keinen `this`–Zeiger,
- können nur auf statische Memberdaten zugreifen!

```
// Klassenrumpf:
class A {
  private:
   static int st_komp;
   ...
  public:
   static void setze_st_komp(int);
   ...
};
...
// Implementierung:
// statisches Member-Datum:
int A::st_komp;
// statische Member-Funktion:
void A::setze_st_komp(int i)
{ st_komp = i;
  ...
}

// Anwendung:
int main(void)
{ A a;
  // Aufruf der statischen Member-Funktion
  // ueber explizite Qualifikation
  A::setze_st_komp(5);
  // Aufruf derselben Funktion ueber das Objekt a
  a.setze_st_komp(7);
  ...
}
```

Konstanten/Referenzen als Komponenten

```
// irgendeine Klasse
class B { ... };

// weitere Klasse
class A {
  private:
   const int i_komp;   // Konstante!
   B &b_komp;          // Referenz!
  ...
  public:
    A( int, B&);
    ...
};

// Implementierung des Konstruktors:
// Erzeugung der const/Referenz-Komponente
// MUSS ueber Initialisierungsliste erfolgen:
A::A( int i, B &b) : i_komp( i + 3), b_komp(b)
{ ... }
```

Klassenkonstanten: (statisch UND konstant!)

```
// Klassenrumpf:
class A {
  private:
    // statische Konstante:
    static const double PI;
  ...
};
...
// Implementierung:
const double A::PI = 3.1415926;
...
// bei INTEGRALEN Klassenkonstanten ist
// Initialierung auch im Klassenrumpf moeglich:
...
class A {
  private:
```

```
    // Deklaration und Initialisierung
    // der Klassenkonstanten
    static const int FELDLAENGE = 100;

    // Verwendung dieser Konstanten
    double d_feld[FELDLAENGE];

  ...
};
...
// Implementierung: (trotzdem erforderlich!)
const int A::FELDLAENGE;
...
// Alternative bei aelteren Compilern
class A {
  private:
    // eingebauter namenloser enum-Typ:
    enum { FELDLAENGE = 100 };

    // Verwendung dieses Namens
    double d_feld[FELDLAENGE];
  ...
};
```

Komponentenzeiger auf Member–Daten:

```
// Klasse: ausnahmsweise alles public:
class A {
  ...
  public:
    int i;
    int j;
    double d;

    int f(void);
    int g(void);
    void h(int);
    ...
};

// Definition eines Zeigers p, der nur auf
// int-Komponenten eines A-Objektes zeigen darf:
int A::*p;

// Adress-Zuweisung an diesen Zeiger:
p = & A::i;   // p zeigt auf Komp. i von A-Objekten
...
p = & A::j;   // p zeigt auf Komp. j von A-Objekten
...
p = & A::d;   // FEHLER: d ist double, kein int
...
```

```
// Zugriff ueber Komponentenzeiger:
p = & A::i;   // p zeigt auf Komp. i von A-Objekten

A a, b;       // zwei A-Objekte
A *op;        // ein Zeiger auf A
...
...a.*p...;   // i-Komponente von a
...b.*p...;   // i-Komponente von b
...
...op->*p...; // i-Komponente des Objektes, auf
              // welches op zeigt!
// entspricht:
...(*op).*p...;

p = & A::j;   // p zeigt auf Komp. j von A-Objekten
...
...a.*p...;   // j-Komponente von a
...b.*p...;   // j-Komponente von b
...
...op->*p...; // j-Komponente des Objektes, auf
              // welches op zeigt!
// entspricht:
...(*op).*p...;
...
```

Komponentenzeiger auf Member–Funktionen:

```
// Klasse: ausnahmsweise alles public:
class A {
  ...
  public:
    int i;
    int j;
    double d;

    int f(void);
    int g(void);
    void h(int);
    ...
};

// Definition eines Zeigers fp, der nur auf
// Member-Funktionen der Klasse A ohne Argument
// und int-Resultat zeigen darf:
int (A::*fp)(void);

// Adress-Zuweisung an diesen Zeiger:
fp = A::f;   // p zeigt auf Member-Funktion f von A
...
fp = A::g;   // p zeigt auf Member-Funktion g von A
...
fp = A::h;   // FEHLER: Member-Funktion h hat
...          // falschen Typ
```

```
// Zugriff ueber Komponentenzeiger:
fp = A::f;   // p zeigt auf Member-Funktion f von A

A a, b;       // zwei A-Objekte
A *op;        // ein Zeiger auf A
...
...(a.*fp)()...;   // Funktion f fuer a aufrufen
...(b.*fp)()...;   // Funktion f fuer b aufrufen
...
...(op->*fp)()...; // Funktion f fuer Objekt
           // aufrufen, auf welches op zeigt!
// entspricht:
...((*op).*fp)()...;

fp = A::g;   // p zeigt auf Member-Funktion g von A
...
...(a.*fp)()...;   // Funktion g fuer a aufrufen
...(b.*fp)()...;   // Funktion g fuer b aufrufen
...
...(op->*fp)()...; // Funktion g fuer Objekt
           // aufrufen, auf welches op zeigt!
// entspricht:
...((*op).*fp)()...;
...
```

Komponentenzeiger:

Vereinfachung der Schreibweise mit `typedef`:

```
class A {
  ...
  public:

    int f(void);
    int g(void);
    void h(int);
    ...
};

typedef int (A::*fp_typ) (void);
// fp_typ ist der Typ: Zeiger auf eine
// Member-Funktion der Klasse A
// mit keinem Argument und int Ergebnis
...
fp_typ fp;
// fp ist ein entsprechender Zeiger
...
fp = A::f;
// fp zeigt auf die (vom Typ her passende)
// Funktion f der Klasse A

A a;          // a ist A-Objekt
A *op = &a;   // op zeigt auf A-Objekt
...
(a.*fp) ();   // Aufruf von f fuer a
(op->*fp) (); // Aufruf von f fuer *op, also a
...
```

befreundete (globale) Funktion:

```
class A
{ ...
  private:
    int i;

  ...
  // Deklaration einer befreundeten (glob.) Funktion
  // unabhaengig vom Zugriffsabschnitt!
  friend int fkt( A &, ...);
  ...
};

// Definition der (globalen) Funktion:
// darf unabhaengig vomZugriffsabschnitt
// auf ALLE Komponenten der A-Objekte,
// mit welcher die Funktion zu tun hat,
// zugreifen!
int fkt (A & a_param,...)
{ A a_obj;
  ...
  a_param.i = ...;   // Zugriff ok, da friend
  ...
  a_obj.i   = ...;   // Zugriff ok, da friend
  ...
}
```

befreundete (globale) Funktion:

kann auch innerhalb des Klassenrumpfes **definiert** werden,

trotzdem **globale** Funktion und **keine** Memberfunktion!

(Ist dann allerdings implizit `inline`!)

```
class A
{ ...
  private:
    int i;

  ...
  // Definition einer befreundeten (glob.) Funktion
  // unabhaengig vom Zugriffsabschnitt!
  friend int fkt( A & a_param, ...)
  { A a_obj;
    ...
    a_param.i = ...;   // Zugriff ok, da friend
    ...
    a_obj.i   = ...;   // Zugriff ok, da friend
    ...
  }
  ...
};
```

befreundete Member–Funktion einer anderen Klasse:

```
class B;      // Vorwaertsdeklaration,
              // nur Namen bekanntmachen!

class A
{ ...
  private:
    int i;
  ...
  // Deklaration einer befreundeten Member-Funktion
  // der Klasse B
  // unabhaengig vom Zugriffsabschnitt!
  friend int B::B_fkt( A &, ...);
  ...
};

class B {
  ...
  public:

  // Deklaration/Definition der Funktion
  // darf unabhaengig vomZugriffsabschnitt
  // auf ALLE Komponenten der A-Objekte,
  // mit welcher die Funktion zu tun hat,
  // zugreifen!
  int B_fkt(A &,...)
  { A a_obj;
    ...
    a_param.i = ...;   // Zugriff ok, da friend
    ...
    a_obj.i   = ...;   // Zugriff ok, da friend
    ...
  }
```

befreundete Klasse:

```
class B;      // Vorwaertsdeklaration,
              // nur Namen bekanntmachen!

class A
{ ...
  private:
    int i;

  ...
  // Deklaration der befreundeten Klasse B
  friend B;
  ...
};

class B
{
  ...
};
```

Alle Member–Funktionen von B, die irgendetwas mit A–Objekten (als Parameter, lokale oder globale Variablen) zu tun haben, können auf all deren Komponenten zugreifen!

Tabelle der C++–Operatoren:

Nr.	Operator	Assoziativität
1	`::`	von links nach rechts
2	`. -> [] () ++ -- typeid() dynamic_cast<>() static_cast<>() reinterpret_cast<>() const_cast<>()`	von links nach rechts
3	`! ~ ++ -- + - * & (type) sizeof new new[] delete delete []`	von rechts nach links
4	`.* ->*`	von links nach rechts
5	`* / %`	von links nach rechts
6	`+ -`	von links nach rechts
7	`<< >>`	von links nach rechts
8	`< <= > >=`	von links nach rechts
9	`== !=`	von links nach rechts
10	`&`	von links nach rechts
11	`^`	von links nach rechts
12	`|`	von links nach rechts
13	`&&`	von links nach rechts
14	`||`	von links nach rechts
15	`?:`	von rechts nach links
16	`= += -= *= /= %= &= ^= |= <<= >>=`	von rechts nach links
17	`,`	von links nach rechts

Standard–Zuweisungs–Operator:

standardmäßig für jede Klasse (`struct`, `class` aber auch `enum`) vorhanden, Ausnahme:

- Klasse enthält (nicht `static`) Referenz–Komponente,
- Klasse enthält (nicht `static`) konstante Komponente,
- Klasse enthält Komponente ohne Zuweisungsoperator

```
class Bruch {
  private:
    int zaehler;
    int nenner;
  public:
    Bruch ( int z = 0, int n = 1)  // Konst. mit
      : zaehler(z), nenner(n) // Init.-Liste
    { }
    ...
};

...
Bruch a (1,3), b, c;  // drei Brueche
...
b = a;  // Zuweisung, es wird
        // komponentenweise zugewiesen
...
c = 7;  // es wird mittels des Konst. aus
        // 7 ein temp. Bruch 7/1 erzeugt
        // und dieser dem c zugewiesen!
...
a = b = c;  // auch moeglich, Wert einer
            // Zuweisung ist Wert der linken
            // Seite nach der Zuweisung
```

Standard–Adress–Operator:

standardmäßig für jede Klasse (`struct`, `class` aber auch `enum`) vorhanden.

```
class Bruch { ... };
...
Bruch  a;      // Bruch-Objekt
Bruch *p;      // Zeiger auf Bruch
...
p = &a;        // &a: Adresse von a
...
```

Standard–Komma–Operator:

standardmäßig für jede Klasse (`struct`, `class` aber auch `enum`) vorhanden.

```
class A {
  ...
  public:
    void f(void)
    ...
};
...
A a,b;
...
... a,b ...  // Komma-Operator
...
(a,b).f();   // fuer das Ergebnis des Ausdrucks
             // (a,b), also fuer b, wird die
             // Funktion f aufgerufen
...
// Bedingte Ausdruecke sind auch moeglich:
 ((Bedingung) ? a : b ).f();
...
```

Operator–Überladung als globale Funktion:

```
class Bruch {
  private:
    int zaehler;
    int nenner;
  public:
    Bruch (int z=0, int n=1)  // Konstruktor mit
     : zaehler(z), nenner(n)  // Init.-Liste
    { }
    // friend-Deklaration der Multiplikation
    friend Bruch operator *(Bruch, Bruch);
    friend Bruch operator -(Bruch);
    ...
};
...
// Definition der Multiplikation von Bruechen
Bruch operator * (Bruch a, Bruch b)
{ Bruch tmp;

  tmp.zaehler = a.zaehler * b.zaehler;
  tmp.nenner  = a.nenner  * b.nenner;
  return tmp;
}
// Definition der Minusoperators fuer Brueche
Bruch operator - (Bruch a)
{ Bruch tmp;

  tmp.zaehler = -a.zaehler;
  tmp.nenner  =  a.nenner;
  return tmp;
}
```

```
// Anwendung:
Bruch a, b, c, d;
...
a = b * c;         // b mit c multiplizieren und das
                   // Ergebnis dem a zuweisen
a = operator *(b,c);  // gleichwertig
...
a = b * 7;         // aus 7 wird mit dem Konstruktor
                   // der temp. Bruch 7/1, b wird mit
                   // diesem multipliziert und das
                   // Ergebnis dem a zugewiesen
a = operator*(b,7);  // gleichwertig
...
a = 7 * c;         // aus 7 wird mit dem Konstruktor
                   // der temp. Bruch 7/1 und dieser
                   // wird mit c multipliziert und
                   // das Ergebnis dem a zugewiesen
a = operator*(7,c);  // gleichwertig
...

a = -b;             // a wird der negierte Wert von b
                    // zugewiesen
a = operator -(b); // gleichwertig
...
// Aber
a = b - c;     // FEHLER: Operation Bruch - Bruch
               // nicht definiert!
...
a = b * c * -d;
// entspricht:
a = operator *( b, operator *(c, operator -(d)));
...
```

globale Operator–Überladung für `enum`–Typen:

```
enum galois2 { Null = 0, Eins};

galois2 operator+(galois2 a, galois2 b)
{
  if ( a != b )
    return Eins;
  else
    return Null;
}

galois2 operator*(galois2 a, galois2 b)
{
  if ( ( a == Eins) && ( b == Eins) )
    return Eins;
  else
    return Null;
}
...
// Anwendung:
galois2 a, x, y, z;
...
a = x * y + z;
...
```

Operator–Überladung als Member–Funktion:

```
class Bruch {
  private:
    int zaehler;
    int nenner;
  public:
    Bruch ( int z=0, int n=1)  // Konstruktor mit
     : zaehler(z), nenner(n)   // Init.-Liste
    { }
    // Deklaration des *-Operators:
    Bruch operator*(Bruch) const;
    Bruch operator-(void) const;
    ...
};

// Definition des * Operators:
Bruch Bruch::operator*(Bruch b) const
{ Bruch tmp;

  tmp.zaehler = zaehler * b.zaehler;
  tmp.nenner  =  nenner * b.nenner;
  return tmp;
}
// Definition des - Operators:
Bruch Bruch::operator-(void) const
{ Bruch tmp;

  tmp.zaehler = - zaehler;
  tmp.nenner = nenner;
  return tmp;
}

// Anwendung:
Bruch a, b, d;       // variable Brueche
Bruch const c(1,3);  // konstanter Bruch
...
a = c * b;        // c mit b multiplizieren und das
                  // Ergebnis dem a zuweisen
a = c.operator*(b);  // gleichwertig
```

```
...
a = b * 7;         // aus 7 wird mit dem Konstruktor
                   // der temp. Bruch 7/1, b wird mit
                   // diesem multipliziert und das
                   // Ergebnis dem a zugewiesen
a = b.operator*(7);  // gleichwertig
...
a = 7 * c;         // FEHLER: keine Typumwandlung im
                   //         ersten Argument!!!
a = 7.operator*(c);  // FEHLER: unsinniger Aufruf,
                     //         7 ist kein Bruch!!!
...
a = -b;              // a wird der negierte Wert von
                     // b zugewiesen
a = b.operator-(); // gleichwertig
...
// Aber
a = b - c;        // FEHLER: Operation Bruch - Bruch
...               // nicht definiert
...
a = b * c * -d;
// entspricht:
a = b.operator*( c.operator*(d.operator-()));
...
```

mehrfache Überladung eines Operators:

```
class A {
  ...
  public:
    // 2. Operand ist Zeiger auf const char
    ... operator+(const char *);
    // 2. Operand ist double
    ... operator+(double);
    // 2. Operand ist double,
    // aber konstante Member-Funktion
    ... operator+(double) const;
  ...
};
...
A a;         // variables Objekt
const A b;   // Konstante
int i;
double x;
...
... a + x ...; // OK, Aufruf von: operator+(double);
... a + i ...; // OK, Aufruf von: operator+(double);
               // hierbei wird int i nach double
               // umgewandelt
... b + x ...; // OK, Aufruf von:
               // operator+(double) const;

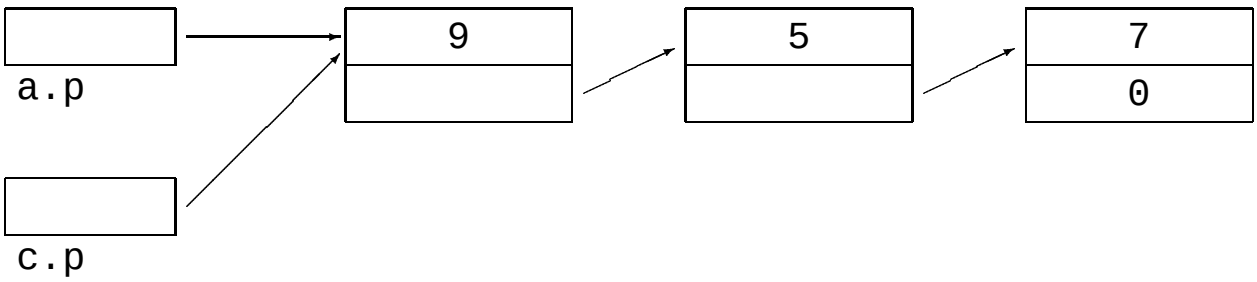
... a + "hallo" ... // OK, Aufruf von:
                    // operator+(const char *);
...
```

Zuweisung und dynamische Komponenten:

```
class Stack { // neuer Datentyp hat Namen:  Stack
  protected:   // Impl.-Details, nicht oeffentlich
    struct listel {     // eingebetteter Typ:
       int eintrag;     // Listenelement
       listel *next;
    } *p;               // Zeiger auf Listenanfang

  public:          // oeffentliche Schnittstelle
    Stack();       // initialisiert leere Liste
    Stack( const Stack &);  // Copy-Konstruktor
    void push(int); // Funktion zum Einkellern
    int pop(void);  // Funktion zum Auskellern
    ~Stack();       // Freigabe der Liste
};
...
// Anwendung:
void fkt(void)
{ Stack a, c;
  a.push(7);
  a.push(5);
  a.push(9);
  c = a;   // Standardzuweisung
}
```

Situation am Funktionsende:



Zuweisung und dynamische Komponenten: Neudefinition

```
class Stack {
  protected:   // Impl.-Details, nicht oeffentlich
    struct listel {     // eingebetteter Typ:
       int eintrag;     // Listenelement
       listel *next;
    } *p;               // Zeiger auf Listenanfang

  public:
  ...
  // Zuweisung deklarieren:
  Stack& operator=(const Stack&);
  ...
};
...
// und implementieren:
Stack& Stack::operator=(const Stack &b)
{
  if ( &b == this)   // Zuweisung an sich!
    return *this;

  // Lineare Liste des aktuellen Objektes
  // erstmal freigeben:
  listel *tmp_neu;
  while ( (tmp_neu = p ) != 0)
  {
    p = p->next;
    delete tmp_neu;
  }
```

```
  // neue Liste wie beim Copy-Konstruktor als Kopie
  // der Liste zu b erzeugen:
  listel *tmp_alt;

  if ( (tmp_alt = b.p) != 0)  // falls Liste von b
  {                           // nicht leer
    // erstes Element kopieren
    p = new listel;
    p->inhalt = tmp_alt->inhalt;
    p->next   = 0;

    tmp_neu = p;
    tmp_alt = tmp_alt->next;

    // ggf. alle weiteren Elemente kopieren
    while ( tmp_alt != 0 )
    {
      tmp_neu->next = new listel;
      tmp_neu->next->inhalt = tmp_alt->inhalt;
      tmp_neu->next->next = 0;

      tmp_alt = tmp_alt->next;
      tmp_neu = tmp_neu->next;
    }
  }

  return *this;
}
```

Folgende Operationszeichen kann man für eigene Datentypen (neu–)definieren:

`+`	`-`	`*`	`/`	`%`	`^`
`&`	`|`	`~`	`!`	`=`	`<`
`>`	`+=`	`-=`	`*=`	`/=`	`%=`
`^=`	`&=`	`|=`	`<<`	`>>`	`<<=`
`>>=`	`==`	`!=`	`<=`	`>=`	`&&`
`||`	`++`	`--`	`->*`	`,`	`->`
`[]`	`()`	`new`	`new[]`	`delete`	`delete[]`

Die übrigen, in folgender Tabelle aufgeführten Operationszeichen kann man nicht überladen:

`dynamic_cast<>()`	`static_cast<>()`	`typeid()`
`(type)`	`.`	`::`
`reinterpret_cast<>()`	`const_cast<>()`	`sizeof`
`.*`	`?:`	

- Anzahl der Argumente (Operanden) kann nicht verändert werden,
- Priorität und Assoziativität kann ebenfalls nicht geändert werden,
- sind etwa `+` und `=` für eine Klasse definiert, so ist `+=` nicht automatisch mitdefiniert,
- will man Operatoren wie gewohnt geschachtelt verwenden, (etwa `a = b = c = d;` oder `a= b*c+d;`) so muss man dafür sorgen, dass die Ergebnistypen entsprechend sind!
- bei globaler Überladung muss mindestens ein Argumenttyp kein Standardtyp sein!

Spezielle Operatoren: Indexoperator `[]`:

- nur als nichtstatische Memberfunktion möglich,
- beliebiger Argumenttyp (nicht notwendigerweise `int`)
- üblicherweise zweimal überladen, einmal als gew. Member–Fkt. und einmal als konstante Member–Fkt..

```
class DoubleFeld { ...
  private:
    double *feldzeiger;
    int feldlaenge;
  public:
    // lokale Fehlerklassen
    struct FeldUnterlauf {};
    struct FeldUeberlauf {};

    // Konstruktor, Destruktor und Zuweisung wegen
    // dynamischer Komponente:
    DoubleFeld(unsigned int);
    DoubleFeld(const DoubleFeld*);
    DoubleFeld& operator=(const DoubleFeld&);

    // Index-Operator:
    // fuer variable Felder
    double& operator[](int);
    // fuer const Felder
    const double& operator[](int) const;
};
...
// Definition des Konstruktors:
DoubleFeld::DoubleFeld( unsigned int laenge)
{ feldzeiger = new double[laenge];
  feldlaenge = laenge;
}
```

```
// Definition des Index-Operators fuer var. Felder:
double& DoubleFeld::operator[](int i)
{ if ( i < 0 )
    throw DoubleFeld::FeldUnterlauf();
  if ( i >= feldlaenge )
    throw DoubleFeld::FeldUeberlauf();
  return feldzeiger[i];
}
// Definition des Index-Operators fuer const Felder:
const double& DoubleFeld::operator[](int i) const
{ if ( i < 0 )
    throw DoubleFeld::FeldUnterlauf();
  if ( i >= feldlaenge )
    throw DoubleFeld::FeldUeberlauf();
  return feldzeiger[i];
}
// Anwendung:
DoubleFeld a(100);       // Feld, Laenge 100
const DoubleFeld b(10);  // const Feld, Laenge 10
...
a[0]  = .1;      // OK!
++a[7];          // OK!
cout << a[99];   // OK!
a[100] = ...;    // LAUFZEITFEHLER!
a[-5] = ...;     // LAUFZEITFEHLER!
...
cout << b[5];    // OK!
b[5] = 7;        // COMPILERFEHLER!
b[7]++;          // COMPILERFEHLER!
...b[-1]...;     // LAUFZEITFEHLER!
...b[11]...;     // LAUFZEITFEHLER!
```

Spezielle Operatoren: Funktionsaufruf `()`:

- nur als nichtstatische Memberfunktion möglich,
- beliebige Anzahl und Typen der Argumente

```
class A {
  public:
    // ein double Argument, Standardparameter 1.0:
    int operator() ( double = 1.0);
    // zwei Argumente, eins int, eins char:
    int operator() ( int, char);
    // zwei Argumente, eins int, eins char,
    // aber konstante Member-Funktion:
    int operator() ( int, char) const;
    // ein Argument, Typ char *:
    int operator() ( char*);
    // ein Argument, Typ const char*:
    int operator() ( const char*);
  ...
};
// Anwendung
A a;  const A b;  char s[10];  double x;
a(x);    // a.operator() (x),
         // Aufruf von operator() ( double = 1.0);
a();     // a.operator() (),
         // Aufruf von operator() ( double = 1.0);
a(7,'c');   // a.operator() (7,'c'),
            // Aufruf von operator() ( int, char);
b(7,'c');   // b.operator() (7,'c'),
        // Aufruf von operator() ( int, char) const;
a(s);   // a.operator() (s),
        // Aufruf von operator() ( char*);
a("hallo"); // a.operator() ("hallo"),
         //Aufruf von operator() ( const char*);
```

Spezielle Operatoren: `->`:

nur als statische Memberfunktion und **unär** möglich!

```
class A {
  ...
  public:
  Typ operator->(void);
  ...
};
...
// Aufruf:
... a->m ... ;
// wird umgesetzt zu:
... (a.operator->()) -> m ...;
```

Beim Überladen von `->*`: keine Besonderheiten!

– entweder global:

```
T1 operator->* ( T2, T3);
```

(`T1`, `T2` und `T3` irgendwelche Typen, wobei `T2` oder `T3` kein Standardtyp ist.)
Der Aufruf

```
T2 a;
T3 b;
...
... a ->* b ...
...
```

wird als

```
... operator->* ( a, b) ...
```

umgesetzt!

– oder als Member–Funktion zu einer Klasse:

```
class A {
  ...
  public:
  T1 operator->*( T2);
  ...
};
```

(`T1` und `T2` beliebige Typen!)

Der Aufruf:

```
A a;
T2 b;
...
... a ->* b ...
...
```

wird als

```
... a.operator->*(b) ...
```

umgesetzt!

Spezielle Operatoren: ++ und --:

– global:

```
A& operator++(A&);      // Praefix
A  operator++(A&, int); // Postfix
A& operator--(A&);      // Praefix
A  operator--(A&, int); // Postfix
...
A a, b;
++a  // Prae, Aufruf von A& operator++(A&)
--a  // Prae, Aufruf von A& operator--(A&)
b++  // Post, Aufruf von A operator++(A&, int)
b--  // bzw. A operator--(A&, int)
     // anstelle des Dummy--Wertes kommt ein
     // zufaelliger an!
```

– als Member:

```
class A {
  public:
    A& operator++();     // Praefix
    A  operator++(int);  // Postfix
    A& operator--();     // Praefix
    A  operator--(int);  // Postfix
  ...
};
...
A a, b;
++a     // Prae, Aufruf von A& A::operator++()
--a     // Prae, Aufruf von A& A::operator--()
b++     // Post, Aufruf von A A::operator++(int)
b--     // bzw. A A::operator--(int)
        // anstelle des Dummy--Wertes kommt ein
        // zufaelliger an!
```

Konversionoperatoren:

```
class A {
  ...
  public:
    A(B);       // konstruiere aus einem B ein A!
    ...
};

// Anwendung:
int fkt(A);
B b;
...
fkt(b); // OK: aus b wird mit dem Konstruktor ein A
        // erzeugt und Fkt. mit diesem A aufgerufen
...

// B bekannter Typ, auch Standardtyp moeglich

class A {
  ...
  public:
  // Deklaration des Konversions-Operators:
  operator B();
  ...
};
...
// Definition des Konversions-Operators:
A::operator B()
{ B tmp;
  ...
  return tmp;
}
...
```

```
class Bruch {
  private:
    int zaehler;
    int nenner;
  public:
    Bruch ( int z=0, int n=1) // Konstruktor mit
     : zaehler(z), nenner(n)  // Init.-Liste
    { }
    // Deklaration der Konversion nach double:
    operator double();
    ...
};
// Definition der Konversion nach double:
Bruch::operator double()
{ return (double) zaehler/nenner;
}
// Anwendung:
double fkt(void)
{ Bruch a,b;
  double f(double);
  double x, y;
  ...
  x = a;      // implizite Konversion
  y = f(b);   // implizite Konversion
  ... = y * (double) a;  // explizite Konversion
  ... = y * double(a);   // explizite Konversion
  ... = y * static_cast<double>(a);
                         // explizite Konversion
  ... = y * a;  // implizite Konversion, da kein
                // anderer *-Operator vorhanden
  return a;   // implizite Konversion
}
```

Ausgabeoperator für eigene Klassen:

```
#include <iostream>

class Bruch {
  protected:
    int zaehler;
    int nenner;
  public:
    Bruch(int z, int n): zaehler(z), nenner(n)
    {}
    void printon(ostream &strm)
    { strm << zaehler << '/' << nenner;
    }
  ...
};

// globale Funktion, nicht befreundet:
ostream& operator<<( ostream &strm, Bruch a)
{
  a.printon(strm);
  return strm;  // wegen Verkettung
}
...
// Anwendung:
Bruch a(1,3), b(2,3);
int i=2;
...
cout << a << " * " << i << " = " << b << endl;
// Ergebnis: 1/3 * 2 = 2/3
...
```

Template–Funktionen

```
// Deklaration einer Template-Funktion:
template <class T>
// oder: template <typename T>
const T& max ( const T&, const T&);

// Definition dieser Template-Funktion:
template <class T>
inline const T& max ( const T& a, const T& b)
{ return ( a > b ? a : b); }

// Anwendung:
int i, j;
long k,l;
double x, y;
Bruch a, b;
...
max( i, j); // zwei int-Argumente: System erzeugt
  // aus dem Template die Funktion
  // const int& max(const int &,const int &);
max( k, l); // zwei long-Argumente: System erzeugt
  // aus dem Template die Funktion
  // const long& max(const long&,const long&);
max( x, y); // zwei double-Argumente: System erzeugt
 // aus dem Template die Funktion
 // const double& max(const double&,const double&);
max( a, b); // zwei Bruch-Argumente: System erzeugt
 // aus dem Template die Funktion
 // const Bruch& max(const Bruch&,const Bruch&);
...
// Keine Typumwandlung:
max( x, i);  // FEHLER!!!
...
```

implizite und explizite Instanziierung

```
template <class T>
const T& max( const T&, const T&);
int i,j;
double x,y;
...
max(i,j); // impl. Instanziierung, erzeugt wird:
   // const int& max( const int &, const int &);
max(x,y); // impl. Instanziierung, erzeugt wird:
// const double& max(const double&,const double&);
max(x,i);   // FEHLER!
...
max<int>(i,j); // expl. Instanz., erzeugt wird:
// const int& max( const int &, const int &);
max<double>(x,y); // expl. Instanz., erzeugt wird:
// const double& max(const double&,const double&);
max<double>(x,i); // KEIN Fehler!, ggf. erzeugt
// und mit Typumwandlung verwendet wird:
// const double& max(const double&,const double&);
max(x,i);  // immer noch FEHLER!
max<double>(j,i); // KEIN Fehler!, ggf. erzeugt
// und mit Typumwandlung verwendet wird:
// const double& max(const double&,const double&);
...
template <class T>
void f(void)
{ T tmp;
  ...
}
...
// nur explizite Instanziierung moeglich!
f<int>();      // T ist int
f<double>();   // T ist double
```

Template–Parameter

1. mehrere Typ–Parameter, impl. und expl. Instanziierung:

```
template <class U, class V>
bool kleiner ( U a, V b)
{ return a < b; }

int i, j;
double x,y;
...
kleiner(i,j); // U ist in, V ist int, Funktion
       // bool kleiner(int,int)  wird erzeugt
kleiner(x,y); // U ist double, V ist double,
// Fkt. bool kleiner(double,double)  wird erzeugt
kleiner(i,y); // U ist int, V ist double,
// Fkt. bool kleiner(int,double)  wird erzeugt
kleiner<long, double>(i,j);  // U ist expl. long,
    // V ist explizit double, erzeugt wird
    // Fkt. bool kleiner(long, double) erzeugt
kleiner<long>(i,j);  // U ist explizit long,
           // V ist implizit int
```

2. “normale“ Funktions-Parameter für Templates,
müssen Konstanten sein und dürfen im Template nicht verändert werden!

```
template <class T, int laenge>// T: Typparameter
void roedel(...)   // laenge: normaler Parameter
{ T hilfsfeld[laenge]; // T und laenge werden
  ...                  // verwendet!
  laenge++;              // FEHLER!
  ...
}
```

3. Default–Template–Parameter:

```
// Default-Parameter fuer Templates
// Default-Wert fuer laenge ist 128
template <class T, int laenge = 128>
void roedel1(...);
....
roedel1<char>();    // T ist char, laenge ist 128
roedel1<int,512>(); // T ist int, laenge ist 512
...
// Default fuer laenge ist 128
// und Default fuer T ist int
template <class T = int, int laenge = 128>
void roedel2(...);
...
roedel2();          // T ist int, laenge ist 128
roedel2<char>();    // T ist char, laenge ist 128
roedel2<int,512>(); // T ist int, laenge ist 512
...
// FEHLER: T hat Default-Wert und laenge nicht!
template <class T = int, int laenge >
void roedel3(...);
...
```

Überladen von Templates

```
// Template zum Vergleich zweier Objekte,
// fuer die der < Operator definiert ist
template <class T>
bool kleiner( T a, T b)
{
  return a < b;
}

// normale Funktion zum Vergleichen
// von Zeichenketten
bool kleiner( const char *a, const char *b)
{
  return  strcmp(a,b) < 0;
}

// Anwendung:
int i, j;
char w[10], v[10];
...
kleiner(i,j); // Aufruf der Template-Instanziierung
              // bool kleiner(int, int)
kleiner(v,w); // Aufruf der "normalen" Funktion
        // bool kleiner(const char *, const char *)
...
// Ueberladen von Templates:
// Maximum von 2 T's:
template <class T>
const T& max( const T&, const T&);

// Maximum von 3 T's:
template <class T>
const T& max( const T&, const T&, const T&);
```

Spezialisierung von Templates

```
// Spezialisierung
// Template fuer beliebige Typen:
template <class T>
void fkt(T);

// gleichnamiges Template fuer
// beliebige ZEIGER--Typen:
template <class T>
void fkt(T*);

//-------------------------------------------------

// Partielle Spezialisierung
// Version fuer beliebige Typen
template <class U, class V>
void fkt( U, V);

// Version, falls 2-tes Argument ein int ist
template <class U>
void fkt(U, int);

// Vollstaendige Spezialisierung:
template <>
void fkt(double, int);
```

Template–Klassen

```
// Deklaration der Template-Klasse:
template <typename T>  // oder: template <class T>
class Stack {
  private:
    T feld[100];
    int sp;
  public:
    Stack();   // Konstruktor
    void push(T&);  // Einkellern eines Elementes
    T pop();       // Auskellern eines Elementes
};

// Definition der zug. Funktionen:
template <typename T>  // oder: template <class T>
Stack<T>::Stack()
{ sp = 0; }

template <typename T>  // oder: template <class T>
void Stack<T>::push( T& elem)
{ if ( sp < 100 )
    feld[sp++] = elem;
  else
  { cerr << "Stack voll!" << endl; exit(1); }
}

template <typename T>  // oder: template <class T>
T Stack<T>::pop()
{ if ( sp > 0)
    return feld[--sp];
  else
  { cerr << "Stack empty!" << endl; exit(1); }
}
```

```
// Anwendung:

// als Objekte
Stack<int> intStack;         // Stack fuer int's
Stack<float*> floatPtrStack;  // Stack, float-Zeiger
Stack<Bruch> bruchStack;    // Stack fuer Brueche
...

// als Funktionsparameter und lokale Variable
void f( Stack<int> s)       // Parameter: int-Stack
{
  Stack<int> istack[10]; // lok. Feld von int-Stacks
  ...
}

// Verwendung von typedef:
// Name fuer den Typen: Stack von int's
typedef Stack<int> IntStack;

void f( IntStack s)       // Parameter: int-Stack
{
  IntStack istack[10]; // lok. Feld von int-Stacks
  ...
}
```

Parameter von Templates

1. **Typ–Parameter:**

```
// Zwei Template-Typ-Parameter:
template <class T1, class T2>
class pair {
  public:
    T1 first;
    T2 second;
};
```

2. **gewöhnliche Parameter** (Konstanten!):

```
// "gewoehnliche" Parameter:
template <class T, int laenge>
class Stack {
  private:
    T feld[laenge];
    int sp;
  public: ...
};
// Instanziierung:

// int-Stack’s der Laenge 50:
Stack<int,50> istack1, istack2;

// char-Stack der Laenge 1000:
Stack<char,1000> cstack;
```

3. **Template–Template–Parameter:**
   (Parameter ist selbst ein Template)

```
// Speicher, als Kellerspeicher realisiert:
template <class T> class Stack { /* ... */ };

// Speicher, als Lineare Liste realisiert:
template <class T> class LListe { /* ... */ };

template <class T,
          template <class U> class SPEICHER>
class A {
  SPEICHER<T> speicher;
  ...
};

// Datentyp: int
// Realisierung ueber Stack:
A<int, Stack> a1;

// Datentyp: char *
// Realisierung ueber LListe:
A<char *, LListe> a2;
```

Spezialisierung:

```
// Spezialisierung von Klassen-Templates:

// Vektorklasse fuer allgemeine Typen:
template <class T>
class Vektor {
{
  ...  // Implementierung fuer allgemeine Typen
};
...

// Vektorklasse fuer Zeigertypen:
template <class T>
class Vektor<T*> {
{
  ... // spezielle Implementierung fuer Zeigertypen
};
...

// Vektorklasse fuer void *:
template <>
class Vektor<void*>
{
  ... // spezielle Implementierung fuer void *
};
...
```

Templates innerhalb eines Templates:

1. **Definition im Klassenrumpf:**

```
template <class Scalar>
class complex {
  public:
    Scalar re;     // Real- und Imaginaerteil
    Scalar im;     // vom Typ Scalar

    // Standard-Konstruktor, komplett definiert:
    complex() : re(0), im(0) {};

    // Konstruktor aus zwei Skalaren,
    // komplett definiert:
    complex(Scalar a,Scalar b):re(a), im(b) { };

    // Template-Konstruktor:
    // ein complex<Scalar> mit einem complex<U>
    // initialisieren, komplett definiert:
    template <class U>
    complex( const complex<U> &c)
     : re( c.re), im(c.im) { }

    // Addition, Multiplikation nur deklariert:
    Scalar operator+(Scalar) const;
    Scalar operator*(Scalar) const;
    ...
};

const complex<int> i(0,1);
// initialisiere comlpex<double> b
// mit complex<int> i:
complex<double> b(i);
...
```

2. **Definition außerhalb des Klassenrumpfes:**

```
// Definition eines Template-Elementes ausserhalb
// des Klassenrumpfes

template <class Scalar>
class complex {
  public:
    Scalar re;   // Real- und Imaginaerteil
    Scalar im;   // vom Typ Scalar

    ...

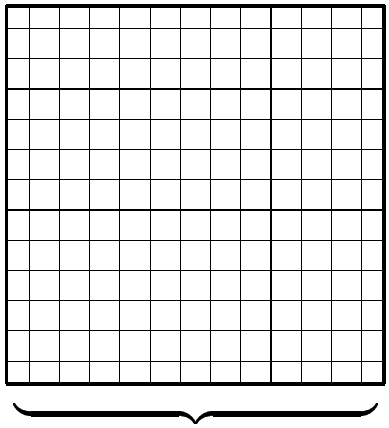
    // Template-Konstruktor:
    // ein complex<Scalar> mit einem complex<U>
    // initialisieren, nur deklariert:
    template <class U>
    complex( const complex<U> &c);

    ...
};
...

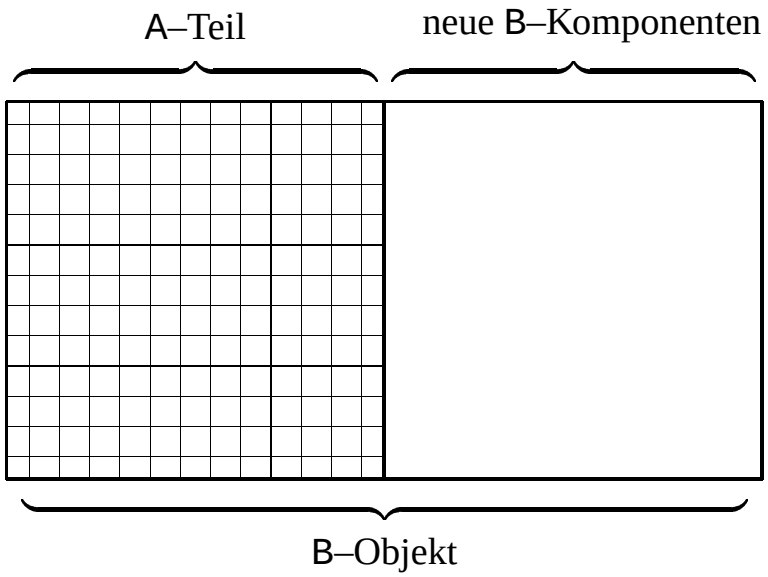
// Definition des Kontruktors eines
// complex<Scalar> aus einen complex<U>:
template <class Scalar>
  template <class U>
  complex<Scalar>::complex( complex<U> &c)
   : re(c.re), im(c.im) // Initialisierungsliste
  { }
...
```

Vererbung

```
class A {
  ...
  ...  // A-Komponenten
  ...
};
```



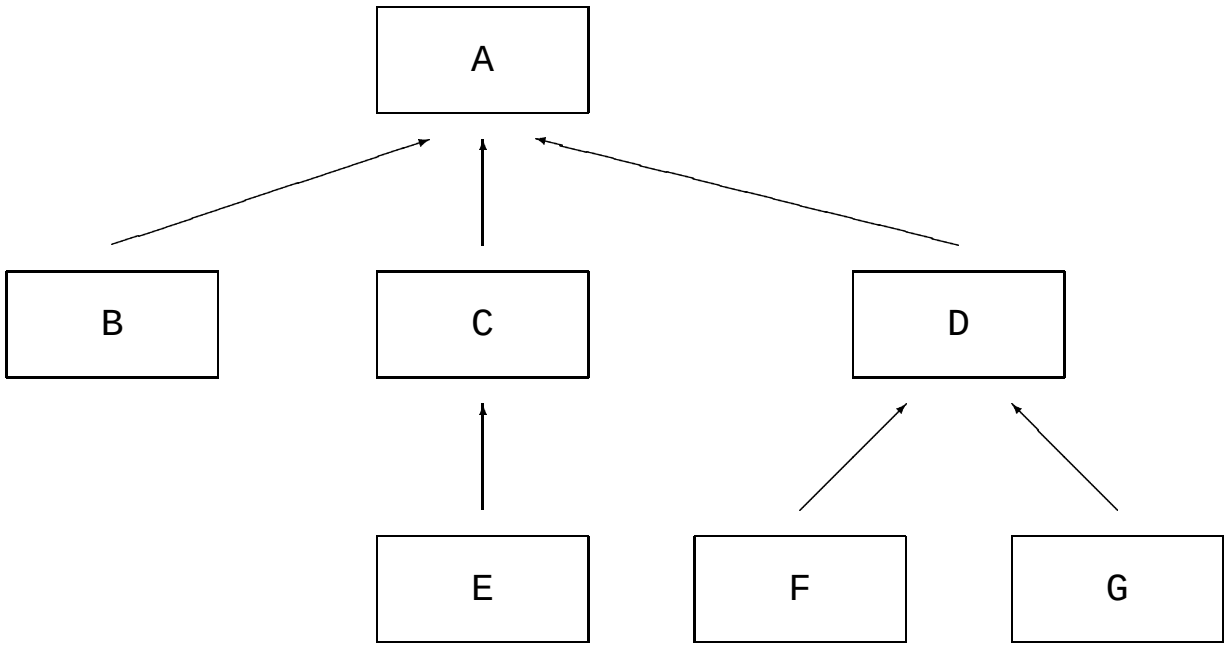

```
class B : public A {
  ...
  ...  // neue B-Komponenten
  ...
};
```



Vererbungsarten

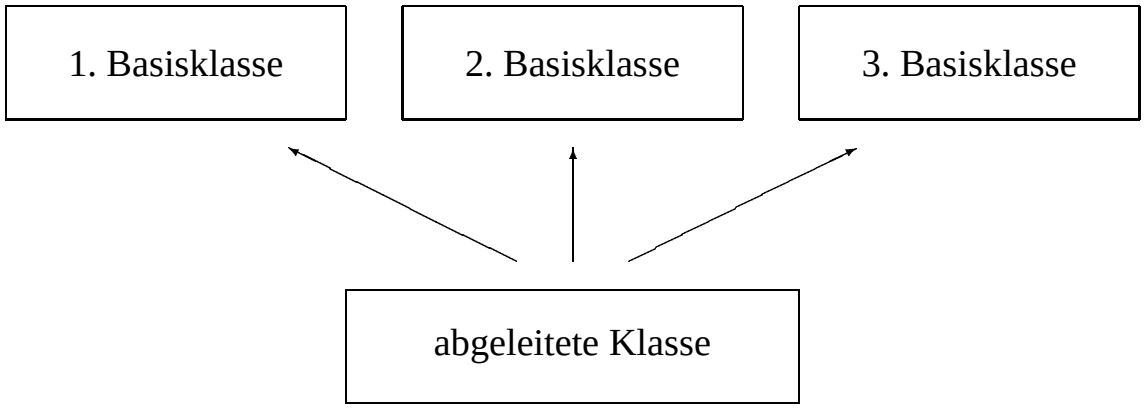
1. **einfache Vererbung**

   abgeleitete Klasse hat **eine** Basisklasse:



2. **Mehrfachvbererbung**

   abgeleitete Klasse hat **mehrere** Basisklassen:

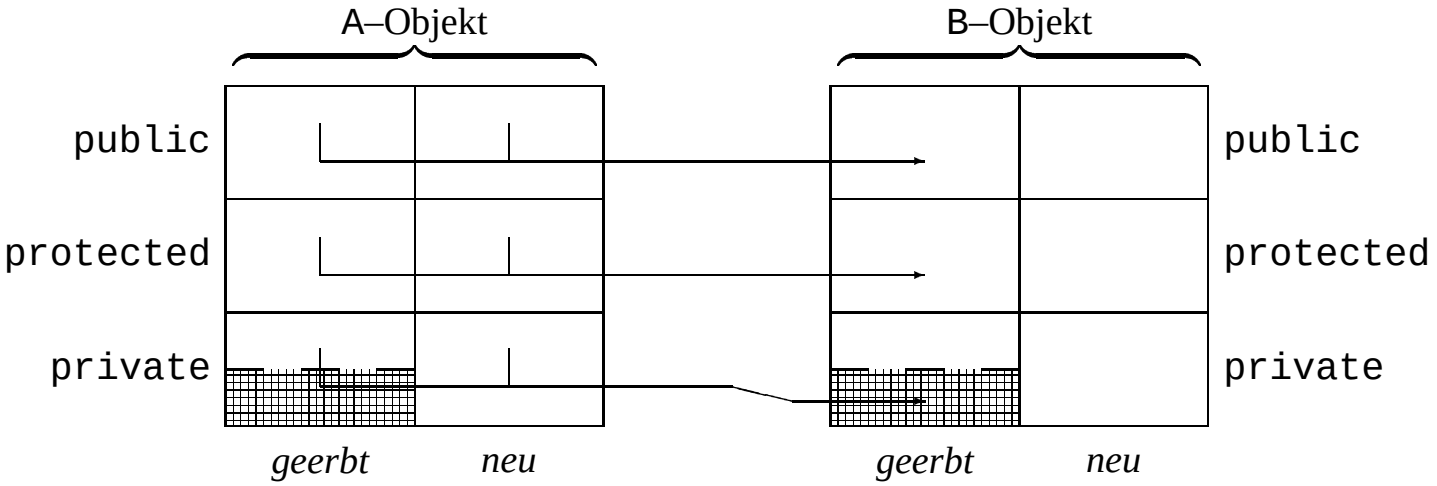


Zugriffsschutz

1. `public`–Vererbung:

```
class A {
  public:
    ...
  protected:
    ...
  private:
    ...
};

class B : public A {
  public:
    ...
  protected:
    ...
  private:
    ...
};
```



Ein B–Objekt **ist ein** A–Objekt, alles, was mit einem A–Objekt machbar ist, ist auch mit einem B–Objekt möglich!

(A–Schnittstelle bleibt erhalten!)

```
class A {
  ...
  public:
   void A_fkt();
   ...
};

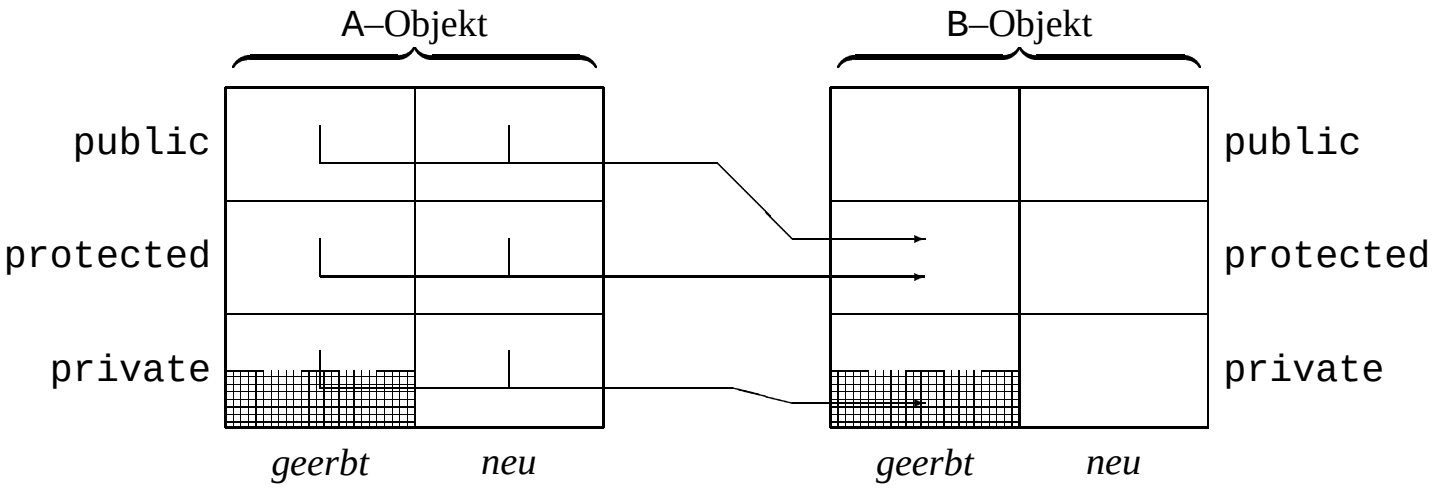
class B: public A {
  ...
  public:
    void B_fkt();
    ...
};
...
void f1(A);   // Parameter vom Typ A
void f2(A&);  // Parameter vom Typ A&
void f3(A*);  // Parameter vom Typ A*
...
B b;     // B-Objekt
A a(b);  // A-Objekt mit B-Objekt initialisieren
A *ap;   // Zeiger auf A
B *bp;   // Zeiger auf B
...
a = b;   // B-Objekt einem A-Objekt zuweisen
...
ap = &b; // Adresse eines B-Objektes einem Zeiger
         // auf A zuweisen,  Zeiger auf A koennen
         // auf B-Objekte zeigen!
...
b.A_fkt(); // A-Memberfunktion fuer B-Objekt
...        // aufrufen
f1(b);     // Funktion mit A-Parameter mit
           // B-Argument aufrufen
...
f2(b);  // Funktion mit A&-Parameter mit
        // B-Argument aufrufen
...
f3(&b); // Funktion mit A*-Parameter mit Adresse
        // eines B-Objektes aufrufen
```

```
...
b = a;  // FEHLER: kann einem B-Objekt kein
        // A-Objekt zuweisen
...
bp = &a; // FEHLER: Adresse eines A ist keine
         // Adress eines B
...
ap = &b;     // OK!
ap->B_fkt(); // FEHLER: obwohl der A-Zeiger auf
             // ein B-Objekt zeigt, kann Zugriff
             // auf neue B-Komponente nicht
             // moeglich
...
void g1(B);  // Funktion mit B-Argument
void g2(B&); // Funktion mit B&-Argument
void g3(B*); // Funktion mit B*-Argument
...
g1(a);       // FEHLER: a ist kein B-Objekt
...
g2(a);       // FEHLER: a ist kein B-Objekt
...
g3(&a);      // FEHLER: Adresse eines A ist keine
             // Adresse eines B
...
```

2. `protected`–Vererbung:

```
class A {
  public:
    ...
  protected:
    ...
  private:
    ...
};

class B : protected A {
  public:
    ...
  protected:
    ...
  private:
    ...
};
```



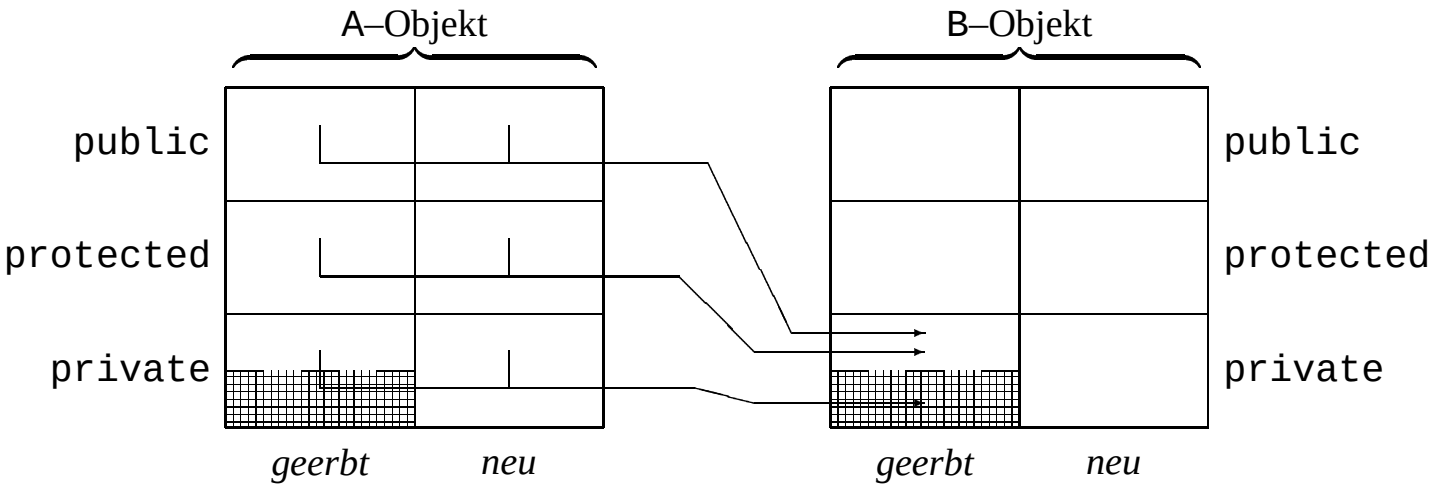
A–Schnittstelle ist für B–Objekte **nicht mehr** vorhanden!

B ist mit Hilfe von A implementiert (Implementationsdetail!).

3. `private`–Vererbung:

```
class A {
  public:
    ...
  protected:
    ...
  private:
    ...
};

class B : private A {
  public:
    ...
  protected:
    ...
  private:
    ...
};
```

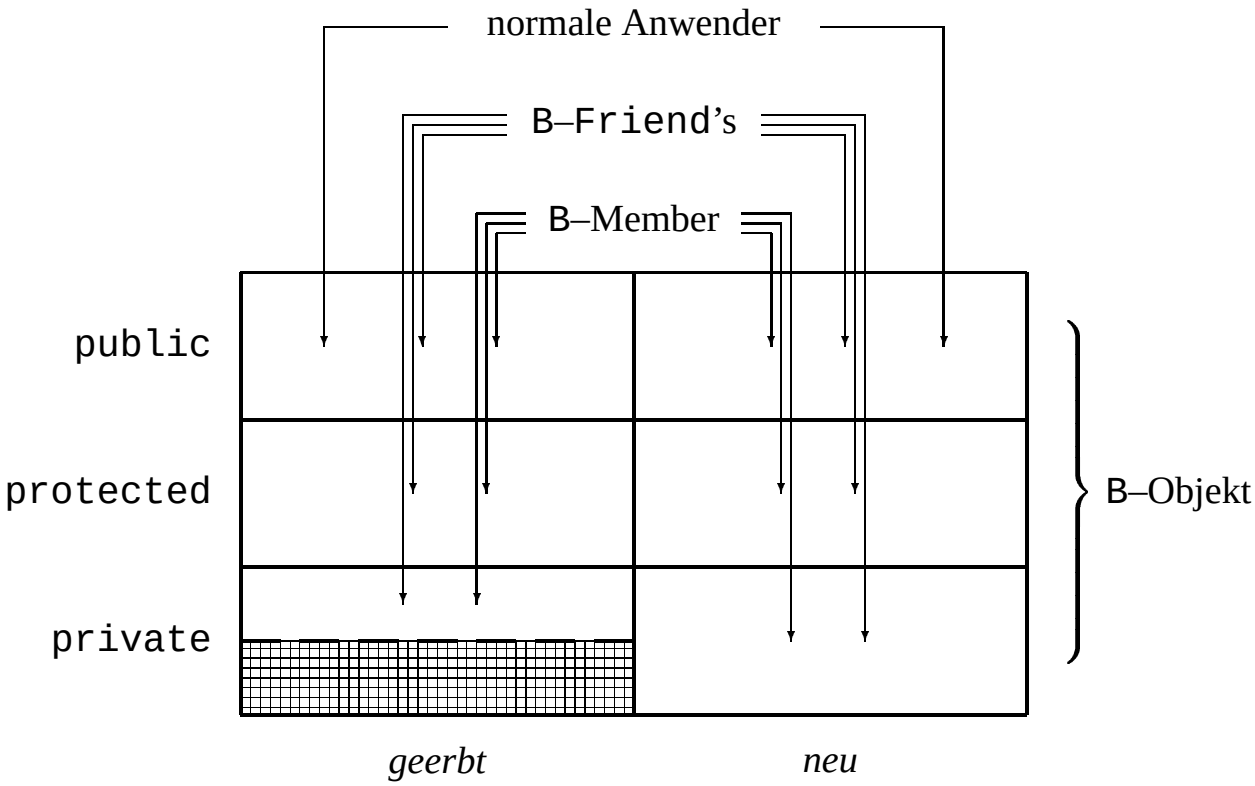


A–Schnittstelle ist für B–Objekte **nicht mehr** vorhanden!

B ist mit Hilfe von A implementiert (Implementationsdetail!).

Zugriff:

```
class B : ... {
  public:
    ...
  protected:
    ...
  private:
    ...
};
```



using–Deklaration:

```
class A {
  public:
    void f1_pub();
    void f2_pub();
    ...
  protected:
    void f1_pro();
    void f2_pro();
    ...
  private:
    ...
};

class B: private A {
 public:
  using A::f1_pub; // A::f1_pub jetzt public in B
              // A::f2_pub immer noch private in B
  ...
 protected:
  using A::f1_pro; // A::f1_pro jetz protected in B
                // A::f2_pro immer noch private in B
  ...
 private:
  ...
};
```

(öffentliche) Vererbung: Neudefinition vorhandener Funktionen

```
class A { ...
  public:
    void printOn(ostream &strm = cout) const
    { strm << "Ich bin ein A-Objekt" << endl; }
};
ostream& operator<< (ostream &strm=cout, A &a)
{ a.printOn(strm); return strm;
}

class B : public A { ...
  public:
    void printOn(ostream &strm = cout) const
    { strm << "Ich bin ein B-Objekt" << endl; }
};

// Anwendung:
int main(void)
{ A a, *ap;
  B b;

  a.printOn(cout);  // "Ich bin ein A-Objekt"
  b.printOn(cout);  // "Ich bin ein B-Objekt"

  cout << a;  // "Ich bin ein A-Objekt"
  cout << b;  // "Ich bin ein A-Objekt" !!!

  ap = &a;
  ap->printOn(cout);  // "Ich bin ein A-Objekt"
  ap = &b;
  ap->printOn(cout);  // "Ich bin ein A-Objekt" !!!
  ...
}
```

virtuelle Funktionen:

```
class A { ...
  public:
    virtual void printOn(ostream &strm = cout) const
    { strm << "Ich bin ein A-Objekt" << endl; }
};
ostream& operator<< (ostream &strm, A &a)
{ a.printOn(strm); return strm;
}

class B : public A { ...
  public:
    virtual void printOn(ostream &strm = cout) const
    { strm << "Ich bin ein B-Objekt" << endl; }
};

// Anwendung:
int main(void)
{ A a, *ap;
  B b;

  a.printOn(cout);  // "Ich bin ein A-Objekt"
  b.printOn(cout);  // "Ich bin ein B-Objekt"

  cout << a;  // "Ich bin ein A-Objekt"
  cout << b;  // "Ich bin ein B-Objekt" !!!

  ap = &a;
  ap->printOn(cout);  // "Ich bin ein A-Objekt"
  ap = &b;
  ap->printOn(cout);  // "Ich bin ein B-Objekt" !!!
  ...
}
```

virtuelle Funktionen

1. Wird eine virtuelle Funktion **über eine Referenz** oder **über eine Adresse** aufgerufen, so entscheidet **nicht der Compiler**, sondern **das Laufzeitsystem**, welche Funktion aufzurufen ist!

2. Neudefinition der virtuellen Funktion in der abgeleiteten Klasse muss gleiche Signatur haben (bei Rückgabetyp sind gewisse Änderungen möglich!).

3. Klasse mit (mindestens) einer virtuellen Funktion heißt **polymorphe Klasse**.

4. Schlüsselwort `virtual` **bereits in der Basisklasse** verwenden!

5. Bei zur Ableitung vorgesehenen Klassen sollte auf **jeden Fall der Destruktor virtuell** sein!

6. Konstruktoren können nicht virtuell sein!
   (Wird in einem Konstruktor eine virtuelle Funktion aufgerufen, wird dieser Aufruf **statisch** umgesetzt!)

7. **Default–Argumente** werden anhand der Funktionsdeklaration **statisch** umgesetzt!

8. Begriff **Polymorphie**:

   - in Basisklassen denken,
   - virtuelle Funktionen verwenden,
   - das Laufzeitsystem selbst entscheiden lassen, welche Funktion für das aktuelle Objekt die richtige ist.

Beispiel für Polymorphie:

```
class Link {
  private:
    Link * next;
  public:
    // Konstruktor
    Link() { next = 0; }
    // vorne einfuegen
    void insert( Link &a)
    { a.next = next;
      next = &a;
    }
    // vorne entfernen
    Link * exsert(void)
    { if ( next == 0)
        return 0;
      Link * tmp = next;
      next = next->next;
      tmp->next = 0;
      return tmp;
    }
    // virtueller Destruktor
    virtual ~Link() {}
    // virtuelle Ausgabefunktion
    virtual void printOn(ostream & strm) const
    { strm << " () "; }
};

//globaler Ausgabeoperator
ostream & operator<<( ostream& strm, Link &l)
{ l.printOn(strm);
  return strm;
}
```

```
class intLink: public Link {
  private:
    int wert;
  public:
    intLink(int a = 0) : wert(a) {}
    virtual void printOn(ostream& strm) const
    { strm << " (" << wert << ") ";}
};

class doubleLink: public Link {
  private:
    double wert;
  public:
    doubleLink(double a = 0.0) : wert(a) {}
    virtual void printOn(ostream& strm) const
    { strm << " (" << wert << ") ";}
};

class stringLink: public Link {
  private:
    char *wert;
  public:
    stringLink(const char *a = "")
    { wert = new char[strlen(a)+1];
      strcpy(wert,a);
    }
    virtual void printOn(ostream& strm) const
    { strm << " (\"" << wert << "\") ";}
    virtual ~stringLink()
    { delete [] wert;
    }
};
```

```
void f(Link &p)
{ // zunaechst mal ein paar int-Werte abspeichern:
  for ( int i = 0; i < 5; ++i)
  { Link * tmp = new intLink(i);
    p.insert(*tmp);
  }
  // jetzt ein paar double Werte abspeichern:
  for ( int i = 0; i < 5; ++i)
  { Link * tmp = new doubleLink(double(i) + .5);
    p.insert(*tmp);
  }
  // jetzt noch zwei Strings abspeichern
  Link * tmp = new stringLink("Hallo");
  p.insert(*tmp);
  tmp = new stringLink("Leute");
  p.insert(*tmp);
  return;
}

int main(void)
{ Link anfang, *tmp; // leere Lineare Liste
  f(anfang); // f fuellt irgendwie Lineare Liste
  // Elemente aus der Liste holen und ausgeben:
  while ( ( tmp = anfang.exsert()) != 0 )
  {  cout << *tmp;
     delete tmp;
  }
  return 0;
}
// Ausgabe:
("Leute") ("Hallo") (4.5) (3.5) (2.5) (1.5) (0.5)
(4) (3) (2) (1) (0)
```

polymorphe Typen: `dynamic cast`

1. **Adressen:**

   - `p` Adresse eines polymorphen Types,
   - `T` irgendein Typ (nicht unbedingt polymorph),
   - `dynamic_cast<T*>(p)` liefert
     - Adresse vom Typ `T *`, falls `*p` als `T`-Objekt aufgefasst werden kann
     - Adresse `0`, falls `*p` nicht als `T`–Objekt aufgefasst werden kann

```
Link anfang, *tmp;
f(anfang);
while ( ( tmp = anfang.exsert()) != 0 )
{ // Ist *tmp ein Link?
  if ( dynamic_cast<Link *>(tmp) != 0)
    cout << "Link ";
  // ist *tmp ein intLink?
  if ( dynamic_cast<intLink*>(tmp) != 0 )
    cout << "intLink " ;
  // Ist *tmp ein doubleLink?
  if ( dynamic_cast<doubleLink *>(tmp) != 0)
    cout << "doubleLink ";
  // Ist *tmp ein stringLink?
  if ( dynamic_cast<stringLink *>(tmp) != 0)
    cout << "stringLink ";
  cout << *tmp << endl;
  delete tmp;
  }

// Ausgabe
Link stringLink  ("Leute")
Link stringLink  ("Hallo")
```

```
Link doubleLink  (4.5)
Link doubleLink  (3.5)
Link doubleLink  (2.5)
Link doubleLink  (1.5)
Link doubleLink  (0.5)
Link intLink  (4)
Link intLink  (3)
Link intLink  (2)
Link intLink  (1)
Link intLink  (0)
```

2. **Referenzen:**
   - `r` Referenz auf polymorphen Typ,
   - `T` irgendein Typ (nicht unbedingt polymorph),
   - `dynamic_cast<T&>(r)` liefert
     - Referenz vom Typ `T &`, falls `r` als `T`-Objekt aufgefasst werden kann
     - Ausnahme vom Typ `bad_cast`, falls `r` nicht als `T`–Objekt aufgefasst werden kann

`type_info` für polymorphe Klassen:

```
class type_info {
  public:
    virtual ~type_info();       // polymorpher Typ

    // Vergleich von type_info-Objekten
    bool operator==(const type_info &) const;
    bool operator!=(const type_info &) const;

    // liefert (implementierungsabh.) String,
    // der den Typ beschreibt
    const char * name() const;

    ...

  private:
    // Copy-Konstruktor verbieten
    type_info(const type_info &);

    // Zuweisung verbieten
    type_info & operator=(const type_info &);
    ...
};

// Anwendung:
...
if ( typeid( *tmp ) == typeid( intLink ) )
  { ... }
else if ( typeid( *tmp ) == typeid( doubleLink ) )
  { ... }
...
// Namen ausgeben:
cout << typeid(*tmp).name() << endl;
```

Vererbung und Templates

1. Template–Klasse von “normaler“ Klasse ableiten:

```
template <class T>
class TLink: public Link {
  private:
    T wert;
  public:
    TLink(const T& a ) : wert(a) {}
    //                   ^^^^^^^
    virtual void printOn(ostream& strm) const
    { strm << " (" << wert << ") ";}
    //             ^^^^^^^
};

void f(Link &p)
{ Link *tmp;
  int i;

  // Abspeichern eines int:
  tmp = new TLink<int>(7);
  p.insert(*tmp);

  // Abspeichern eines double:
  tmp = new TLink<double>(7.5);
  p.insert(*tmp);

  // Abspeichern eines int-Zeigers
  tmp = new TLink<int *>(&i);
  p.insert(*tmp);

  ...
}
```

2. Ableiten einer Template–Klasse von anderer Template–Klasse:

```
template <class T>
class Vector {
  protected:
    T *feld;
    int len;
  public:
    // loakel Fehlerklasse:
    struct Feldzugriffsfehler {};

    // Konstruktor
    Vector( int = 10);

    // wegen dynamischer Komponente
    // Copy-Konstruktor
    Vector(const Vector<T> &);
    // Zuweisung
    const Vektor<T>& operator=(const Vector&);
    // Destruktor
    virtual ~Vector();

    // Elementzugriff:
    T& operator[](int)
      throw(Feldzugriffsfehler);

    const T& operator[](int) const
      throw(Feldzugriffsfehler);
};

template <class T>
class Vec: protected Vector<T> {
  protected:
    int base;
```

```
  public:
    // Konstruktor: l ist Feldlaenge,
    // b ist Index des ersten Feldelementes
    Vec(int l, int b) : Vector<T>(l), base(b) {};

    T& operator[](int i)
      throw(Vector<T>::Feldzugriffsfehler)
    { return Vector<T>::operator[](i - base);
    }

    const T& operator[](int i) const
      throw(Vector<T>::Feldzugriffsfehler)
    { return Vector<T>::operator[](i - base);
    }
};

// Trick: abgeleitete Template-Klasse der
// Template-Basisklasse als Argument uebergeben:

template <class T> class alt { ... };

template <class T> class neu: public alt<neu<T> >
{ ... };
```

3. “normale“ Klasse kann nur von instanziierter Template–Klasse abgeleitet werden:

```
template <class T>
class Vector { ... };

class intvector : public Vector<int>
{ ... };
```

rein virtuelle Funktionen, abstrakte Basisklasse

```
class intStack {
  public:
    virtual void push(int) = 0;
    virtual int pop() = 0;
};

intStack a;  // FEHLER: kann keinen intStack erz.!
intStack *p; // OK, Zeiger auf intStack geht!

void f(intStack);   // FEHLER: intStack unmoeglich!
void g(intStack &); // OK, Referenz geht!
void h(intStack *); // OK, Zeiger geht!

class Feld_intStack: public intStack {
  protected:
    int feld[100];
    int sp;
  public:
    // Konstruktor
    Feld_intStack() { }
    // Realisation der Funktion push
    virtual void push(int wert) { ... }
    // Realisation der Funktion pop
    virtual int pop() { ... }
};

class Listen_intStack: public intStack {
  protected:
    struct listel {
      int wert;
      listel *next;
    } *p;
```

```
  public:
    // Konstruktor
    Listen_intStack() { }
    // Realisation der Funktion push
    virtual void push(int wert) { ... }
    // Realisation der Funktion pop
    virtual int pop() { ... }
};

Feld_intStack fst;   // ist auch ein intStack!
Listen_intStack lst; // ist auch ein intStack!

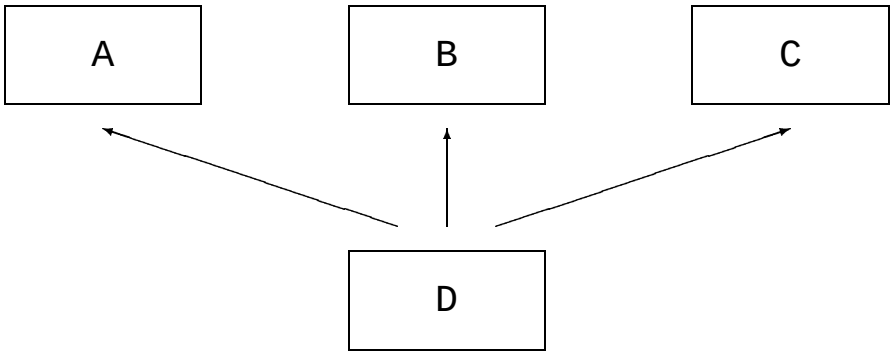
void f(intStack &); // Funktion mit
...                 // intStack-Referenz-Parameter
f(fst);             // rufe Funktion f mit
...                 // Feld_intStack als Arg. auf!
f(lst);             // rufe Funktion f mit
...                 // Listen_intStack als Arg. auf!

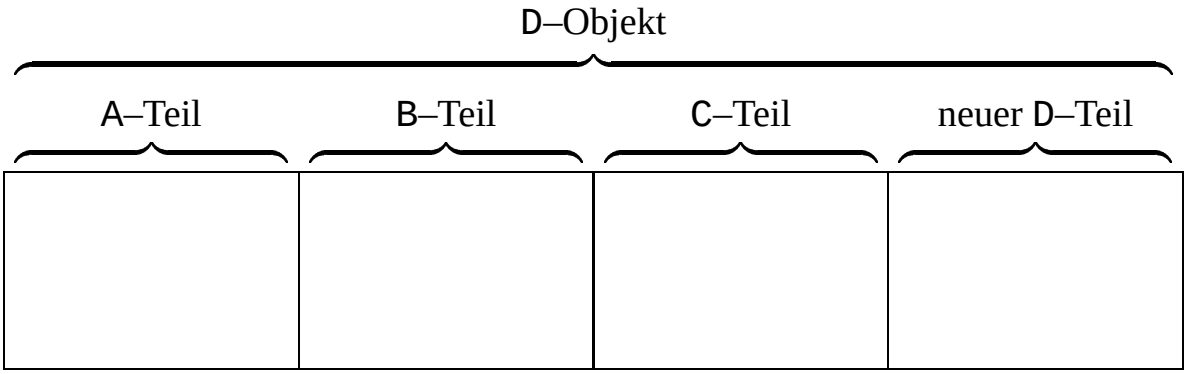
// Definition er Funktion f:
void f( intStack &stack)
{
  // verwende stack als intStack, gleichgueltig
  // welcher konkrete Stack hinter der
  // Referenz steckt!
  ...
  stack.push(7);
  ...
  erg = stack.pop();
  ...
}
```

Mehrfachvererbung:

```
class A { ... };
class B { ... };
class C { ... };

class D : public A, private B, protected C {
...
};
```





Zugriffsschutz wie bei einfacher Vererbung.

Beispiel für Mehrfachvererbung:

```
// Abstrakte Basisklasse fuer Stack's:
class intStack {
  public:
    virtual void push(int) = 0;
    virtual int pop() = 0;
};

// Template-Klasse: Vektoren mit Indexpruefung:
template <class T>
class Vector {
  protected:
    T *feld;
    int len;
  public:
    // loakel Fehlerklasse:
    struct Feldzugriffsfehler {};
    // Konstruktor
    Vector( int = 10);
    // wegen dynamischer Komponente
    // Copy-Konstruktor
    Vector(const Vector<T> &);
    // Zuweisung
    const Vektor<T>& operator=(const Vector&);
    // Destruktor
    virtual ~Vector();
    // Elementzugriff:
    T& operator[](int)
      throw(Feldzugriffsfehler);
    const T& operator[](int) const
      throw(Feldzugriffsfehler);
};
```

```
// Stack, mittels Vector<int> realisiert:
class intvectorstack :
   public intStack,      // oeffentlich: intStack
   protected Vector<int> // Implementierungsdetail!
{ int sp;

  public:
    intvectorstack() : sp(0), Vector<int>(100) {}

    virtual void push(int i) { (*this)[sp++] = i;}
    virtual int pop(){ return (*this)[--sp];}
};
```

Mehrfachvererbung: Namenskonflikte

```
class A {
  ...
  public:
    void f(int);
    ...
};

class B {
  ...
  public:
    int f(double);
    ...
};

class C: public A, public B {
...
};

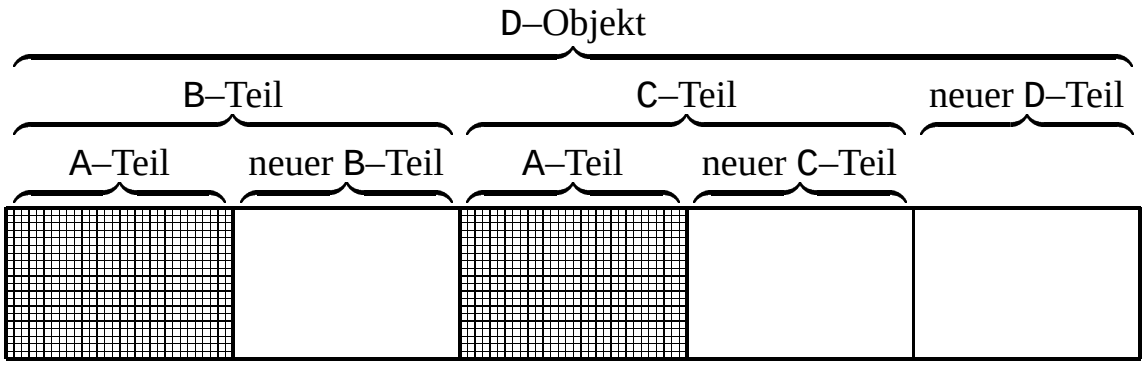
C c;
int i;
double x;
...
c.f(i); // FEHLER: A::f(int) oder B::f(double) ???
c.f(x); // FEHLER: A::f(int) oder B::f(double) ???
...
c.A::f(i);    // rufe A::f(int) auf
c.B::f(x);    // rufe B::f(double) auf
c.A::f(x);    // rufe A::f(int) auf, wandle hierzu
              // den double-Wert von x nach int um!
...
```

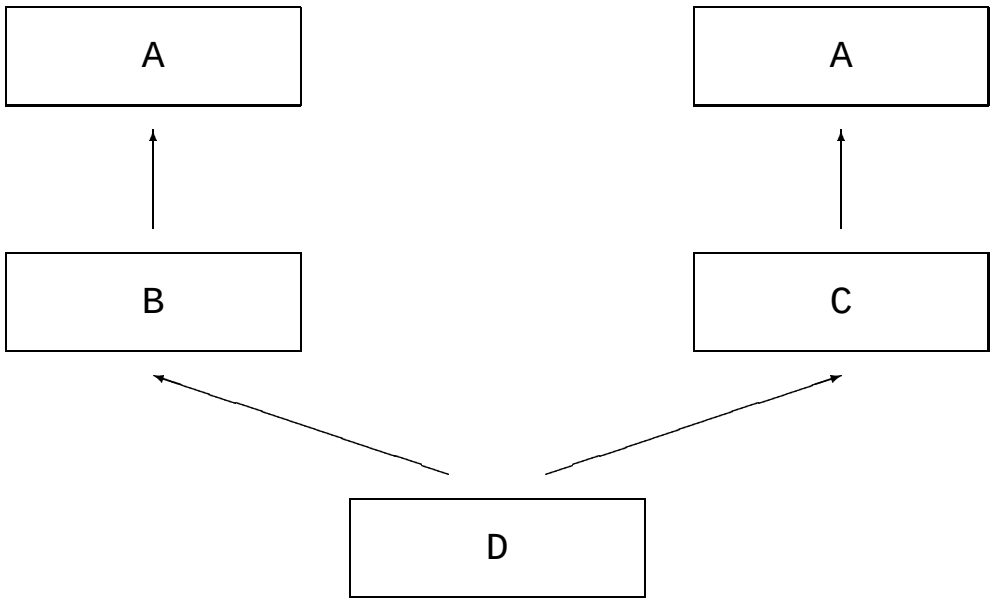
mehrfache Basisklasse:

```
class A {
  ...
  public:
    void f(void);
    ...
};

class B: public A { ... };
class C: public A { ... };

class D: public B, public C { ... };
```

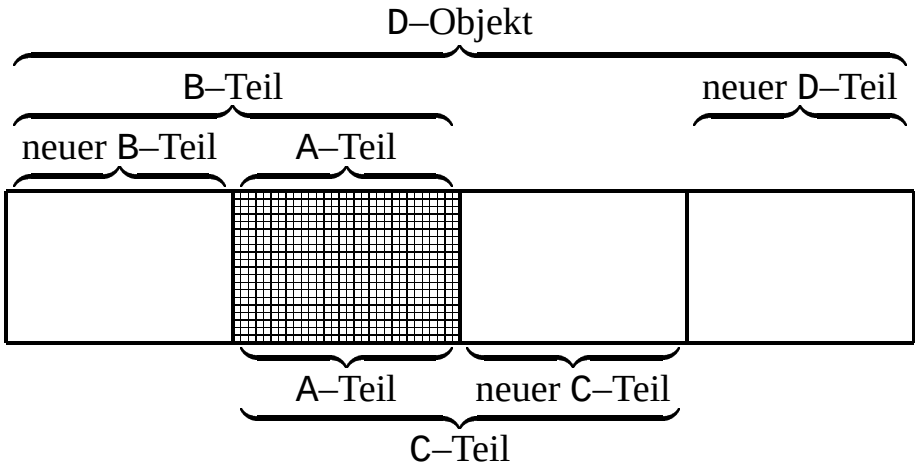


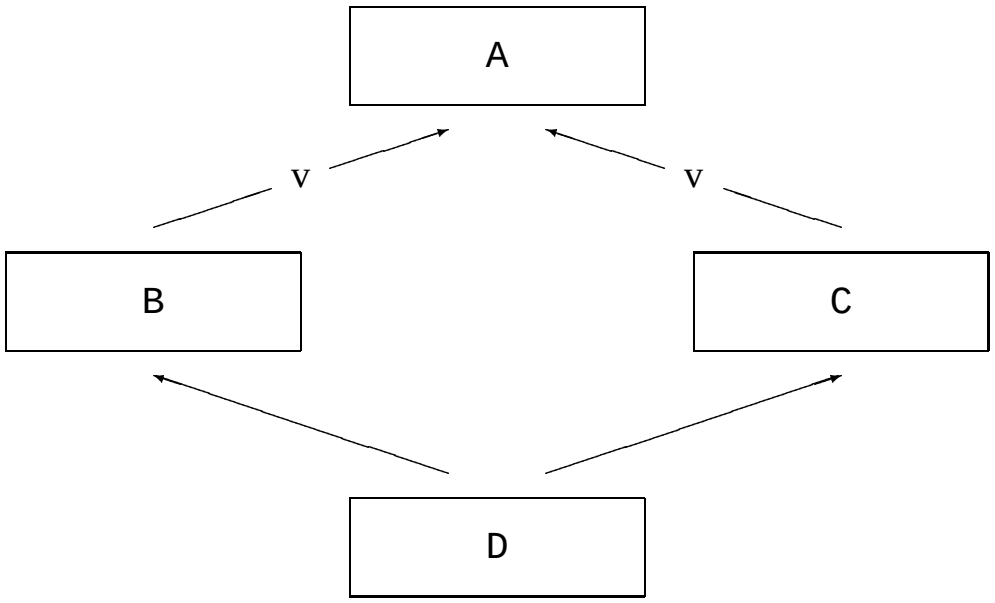


virtuelle Basisklasse:

```
class A { ... };

class B: virtual public A { ... };
//       ^^^^^^^
class C: virtual public A { ... };
//       ^^^^^^^
class D: public B, public C { ... };
```





Mehrfachvererbung:

1. Mehrfache direkte Basisklasse nicht möglich:

```
class A { ... };

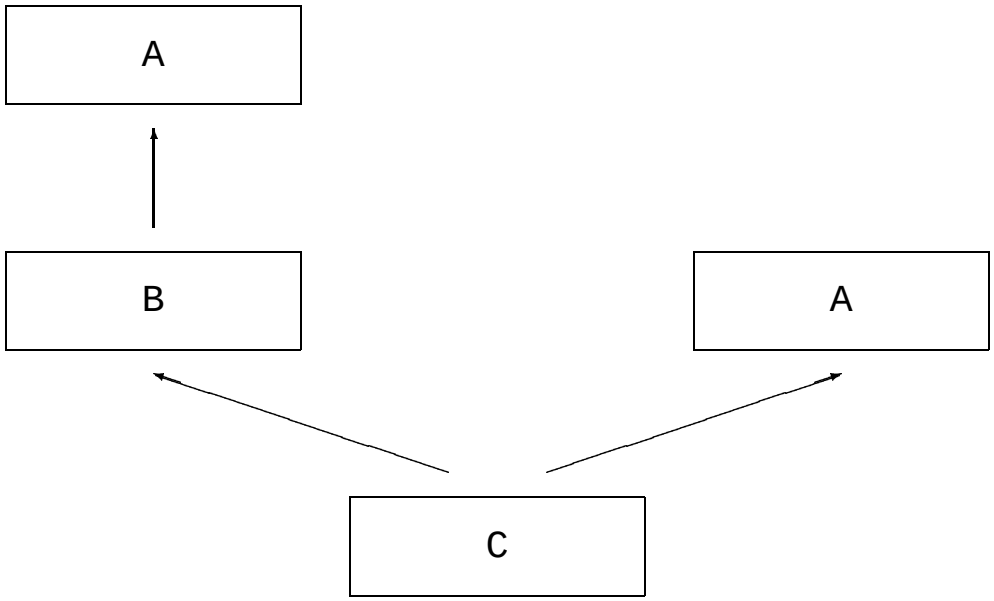
// FEHLER: mehrfache direkte Basisklasse!!
class B: public A, public A { ... };
...
```

2. Gleichzeitig direkte und indirekte Basisklasse geht:

```
class A { ... };
class B : public A { ... };

class C : public B, public A { ... };
...
```

   Schaubild:



3. Hat eine Klasse `D` eine Klasse `A` (ggf. auf mehreren Wegen) virtuell geerbt, so muss ein `A`–Konstruktor in der Initialisierungsliste des `D`–Konstruktors aufgeführt sein (falls nicht der parameterlose `A`–Konstruktor genommen werden soll!).

4. Hat eine Klasse `D` eine Klasse `A` (ggf. auf mehreren Wegen) virtuell geerbt, so kann auf eine Komponente, etwa eine Member–Funktion, des virtuellen `A`–Teils (falls explizite Qualifikation aufgrund von Mehrdeutigkeiten erforderlich ist) über den Namen der virtuellen Klasse direkt zugegriffen werden.

5. Gleichzeitig virtuelle und direkte Basisklasse nicht möglich:

```
class A { ... };

class B : virtual public A { ... };
class C : virtual public A { ... };

class D : public B, public C, public A { ... };
   // FEHLER: gleichzeitig direkte und virtuelle
   // Basisklasse geht nicht!!!
...
```

6. Gleichzeitig virtuelle und nicht virtuelle, aber indirekte Basisklasse ist möglich:

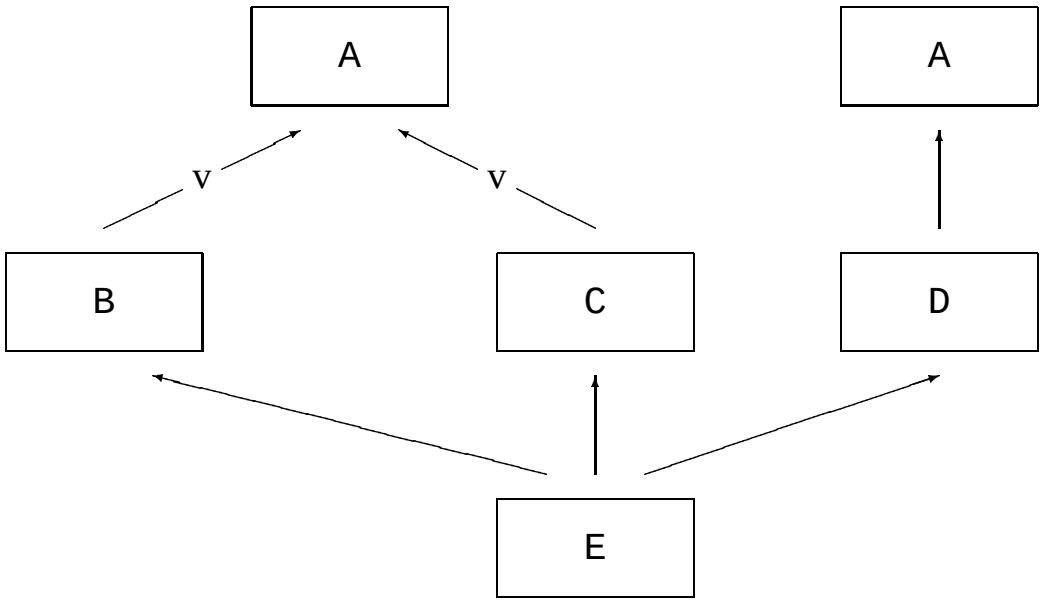
```
class A {
  ...
  public:
    void f(void);
    ...
};

class B: virtual public A { ... };
class C: virtual public A { ... };

class D: public A { ... };

class E: public B, public C, public D { ... };
...
```

Schaubild:



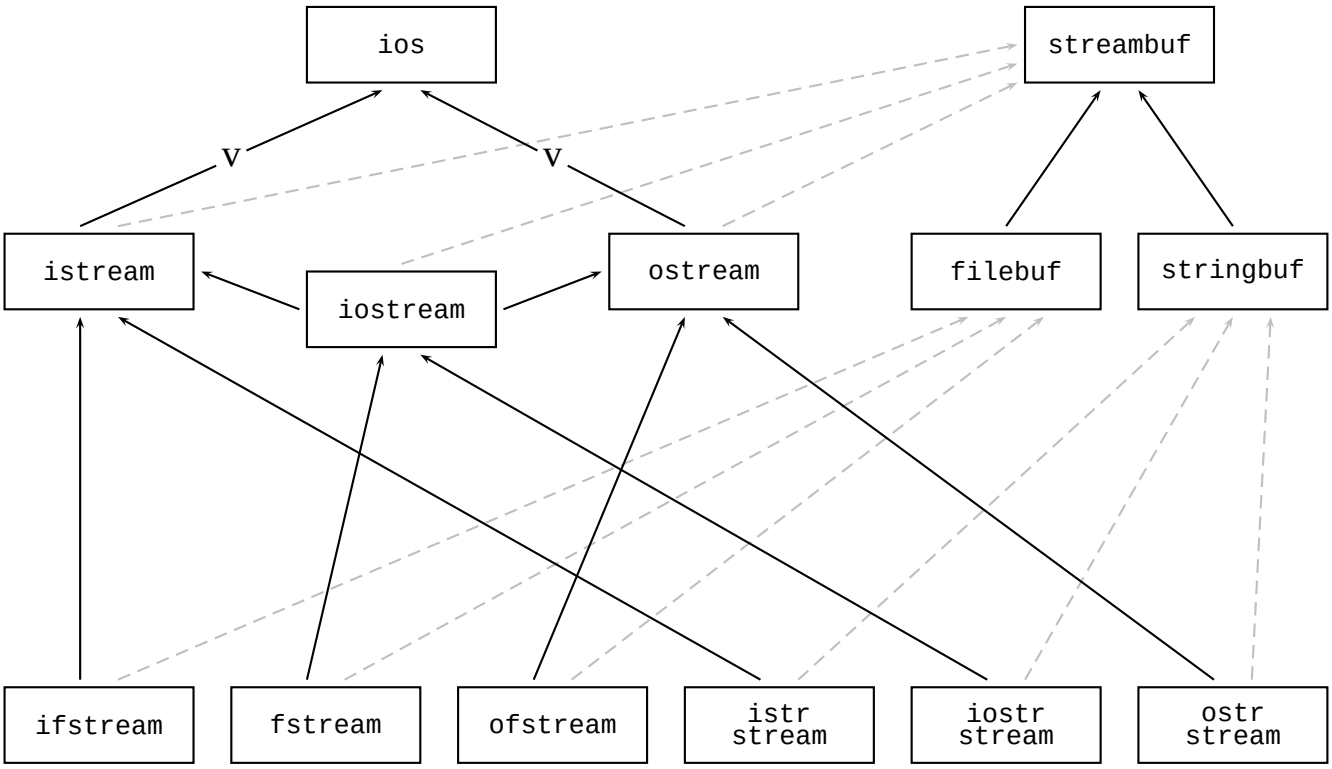
Anwendung:

```
E e;

e.D::A::f();  // bei diesem Aufruf arbeitet die
         // Funktion f mit dem nicht virtuellen,
         // ueber D geerbten  A-Teil von e

// folgende drei Aufrufe sind gleichwertig,
// die Funktion f arbeitet jeweils mit dem
// virtuellen, ueber B und C geerbten A-Teil
e.A::f();
e.B::A::f();
e.C::A::f();
...
```

Klassen zur Ein–/Ausgabe mit Zeichen vom Typ `char`



Bedeutung der Klassen:

- `ios` Grundlegende Streameinstellungen, Fehlerzustände,
- `istream` gegenüber `ios` zusätzlich Eingabe–Operationen,
- `ostream` gegenüber `ios` zusätzlich Ausgabe–Operationen,
- `iostream` Ein– und Ausgabeoperationen,
- `ifstream` Lesen von Dateien,
- `ofstream` Ausgabe auf Dateien,
- `fstream` Ein– und Ausgabe von bzw. auf eine Datei,
- `istrstream` Lesen von Strings,
- `ostrstream` Schreiben auf Strings,
- `iostrstream` Lesen und Schreiben in und auf Strings.

Ausgabefunktionen:

- Ausgabeboerator `<<` (für zahlreiche Typen vordefiniert!)
- `ostream& ostream::put(char);`
- `ostream& ostream::write(const char *p, streamsize n);`
- `ostream& ostream::flush();`

Ausgabemanipulatoren:

- `endl`, `flush` und `ends`
  "Aufruf"etwa: `cout << ... << endl;`

Ausgabefeldbreite erfragen und setzen:

- `streamsize ios::width();`
  Abfragen der Feldbreite.
- `streamsize ios::width(streamsize n=0);`
  Setzen der Feldbreite (Argument `0`: Setzen auf Standardwert).
- `cout << ... << setw(n) << ...;`
  Manipulator zum Setzen der Feldbreite (`<iomanip>` includen).

Präzision erfragen und setzen:

- `streamsize ios::precision();`
  Abfragen der Präzision.
- `streamsize ios::precision(streamsize n=0);`
  Setzen der Präzision (Argument `0`: Setzen auf Standardwert).
- `cout << ... << setprecision(n) << ...;`
  Manipulator zum Setzen der Präzision.

Füllzeichen erfragen und setzen:

- `char ios::fill();`
  Abfragen des Füllzeichens.
- `char ios::fill(char c);`
  Setzen des Füllzeichens.
- `cout << ... << setfill(c) << ...;`
  Manipulator zum Setzen des Füllzeichens.

Weitere Manipulatoren:

- `cout <<...<< right << ...;` rechtsbündige Ausgabe
- `cout <<...<< left  << ...;` linksbündige Ausgabe
- `cout <<...<< internal <<...;` Vorz. links, Rest rechts
- `cout <<...<< dec  << ...;` dezimal
- `cout <<...<< oct  << ...;` oktal
- `cout <<...<< hex  << ...;` hexadezimal
- `cout <<...<< setbase(n)  << ...;` Basis n (2,8,16)
- `cout <<...<< scientific  << ...;` mit Exp.
- `cout <<...<< fixed  << ...;` ohne Exp.
- `cout.unsetf(ios::floatfield);`
  Zurückstellen auf Standard, moderate Werte werden "`fixed`", andere "`scientific`" ausgegeben, **kein** Manipulator
- `cout <<...<< showbase << ...;` mit `0` bzw. `0x`
- `cout <<...<< noshowbase << ...;` ohne`0` bzw. `0x`
- `cout <<...<< showpoint<< ...;` mit Dez.–Pkt.
- `cout <<...<< noshowpoint << ...;` ohne Dez.–Pkt.
- `cout <<...<< showpos << ...;` mit Vorz.
- `cout <<...<< noshowpos  << ...;` ggf. ohne Vorz.
- `cout <<...<< uppercase  << ...;` Großbuchst.
- `cout <<...<< nouppercase  << ...;` Kleinbuchst.

Einstellungen direkt beeinflussen:

(Ganzzahliger) Datentyp `ios::fmtflags`.

Funktionen, um auf Einstellungen direkt zuzugreifen und zu setzen:

- `fmtflags ios::flags();`
- `fmtflags ios::flags(fmtflags neu);`
- `fmtflags ios::setf(fmtflags fmtfl);`
- `fmtflags ios::setf(fmtflags f, fmtflags m);`
- `fmtflags ios::unsetf(fmtflags fmtfl);`

Zugehörige Masken und Bit's:

- Maske: `ios::adjustfield`, zugehörige Bit's:
  `ios::right`, `ios::left`, `ios::internal`
- Maske: `ios::basefield`, zugehörige Bit's:
  `ios::dec` , `ios::oct` , `ios::hex`
- Maske: `ios::floatfield`, zugehörige Bit's:
  `ios::scientific`, `ios::fixed`
- Sonstige Bit's (ohne zugehörige Maske):
  `ios::showbase`, `ios::showpoint`, `ios::showpos`, `ios::uppercase`, `ios::unitbuf`

Manipulatoren zur direkten Beeinflussung der Einstellungen:

- `cout << ... << setiosflags(fmtfl) << ... ;`
  entspricht: `cout.setf(fmtfl);`
- `cout << ... << resetiosflags(fmtfl) << ... ;`
  entspricht: `cout.unsetf(fmtfl);`

Eingabefunktionen:

- Der Eingabeoperator `>>` (für zahlreiche Typen vordefiniert!)
- `int istream::get();`
  liest ein Zeichen, gibt Zeichen (oder `EOF`) zurück
- `istream& istream::get(char &c);`
  liest ein Zeichen und speichert es im Argument, gibt Stream zurück.
- `istream& istream::get( char *p, streamsize n );`
  liest maximal bis ausschließlich dem nächsten Zeilenvorschub `'\n'` und höchstens `n-1` Zeichen. Hängt `'\0'` an das Gelesene an.
- `istream& istream::get(char*p,streamsize n,char c);`
  wie oben, Zeichen `c` anstelle des `'\n'`.
- `istream& istream::getline(char*p,streamsize n);`
  wie oben, `'\n'` wird jedoch gelesen, aber nicht abgespeichert.
- `istream& istream::getline(char*p,streamsize n,char c);`
  wie oben, Zeichen `c` anstelle des `'\n'`.
- `istream& istream::read(char *p,streamsize n);`
  liest `n` Zeichen (bei `EOF` auch weniger), hängt kein `'\0'` an.
- `int istream::peek();` liefert nächstes Zeichen, ohne es zu lesen.
- `istream& istream::unget();` schreibt zuletzt gelesenes Zeichen "zurück".
- `istream& istream::putback(char c);` schreibt das Zeichen `c` zurück (`c` muss zuletzt gelesenes Zeichen sein!).
- `istream& istream::ignore();` überliest ein Zeichen
  `istream& istream::ignore(streamsize n);` überliest `n` Zeichen.
- `int istream::sync();` leert Eingabepuffer.

Eingabeformatierung:

- Feldbreite, Funktion `witdh()` und Manipulator `setw(n)` wie bei Ausgabe
- Basis: Bitfeld `ios::basefild` und Bits `ios::dec`, `ios::oct` und `ios::hex` und Funktionen und Manipulatoren wie bei der Ausgabe.
- Neues Bit: `ios::skipws`: Überlesen von Zwischenraumzeichen.
  Zugehörige Manipulatoren: `cin >> skipws;` und `cin >> noskipws;` (Überlesen ein– bzw. ausschalten).
- Manipulator `cin >> ws;` zum einmaligen Überlesen von Zwischenraumzeichen.

Fehlerzustände eines Streams:

Fehlerbits:

`ios::goodbit`	alles ok
`ios::eofbit`	Ende des Streams erreicht
`ios::failbit`	letzter Lesevorgang fehlerhaft
`ios::badbit`	Stream unbrauchhbar

Funktionen zur Abfrage der einzelnen Fehlerbits:

- `bool ios::good();`
- `bool ios::eof();`
- `bool ios::fail();`
- `bool ios::bad();`

Ggf. "Umwandlung" eines Streams nach `bool`:

```
while( cin ) {...}
```

entspricht:

```
while ( ! cin.fail()) {...}
```

Datentyp `ios::iostate` beinhaltet Fehlerzustände.
Weitere Funktionen:

- `ios::iostate ios::rdstate();`
  liefert augenblicklichen Fehlerzustand.
- `void ios::clear();`
  löscht augenblicklich gesetzten Fehlerflaggen.
- `void ios::clear(ios::iostate);`
  setzt den Fehlerzustand auf den angegebenen.
- `void ios::setstate(ios::iostate);`
  fügt die im Argument angegebenen Flaggen dem Fehlerzustand hinzu.

Beispiel: Eingabe von Brüchen:

```
class Bruch {
  protected:
    int zaehler;
    int nenner;
  public:
    // Konstruktor:
    Bruch(int =0, int =1);
    // multiplikative Zuweisung:
    const Bruch & operator*=(const Bruch &);
    // Ausgabe auf Stream:
    void printOn(ostream & =cout) const;
    // Lesen von Stream:
    void scanFrom(istream & =cin);
    ...  // sonstiges:
};

// Ausgabeoperator << global ueberladen
ostream& operator<<(ostream &strm, const Bruch &b)
{ b.printOn(strm); return strm; }

// Eingabeoperator >> global ueberladen
istream& operator>>(istream &strm, Bruch &b)
{ b.scanFrom(strm); return strm; }

void Bruch::printOn( ostream & strm) const
{ strm << zaehler << '/' << nenner;
}

void Bruch::scanFrom(istream &strm)
{ int z, n;

  strm >> z;  // falls Fehler: failbit durch System
```

```
  strm >> ws; // Zwischenraumzeichen ueberlesen:

  // optionales '/' einlesen
  if ( strm.peek() == '/' ) // naechstes Zeichen
    strm.get();             // falls '/', dann lesen!

  strm >> n;  // falls Fehler: failbit durch System

  if ( !strm )  // falls jetzt bereits Fehler
    return;

  // Nenner == 0 ???
  if ( n == 0)
  { strm.setstate (ios::failbit);
    return;  // failbit setzen und beenden
  }

  // Nenner positiv machen:
  if ( n < 0 )
  { zaehler = -z;
    nenner  = -n;
  }
  else
  { zaehler = z;
    nenner  = n;
  }

  return;
}
```

Manipulatoren (ohne Argumente) selbstdefinieren:

1. Ausgabemanipulator:
   - zugrundeliegende Operatorfunktion:

```
ostream&
ostream::operator<<(ostream& (*m)(ostream &))
{ return (*m) (*this); }
```

   - Verwendung:

```
// Manipulatorfunktion definieren:
ostream& man_fkt(ostream &strm) { ... }

// Aufruf:
cout << man_fkt;
// wird umgesetzt zu: man_fkt(cout)
```

2. Eingabemanipulator:
   - zugrundeliegende Operatorfunktion:

```
istream&
istream::operator>>(istream& (*m)(istream &))
{ return (*m) (*this); }
```

   - Verwendung (etwa):

```
// Manipulatorfunktion definieren:
istream& ignoreline(istream &strm)
{ char c;
  // Alles bis einschliesslich
  // naechstem '\n' lesen:
  while ( strm.get(c) && c != '\n') ;
  return strm;
}
// Aufruf:
cin >> ignoreline;   // wird umgesetzt zu:
                     // ignoreline(cin)
```

Manipulator (mit einem Argument) selbstdefinieren:

Beispiel: n Blanks ausgeben (etwa $n = 10$):
`cout << ... << space(10) << ... ;`

1. generelle Vorbereitung:

   (a) Klasse definieren, so dass der Funktionsaufruf `space(10)` ein Objekt dieser Klasse zurückgibt. In der Klasse muss die gewünschte Funktionalität enthalten sein!

```
class manipulator_objekt {
   private:
// Zeiger auf Manipulator-Funktion
     ostream& (*mo_fkt_ptr)(ostream &, int);
// Argument fuer Manipulator-Funktion
     int arg;
   public:
// Konstruktor:
     manipulator_objekt(
        ostream& (*fkt_ptr)(ostream &, int),
        int wert)
         : mo_fkt_ptr(fkt_ptr), arg(wert) {}
// Anwenden:  Aufruf der gespeicherten Funktion
//            mit dem gespeicherten Argument
     ostream & anwenden( ostream & strm)
     { // gespeicherte Funktion auf strm mit
       // gespeichertem Argument anwenden:
       return (*mo_fkt_ptr)( strm, arg);
     }
};
```

(b) Klasse, in der die Manipulator-Funktion abgespeichert ist und welche beim Aufruf der Operatorfunktion `operator()(int)` ein passendes Objekt der Klasse `manipulator_objekt` zurückgibt:

```
class manipulator {
  private:
    // Zeiger auf manipulator_funktion
    ostream & (*fkt_ptr)( ostream &, int);
  public:
// Konstruktor:
// Parameter ist passender Funktionszeiger:
    manipulator( ostream& (*f)(ostream &, int))
       : fkt_ptr(f) { }
// Ueberladung des ()-Operators:
    manipulator_objekt operator()(int n)
    { return manipulator_objekt( fkt_ptr, n);}
};

// globale Ueberladung des Ausgabeoperators <<
// fuer ein manipulator_objekt:
ostream&
operator<<(ostream& strm,manipulator_objekt mo)
{  mo.anwenden(strm); // ueberkreuzt anwenden:
}
```

2. konkrete Manipulator–Funktion definieren:

```
// Manipulator-Funktion
ostream& spaces(ostream & strm, int n)
{ for ( int i = 0; i < n; ++i)
    strm << ' ';
  return strm;
}
```

3. Manipulator erzeugen, diesen mit der definierten Manipulator–Funktion `spaces` verknüpfen:

   ```
   manipulator space(spaces);
   ```

   (Im Objekt `space` der Klasse `manipulator` ist jetzt in der Funktions–Zeigerkomponente `fkt_ptr` die Adresse der Manipulator–Funktion `spaces` abgelegt!)

4. Manipulator anwenden:

   ```
   cout << space(10)
                ①
   cout << manipulator_objekt(spaces,10)
                              ②
                spaces(cout,10)
   ```

   ① `space(10)`
   Bei diesem Aufruf der Operator–Funktion `operator()` für das Objekt `space` wird ein `manipulator_objekt` zurückgegeben, in dem die Manipulator–Funktion `spaces` (als Funktionszeiger) und das entsprechende ganzzahlige Argument `10` abgespeichert ist.

   ② `cout << manipulator_objekt(spaces,10)`
   Durch die Überladung von `operator<<` für solche `manipulator_objekte` wird die gespeicherte Funktion mit dem gespeicherten Argument für den `ostream cout` aufgerufen!

Dateibehandlung:

1. Klassen:
   - `ofstream` (von `ostream` abgeleitet) Schreiben auf Datei
   - `ifstream` (von `istream` abgeleitet) Lesen von Datei
   - `fstream` (von `iostream` abgeleitet) Lesen von und Schreiben auf Datei
2. Verwendung:

```
void f(void)
{
  // Datei zur Ausgabe oeffnen
  ofstream ausgabe("Ausgabe.txt");

  // Datei zur Eingabe oeffnen
  ifstream eingabe("Eingabe.txt");

  ...
  ausgabe << ... << endl << ...;
  eingabe >> ... >> ws >> ...;

  ...
} // hier beim Ende des Blockes wird der
  // Destruktor fuer die beteiligten Stroeme
  // aufgerufen - diese sorgen fuer das
  // Schliessen der Dateien!
```

Feinheiten zum Öffnen und Schließen von Dateien:

1. Modus beim Öffnen angeben:

`ios::in`	Zum Lesen öffnen.
`ios::out`	Zum Schreiben öffnen.
`ios::app`	Jede Schreiboperation erfolgt am Dateiende.
`ios::ate`	Datei wird beim Öffnen (einmalig) auf's Dateiende positioniert.
`ios::binary`	Binär–Modus statt Text–Modus.
`ios::trunc`	Datei wird beim Öffnen auf Länge 0 gekürtzt.

```
ofstream ausgabe("Ausgabe.txt", ios::out);
ifstream eingabe("Eingabe.txt", ios::in);
```

2. Öffnen überprüfen:

```
...
ifstream eingabe("Eingabe.txt");
if ( !eingabe )
{ ... }    // Oeffnen hat nicht geklappt!
...
```

3. Öffnen und Schließen mit Member–Funktionen:

```
ostream ausgabe("Ausgabe.txt");
...
ausgabe.close(); // ausserplanm. Schliessen,
       // ostream ausgabe noch vorhanden, aber
       // nicht mehr mit einer Datei verknuepft!
...
ausgabe.open("Ausgabe2.txt", ios::out);
   // ausgabe mit anderer Datei verknuepfen!
...
```

Dateipositionierung:

1. Datentypen, Konstanten:
   - `pos_type`: in Bibliothek definierter ganzzahliger Type für Positionsangaben.
   - `ios::seekdir`: Datentyp zur Beschreibung eines Bezugspunktes innerhalb einer Datei.
     Konkrete Bezugspunkte:

`ios::beg`	Dateianfang
`ios::cur`	augenblickliche Dateiposition
`ios::end`	Dateiende

   - `off_type`: in den Bibliothek definierte ganzzahliger Typ zur Beschreibung von Positionsdifferenzen.
2. Positionierung eines `ostream`:
   - `pos_type ostream::tellp();` Abfragen der aktuellen Position.
   - `ostream& ostream::seekp(pos_type pos);`
     Setzt Position auf Wert.
   - `ostream& ostream::seekp(off_type anz, ios::seekdir bezug);`
     setzt neue Position auf `anz` hinter/vor den angegebenen Bezugspunkt `bezug`.
3. Positionierung eines `istream`:
   - `pos_type istream::tellg();` Abfragen der aktuellen Position.
   - `istream& istream::seekg(pos_type pos);`
     Setzt Position auf Wert.
   - `istream& istream::seekg(off_type anz, ios::seekdir bezug);`
     setzt neue Position auf `anz` hinter/vor den angegebenen Bezugspunkt `bezug`.

Ströme und Ausnahmen:
(funktioniert mit unserem Compiler noch nicht!)

Man kann einen Strom so einstellen, dass er beim Setzten eines Zustandsbits (etwa `ios::failbit`) eine Ausnahme vom Typ `ios::failure` auswirft!

- `void ios::exceptions(ios::iostate maske);`
  In der Maske angegebene Zustandsbits werden mit Ausnahme verknüpft.
  Argument `0`: alle Verknüpfungen aufheben.
- `ios::iostate ios::exceptions();`
  Abfragen, für welche Zustandsbits eine Verknüpfung mit einer Ausnahme besteht.

Beispiel:

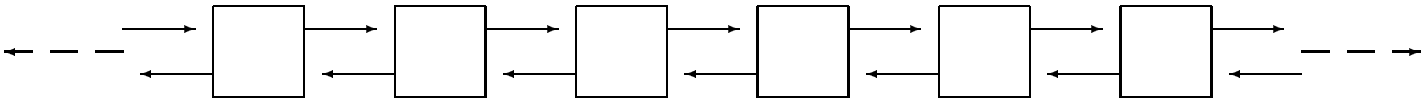
```
...
// Alte Einstellungen merken
ios::iostate alt = cin.exceptions();
// immer, wenn failbit gesetzt wird, wird auch eine
// Ausnahme vom Typ ios::failure ausgeloest.
cin.exceptions(ios::failbit);
...
try { ...
      cin >> ...
      ...
}
catch( ios::failure f)
{ cerr << "Eingabefehler!" << endl;
  ...
}
// alte Einstellungen restaurieren:
cin.exceptions(alt);
...
```

Standard–Container

1. `vector<T>`
   - Header–Datei `<vector>`
   - Verallgemeinerung von Feldern:

   - darauf optimiert, am Ende zu “wachsen“.
2. `deque<T>`
   - Header–Datei `<deque>`
   - Verallgemeinerung von Feldern:

   - darauf optiomiert, nach beiden Seiten zu “wachsen“.
3. `list<T>`
   - Header–Datei `<vector>`
   - doppelt verkettete Liste:



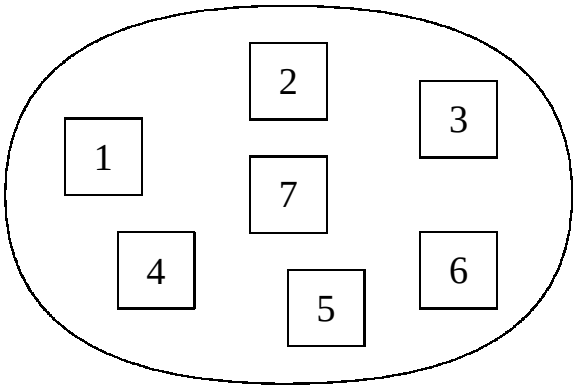
   - daraufhin optimiert, an beliebigen Stellen Elemente einzufügen und zu entfernen.

4. `set<T>` bzw. `multiset<T>`

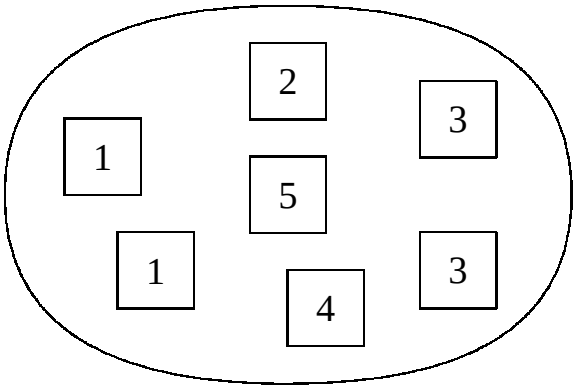
   – Header–Datei `<set>`

   – Mengenklassen (`set<T>` ohne, bei `multiset<T>` mit gleichen Elementen):

   `set<int>`:



   `multiset<int>`:



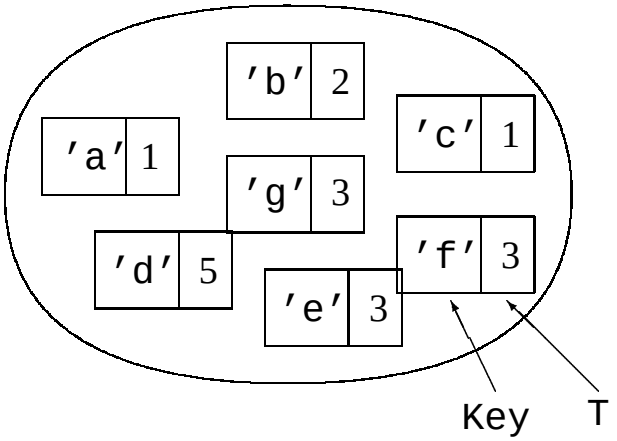
   – darauf optimiert, Elemente schnell aufzufinden.

5. `map<Key,T>` bzw. `multimap<Key,T>`

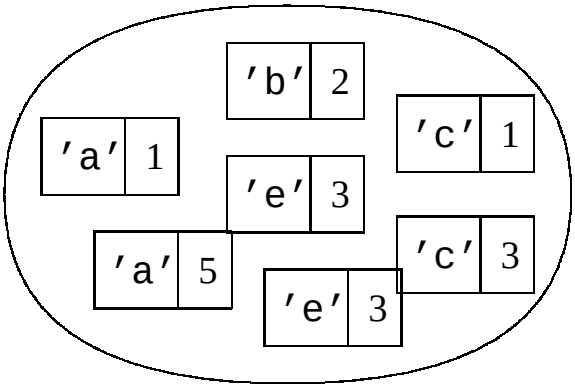
   – Header–Datei `<map>`

   – Mengen von Paaren (`map<Key,T>` ohne, bei `multimap<Key,T>` mit gleichen Schlüsseln):

   `map<char,int>`:



   `multimap<char,int>`:

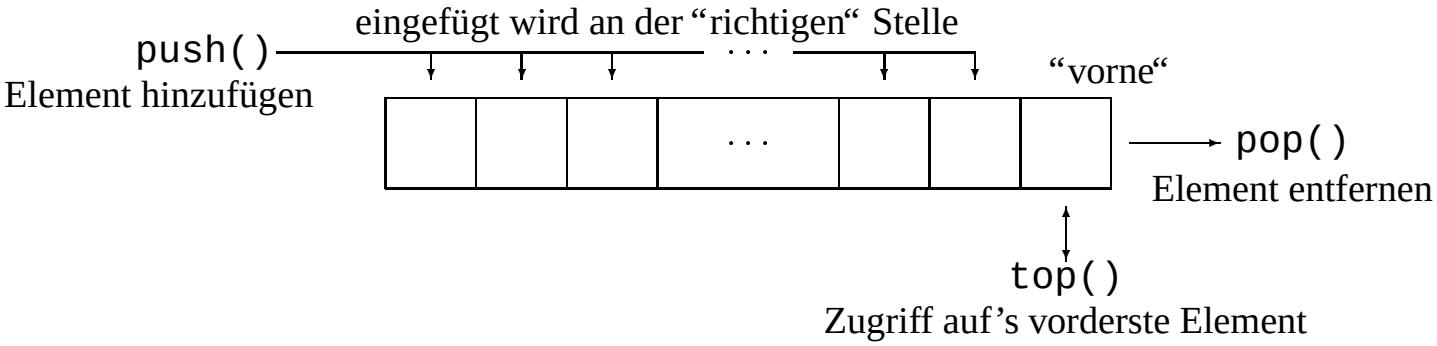


   – auf schnellen Zugriff über Schlüssel optimiert.

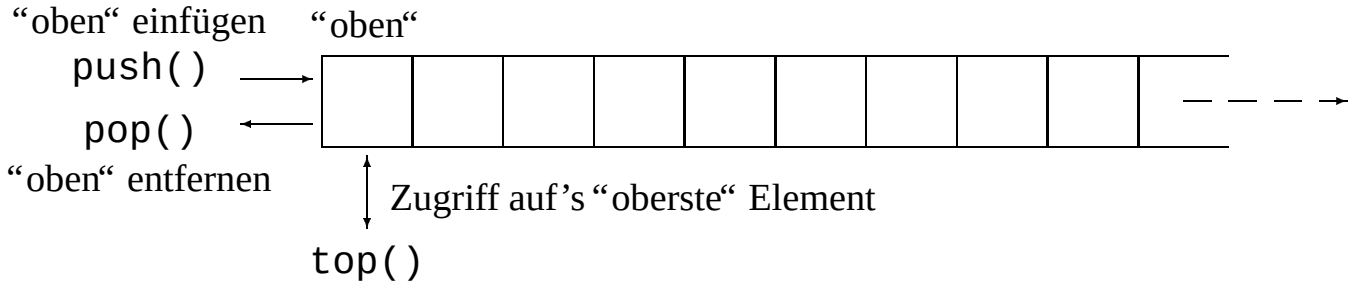
Container–Adapter
(spezielle Schnittstellen zu Container–Klassen)

1. `queue<T>`
   - Header–Datei `<queue>`
   - aufbauend auf `deque<T>` oder `list<T>`
   - Funktionalität: “vorne“ einfügen, “hinten“ entfernen:

2. `priority_queue<T>`
   - Header–Datei `<queue<T>`
   - aufbauend auf `deque<T>` oder `list<T>`
   - Funktionalität: Einfügen anhand Priorität, “vorne“ entfernen:



3. `stack<T>`
   - Header–Datei `<stack>`
   - Funktionalität: “oben“ einfügen und entfernen:



Die Containerklasse `vector<T>`**:**

Konstruktoren:

```
vector<T>::vector();  // Std.-Konstr., leerer Vektor
vector<T>::vector(const vector<T>&); // Copy-Konstr.

vector<T>::vector(size_type n); // n Std.-Elemente
vector<T>::vector(size_type n, const T& wert); // n
                                  // Kopien von wert
```

Größe und Kapazität:

```
size_type vector<T>::size();
size_type vector<T>::capacity();
bool vector<T>::empty();
size_type vector<T>::max_size();

void vector<T>::reserve(size_type n);

void vector<T>::resize(size_type n);
void vector<T>::resize(size_type n, const T& wert);

void vector<T>::push_back(const T& wert);//anhaengen
void vector<T>::pop_back();      // Element loeschen
```

Elementzugriff:

```
T& vector<T>::operator[](size_type n);
const T& vector<T>::operator[](size_type n) const;

T& vector<T>::at(size_type n);
const T& vector<T>::at(size_type n) const;
```

```
T& vector<T>::front();
const T& vector<T>::front() const;

T& vector<T>::back();
const T& vector<T>::back() const;
```

Zuweisungen:

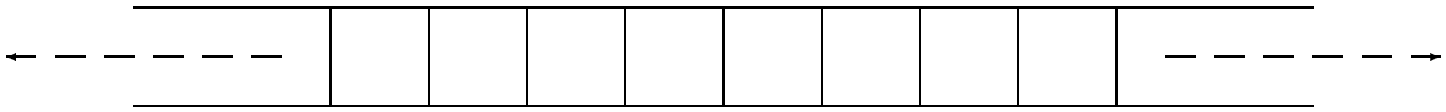
```
const vector<T>&
  vector<T>::operator=(const vector<T> &);

void vector<T>::assign(size_type n, const T &wert);

void vector<T>::assign( iter anf, iter end);

void vector<T>::swap(vector<T> &);
// Aufruf:  a.swap(b)
void swap(vector<T> &a, vector<T> &b);
// Aufruf: swap(a,b);
```

Die Containerklasse `deque<T>`:

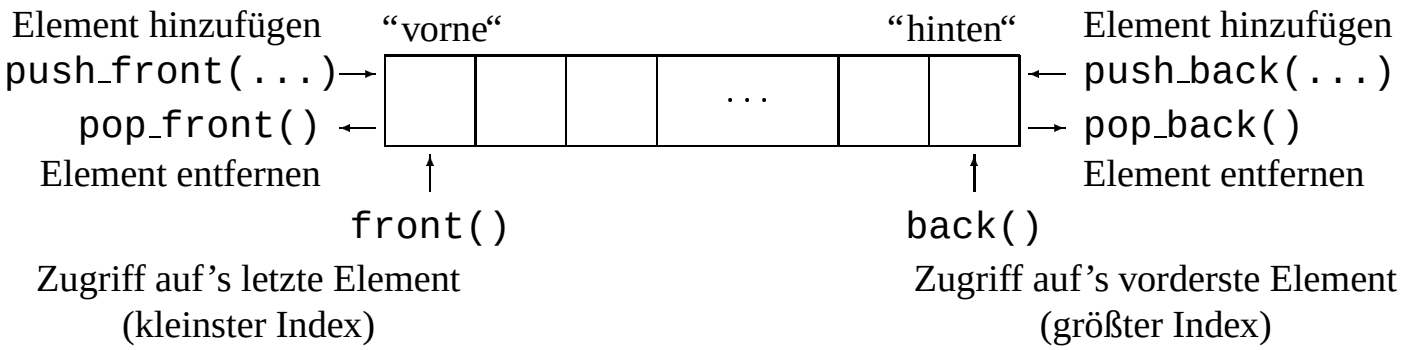


Die wesentliche Funktionalität:

```
void deque<T>::push_back(const T&);
void deque<T>::pop_back();
T& deque<T>::back();

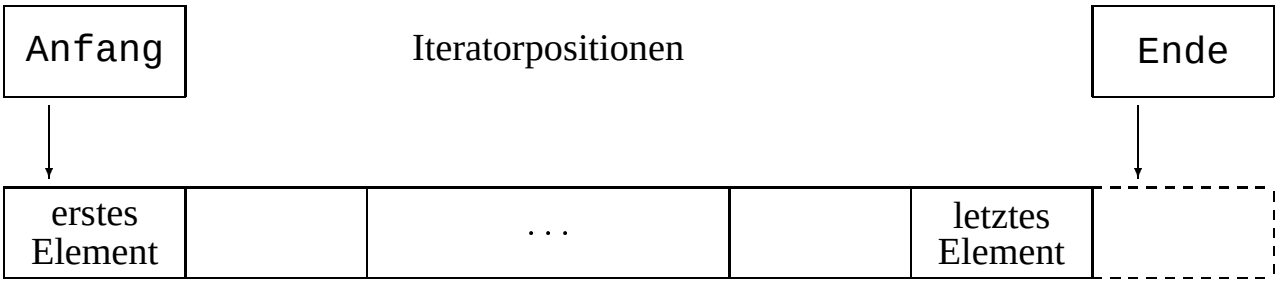
void deque<T>::push_front(const T&);
void deque<T>::pop_front();
T& deque<T>::front();
```

Schaubild:



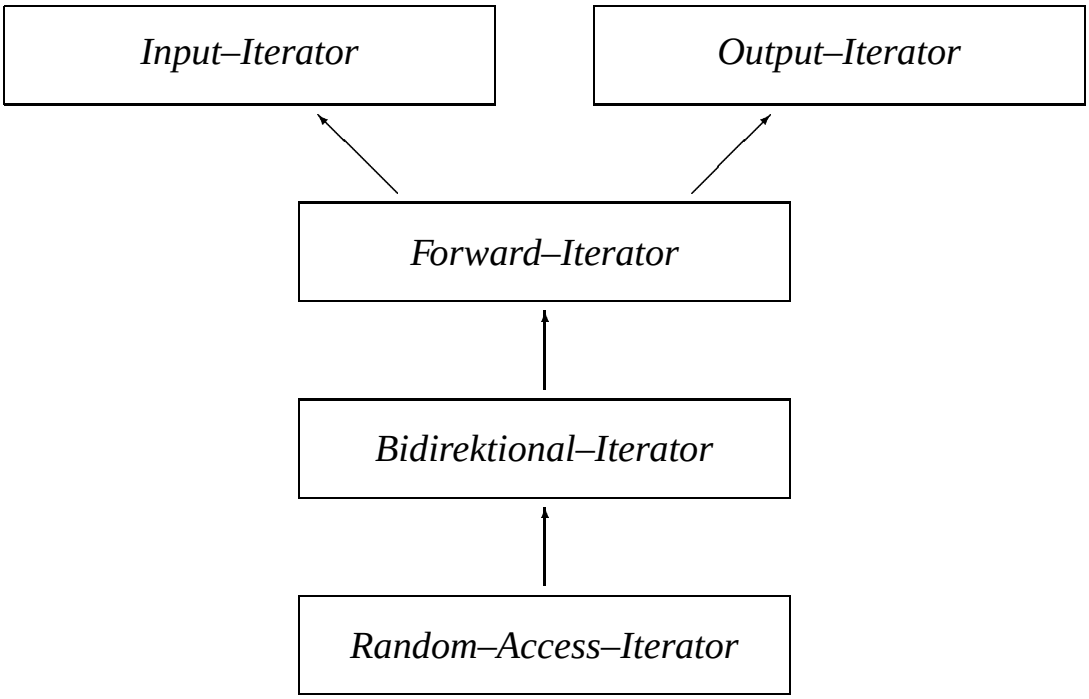
Iteratoren

- Verallgemeinerung des "Zeigerkonzeptes"
- Zugriff auf alle Elemente einer "Sequenz" (Elemente eines Containers), der Reihe nach

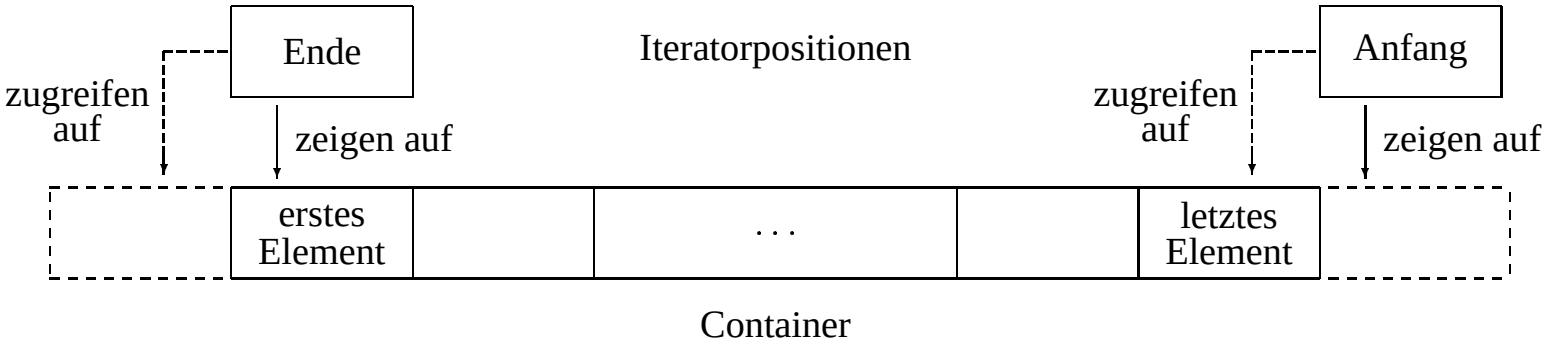


Iterator–Kategorien:

(unterschiedliche Funktionalität, keine Vererbung)



Rückwärts–Iterator (Iterator–Adapter)



Funktionalität von Iteratoren:

Operator	Bedeutung	Forward	Bidirekt.	Ran.–Acc.
`iter1 == iter2`	Test auf Gleichheit (gleiches "Reservoir" und gleiche Position)	×	×	×
`iter1 != iter2`	Test auf Ungleichheit	×	×	×
`*iter`	Lesezugriff auf das aktuelle Element	×	×	×
`iter->komp`	Zugriff auf eine Komponente des aktuellen Elementes	×	×	×
`++iter` `iter++`	Weitersetzen des Iterators	×	×	×
`iter1 = iter2`	Zuweisungsoperator	×	×	×
`--iter` `iter--`	Zurücksetzen des Iterators		×	×
`iter[n]`	Zugriff auf das n–te Element hinter bzw. vor der augenblicklichen Iteratorposition.			×
`iter += n`	Iterator um n Positionen weitersetzen (bzw. zurück, falls $n < 0$			×
`iter -= n`	Iterator um n Positionen zurücksetzen (bzw. vor, falls $n < 0$)			×
`iter + n` `n + iter`	Iterator für das n–te folgende (bzw. vorherige, falls $n < 0$) Element			×
`iter -n`	Iterator für das n–te vorhergehende (bzw. folgende, falls $n < 0$) Element			×
`iter1 - iter2`	Abstand der beiden Iteratoren liefern			×

`iter1 < iter2`
`iter1 > iter2`
`iter1 <= iter2`
`iter1 >= iter2`
	Vergleiche			×

Iteratoren der Standardcontainer:

Container	Forward	Bidirektoinal	Random–Access
`vector<T>`			×
`deque<T>`			×
`list<T>`		×	
`set<T>` `multiset<T>`		× ×	
`map<KEY,T>` `multimap<KEY,T>`		× ×	
`stack<T>`	haben keine Iteratoren		
`queue<T>`	haben keine Iteratoren		
`priority_queue<T>`	haben keine Iteratoren		
`bitset<n>`	haben keine Iteratoren		

- Iterator–Typ hängt vom Container und `<T>` ab:

  ContainerTyp`::iterator`
  ContainerTyp`::reverse_iterator`

- Elementfunktionen, welche entsprechende Iteratoren und Iteratorpositionen liefern:

  – *IteratorTyp ContainerTyp*`::begin();`
  Liefert einen (Vorwärts)–Iterator auf das erste Element des Containers.

  – *IteratorTyp ContainerTyp*`::end();`
  liefert einen (Vorwärts)–Iterator hinter das letzte Element des Containers.

  – *IteratorTyp ContainerTyp*`::rbegin();`
  liefert einen (Rückwärts)–Iterator auf das letzte Element des Containers.

  – *IteratorTyp ContainerTyp*`::rend();`
  liefert einen (Rückwärts)–Iterator vor das erste Element des Containers.

Beispiele für Iterator–Anwendungen:

1. Liste bzw. Vektor vorwärts durchlaufen:

```
#include <list>
#include <vector>

class A { ... };   // selbstdef. Typ

list<int> intlist;  // Liste von int’s
vector<A> Avector;  // Vektor von A’s


...
// Iterator fuer list<int>:
list<int>::iterator ilist_it;

// Iterator fuer vector<A>:
vector<A>::iterator Avec_it;
...
// durchlaufe Liste:
for ( ilist_it = intlist.begin();
      ilist_it != intlist.end();
      ++ilist_it)
{ // mach was mit aktuellem Element *ilist_it
}
...
// durchlaufe Vektor:
for ( Avec_it = Avector.begin();
      Avec_it != Avector.end();
      ++Avec_it)
{ // mach was mit aktuellem Element *Avec_it
}
...
```

2. Liste gleichzeitig vorwärts und rückwärts durchlaufen:

```
#include <list>

class A { ... };   // selbstdefinierter Datentyp
...
list<A> Alist;     // Liste von A-Objekten

// Vorwaerts-Iterator
list<A>::iterator av_it;

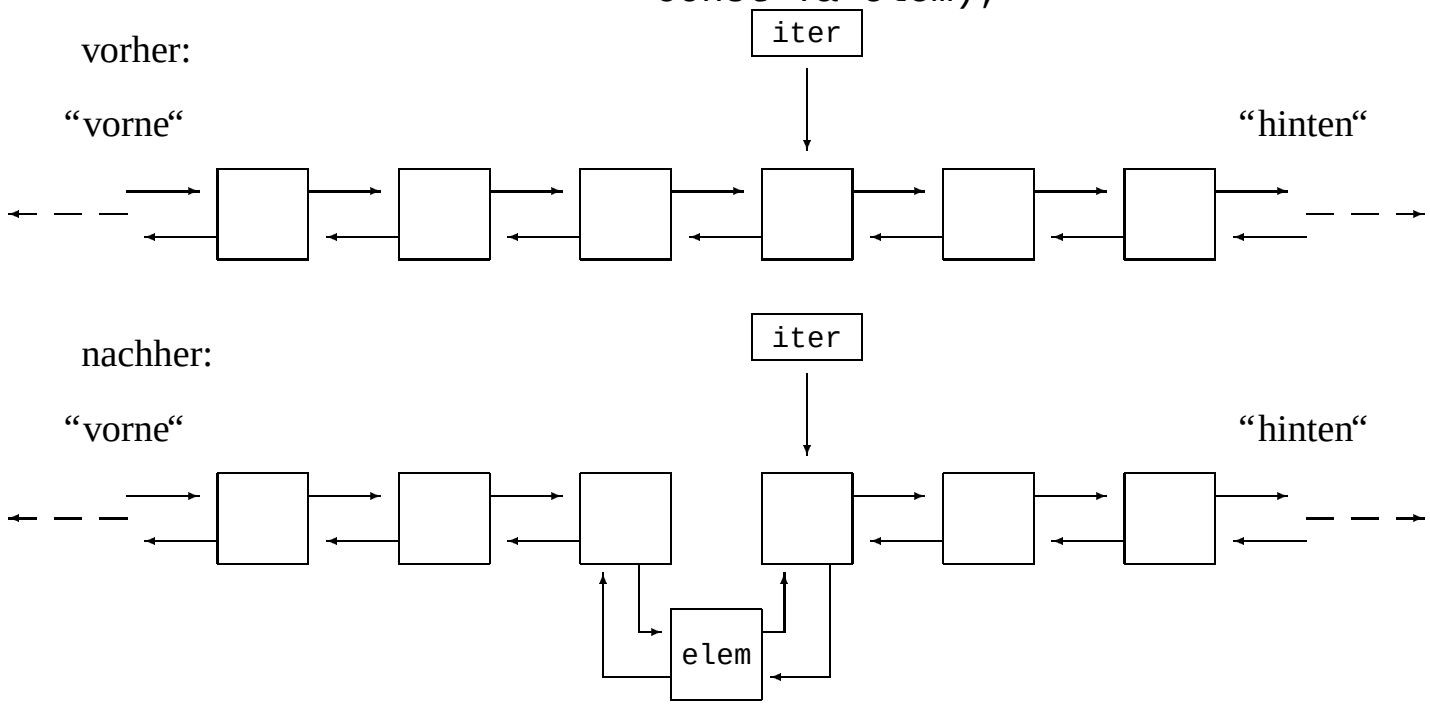
// Rueckwaerts-Iterator
list<A>::reverse_iterator ar_it;
...
/* av_it laeuft von vorne nach hinten und
   ar_it laeuft von hinten nach vorne
*/
for ( av_it=Alist.begin(), ar_it=Alist.rbegin();
      ar_it != Alist.rend();
      ++av_it, ++ar_it)
{
  // Zugriff mittels *ar_it und *av_it
  ...
}
...
```

Die Containerklasse `list<T>`:

Die wesentlichen Listenoperationen:

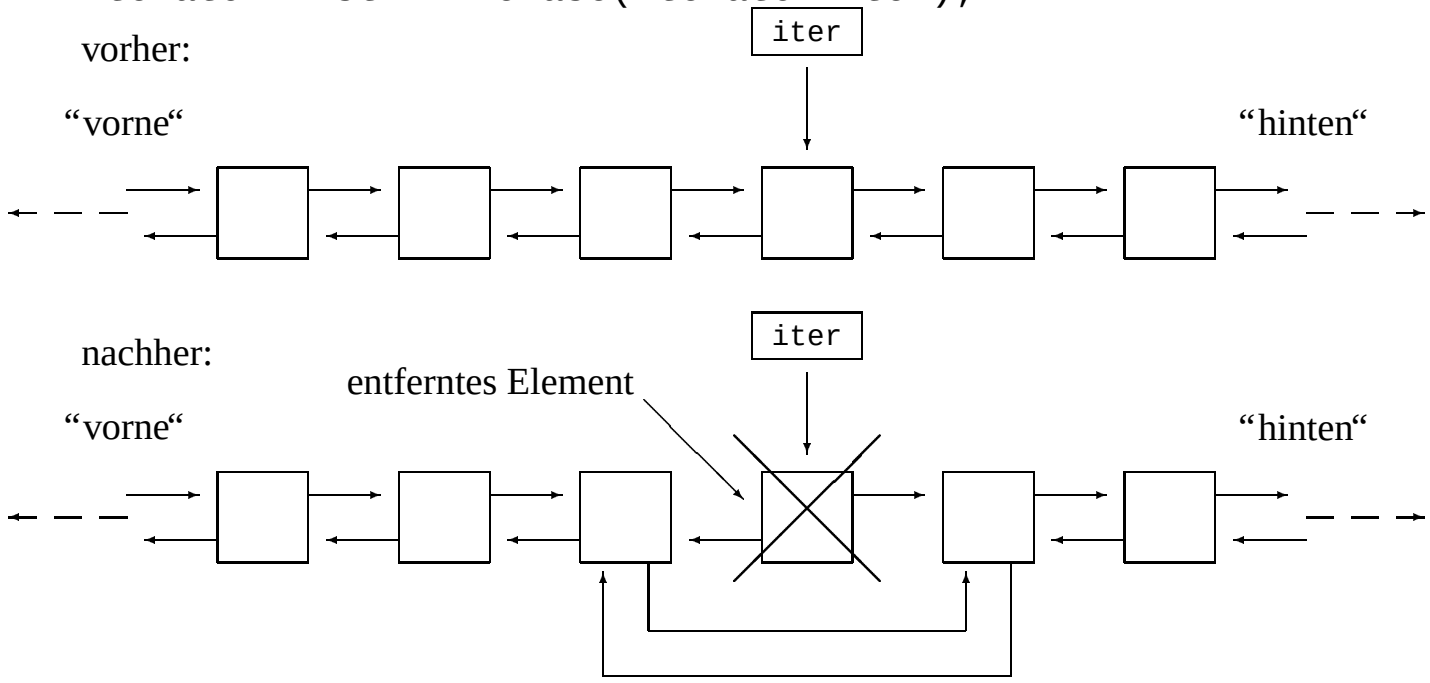
– Einfügen an beliebiger Position:

```
iterator list<T>::insert(iterator iter,
                         const T& elem);
```



– Löschen an beliebiger Position:

```
iterator list<T>::erase(iterator iter);
```



Basisklassen für Funktionsobjekte:

```
template <class T1, class T2, class T3>
struct binary_function {
  typedef T1 first_argument_type;
  typedef T2 second_argument_type;
  typedef T3 result_type;
};

template <class T1, class T2>
struct unary_function {
  typedef T1 first_argument_type;
  typedef T2 result_type;
};
```

Standardoperatoren als Funktionsobjekte:

Name		Wirkung	Name		Wirkung
`plus`	binär	`a + b`	`minus`	binär	`a - b`
`multiplies`	binär	`a * b`	`divides`	binär	`a / b`
`modulus`	binär	`a % b`	`negate`	unär	`-a`

Realisiert als Template, etwa:

```
template <class T>
struct plus : public binary_function<T,T,T> {
  T operator()(const T& x, const T& y) const
  { return x + y; }
};
```

Standardoperatoren als Prädikate:

Name		Wirkung	Name		Wirkung
`equal_to`	b	`a == b`	`not_equal_to`	b	`a != b`
`greater`	b	`a > b`	`less`	b	`a < b`
`greater_equal`	b	`a >= b`	`less_equal`	b	`a <= b`
`logical_and`	b	`a && b`	`logical_or`	b	`a || b`
`logical_not`	u	`!a`			

Funktionsadapter

macht Funktionsobjekte aus gewöhnlichen Funktionen (oder Funktionszeigern) (also Instantiierung der Templates `binary_function` bzw. `unary_function`):

```
int f(double , char);

...ptr_fun(f)...;
// liefert binary_function<double, char, int>
...
```

Realisierung als Template in Standardbibliothek:

```
template <class A, class B, class R>
class pointer_to_binary_function :
public binary_function<A,B,R> {
  protected:
    R (*F_ptr)(A, B);
  public:
    pointer_to_binary_function() {}
    explicit pointer_to_binary_function(R (*x)(A, B))
        : F_ptr(x) {}
      R operator()(A a, B b) const {
      return F_ptr(a, b);
    }
};

template <class A, class B, class R>
inline pointer_to_binary_function<A,B,R>
ptr_fun(R (*x)(A, B)) {
  return pointer_to_binary_function<A,B,R>(x);
}
```

Analog für unäre Funktionen, ähnlich für Member–Funktionen.

nicht modifizierende Algorithmen für Sequenzen:

1. ```
   template <class InIt, class UnOp>
   UnOp for_each(InIt anf, InIt ende, UnOp f);
   ```

   - wendet auf jedes Element der Sequenz das unäre Funktionsobjekt `f` an,
   - Ergebnis der Anwendung von `f` wird jeweils ignoriert,
   - gibt das Funktionsobjekt selbst als Ergebnis zurück.

   Beispiel:

   ```
   class FEHLER {};     // Fehlerklasse:
   template <class T>
   class MAX_OP {
     private: T max;
        bool leer; // bislang kein Element gesehen
     public:  MAX_OP() : leer(true) {}
       void operator()( T& wert)
       { if ( leer ) { max = wert; leer = false; }
         else if ( max < wert) max = wert;
       }
       operator T() // Maximalwert zurueckgeben
       { if ( leer ) throw FEHLER();
         return max;
       }
   };

   void f(vector<int> v, deque<double> d )
   { cout <<
       for_each(v.begin(),v.end(),MAX_OP<int>())
       << endl;
     cout <<
       for_each(d.begin(),d.end(),MAX_OP<double>())
       << endl;
   }
   ```
   ```

2. ```
   template <class InIt, class T>
   InIt find( InIt anf, InIt ende, const T &wert);
   ```

   - suche in der Sequenz nach dem ersten Element, welches mit dem angeg. Wert übereinstimmt,
   - gibt Iteratorposition auf Treffer (oder `ende`) zurück.

   ```
   template <class InIt, class T>
   InIt find_if( InIt anf, InIt ende, UnPred pred);
   ```

   - suche in der Sequenz nach dem ersten Element, welches das angeb. Prädikat erfüllt,
   - gibt Iteratorposition auf Treffer (oder `ende`) zurück.

3. ```
   template <class InIt, class T>
   int_type count(InIt anf,InIt ende,const T& wert);
   ```

   gibt Anzahl der Elemente der Sequenz zurück, die mit dem angegeb. Wert übereinstimmen.

   ```
   template <class InIt, class UnPred>
   int_type count_if(InIt anf,InIt ende,UnPred pred);
   ```

   gibt Anzahl der Elemente der Sequenz zurück, auf die das angegeb. Prädikat zutrifft.

4. ```
   template <class ForIt, class ForIt2>
   ForIt search( ForIt anf, ForIt ende,
                 ForIt2 anf2, ForIt2 ende2);
   ```

   sucht in der Sequenz `[anf,ende)` nach einer Teilsequenz, welche elementweise mit der zweiten Sequenz `[anf2,ende2)` übereinstimmt.

   Gibt Iteratorposition auf Treffer (Anfang) oder `ende` zurück.

5. . . . (zahlreiche weitere Algorithmen)
   ```

modifizierende Algorithmen für Sequenzen:

etwa: aufsteigendes Sortieren einer Sequenz von Elementes des Types `T`:

1. bzgl. des Vergleiches mit `<`.

   Dieser Vergleich muss die Eigenschaften haben:

   - *strikt*, d.h. `a < a` ist *falsch* für jedes `T`
   - *transitiv*, d.h. `a < b` und `b < c` , so folgt: `a < c`
   - *Gleichheit*: sind `a < b` und `b < a` beide falsch, so werden `a` und `b` (bzgl. `<`) als *gleich* angesehen.

     Diese Gleichheit muss wiederum transitiv sein.

   ```
   template <class RanIt>
   void sort( RanIt anf, RanIt ende);
   ```

2. bzgl. eines durch ein binäres Prädikat gegebenes Vergleichskriterium (gleiche Eigenschaften wie oben `<`):

   ```
   template <class RanIt, class BinPred>
   void sort( RanIt anf, RanIt ende, BinPred comp);
   ```

3. ```
   template <class RanIt>
   void stable_sort( RanIt anf, RanIt ende);

   template <class RanIt, class BinPred>
   void stable_sort( RanIt anf, RanIt ende,
         BinPred comp);
   ```

   stabile Sortierung: “gleiche“ behalten relative Reihenfolge.

4. . . . (zahlreiche weitere Algorithmen)
   ```

Funktionen für Gleitkommatypen

`T` sei einer der Typen `float`, `double` oder `long double`

`T abs(T d);`	Absolutbetrag
`T fabs(T d);`	Absolutbetrag
`T ceil(T d);`	kleinster Integer nicht kleiner als `d`
`T floor(T d);`	größter Integer nicht größer als `d`
`T sqrt(T d);`	Quadratwurzel aus `d` (darf nicht negativ sein)
`T pow(T d, T e);`	`d` hoch `e`, Fehler, falls `d` gleich 0 und `e` negativ bzw. falls `d` kleiner 0 und `e` nicht ganzzahlig
`T pow(T d, int i);`	`d` hoch `i`, Fehler, falls `d` gleich 0 und `i` negativ
`T sin(T d);`	Sinus
`T cos(T d);`	Cosinus
`T tan(T d);`	Tangens
`T asin(T d);`	Arcus–Sinus
`T acos(T d);`	Arcus–Cosinus
`T atan(T d);`	Arcus–Tangens
`T atan2(T x, T y);`	entspricht `atan(x/y)`
`T sinh(T d);`	Sinus hyperbolicus
`T cosh(T d);`	Cosinus hyperbolicus
`T tanh(T d);`	Tangens hyperbolicus
`T exp(T d);`	Exponentialfunktion
`T log(T d);`	natürlicher Logartihmus (`d` muss größer 0 sein!)
`T log10(T d);`	Logartihmus zur Basis 10 (`d` muss größer 0 sein!)
`T modf(T d, T* p);`	Nachkommateil von `d` als Ergebnis, ganzzahliger Teil nach `*p`
`T frexp(T d,int* p);`	Zahl `x` im Intervall $[0.5, 1)$ und ganzzahliges `y` finden mit `d = x * pow(2,y)`, `x` als Funktionsergebnis und `y` nach `*p`
`T fmod(T d, T* m);`	Rest der Gleitkommadivision von `d/m`, gleiches Vorzeichen wie `d`. (`m` darf nicht 0 sein!)
`T ldexp(T d,int i);`	liefert `d * pow(2,i)`

Komplexe Zahlen:

Sei `T` sei einer der Typen `float`, `double` oder `long double`, dann ist `complex<T>` der zugehörige Typ komplexer Zahlen.

```
template <class T>
class complex {
  ...
  public:
    // Konstruktoren
    complex(const T& re = 0.0, const T& im = 0.0);
    template <class U>
    complex(const complex<U> & c);
    ...
    // Zuweisung
    complex<T>& operator=( const T& w);

    template <class U>
    complex<T>& operator=( complex<U> & w);
    ...
    // Realteil und Imaginaerteil liefern,
    // Memberfunktionen:
    T real() const;
    T imag() const;

    // Realteil und Imaginaerteil liefern,
    // (befreundete) globale Funktionen:
    friend T real( const complex<T> &c);
    friend T imag( const complex<T> &c);

    T norm() const;  // Norm liefern
    T abs()  const;  // Absolutbetrag liefern
    T arg()  const;  // Winkel der Polardarstellung

    // befreundete Vergleichs-Operator
friend bool operator==(complex<T>& a,complex<T>&b);
friend bool operator!=(complex<T>& a,complex<T>&b);
    ...
};
```

```
// aus Polarkoordinaten temp. kompl. Zahl erzeugen:
template <class T>
complex<T> polar( const T& abs, const T& phi);

// konjugiert komplexe Zahl:
template <class T>
complex<T> conj( const complex<T>& w);

// Ein-/Ausgabe:
ostream& operator<<(ostream& s,const complex<T>& c);
istream& operator>>(istream& s,complex<T>& c);
```

Transzendente Funktionen:

Exponentialfunktion	`exp(c)`
natürlicher Logarithmus (Hauptzweig)	`log(c)`
10–er Logarithmus	`log10(c)`
Potenzfunktion entspricht	`pow( c1, c2)` `exp(c2*log(c1))`
Quadratwurzel (welche?)	`sqrt(c)`
Sinusfunktion	`sin(c)`
Cosinusfunktion	`cos(c)`
Tangensfunktion	`tan(c)`
Sinus hyperbolicus	`sinh(c)`
Cosinus hyperbolicus	`cosh(c)`
Tangens hyperbolicus	`tanh(c)`

Mathematische Vektoren:

(T sei einer arithmetischer Typ)
`valarray<T>`: Vektor von diesem Typ mit zusätzlichen (in der Mathematik üblichen) Operationen.

Erzeugung:

```
double x = ... ;
double f[200] = ... ;
valarray<double> a;         // Laenge 0
valarray<float>  b(50);     // Laenge 50
valarray<double> c(x, 50);  // Laenge 50, jedes
                  // Element mit x initialisiert
valarray<double> d(f, 100); // Laenge 100, mit Feld
                            // initialisiert
```

Größe abfragen, ändern:

```
n = a.size();        // Groesse abfragen
a.resize(500);       // vergroessern
a.resize(500, x);    // vergroessern mit Belegung
```

Zuweisung und Indizierung:

```
valarray<double> a(100), b(100), c(200);
double x;
...
a = b; // Zuweisung von valarrays gleicher Laenge ok
a = c; // FEHLER: unterschiedliche Laenge
...
a = x; // alle 100 Elem. von a bekommen Wert von x
...
x = a[0];   // Indizierung ok
x = a[99];  // Indizierung ok
x = a[200]  // LAUFZEITFEHLER, Compiler merkt nichts
x = a[-23]  // LAUFZEITFEHLER, Compiler merkt nichts
```

unäre Operatoren:
Ist einer der unären arithmetischen Operatoren `+` (Vorzeichen), `-` (Vorzeichen), `~` (Komplement) oder `!` (Negation) für den Typ `T` definiert, so ist er auch für den Type `valarray<T>` definiert und wird elementweise angewendet:

```
valarray<double> a(100), b(100);
valarray<int> c(10), d(10);
valarray<bool> e(10);
...
a = -b;   // Vorzeichen
a = ~b;   // FEHLER: ~ fuer double nicht definiert

c = ~d;   // Komplement
e = !d;   // Negation
```

binäre Operatoren
Ist einer der binären Operatoren:

Operator	Operation	Operator	Operation
+	Addition	-	Subtraktion
*	Multiplikation	/	Division
%	Modulo	^	Bit–Exklusives Oder
&	Bit–Und	|	Bit–Inklusives Oder
<<	Links–Shift	>>	Rechts–Shift
&&	Logisches Und	||	logisches Oder
==	Test auf Gleichheit	!=	Test auf Ungleichheit
<	Test auf kleiner	>	Test auf größer
<=	Test auf kleiner gleich	>=	Test auf größer gleich

für den Datentyp `T` definiert, so ist er auch für `valarray<T>` definiert, es wird die Operation komponentenweise durchgeführt, etwa:

```
valarray<double> a(100), b(100), c(100);
valarray<bool> d(100);
double x;
c = a + b;   // gleiche Laenge!
c = a + x;   // x wird auf jedes Elem. von a addiert
c = x + a;   // x wird auf jedes Elem. von a addiert
d = a < b;   // gleiche Laenge
d = a < x;   // jedes Element von a wird mit x vergl.
d = x < a;   // x wird mit jedem Element von a vergl.
```

weitere Funktionen für `valarray<T>`:

```
valarray<double> a(100);
double x;
x = a.sum();     // Summe aller Elemente
x = a.min();     // kleinstes Element
x = a.max();     // groesstes Element
```

Mathematische Funktionen werden elementweise angewendet:

Funktion	Bedeutung	Funktion	Bedeutung
`abs(a)`	Absolutbetrag	`exp(a)`	Exponentialfunktion
`sqrt(a)`	Quadratwurzel	`log(a)`	nat. Logarithmus
`log10(a)`	Log. zur Basis 10	`sin(a)`	Sinus–Funktion
`cos(a)`	Cosinus–Funktion	`tan(a)`	Tangens–Funktion
`sinh(a)`	Sinus hyperbolicus	`cosh(a)`	Cosinus hyperbolicus
`tanh(a)`	Tangens hyperbolicus	`asin(a)`	Arcus Sinus
`acos(a)`	Arcus Cosinus	`atan(a)`	Arcus Tangens
`pow(a,b)`	k–te Komp. des Erg. ist `pow( a[k], b[k])`		
`pow(a, x)`	k–te Komp. des Erg. ist `pow( a[k], x)`		
`pow(x, a)`	k–te Komp. des Erg. ist `pow( x, a[k])`		
`atan2(a,b)`	k–te Komp. des Erg. ist `atan2( a[k], b[k])`		
`atan2(a,x)`	k–te Komp. des Erg. ist `atan2( a[k], x)`		
`atan2(x,a)`	k–te Komp. des Erg. ist `atan2( x, a[k])`		