FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

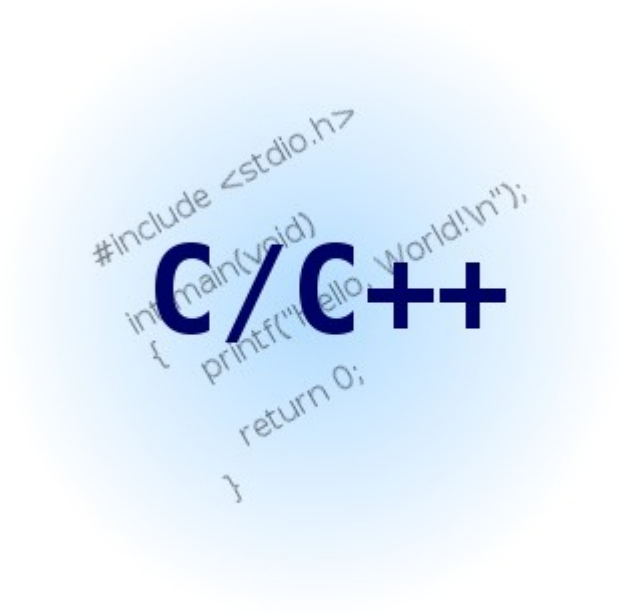
Programmieren mit C++

(Modulcode 90820)

Prof. Dr. Andreas Terstegge

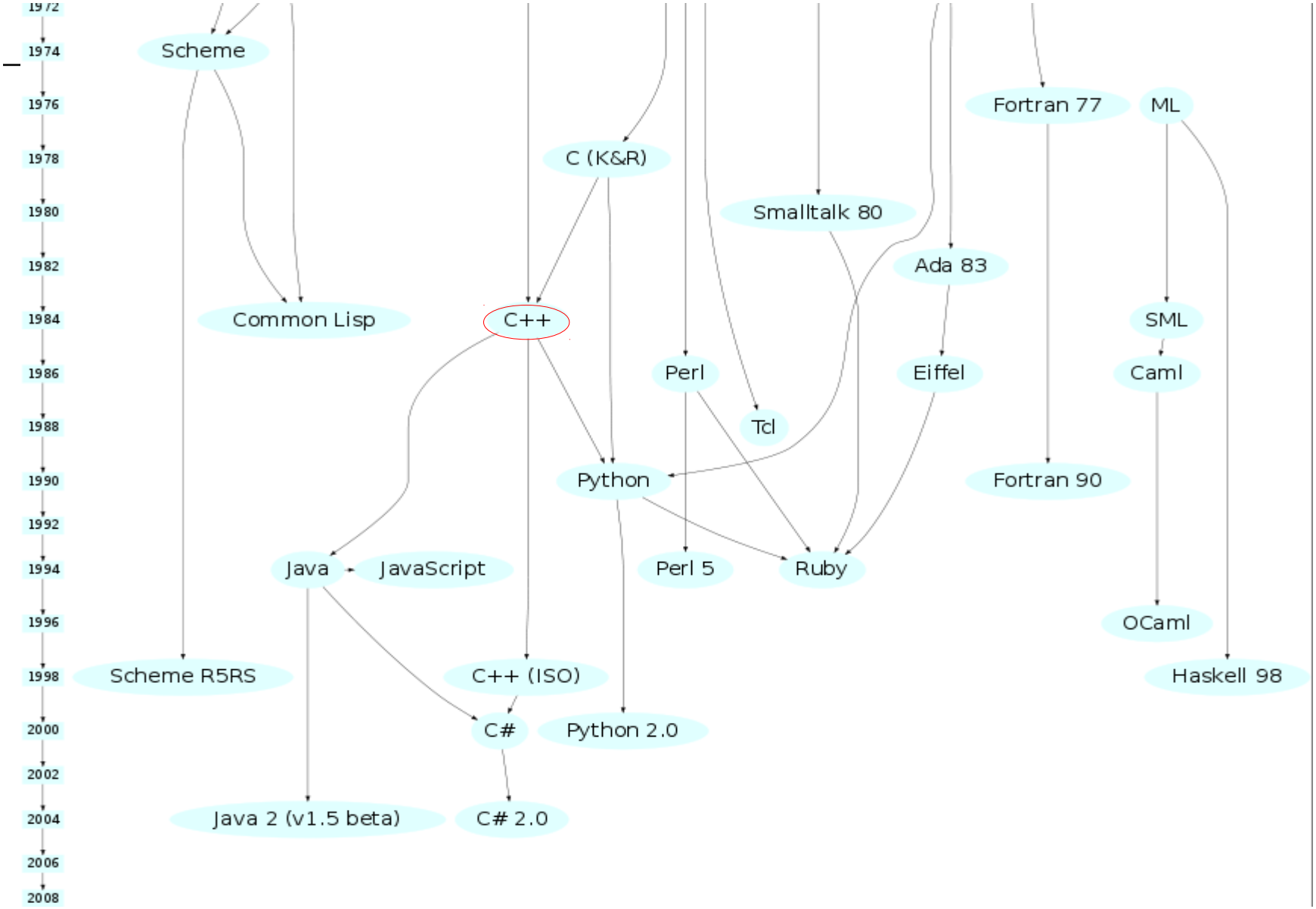
WS 2014/15

Geschichte der Sprache C++



Erfinder der Sprache:
Bjarne Stroustrup (* 30.12.50)

Geschichte der Sprache C++



Geschichte der Sprache C++

Wichtige Meilensteine

1980	Bjarne Stroustrup beginnt Arbeit an „C with Classes“
1983	Benennung der Sprache C++
1985	Stroustrup: *„The C++ Programming Language“*
1987	Erster gnu C++ Compiler (1.13)
1990	The Annotated C++ Reference Manual (ARM)
1998	**C++98 (ISO/IEC 14882:1998)**
2003	C++03 (ISO/IEC 14882:2003) (nur bug-fixes)
2007	C++TR1 (ISO/IEC TR 19768:2007) kein echter Standard, Erweiterung der Standardbibliothek durch Boost-Community
2009	Stroustrup: *„The C++ Programming Language“*, 4. Auflage
2011	**C++11 (vorher C++0x) (ISO/IEC 14882:2011)**
2014	**C++14 (ISO/IEC 14882:2014(E))** (bug fixes, kleinere Erweiterungen)

Verbreitung der Sprache C++ (Stand Ende September 2014, http://www.tiobe.com)

Sep 2014	Sep 2013	Change	Programming Language	Ratings	Change
1	1		C	16.721%	-0.25%
2	2		Java	14.140%	-2.01%
3	4	▲	Objective-C	9.935%	+1.37%
4	3	▼	C++	4.674%	-3.99%
5	6	▲	C#	4.352%	-1.21%
6	7	▲	Basic	3.547%	-1.29%
7	5	▼	PHP	3.121%	-3.31%
8	8		Python	2.782%	-0.39%
9	9		JavaScript	2.448%	+0.43%
10	10		Transact-SQL	1.675%	-0.32%
11	11		Visual Basic .NET	1.532%	-0.31%

C++ - Schlüsselwörter

C++ hat 83 reservierte Schlüsselwörter (C hat 32, hier eingefärbt)

alignas	char32_t	enum	namespace	short	typedef
alignof	class	explicit	new	signed	typeid
and	compl	export	not	sizeof	typename
and_eq	const	extern	not_eq	static	union
asm	const_cast	false	nullptr	static_assert	unsigned
auto	constexpr	float	operator	static_cast	using
bitand	continue	for	or	struct	virtual
bitor	decltype	friend	or_eq	switch	void
bool	default	goto	private	template	volatile
break	delete	if	protected	this	wchar_t
case	do	inline	public	thread_local	while
catch	double	int	register	throw	xor
char	dynamic_cast	long	reinterpret_cast	true	xor_eq
char16_t	else	mutable	return	try	

Neue C++ - Schlüsselwörter Kategorien

Klassen etc.: `class, private, protected, public, this, friend, mutable, virtual, inline, operator, explicit`

Templates: `template, export, typename`

Speichermanagement: `new, delete`

Ausnahmen (engl. Exceptions): `try, catch, throw`

Namensräume: `namespace, using`

Typumwandlung: `const_cast, dynamic_cast, reinterpret_cast, static_cast`

Datentypen und Größen: `auto` (neue Bedeutung), `bool, true, false, char16_t, char32_t, wchar_t, alignas, alignof`

Sonstiges: `asm, constexpr, decltype, nullptr, static_assert, thread_local, typeid`

Alternative Darstellung für Operatoren (unwichtig...):
`and, and_eq, bitand, bitor, compl ,not ,not_eq, or, or_eq, xor, xor_eq`

Literatur zu C++

- Breymann, Ulrich, *Der C++ Programmierer*, Hanser, 1. Auflage 2009
  - ➠ sehr gutes und umfassendes Lehrbuch
- Stroustrup, *The C++ Programming Language*, 3. Auflage

  (14. Druck), Addison-Wesley, 2000, ISBN 0-201-88954-4 or ISBN 0-201-70073-5.

  Stroustrup, *Die C++ Programmiersprache*, **Dritte oder Vierte Auflage**,

  Addison-Wesley, 1997 or 2000, ISBN 3-8273-1296-5 or 3-8273-1660-X.
  - ➠ Behandelt die komplette Sprache, allerdings etwas akademisch...
- Lippman and Lajoie, *C++ Primer*, **Third Edition**, Addison-Wesley, 1998, ISBN 0-201-82470-1.
  - ➠ Tutoriums-Stil. Besser für Anfänger
- Koenig and Moo, *Accelerated C++*, Addison-Wesley, 2000, ISBN 0-201-70353-X.
  - ➠ Gutes Buch zur Einführung mit Schwerpunkt auf konkreter Problemlösung in C++
- Josuttis, *Objektorientiertes Programmieren in C++*, **2. Auflage,**

  Addison-Wesley, 2001, ISBN 3-8273-1771-1.
- ...

Informationen im WEB

- FZJ C++ WWW Information Index

  http://www2.fz-juelich.de/jsc//cv/lang/cplusplus/cplusplus_ext

- Offizielles C++ Online FAQ

  http://www.parashift.com/c++-faq-lite/

- The Association of C & C++ Users

  http://www.accu.org/

  - Buchbesprechungen von über 2400 Büchern

  http://accu.org/index.php?module=bookreviews&func=search

- cplusplus.org: Viele Informationen rund um C++

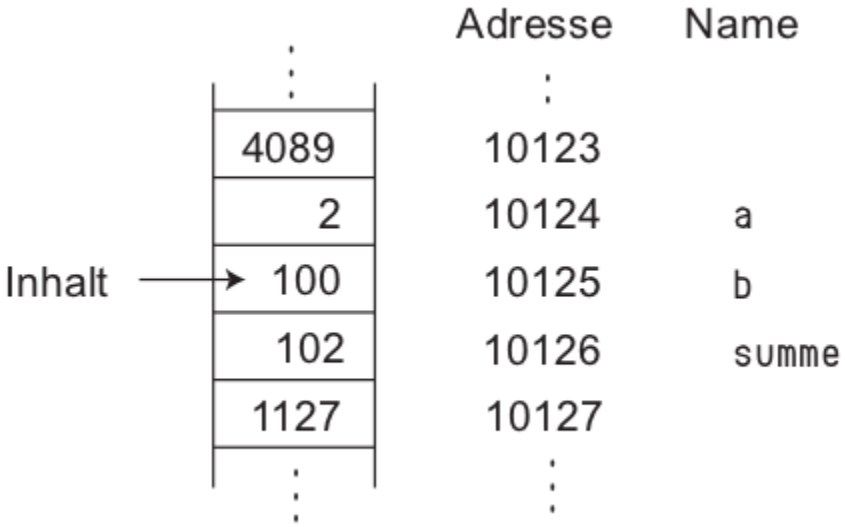
  http://www.cplusplus.com/

Das erste Programm... :)

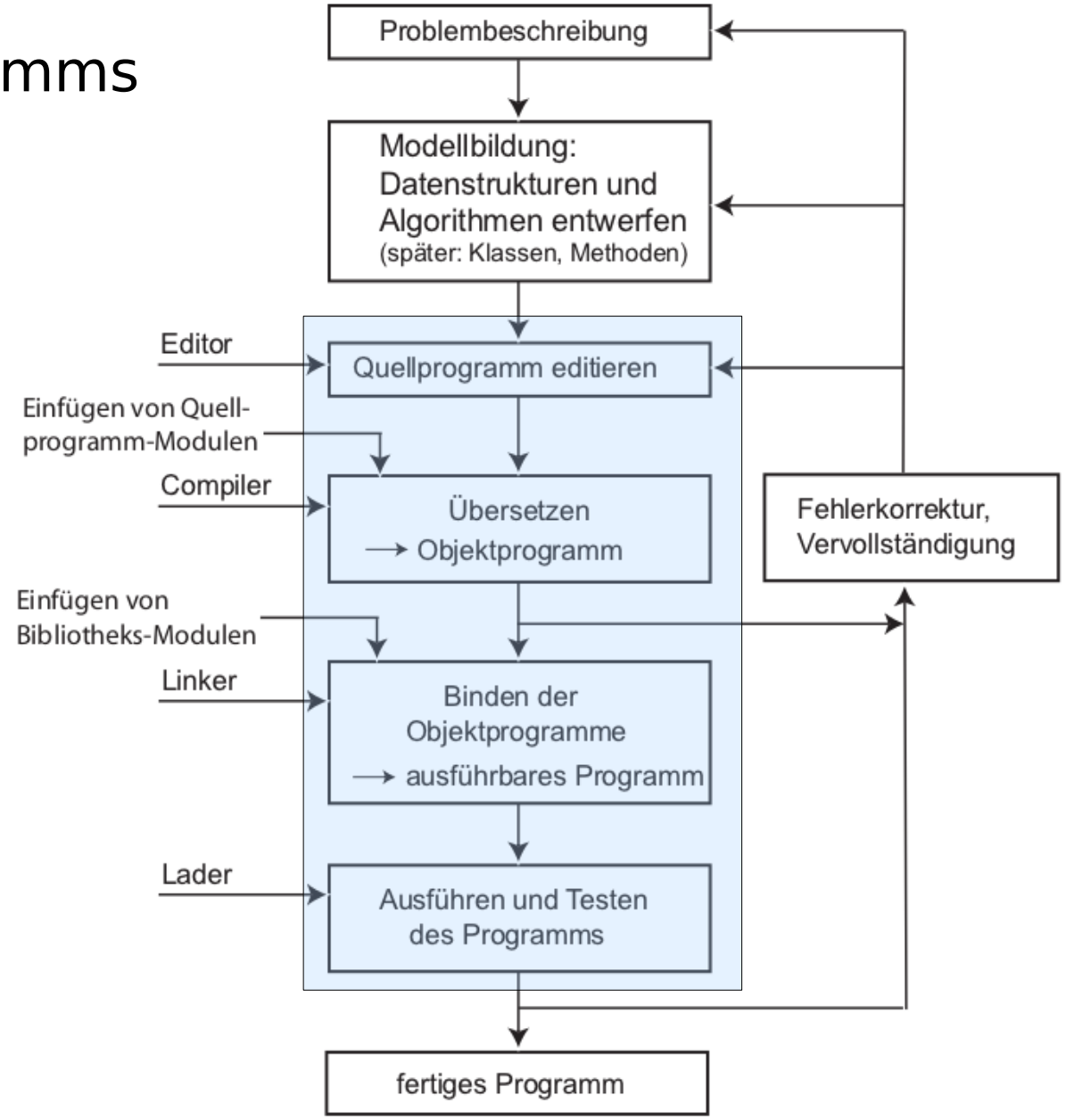
```
#include<iostream>
using namespace std;
int main() {
  // Programm zur Berechnung der
  // Summe zweier Zahlen
  int summe;
  int a;
  int b;
  // Lies die Zahlen a und b ein
  cout << "a und b eingeben : ";
  cin >> a >> b;

  /* Berechne die Summe beider Zahlen
       */
  summe = a + b;

  // Zeige das Ergebnis auf dem
  // Bildschirm an
  cout << "Summe=" << summe;
  return 0;
}
```

```
a und b eingeben : 2<ENTER>
100<ENTER>
Summe=102
```

Kommentare: // (einzeilig) oder
/* */ (mehrzeilig)



Erzeugung eines lauffähigen Programms



Namenskonventionen

Namen (u.a. von Variablen und Funktionen) unterliegen folgenden Konventionen:

- Ein Name besteht aus Buchstaben, Ziffern und dem Unterstrich _

- Ein Name beginnt immer mit einem Buchstaben oder _ , wobei der _ aber vermieden werden sollte (Namen mit __ am Anfang werden oft systemintern verwendet!

- Namen dürfen nicht mit Schlüsselwörtern identisch sein.

- Namen dürfen nur eine (systemabhängige) maximale Länge besitzen (z.B. 31 oder 255 Zeichen).

- Namen sind case-sensitiv, d.h. `foo` und `Foo` sind verschieden.

Beispiele:

```
int 1_Zeile;           // NOK Ziffer am Anfang
int Eine Variable;     // NOK Leereichen ..
inr AnzahlDerZeilen;   // OK camel casing
int Anzahl_der_Zeilen; // OK underscoring
```

Compiler
Beispiel: GNU g++

`g++ --help`	die wichtigsten Optionen anzeigen
`g++ --version`	Compiler-Version anzeigen
`g++ -c summe.cpp`	nur Compilieren (summe.o wird erzeugt)
`g++ -o summe summe.o`	Linken
`g++ summe.cpp`	Compilieren und Linken

Mehrere Dateien:

`g++ a1.cpp main.cpp`	Compilieren und Linken

oder einzeln

`g++ -c a1.cpp`	Compilieren
`g++ -c main.cpp`	Compilieren
`g++ a1.o main.o`	Linken, Ergebnis a.out
`g++ -o main.exe a1.o main.o`	Linken, Ergebnis main.exe

Benutzung von Bibliotheken:

`g++ -o main.e a1.o main.o -ljpeg`	Die Bibliothek libjpeg.a wird benutzt

Compiler (GNU g++)
wichtige Kommandozeilenparameter

`-c`	Nur Kompilieren	`-O`	Optimierung einschalten
`-o` *`file`*	Ausgabe der Binärdatei in Datei *file*	`-g`	Debugging einschalten
`-D`*`macro[=defn]`*	Definiere Präprozessor-Symbol *macro*	`-Wall`	Alle Warnings ausgeben
`-I`*`dir`*	Suche nach Header-Dateien in *dir*	`--help`	Hilfe für Compileraufruf
`-U`*`macro`*	Lösche Präprozessor-Symbol *macro*	`--version`	Zeige Compiler-Version
`-E`	Gebe Ausgabe des Präprozessors aus	`-x` *`language`*	Sprachstandard einstellen
`-L`*`dir`*	Suche nach Bibliotheken in *dir*	`-v`	'verbose': Zeige interne Verarbeitungsschritte
`-l`*`name`*	Binde `lib`*`name`*`.a` dazu		

Compiler und Präprozessor

Präprozessor-Anweisungen - Beginnen immer mit '#'

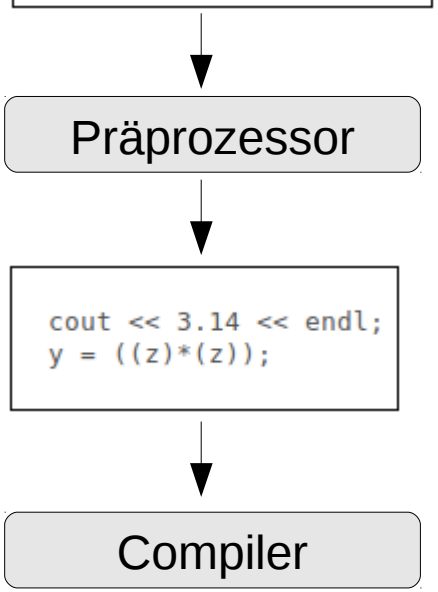
`#include "mine.h"`	Füge Datei ein. Suche erst im aktuellen Verzeichnis.
`#include <stdio.h>`	Füge Datei ein. Suche erst im System-Verzeichnis.
`#define TRUE 1`	Makro-Substitution. Substituiere String TRUE mit 1
`#define min(a,b) (a<b)?(a):(b)`	Makro-Substitution mit Parametern.
`#define abs(a) (a<0)?(-(a)):(a)`	Makro-Substitution mit Parameter.
`#define note /* comment */`	Alle Vorkommen von *note* werden durch einen Kommentar ersetzt.
`#undef TRUE`	Lösche ('undefine') eine Makro-Namen.
`#error message`	Breche Kompilierung an dieser Stelle mit 'message' ab.
`#if expression`	Bedingte Kompilierung (bzw. Quelltextweitergabe)
`#elif expression`	„else if“ Ausdruck. Zur Kaskadierung von if/else
`#else`	„else“ Ausdruck
`#endif`	Ende des Bedingungs-Bereichs
`#ifdef macroname`	Wie #if, Ausdruck ist wahr, wenn Makroname definiert ist
`#ifndef`	Wie #if, Ausdruck ist wahr, wenn Makroname NICHT definiert ist
`#line number [filename]`	Setze Startpunkt für __LINE__ and __FILE__
`#pragma`	Sonderbefehle für Compiler

Compiler und Präprozessor

Beispiele für 2 Präprozessor-Makros:

```
// nicht empfehlenswert! (Begründung folgt)
#define PI 3.14
#define schreibe cout
#define QUAD(x) ((x)*(x))
```

```
schreibe << PI << endl;
y = QUAD(z);
```

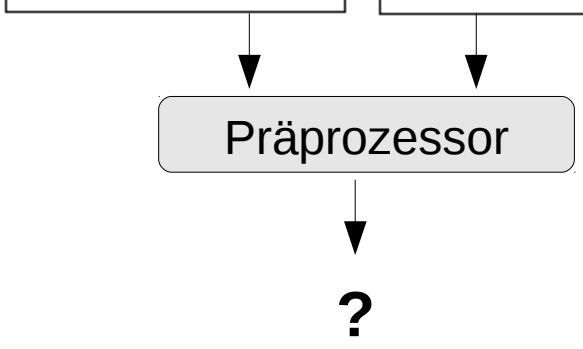



```
cout << 3.14 << endl;
y = ((z)*(z));
```

Wo liegen bei diesen beiden Beispielen Probleme beim Einsatz Von Makros?

```
// Makrodefinition
#define MULT(a,b) ((a)*(b))

// Makroaufruf
int a, b = 2, c = 3;
a = MULT(b,c); / / ok
a = MULT(b,"Fehler!");
```

```
int z = 3;
int y = QUAD(++z);
```



Einfache Datentypen und Operatoren
Ganze Zahlen

Bit-Größe der integer-Typen
ist im Standard nicht definiert:
nur Bits(short) <= Bits(int) <= Bits(long) <= Bits(long long)

`short` (oder `short int`)	16 Bits
`int`	32 Bits (oder 16 Bits)
`long` (oder `long int`)	64 Bits (oder 32 Bits)
`long long` (oder `long long int`)	mindestens soviel wie long

Darstellung von negativen
Zahlen mit 2-er Komplement
(hier: `signed short`)

binär	dezimal
0111 1111 1111 1111	32767
0000 0000 0000 0000	0
1111 1111 1111 1111	-1
1111 1111 1111 1110	-2
1000 0000 0000 0000	-32768

Wertebereiche:

`signed`	16 Bits	$-2^{15} \ldots 2^{15}-1$	=	−32 768 ... 32 767
`signed`	32 Bits	$-2^{31} \ldots 2^{31}-1$	=	−2 147 483 648 ... 2 147 483 647
`unsigned`	16 Bits	$0 \ldots 2^{16}-1$	=	0 ... 65 535
`unsigned`	32 Bits	$0 \ldots 2^{32}-1$	=	0 ... 4 294 967 295

Einfache Datentypen und Operatoren
Ganze Zahlen – Wertebereich auf 32-bit System

```
#include <iostream>
#include<climits>  // hier sind die Bereichsinformationen
using namespace std;

int main() {
   cout << "Grenzwerte für Ganzzahl-Typen:"
        << endl;  // = neue Zeile (endline)
   cout << "INT_MIN =       " << INT_MIN << endl;
   cout << "INT_MAX =       " << INT_MAX << endl;
   cout << "LONG_MIN =      " << LONG_MIN << endl;
   cout << "LONG_MAX =      " << LONG_MAX << endl;
   // C++11
   cout << "LLONG_MIN =     " << LLONG_MIN << endl;
   cout << "LLONG_MAX =     " << LLONG_MAX << endl;

   cout << "unsigned-Grenzwerte:" << endl;
   cout << "UINT_MAX =      " << UINT_MAX << endl;
   cout << "ULONG_MAX =     " << ULONG_MAX << endl;
   // C++11
   cout << "ULLONG_MAX =    " << ULLONG_MAX << endl;
   cout << "Anzahl der Bytes für:" << endl;
   cout << "int       " << sizeof(int) << endl;
   cout << "long      " << sizeof(long) << endl;
   // C++11
   cout << "long long " << sizeof(long long) << endl;
}
```

(64-bit System ...)

```
Grenzwerte für Ganzzahl-Typen:
INT_MIN =       -2147483648
INT_MAX =       2147483647
LONG_MIN =      -9223372036854775808
LONG_MAX =      9223372036854775807
LLONG_MIN =     -9223372036854775808
LLONG_MAX =     9223372036854775807
unsigned-Grenzwerte:
UINT_MAX =      4294967295
ULONG_MAX =     18446744073709551615
ULLONG_MAX =   18446744073709551615
Anzahl der Bytes für:
int       4
long      8
long long 8
```

Einfache Datentypen und Operatoren
Ganze Zahlen

- Der Compiler erkennt ggf. keine Über- oder Unterläufe, daher immer einen passenden Datentyp wählen!
- Integer-Datentypen sind standardmäßig `signed` (`signed int` und `int` sind identisch!)
- Literale:

Prefixe: `0` : Oktalzahl, jede Ziffer entspricht 3 Bit)
`0x` oder `0X` : Hexadezimalzahl, jede Ziffer entspricht 4 Bit

Suffixe: `U` oder `u`: unsigned Integer
`L` oder `l`: long Integer
`LL` oder `ll`: long long Integer (nur C++11)

Beispiele:

```
2L          = long
0302        = int (oktale Konstante)
100UL       = unsigned long
0x10        = int (hexadezimale Konstante)
0xFL        = long (hexadezimale Konstante)
```

Einfache Datentypen und Operatoren
Ganze Zahlen - Operatoren

Operator	Beispiel	Bedeutung
Arithmetische Operatoren:		
+	+i	unäres Plus (kann weggelassen werden)
-	-i	unäres Minus
++	++i	vorherige Inkrementierung um eins
	i++	nachfolgende Inkrementierung um eins
--	--i	vorherige Dekrementierung um eins
	i--	nachfolgende Dekrementierung um eins
+	i + 2	binäres Plus
-	i - 5	binäres Minus
*	5 * i	Multiplikation
/	i / 6	Division
%	i % 4	Modulo (Rest mit Vorzeichen von i)
=	i = 3 + j	Zuweisung
*=	i *= 3	i = i * 3
/=	i /= 3	i = i / 3
%=	i %= 3	i = i % 3
+=	i += 3	i = i + 3
-=	i -= 3	i = i - 3

relationale Operatoren:		
<	i < j	kleiner als
>	i > j	größer als
<=	i <= j	kleiner gleich
>=	i >= j	größer gleich
==	i == j	gleich
!=	i != j	ungleich
Bit-Operatoren:		
<<	i << 2	Linksschieben (Multiplikation mit 2er-Potenzen)
>>	i >> 1	Rechtsschieben (Division durch 2er-Potenzen)
&	i & 7	bitweises UND
^	i ^ 7	bitweises XOR (Exklusives Oder)
|	i | 7	bitweises ODER
~	~i	bitweise Negation
<<=	i <<= 3	i = i << 3
>>=	i >>= 3	i = i >> 3
&=	i &= 3	i = i & 3
|=	i |= 3	i = i | 3

Einfache Datentypen und Operatoren
Reele Zahlen

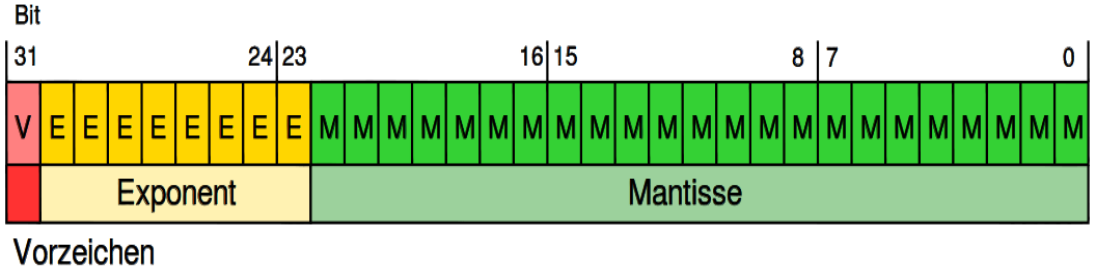
Wertebereiche:

Typ	Bits	Zahlenbereich		Stellen Genauigkeit
float	32	$\pm\ 1.1*10^{-38}$	$. . . \pm\ 3.4*10^{38}$	7
double	64	$\pm\ 2.2*10^{-308}$	$. . . \pm 1.8*10^{308}$	15
long double	96	$\pm\ 3.4*10^{-4932}$	$. . . \pm 1.2*10^{4932}$	24

Aufteilung der Bits:

	`float`	`double`	`long double`
Vorzeichen	1	1	1
Mantisse	23	52	80
Exp-Vorzeichen	1	1	1
Exponent	7	10	14
Gesamtanzahl Bits	**32**	**64**	**96**

i.d.R. Nutzung des IEEE Standard 754 (hier `float`):



Einfache Datentypen und Operatoren
Reele Zahlen - Wertebereich

```
#include <iostream>
#include<limits>           // hier sind die Bereichsinformationen
using namespace std;

int main() {
   cout << "Grenzwerte für Float-Zahl-Typen:" << endl;
   cout  << "Float-Min: " << numeric_limits<float>::min() << endl;
   cout  << "Float-Max: " << numeric_limits<float>::max() << endl;
   cout  << "Double-Min: " << numeric_limits<double>::min() << endl;
   cout  << "Double-Max: " << numeric_limits<double>::max() << endl;
   cout  << "Long-Double-Min: " << numeric_limits<long double>::min()
                                                          << endl;
   cout  << "Long-Double-Max: " << numeric_limits<long double>::max()
                                                          << endl;

   cout << "Anzahl der Bytes für:" << endl;
   cout << "float         " << sizeof(float) << endl;
   cout << "double        " << sizeof(double) << endl;
   cout << "long double   " << sizeof(long double) << endl;
}
```

```
Grenzwerte für Float-Zahl-Typen:
Float-Min: 1.17549e-38
Float-Max: 3.40282e+38
Double-Min: 2.22507e-308
Double-Max: 1.79769e+308
Long-Double-Min: 3.3621e-4932
Long-Double-Max: 1.18973e+4932
Anzahl der Bytes für:
float         4
double        8
long double   12
```

Einfache Datentypen und Operatoren
Reele Zahlen

-Literale:

[Vorzeichen]Vorkommastellen.Nachkommastellen[[e,E]Exponent][Suffix]

Suffixe: `f` oder `F`: `float`

`L` oder `l`: `long double`

Ohne Suffix ist eine Zahl vom Typ `double`!

Beispiel:

```
-123.789e6f : float
 1.8E6      : double
 88.009     : double
 1e-03      : double
 1.8L       : long double
 5f         : float
```

Einfache Datentypen und Operatoren
Reele Zahlen – endliche Genauigkeit

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

int main() {
     float a = 1.234567E-7, b = 1.000000, c = -b, s1, s2;
     s1 = a + b;
     s1 += c;                // entspricht s1 = s1 + c;
     s2 = a;
     s2 += b + c;
     cout << s1 << '\n';
     cout << s2 << '\n';
}
```

```
1.19209e-07
1.23457e-07
```

- Subtraktion zweier fast gleicher Werte: „numerischen Auslöschungen“

- Divison durch zu kleine Werte oder Multiplikation mit zu großen Werten führt zum Überlauf (*overflow*).

- Bei zu kleinen Werten kann es zu einem Unterlauf (*underflow*) kommen. Das Resultat ist dann 0.

- Ergebnisse können von der Reihenfolge der Berechnungen abhängen (s.o.)

Einfache Datentypen und Operatoren
Konstante

Konstante Werte können durch das Schlüsselwort 'const' vor der Deklaration ausgedrückt werden:

```
const int ANZAHL = 90;
const double MY_PI = 3.14159;
```

- Konstante Werte dürfen nur bei der Deklaration zugewiesen werden!
- Konstante werden typischerweise großgeschrieben!

<u>Weitere Möglichkeiten:</u>

- Präprozessor-Definition:

```
#define MY_PI 3.14159;
```

Nachteile: Präprozessor-Definition kann nachträglich geändert oder gelöscht werden. Keine 'echte' Konstante, sondern nur eine Text-Ersetzung durch den Präprozessor.

Einfache Datentypen und Operatoren
Zeichen (*char*)

Datentypen:

`char`
8-Bit, je nach System
`signed` oder `unsigned`

`signed char`
8 Bit, -128...127

`unsigned char`
8 Bit, 0...255

`wchar_t` ('wide' char)
> 8 Bit, für z.B. UTF8

Escape-Sequenzen

Zeichen	Bedeutung	ASCII-Name
\a	Signalton	BEL
\b	Backspace	BS
\f	Seitenvorschub	FF
\n	neue Zeile	LF
\r	Zeilenrücklauf	CR
\t	Tabulator	HT
\v	Zeilensprung	VT
\\	Backslash	
\'	'	
\"	"	
	z = Platzhalter	
z	*z* ist eine Folge von Oktalziffern,	
	Beispiel: \377	
\0	Spezialfall davon (Nullbyte)	NUL
\x*z*, \X*z*	*z* ist eine Folge von Hex-Ziffern, Beispiel: \xDB oder \X1f	

Einfache Datentypen und Operatoren
Zeichen (*char*)

Literale und Zuweisung:

```
char STERN = '*';
char a;
a = 'a';
a = STERN;
a = '\t';
```

`char` ↔ `int` Umwandlung:

```
char c;
int i;
i = static_cast<int>(c);
i = (int) c;
i = int(c);
i = c;

i = 66; // ASCII-Wert von 'B'
c = static_cast<char>(i);
char c = '5';
int ziffer = c - '0';
```

Sinnvolle Operatoren für char:

Operator	Beispiel	Bedeutung
=	d = 'A'	Zuweisung
<	d < f	kleiner als
>	d > f	größer als
<=	d <= f	kleiner gleich
>=	d >= f	größer gleich
==	d == f	gleich
!=	d != f	ungleich

Das Rechnen mit echten Zeichen-Werten (ASCII) gibt normalerweise keinen Sinn.
In Ausnahmefällen können einfache arithmetische Operationen Sinn geben.
(siehe Berechnug des Ziffernwertes aus dem ASCII-Wert auf der linken Seite)

Einfache Datentypen und Operatoren
Zeichen – ASCII Tabelle

Dez	Hex	Okt	
0	0x00	000	NUL
1	0x01	001	SOH
2	0x02	002	STX
3	0x03	003	ETX
4	0x04	004	EOT
5	0x05	005	ENQ
6	0x06	006	ACK
7	0x07	007	BEL
8	0x08	010	BS
9	0x09	011	TAB
10	0x0A	012	LF
11	0x0B	013	VT
12	0x0C	014	FF
13	0x0D	015	CR
14	0x0E	016	SO
15	0x0F	017	SI
16	0x10	020	DLE
17	0x11	021	DC1
18	0x12	022	DC2
19	0x13	023	DC3
20	0x14	024	DC4
21	0x15	025	NAK
22	0x16	026	SYN
23	0x17	027	ETB
24	0x18	030	CAN
25	0x19	031	EM
26	0x1A	032	SUB
27	0x1B	033	ESC
28	0x1C	034	FS
29	0x1D	035	GS
30	0x1E	036	RS
31	0x1F	037	US

Dez	Hex	Okt	
32	0x20	040	SP
33	0x21	041	!
34	0x22	042	"
35	0x23	043	#
36	0x24	044	$
37	0x25	045	%
38	0x26	046	&
39	0x27	047	'
40	0x28	050	(
41	0x29	051	)
42	0x2A	052	*
43	0x2B	053	+
44	0x2C	054	,
45	0x2D	055	-
46	0x2E	056	.
47	0x2F	057	/
48	0x30	060	0
49	0x31	061	1
50	0x32	062	2
51	0x33	063	3
52	0x34	064	4
53	0x35	065	5
54	0x36	066	6
55	0x37	067	7
56	0x38	070	8
57	0x39	071	9
58	0x3A	072	:
59	0x3B	073	;
60	0x3C	074	<
61	0x3D	075	=
62	0x3E	076	>
63	0x3F	077	?

Dez	Hex	Okt	
64	0x40	100	@
65	0x41	101	A
66	0x42	102	B
67	0x43	103	C
68	0x44	104	D
69	0x45	105	E
70	0x46	106	F
71	0x47	107	G
72	0x48	110	H
73	0x49	111	I
74	0x4A	112	J
75	0x4B	113	K
76	0x4C	114	L
77	0x4D	115	M
78	0x4E	116	N
79	0x4F	117	O
80	0x50	120	P
81	0x51	121	Q
82	0x52	122	R
83	0x53	123	S
84	0x54	124	T
85	0x55	125	U
86	0x56	126	V
87	0x57	127	W
88	0x58	130	X
89	0x59	131	Y
90	0x5A	132	Z
91	0x5B	133	[
92	0x5C	134	\
93	0x5D	135	]
94	0x5E	136	^
95	0x5F	137	_

Dez	Hex	Okt	
96	0x60	140	`
97	0x61	141	a
98	0x62	142	b
99	0x63	143	c
100	0x64	144	d
101	0x65	145	e
102	0x66	146	f
103	0x67	147	g
104	0x68	150	h
105	0x69	151	i
106	0x6A	152	j
107	0x6B	153	k
108	0x6C	154	l
109	0x6D	155	m
110	0x6E	156	n
111	0x6F	157	o
112	0x70	160	p
113	0x71	161	q
114	0x72	162	r
115	0x73	163	s
116	0x74	164	t
117	0x75	165	u
118	0x76	166	v
119	0x77	167	w
120	0x78	170	x
121	0x79	171	y
122	0x7A	172	z
123	0x7B	173	{
124	0x7C	174	|
125	0x7D	175	}
126	0x7E	176	~
127	0x7F	177	DEL

Einfache Datentypen und Operatoren
Logischer Datentyp `bool`

- Name geht zurück auf Georg Boole (1815-1864)
- Literale (Konstanten): **`true`** und **`false`**
- Ausgabe standardmäßig als Zahl (0/1), nach setzten der boolalpha-Flagge auch als Text (wie bei Java).
- Speicherung als ein Byte (`sizeof(bool)` ist 1)

Beispiel:

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

int main() {
  bool istGrossBuchstabe;
  char c;
  cin >> c;
  istGrossBuchstabe = (c >= 'A' ) && (c <= 'Z');
  cout << "Ergebnis: " << istGrossBuchstabe << endl;
  cout.setf(ios_base::boolalpha); // Textformat einschalten
  cout << "Ergebnis: " << istGrossBuchstabe << endl;
}
```

```
U<ENTER>
Ergebnis: 1
Ergebnis: true
```

Einfache Datentypen und Operatoren
Logischer Datentyp `bool`

Typumwandlung von `bool` zu integer-Werten:

```
bool wahrheitswert = true;
wahrheitswert = !wahrheitswert;  // Negation
cout << wahrheitswert << endl;  // 0, d.h. false
// Beispiel mit int-Zahlen: Aus 0 wird 1 und aus einer Zahl ungleich 0
// wird durch die Negation eine 0:
int i = 17, j;
j = !i;  // 0 (implizite Typumwandlung)
i = !j;  // 1 (implizite Typumwandlung)
// Typumwandlung von int nach bool
wahrheitswert = 99;  // true
wahrheitswert = 0;   // false
```

Operatoren für logische Datentypen:

Operator	Beispiel	Bedeutung
!	!i	logische Negation
&&	a && b	logisches UND
||	a || b	logisches ODER
==	a == b	Gleichheit
!=	a != b	Ungleichheit
=	a = a && b	Zuweisung

Einfache Datentypen und Operatoren
Referenzen

- Datentyp, der einen Verweis auf ein Objekt/eine Variable liefert.
- Eine Referenz ist daher so etwas wie ein Alias-Namen!
- Syntax: Das &-Zeichen steht zwischen Datentyp und Namen
- Der &-Operator hat damit (leider) 3 Bedeutungen:
  `address-of`-Operator, bitweises UND, Deklaration als Referenz
- Referenzen **müssen** bei der Deklaration initialisiert werden!
- Referenzen können (im Gegensatz zu Pointern) **NIE** null werden!

Beispiel:

```
int i = 2, j = 9;
int& r = i;  // Referenz auf i (r ist ein Alias für i)
int& s = r;  // Was passiert hier?
r = 10;      // aendert i und s
r = j;       // Wirkung: i = j; KEINE Umdefinition der Ref.
j = 5;
cout << i << endl << r << endl << s << endl << j << endl;
cout << &i << endl;  // Auf welche Adresse zeigt i ?
cout << &s << endl;  // Auf welche Adresse zeigt s ?
```

```
9
9
9
5
0xbf8e9c90
0xbf8e9c90
```

Einfache Datenty Ausdrücke

Wie werden folgende Ausdrücke ausgewertet?

```
i = 1,2;
a[2,3];
if(a =! 2)
x = msb << 4 + lsb
c = getchar() != EOF
val&mask != 0
a < b < c
cout << a <<2 ;
int *fp()
std::cout << a & b;
*p++
a = b = c;  a *= b += c;
int i=2;
int j=i++;
i = 3 * i++;
int t=0;
int sum = (t=3) + (++t);
```

Merke: Reihenfolge der Auswertung von Unter-ausdrücken ist **NICHT** definiert!

Precedence	Operator	Description	Associativity
1	::	Scope resolution	Left-to-right
2	++ --	Suffix/postfix increment and decrement	
	()	Function call	
	[]	Array subscripting	
	.	Element selection by reference	
	->	Element selection through pointer	
3	++ --	Prefix increment and decrement	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(*type*)	Type cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of	
	new, new[]	Dynamic memory allocation	
	delete, delete[]	Dynamic memory deallocation	
4	.* ->*	Pointer to member	Left-to-right
5	* / %	Multiplication, division, and remainder	
6	+ -	Addition and subtraction	
7	<< >>	Bitwise left shift and right shift	
8	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
9	== !=	For relational = and ≠ respectively	
10	&	Bitwise AND	
11	^	Bitwise XOR (exclusive or)	
12	|	Bitwise OR (inclusive or)	
13	&&	Logical AND	
14	||	Logical OR	
15	?:	Ternary conditional	Right-to-left
	=	Direct assignment (provided by default for C++ classes)	
	+= -=	Assignment by sum and difference	
	*= /= %=	Assignment by product, quotient, and remainder	
	<<= >>=	Assignment by bitwise left shift and right shift	
	&= ^= |=	Assignment by bitwise AND, XOR, and OR	
16	throw	Throw operator (for exceptions)	
17	,	Comma	Left-to-right

Einfache Datentypen und Operatoren
Standard-Typumwandlungen

In C++ können eingebaute Datentypen implizit (d.h. ohne explizite Umwandlung) ineinander umgewandelt werden. Daher ist z.B. gültig:

```
double d = 12e35;
int i1 = 3.1415927;
int i2 = d;
unsigned int i3 = -1;
char c = i2;
bool b = 3.141;
```

Dabei kann es allerdings zu unerwünschten Effekten kommen:

- Genauigkeitsverlust bei `double` → `float` (Mantisse und Exponent)
- Abschneiden des Nachkommateils bei `double`/`float` → `int`
- Bereichsüberschreitung bei großer `double`-Zahl → `int`
- Abschneiden der höherwertigen Bits bei z.B. `int` → `char` oder `long long` → `int`
- Vorzeichenverlust/Wertänderung bei z.B. -1 → `unsigned char`/`short`/`int`

Java ist in dieser Hinsicht deutlich restriktiver!!

Namensräume

Namensräume sind eine Möglichkeit, den globalen Namensraum in definierte Unterbereiche aufzuteilen (ähnlich wie `packages` in Java).

Mit der `using`-Anweisung kann der Suchbereich um ganze Namensräume oder einzelne Funktionen/Variablen erweitert werden:

```
// 1. Pauschale Nutzung
using namespace std; // macht alles aus std bekannt!
cout << “Hallo“ << endl;

// 2. Gezielte Nutzung einzelner Elemente
using std::cout;
using std::endl;
cout << “Hallo“ << endl;

// 3. Kein using, sondern mit qualifizierten Namen (Namensraum und scope-Operator)
std::cout << “Hallo“ << std::endl;
```

Gültigkeitsbereich und Sichtbarkeit

- Deklarationen nur in DEM Block gültig, in dem sie deklariert wurden.
- Variablen sind auch für innerhalb des Blocks neu angelegte innere Blöcke gültig
- Sichtbarkeit wird eingeschränkt, wenn Variablen gleichen Namens weiter 'innen' deklariert werden.
- Mit dem Scope-Operator `::` können Namensräume (auch der globale Namensraum) direkt angesprochen werden.

Beispiel:

```
#include<iostream>
using namespace std;
int a = 1, b = 2;

int main()  {   // Ein neuer Index beginnt.
    cout << "globales a= " << a << endl;  // Ausgabe von a
    int a = 10;
    cout << "lokales a= " << a << endl;
    cout << "globales ::a= " << ::a << endl;
    {
       int b = 20;
       int c = 30;
       cout << "lokales b = " << b << endl;
       cout << "lokales c = " << c << endl;
       cout << "globales ::b = " << ::b << endl;
    }
    cout << "globales b wieder sichtbar: b = " << b << endl;
 // cout << "c = " << c << endl;} // Fehler
}
```

Kontrollstrukturen

Ein C++-Programm ist formatfrei (d.h. Leerzeichen und neue Zeilen stören nicht...) und case-sensitiv!

Ein C++-Programm besteht aus **Anweisungen**, z.B.

- -Deklarationsanweisung `int i; // An bel. Stelle!!`
- -Ausdrucksanweisung `cout << „hallo“;`
- -Zuweisung `a=b;`
  lvalue = rvalue
- -Auswahlanweisung `if(..) …` `switch(..) …`
- -Schleifenanweisung `while(..) …`
- -Verbundanweisung, Block `{ … }`

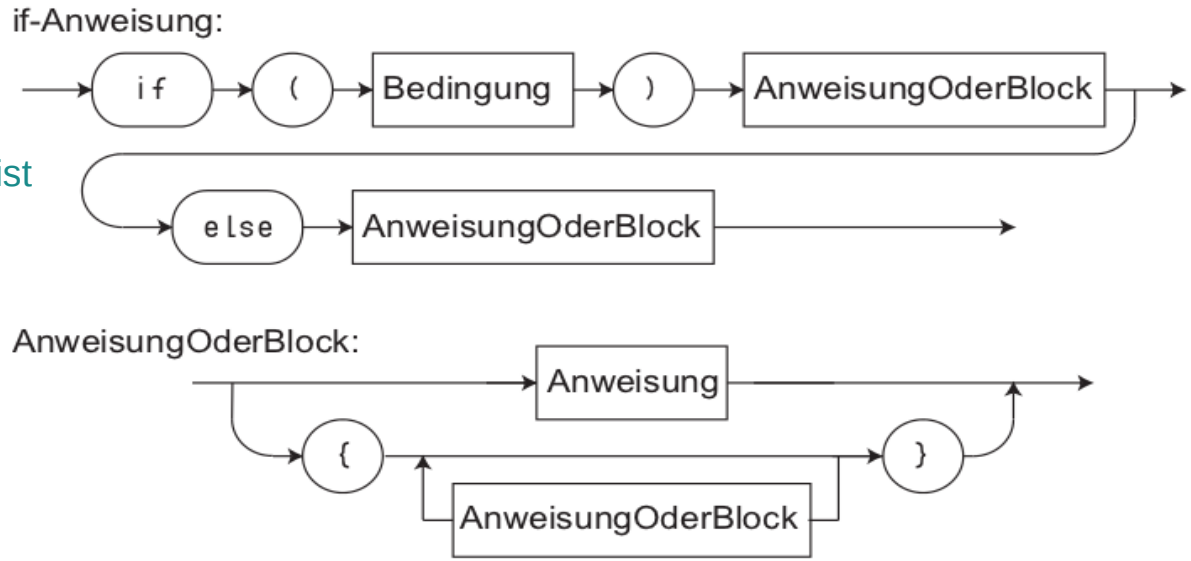
Sequenz
Einfache Reihung von Anweisungen.
Möglichst nur eine Anweisung pro Zeile!

```
a = b + 1;
a += a;
cout << „Das Ergebnis ist „ << a;
```

Kontrollstrukturen
Verzweigung / Auswahlanweisung

Auswahlanweisung

Verzweigung im Programmfluss mit `if`:



Der else-Zweig ist optional!

```
if (a != 42) {
  cout << „a ist nicht 42“ << endl;
}
else
  cout << „a ist 42!!“ << endl;
```

Fallen:

```
if (a == b)
  if (a < c)
    cout << „a=b und a<c“ << endl;
else
  cout << „a ungleich b“ << endl;
```

Kontrollstrukturen
Bedingungsoperator ? :

Bedingungsoperator ? :

- Bedingte Auswertung eines Ausdruckes.
- Einziger Operator mit 3 Operanden!
- Ausdruck1 wird evaluiert, wenn die Bedingung true ist.
  Ansonsten wird Ausdruck2 evaluiert.


```
// Die Anweisung
max = a > b ? a : b;

// ist äquivalent zu

if(a > b) {
  max = a;
}
else {
  max = b;
}
```

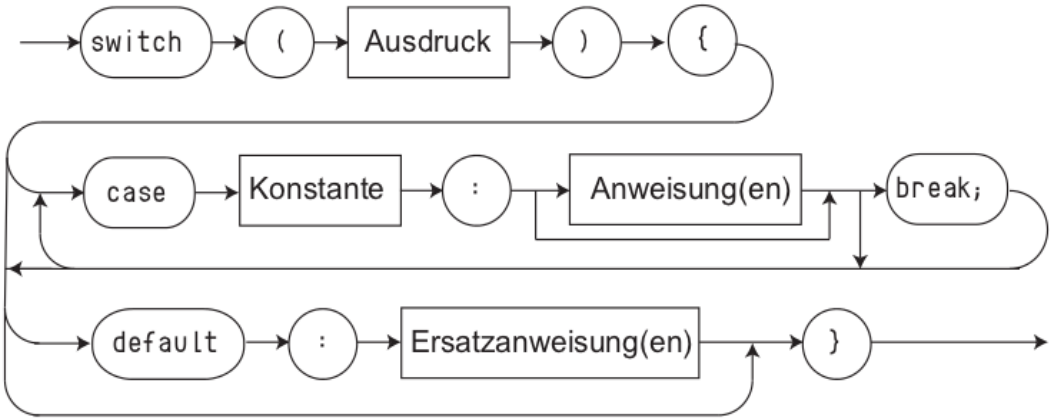
Die Lesbarkeit des Operators ist nicht sehr hoch → nur in Ausnahmefällen nutzen.

Kontrollstrukturen
Mehrfachauswahl/Fallunterscheidung

Fallunterscheidung mit switch

- Ausdruck nach `case` muss ein Integer-Typ sein (auch enums..)!
- `break` kann für ODER-Logik weggelassen werden.

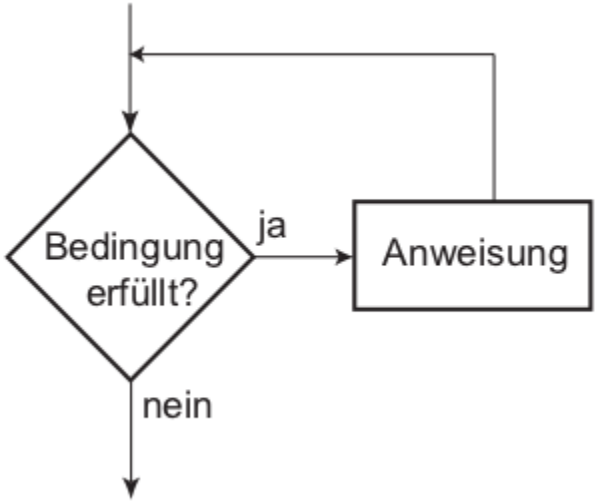
```
#include<iostream>
using namespace std;
int main() {
  int a = -1;
  char c;
  cout << "Zeichen ?" ;
  cin >> c;
  switch(c) {
    case 'I': a = 1;    break;
    case 'V': a = 5;    break;
    case 'X': a = 10;   break;
    case 'L': a = 50;   break;
    case 'C': a = 100;  break;
    case 'D': a = 500;  break;
    case 'M': a = 1000; break;
    default: a = 0;
  }
}
```



Kontrollstrukturen
Wiederholungen mit `while`

Wiederholung mit while:

- Klassische Form der kopfgesteuerten Schleife
- Vorsicht wie bei `if` : Bedingung kann auch `int`-Ausdruck sein...
- `break`: Beende Schleife
- `continue`: Überspringe restlichen Code im Schleifenkörper → nächste Iteration

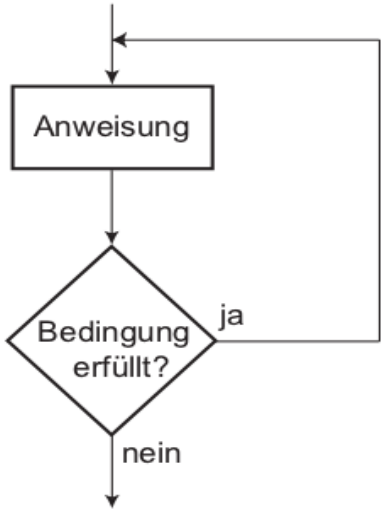



```
// Was macht der folgende Code?////
////////////////////////////////////
int sum = 0;
int n = 1;
int grenze = 99;
while(n++ <= grenze) {
  if (n%2) continue;
  sum += n;
}
```

Kontrollstrukturen
Wiederholungen mit `do-while`

Wiederholung mit do-while:

- Klassische Form der fußgesteuerten Schleife
- Schleifenkörper wird immer mindestens einmal ausgeführt!
- `break`: Beende Schleife
- `continue`: Überspringe restlichen Code im Schleifenkörper → nächste Iteration

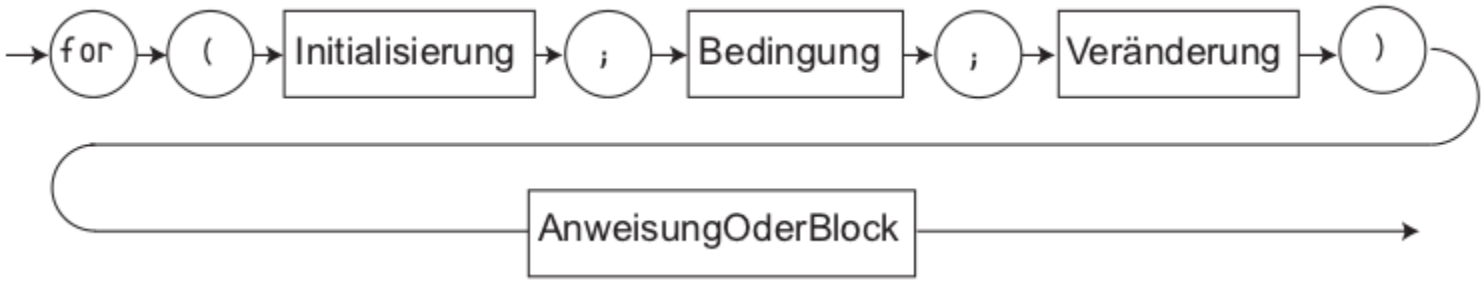



```
// Was macht der folgende Code?////
////////////////////////////////////
int sum = 0;
int n = 1;
int grenze = 99;
do {
  if (n%2) continue;
  sum += n;
} while(n++ <= grenze);
```

Kontrollstrukturen
Wiederholungen mit `for`

Wiederholung mit for:

- Kopfgesteuerte Schleife mit *Initialisierung* und *Veränderung*
- `break`: Beende Schleife
- `continue`: Überspringe restlichen Code im Schleifenkörper und führe nächste Veränderung durch!




```
for (Initialisierung; Bedingung; Veränderung)
  Anweisung
```

ist äquivalent mit:
(strenggenommen ohne `continue`!)

```
{
  Initialisierung;
  while(Bedingung) {
    Anweisung
    Veränderung;
  }
}
```

Kontrollstrukturen
Wiederholungen mit `for`

Sichtbarkeit von Deklarationen im Schleifenkopf:

```
int i;
for(i = 0; i < 100; ++i) {
  // Programmcode, i ist hier bekannt
}
// i ist weiterhin bekannt ...
/////////////////////////////////////
for(int j = 0; j < 100; ++j) {
  // Programmcode, j ist hier bekannt
}
// j ist hier nicht mehr bekannt ...
```

Einsatz des Komma-Operators:

```
int sum;
for(int i = 1, sum = 0; i <= 100; sum += i, ++i);
```

Dinge, die man nicht machen sollte:

- Laufvariablen vom Typ reeler Zahlen (`float, double`)
- Veränderung der Laufvariablen im Schleifenrumpf

Benutzerdefinierte und zusammengesetzte Datentypen – Aufzählungstypen (enums)

- Datentyp zur Darstellung i.d.R. nicht-numerischer Aufzählungen
- Wird intern auf integer abgebildet, ist aber eigenständiger Typ
- Interne integer-Werte können optional angegeben werden.


```
enum Wochentag {sonntag, montag, dienstag, mittwoch, donnerstag, freitag,
                samstag};
enum Farbtyp {rot = 0, gruen = 1, blau = 2, gelb = 4} farbe;

Wochentag derFeiertag, einWerktag, heute = dienstag;

int i = dienstag;      //richtig (implizite Umwandlung nach int)
heute = montag;        //richtig
heute = i;             //Fehler, Datentyp inkompatibel
montag = heute;        //Fehler (montag ist Konstante)
i = rot + blau;        //möglich (implizite Umwandlung nach int)
farbe = rot + blau;    //Fehler, Rückwandlung des int-Ergebnisses
                       //in Typ Farbtyp ist nicht möglich
```

Benutzerdefinierte und zusammengesetzte Datentypen – `vector<>`

- Container zur Aufnahme eines Arrays von Elementen desselben Datentyps (interner Aufbau folgt später...).
- Leichter verwendbar als die eingebauten C-Arrays (kommen später)

Benutzung:

```
#include <vector>
#include <cstddef>  // für size_t
using namespace std;

vector<int> v(10);      // Vector von 10 int-Werten
vector<double> w(20);   // Vector von 20 double-Werten

size_t size = v.size();  // Liefert die Größe des Vektors.
                         // size_t ist typischerweise 'unsigned int'

v[0] = 42;              // Setze erstes Element auf 42. Index ab 0 !
int value = v[8];       // Lese 9. Element und speichere in value

value = v.at(8);        // Wie oben, nur mit Bereichsüberprüfung!!

v.push_back(21);        // Dynamische Größenveränderung! v hat jetzt 11
                        // Elemente, der letzte Wert ist 21
```

Benutzerdefinierte und zusammengesetzte Datentypen – `string`

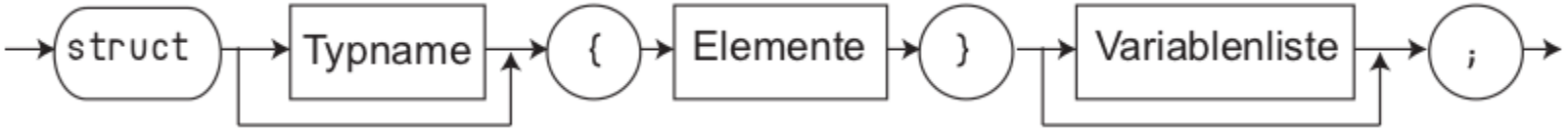
- Container zur Aufnahme einer Zeichenkette (Array aus `char`). Weitere Details folgen später...
- Leichter und sicherer verwendbar als C-`char`-Arrays und zugehörige Funktionen (strcat, strcpy, strlen, ...)

Benutzung:

```
#include<string>  // Standard-String einschließen
using namespace std;

string einString( " hallo " ); // Deklaration und Definition
char c = einString[2];         // wie bei vector
char d = einString.at(2);      // wie bei vector

einString = 'X';              // Autom. Umwandlung von char in string
einString = "Neuer Inhalt";  // Setze neuen Wert, alter wird freigegeben

string nochEiner("huhu");
einString = nochEiner;        // Echte Kopie, einString enthält "huhu"
string einDritter(einString);// Echte Kopie, einDritter enthält "huhu"

string neu1 = einString + "123"; // OK
string neu2 = "123" + einString; // OK
string neu3 = "Hallo" + "Echo";  // geht NICHT
```

Benutzerdefinierte und zusammengesetzte Datentypen – strukturierte Datentypen (`struct`)

- Zusammenfassung zusammengehörender Datenelemente unter einem Namen, der einen neuen Datentyp darstellt (kein `typedef` nötig wie in C).
- Zugriff auf Elemente mit . - Operator
- Bilden Grundlage für Klassen, können auch Methoden enthalten!




```
enum Farbtyp {rot, gelb, gruen};

struct Punkt {      // Punkt ist ein Typ.
    int x;
    int y;
    bool istSichtbar;
    Farbtyp dieFarbe;
} p; // p ist ein Punkt-Objekt

Punkt q ;// q ist ebenfalls ein Punkt-Objekt

// Zugriff
p.x = 270; p.y = 20; // Koordinaten von p
p.istSichtbar = false;
p.dieFarbe = gelb;
```

Einfache Ein- und Ausgabe
Standard Ein- und Ausgabe

3 Standard-Streams:

`cin` Standardeingabe (Tastatur)
`cout` Standardausgabe (Bildschirm)
`cerr` Standardfehlerausgabe (Bildschirm)

Besonderheiten bei der Eingabe:

- Führende Zwischenräume (auch \t, \r, \v, \f, \n) werden ignoriert.
  Alternative: Einlesen einzelner Zeichen oder einer ganzen Zeile:

```
// Einzelnes Zeichen einlesen
char c;
cin.get(c);
```

```
// Ganze Zeile einlesen
string s;
getline(cin, s);
```

- Zwischenräume (bzw. auch Zeichen, die nicht zu einem Datentyp passen, z.B. ein Buchstabe bei einer `int`-Zahl) werden als Endekennung genutzt
- Eingaben werden erst nach dem Drücken der <ENTER>-Taste in den Tastaturpuffer gelegt und an das Programm weitergegeben
- Ungenutzte Zeichen (je nach Parameter von `cin`) bleiben im Tastaturpuffer, und werden beim nächsten `cin` ausgewertet.

Einfache Ein- und Ausgabe
Standard Ein- und Ausgabe

- Der gesamte `cin`-Ausdruck kann zu einem bool konvertiert werden (auch als Parameter von `if/while` ...), der `true` liefert, wenn alle Parameter erfolgreich eingelesen werden konnten, sonst `false`.

Beispiele:

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

int main() {

 string s;
 int i;
 double d;

 bool erg = (cin >> s >> i >> d);

 cout << " s: " << s;
 cout << " i: " << i;
 cout << " d: " << d;
 cout << " erg: " << erg << endl;
}
```

```
     Hallo 42 .9e6<ENTER>
 s: Hallo i: 42 d: 900000 erg: 1

Ein kleiner Test<ENTER>
 s: Ein i: 0 d: 4.86135e-270 erg: 0

  NochEinKleinerTest 2.7<ENTER>
 s: NochEinKleinerTest i: 2 d: 0.7 erg: 1

MATSE<ENTER>
    34<ENTER>
3.1415927<ENTER>
 s: MATSE i: 34 d: 3.14159 erg: 1

1 2 3 4 5 6 7<ENTER>
 s: 1 i: 2 d: 3 erg: 1
```

Einfache Ein- und Ausgabe
Standard Ein- und Ausgabe

Formatierung der Ausgabe:
Mit `cout.`*`funktion(...)`*`;` kann das Verhalten der Ausgabe manipuliert werden, z.B.:

```
cout.width(7);  // Setze die Feldbreite auf 7 Zeichen (wirkt nur auf das
                // folgende cout!!
cout.fill('0'); // Setze Füllzeichen auf '0'
cout.precision(4); // Setze Anzahl Ziffern bei Fließkommazahlen
```

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

int main() {

 cout.precision(5);
 cout.fill('#');
 double d = 0.111111111;
 for (int i=0; i < 10; i++) {
   cout.width(10);
   cout << d << endl;
   d += 12.0;
 }
}
```

```
###0.11111
####12.111
####24.111
####36.111
####48.111
####60.111
####72.111
####84.111
####96.111
####108.11
```

Einfache Ein- und Ausgabe
Ein- und Ausgabe mit Dateien

- Mit `#include<fstream>` gibt es die neuen Datentypen

  `ifstream` Input File Stream
  `ofstream` Output File Stream,

  mit denen man im Wesentlichen wie mit `cin` und `cout` arbeiten kann!

Wichtige Methoden:

```
open("dateiname" [, flags]);  // Datei öffnen
```

Bei Textdateien können die flags weggelassen werden.
Bei Binärdateien: `ios::binary|ios:in` bei `ifstream`
`ios::binary|ios.out` bei `ofstream`

```
close();  // Datei schliessen

get(c);   // bei ifstream: Einzelnes Zeichen lesen
put(c);   // bei ofstream: Einzelnes Zeichen schreiben
```

Einfache Ein- und Ausgabe
Ein- und Ausgabe mit Dateien

Beispiel:

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

int main() {

  string s; int i; double d;
  cin >> s >> i >> d;

  ofstream ofs;
  ofs.open("test.file");
  if (ofs) {
    ofs << s << " " << i << " " << d;
    ofs.close();
  } else {
    cerr << "open ofs Fehler" << endl;
    exit(-1);
  }

  s=""; i=0; d=0.0;

  ifstream ifs;
  ifs.open("test.file");
  if (ifs) {
    ifs >> s >> i >> d;
    ifs.close();
  } else {
    cerr << "open ofs Fehler" << endl;
    exit(-1);
  }

  cout << " s: " << s;
  cout << " i: " << i;
  cout << " d: " << d << endl;
}
```

```
EinText 12 123.45<ENTER>
 s: EinText i: 12 d: 123.45
```

Inhalt der Datei test.file:

```
EinText 12 123.45
```

Programmstrukturierung
Funktionen

- Klassisches „Teile und herrsche“-Prinzip
- Basis für *Funktionsbibliotheken*
- Deklaration („Funktionsprototyp“) und Definition können getrennt sein

Deklaration:


```
int          maximum     ( double d1, double d2 );
```

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

unsigned long fakultaet(int);// Funktionsprototyp
int main() {
   cout << "Fakultät von 42 ist "
        << fakultaet(42) << endl;    // Aufruf
}

// Funktionsimplementation (Definition)
unsigned long fakultaet(int zahl) {
   unsigned long fak = 1;
   for(int i = 2; i <= zahl; ++i)
      fak *= i;
   return fak;
}
```

Mögliche Parameterlisten (bei Deklaration können Parameter-Namen weggelassen werden):

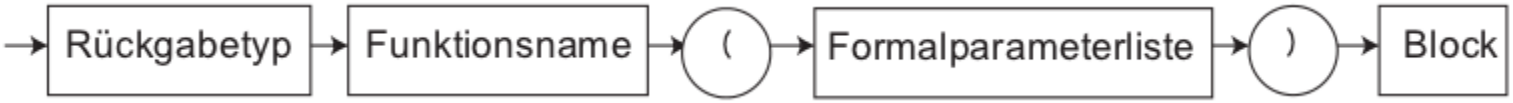
```
int func1();
double func2(void);
void func3(int, char);
char * func4(int i, char y);
```

Bei `void` als Rückgabetyp braucht die Funktion keine `return`-Anweisung am Ende!

Programmstrukturierung
Funktionen

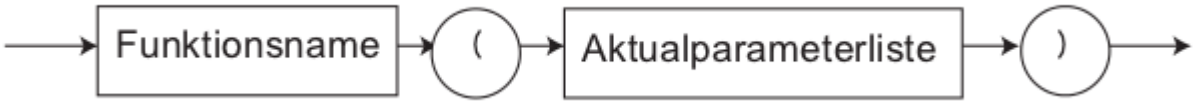
- Bei der Definition müssen Parameternamen angegeben werden.
- Die Parameter verhalten sich wie lokale Variablen, die nur in der Funktionsdefinition sichtbar sind.

Definition:



- Beim Aufruf werden die aktuell übergebenen Parameter, die passende Typen haben müssen, an die Stelle der Funktionsparameter gesetzt.
- Der Funktionsaufruf selber verhält sich wie ein Ausdruck vom Typ des Rückgabetyps.

Aufruf:



Programmstrukturierung
Funktionen mit Gedächtnis: `static`

- Normale lokale Variable in Funktionen werden zerstört, so bald der sie enthaltende Block verlassen wird.
- Bei der Deklaration einer Variable als `static` wird die Variable an einen festen Speicherort gelegt, und nur beim ersten Aufruf der Funktion initialisiert. Dadurch wirkt sie wie ein „Gedächtnis“ der Funktion.

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

void func( ) {   // zählt die Anzahl der Aufrufe
    static int anz = 0;
    cout << "Anzahl = " << ++anz << endl;
}

int main() {
   for (int i = 0; i < 6; ++i)  func( );
}
```

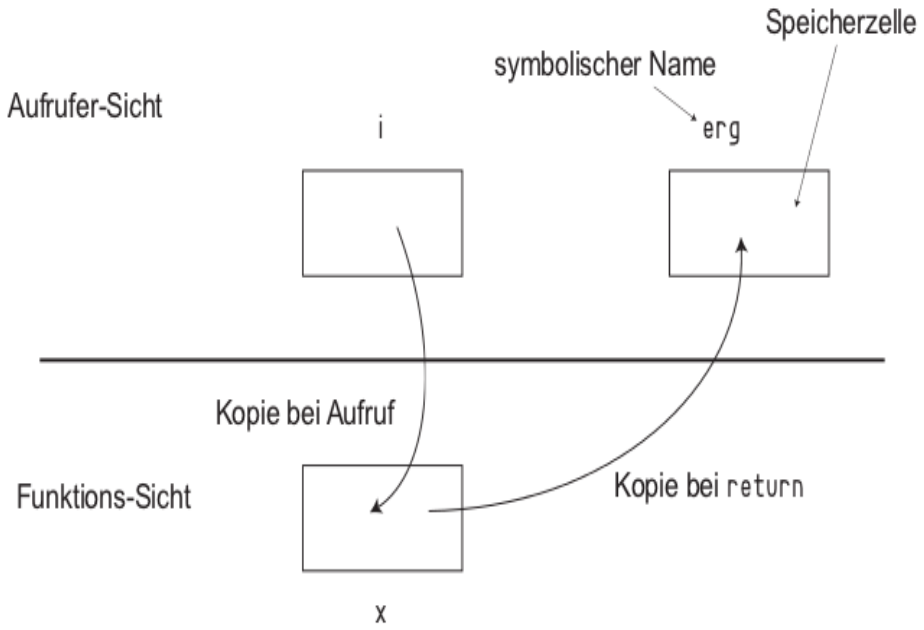
```
Anzahl = 1
Anzahl = 2
Anzahl = 3
Anzahl = 4
Anzahl = 5
Anzahl = 6
```

- `static` wird auch zur Steuerung der Sichtbarkeit von globalen Variablen verwendet (s.u.). In C++ sollte dafür besser ein anonymer `namespace` verwendet werden!

Programmstrukturierung
Schnittstellen zum Datentransfer

Übergabe per Wert (*call by value*)

```
#include<iostream>
using namespace std;
int addiere_5(int);       // Deklaration

int main() {
    int erg, i = 0;
    cout << i << " = Wert von i\n";
    erg = addiere_5(i);
    cout << erg << endl;
    cout << i << " = i unverändert!\n";
}

int addiere_5(int x) {    // Definition
     x += 5;
     return x;
}
```



Übergabe per Wert ist sinnvoll wenn:

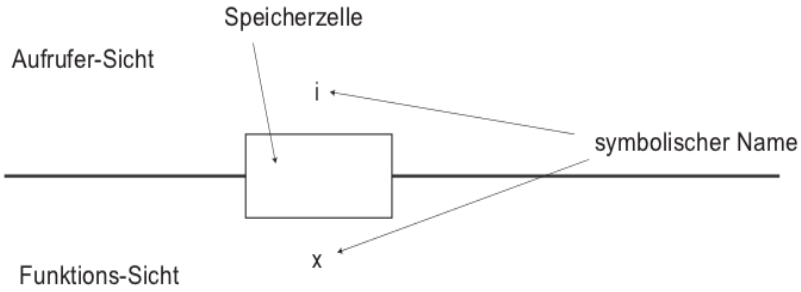
- der übergebene Wert nicht verändert werden soll/darf
- der übergebene Wert nicht zu viel Speicher verwendet (z.B. eingebaute Datentypen).

Programmstrukturierung
Schnittstellen zum Datentransfer

Übergabe per Referenz (*call by reference*)

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

void addiere_7(int&); // Parameter ist Referenz!

int main() {
    int i = 0;
    cout << i << endl;
    addiere_7(i);      // Syntax wie bei
                       // Übergabe per Wert!
    cout << i << endl;
}

void addiere_7(int& x) {
    x += 7;          // Original wird geändert!
}
```

Gleiches Objekt wird mit zwei ggf. verschiedenen Namen benutzt:



- Position des `&` ist egal (`int& i` oder `int &i`)
- Wenn Parameter nicht geändert werden soll: **`const <Typ> & <Name>`**
- Java übergibt alle Objekte-Paramter *per Referenz*, und alle einfachen Datentypen *per Wert*! C++ bietet da mehr Freiheiten!

Rückgabe von Referenzen

- Analog zu den Parametern gibt es Rückgabe per Wert (Kopie) und Rückgabe per Referenz
- Was ist bei folgendem Programm falsch? Warum wird in der letzten Zeile ein anderer Wert für `z` ausgegeben, obwohl davor `z` gar nicht verändert wurde?

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

int& maxwert(int a, int b) {
  if(a > b) return a;
    else
  return b;
}

int main() {
  int x = 17, y = 4;
  int& z = maxwert(x, y);
  cout << z << endl;
  int& z2 = maxwert(y, x);
  cout << z << endl;
}
```

```
17
4
```

- Wäre
  `int& maxwert(int &a, int &b)`
  eine funktionierende Alternative?

Programmstrukturierung
Schnittstellen zum Datentransfer

Vorgegebene Parameterwerte und variable Parameterzahl

- Die Deklaration (oder Definition) einer Funktion kann Standardwerte für Parameter enthalten.
- Dadurch können z.B. Funktionen um neue Funktionalitäten bzw. Aufrufmöglichkeiten erweitert werden:

```
// Funktionsprototyp, 2. Parameter mit Vorgabewert:
void preisAnzeige(double preis,
                  const string& waehrung="Deutsche Mark");

// Hauptprogramm
int main() {
     preisAnzeige(12.35);   // Default-Parameter wird eingesetzt
     preisAnzeige(99.99,"US-Dollar");
}

// Funktionsimplementation
void preisAnzeige(double preis, const string& waehrung) {
      cout << preis << ' ' << waehrung << endl;
}
```

```
12.35 Deutsche Mark
99.99 US-Dollar
```

- Bei Aufruf werden die Parameter von links nach rechts ersetzt. Man kann keine Parameter mit Standard-Werten 'überspringen'...

Programmstrukturierung
Schnittstellen zum Datentransfer

Überladen von Funktionen

- Eine Funktion kann mit demselben Namen in mehreren Variationen existieren, wenn die *Parameterlisten* unterschiedlich sind!
  Dabei sind z.B. `const int`, `int`, `int &`, `int *` verschieden!
- Ein unterschiedlicher Rückgabetyp reicht *nicht* zur Unterscheidung.

```
double maximum(double x, double y) {
    return x > y ?  x : y;
}

// zweite Funktion gleichen Namens, aber unterschiedlicher Signatur
int maximum(int x, int y) {
   return x > y ?  x : y;
}

int main() {
   double a=100.2, b= 333.777;
   int c=1700, d = 1000;
   cout << maximum(a,b) << endl;  // Aufruf von maximum(double, double)
   cout << maximum(c,d) << endl;  // Aufruf von maximum(int, int)
}
```

- Der Compiler versucht, die am besten passende Funktion zu finden.
  Bei Mehrdeutigkeiten gibt es eine Fehlermeldung!

Programmstrukturierung
Schnittstellen zum Datentransfer

Die Funktion main()

- Ist der Einsprungpunkt bei Programmausführung
- In C++ muss die Funktion einen `int` zurückliefern!
- Trotzdem darf das `return(...)` am Ende weggelassen werden, die Funktion gibt dann automatisch 0 zurück (Code für „kein Fehler“)
- Mit `exit(int)` aus `<cstdlib>` kann das Programm auch vorzeitig beendet werden.
- Es exisitieren 2 Versionen:

```
int main() { … }
```
Kein Zugriff auf Kommandozeilen-Parameter

```
int main(int argc, char* argv[]) { … }
```

`argc` = Anzahl der Parameter + 1

`argv[]` = Array aus C-Strings mit Kommandozeilen-Parametern
`argv[0]` ist Programmname!

Programmstrukturierung
Einbinden vorübersetzter Programmteile

- Normalerweise enthalten die Header-Dateien (`*.h`) die 'Schnittstelle', also Funktionsprototypen, ggf. globale Variablen und Klassen.
- Die Implementierungs-Dateien (`*.cpp`) enthalten den Programmcode.
- Bei größeren Programmsystemen sollte es eine eigene Datei für die `main()`-Funktion geben.

a.h

```
void func_a1();
void func_a2();
```

b.h

```
void func_b();
```

meinprog.cpp

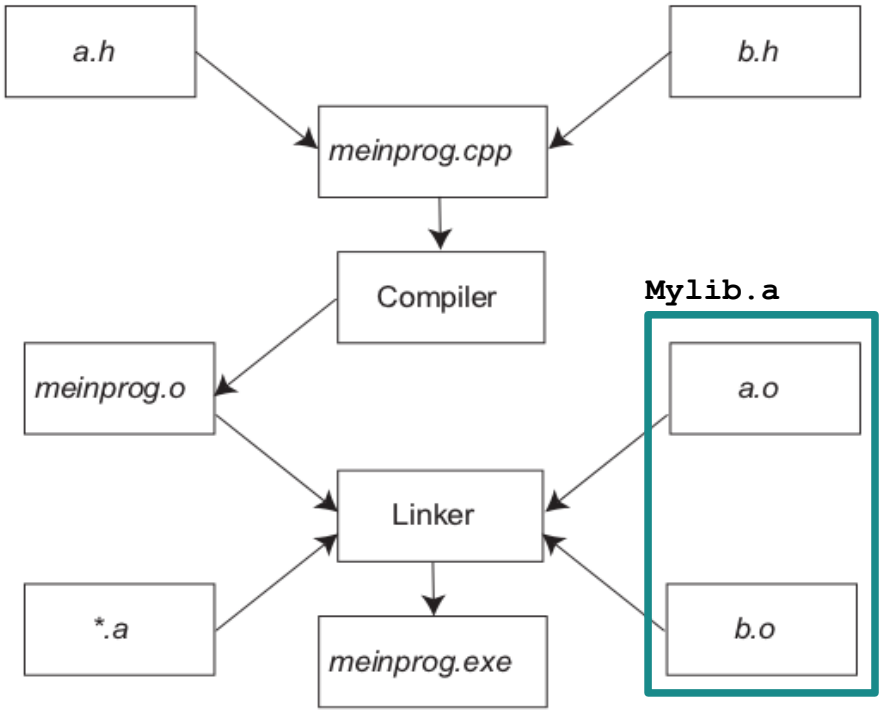
```
 / meinprog.cpp
#include "a.h"
#include "b.h"
int main() {
func_a1( );
func_a2( );
func_b( );
}
```

a.cpp

```
#include "a.h"
void func_a1() {
// Code zu func_a1
}
void func_a2() {
// Code zu func_a2
}
```

b.cpp

```
#include "b.h"
void func_b() {
// Code zu func_a1
}
```



Programmstrukturierung
Dateiübergreifende Gültigkeit und Sichtbarkeit

Speicherklassen und Modifizierer im Überblick

Jede Variable besitzt eine Speicherklasse:

`auto` (oder k.A.)	„Automatische“ (lokale) Variable. Wird i.d.R. Auf dem Stack angelegt. **Default**, wenn keine Speicherklasse angegeben wird!
`register`	Hinweis für den Compiler, die (lokale) Variable oder den Funktionsparameter möglichst in einem Register zu speichern. Bei Berechnung z.B. der Adresse einer solchen Variable wird sie allerdings immer im Arbeitsspeicher abgelegt.
`extern`	Sagt dem Compiler, dass die Variable in einer anderen Übersetzungseinheit definiert (also 'extern') ist. Bei der Definition einer `const`-Variable bedeutet `extern`, dass die Variable auch in anderen Übersetzungseinheiten sichtbar sein kann!
`static`	Bei lokalen Variablen: Variable mit 'Gedächtnis' (s.o.). Bei globalen Variablen: Die Variable ist nicht in anderen Übersetzungseinheiten sichtbar. Statische Variablen werden mit 0 initialisiert.
`mutable`	Spielt bei Klassen eine Rolle → wird später besprochen

Programmstrukturierung
Dateiübergreifende Gültigkeit und Sichtbarkeit

Speicherklassen und Modifizierer im Überblick

Zusätzlich zur Speicherklasse können Modifizierer angegeben werden:

const	Die Variable ist konstant und kann nicht verändert werden.
volatile	Der Wert der Variable kann sich auf eine nicht vom Compiler feststellbare Art und Weise verändern, z.B. durch HW-Ereignisse oder Multithreading. Dadurch, dass sich der Compiler nicht auf den zuletzt geschriebenen Wert verlassen kann, kann der Code ggf. schlechter optimiert werden.

Datei1.cpp

```
int glob1 = 90;
static int glob2 = 91;

const double d1=9.1;
extern const double d2=9.2;

void foo() {
  extern int g1;
  extern int g2;
  auto int i;  // wie int i;
  static int j;
}
```

Datei2.cpp

```
extern int glob1;
extern int glob2;

extern double d1;
extern double d2;

int g1;
```

```
Datei1.cpp:(.text+0xf): undefined
reference to 'g2'
Datei2.cpp:(.text+0x19): undefined
reference to 'glob2'
Datei2.cpp:(.text+0x29): undefined
reference to `d1'
collect2: ld returned 1 exit status
```

Programmstrukturierung
Übersetzungseinheit, Deklaration, Definition

Ein paar wichtige Begriffsdefinitionen:

- **Übersetzungseinheit** (engl. *compilation unit*)
  Den Programmcode, den der Compiler bei einem Durchlauf verarbeitet, nennt man Übersetzungseinheit
- **Deklaration**
  Eine Deklaration führt einen Namen in ein Programm ein und gibt dem Namen eine Bedeutung.
- **Definition**
  Eine Deklaration ist auch eine Definition, wenn *mehr* als nur der Name eingeführt wird, zum Beispiel wenn *Speicherplatz* für Daten oder Code angelegt oder die *innere Struktur* eines Datentyps beschrieben wird, aus der sich der benötigte Speicherplatz ergibt.

```
// Deklarationen
extern int a;
extern const float PI;
int f(int);
struct meinStrukt;
enum meinEnum;
```

```
// Definitionen
int a;
extern const float PI = 3.14159;
int f(int x) { return x * x;}
struct meinStrukt {
  int c;
  int d;
};
```

```
meinStrukt X;
enum meinEnum {li,re};
meinEnum Y;
```

Programmstrukturierung
One Definition Rule

- Jede Variable, Funktion, Struktur, Konstante und so weiter in einem Programm hat genau eine Definition. Das gilt auch bei mehreren Übersetzungseinheiten in einem Programm!

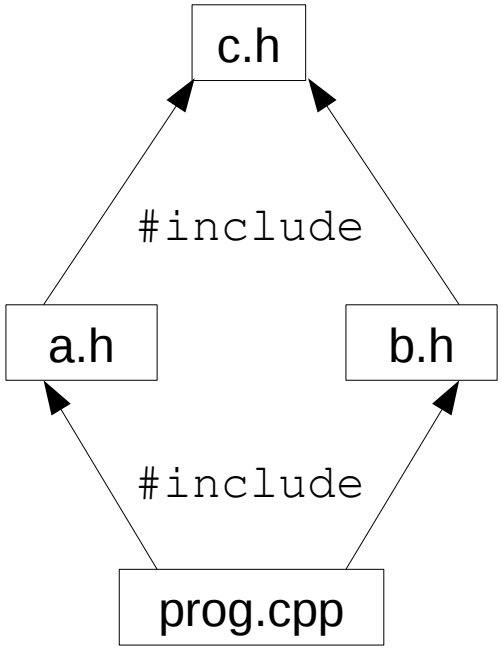
- Daher enthält eine **Header-Datei** typischerweise:
  - Reine Deklarationen von Funktionen (Funktionsprototypen), (globaler) Variablen und Konstanten
  - Definition von Konstanten, die nur in der Übersetzungseinheit sichtbar sind
  - Definition von Datentypen wie **`enum`** oder **`struct`** (später auch **`class`**)

- Und eine **Implementations-Datei** enthält:
  - Funktionsdefinitionen (Implementation)
  - Definition globaler Variablen/Objekte
  - Definition und Initialisierung globaler Konstanten

Programmstrukturierung
Include Guards

- Häufig wird eine Header-Datei unfreiwillig mehrfach eingebunden (siehe Abbildung rechts).

- Da Header-Dateien auch Definitionen enthalten, würde es dann zu Mehrfachdefinitionen und somit zu Compilerfehlern kommen!



Abhilfe:

1. Über Präprozessor-Makros

```
// c.h
#ifndef C_H
#define C_H

void func_c1();
void func_c2();
enum Farbtyp {rot,
gruen, blau, gelb};

#endif // C_H
```

2. Über `#pragma`-Anweisung (ggf. nicht portabel!)

```
// c.h
#pragma once

void func_c1();
void func_c2();
enum Farbtyp {rot,
gruen, blau, gelb};
```

Programmstrukturierung
Verifizieren logischer Annahmen

- Oft ist es sinnvoll, logische Annahmen während der Übersetzungs- oder Laufzeit zu überprüfen.

Während der Laufzeit: `assert`-Makro

- Steuerung des Makros über das `NDEBUG`-Makro: Wenn `NDEBUG` definiert ist (auch z.B. über die Compiler-Kommandozeile mit ... `-DNDEBUG` ...) expandiert das `assert`-Makro zu 'nichts';

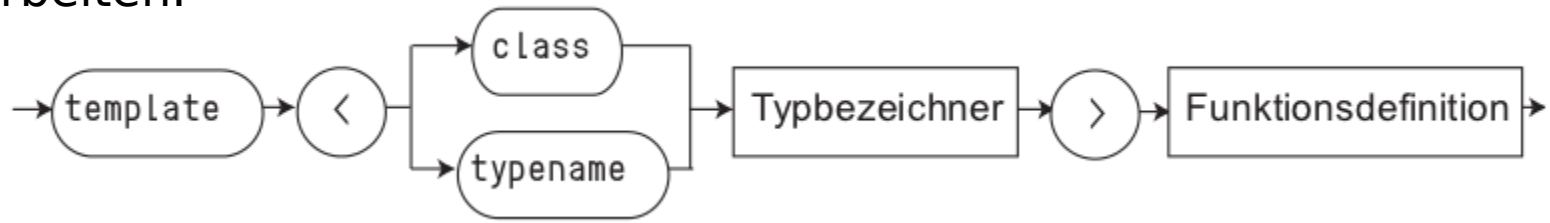
```
#include<cassert> // enthält Makrodefinition
const int GRENZE = 100;
int index;
// .... Berechnung von index
// Test auf Einhaltung der Grenzen:
assert(index >= 0 && index < GRENZE);
```

Während der Übersetzungszeit (neues Schlüsselwort in C++11)**:**

```
static_assert(sizeof(long) > sizeof(int), "long hat nicht mehr Bits als int!" );
```

Programmstrukturierung
Funktions-Templates (generische Funktionen)

Oft ist der Algorithmus in einer Funktion für verschiedene Eingabe-Datentypen identisch. Bisher hatten Funktionen aber feste Parameter-Datentypen. Funktions-Templates können verschiedene Datentypen verarbeiten.




```
template<typename T> // Template mit T als Parameter für den Datentyp (Platzhalter)
void tausche(T& a, T& b) {
  // a und b vertauschen
  const T TEMP = a;
  a = b;
  b = TEMP;
}
```

```
int main() {
  int i1 = 1, i2=2;
  double d1 = 1.0, d2 = 2.0;
  tausche(i1, i2);   // auch tausche<int>(...), aber Typangabe i.d.R. nicht notwendig
  tausche(d1, d2);
  // tausche(i1, d2); // Funktioniert nicht!
}
```

Programmstrukturierung
Funktions-Templates (generische Funktionen)

- Die Angabe der Template-Parameter als `<class arg>` oder `<typename arg>` macht keinen Unterschied. Manchmal verwendet man `typename` um auszudrücken, dass neben eigenen Datentypen (Klassen) auch elementare Datentypen wie `int` und `double` verwendet werden.

- Tempates werden zur Übersetzungszeit ausgewertet, d.h. der Compiler ersetzt im Template-Quellcode die 'Lücken' (also die Vorkommen der Template-Parameter) mit den konkreten Datentypen, die er aus den Aufrufparametern ableitet, oder die er durch konkrete Angabe der Datentypen (in `<>`-Klammern hinter dem Funktionsnamen beim Aufruf) übermittelt bekommt.

- Ein Template ist daher vergleichbar mit normalen C/C++-Präprozessor-Makros, allerdings viel sicherer in der Benutzung!

Programmstrukturierung
Funktions-Templates (generische Funktionen)

- Die Template-Parameter können mehrere Lücken oder auch Parameter von eingebauten Datentypen enthalten. Beim Aufruf der Template-Methode werden die ggf. notwendigen Parameter dann in spitzen Klammern nach dem Funktionsnamen angegeben, die zur Übersetzungszeit ausgewertet werden können müssen.

```
template<int T, typename U, typename V> // T, U und V sind 'Lücken'
void meineFunktion(U& a, V& b) {
  U var1;
  if (T == 0) { ...
}
int main() {
  int i=1, j=2;
  meineFunktion<12>(i, j);
}
```

- Wo ist der Unterschied zu folgender Template-Methode?

```
template<typename U, typename V> // U und V sind 'Lücken'
void meineFunktion(int i, U& a, V& b) {
  ...
}
int main() {
  int i=1, j=2;
  meineFunktion(12, i, j);
}
```

Programmstrukturierung
Funktions-Templates (generische Funktionen)

Spezialisieren von Templates:
Neben dem generischen Template-Code kann man für spezielle Datentypen eine separate Implementierung vorgeben. Der Compiler erkennt eine Spezialisierung an einer leeren Template-Parameterliste und dem genannten `<Typ>` nach dem Funktionsnamen:

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

template<typename T>
void tausche(T& a, T& b) {
  cout << "Template Funktion" << endl;
}

template<>
void tausche<bool>(bool& a, bool& b) {
  cout << "bool Funktion" << endl;
}

int main()
{
  int i,j;      tausche(i,j);
  double u,v;   tausche(u,v);
  bool a, b;    tausche(a,b);
}
```

```
Template Funktion
Template Funktion
bool Funktion
```

Programmstrukturierung
Funktions-Templates (generische Funktionen)

Einbinden von Templates:

- Templates sind Codeschablonen, die der Compiler zur Übersetzungszeit mit konkreten Datentypen übersetzen muss. Der Code eines Templates kann daher nicht wie eine normale *.cpp-Datei in ein einzelne *.o-Datei übersetzt werden, sondern muss dem Compiler über die Header-Datei zur Verfügung gestellt werden:

```
// Datei schablone.t
#ifndef SCHABLONE_T
#define SCHABLONE_T
// hier folgen die Template-Deklarationen
// ...
// ---- Implementierung -----
// hier folgen die Template-Definitionen
// ...
#endif
```

```
// Datei schablone.t
#ifndef SCHABLONE_T
#define SCHABLONE_T
// ---- Implementierung -----
// hier folgen direkt die Template-
// Definitionen ohne vorherige Deklaration
// ...
#endif
```

- Zur besseren Unterscheidung von normalen Header-Dateien kann man Template-Dateien eine andere Endung geben (z.B. *.t)

Programmstrukturierung
inline-Funktionen

- Funktionen können optional als `inline` deklariert werden:

```
inline int quadrat(int x) {
  return x * x;
}
```

- Dadurch wird dem Compiler empfohlen, keinen echten Funktionsaufruf auf Assemblerebene auszuführen, sondern den Funktionscode bei Aufruf der Funktion an die entsprechende Stelle 'einzusetzen'.

- `inline` sollte nur in Header-Dateien verwendet werden, damit die Tatsache, dass es die Funktion auf Linker-Ebene ggf. gar nicht gibt, auch an alle Benutzer der Header-Datei weitergegeben wird.

- Durch Verwendung von inline kann der Code ggf. schneller und größer werden.

Programmstrukturierung
Namensräume

- Benutzung von Namensräumen wurde schon besprochen.
- Um Deklarationen oder Funktionen in einen Namesraum zu legen, werden sie mit `namespace XXX { … }` umgeben:

```
// abc.h (Funktionen der ABC-GmbH)
namespace abc {
  int print(const char * );
  void func(double);
}
```

```
// xyz.h (Funktionen der XYZ-GmbH)
namespace xyz {
  int print(const char * );
  void func(double);
}
```

```
// main.cpp
#include "abc.h"
#include "xyz.h"
int main() {
  using namespace abc;
  print( "hello world !" ); //oder z.B.
  xyz::print( "hello world !" );
}
```

- Man kann Abkürzungen für Namesräume definieren:

```
namespace SpecialSoftwareGmbH_KlassenBibliothek {
  // ....
}
namespace sskb = SpecialSoftwareGmbH_KlassenBibliothek;
using namespace sskb; // Benutzung der Abkürzung
```

Programmstrukturierung
#include-Dateien

- Aus z.B. `#include <stddef.h>` in der Sprache C wird
  `#include <cstddef>` in C++, wobei sich der Inhalt im Namensraum `std` befindet!

  C-Header: cassert ,cctype ,cerrno ,cfloat ,ciso646 ,climits ,clocale, cmath, csetjmp, csignal, cstdarg, cstdbool, cstddef, cstdint, cstdio, cstdlib, cstring, ctime, cwchar, cwctype

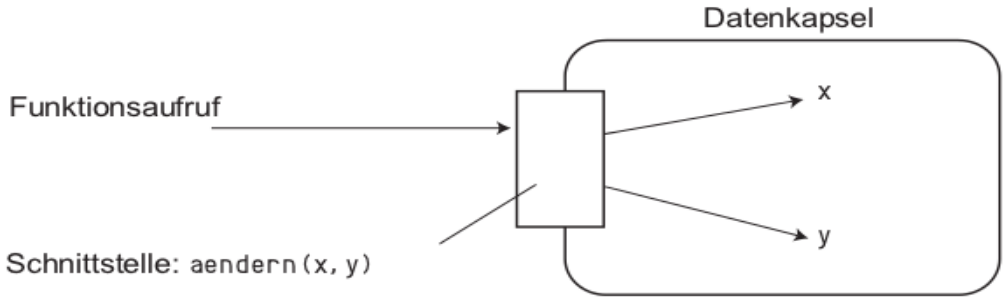
- Einbinden von C-Funktionen auf Linkerebene erfolgt mit `extern "C"`:

```
#ifdef __cplusplus
extern "C" {
#endif
 // .. hier folgen die C-Prototypen
 // oder z.B. eine C-Header-Datei
#ifdef __cplusplus
}
#endif
```

- Das Makro `__cplusplus` wird automatisch bei einem C++-Compiler gesetzt, und kann dazu benutzt werden, den Compilertyp herauszufinden (C oder C++).

Objektorientierung I
Abstrakter Datentyp (ADT)

- Ein Abstrakter Datentyp besteht aus:
  1. gekapselten Daten (**Attribute**) mit ihren jeweiligen Datentypen
  2. öffentlichen **Methoden** zur Veränderung der Daten



- In objektorientierten Sprachen wird ein ADT mit Hilfe ein Klasse realisiert:

```
//ort1.h
#ifndef ORT_H
#define ORT_H
class Ort {
  public:
    int getX() const;
    int getY() const;
    void aendern(int x, int y);  // x,y = neue Werte
  private:
    int xKoordinate;
    int yKoordinate;
};  // Semikolon nicht vergessen!
#endif // ORT_H
```

Objektorientierung I
Klassen und Objekte

- Eine Klasse ist selber ein eigenständiger Datentyp
- Konkrete Realisierungen einer Klasse nennt man Objekt oder Instanz der Klasse.
- Eine Klasse ist somit eine Abstraktion/Schablone für Objekte mit gleichen Eigenschaften
- Die Summe aller konkreten Werte der Eigenschaften eines Objektes nennt man auch den (inneren) Zustand des Objektes
- Die Schlüsselwörter `private` und `public` markieren die öffentlichen und nicht-öffentlichen Bereiche. Diese Markierungen können in einer Klasse mehrfach vorkommen und in beliebiger Reihenfolge angeordnet werden! Default ist `private` (bei structs `public`)
- `int foo() const;` bedeutet, dass die Methode den Zustand des Objekts nicht ändert (und auch nicht ändern darf!)
- Jedes Objekt besitzt eine eindeutige Identität (z.B. seine Adresse im Arbeitsspeicher), auch wenn es genau den gleichen Zustand wie ein anderes Objekt besitzt.

Objekterzeugung und -benutzung

- im einfachsten Fall wird eine Klasse benutzt wie ein eingebauter Datentyp (normale Variablendefinition, Variable auf Stack):

```
#include "ort.h" // Definition der Klasse einlesen
#include<iostream>
using namespace std;
int main() {     // Anwendung der Ort1-Klasse
  Ort einOrt;  // Objekt erzeugen
  einOrt.aendern(100, 200);
  cout << "Der Ort hat die Koordinaten x = "
       << einOrt.getX() << " und y = "
       << einOrt.getY() << endl;
}
```

- Zugriff auf die öffentlichen Methoden und Attribute mit dem `.`-Operator
- In Java können Objekte **nur** dynamisch angelegt werden, in C++ gibt es mehr Möglichkeiten: dynamisch, oder Speicherklasse `auto` [also auf dem Stack], oder auch globale Variable)
- Bei Objekterzeugung wird eine spezielle Methode, der sog. Konstrukor aufgerufen, der vom Benutzer definiert werden kann

Implementierung der `Ort1`-Klasse:

```
// Ort.cpp
#include "ort.h"
#include<iostream>
using namespace std;

int Ort::getX() const { return xKoordinate; }
int Ort::getY() const { return yKoordinate; }
void Ort::aendern(int x, int y) {
  xKoordinate = x;
  yKoordinate = y;
}
```

- Innerhalb der Methoden der Klasse darf auf die Attribute der Klasse (z.B. `xKoordinate`) zugegriffen werden
- Der Bereichsoperator `::` wird verwendet, um dem Compiler mitzuteilen, zu welcher Klasse die Methode gehört.
- Die Signatur der Methoden in der Implementations-Datei muss exakt der Signatur in der Header-Datei entsprechen (einschließlich `const` etc.)

`inline`-Elementfunktionen

- Kurze Funktionen (wie z.B. zum Lesen und Setzen von Attributen) sind häufig `inline`
- Da der Compiler den entsprechenden Code beim Aufruf der Methode 'einsetzt', muss er sich in der Header-Datei der Klasse befinden
- Dadurch wird allerdings die strikte Trennung von Schnittstelle und Implementierung aufgegeben!
- 2 Möglichkeiten:

Direkte Definition der Methode

```
// Ort.h
class Ort {
  public:
    int getX() const {
      return xKoordinate;
    }
    int getY() const {
      return yKoordinate;
    }
  private:
    int xKoordinate,
    yKoordinate;
};
```

Deklaration und Definition in der *.h-Datei

```
// Ort.h
class Ort {
  public:
    int getX() const;
    int getY() const;
  private:
    int xKoordinate,
    Int yKoordinate;
};

// ========= inline - Implementierung
inline int Ort::getX() const
{ return xKoordinate; }
inline int Ort::getY() const
{ return yKoordinate; }
```

Standardkonstruktor

- Für jede Klasse wird automatisch ein Standardkonstruktor angelegt, der *nichts* macht (auch z.B. keine Initialisierung von Attributen)
- Konstruktoren haben in der Deklaration KEINEN Rückgabetyp
- Eigene Definition des Standardkonstruktors:

```
// Ort.h
class Ort {
  public:
    Ort();
};
```

```
// Ort.cpp
Ort::Ort() {
  xKoordinate = 0;
  yKoordinate = 0;
}
```

- **Vorsicht:** Bei Instantiierung eines Objektes als `auto`-Variable dürfen (im Gegensatz zu allgemeinen Konstruktoren) keine Klammern angegeben werden:

```
int main() {
  Ort OrtObjekt1;    // OK !
  Ort OrtObjekt2(); // Fehler! Deklaration einer Funktion!!!
}
```

Allgemeine Konstruktoren

- Deklaration und Definition analog zum Standardkonstruktor, nur haben allgemeine Konstruktoren Parameter!

```
// Ort.h
class Ort {
  public:
    Ort(int i, int y);
};
```

```
// Ort.cpp
Ort::Ort(int x, int y) {
  xKoordinate = x;
  yKoordinate = y;
}
```

- Wenn ein allgemeiner Konstruktor definiert wurde, wird der automatisch erzeugte Standardkonstruktor gelöscht. Ein selber definierter Standard-Konstruktor funktioniert aber!
- Der Compiler sucht den am besten passenden Konstruktor aus. Bei Mehrdeutigkeiten gibt es eine Fehlermeldung.
- Standard-Parameter können auch bei Konstruktoren eingesetzt werden:

```
// Ort1.h
class Ort {
  public:
    Ort(int i, int y=100);
};
```

Initialisierungslisten

- Manchmal können Attribute nicht im Konstruktor-Code initialisiert werden:

```
// Klasse.h
class Klasse {
  public:
    Klasse(int);
  private:
    const int ic;
    int & ir;
};
```

```
// Klasse.cpp
Klasse::Klasse(int i) {
  ic = i; // FEHLER
  ir = i; // FEHLER
}
```

- Initialisierungslisten helfen in diesem Fall. Die Liste wird zwischen dem Ende der Parameterliste und dem Anweisungsblock angegeben:

```
// Klasse.cpp
Klasse::Klasse(int i) : ic(i), ir(i) {
}
```

- Der Code der Initialisierungsliste wird vor dem eigentlichen Konstruktor-Code ausgeführt.
- Die Reihenfolge der Initialisierungen richtet sich nach der Reihenfolge der Attribute in der Klassendeklaration, **NICHT** nach der Reihenfolge der Liste!

Klasse `Ort` im Überblick

```
// ort.h
#ifndef ORT_H
#define ORT_H
#include<iostream>

class Ort {
  public:
    Ort(int einX = 0, int einY = 0) : xKoordinate(einX), yKoordinate(einY) { }
    int getX() const { return xKoordinate;}
    int getY() const { return yKoordinate;}
    void aendern(int x, int y) {
      xKoordinate = x;
      yKoordinate = y;
    }
  private:
    int xKoordinate;
    int yKoordinate;
};

inline void anzeigen(const Ort& o) { // globale Funktion
  std::cout << ' ( ' << o.getX() << " , " << o.getY() << ' ) ' ;
}
#endif
// ORT_H
```

Kopierkonstruktor

- Soll eine Kopie eines bereits existierenden Objektes erzeugen. Dabei wird das Quell-Objekt als (konstante) Referenz übergeben:

```
// Ort.h
class Ort {
  public:
    ...
    Ort(const Ort &);
};
```

```
// Ort.cpp
Ort::Ort(const Ort & einOrt) {
  xKoordinate = einOrt.xKoordinate;
  yKoordinate = einOrt.yKoordinate;
}
```

- Der Compiler erzeugt für jede Klasse automatisch einen Kopierkonstruktor, der alle Attribute bitweise kopiert!
- Bei eigener Definition des Kopierkonstruktors werden standardmäßig **keine** Attribute kopiert! Der Benutzer muss das selber programmieren!
- Benutzung des Kopierkonstruktors:

```
Ort einOrt(19, 39);
Ort derZweiteOrt = einOrt;      // Aufruf des Kopierkonstruktors
Ort derDritteOrt(einOrt);       // Aufruf des Kopierkonstruktors
```

- Die zweite Zeile in obigem Code ist keine Zuweisung, sondern auch der Aufruf eines Kopierkonstruktors!

Welche Konstruktoren werden hier aufgerufen?

```
Ort o1, o2;
o1 = Ort(8,7);
Ort o4 = o1;
o2 = o1;
Ort o3 = Ort(1, 17);
```

- **Merke**: Die Übergabe von Objekten an eine Funktion per Wert und die Rückgabe eines Ergebnisobjekts wird ebenfalls als Initialisierung betrachtet, ruft also den **Kopierkonstruktor** implizit auf!

```
// in Ort.cpp
…
Ort::Ort(const Ort &) { cout << "copy CTOR" << endl; }
…
Ort ortsverschiebung(Ort derOrt, int dx, int dy) {
  derOrt.aendern(derOrt.getX() + dx, derOrt.getY() + dy);
  return derOrt; // Rückgabe des veränderten Orts
}
```

```
int main() {
  Ort einOrt(10, 300);
  Ort verschobenerOrt = ortsverschiebung(einOrt, 10, -90);
}
```

```
copy CTOR
copy CTOR
```

Typumwandlungskonstruktor

- Dient zur automatischen Umwandlung eines anderen Datentyps in den 'eigenen' Datentyp (in unserem Fall `Ort`)
- Dieser Konstruktor erhält daher nur den umzuwandelnden Datentyp als Parameter, und ist ein Spezialfall eines allgemeinen Konstruktors!

```
// Ort.h
class Ort {
  public:
    Ort(const std::string & s);
};
```

```
// Ort.cpp
Ort::Ort(const std::string & s) {
  // Code zur Umwandlung des Strings
  // in die beiden Koordinaten
}
```

- Dadurch sind dann z.B. folgende Benutzungen möglich:

```
Ort nochEinOrt(string( "21 99" )); // möglich
Ort einWeitererOrt( "(55 , 8)" );  // ebenfalls OK
string wo( "20,400" );
Ort hier = ortsverschiebung(wo, 10, -90);
Ort dort = ortsverschiebung(Ort(wo), 10, -90); // besser
```

Implizite Typumwandlung: Der Compiler erzeugt ein temporäres Ort-Objekt,und übergibt es der Methode!

Explizite Typumwandlung (besser): Der Benutzer ruft den entsprechenden Typumwandlungskonstruktor selber auf!

Schlüsselwort `explicit`

- Manchmal möchte man implizite Typ-Umwandlungen nicht zulassen damit keine Fehler oder Fehlbenutzungen entstehen
- Wenn der Konstruktor zusätzlich als `explicit` deklariert wird, sind explizite Typ-Umwandlungen weiterhin erlaubt, aber implizite Typ-Umwandlungen verboten!

```
// Ort.h
class Ort {
  public:
    explicit Ort(const std::string & s);
};
```

- Das Verhalten ändert sich dann entsprechend:

```
Ort o1;
string wo( "10, 200" );
o1 = wo;       // jetzt ein Fehler! Implizite Typumwandlung!
o1 = Ort(wo); // erlaubte explizite Typumwandlung
```

Beispiel: Klasse für Rationale Zahlen (Brüche)

```
// rational.h
#ifndef RATIONAL_H
#define RATIONAL_H
class Rational {
  public:
    Rational();
    Rational(long z, long n); // allgemeiner Konstruktor
    Rational(long);           // Typumwandlungskonstruktor

    // Abfragen
    long getZaehler() const;
    long getNenner() const;

    // arithmetische Methoden für +=, -=, *=, /=, später überladene Operatoren ...
    void add (const Rational& r);
    void sub (const Rational& r);
    void mult(const Rational& r);
    void div (const Rational& r);

    // weitere Methoden
    void set(long zaehler, long nenner);
    void eingabe();
    void ausgabe() const;
    void kehrwert();
    ...
```

```
    ...
    void kuerzen();

  private:
    long zaehler, nenner;
};

// inline-Methoden
inline Rational::Rational()
 : zaehler(0), nenner(1) {}
inline Rational::Rational(long z, long n)
 : zaehler(z), nenner(n) {}
inline Rational::Rational(long ganzeZahl)
 : zaehler(ganzeZahl), nenner(1) {}

inline long Rational::getZaehler() const {return zaehler;}
inline long Rational::getNenner()  const {return nenner;}

// globale Funktionsprototypen
const Rational add (const Rational& a, const Rational& b);
const Rational sub (const Rational& a, const Rational& b);
const Rational mult(const Rational& a, const Rational& b);
const Rational div (const Rational& z, const Rational& n);

#endif
```

Objektorientierung I
Klassen und Objekte

```
// rational.cpp (Definition der Methoden und globalen Funktionen)
#include "rational.h"
#include<iostream>
#include<cassert>
using namespace std;

void Rational::set(long z, long n) {
  zaehler = z;
  nenner = n;
  assert(nenner != 0);
  kuerzen();
}

void Rational::eingabe() {
  // Bildschirmausgabe nur zu Demonstrationszwecken.
  cout << "Zähler : " ;
  cin >> zaehler;
  cout << "Nenner : " ;
  cin >> nenner;
  assert(nenner != 0);
  kuerzen();
}

void Rational::ausgabe() const {
  cout << zaehler << '/' << nenner << endl;
}
...
```

Objektorientierung I
Klassen und Objekte

```
..
void Rational::kehrwert() {
  long temp = zaehler;
  zaehler = nenner;
  nenner = temp;
  assert(nenner != 0);
}

void Rational::add(const Rational& r) {
  zaehler = zaehler * r.nenner + r.zaehler * nenner;
  nenner  = nenner  * r.nenner;
  kuerzen();
}
void Rational::sub(const Rational& s) {
  Rational r = s;
  // Das temporäre Objekt r wird benötigt, weil s wegen der const-Eigenschaft nicht
  // verändert werden kann (und darf). Eine Alternative wäre die Übergabe per Wert
  // - dann könnte mit der lokalen Kopie gearbeitet werden. Die Subtraktion kann
  // durch Addition des negativen Arguments erreicht werden.
  r.zaehler *= -1;
  add(r);
}
void Rational::mult(const Rational& r) {
  zaehler = zaehler * r.zaehler;
  nenner  = nenner  * r.nenner;
  kuerzen();
}
...
```

Objektorientierung I
Klassen und Objekte

```
…

void Rational::div(const Rational& n) {
  Rational r = n;  // Siehe sub()
  // Division = Multiplikation mit dem Kehrwert
  r.kehrwert();
  mult(r);
}

// Die globale Funktion ggt() wird zum Kürzen benötigt. Sie berechnet den größten
// gemeinsamen Teiler. Verwendet wird ein modifizierter Euklid-Algorithmus, in dem
// Subtraktionen durch die schnellere Restbildung ersetzt werden.
long ggt(long x, long y) {
  long rest;
  while(y > 0) {
    rest = x % y;
    x = y;
    y = rest;
  }
  return x;
}

void Rational::kuerzen() {
  long teiler = ggt( abs(zaehler), abs(nenner));
  zaehler = zaehler/teiler;
  nenner  = nenner /teiler;
}
```

Objektorientierung I
Klassen und Objekte

```
…
// Es folgen die globalen arithmetische Funktionen für die
// Operationen mit 2 Argumenten (binäre Operationen)
// Dabei werden die schon existierenden unären Operationen wiederverwendet!

const Rational add(const Rational& a, const Rational& b) {
  Rational r = a;
  r.add(b);
  return r;
}

const Rational sub(const Rational& a, const Rational& b) {
  Rational r = a;
  r.sub(b);
  return r;
}

const Rational mult(const Rational& a, const Rational& b) {
  Rational r = a;
  r.mult(b);
  return r;
}

const Rational div(const Rational& z, const Rational& n) {
  Rational r = z;
  r.div(n);
  return r;
}
```

Beispielhafte Anwendung der Klasse

```
#include "rational.h"

int main() {

  Rational bruch1(2,4);
  Rational bruch2 = 10;

  Rational bruch3 = add(bruch1, bruch2);
  bruch3.sub(4);

  bruch3.ausgabe();
}
```

- Die Verwendung des *Typumwandlungskonstruktors* vereinfacht die Klasse, indem er die benötigte Methodenzahl verringert!

`const`-Objekte und Methoden

- Objekte können wie einfache Variablen als *konstant* deklariert werden!
- Nur der Konstruktor und Destruktor solcher Objekte darf aufgerufen werden, und Methoden, die als *konstant* deklariert und definiert wurden!

```
// Ort.h
class Ort {
  public:
    ausgabe1();
    ausgabe2() const;
};
```

```
int main() {
  const Ort my_ort;
  my_ort.ausgabe1();   // FEHLER
  my_ort.ausgabe2();   // OK!
}
```

- 'Konstante Methode' bedeutet in diesem Zusammenhang, dass die Methode keine Attribute des Objektes verändern darf. Der Compiler überprüft diese Bedingung!
- 'Konstante Methoden' sind natürlich auch auf nicht-`const` Objekte aufrufbar!
- Die Konstantheit der Methode ist bei der Überladung von Methoden ein Unterscheidungskriterium! Je nach Objekttyp (konstant oder nicht) würde dann eine andere Methode aufgerufen!

Destruktoren

- Destruktoren werden beim Löschen von Objekten aufgerufen.
- Deklaration des Destruktors wie die des Standard-Konstruktors, aber mit einer Tilde ~ vor dem Namen.
- Der Compiler erzeugt für jede Klasse automatisch einen Destruktor, der keine Funktionalität besitzt. Dieser kann durch eine eigene Definition ersetzt werden.
- Destruktoren sind sinnvoll, wenn eine Klasse noch ggf. Ressourcen (Dateien, dynamischen Speicher) benutzt, der vor der Zerstörung des Objektes noch freigegeben werden muss.
- Die Destruktoren werden in umgekehrter Reihenfolge aufgerufen wie die Konstruktoren.
- Merke: Bei globalen Objekten wird der Konstruktor noch vor der `main`-Methode aufgerufen, der Destruktor erst nach Zerstörung aller in `main` verwendeten Objekte

Objektorientierung I
Klassen und Objekte

```
#include<iostream>
using namespace std;
class Beispiel {
  int zahl;       // zur Identifizierung
  public:
    Beispiel(int i = 0); // Konstruktor
   ~Beispiel();          // Destruktor
};

Beispiel::Beispiel(int i) : zahl(i) {
  cout << "Objekt " << zahl << " wird erzeugt." << endl;
}
Beispiel::~Beispiel() {
  cout << " Objekt " << zahl << " wird zerstört." << endl;
}
// globale Variable, durch Vorgabewert mit 0 initialisiert
Beispiel ein_globales_Beispiel;

int main() {
  cout << "main wird begonnen \n" ;
  Beispiel einBeispiel(1);
  {
    cout << "neuer Block\n";
    Beispiel einBeispiel(2);
    cout << "Block wird verlassen \n";
  }
  cout << "main wird verlassen \n" ;
}
```

```
Objekt 0 wird erzeugt.
main wird begonnen
Objekt 1 wird erzeugt.
neuer Block
Objekt 2 wird erzeugt.
Block wird verlassen
Objekt 2 wird zerstört.
main wird verlassen
Objekt 1 wird zerstört.
Objekt 0 wird zerstört.
```

Gegenseitige Abhängigkeit von Klassen

- Wenn Klassen sich gegenseitig benötigen, kann es zu Problemen kommen:

```
// Datei A.h
#ifndef A_h
#define A_h
#include"B.h"
class A {
  void foo(B & b);
};
#endif
```

```
// Datei B.h
#ifndef B_h
#define B_h
#include"A.h"
class B {
  void foo(A & a);
};
#endif
```

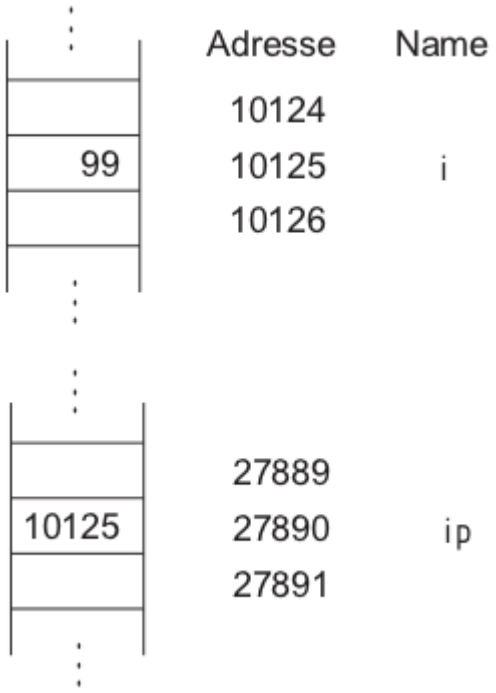
- Lösung: Wenn in der Header-Datei einer Klasse B die Klasse A nur als Referenz oder Zeiger benutzt wird, reicht es, die Klasse als **Vorwärtsdeklaration** bekannt zu machen:

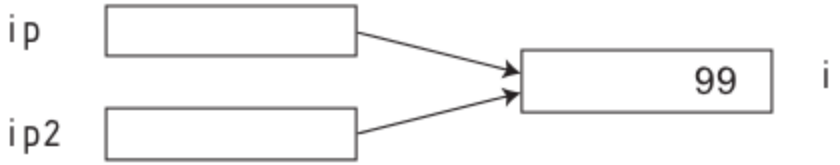
```
// Datei A.h
#ifndef A_h
#define A_h

class B;
class A {
  void foo(B & b);
};
#endif
```

```
// Datei B.h
#ifndef B_h
#define B_h

class A;
class B {
  void foo(A & a);
};
#endif
```

Zeiger
Zeiger und Adressen

- Zeiger sind die Grundlage für dynamisch angelegte Objekte, C-Arrays und C-Strings
- Zeiger erlauben große Freiheiten, haben aber auch ihre Tücken!
- Zeiger sind grundsätzlich wie int-Variablen, nur ihr Inhalt wird als Adresse behandelt.






```
int i = 99;
int *ip  = &i;
int *ip2 = ip;

 i   = 99; // alle 3 Anweisungen setzen i!
*ip  = 99;
*ip2 = 99;
```

- * in Deklaration: Deklaration als Zeiger
- * vor Zeigern: Dereferenzierung, also das, worauf der Zeiger zeigt
- & vor Ausdrücken: *address-of*-Operator

Zeiger
Zeiger und Adressen

- Zeiger müssen nicht initialisiert werden, und enthalten ohne Zuweisung keinen definierten Wert.
- Vorsicht: Bei `int *ip, x;` ist x vom Typ int, also kein Zeiger! Der Stern bezieht sich nur auf den folgenden Namen!
- Der spezielle Zeigerwert `0` zeigt auf 'nichts'. In C wird der Wert durch `NULL` dargestellt, was aber zu Problemen aufgrund unterschiedlicher Definitionen von `NULL` führte. In C++11 gibt es ein neues Schlüsselwort dafür: `nullptr`
- Zeiger auf Referenzen sind nicht möglich!
- Typprüfung: Der Typ eines Zeigers wird in C++ überprüft! Typumwandlungen mit `static_cast` sollten nicht ohne wichtigen Grund verwendet werden (Typkontrolle des Compilers wird umgangen)!

```
char *cp;  // Zeiger auf char
void *vp;  // Zeiger auf void ('nichts')
vp = cp;   // OK
cp = vp;   // Fehlermeldung des Compilers
cp = static_cast<char*>(vp); // OK, s.o.
```

Zeiger
Zeiger und Adressen

- Bei Zeigern muss die Lebensdauer der Objekte, auf die gezeigt wird, beachtet werden! Der Zugriff auf nicht mehr existierende Objekte kann zu falschen Werten und zum Systemabsturz führen!

```
int i = 9;
int * ip = &i; // Zeiger ip zeigt auf i
*ip = 8;       // i erhält den Wert 8
{   // neuer Block beginnt
    int j = 7;
    ip = &j;  // Zeiger ip zeigt auf j
    // weiterer Programmcode
}   // Blockende, j wird ungültig
*ip = 8;      // gefährlich!
```

- Zeiger oder auch das, worauf sie zeigen, können konstant sein:

```
int *ip;  // Nichts konstant
const int *ip;  // Zeiger auf konstanten int
int const *ip;  // Zeiger auf konstanten int
int * const ip; // konstanter Zeiger auf int
int const * const ip; // konstanter Zeiger auf konstanten int
```

- `const` wirkt immer auf das, was links davon steht (von rechts nach links zu lesen). Nur am Anfang kann `const` auch vorangestellt werden.

Zeiger
C-Arrays

- C-Arrays sind ein- oder mehrdimensionale Felder eines Datentyps mit fester Größe. Bei der Deklaration wird die Größe in eckigen Klammern angegeben:

```
const int ANZAHL = 5;
int Tabelle[ANZAHL];
int tabelle2[20];
```

- Die Größenangabe muss zur Übersetzungszeit bekannt sein.
- Bei mehreren Dimensionen werden mehrere Klammerpaare angegeben.
- Der Zugriff auf die Elemente erfolgt wie beim `vector` mit dem Index-operator `[ ]` oder durch Zeigerarithmetik. Der Index läuft von `0` bis `N-1`.
- Der Name des Arrays ist ein konstanter Zeiger auf das erste Element, und ist daher ein R-Wert (kann nicht auf der linken Seite von Zuweisungen stehen). Ansonsten ist der Name aber wie ein normaler Zeiger zu benutzen.

Zeiger
C-Arrays

- Der Compiler übersetzt den Zugriff über den Index-Operator in die entsprechende Zeigerarithmetik:

```
char a[10];
a[5] = '?';  // Setzte 6. Element
im Array
// wird intern übersetzt in:
*(a+5) = '?';
```

- Der Zugriff über den Index-Operator wird nicht auf seine Grenzen hin überprüft!
- Beim Erhöhen eines Zeigers um 1 wird nicht auf die nächste Speicheradresse (das nächste Byte) gezeigt, sondern auf das nächste Element im Array:

```
int main()
{
  double d[10];
  double *dp1 = d;   // Zeigt auf 1. Element
  double *dp2 = d+2; // Zeigt auf 3. Element
  cout << (dp2-dp1) << " " << dp1 << " " << dp2 << endl;
  cout << ((unsigned long)(dp2) - (unsigned long)(dp1)) << endl;
}
```

```
2 0xbfc8d598 0xbfc8d5a8
16
```

Zeiger
C-Arrays

- Für den Namen eines Arrays braucht der Compiler (wie für andere Konstanten) keinen Speicherplatz zu reservieren, sondern der Wert wird bei Benutzung an die entsprechende Stelle eingesetzt.
- Die Größe eines Arrays kann mit **`sizeof()`** bestimmt werden. Dabei wird die Größe des Parameters *in Bytes* zurückgegeben. Bei der Übergabe eines Arrays als Funktionsparameter geht diese Information verloren, da das übergebene Array nur als Zeiger interpretiert wird:

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

void func(double * dp) {
  cout << "Größe in Funktion: " << sizeof(dp) << endl;
}

int main()
{
  double d[10];
  cout << "Größe nach Definition: " << sizeof(d) << endl;
  func(d);
}
```

```
Größe nach Definition: 80
Größe in Funktion: 4
```

Zeiger
C-Arrays

- Arrays können bei der Definition auch initialisiert werden. Die Größenangabe kann dann optional weggelassen werden, da der Compiler in diesem Fall die Größe automatisch bestimmt. Falls die Größe angegeben wird, und weniger Elemente in der Initialisierungs-liste angegeben werden, werden die restlichen Werte mit 0 initialisiert:

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

int main()
{
  int feld1[] = { 1, 3, 5 };
  cout << sizeof(feld1)/sizeof(int) << endl;

  int feld2[5] = { 1, 3, 5 };
  // entspricht { 1, 3, 5, 0, 0}
  cout << sizeof(feld2)/sizeof(int) << endl;
}
```

```
3
5
```

Zeiger
C-Zeichenketten

- Eine C-Zeichenketten ist ein Spezialfall eines Arrays: Ein Array aus Elementen vom Typ `char`, das mit `0` (Zeichenliteral `'\0'`) abgeschlossen ist.
- Bei Literalen von C-Zeichenketten (z.B. `"Hallo"`) wird `'\0'` am Ende automatisch hinzugefügt.
- Bei `cout` gibt der Ausgabeoperator für `char*` und `const char*` die entsprechende Zeichenkette bis `'\0'` aus, nicht den Zeigerwert:

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

int main()
{
  char feld1[] = "Hallo MATSE1";       // 12 Buchstaben + '\0'
  char feld2[3]= "Hallo";              // Fehler! Kein Platz!
  char feld3[3]= { 'a','b','c' };      // Ohne '\0'
  const char *feld4  = "Hallo MATSE2"; // Ohne const Warnung!

  cout << feld1 << " " << sizeof(feld1) << endl;
  cout << feld3 << " " << sizeof(feld3) << endl;
  cout << feld4 << " " << sizeof(feld4) << endl;
}
```

```
Hallo MATSE1 13
abcHallo MATSE1 3
Hallo MATSE2 4
```

Zeiger
C-Zeichenketten

- Die Initialisierung einer C-Zeichenkette kann auch über mehrere Zeilen gehen:

```
char nachricht[] = "Das ist ein kleines\n"
                   "Beispiel für ein langes\n"
                   "C-String Literal :)\n";
```

- In der Header-Datei `<cstring>` gibt es viele vordefinierte Funktionen für C-Zeichenketten, z.B. `strlen()`.
- Bei der Eingabe einer C-Zeichenkette muss Speicher vorhanden sein:

```
char *x;
cin >> x;  // Fehler! Zeiger nicht initialisiert, und zeigt
           // irgendwo hin...
x = "Hallo";
cin >> x;  // Fehler! Daten liegen i.d.R. Im schreibgeschützten Bereich!

char bereich[100];
cin >> bereich;  // OK, ignoriert leading whitespaces, und liest bis zum
                 // ersten Zwischenraum. Größe immer noch Risiko!

const int ZMAX=100;
char zeile[ZMAX];
cin.getline(zeile, ZMAX);  // OK !!
```

Beispiele für C-String Algorithmen

Hier nur zur Veranschaulichung und Darstellung von schwer lesbarem Code :)

- Stringlänge berechnen:

```
char nachricht[] = "Hallo MATSE";
const char * cp = nachricht;

int len;
while (*cp++) ;
len = cp-nachricht-1; // len ist 11 (ohne '\0')
```

- Strings kopieren:

```
char nachricht[] = "Hallo MATSE";
char kopie[100];  // Platz muss reichen!
const char * source_p = nachricht;
char * dest_p = kopie;

while (*dest_p++ = *source_p++) ;
```

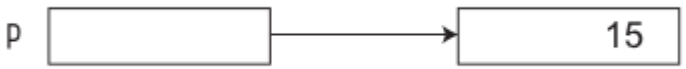
- Die Funktionen `strlen` und `strcpy` aus `<cstring>` leisten die gleichen Funktionalität, und sollten benutzt werden!

Zeiger
Dynamische Datenobjekte

Dynamisches Anlegen von Objekten

- Mit den Operatoren `new` und `delete` kann Speicher vom Heap alloziert und freigegeben werden.
- Damit unterliegen diese Objekte *nicht* den Gültigkeitsbereichregeln für Variablen.
- `new` erkennt die benötigte Menge Speicher automatisch am Datentyp
- Der Zugriff auf dynamisch erzeugte Objekte geschieht ausschließlich über Zeiger (`new` liefert einen entsprechenden Zeiger zurück)!
- `malloc` und `free` sollten in C++-Programmen nicht verwendet werden!

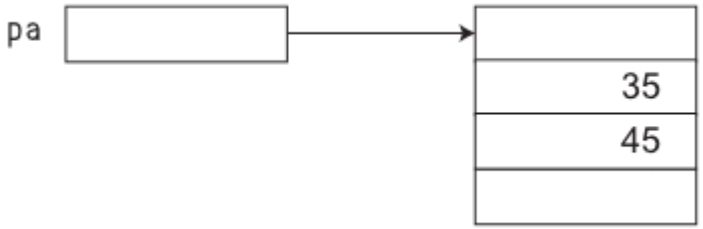
```
int * p;      // Zeiger auf int
p = new int; // int-Objekt erzeugen
*p = 15;     // Wert zuweisen
cout << * p << endl;   // 15
```



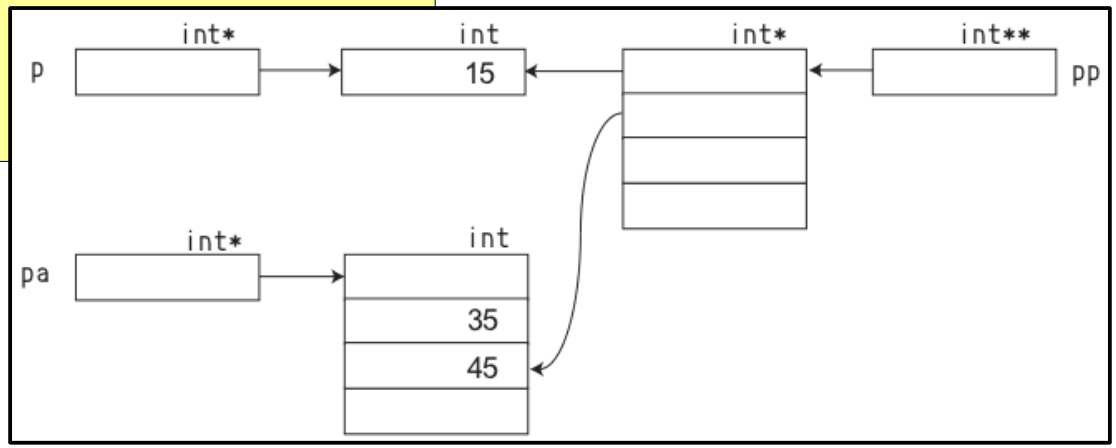
Zeiger
Dynamische Datenobjekte

- Mit `new` können auch Arrays alloziert werden!
- Der Zugriff auf die Elemente geschieht wie bei statisch deklarierten Arrays.

```
// Operator new [ ]
int *pa = new int[4]; // Array von int-Zahlen
pa[1] = 35;
pa[2] = 45;
cout << pa[1] << endl;  // 35
```



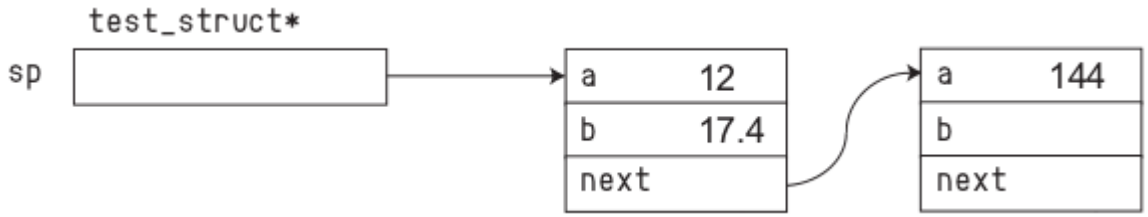

```
int **pp = new *int[4];  Array von 4 Zeigern auf int
pp[0] = p;
pp[1] = &pa[2];
cout << *pp[0] << endl;  // 15
cout << **pp   << endl;  // 15
cout << *pp[1] << endl;  // 45
```



Zeiger
Dynamische Datenobjekte

- Mit `new` können natürlich auch neue Instanzen von Strukturen und Klassen erzeugt werden!
- Der Zugriff auf Elemente des Objektes mit Hilfe eines Zeigers auf das Objekt geschieht mit dem **–>** Operator!

```
struct test_struct {
  int a;
  double b;
  test_struct * next;
};

test_struct *sp= new test_struct; // Ein dynamisches Objekt erzeugen
sp->a = 12;     // entspricht (*sp).a
sp->b = 17.4;
// ein weiteres Objekt erzeugen
sp->next = new test_struct;
// Zugriff auf Element a des neuen Objekts
sp->next->a = 144;
```



Dynamisches Löschen von Objekten

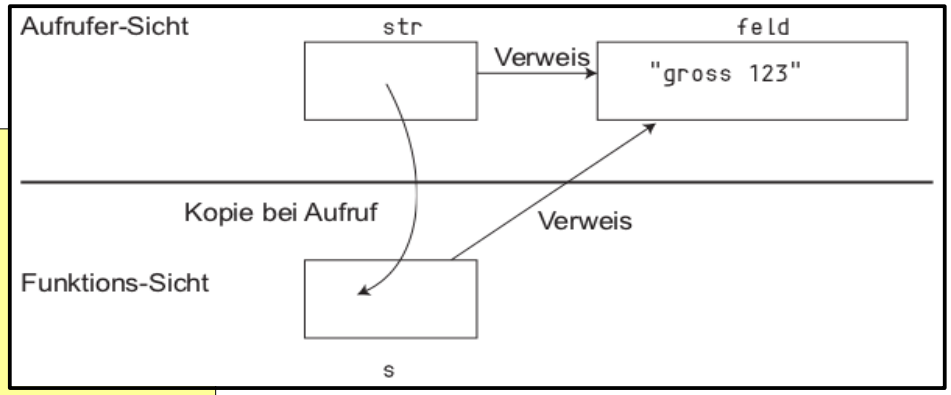
- Mit `delete` wird der vorher bereitgestellte Speicher wieder freigegeben. Als Parameter wird der Zeiger auf das Objekt angegeben.

```
// Lösche die Objekte aus den Beispielen zuvor
delete p;
delete [] pa;
delete [] pp;
// Reihenfolge beachten!
delete sp->next;
delete sp;
```

- Wenn zuvor ein Array mit `new …[]` alloziert wurde, muss es hinterher mit `delete []` gelöscht werden!
- delete darf nur einmal aufgerufen werden!
- Ein Aufruf von `delete` mit einem NULL-Zeiger ist unschädlich!
- Beim Löschen eines Objektes wird automatisch der Destruktor aufgerufen.
- Fehlende oder falsche delete-Anweisungen sind Ursache für Speicherlecks (engl. *memory leaks*)

Zeiger als Funktionsparameter

- Zeiger können genau wie andere Variablen *per Wert* oder *per Referenz* übergeben werden.



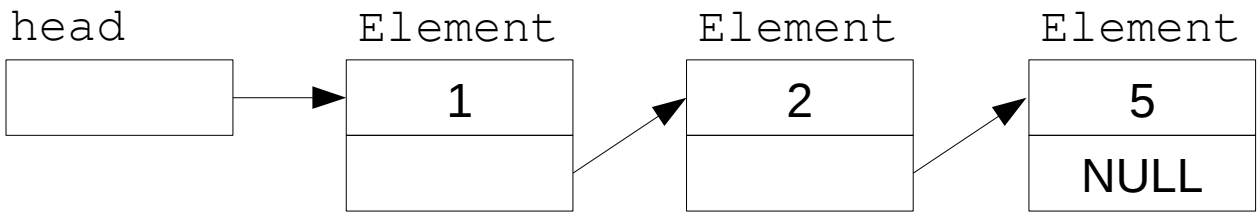
Übergabe *per Wert*:

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

void upcase(char *); // Prototyp

int main( ) {
  char str[] = "gross 123";
  upcase(str);
  cout << str << endl; // GROSS 123
}

void upcase(char *s) {
  const int DIFFERENZ = 'a'-'A';
  while(*s) {
    if(*s >= 'a' && *s <= 'z') {
       *s -= DIFFERENZ;
    }
    s++;
  }
}
```

Der Zeiger wird bei der Übergabe per Wert zwar kopiert, zeigt aber auf dasselbe Feld bzw. Objekt!

Zeiger
Zeiger und Funktionen

- Zeiger können natürlich auch als *Referenz* übegeben werden: Dann kann die aufgerufenen Methode nicht nur das Objekt, worauf der Zeiger zeigt, verändern, sondern auch den Wert des Zeigers selber, der beim Funktionsaufruf als Parameter übergeben wird.
- Beispiel: Einfach verkette Liste. Wie kann man möglichst einfach neue Elemente hinzufügen und löschen?
- Die Liste soll die Elemente (jedes Element hat als 'value' einen Integer-Wert) automatisch *der Größe nach* aufsteigend sortieren.
- Die Funktion zum Löschen soll *alle* Elemente mit einem bestimmen Wert aus der Liste löschen.



Zeiger
Zeiger und Funktionen

Übergabe per Referenz (am Beispiel einer einfach verketteten Liste):

```
struct Element {
  int value;
  Element *next;
  Element(int v, Element *n)
   : value(v), next(n) { }
};

void insert(Element* &rp, int v) {
  if (rp != NULL) {
    if (v > rp->value)      // Sortieren
      insert(rp->next, v); // Rekursion
    else rp=new Element(v, rp);
  }
  else rp=new Element(v, NULL);
}

void remove(Element* &rp, int v) {
  if (rp != NULL) {
    if (rp->value == v) {
      Element *tmp=rp;
      rp = rp->next;
      delete tmp;
      remove(rp, v);
    }
    else remove(rp->next, v);// Rekursion
  }
}
```

```
void print(Element *p) {
  while(p) {
    std::cout << p->value << " ";
    p=p->next;
  }
  std::cout << std::endl;
}

int main(void) {

  Element * head = NULL;
  insert(head, 2);
  insert(head, 5);
  insert(head, 1);
  print(head);

  remove(head, 2);
  print(head);
}
```

```
1 2 5
1 5
```

Zeiger
Parameter der `main()`-Funktion

- Die `main()`-Funktion kann, wie schon gesehen, die Kommandozeilen-parameter auswerten.
- Neben der Form

  ```
  int main(int argc, char* argv[])
  ```

  gibt es auch die Form

  ```
  int main(int argc, char* argv[], char * env[])
  ```

  die auch die Umbegungsvariablen des Betriebssystems auswerten kann.
- `argv` enthält als ersten Wert des Namen des Programms, und es gilt: `argv[argc] == 0`, und am Ende von `env` gilt: `env[] == 0`

```
int main(int argc, char * argv[], char * env[]) {
  int i = 0;
  while(argv[i]) {
    cout << argv[i++] << endl;
  }
  cout << "\n*** Umgebungs-Variablen : ***\n" ;
  i = 0;
  while(env[i]) {
    cout << env[i++] << endl;
  }
}
```

```
$ ./a.out Das sind die Parameter
./a.out
Das
sind
die
Parameter

*** Umgebungs-Variablen : ***
ORBIT_SOCKETDIR=/tmp/orbit-aterstegge
SSH_AGENT_PID=1687
...
```

Zeiger
Rückgabe von Zeigern

- Genau wie bei Referenzen muss der Programmierer sicherstellen, dass bei der Rückgabe von Zeigern die Objekte auch entsprechend lange 'leben'.

Negativ-Beispiel (Funktion zum Kopieren eines Strings):

```
char *cstr_copy1(const char *text) {
  char neu[100]; // Speicherplatz besorgen
  char *n = neu;
  while(*n++ = *text++); // Daten nach neu kopieren
  return neu;     // Fehler!
}
```

Positiv-Beispiel:

```
char *cstr_copy(const char *text) {
  char *neu = new char[strlen(text) + 1];
  char *n = neu;
  while(*n++ = *text++); // Daten nach neu kopieren
  return neu;
} // Der Aufrufer muss für die Speicherfreigabe sorgen!
```

- Was passiert, wenn ein mit **new** erzeugtes Objekt per Wert zurückgegeben wird?

Zeiger
Mehrdimensionale statische C-Arrays

- Es können auch mehrdimensionale Arrays (hier zunächst statisch) angelegt werden. Jede Dimension erhält dann ein eigenes Klammerpaar:

```
int main() {
  const size_t DIM1 = 2;
  const size_t DIM2 = 3;
  int matrix [DIM1][DIM2] = { {1,2,3}, {4,5,6} };
  for (size_t i = 0; i < DIM1; ++i) {
    for (size_t j = 0; j < DIM2; ++j) {
      cout << matrix[i][j] << ' ' ;
    }
    cout << endl;
  }
}
```

```
1 2 3
4 5 6
```

- Im obigen Beispiel hat die Matrix 2 Zeilen und 3 Spalten.
- Die Initialisierungsliste kann auch eindimensional geschrieben werden: `{1,2,3,4,5,6}`. Die Werte werden Zeile für Zeile initialisiert.
- Bei statischen Arrays müssen die Größenangaben zur Übersetzungszeit bekannt sein.
- Vorsicht: `matrix[2,3]` entspricht `matrix[3]`, NICHT `matrix[2][3]`

Zeiger
Mehrdimensionale statische C-Arrays

- Bei der Übergabe von mehrdimensionalen C-Arrays als Funktionsparameter muss (wie vorher schon gesehen) die Länge der ersten Dimension zusätzlich als Parameter übergeben werden. Die anderen Dimensionen werden dem **Datentyp** zugeordnet (!), und der Compiler überprüft die entsprechende Zeigerkompatibilität:

```
void ausgabe1(int *m, int len) { }

void ausgabe2(int (*m)[3], int len) {
  cout << sizeof(m) << endl << sizeof(m[0]) << endl
       << sizeof(m[0][0]) << endl;
  for (size_t i = 0; i < len; ++i) {
    for (size_t j = 0; j < sizeof(m[0]) /
                           sizeof(m[0][0]); ++j) {
      cout << m[i][j] << ' ';
    }
    cout << endl;
  }
}
int main() {
  const size_t DIM1 = 2, DIM2 = 3;
  int matrix[DIM1][DIM2] = {{1,2,3},{4,5,6}};
  ausgabe1(matrix, DIM1);  // FEHLER!! Falscher Datentyp!
  ausgabe2(matrix, DIM1);
}
```

```
4
12
4
1 2 3
4 5 6
```

Zeiger
Mehrdimensionale statische C-Arrays

- Wie kann man beliebige 2-dimensionale statische Arrays ausgeben?
  → Lösung mit entsprechendem Funktions-Template:

```
template<typename TYP>
void ausgabe2D(TYP tabelle, size_t n) {
  const size_t SPALTEN = sizeof tabelle[0] /
                         sizeof tabelle[0][0];
  for(size_t i = 0; i < n;++i) {
    for(size_t j = 0; j < SPALTEN;++j) {
      cout << tabelle[i][j] << ' ';
    }
    cout << endl;
  }
}

int main() {
  int matrix1[2][3] = {{1,2,3},{4,5,6}};
  ausgabe2D(matrix1, 2); // Erste Dimension muss immer
                         // angegeben werden!!
}
```

```
1 2 3
4 5 6
```

- Auch bei mehreren Dimensionen wird der Array-Zugriff in Zeiger-Arithmetik umgesetzt: **`m[i][j]` → `*(m[i] + j)` → `*(*(m+i) + j)`**

Zeiger
Mehrdimensionale dynamische C-Arrays

- Dynamische mehrdimensionale Arrays können auf 2 verschiedene Arten angelegt werden:

1. Arrays mit fester Feldgröße:

- **new** liefert einen konstanten Zeiger auf das erste Element zurück, nur der Zeigertyp muss entsprechend angepasst werden:

```
int * const a1 = new int[10];           // a1 ist Zeiger auf das erste Element
int (* const a2)[5] = new int[10][5];   // Dim. 5 muss auf beiden Seiten identisch sein!
int (* const a3)[5][2] = new int [20][5][2];  // u.s.w.
```

- `const` kann auch weggelassen werden, sollte aber aus Sicherheitsgründen nicht fehlen (damit der Zeiger nicht überschrieben wird!).
- Alle Dimensionen außer der ersten gehen in den Datentyp ein!
- `sizeof()` funktioniert nicht bei der ersten Dimension:

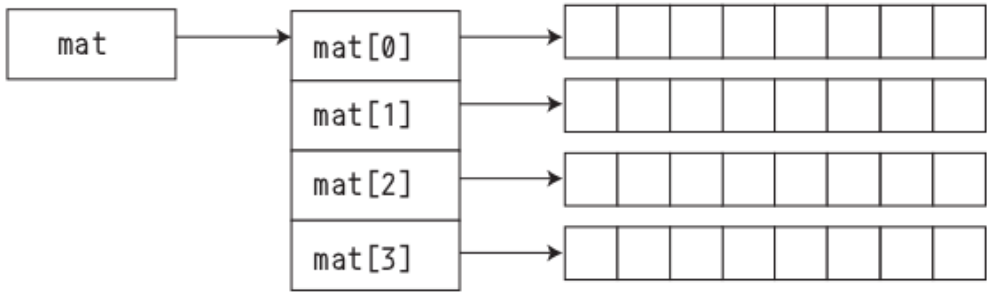
```
int (* const a3)[5][2] = new int [20][5][2];
cout << sizeof(a3) << endl;
cout << sizeof(a3[0]) << endl;
cout << sizeof(a3[0][0]) << endl;
cout << sizeof(a3[0][0][0]) << endl;
```

```
4
40
8
4
```

Zeiger
Mehrdimensionale dynamische C-Arrays

2. Arrays mit variabler Feldgröße:

- Alle Dimensionen werden *zur Laufzeit* bestimmt. Dadurch sind z.B. auch „nicht-rechteckige“ Arrays möglich.
- Keine Dimension ist Teil des Datentyps, und `sizeof()` funktioniert mit keiner Dimension!




```
int z,s;          // Zeilen , Spalten
cout << " Zeilen und Spalten eingeben : ";
cin >> z >> s;    // erst zur Laufzeit bekannt
// Feld von Zeigern auf Zeilen anlegen:
int* * const mat = new int* [z]; // mat ist ein konstanter Zeiger auf Zeiger auf int
// jeder Zeile Speicherplatz zuordnen:
for(size_t i = 0; i < z; ++i) {
  mat[i] = new int[s];
}
```

Zeiger
Mehrdimensionale dynamische C-Arrays

- Benutzung des Arrays ist wie bei statischen Arrays: Entweder mit dem Index-Operator (wie in diesem Beispiel) oder mit Zeigerarithmetik.

```
// Beispiel für die Benutzung der dynamisch erzeugten Matrix
for(size_t iz = 0; iz < z; ++iz) {
  for(size_t is = 0; is < s; ++is) {
    mat[iz][is] = iz * s + is;
    cout << mat[iz][is] << ' \ t ' ;
  }
  cout << endl;
}
```

- Bei dynamischen Arrays muss nach Gebrauch der Speicher wieder freigegeben werden (in umgekehrter Reihenfolge wie die Bereitstellung):

```
// Matrix freigeben
for(size_t zeile = 0; zeile < z; ++zeile) {
  delete [] mat[zeile]; // zuerst Zeilen freigeben
}
delete [] mat; // Feld mit Zeigern auf Zeilenanfänge freigeben
```

Zeiger
Klasse für dynamische 2D-Arrays

- Praktisch wäre eine Template-Klasse für dynamische 2D-Arrays. Die Benutzung könnte so aussehen:

```
#include "array2d.h"
#include<iostream>
using namespace std;

int main() {
  Array2d<int> arr(5, 7);  // einfache Definition
  arr.init(0);             // Initialisierung
  for(int z=0; z < arr.getZeilen(); ++z) {
    for(int s=0; s < arr.getSpalten(); ++s) {
      arr[z][s] = 10 * z + s;    // normale Nutzung, schreibend und ...
      cout << arr[z][s] << " " ; // lesend
    }
    cout << endl;
  }

  cout << "Array =:" << endl;
  printArray(arr); // Funktion zur Anzeige des Arrays, um Schleifen zu sparen
  // Konstruktor, der das Array initialisiert:
  Array2d<double> arrd(3, 4, 99.013);     // Array mit double-Zahlen
  printArray(arrd);
  Array2d<string> arrs(2, 5, " hello " ); // Array mit Strings
```

Zeiger
Klasse für dynamische 2D-Arrays

Realisierung:

```
#ifndef ARRAY2D_H
#define ARRAY2D_H
#include<stdexcept>
#include<algorithm>
#include<iostream>

template<typename T>
class Array2d {
  public:
    Array2d(int zeilen, int spalten);
    Array2d(int zeilen, int spalten, const T& init);
    Array2d(const Array2d& arr);
   ~Array2d();
    Array2d& operator=(const Array2d& arr);
    int getZeilen() const;
    int getSpalten() const;
    void init(const T& wert); // Alle Elemente mit wert initialisieren
    const T& at(int z, int s) const;
    T& at(int z, int s);
    const T * operator[](int z) const;
    T * operator[](int z);
  private:
    int zeilen;
    int spalten;
    T* *arr;
};
```

Zeiger
Klasse für dynamische 2D-Arrays

```
/* ----------- Implementierung ----------------------*/
template<typename T>
Array2d<T>::Array2d(int z, int s)                // Konstruktor
  : zeilen(z), spalten (s), arr(new T* [z]) {
  for(int z=0; z < zeilen; ++z) {
    arr[z] = new T[spalten];
  }
}
template<typename T>
Array2d<T>::Array2d(int z, int s, const T& wert) // Konstruktor
  : zeilen(z), spalten (s), arr(new T* [z]) {
  for(int z=0; z < zeilen; ++z) {
    arr[z] = new T[spalten];
  }
  init(wert);
}

template<typename T>
Array2d<T>::Array2d(const Array2d& a)            // Copy-Konstruktor
  : zeilen(a.zeilen), spalten (a.spalten),
    arr(new T *[zeilen]) {
  for(int z=0; z < zeilen; ++z) {
    arr[z] = new T[spalten];
    for(int s = 0; s < spalten; ++s) {
      arr[z][s] = a.arr[z][s];
    }
  }
}
```

Zeiger
Klasse für dynamische 2D-Arrays

```
template<typename T>
Array2d<T>::~Array2d() {              // Destruktor
  for(int z=0; z < zeilen; ++z) {
    delete [] arr[z];
  }
  delete [] arr;
}

template<typename T>
Array2d<T>& Array2d<T>::operator=(const Array2d& a) {
  Array2d<T> tmp(a); // temporäres Objekt erzeugen
  std::swap(zeilen,  tmp.zeilen);   // Verwaltungsdaten vertauschen
  std::swap(spalten, tmp.spalten);
  std::swap(arr, tmp.arr);
  return * this;
}

template<typename T>
inline int Array2d<T>::getZeilen() const {
    return zeilen;
}

template<typename T>
inline int Array2d<T>::getSpalten() const {
    return spalten;
}
```

Zeiger
Klasse für dynamische 2D-Arrays

```
template<typename T>
void Array2d<T>::init(const T& wert) {
  for(int z=0; z < zeilen; ++z) {
    for(int s = 0; s < spalten; ++s) {
      arr[z][s] = wert;
    }
  }
}
template<typename T>
inline const T * Array2d<T>::operator[](int z) const {
  return arr[z];
}
template<typename T>
inline T * Array2d<T>::operator[](int z) {
  return arr[z];
}

template<typename T>
inline const T& Array2d<T>::at(int z, int s) const {
  if(z < 0 || z >= zeilen) {
    throw std::range_error( "Array2d : Zeile außerhalb des erlaubten Bereichs " );
  }
  if(s < 0 || s >= spalten) {
    throw std::range_error( "Array2d : Spalte außerhalb des erlaubten Bereichs " );
  }
  return arr[z][s];
}
```

Zeiger
Klasse für dynamische 2D-Arrays

```
template<typename T>
inline T& Array2d<T>::at(int z, int s) {
  if(z < 0 || z >= zeilen) {
    throw std::range_error( "Array2d : Zeile außerhalb des erlaubten Bereichs " );
  }
  if(s < 0 || s >= spalten) {
    throw std::range_error( "Array2d : Spalte außerhalb des erlaubten Bereichs " );
  }
  return arr[z][s];
}

// Globale Funktion zur Ausgabe
template<typename T>
void printArray(const Array2d<T>& a) {
  for(int z=0; z < a.getZeilen(); ++z) {
    for(int s=0; s < a.getSpalten(); ++s) {
      std::cout << a[z][s] << " " ;
    }
    std::cout << std::endl;
  }
}
#endif
```

Zeiger
Binäre Ein- und Ausgabe

- Bei der binären Ein- und Ausgabe werden die Daten nicht in eine Textdarstellung gewandelt, sondern direkt verarbeitet.
- Dazu wird der `read()` und `write()` Funktion ein **`char *`**-Zeiger auf den Anfang der Daten übergeben:

```
#include<cstdlib>  // für exit( )
#include<fstream>
#include<iostream>
using namespace std;

int main( ) {
  ofstream ziel;
  // ios::binary mit ios::out verwenden
  ziel.open( "double.dat" , ios::binary|ios::out);
  if(!ziel) {
    cerr << " Datei kann nicht geöffnet werden ! \n" ;
    exit(-1);
  }
  double d = 1.0;
  for(int i = 0;i < 20; ++i, d *= 1.1) // Schreiben von 20 Zahlen
    ziel.write(reinterpret_cast<const char *>(&d), sizeof(d));
} // ziel.close() wird vom Destruktor durchgeführt
```

Zeiger
Binäre Ein- und Ausgabe

- Daten binär Einlesen:

```
int main( ) {
  ifstream quelle;
  quelle.open( "double . dat" , ios::binary|ios::in);
  if(!quelle) { // muss existieren
    cerr << " Datei kann nicht geöffnet werden ! \ n" ;
    exit(-1);
  }
  double d;
  while(quelle.read(reinterpret_cast<char *>(&d), sizeof(d)))
    cout << " " << d << endl;
  }
} // quelle.close() wird vom Destruktor durchgeführt
```

Ein- und Ausgabe mit ASCII-Code

+ Lesbar, editierbar
+ streaming mit << und >> kann verwendet werden
- dauert länger
- Dateigröße nicht exakt vorhersehbar

Binäre Ein- und Ausgabe

+ schnell (Ausgabe z.B. ein Matrix mit einer Anweisung!
+ definierte Dateigröße
- nicht lesbar

Zeiger
Zeiger auf Funktionen

- Zeiger auf Funktionen sind nützlich, wenn erst zur Laufzeit entschieden wird, welche Funktion tatsächlich aufgerufen werden soll.
- Grundlage für Polymorphie, bzw. *late binding*
- Bei der Deklaration muss entsprechend geklammert werden:

```
int   *func(int, int); // Deklaration einer Funktion, die int* zurückliefert!!!
int  (*func)(int, int); // Zeiger auf eine Funktion, die int  zurückliefert
int *(*func)(int, int); // Zeiger auf eine Funktion, die int* zurückliefert
```

Anwendung:

```
int max(int x, int y) { return x > y ? x : y;}
int min(int x, int y) { return x < y ? x : y;}

int main() {
  int a = 1700, b = 1000;
  int (*fp)(int, int);  // fp ist Zeiger auf eine Funktion

  fp = max; // Zuweisung des Funktionszeigers zur Laufzeit
  cout << (*fp)(a, b) << endl; // Dereferenzierung und Aufruf
  cout << fp(a, b) << endl;    // Implizite Umwandlung Zeiger → Funktion
}
```

Zeiger
Zeiger auf Funktionen

- 2. Beispiel: `qsort()` aus `<cstdlib>`:
  (in C++ würde man an Stelle der `void`-Zeiger ein Template nehmen, und an Stelle eines C-Arrays einen `vector` sortieren...)

```
void qsort(void *feldname,
           size_t zahlDerElemente,
           size_t bytesProElement,
           int (*cmp)(const void * a, const void * b)); // Vergleichsfunktion
```

- `cmp()` soll `-1` liefern, wenn `a<b`, `0` wenn `a==b` und `1` wenn `a>b`.

```
// Definition der Vergleichsfunktion
int icmp(const void * a, const void * b) {
  int ia = * static_cast<const int * >(a); // Typumwandlung auf int* und Deref.
  int ib = * static_cast<const int * >(b);
  if(ia == ib) return 0;
  return ia > ib? 1 : -1;
}
```

```
int main() {
  int ifeld[] = {100,22,3,44,6,9,2,1,8,9};
  size_t anzahlElemente = sizeof(ifeld)/sizeof(ifeld[0]);
  qsort(ifeld, anzahlElemente, sizeof(ifeld[0]), icmp);
  for(size_t i = 0; i < anzahlElemente; ++i) // Ausgabe des sortierten Feldes
    cout << ' ' << ifeld[i];
  cout << endl;
}
```

Zeiger
Der `this`-Zeiger

- Das Schlüsselwort **`this`** ist innerhalb der Methoden einer Klasse bzw. Struktur ein Zeiger auf das Objekt selber.
- Die Dereferenzierung **`*this`** ist das Objekt selber, bzw. kann wie ein Alias-Namen für das Objekt aufgefasst werden.
- Machmal ist `this` notwendig, um z.B. eindeutig auf ein Attribut einer Klasse zu verweisen:

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

class A {
  private:
    int a;
  public:
    void set1(int a) { a=a; } // Parameter a wird gesetzt!
    void set2(int a) { this->a=a; }
    int  get() { return a; }
};

int main() {
  A mya;
  mya.set1(8);  cout << mya.get() << endl;
  mya.set2(8);  cout << mya.get() << endl;
}
```

```
-1217773580
8
```

Zeiger
Komplexe Deklarationen lesen

- Komplexe Funktionsdeklarationen können in C/C++ schnell unübersichtlich werden, wenn C-Arrays und Zeiger massiv verwendet werden.
- Beispiel: Was ist `char *(*(*seltsam)(double, int))[3]` ????
- Rangfolge von typischen Elementen in Deklarationen:

`( )`	Ordnungsklammern
`f( )`	Funktionsklammern
`[ ]`	Array
`*`	Zeiger

Vorgehen:

- Suche den Funktions- oder Variablennamen
- Nach der obigen Rangfolge von innen nach außen weiterhangeln...
- Nur das erste `*` links oder Klammerpaar `[]` rechts beeinflußt die Interpretation des Namens
- Jetzt kommt ein Beispiel :)

Zeiger
Komplexe Deklarationen lesen

Beispiel:

```
char *(*(*seltsam)(double, int))[3]
```
Name → seltsam

```
char *(*(*seltsam)(double, int))[3]
```
... ist Zeiger

```
char *(*(*seltsam)(double, int))[3]
```
... auf Funktion mit Parametern `(double,int)`

```
char *(*(*seltsam)(double, int))[3]
```
… die einen Zeiger zurückliefert

```
char *(*(*seltsam)(double, int))[3]
```
... auf ein Array mit 3 Elementen

```
char *(*(*seltsam)(double, int))[3]
```
... die jeweils einen Zeiger enthalten

```
char *(*(*seltsam)(double, int))[3]
```
... auf den Typ `char`

Nützliches Tool zum 'Dekodieren' von Deklarationen:
cdecl

Zeiger
typedef

- Mit typedef können komplizierte Typen abgekürzt (und lesbarer) geschrieben werden.
- Entweder wird ein existierender Datentyp mit einem Alias versehen, oder eine Funktionsdeklaration dient als Vorlage für weiter Funktions-Deklarationen

```
char *(*(*seltsam)(double, int))[3];   // Normale Deklaration wie vorher

typedef double REAL;  // Alias für elementaren Datentyp
REAL a;

typedef struct { int i; int j; } NEUER_TYP;  // Alias für zusammengesetzten Datentyp
NEUER_TYP neu;

typedef char *(*(*SELTSAMERTYP1)(double, int))[3]; // Alias für unser 'seltsam'
SELTSAMERTYP1 seltsam1;

typedef char *(* ZeigerAufArrayVon3CharZeigern)[3]; // Alias für Rückgabetyp
typedef ZeigerAufArrayVon3CharZeigern (* SELTSAMERTYP2)(double, int);
SELTSAMERTYP2 seltsam2;
```

Zeiger
Standard-Typumwandlungen für Zeiger

- `0` kann jedem Zeiger zugeordnet werden!
- Jeder Zeiger kann zu `bool` konvertiert werden (`NULL` → `false`).
- Jeder Zeiger kann auf einen void-Zeiger konvertiert werden.
- Jeder Zeiger auf ein Objekt der Klasse B kann auf einen Zeiger der Basisklasse A konvertiert werden.
- Arrays können zu Zeigern konvertiert werden (siehe vorher)
- Funktionen können in Funktionszeiger (oder anders herum) umgewandelt werden:

```
void drucke(int x) { cout << x << endl; }

int main() {
  void (*f1)(int) = &drucke; // Explizit → keine Umwandlung
  void (*f2)(int) = drucke;  // Implizit: Funktion → Zeiger
  f2(42);                    // Implizit: Zeiger → Funktion
  (*f1)(42);                 // Explizit: Zeiger → Funktion
}
```

Objektorientierung 2
Klassen, die dynamischen Speicher verwalten

- Das Bereitstellen und Freigeben von dynamischem Speicher kann fehleranfällig sein!

- Daher sollte das Bereitstellen/Umkopieren/Löschen von dynamischem Speicher möglichst '*hinter den Vorhängen*' passieren, z.B. durch eine Klasse gekapselt.

- Zusätzlich ist die Bearbeitung von C-Strings sehr umständlich und fehleranfällig. So setzen machen Funktionen (z.B. `strcpy(dst, src)`) vorraus, dass die Parameterzeiger auf einen genügend großen Speicherbereich zeigen.

  → Lösung: `string`-Klasse (aus `<string>`)

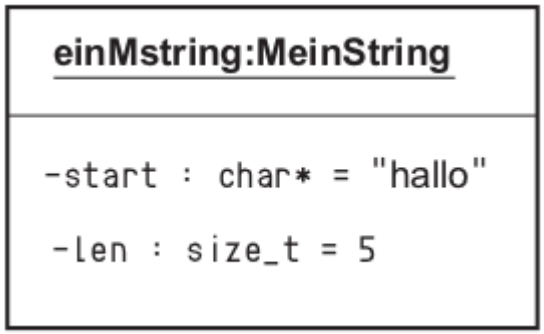
- Zur Veranschaulichung wird hier die vereinfachte Klasse **`MeinString`** vorgestellt, die neben der Speicherverwaltung auch die Prüfung des terminierenden Nullbytes `'\0'` des C-Strings übernimmt!

Objektorientierung 2
Klassen `MeinString`

- Jede Klasse, die dynamischen Speicher verwaltet, braucht einen *Destruktor*, *Kopierkonstruktor* und einen *Zuweisungsoperator*! (Details folgen noch...)

- Die Klasse `MeinString` versucht, zur Standard-String-Klasse bzgl. der Methodennamen kompatibel zu sein!

- Als zentrales Attribut verwendet `MeinString` einen privaten `char *`-Zeiger, der auf den (gekapselten) aktuellen C-String zeigt, der wie üblich `'\0'`-terminiert ist. Zusätzlich wird (aus Performancegründen) die Länge dieses Strings gespeichert.



UML-Objektdiagramm einer `MeinString`-Instanz

Objektorientierung 2
Klassen `MeinString`

```
// Einfache String-Klasse. Erste, nicht vollständige Version
#ifndef MEINSTRING_H
#define MEINSTRING_H
#include<cstddef>                    // size_t
#include<iostream>                   // ostream

class MeinString {
  public:
   MeinString();                     // Default-Konstruktor
   MeinString(const char *);         // allgemeiner bzw. Typumwandlungs-Konstruktor
   MeinString(const MeinString&);    // Kopierkonstruktor
   ~MeinString();                    // Destruktor
   MeinString& assign(const MeinString&); // Zuweisung eines MeinString
   MeinString& assign(const char *);      // Zuweisung eines char*
   const char& at(size_t position) const; // Zeichen holen
   char& at(size_t position);             // Zeichen holen,
                    //  die Referenz erlaubt Ändern des Zeichens
   size_t length() const { return len;}   // Stringlänge zurückgeben
   const char* c_str() const { return start; } // C-String zurückgeben
  private:
   size_t len;                       // Länge
   char *start;                      // Zeiger auf den Anfang
};

void anzeigen(std::ostream& os, const MeinString& m); // globale Methode zur
                                                      // Ausgabe
#endif
```

Implementierung

Default-Konstruktor:

```
MeinString::MeinString()          // Default-Konstruktor
: len(0), start(new char[1]) {    // Platz für '\0'
    *start = '\0';                // leerer String
}
```

Allgemeiner Konstruktor (auch Typumwandlungs-K.):

```
MeinString::MeinString(const char *s)      // allg. Konstruktor
: len(strlen(s)), start(new char[len+1]) {
    strcpy(start, s);
}
```

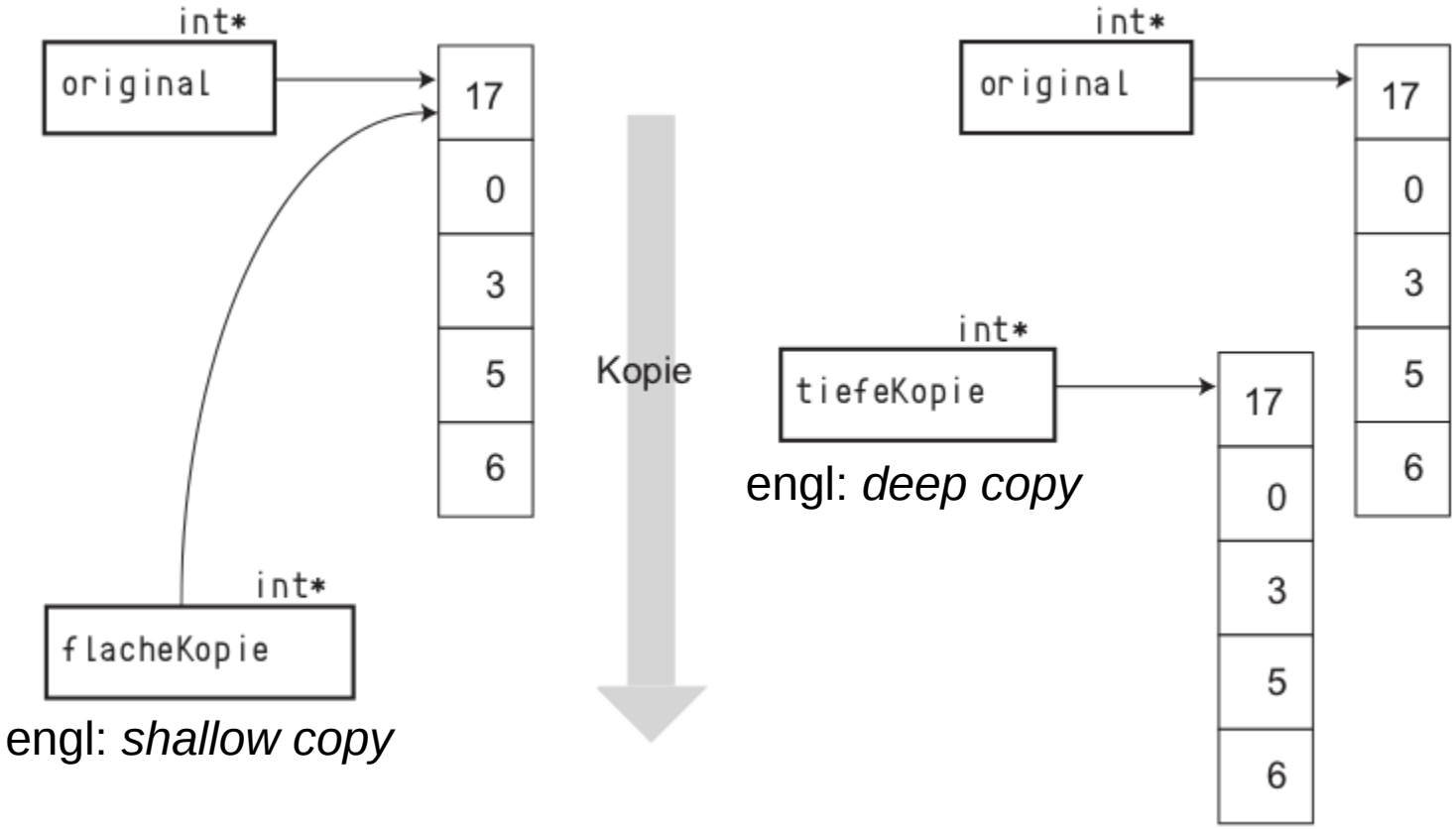
Achtung: Reihenfolge der Initialisierung hängt von der Reihenfolge der Deklarationen in der Klasse ab (hier `len`, dann `start`)!!

Kopier-Konstruktor:

```
MeinString::MeinString(const MeinString &s)      // Kopier-Konstruktor
: len(s.len), start(new char[len+1]) {
    strcpy(start, s.start);
}
```

Klassen `MeinString`

Der Kopier-Konstruktor ist hier *notwendig*, um eine '*tiefe*' Kopie zu erzeugen:



engl: *shallow copy*

engl: *deep copy*

Klassen `MeinString`

Destruktor:

```
MeinString::~MeinString() {       // Destruktor
    delete [] start;
}
```

Zuweisungs-Methoden:

```
MeinString& MeinString::assign(const MeinString& m)  { // Zuweisung eines MeinString
    char *p = new char[m.len+1]; // neuen Platz beschaffen
    strcpy(p, m.start);
    delete [] start;             // alten Platz freigeben
    len   = m.len;               // Verwaltungsinformation aktualisieren
    start = p;
    return *this;
}

MeinString& MeinString::assign(const char *s) {  // Zuweisung eines char *
    size_t L = strlen(s);
    char *p = new char[L+1];
    strcpy(p, s);
    delete [] start;             // alten Platz freigeben
    len   = L;                   // Verwaltungsinformation aktualisieren
    start = p;
    return *this;
}
```

Objektorientierung 2
Klassen `MeinString`

- Bei den Zuweisungs-Methoden wird einem bereits existierenden Objekt ein neuer Inhalt zugewiesen → daher muss der aktuell benutzte Speicher freigegeben werden!

- Sonderfall: Selbstzuweisung. In diesem Fall könnte optimiert werden, da die Zuweisungs-Methoden in diesem Fall gar nichts zu tun hätten...

Wie sähe so ein Selbsttest konkret aus ??

Index-Methoden:

```
char& MeinString::at(size_t position)  {// Zeichen per Referenz holen
    assert(position < len);  // Nullbyte lesen ist nicht erlaubt
    return start[position];
}

const char& MeinString::at(size_t position) const   {// Zeichen holen
    assert(position < len);  // Nullbyte lesen ist nicht erlaubt
    return start[position];
}
```

- Zugriff auf `'\0'` wird verboten!
- Die erste Variante erlaubt z.B. `meinString.at(0) = 'M';`

Objektorientierung 2
Klassen `MeinString`

Globale Ausgabe-Methode:

```
void anzeigen(std::ostream& os, const MeinString& m)  {
        os << m.c_str();
}

…
MeinString mys("Hallo");
anzeigen(cout, mys);      // Aufruf der globalen Methode
…
```

- Durch den allgemeinen Konstruktor (auch Typumwandlungsk.) und die `assign(const char * s)`-Methode sind z.B. folgende Anweisungen möglich:

```
MeinString einString;                  // Standardkonstruktor
MeinString nochEinString( "hallo" );   // allg. Konstruktor
einString.assign(nochEinString);       // Zuweisung
einString.assign( "neuer Text" );      // Zuweisung
einString.assign(einString);           // Zuweisung auf sich selbst?
einString = nochEinString;             // geht (noch) nicht ...
```

Optimierung der MeinString-Klasse

- Bisher wird bei vielen Methoden neuer Speicher angefordert, auch wenn z.B. bei einer `assign()`-Methode der aktuelle Speicher für den neuen Inhalt ausreichen würde :(
- Idee: Neben der Länge (`length()`) hat jeder String auch eine Kapazität (`capacity()`), die immer größer oder gleich der Länge ist!
- Wenn nun z.B. bei einer Zuweisung der Kapazität des `MeinString`s ausreicht, wird kein `new` und `delete` aufgerufen.

```
// Auszug aus meinstring.h
public:
  size_t capacity() const { return cap; } // Kapazität zurückgeben
  void reserve(size_t bytes);             // Platz reservieren mit Erhalt des Inhalts
  void shrink_to_fit();                   // Kapazität auf aktuellen Bedarf schrumpfen
private:
  size_t cap;                        // Kapazität (direkt nach len eintragen)
  void reserve_only(size_t bytes);   // Hilfsmethode: Nur Platz reservieren
```

Objektorientierung 2
Klassen `MeinString`

```
// Auszug aus meinstring.cpp
MeinString::MeinString()
: len(0), cap(0), start(new char[1]) {  // Platz für '\0'
  *start = ' \0 ';                      // leerer String
}

MeinString::MeinString(const char * s)  // allg. Konstruktor
: len(strlen(s)), cap(len), start(new char[cap+1]) {
  strcpy(start, s);
}

MeinString::MeinString(const MeinString& m) // Kopierkonstruktor
: len(m.len), cap(len), start(new char[cap+1]) {
  strcpy(start, m.start);
}

void MeinString::reserve(size_t groesse) {
  if(groesse > cap) {
    char * temp = new char[groesse+1];  // neuen Platz beschaffen (+1 wegen '\0')
    strcpy(temp, start);                // umkopieren
    delete [] start;                    // alten Platz freigeben
    start = temp;                       // Verwaltungsinformation aktualisieren
    cap = groesse;                      // Verwaltungsinformation aktualisieren
  }
}
```

Objektorientierung 2
Klassen `MeinString`

```
void MeinString::reserve_only(size_t groesse) {
  if(groesse > cap) {
    char * temp = new char[groesse+1]; // neuen Platz beschaffen
    delete [] start; // alten Platz freigeben
    start = temp;    // Verwaltungsinformation aktualisieren
    cap = groesse;   // Verwaltungsinformation aktualisieren
  }
}

void MeinString::shrink_to_fit() {
  if(cap > len) {
    char * temp = new char[len+1]; // neuen Platz beschaffen
    strcpy(temp, start);           // umkopieren
    delete [] start;               // alten Platz freigeben
    start = temp;                  // Verwaltungsinformation aktualisieren
    cap = len;                     // Verwaltungsinformation aktualisieren
  }
}

MeinString& MeinString::assign(const MeinString& m) {
  reserve_only(m.len);
  strcpy(start, m.start);
  len = m.len;
  return * this;
}
```

```
MeinString& MeinString::assign(const char * s) {
  size_t temp = strlen(s);
  reserve_only(temp);
  strcpy(start, s);
  len = temp;
  return * this;
}
```

Objektorientierung 2
friend-Funktionen

- Die globale `anzeigen`()-Funktion der Klasse `MeinString` darf nur auf öffentliche Elemente der Klasse zugreifen (hier: `c_str()`).

- Manchmal möchte man auch auf `private`-Elemente einer Klasse von einer globalen Funktion aus zugreifen...

- Lösung: `friend`-Funktion:

```
class MeinString {
  public:
  ...
    friend void anzeigen(std::ostream& os, const MeinString&);
};

void anzeigen(std::ostream& os, const MeinString& m) {
  os << m.start;  // Zugriff auf privates Element!!
}
```

- Auch ganze Klassen können 'befreundet' sein:
  `friend class foo;` im würde allen Methoden der Klasse foo Zugriff gewähren!
- `friend` ist nicht transitiv!! `A` Freund von `B` ∧ `B` Freund von `C`,
  `A` ist nicht Freund von `C` !!

Objektorientierung 2
Klassenspezifische Daten und Methoden

- Klassenvariablen und Klassenmethoden existieren nur einmal für alle Instanzen einer Klasse, also auch, wenn noch gar keine Instanz der Klasse existiert.
- Bei der Deklaration werden die Elemente dazu als **`static`** markiert.
- Klassenmethoden dürfen nur auf Klassenvariaben zugreifen.
- Der Zugriff auf Klassenelemente erfolgt mit **`Klasse::<element>`** oder auch (wie bisher) über beliebige Instanzen der Klasse.
- Klassenvariablen müssen einmal definiert (und ggf. initialisiert) werden (i.d.R. in der `*.cpp`-Datei). Dazu wird folgende Syntax verwendet: `<Typ> Klasse::<Variablenname> [= Initialwert];`
- Klassenspezifische Konstanten und `enums` brauchen nicht außerhalb der Klassendefinition definiert werden.
- Typisches Anwendungsbeispiel: Ein Zähler für die Instanzen einer Klasse...

Objektorientierung 2
Klassenspezifische Daten und Methoden

- Die Klasse zählt die Anzahl der Instanzen der Klasse, und gibt jeder Instanz eine eindeutige Seriennummer:

Header-Datei:

```
#ifndef NUMOBJ_H
#define NUMOBJ_H

class NummeriertesObjekt   {// noch nicht vollständig!!! (siehe Text)
  public:
    NummeriertesObjekt();
    NummeriertesObjekt(const NummeriertesObjekt&);
    ~NummeriertesObjekt();

    unsigned long seriennummer() const { return serienNr; }
    static int anzahl() { return anz;} // Zugriff nur auf static-Elemente!!

  private:
    static int anz;                 // ANZAHL der Instanzen dieser Klasse
    static unsigned long maxNummer;
    const unsigned  long serienNr;  // eindeutige SERIENNUMMER
};
#endif
```

Objektorientierung 2
Klassenspezifische Daten und Methoden

Implementations-Datei:

```
...
#include"numobj.h"

// Initialisierung und Definition der klassenspezifischen Variablen:
int           NummeriertesObjekt::anz       = 0;
unsigned long NummeriertesObjekt::maxNummer = 0L;

// Default-Konstruktor
NummeriertesObjekt::NummeriertesObjekt() : serienNr(++maxNummer) {
   ++anz;
   cout << "Objekt Nr. " << serienNr << " erzeugt" << endl;
}
// Kopierkonstruktor
NummeriertesObjekt::NummeriertesObjekt(const NummeriertesObjekt& nobj)
   :  serienNr(++maxNummer) {
   ++anz;
   cout << "Objekt Nr. " << serienNr << " erzeugt" << endl;
}
// Destruktor
NummeriertesObjekt::~NummeriertesObjekt() {
   --anz;
   cout << "Objekt Nr. " << serienNr << " gelöscht" << endl;
}
```

Objektorientierung 2
Klassen-Templates

- Genau wie für Funktionen gibt es auch Templates für ganze Klassen: `template<typename T> class A { … };`
- Innerhalb der Klassendefinition heißt die Klasse dann A<T>, außer bei den Konstruktor- und Destruktor-Namen.
- Außerhalb der Klassendefinition müssen die Methoden mit der entsprechenden template-Angabe definiert werden:
  `template<typename T>  int A<T>::foo() { … }`
- Die Template-Methoden sollten `inline` sein, oder direkt in der Header-Datei definiert werden.
- Klassen-Templates können spezialisiert werden:
  `template<> class A<bool> { … };`
- Klassen-Templates können auch mehrere Parameter haben, die auch elementare Typen oder Zeiger sein können (incl. default-Werten):
  `template<typename T1, typename T2=char, bool FLAG=true> …`
- Typisches Beispiel: Eine generische Stack-Klasse:

Objektorientierung 2
Klassen-Templates

Header-Datei:

```
// ein einfaches Stack-Template
#ifndef SIMSTACK_T
#define SIMSTACK_T
#include<cassert>

template<typename T, unsigned int maxSize=20>
class SimpleStack {
  public:
    SimpleStack() : anzahl(0){}
    bool empty() const { return anzahl == 0;}
    bool full() const  { return anzahl == maxSize;}
    unsigned int size() const { return anzahl;}
    void clear() { anzahl=0; }          // Stack leeren

    // Vorbedingung für top und pop: Stack ist nicht leer
    const T & top() const;              // letztes Element sehen
    T pop();                            // Element entnehmen

    // Vorbedingung für push: Stack ist nicht voll
    void push(const T &x);             // x auf den Stack legen

   private:
    unsigned int anzahl;
    T array[maxSize];           // Behälter für Elemente
};
```

Objektorientierung 2
Klassen-Templates

Header-Datei (Fortsetzung):

```
template<typename T, unsigned int m>
const T & SimpleStack<T, m>::top() const {
    assert(!empty());
    return array[anzahl-1];
}

template<typename T, unsigned int m>
T SimpleStack<T, m>::pop() {
    assert(!empty());
    return array[--anzahl];
}

template<typename T, unsigned int m>
void SimpleStack<T, m>::push(const T &x) {
    assert(!full());
    array[anzahl++] = x;
}

#endif
```

```
// Benutzung:
SimpleStack<int,100> s1; // Stack mit 100 int-Werten
SimpleStack<double>  s2; // Stack mit 20  double-Werten
s1.push(12);
…
```

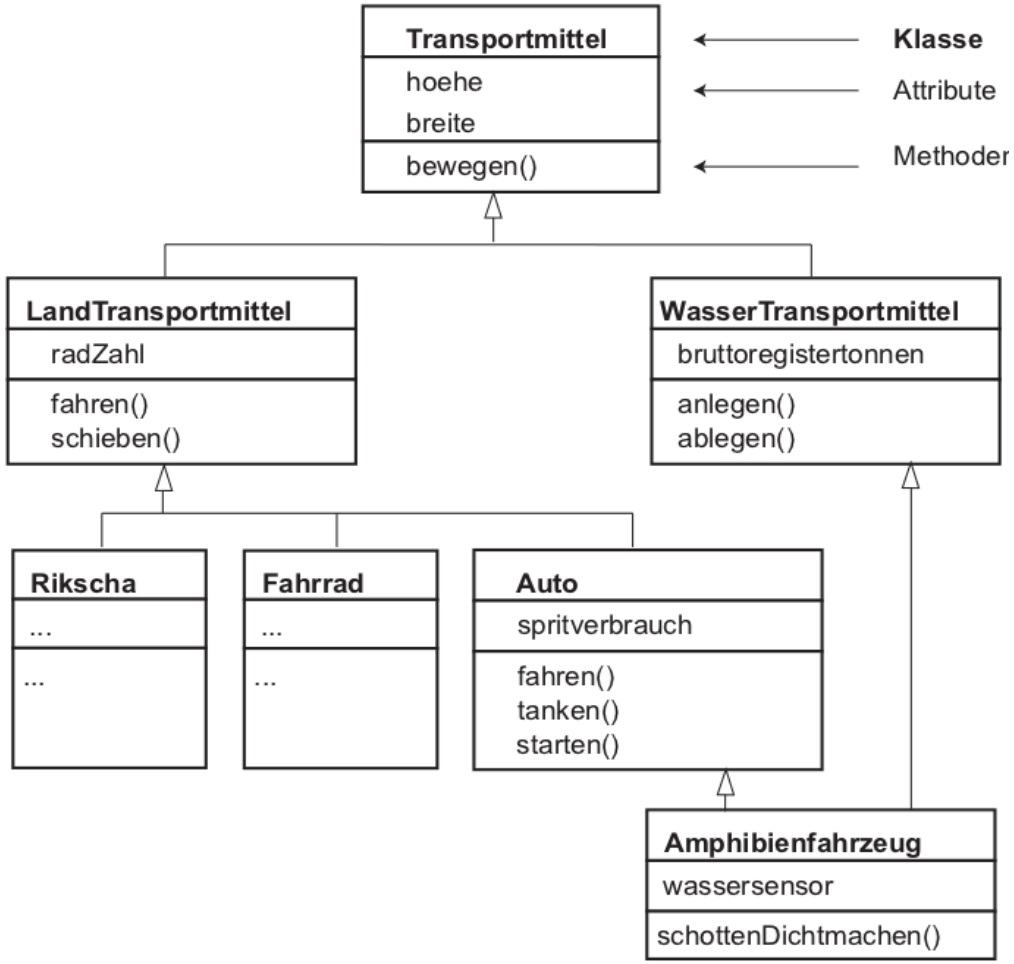
Objektorientierung 2
Template-Metaprogrammierung

- Mit Templates kann der Compiler dazu gebracht werden, zur Übersetzungszeit Berechnungen durchzuführen.
- Dabei gibt es keine veränderlichen Variablen wie bisher, sondern einmal definierte Werte bleiben erhalten.
- Schleifen werden dabei durch Rekursionen realisiert.
- Durch den Einsatz von **enum** kann das Reservieren von Speicher für Werte vermieden werden.
- Metaprogrammierung ist eine Art der *funktionalen Programmierung*!

```
template<int n>
struct Zweihoch {
  enum { wert = 2*Zweihoch<n-1>::wert };
};

template<>
struct Zweihoch<0> {
   enum { wert = 1};
};

int main() {
  std::cout << Zweihoch<11>::wert << std::endl;  // Ausgabe: 2048
}
```

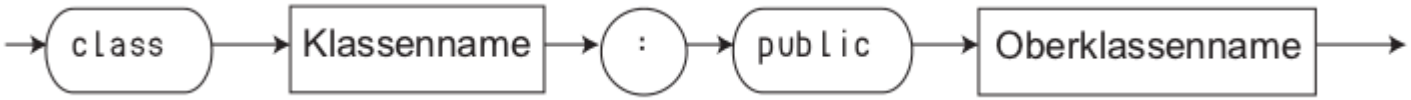
Vererbung

- Durch Vererbung wird in objektorientierten Sprachen eine '*ist-ein*' Beziehung dargestellt.
- Die Oberklasse ist eine Abstraktion oder Generalisierung von ähnlichen Eigenschaften und Verhaltensweisen.
- Unterklassen erben von der Oberklasse Attribute und Methoden, nur die Abweichungen bzw. Ergänzungen müssen programmiert werden.



Vererbung

Syntax (einer `public`-Vererbung):

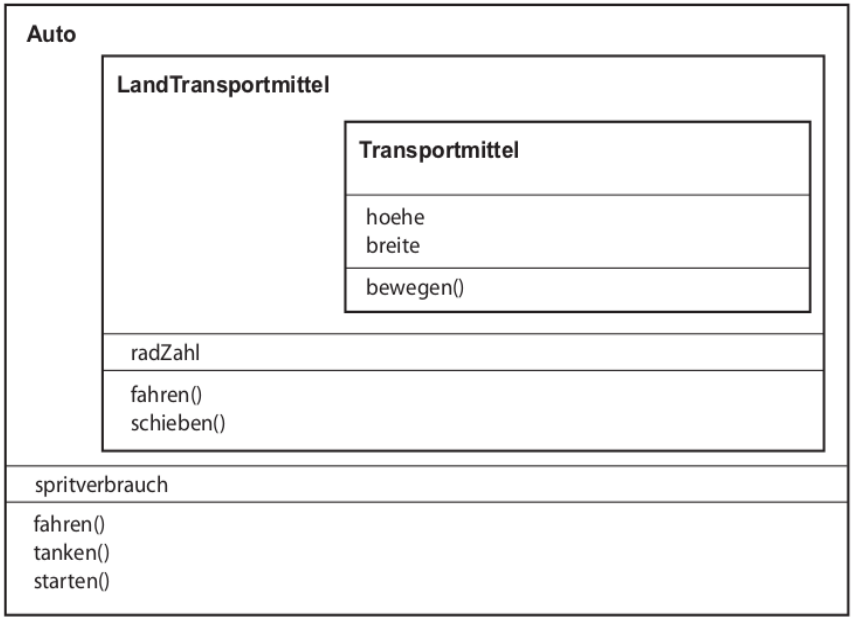



```
class Transportmittel {
  public:
    void bewegen();
  private:
    double hoehe, breite;
};
class LandTransportmittel
 : public Transportmittel {  // erben
  public:
    void fahren();
    void schieben();
  private:
    int radZahl;
};
class WasserTransportmittel
 : public Transportmittel { // erben
  public:
    void anlegen();
    void ablegen();
  private:
    double bruttoregistertonnen;
};
```

```
class Auto
 : public LandTransportmittel { // erben
  public:
    void fahren(); // überschreibt!
    void tanken();
    void starten();
  private:
    double spritverbrauch;
};

// Mehrfachvererbung:
class Amphibienfahrzeug
 : public Auto, public WasserTransportmittel {
  public:
    void schottenDichtmachen();
  private:
    const char * wassersensor;
};
```

Vererbung

- Jedes abgeleitete Objekt enthält ein (anonymes) Objekt der Oberklasse.
- Methoden der Oberklasse sind auf dem abgeleiteten Objekt aufrufbar, falls die Methode als `public` deklariert ist.
- Die abgeleitete Klasse kann Methoden definieren, die die gleiche Signatur wie eine der geerbten Methoden hat.
  Dadurch wird die geerbte Methode 'überschrieben' (im Beispiel: `fahren()`).



- Eine Klasse ist ein Datentyp in C++. Eine abgeleitete Klasse ist ein Subtyp der Oberklasse, und zu dieser kompatibel:

```
einLandTransportmittel = einAuto; // OK, aber Datenverlust!
einAuto = einLandTransportmittel; // FEHLER!!
```

Vererbung
Zugriffsschutz

- Falls eine Oberklasse keinen Standard-Konstruktor bereitstellt, muss im Konstruktor der abgeleiteten Klasse in der Initialisierungsliste der Konstruktor der Oberklasse aufgerufen werden:

```
Auto() : LandTransportmittel(4) { … }
```

  Attribute der Oberklasse dürfen in der Initialisierungsliste nicht angegeben werden!

- Konstruktoren, Destruktoren, Zuweisungsoperatoren und '`friend`s' werden NICHT vererbt!

Zugriffsschutz

`public`:	Keine Zugriffsbeschränkung
`private`:	Nur innerhalb der eigenen Klasse und von `friend`-Klassen und -Funktionen zugreifbar
`protected`:	Wie private, aber Zugriff zusätzlich von allen abgeleiteten Klassen erlaubt.

Wie werden die Zugriffsrechte bei der Vererbung weitergegeben?

Vererbung
Zugriffsschutz

- Das Zugriffsrecht ist abhängig von der Vererbung (default `private`):

	Zugriffsrecht in der abgeleiteten Klasse		
Zugriffsrecht in der Basisklasse	`public`-Vererbung	`protected`-Vererbung	`private`-Vererbung
`private`	kein Zugriff	kein Zugriff	kein Zugriff
`protected`	`protected`	`protected`	`private`
`public`	`public`	`protected`	`private`

→ Der Standard-Fall ist die `public`-Vererbung. `protected`- und `private`-Vererbungen spielen eine untergeordnete Rolle.

- `struct s { …` ist eine Abkürzung für `class s { public:` ... Grundsätzlich sind alle Eigenschaften von Klassen auch auf Strukturen anwendbar!

Vererbung
Überschreiben von Methoden

- Überschriebene Methoden besitzen die gleiche Signatur wie eine Methode der Basisklasse.
- Der Compiler entscheidet zur Übersetzungszeit, welche Methode aufgerufen wird (*statische Bindung*):

```
class base {
  void foo(void);
};
class derived : public base {
  void foo(void);
};
base    b, *pb=&b, &rb=b;
derived d, *pd=&d, &rd=d;
base    &rbd = d;
pb->foo();  // base::foo()
rb.foo();   // base::foo()
pd->foo();  // derived::foo()
rd.foo();  // derived::foo()
pb=pd;      // OK, d 'ist ein' b
pb->foo();  // base::foo() !!!
rbd.foo(); // base::foo() !!!
```

- Obwohl der Zeiger `pb` auf ein `derived`-Objekt zeigt, wird die Methode der Basisklasse `base` aufgerufen, da `pb` vom Typ `base` ist!

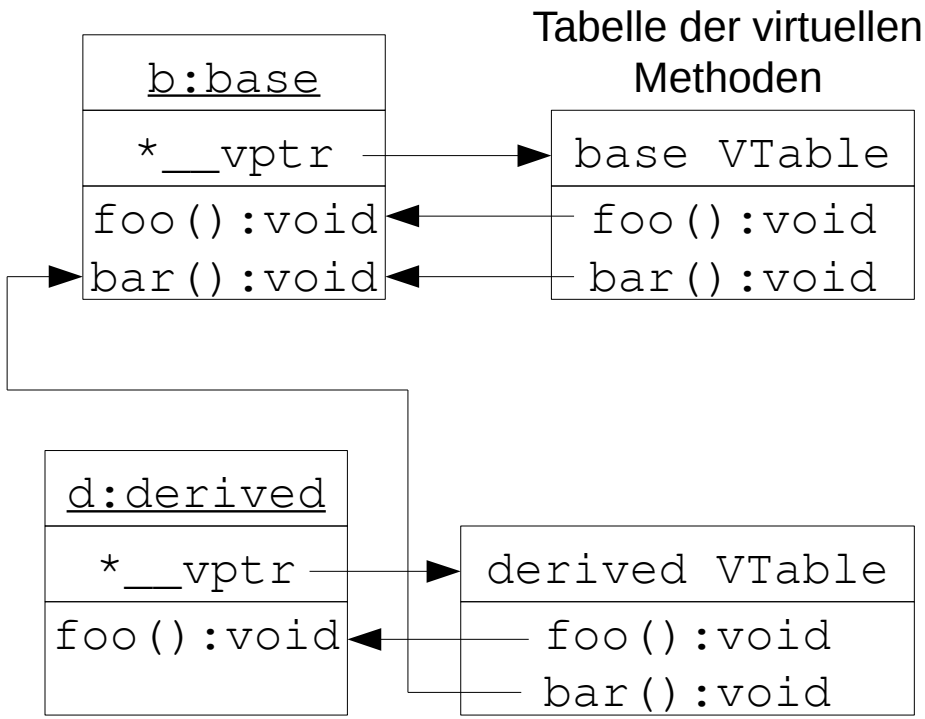
Vererbung
Polymorphismus

- Polymorphismus bedeutet „Vielgestaltigkeit“ und bedeutet in der objektorientierten Programmierung, dass erst zur *Laufzeit* die zum jeweiligen Objekt passende Realisierung einer Methode ermittelt wird.
- Den Vorgang nennt man *dynamischen Binden* (engl. *dynamic binding* oder *late binding*)
- Eine zur Laufzeit ausgewählte Methode heißt *virtuelle* Methode, und wird mit dem Schlüsselwort `virtual` deklariert.
- Hinter den Vorhängen wird dieses Verhalten mit einem speziellen Zeiger realisiert, der auf eine Klassen-eigene spezielle Tabelle (engl. *virtual table*) zeigt.
- Durch diesen Mechanismus werden Objekte etwas größer, und Methodenaufrufe geringfügig langsamer. In Java ist dieses Verhalten der Standard, d.h. statische Bindung existiert dort nicht!
- **Regel:** Alle geerbten Methoden, die überschrieben werden, sollten in der Basisklasse als `virtual` deklariert sein!

Vererbung
Polymorphismus

- Virtuelle Methoden sind automatisch auch in allen abgeleiteten Klassen virtuell. Trotzdem sollte auch dort das Schlüsselwort **`virtual`** verwendet werden, auch wenn es nicht notwendig ist.

```
class base {
  virtual void foo(void);
  virtual void bar(void);
};

class derived : public base {
  // virtual eigentl. nicht notwendig
  virtual void foo(void);
};
base    b, *pb=&b, &rb=b;
derived d, *pd=&d, &rd=d;
base    &rbd = d;

pb->foo();  // base::foo()
rb.foo();   // base::foo()
pd->foo();  // derived::foo()
rd.foo();  // derived::foo()
pb=pd;      // OK, d 'ist ein' b
pb->foo();  // derived::foo() !!!
rbd.foo();  // derived::foo() !!!
rbd.bar();  // base::bar()
```



Vererbung
Polymorphismus

- Beim Überschreiben von virtuellen Methoden muss der Programmierer selber sicherstellen, dass die Signaturen identisch sind! Ein Kontrollmechanismus wie `@Override` in Java gibt es erst in C++11 (`virtual void foo(void) override { … }`).
- Polymorphie funktioniert nur über Zeiger oder Referenzen der Basisklasse.
- Destruktoren sollen bei Ableitungshierarchien auch virtuell sein: Beispiel bei statischer Bindung:

```
base *pb; derived *pd = new derived;
// ...später
pb = pd; delete pb; // ruft lediglich base::~base()!!!
```

- Eine virtuelle Methode in einer Basisklasse kann als Schnittstelle für abgeleitete Klassen gelten, auch wenn die Methode in der Basisklasse (noch) nicht definiert ist. Basisklassen-Zeiger auf abgeleitete Objekte rufen immer die passende Methode der abgeleiteten Klasse auf.

Vererbung
Abstrakte Klassen

- Oft soll eine Basisklasse für einzelne Methoden (oder für alle Methoden → vgl. Java `interface`) lediglich eine Schnittstelle vorgeben, ohne diese selber zu implementieren.
- Dazu kann eine Methode als abstrakte Methode angelegt werden, wobei die zugehörige Klasse automatisch auch abstrakt wird → von diesen Klassen können KEINE Instanzen angelegt werden!
- Das syntaktische Mittel für abstrakte Methoden in C++ sind *reine virtuelle Methoden* (engl. *pure virtual method*).
- Dazu wird die Methode durch die Ergänzung `=0` deklariert:

```
class base {
  virtual void foo(void) = 0;
};
```

- Wenn eine abgeleitete Klasse vergisst, die rein virtuelle Methode zu implementieren, kommt es bei der Instanziierung der abgeleiteten Klasse zu einem Compilerfehler!

Vererbung
Abstrakte Klassen

Beispiel für eine abstrakte Basisklasse:

```
// shape.h

class Shape {  // abstrakte Klasse
  protected: int x_, y_; // Ursprungskoordinaten
  public:
    Shape(int x, int y) : x_(x), y_(y) {}
    virtual double area() const = 0;
    void move(int dx, int dy) { x_ += dx; y_ += dy; }
};

class Rectangle : public Shape {  // konkrete Klasse
  private: int h_, w_; // Höhe und Breite
  public:
    Rectangle(int x, int y, int h, int w)
     : Shape(x, y), h_(h), w_(w) {}
    virtual double area() const { return h_ * w_; }
};

class Circle : public Shape {  // konkrete Klasse
  private: int r_;  // Radius
  public:
    Circle(int x, int y, int rad) : Shape(x, y), r_(rad) {}
    virtual double area() const { return r_ * r_ * 3.1415926; }
};
```

Vererbung
Abstrakte Klassen

Anwendung zur Flächenberechnung:

```
// main.cpp
#include <iostream>
using namespace std;
#include "shape.h"

int main(int, char**) {
  Shape *shapes[3];
  double sum = 0.0;

  shapes[0] = new Rectangle(2,6,2,10);
  shapes[1] = new Circle(0,0,1);
  shapes[2] = new Rectangle(3,4,5,5);

  for (int i=0; i<3; ++i)
    sum += s[i]->area();

  cout << "area: " << sum << endl;

  for (int i=0; i<3; ++i)
    shapes[i]->move(10, -4);
}
```

```
48.1416
```

Vererbung
Abstrakte Klassen

- Abstrakte (rein virtuelle) Methoden einer abstrakten Klasse **A** haben in der Regel keine Definition, können aber eine besitzen! Dadurch wird aber nicht die Tatsache verändert, dass die Klasse **A** abstrakt bleibt, und von ihr keine Instanzen gebildet werden können.
- Wenn eine Klasse **A** von einer abstrakten Klasse **B** erbt, ist auch die Klasse **A** abstrakt, es sei denn, sie definiert alle rein virtuellen Methoden der Basisklasse.

Zur Wiederholung die Regel:

- Wenn von einer Klasse A abgeleitet wird, dann sollten alle Methoden, die überschrieben werden, als `virtual` deklariert sein!
- Zusätzlich sollte der Destruktor als `virtual` deklariert werden!

- Dazu ein letztes Beispiel:

Vererbung
Abstrakte Klassen

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

class A {
    int iwert;
  public:
    A(int iw) : iwert(iw) { }
     virtual ~A() { cout << "A Destruktor" << endl; }
};

class B : public A {
    double dwert;
  public:
    B(int iw, double dw) : A(iw), dwert(dw) { }
    ~B() { cout << "B Destruktor" << endl; }
};
```

```
int main() {
  A *mya  = new A(42);
  B *myb1 = new B(42, 1.1);
  A *myb2 = new B(43, 1.2);

  cout << sizeof(*mya) << endl;
  cout << sizeof(*myb1) << endl;
  cout << sizeof(*myb2) << endl;

  cout << "Lösche mya:" << endl;
  delete mya;
  cout << "Lösche myb1:" << endl;
  delete myb1;
  cout << "Lösche myb2:" << endl;
  delete myb2;
}
```

Ausgabe mit **virtual**:

```
8
16
8
Lösche mya:
A Destruktor
Lösche myb1:
B Destruktor
A Destruktor
Lösche myb2:
B Destruktor
A Destruktor
```

Ausgabe ohne **virtual**:

```
4
12
4
Lösche mya:
A Destruktor
Lösche myb1:
B Destruktor
A Destruktor
Lösche myb2:
A Destruktor
```

fehlt rechts!
nur der A-Teil wird zerstört!

Vererbung
Probleme bei der Modellierung mit Vererbung

- Vererbung bedeutet eine 'ist-ein'-Beziehung.
- Barbara Liskov hat die Beziehung zwischen Basisklasse **T** und Unterklasse **S** so formuliert ('Liskovsches Substitutionsprinzip'):

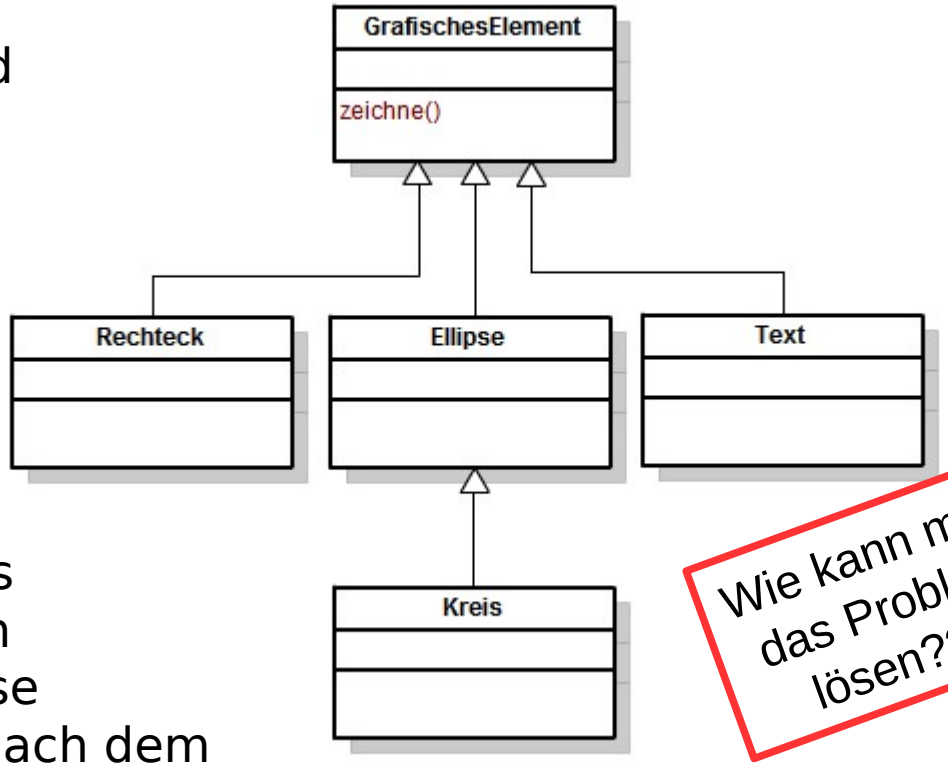
  Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S where S is a subtype of T.

  "A Behavioral Notion of Subtyping, Barbara Liskov und Jeannette Wing

- Übersetzt: Eine abgeleitete Klasse enthält alle Eigenschaften der Oberklasse. Dementsprechend kann die abgeleitete Klasse überall da eingesetzt werden, wo die Oberklasse erwartet wird.
- Konsequente Formulierung der 'ist-ein'-Beziehung...
- `public`-Ableitungen geben nur Sinn, wenn diese Bedingung gilt!
- Aber wie sieht das bei folgendem Beispiel aus:

Vererbung
Probleme bei der Modellierung mit Vererbung

- Seit Euklid ist bekannt, dass ein Quadrat ein Rechteck und ein Kreis eine Ellipse ist...
- Dieser Sachverhalt ist leicht durch Vererbung darstellbar (siehe Abb.):
- Aber: Eine Ellipse hat z.B. Methoden `setAxis1(...)` und `setAxis2(...)`, die von Kreis geerbt werden. Wenn nun ein Kreis an die Stelle einer Ellipse gesetzt würde, könnte man nach dem *Liskovschen Substitutionsprinzip* auch auf dem Kreis diese Methoden aufrufen → **nicht** gut...

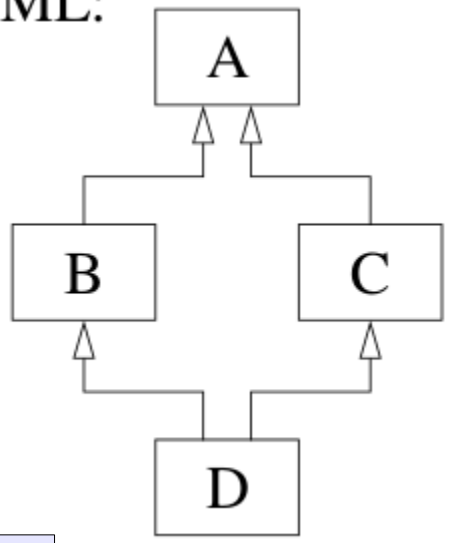


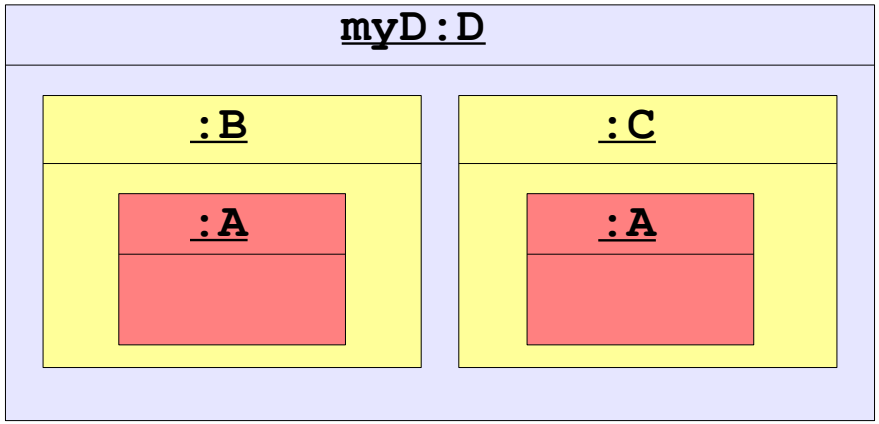
- Grundproblem: Programmiertechnisch ist ein '*ist-ein*' immer eine Spezalisierung oder Erweiterung. Mathematisch und umgangssprachlich kann es auch eine Einschränung oder Verminderung sein...

Vererbung
Mehrfachvererbung

- In C++ kann eine Klasse von mehreren Basisklassen erben → Mehrfachvererbung
- Dargestellt ist hier ein Spezialfall, das sog. '*Diamond-Problem*', bei dem beide Basisklassen wieder von einer gemeinsamen Klasse erben...
- Mehrfachvererbung kann bei der Modellierung hilfreich sein, verursacht aber auch einige Probleme, weshalb sie in manchen Sprachen (z.B. Java) nicht umgesetzt wird...
- Welches Objekt enthält welche Objekte? Graphisch:

UML:





Vererbung
Mehrfachvererbung

- Erster offensichtlicher Nachteil: Ein **D**-Objekt enthält 2 **A**-Instanzen!
- Und: Es existieren Namenskonflikte:

```
class B {
  public: void foo();
};

class C {
  public: void foo();
};

class D : public B, public C  { };

int main() {
  D myD;
  myD.foo(); // Error: 'foo' is ambiguous
  myD.B::foo();  // OK
  myD.C::foo();  // OK
}
```

```
class A {
  public: void foo();
};

class B : public A { };

class C : public A { };

class D : public B, public C  {
};

int main() {
  D myD;
  myD.foo(); // Error: 'foo' is ambiguous
  myD.B::foo(); // OK!
}
```

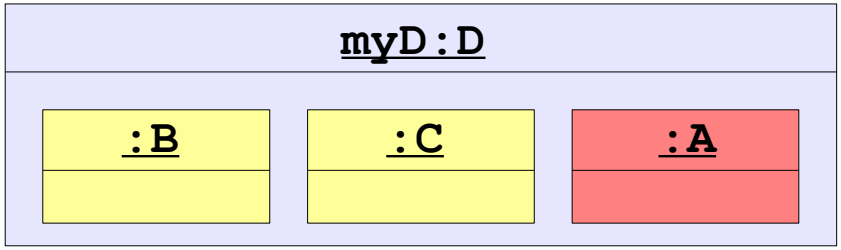
- Basisklassen-Zeiger funktionieren nicht mehr:

```
int main() {
  D myD;
  A *ptr = &myD; // 'A' is an ambiguous base of 'D'
  A *ptr = (B*) &myD; // OK, ptr zeigt auf A-Teil von B
}
```

Vererbung
Mehrfachvererbung

- Verbesserung der Probleme durch *virtuelle Basisklassen*, die die Duplizierung von Basis-Objekten verhindern:

```
class A {
  public: void foo();
};

class B : virtual public A { };

class C : virtual public A { };

class D : public B, public C  {
};

int main() {
  D myD;
  myD.foo(); // OK !!!
}
```



- Bei der Übergabe von Konstruktor-Parametern ist bei virtuellen Basisklassen allerdings zu beachten, dass die Initialisierung genommen wird, die '*am Ende*' der Vererbungshierarchie angegeben wird! Andere Initialisierungen werden *ignoriert*!

- Beispiel:

Vererbung
Mehrfachvererbung

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

class A {
  public: void foo();
          A(int i) { cout << i << endl;}
};

class B : virtual public A {
  public: B(int i) : A(42) { }  // A-Konstruktor wird ignoriert!!
};

class C : virtual public A {
  public: C(int i) : A(43) { }  // A-Konstruktor wird ignoriert!!
};

class D : virtual public B, virtual public C {
  public: D(int i) : B(i), C(i), A(i) { }  // A-Konstruktor muss!! angegeben werden
};

int main() {
  D myD(9);   // Gibt '9' aus !!!
}
```

- Zusammengefasst: Mehrfachvererbung sollte möglichst nicht verwendet werden. Falls doch, sollten virtuelle Basisklassen eingesetzt werden.

Vererbung
Standard-Typumwandlungs-Operatoren

- Grundsätzlich gilt: Eine explizite Typumwandlung (engl. *cast*) ist häufig ein Zeichen für schlechtes SW-Design, und sollte vermieden werden.
- Bisherige C-Syntax: **`(neuer Typ)Ausdruck`**, z.B. `int i=(int)3.141;`
- In C++ gibt es eine neue, bessere Syntax:

`static_cast<Typ>(Ausdruck)`

Typumwandlung zur **Übersetzungszeit**:

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

int main() {
  A myA; // Basis-Objekt
  B myB; // abgeleitetes Objekt
  A *aptr;
  B *bptr = &myB; // OK
  aptr = bptr;    // OK! Basis-Zeiger auf B Objekt

  bptr = aptr;    // Fehler !!!
  bptr = (B *)aptr;  // gefährlicher C-Stil
  bptr = static_cast<B *>(aptr);  // C++ Stil. OK wenn aptr wirklich auf ein
                                  // B-Objekt zeigt!
}
```

Vererbung
Standard-Typumwandlungs-Operatoren

`dynamic_cast<Typ>(Ausdruck)`

Typumwandlung zur **Laufzeit**, falls die Typumwandlung nicht schon zur Übersetzungszeit bestimmt werden kann.

- der Typ muss ein Zeiger oder eine Referenz auf eine Klasse sein.
- Falls der Ausdruck ein Zeiger ist und nicht auf ein passendes Objekt zeigt, wird ein Null-Zeiger `(Typ *)0` zurückgegeben.
- Falls das Argument eine nicht-passende Referenz ist, wird eine `bad-cast` Ausnahme (engl. *exception*) ausgeworfen.

```
#include <iostream>
using namespace std;
class A {
  public:
    virtual void foo() {
      cout << "A::foo()" << endl;
    }
};
class B : public A {
  public:
    void foo() {
      cout << "B::foo()" << endl;
    }
};
```

```
class C : public A {
  public:
    void foo() {
      cout << "C::foo()" << endl;
    }
};
int main() {
  A *a1[4]={new B, new C, new B, new C};

  for (int i=0; i < 4; i++) {
    A *ptr = dynamic_cast<C*>( a1[i] );
    if (ptr) {  ptr->foo(); }
  }
}
```

```
C::foo()
C::foo()
```

Vererbung
Standard-Typumwandlungs-Operatoren

`const_cast<Typ>(Ausdruck)`

Typumwandlung zur **Übersetzungszeit**, die die `const`-Eigenschaft eines Ausdrucks verändert.

- Ersetzt die früher übliche Form `(T)Obj`, die eine erzwungene Typumwandlung eines beliegiben Datentyps nach `T` bewirkt.
- Ausdruck ist ein Zeiger oder eine Referenz.
- Kann verwendet werden, um die `const`-Eigenschaft zu entfernen oder hinzuzufügen.
- Sollte mit Vorsicht eingesetzt werden, da `const` seinen Sinn hat...

```
int main() {
  const int i = 42;
  int & ir1 = i;  // Fehler! ir ist nicht const!
  int & ir2 = const_cast<int &>(i); // entferne const → OK!

  int j = 100;
  const int & ir3 = const_cast<const int &>(j);
  const int & ir4 = j; // Funktioniert auch!!
  ir3 = 101;      // Fehler! ir3 ist const!
}
```

Vererbung
Standard-Typumwandlungs-Operatoren

`reinterpret_cast<Typ>(Ausdruck)`

Typumwandlung zur **Übersetzungszeit**, die außer der `const`-Eigenschaft eines Ausdrucks praktisch jede andere Typumwandlung durchführen kann.

- 're-interpretiert' den Datentyp

- sollte nur in ganz selten Fällen eingesetzt werden!

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

int main() {
  float d = 3.14159;

  int   i = d;                              // implizite Umwandlung float → int
  int *pi = reinterpret_cast<int *>(&d); // float* → int* Konvertierung
  cout << i << endl << hex << *pi << endl;  // Gibt als zweite Ausgabe die
                                            // Hexadezimale Darstellung von
                                            // 3.14159 aus.
}
```

```
3
40490fd0
```

Vererbung
Typinformationen zur Laufzeit

- **dynamic_cast<>** wandelt erst zur Laufzeit Zeiger und Referenzen um und führt entsprechende Überprüfungen durch.
- Allgemeiner kann mit der Methode **typeid()** aus **<typeinfo>** ein Objekt vom Typ **type_info&** bestimmt werden, das z.B. auch mit dem ==-Operator verglichen werden kann. **type_info** enthält auch eine Methode **const char* name() const;** , mit der der Name des Typs ausgegeben werden kann!

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

struct Base {};
struct Derived : Base {};

int main() {
  // built-in types:
  int i;
  int * pi;
  cout << "int is: " << typeid(int).name() << endl;
  cout << "  i is: " << typeid(i).name() << endl;
  cout << " pi is: " << typeid(pi).name() << endl;
  cout << "*pi is: " << typeid(*pi).name() << endl << endl;
}
```

```
int is: i
  i is: i
 pi is: Pi
*pi is: i
```

Vererbung
Typinformationen zur Laufzeit

```
#include <iostream>
#include <typeinfo>
using namespace std;
struct Base {};
struct Derived : Base {};
struct Poly_Base {virtual void Member(){}};
struct Poly_Derived: Poly_Base {};

int main() {
  // non-polymorphic types:
  Derived derived;
  Base* pbase = &derived;
  cout << "derived is: " << typeid(derived).name() << endl;
  cout << " *pbase is: " << typeid(*pbase).name() << endl;
  cout << boolalpha << "same type? ";
  cout << ( typeid(derived)==typeid(*pbase) ) << endl << endl;

  // polymorphic types:
  Poly_Derived polyderived;
  Poly_Base* ppolybase = &polyderived;
  cout << "polyderived is: " << typeid(polyderived).name()
       << endl;
  cout << " *ppolybase is: " << typeid(*ppolybase).name()
       << endl;
  cout << boolalpha << "same type? ";
  cout << ( typeid(polyderived)==typeid(*ppolybase) )
       << endl << endl;
}
```

```
derived is: 7Derived
 *pbase is: 4Base
same type? false

polyderived is:
12Poly_Derived
 *ppolybase is:
12Poly_Derived
same type? true
```

Vererbung
`using`-Schlüsselwort im Vererbungskontext

- `using` wird im Kontext von Namensräumen verwendet, aber auch um die Sichtbarkeit von Attributen und Methoden zu verändern!

```
class Basis {
  protected:
   int i;
   void foo() { };
};

class Abgeleitet : public Basis {
  public:
    using Basis::foo; // foo wird ab jetzt public
    using Basis::i;   // i wird ab jetzt public
};

int main()
{
  Basis b;
  Abgeleitet a;
  b.foo(); // Fehler: protected!
  a.foo(); // OK: public!
  a.i = 42;// OK: public!
}
```

Fehlerbehandlung

- Bei der Behandlung von Fehlersituationen in Programmen gibt es verschiedene Strategien:

1.) Sofortiger Programmabbruch (z.B. wie bei **`assert()`** oder **`exit()`**)

2.) Übergabe eines Parameters per Referenz und Überprüfung auf einen Fehlerwert beim Aufrufer.

3.) Setzten der aus der C-Welt bekannten globalen Fehlervariable `errno`

4.) Aufruf einer speziellen Fehlerbehandlungs-Funktion.

5.) 'Kopf in den Sand' – Fehler einfach ignorieren...

- Die Ansätze 2 und 4 sind denkbar. Zusätzlich bietet C++ noch das Konzept der Ausnahmebehandlung (engl. *exception handling*) an.

- Diese Methode entlastet den Programmcode von vielen Zeilen zur Fehlerüberprüfung, und trennt deutlich den Anwendungs- und Fehlerbehandlungscode.

Fehlerbehandlung

- Die grundsätzliche Strategie ist folgende:
  1. Eine Funktion versucht (englisch **try**) die Erledigung einer Aufgabe.
  2. Wenn sie einen Fehler feststellt, den sie nicht beheben kann, wirft (englisch **throw**) sie eine Ausnahme (englisch exception) aus.
  3. Die Ausnahme wird von einer Fehlerbehandlungsroutine aufgefangen (englisch **catch**), die den Fehler bearbeitet.

Im Unterschied zur 4. Strategie wird nicht in die aufrufende Funktion zurückgesprungen, sondern die Ausführung im Fehlerfall bei der Fehlerbehandlungsroutine fortgesetzt!

Hinweise:

- `throw` kann Objekte eines beliebigen Datentyps zuwerfen
- Beim Verlassen des `try`-Blocks werden (wie auch sonst) alle lokalen Variablen abgebaut (und ggd. der Destruktor aufgerufen)
- `...` bei catch bedeutet: bel. Objekt(e)

```
void func() {
  if (fehler)
    throw <Objekt>;
}

try {
  // Code ...
  func();
}
catch(Typ & name) {
 // Fehlerbeh.
}
catch(...) { // default-handler }
```

Fehlerbehandlung

- Die catch-Parameter (es können mehrere sein!) werden sinnvollerweise als Referenz übergeben, um Kopien der Objekte zu vermeiden.
- Innerhalb eines `catch`-Blockes kann mit `throw;` die Ausnahme an die höhere Behandlungseben weitergeworfen werden.

Deklarationen und throw

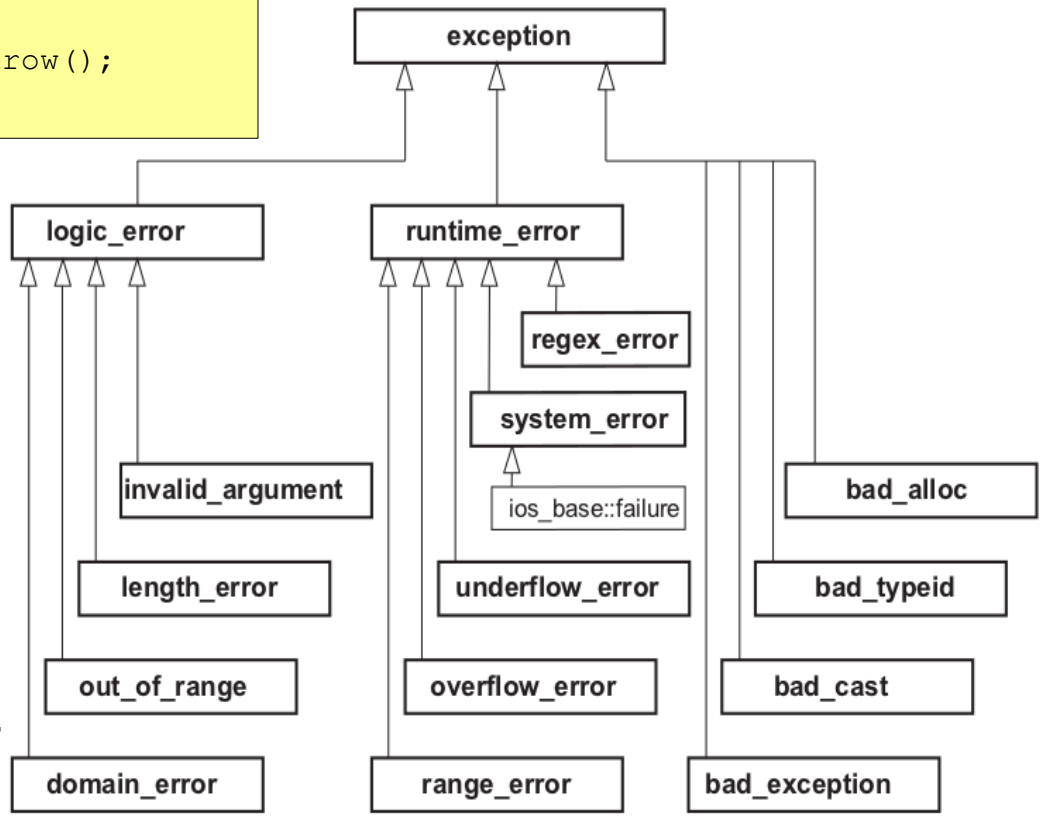
- Bei der Deklaration von Methoden/Funktionen kann angegeben werden, ob die Methode/Funktion Ausnahmen erzeugt:
  1. `void foo();`
     → kann beliebige Ausnahmen auswerfen.
  2. `void foo() throw(const char*);`
     → verspricht, nur die deklarierten Ausnahmen auszuwerfen.
  3. `void foo() throw();`
     → verspricht, *keine* Ausnahmen auszuwerfen.
- Die von `throw` 'geworfenen' Objekte können von der Klasse `std::exception` (aus `<exception>`) abgeleitet werden.
- Dadurch können die Standard-Methoden dieser Basisklasse verwendet werden.

Fehlerbehandlung

Schnittstelle von exception:

```
class exception {
  public:
    exception() throw();
    exception(const exception&) throw();
    exception& operator=(const exception&) throw();
    virtual ~exception() throw();
    virtual const char * what() const throw();
};
```

- Die Methode `what()` liefert eine Fehlerbeschreibung als C-String zurück.
- Es existieren diverse vordefinierte Ausnahmen, die benutzt werden können
- Es exitieren i.d.R. Konstruktoren, die einen `const string &` als Parameter haben!



Fehlerbehandlung

Klasse	Bedeutung	Header
`exception`	Basisklasse	`<exception>`
`logic_error`	theoretisch vermeidbare Fehler, zum Beispiel Verletzung von logischen Vorbedingungen	`<stdexcept>`
`invalid_argument`	ungültiges Argument bei Funktionen	`<stdexcept>`
`length_error`	Fehler in Funktionen der Standard-C++-Bibliothek, wenn ein Objekt erzeugt werden soll, das die maximal erlaubte Größe für dieses Objekt überschreitet	`<stdexcept>`
`out_of_range`	Bereichsüberschreitungsfehler	`<stdexcept>`
`domain_error`	anderer Fehler des Anwendungsbereichs	`<stdexcept>`
`runtime_error`	nicht vorhersehbare Fehler, zum Beispiel datenabhängige Fehler	`<stdexcept>`
`regex_error`	Fehler bei regulären Ausdrücken	`<regex>`
`system_error`	Fehlermeldung des Betriebssystems	`<system_error>`
`range_error`	Bereichsüberschreitung	`<stdexcept>`
`overflow_error`	arithmetischer Überlauf	`<stdexcept>`
`underflow_error`	arithmetischer Unterlauf	`<stdexcept>`
`bad_alloc`	Speicherzuweisungsfehler	`<new>`
`bad_typeid`	falscher Objekttyp	`<typeinfo>`
`bad_cast`	Typumwandlungsfehler	`<typeinfo>`
`bad_exception`	Verletzung der Exception-Spezifikation (siehe `unexpected()`	`<exception>`

Fehlerbehandlung

Besondere Fehlerbehandlungsfunktionen

- Bei der Fehlerbehandlung können selber wieder Fehler auftreten. Dazu sind in `<exception>` folgende Methoden deklariert:

`terminate()`

Die Standardimplementierung von `void terminate()` beendet das Programm. Die Funktion `terminate()` wird (unter anderem) aufgerufen, wenn

- der Exception-Mechanismus keine Möglichkeit zur Bearbeitung einer geworfenen Exception findet;
- der vorgegebene `unexpected_handler()` gerufen wird;
- ein Destruktor während des Aufräumens (engl: *stack unwinding*) eine Exception wirft oder wenn
- ein statisches (nichtlokales) Objekt während der Konstruktion oder Zerstörung eine Exception wirft.

Fehlerbehandlung

`unexpected()`

Die Funktion void `unexpected()` wird aufgerufen, wenn eine Methode ihre Versprechungen nicht einhält und eine Exception wirft, die in der Exception-Spezifikation nicht aufgeführt wird. `unexpected()` kann nun selbst eine Exception auslösen, die der verletzten Exception-Spezifikation genügt, sodass die Suche nach dem geeigneten Exception-Handler weitergeht. Falls die ausgelöste Exception nicht der Exception-Spezifikation entspricht, sind zwei Fälle zu unterscheiden:

1. Die Exception-Spezifikation der Methode führt `bad_exception` auf: Die geworfene Exception wird durch ein `bad_exception`-Objekt ersetzt.
2. Die Exception-Spezifikation der Methode führt `bad_exception` nicht auf:
   `terminate()` wird aufgerufen.

Fehlerbehandlung

`uncaught_exception()`

Die Funktion **`bool uncaught_exception()`** gibt **`true`** nach der Auswertung einer Exception zurück, bis die Initialisierung im Exception-Handler abgeschlossen ist oder **`unexpected()`** wegen der Exception aufgerufen wurde. Ferner wird **`true`** zurückgegeben, wenn **`terminate()`** aus irgendeinem Grund (außer einem expliziten Aufruf) begonnen wurde.

- Die Handler für **`terminate()`** und **`unexpected()`** können selber definiert werden:

```
typedef void ( * unexpected_handler)();
typedef void ( * terminate_handler)();
unexpected_handler set_unexpected(unexpected_handler f) throw();
terminate_handler set_terminate(terminate_handler f) throw();
```

- Ein **`unexpected_handler`** soll **`bad_exception`** oder eine der Exception-Spezifikation genügende Exception werfen oder **`terminate()`** rufen oder das Programm mit **`abort()`** oder **`exit()`** beenden.
- Ein **`terminate_handler`** soll das Programm ohne Rückkehr an den Aufrufer beenden.

Fehlerbehandlung

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

void my_unexpected_handler() throw(logic_error) {
  cout << "my_unexpected_handler() called" << endl;
  throw logic_error("a logic error");
}
void my_terminate_handler() throw(logic_error) {
  cout << "my_terminate_handler() called" << endl;
}
void foo() throw(range_error, bad_exception) {
  cout << "foo() called" << endl;
  throw logic_error("hallo");  // nicht in Deklaration!!
}
int main() {
  set_unexpected(my_unexpected_handler);
  set_terminate (my_terminate_handler);
  try {
    foo();
  }
  catch (exception & e) {
    cout << "In catch Block..." << endl;
    cout << e.what() << endl;
  }
  foo();
}
```

```
foo() called
my_unexpected_handler() called
In catch Block...
std::bad_exception
foo() called
my_unexpected_handler() called
my_terminate_handler() called
Abgebrochen
```

Kombination von assert und Exceptions

- Zur Überprüfung von Bedingungen (Vor-, Nach- und invariante Bedingungen) kann das **assert**-Makro eingesetzt werden. Allerdings hat dieses den Nachteil, dass im Fehlerfall das Programm einfach abgebrochen wird.
- Falls eine Fehlerbehandlung realisiert werden soll, kann z.B. ein eigenes Makro unter Verwendung von Exceptions geschrieben werden
- Wie beim Original soll kein Code erzeugt werden, wenn Präprozessor-variable **NDEBUG** gesetzt ist.

```
// Datei assertex.h
#ifndef ASSERTEX_H
  #define ASSERTEX_H
  #ifdef NDEBUG
    #define Assert(bedingung, ausnahme) ; // Leeranweisung
  #else
    #define Assert(bedingung, ausnahme) \
    if(!(bedingung)) throw ausnahme
  #endif
#endif // ASSERTEX_H
```

Speicherbeschaffung und Exceptions

- In C muss bei jedem **malloc**-Aufruf überprüft werden, ob der Rückgabewert im Falle von Speichermangel **NULL** ist.
- In C++ erzeugt der **new**-Operator eine **bad_alloc** Ausnahme. Das früher übliche Verhalten der Rückgabe von **0** kann durch ein **nothrow**-Argument erreicht werden: **T *p = new(nothrow) T;**
- In Pseudocode funktioniert der new-Operator etwa folgermaßen:

```
void * ergebnis;
do {
  ergebnis = beschaffe_irgendwie_Speicher();
  if(ergebnis == NULL) { // d.h. nicht erfolgreich
    if(new_handler == NULL) {  // d.h. nicht definiert
      throw bad_alloc();
    }
    else {
      new_handler();
    }
  }
} while(ergebnis == NULL);
return ergebnis;
```

- Diese Schleife wird erst beendet, wenn
  1. Speicher erfolgreich beschafft wurde, oder
  2. Speichermangel vorliegt, und kein **new_handler** definiert wurde, oder
  3. Speichermangel vorliegt, und der **new_handler** nicht zurückkehrt.

Speicherbeschaffung und Exceptions

- Dabei ist der **new_handler** eine Funktion, die vom Benutzer gesetzt werden kann. Diese Funktion wird typischerweise entweder
  - Speicher freigeben und zurückkehren,
  - 'aufräumen' und das Programm beenden, oder
  - selber eine Ausnahme erzeugen, die z.B. von einem übergeordneten **catch**-Block aufgefangen wird.

```
#include<iostream>
#include<new> // set_new_handler() und bad_alloc
using namespace std;
const unsigned int BLOCKGROESSE = 64000,
                   MAXBLOECKE = 50000;

int * reserveSpeicher = 0;

void speicherfehler() throw (bad_alloc) {
  if(reserveSpeicher) {  // Falls noch Reserven vorhanden, diese freigeben...
    cerr << "Einmal Platz schaffen !\n" ;
    delete [] reserveSpeicher;
    reserveSpeicher = 0; // deaktivieren
  }
  else {
    cerr << "Exception auslösen !\n" ;
    throw bad_alloc();
  }
}
```

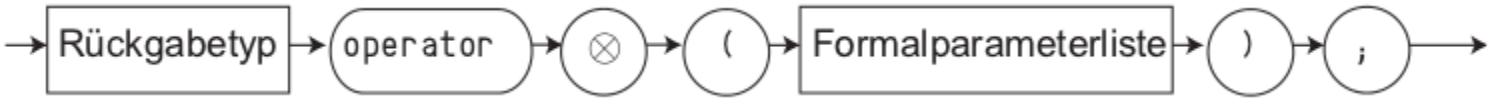
Speicherbeschaffung und Exceptions

```
int main() {

  // eigene Fehlerbehandlungsfunktion
  set_new_handler(speicherfehler);

  // Reserve beschaffen
  reserveSpeicher = new int[10 * BLOCKGROESSE];

  int * ip[MAXBLOECKE] = {0};
  unsigned int blockNr = 0;  // Zähler für Blöcke
  try {
    while(blockNr < MAXBLOECKE) { // Speicher fressen
      ip[blockNr] = new int[BLOCKGROESSE];
      cout << "Block " << blockNr++ << endl;
    }
    if(blockNr == MAXBLOECKE-1) { // Zählung ab 0
      cout << "Block-Array gefüllt" << endl;
    }
  }
  catch(const bad_alloc& exc) {
    cerr << ++blockNr
         << " Blöcke beschafft\n"
         << " bad_alloc ausgeworfen ! Grund: "
         << exc.what() << endl;
  }
}
```

```
Block 0
Block 1
Block 2
…
Block 12455
Block 12456
Block 12457
Einmal Platz schaffen !
Block 12458
Block 12459
Block 12460
Block 12461
Block 12462
Block 12463
Block 12464
Block 12465
Block 12466
Exception auslösen !
12468 Blöcke beschafft
 bad_alloc ausgeworfen !
Grund: std::bad_alloc
```

Überladen von Operatoren

- Den vordefinierten Operatorsymbolen in C++ kann man für Klassen neue Bedeutungen zuordnen!
- Dazu werden (ganz ähnlich wie bei Methoden) diese Operatoren *überladen*.
- Ein Beispiel ist der `<<`-Operator, der neben seiner Ursprungsbedeutung (Links-Shift) für die Klasse `ostream` und verschiedene Parametertypen überladen ist.
- Deklaration eines Operators (der Methodenname besteht aus dem Schlüsselwort operator und dem angehängten Operatorzeichen):



- Der Compiler wandelt die Benutzung eines Operators in die entsprechende Methode um:

aus `Matrix a,b,c;` `a = b + c;`
wird `a.operator=( b.operator+(c) );`

Überladen von Operatoren

- Operator-Methoden können entweder Element-Methoden einer Klasse sein, oder als globale Funktion definiert werden:

Element-Funktion	Syntax	Ersetzung durch
nein	`x ⊗ y`	`operator⊗(x,y)`
	`⊗ x`	`operator⊗(x)`
	`x ⊗`	`operator⊗(x,0)`
ja	`x ⊗ y`	`x.operator⊗(y)`
	`⊗ x`	`x.operator⊗()`
	`x ⊗`	`x.operator⊗(0)`
	`x = y`	`x.operator=(y)`
	`x(y)`	`x.operator()(y)`
	`x[y]`	`x.operator[](y)`
	`x->`	`(x.operator->())->`
	`(T)x`	`x.operator T()`
	`static_cast<T>(x)`	`x.operator T()`
	`new T`	`T::operator new(sizeof(T))`
	`delete p`	`T::operator delete(p)`
	`new T[n]`	`T::operator new[](n * sizeof(T))`
	`delete[] p`	`T::operator delete[](p)`

`T` ist Platzhalter für einen Datentyp, `p` ist vom Typ `T*`.

Überladen von Operatoren

- Die Element-Operatormethoden haben immer einen Parameter weniger als die entsprechenden globalen Varianten, da eine Element-Operatormethode automatisch auf dem 'eigenen' Objekt (als linke Seite des Ausdrucks) aufgerufen wird.

- In der vorherigen Tabelle kann man den Platzhalter ersetzen durch:
  - unäre Operatoren: `+ - * & ~ ! -> ++ -- (++ --` auch 'post'`)`
  - binäre Operatoren: `+ - * / % << >> & | ^`
    `+= -= *= /= %= <<= >>= &= |= ^=`
    `== != < <= > >= && || ,` (Komma) `->*`

<u>Folgende Einschränkungen gelten:</u>

- Die Operatoren `. .* :: ?: typeid sizeof throw` und die C++ Typ-umwandlungsoperatoren dürfen nicht überladen werden.
- Es dürfen keine neuen Operatoren erzeugt werden.
- Die Anzahl der Operanden, die Priorität und Assoziativität der einzelnen Operatoren ist in der Sprache festgelegt und kann nicht verändert werden!
- Mindestens ein Operand muss ein nutzerdefinierter Datentyp sein.

Überladen von Operatoren

Beispiel: Rationale Zahlen

- Gewünscht wäre folgender Programmcode:

```
Rational a,b,c;
cin >> a;          // überladener Operator
cin >> b;          // überladener Operator
c = a + b;         // überladener Operator
cout << c;         // überladener Operator
```

- Möglicher Element-Operator für die Addition:

```
class Rational {

const Rational operator+(const Rational & rhs)
  return Rational( getZaehler() * rhs.getNenner() +
                   getNenner()  * rhs.getZaehler(),
                   getNenner()  * rhs.getNenner() );
};
```

- Aber wie sieht es aus mit Ausdrücken der Form:

```
c = a + 3;         // oder
c = 3 + a;
```

→ `3.operator(a)` funktioniert nicht auf einer Integer-Zahl...

Überladen von Operatoren

- Lösung mit globalem Operator und entsprechenden Versionen mit Integer-Parameter:

```
class Rational {
};

const Rational operator+(const Rational & lhs, const Rational & rhs);
const Rational operator+(long lhs, const Rational & rhs) {
  return Rational(lhs, 1) + rhs;
}
const Rational operator+(const Rational & lhs, long rhs) {
  return lhs + Rational(rhs, 1);
}
```

- Wenn ein Typumwandlungskonstruktor `long` → `Rational` existiert, dann können die zwei letzten Funktionen weggelassen werden, weil nun der Compiler einen Integer automatisch in ein `Rational`-Objekt umwandelt!

```
class Rational {
  Rational() { … }
  Rational(long l) { … }
};

const Rational operator+(const Rational & lhs, const Rational & rhs);
```

Überladen von Operatoren

- Eine gute Strategie ist es, zunächst die Kurzformoperatoren (**+=**, **-=**, u.s.w.) zu implementieren, da sich die Umsetzung der zugehörigen normalen Operatoren dann deutlich einfacher umsetzen lässt:

```
class Rational {
  const Rational & operator+=(const Rational & rhs) {
    zaehler = zaehler * rhs.nenner + nenner * rhs.zaehler;
    nenner  = nenner  * rhs.nenner;
    return *this;
  }
};

const Rational operator+(const Rational & lhs, const Rational & rhs) {
  return Rational(lhs) += rhs;   // Ruft den Kopierkonstruktor und
                                 // den += Operator auf!
}
```

Der Ausgabeoperator

- Es wäre möglich, den **<<**-Operator als Elementmethode zu definieren:

```
class Rational {
  ostream & operator<<(ostream & stream) {
    stream << zaehler << "/" << nenner << endl;
    return stream;
  }
};
```

Überladen von Operatoren

- Die Benutzung des Operators sähe dann allerdings so aus (der Operator wird auf dem **Rational**-Objekt aufgerufen):

```
Rational a(3,4);
a << cout;
```

- Daher wird der Ausgabeoperator immer als globaler Operator mit 2 Parametern formuliert!
- Der Operator liefert eine Referenz auf den verwendeten **ostream** zurück, um eine Verkettung zu ermöglichen:

```
class Rational {
  …
};

ostream & operator<<(ostream & stream, const Rational & rhs) {
  stream << rhs.getZaehler() << "/" << rhs.getNenner() << endl;
  return stream;
}

cout << a << b; // Entspricht (cout << a) << b;
```

- Falls nötig, kann der Ausgabeoperator als **friend** der Klasse angegeben werden. Dann hätte er direkten Zugriff auf die privaten Elemente.

Überladen von Operatoren

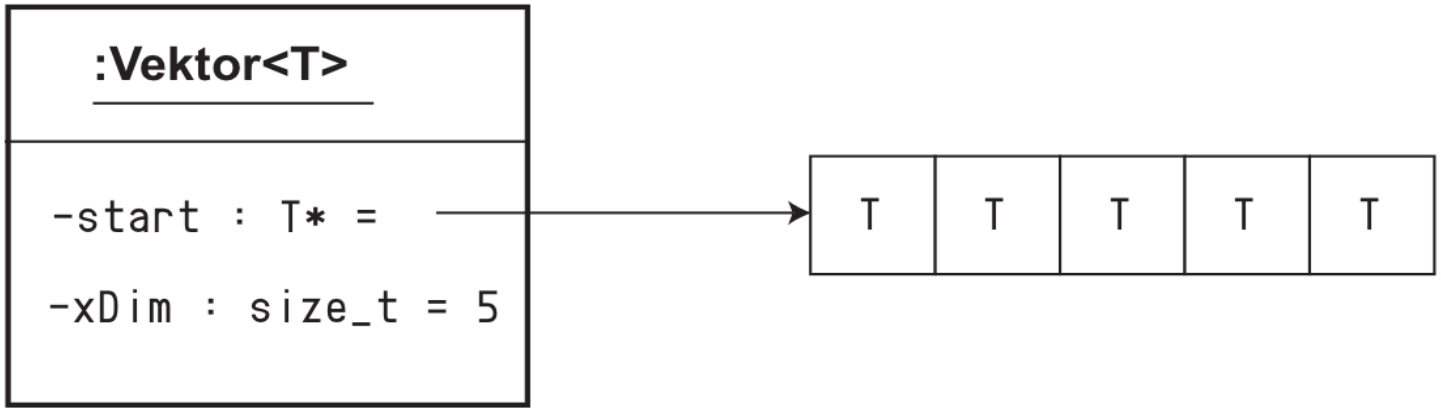
Eine Klasse für Vektoren

- Die vorgestellte Klasse Vektor soll zeigen, wie die Klasse std::vector im Prinzip arbeitet. Die Unterschiede zu einem C-Array sind:

  Der Zugriff über den Index-Operator `[]` ist sicher (wie mit `at()` beim `std::vector`

  - Es gibt einen Zuweisungsoperator (`Vektor v1, v2;  v1 = v2;`)
  - Die Klasse kann leicht um weitere Funktionen (Skalarprodukt, Addition, etc.) erweitert werden.

Objektdiagramm für ein Objekt der Klasse `Vektor`:



Überladen von Operatoren

```
// dynamische Vektor-Klasse
#ifndef VEKTOR_T
#define VEKTOR_T
#include<stdexcept>

template<typename T>
class Vektor {
  public:
    Vektor(size_t x = 0);                  // Allg. Konstruktor 1
    Vektor(size_t n, T t);                 // Allg. Konstruktor 2:
                                              // n Elemente mit Wert t
    Vektor(const Vektor<T>& v);            // Kopierkonstruktor
    virtual ~Vektor() { delete [] start;} // Destruktor

    size_t size() const {return xDim;}

    void groesseAendern(size_t);           // dynamisch ändern

    // Indexoperator inline
    T& operator[](int index) {
         if( index < 0  ||  index >= (int)xDim) {
            throw std::out_of_range("Indexüberschreitung!");
         }
         return start[index];
    }
```

Überladen von Operatoren

```
    // Indexoperator für nicht-veränderliche Vektoren
    const T& operator[](int index) const {
         if( index < 0  ||  index >= (int)xDim) {
            throw std::out_of_range("Indexüberschreitung!");
         }
         return start[index];
    }

     // Zuweisungsoperator
    Vektor<T>& operator=(const Vektor<T>&);

    // Zeiger auf Anfang und Position nach dem Ende für Vektoren
    // mit nichtkonstanten und konstanten Elementen
    T* begin() { return start;}
    T* end()   { return start + xDim;}
    const T* begin() const { return start;}
    const T* end()   const { return start + xDim;}
  private:
    size_t  xDim;                    // Anzahl der Datenobjekte
    T   *start;                      // Zeiger auf Datenobjekte
};

// … es geht gleich weiter …
```

Überladen von Operatoren

Konstruktoren:

```
template<typename T>
inline Vektor<T>::Vektor(size_t x)       //  Allg. Konstruktor 1
: xDim(x), start(new T[x]) { }           //  (hat default-Parameter)

template<typename T>
Vektor<T>::Vektor(size_t n, T wert)      //  Allg. Konstruktor 2
: xDim(n), start(new T[n]) {
     for (size_t i = 0; i < xDim; ++i) {
         start[i] = wert;
     }
}

template<typename T>
Vektor<T>::Vektor(const Vektor<T> &v)    // Kopierkonstruktor
    : xDim(v.xDim), start(new T[xDim]) {
    for (size_t i = 0; i < xDim; ++i) {
        start[i] = v.start[i];
    }
}
```

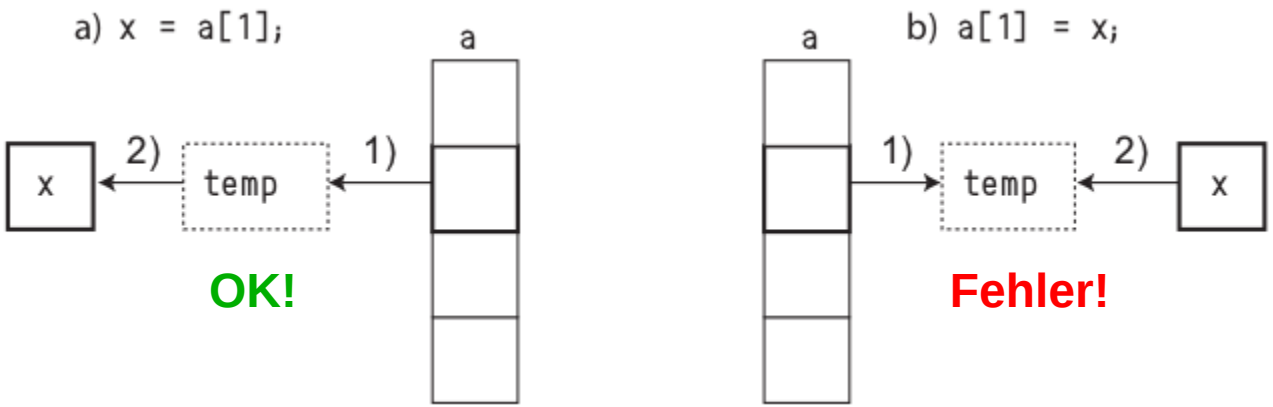
- Der Kopierkonstruktor erzeugt eine tiefe Kopie des übergebenen Objektes!
- Hinweis: Alle auf den Typ `T` ausgeführten Operatoren (z.B. `start[i]=wert;` können wieder überladene Operatoren einer Klasse sein!

Überladen von Operatoren

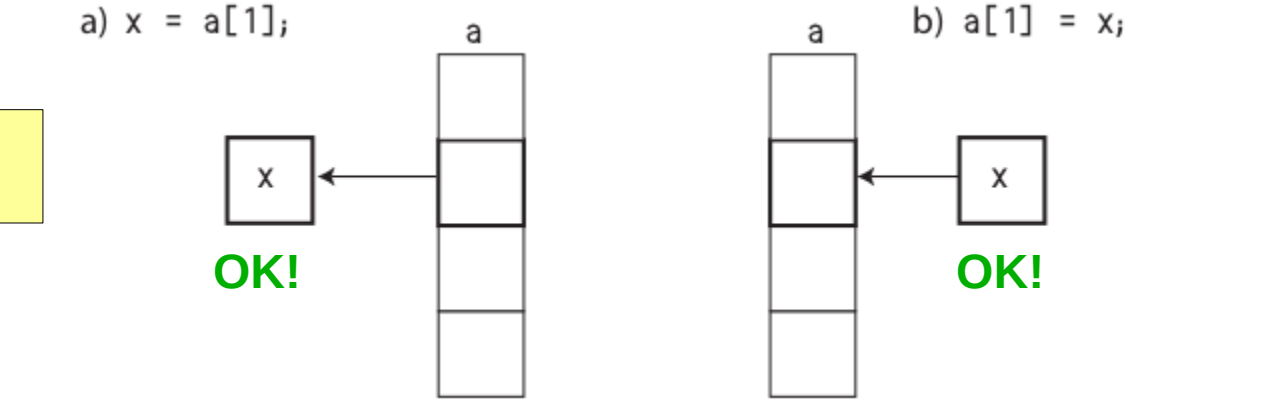
Index-Operatoren:

- Die Index-Operatoren liefern eine Referenz auf ein Objekt vom Typ `T` zurück. Warum?

```
T operator[](int i);
```




```
T & operator[](int i);
```



Überladen von Operatoren

Zuweisungs-Operator:

- Der Zuweisungsoperator muss eine tiefe Kopie erzeugen, da die Klasse dynamischen Speicher verwaltet (`T * start;`)

```
template<typename T>
Vektor<T> &Vektor<T>::operator=(const Vektor<T> &v) {
    T * temp = new T[v.xDim];   // neuen Platz beschaffen
    for (size_t i = 0; i < v.xDim; ++i) {
        temp[i] = v.start[i];
    }
    delete[] start;         // Speicherplatz freigeben
    xDim  = v.xDim;
    start = temp;
    return *this;
}
```

- Als Rückgabe wird eine Referenz auf das Aufgerufene Objekt zurückgeliefert, damit Verkettungen (`v1=v2=v3;`) möglich sind.
- Das 'verspätete' Freigeben des aktuellen Speichers verhindert Inkonsistenzen, falls `new` fehlschlägt!

Überladen von Operatoren

Zuweisungs-Operator:

- Wenn Klassen Instanzen anderer Klassen enthalten, werden ...
  - Datenelemente *elementarer Datentyen* bitweise kopiert.
  - Datenelemente *selbstdefinierter Typen* per `operator=` kopiert.

Beispiel:

- Die letzte Zeile im rechts dargestellten Codefragment bewirkt:

```
c1.A::operator=(c2.A); // Kopie des anonymen
                       // Subobjektes
c1.einB.operator=(c2.einB); // Kopie des
                            // enthaltenen B-
                            // Objektes
c1.x = c2.x;  // Kopie eines Grunddatentyps
```

```
class A;
class B;
class C : public A {

  private:
    B einB;
    int x;
};
…
C c1, c2;
c1 = c2;
```

Überladen von Operatoren

- Eine weitere Optimierung des Zuweisungsoperators könnte die Überprüfung auf Selbstzuweisung (`v1=v1`) sein. Dieser Fall könnte mit der Abfrage `if (this != &rhs) …` umgangen werden.

Dynamisches Ändern der Vektorgröße

```
template<typename T>
void Vektor<T>::groesseAendern(size_t neueGroesse) {
    // neuen Speicherplatz besorgen
    T *pTemp = new T[neueGroesse];

    // die richtige Anzahl von Elementen kopieren
    size_t kleinereZahl =
               neueGroesse > xDim ? xDim : neueGroesse;
    for (size_t i = 0; i < kleinereZahl; ++i) {
        pTemp[i] = start[i];
    }
    // alten Speicherplatz freigeben
    delete [] start;

    // Verwaltungsdaten aktualisieren
    start = pTemp;
    xDim  = neueGroesse;
}
#endif // VEKTOR_T
```

Überladen von Operatoren

Inkrement- und Dekrement-Operator ++ --

- Die unären Operatoren ++ und – gibt es in zwei Varianten:
  Prä-Inkremet/Dekrement **`++x --x`**, und
  Post-Inkrement/Dekrement **`x++ x--`**

- Beide Versionen können bei der Operatorüberladung unterschieden werden: Die *Post*-Version erhält einen zusätzlichen (Dummy)-Parameter vom Typ **`int`**:

```
class A {

  public:

  A &  operator++()    { … }  // Prä-Inkrement
  A    operator++(int) { … }  // Post-Inrement

};
```

- Die dargestellten Operatoren können auch als globale Operatoren (dann mit einem [ersten] Parameter mehr: Einer Referenz auf das betreffende Objekt) definiert werden!

Überladen von Operatoren

Beispiel: Eine Datums-Klasse

```
#ifndef DATUM_H
#define DATUM_H
#include<string>

class Datum {
  public:
    Datum();                          // Default-Konstruktor
    Datum(int t, int m, int j = 1997); // allgemeiner Konstruktor
    void set(int t, int m, int j); // Datum setzen
    void aktuell();                   // Systemdatum setzen
    bool istSchaltjahr() const;
    Datum& operator++();              // Tag hochzählen, präfix
    Datum operator++(int);            // Tag hochzählen, postfix
    int tag()   const;
    int monat() const;
    int jahr()  const;

  private:
     int tag_, monat_, jahr_;
};
```

Überladen von Operatoren

Beispiel: Eine Datums-Klasse

```
bool istSchaltjahr(int jahr); // Impl. s.u.

// inline-Methoden
inline Datum::Datum(int t, int m, int j) { set(t, m, j);}

inline int Datum::tag()   const { return tag_;  }
inline int Datum::monat() const { return monat_;}
inline int Datum::jahr()  const { return jahr_; }
inline bool Datum::istSchaltjahr() const {
  return ::istSchaltjahr(jahr_);
}

// global
bool istGueltigesDatum(int t, int m, int j);

inline bool istSchaltjahr(int jahr)  {
   return  (jahr % 4 == 0 && jahr % 100) || jahr % 400 == 0;
}

#endif
```

Überladen von Operatoren

Implementierung in der *.cpp-Datei:

```
Datum::Datum()  { aktuell();}

void Datum::set(int t, int m, int j) {
   assert(istGueltigesDatum(t, m, j));
    // besser: Exception verwenden, siehe Übungsaufgabe
    tag_ = t;
    monat_ = m;
    jahr_ = j;
}

void Datum::aktuell() {   // Systemdatum eintragen
    // time_t, time(), tm, localtime()} sind in <ctime> deklariert !
    time_t now = time(NULL);
    tm *z = localtime(&now);           // Zeiger auf struct tm
    jahr_  = z->tm_year + 1900;
    monat_ = z->tm_mon+1;              // localtime liefert 0..11
    tag_   = z->tm_mday;
}
```

Überladen von Operatoren

```
// Präfix-Version
/////////////////
Datum& Datum::operator++() {      // Datum um 1 Tag erhöhen
   ++tag_;
   // Monatsende erreicht?
   if(!istGueltigesDatum(tag_, monat_, jahr_)) {
     tag_ = 1;
     if (++monat_ > 12) {
        monat_ = 1;
        ++jahr_;
     }
   }
   return *this;
}

// Postfix-Version
//////////////////
Datum Datum::operator++(int) {         // Datum um 1 Tag erhöhen
   Datum temp = *this;
   ++*this;                            // Präfix ++ aufrufen
   return temp;
}
```

- Warum liefert die Präfix-Version eine Referenz auf `Datum` zurück, die Postfix-Version aber nicht?

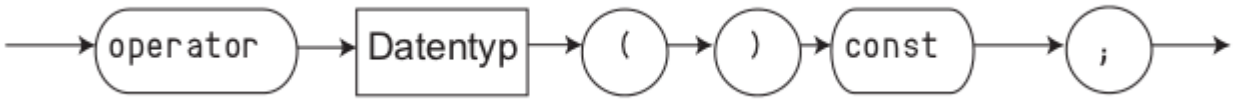
Überladen von Operatoren

```
// globale Funktion
bool istGueltigesDatum(int t, int m, int j) {
     // Tage pro Monat(static vermeidet Neuinitialisierung):
     static int tmp[]={31,28,31,30,31,30,31,31,30,31,30,31};

     tmp[1] = istSchaltjahr(j) ? 29 : 28;

     return     m >= 1    && m  <= 12
             && j >= 1583 && j  <= 2399  // oder mehr
             && t >= 1    && t  <= tmp[m-1];
}
```

Überladen von Operatoren

Typumwandlungs-Operator:

- Ein selbstdefinierter Datentyp (z.B. eine Klasse) soll in einen anderen Datentyp umgewandelt werden.
- Es gibt 2 Möglichkeiten:
  - Definition eines Typumwandlungs-Konstruktors (siehe vorher)
    → Umwandlung von einem anderen Datentyp in den eigenen Typ.
  - Definition eines Typumwandlungs-*Operators*:
    → Umwandlung des eigenen Datentyps in einen anderen Typ.



- Beispiel: Ein Objekt des eigenen Datentyps (z.B. die Datums-Klasse) soll automatisch in einen String umgewandelt werden.
- Alternativ könnte auch eine `toString()`-Methode geschrieben werden, die die Typprüfung des Compilers nicht umgeht...

Überladen von Operatoren

```
// datum.h
#include<string>

class Datum {
  // .... wie vorher
  operator std::string() const;
};
```

```
// in datum.cpp:
Datum::operator std::string() const {
  std::string temp("tt.mm.jjjj");
  temp[0] = tag_/10 + '0'; // implizite Umwandlungen
  temp[1] = tag_%10 + '0';
  temp[3] = monat_/10 + '0';
  temp[4] = monat_%10 + '0';

  int pos = 9;            // letzte Jahresziffer
  int j = jahr_;
  while(j > 0) {
    temp[pos] = j % 10 + '0'; // letzte Ziffer
    j /= 10;              // letzte Ziffer abtrennen
    --pos;
  }
  return temp;
}
```

Anwendung:

```
using namespace std;
Datum d;
string s1 = d;                          // implizit
string s2 = (string)d;                  // explizit
string s3 = static_cast<string>(d);     // explizit
cout << s1 << endl;                     // Ausgabe
```

Überladen von Operatoren
Operatoren -> und *

Operatoren -> und *:

- Die Operatoren können grundsätzlich für alles Mögliche eingesetzt werden, die häufigste Anwendung sind aber 'intelligente Zeiger' (engl. *smart pointer*)
- Die Operatoren sind unär, und sollten bei Benutzung als *smart pointer* einen Zeiger bzw. einen dereferenzierten Zeiger (ein Objekt) zurück-liefern.

Was könnten intelligente Zeiger leisten? Er könnte ...

- nie einen undefinierten Wert besitzen (entweder `0` oder ein existierendes Objekt
- bei der Dereferenzierung eines `0`-Zeigers eine Ausnahme erzeugen
- automatisch `delete` aufrufen, sobald der Zeiger nicht mehr gebraucht wird, oder besser:
- beim Kopieren des Zeigers mitzählen, wie viele Kopien des Zeigers existieren, und erst bei Löschung des letzten Zeigers `delete` aufrufen (sog. *reference counting*)

Überladen von Operatoren
Operatoren -> und *

Beispiel: Eigene **SmartPointer**-Klasse:

- Realisiert kein *reference counting*, aber
- Besitzt immer einen definierten Wert
- Wirft eine Ausnahme bei Dereferenzierung von **0**
- Kann implizit in **bool** umgewandelt werden (wie normale Zeiger)

Definition der Ausnahme:

```
#ifndef NULLPOINTEREXCEPTION_H
#define NULLPOINTEREXCEPTION_H
#include<stdexcept>

class NullPointerException : public std::runtime_error {
 public:
   NullPointerException()
      : std::runtime_error("NullPointerException!") {
   }
};

#endif
```

Überladen von Operatoren
Operatoren -> und *

```
#ifndef SMARTPTR_T
#define SMARTPTR_T
#include"NullPointerException.h"

template<typename T>
class SmartPointer {
   public:
      SmartPointer(T *p=0);
      ~SmartPointer();   // nicht virtual: Vererbung ist nicht geplant
      T* operator->() const;
      T& operator*() const;
      operator bool() const;
   private:
      T* zeigerAufObjekt;
      void check() const; // Prüfung auf nicht-Null

    void operator=(const SmartPointer& );    // p1 = p2;  verbieten
    SmartPointer(const SmartPointer&);       // Kopien mit Copy-CTOR verbieten
};

template<typename T>
inline void SmartPointer<T>::check() const {
   if(!zeigerAufObjekt) {
     throw NullPointerException();
   }
}
```

Überladen von Operatoren
Operatoren -> und *

```
template<typename T>
inline SmartPointer<T>::SmartPointer(T *p) : zeigerAufObjekt(p) {
}

template<typename T>
inline SmartPointer<T>::~SmartPointer() {
    delete zeigerAufObjekt;
}

template<typename T>
inline T* SmartPointer<T>::operator->() const {
    check();
    return zeigerAufObjekt;
}

template<typename T>
inline T& SmartPointer<T>::operator*() const {
    check();
    return *zeigerAufObjekt;
}

template<typename T>
inline SmartPointer<T>::operator bool() const {
   return bool(zeigerAufObjekt);
}

#endif    // SMARTPTR_T
```

Überladen von Operatoren
Operatoren -> und *

```
#include"smartptr.t"
#include<iostream>
using namespace std;

class A {
    public:
      void hi() { cout << "hier ist A::hi()" << endl; }
};


// Übergabe per Referenz:
template<class T>
void perReferenz(const SmartPointer<T>& p) {
    cout << "Aufruf: perReferenz(const SmartPointer<T>&):";
    p->hi(); // (p.operator->())->hi();
}
```

- Warum wird der **SmartPointer** in der obigen Methode per Referenz übergeben? Wäre eine Übergabe per Wert möglich?

Überladen von Operatoren
Operatoren -> und *

```
int main() {
    cout << "Zeiger auf dynamische Objekte:" << endl;
    cout << "Konstruktoraufruf" << endl;
    SmartPointer<A> spA(new A);

    cout << "Operator ->" << endl;
    spA->hi();

    cout << "Operator *" << endl;
    (*spA).hi();

     // Parameterübergabe eines SmartPointer
    perReferenz(spA);

   /* Wirkung der Sicherungsmaßnahmen */

    SmartPointer<A> spUndef;
    try {
    if (!spUndef)
      cout << "undefinierter Zeiger:" << endl;
      spUndef->hi();                  // Laufzeitfehler!
      (*spUndef).hi();                // Laufzeitfehler!
    } catch(NullPointerException ex) {
       cout << "Laufzeitfehler: " << ex.what() << endl;
    }
}
```

```
Zeiger auf dynamische Objekte:
Konstruktoraufruf
Operator ->
hier ist A::hi()
Operator *
hier ist A::hi()
Aufruf: perReferenz(const
SmartPointer<T>&):hier ist
A::hi()
undefinierter Zeiger:
Laufzeitfehler:
NullPointerException!
```

Überladen von Operatoren
Operatoren -> und *

Smart Pointer im neuen Standard C++11:

- Früher: **`auto_ptr`** → wird nicht mehr empfohlen!

- **`unique_ptr`**
  Verhält sich ähnlich wie **`SmartPointer`**: Alleiniger (*unique*) Zeiger auf ein dynamisches Datenobjekt.

- **`shared_ptr`**
  Zeiger auf ein Datenobjekt, der auch kopiert und zugewiesen werden kann. Implementiert intern das *reference counting*.

- **`weak_ptr`**
  Zeiger auf Objekte, die bereits von **`shared_ptr`** verwaltet werden. Dient der Auflösung von zyklischen Verkettungen.

Überladen von Operatoren
Objekte als Funktion → Funktoren

- Mit dem Operator `operator()()` kann eine Instanz einer Klasse wie eine Funktion aufgerufen werden. Klassen, die hauptsächlich über diesen Operator aufgerufen werden, nennt man *Funktoren*.
- Anzahl und Typ der Parameter sind frei wählbar, genau wie der Rückgabetyp des Operators.
- Funktoren besitzen ansonsten alle Eigenschaften wie normale Klassen!
- Dadurch lassen sich leicht Funktionen realisieren, die ein Gedächtnis (Klassen-Elemente) haben, oder deren Verhalten zur Laufzeit verändert werden kann.

Beispiel: Ein Funktor zur Berechnung von `sin(x)` mit unterschiedlichen Winkelangaben (Bogenmass, Grad, Neugrad)

Überladen von Operatoren
Objekte als Funktion → Funktoren

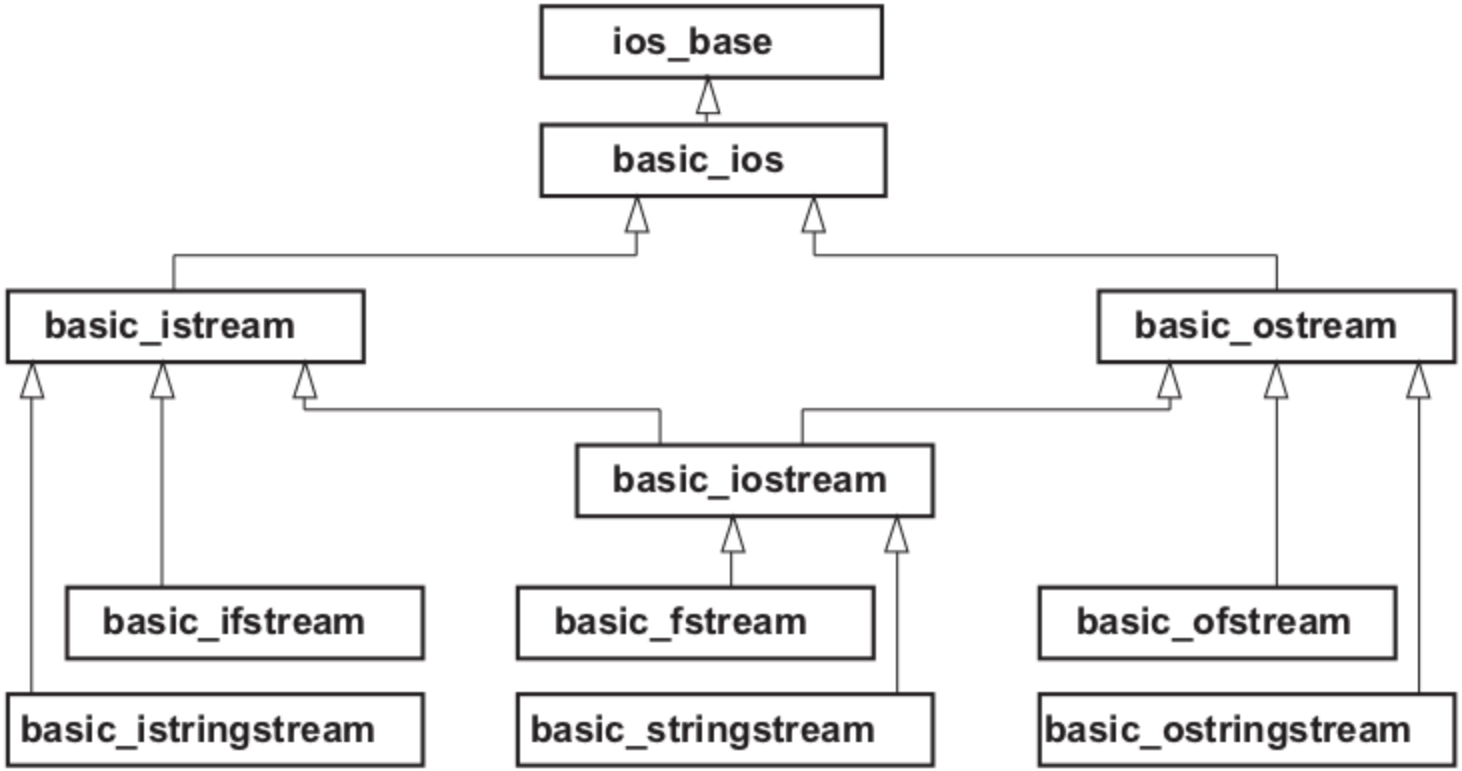
```
#ifndef SINUS_H
#define SINUS_H
#include<cmath>   //  sin(), Konstante M_PI für pi
#include<stdexcept>

class Sinus {
    public:
      enum Modus { bogenmass, grad, neugrad};
      Sinus(Modus m = bogenmass) : berechnungsart(m) { }

      double operator()(double arg) const {
         double erg;
         switch(berechnungsart) {
           case bogenmass : erg = std::sin(arg);          break;
           case grad      : erg = std::sin(arg/180.0*M_PI); break;
           case neugrad   : erg = std::sin(arg/200.0*M_PI); break;
           default   : ;  // kann hier nicht vorkommen
         }
         return erg;
      }
 private:
      Modus berechnungsart;
};
#endif
```

Überladen von Operatoren
Objekte als Funktion → Funktoren

```
#include<iostream>
#include"sinus.h"

using namespace std;

void sinusAnzeigen(double arg, const Sinus& funktor) {
    cout << funktor(arg) << endl;
}

int main() {
    Sinus sinrad;
    Sinus sinGrad(Sinus::grad);
    Sinus sinNeuGrad(Sinus::neugrad);

    // Aufruf der Objekte wie eine Funktion
    cout << "sin(" << M_PI/4.0 <<" rad) = " << sinrad(M_PI/4.0) << endl;
    cout << "sin(45 Grad)               = " << sinGrad(45.0)    << endl;
    cout << "sin(50 Neugrad)            = " << sinNeuGrad(50.0) << endl;

    // Übergabe eines Funktors an eine Funktion
    sinusAnzeigen(50.0, sinNeuGrad);

    // ...
}
```

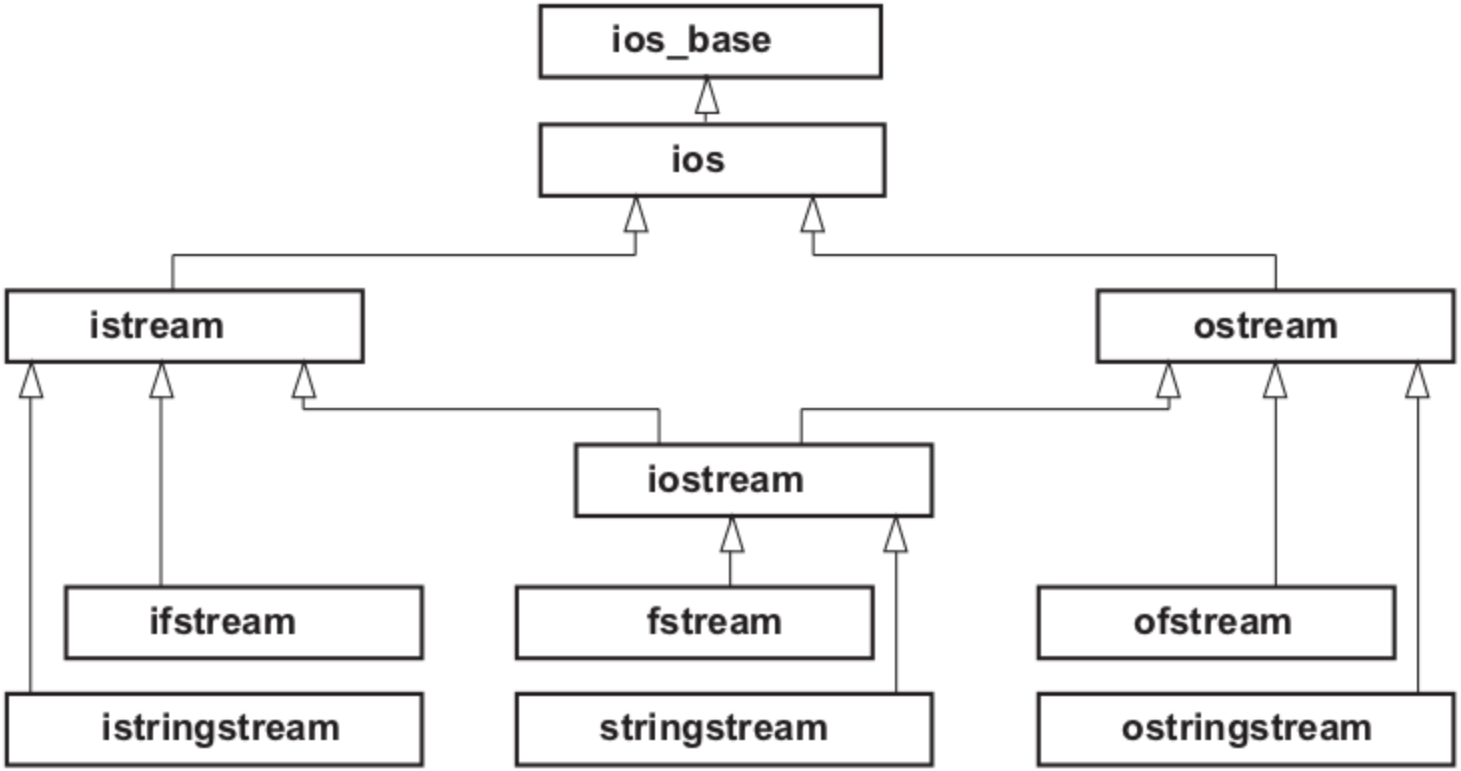
Dateien und Ströme

- Das Ein- und Ausgabesystem von C++ basiert auf einer Hierarchie von Template-Klassen:



Dateien und Ströme

- Spezialisierung für `char`:
  z.B. `typedef basic_ofstream<char> ofstream;`



Ausgabe

- In der Klasse **ostream** existiert der **operator<<()** für alle internen Datentypen.
- Der Rückgabetyp sollte eine Referenz auf **ostream** sein (daisy chaining...)
- Die Ausgabe eigenen Datentypen kann (wie schon gesehen) über eine globale Operatormethode realisiert werden.
- Bisherige Methoden:

```
ostream & put(char); // Ausgabe eines einzelnen Zeichens
ostream & write(const char *, size_t); // (binäre) Ausgabe einer Folge von chars
```

- Bisherige Formatierungen:

```
cout.width(10);  // Breite des folgenden! Ausgabe in den Stream
cout.fill('0');  // Setzen des Füllzeichens, falls Breite nicht komplett benutzt
```

Dateien und Ströme

Steuerung über Flaggen

- Die Klasse ios definiert einen Typ für 'Flaggen': `ios::fmtflags`
- Mit diesen Flaggen sind weitergehende Formatierungen möglich:

Name	Bedeutung
`boolalpha`	`true/false` alphabetisch ausgeben oder lesen
`skipws`	Zwischenraumzeichen ignorieren
`left`	linksbündige Ausgabe
`right`	rechtsbündige Ausgabe
`internal`	zwischen Vorzeichen und Wert auffüllen
`dec`	dezimal
`oct`	oktal
`hex`	hexadezimal
`showbase`	Basis anzeigen
`showpoint`	nachfolgende Nullen ausgeben
`uppercase`	E,X statt e,x
`showpos`	+ bei positiven Zahlen anzeigen
`scientific`	Exponential-Format
`fixed`	Gleitkomma-Format
	Puffer leeren (flush):
`unitbuf`	– nach jeder Ausgabeoperation
`stdio`	– nach jedem Textzeichen

Vordefinierte Bitmasken:

Name	Wert
`adjustfield`	`left | right | internal`
`basefield`	`oct | dec | hex`
`floatfield`	`fixed | scientific`

Dateien und Ströme

Setzen und Lesen der Flaggen:

```
ostream Ausgabe;
ios::fmtflags altesFormat;
ios::fmtflags neuesFormat = ios::left|ios::oct|ios::showpoint|ios::fixed;
// gleichzeitiges Lesen und Setzen aller Flags
altesFormat = Ausgabe.flags(neuesFormat);
// Setzen eines Flags
Ausgabe.setf(ios::hex); // gleichwertig damit ist:
Ausgabe.flags(Ausgabe.flags() | ios::hex);
// Zurücksetzen eines Flags
Ausgabe.unsetf(ios::hex);
// Zur Verhinderung von Widersprüchen besser:
Ausgabe.setf(ios::hex, ios::basefield); // Setzte erst alle Flags der Maske zurück!!
```

Zwischenspeichern der Ausgabe:

- Normalerweise ist die Flagge **`unitbuf`** gesetzt, und eine Ausgabe erfolgt nach jeder Ausgabeoperation.
- Falls **`unitbuf`** nicht gesetzt ist, erfolgt die Ausgabe erst nach **`cout.flush()`**, das auch automatisch mit **`endl`** ausgeführt wird.

Dateien und Ströme

Weite von Fließkommazahlen:

- `cout.precision(int)` legt die Anzahl der *Gesamtziffern* fest, sofern `fixed` oder `scientific` *nicht* gesetzt sind. Andernfalls legt precision die Anzahl der *Nachkommastellen* fest!
- Die Formatierung bleibt so lange erhalten, bis ein neuer Wert angegeben wird.
- Falls die Anzahl der Ziffern vor dem Komma den Wert von `precision` überschreitet, wird auf die wissenschaftliche Notation (mit Exponent) umgeschaltet.
- Die Methode kann auch genutzt werden, um die aktuell eingestellte Ziffernzahl festzustellen.

```
double f = 1234.123456789012345;
cout.setf(ios::scientific, ios::floatfield);
cout.precision(4);
cout << f << endl;  // 1.2341e+03
cout.setf(ios::fixed, ios::floatfield);
cout.precision(8);
cout << f << endl;  // 1234.12345679
```

Dateien und Ströme

Eingabe:

- In der Klasse **istream** existiert der **operator>>()** für alle internen Datentypen.
- Der Rückgabetyp sollte eine Referenz auf **istream** sein (daisy chain...)
- Die Eingabe eigenen Datentypen kann (wie bei der Ausgabe) über eine globale Operatormethode realisiert werden:

```
istream& istream::operator>>(T& var) {
 // überspringe Zwischenraumzeichen
 // lies die Variable var des Typs T aus dem istream ein
  return * this;
}
```

- Falls Zeichen oder Bytes eingelesen und Zwischenraumzeichen nicht ignoriert werden sollen, kann die Elementfunktion istream& get() genommen werden, die in mehreren überladenen Versionen existiert.

```
// zeichenweises Kopieren der Standardeingabe
char c;
while(cin.get(c)) {
  cout << c;
}
```

Dateien und Ströme

Weitere Varianten:

- Lesen von **N** Zeichen bis zum Terminatorzeichen **t**, das auch gespeichert wird (und danach noch das '\0' als Endkennung):

```
istream& get( char *, unsigned int, char t='\n');
```

- Wie Methode oben, aber keine Speicherung des Terminatorzeichens:

```
istream& getline(char *, unsigned int, char t= '\n' );
```

- Lese und verwerfe alle Zeichen, bis entweder das Terminatorzeichen **t** oder **N** andere Zeichen gelesen wurden (EOF ist ein Makro, das das Dateiende kennzeichnet, *end-of-file*):

```
istream& ignore(int n, int t = EOF);
```

- Gebe ein Zeichen an den istream zurück:

```
istream& putback(char c);
```

- Lese ein Zeichen als ASCII-Code, bei EOF wird -1 zurückgegeben:

```
int get();
```

Dateien und Ströme

- Einlesen (und Ausgabe) der Standardeingabe Zeichen für Zeichen:

```
int i;
while(cin.good() && (i = cin.get()) != EOF) { // d.h. weder EOF noch Lesefehler
  cout.put(static_cast<char>(i));
}
```

- **`good()`** überprüft, ob keins der 3 internen Bits
  eofbit (Ende des Datenstroms)
  failbit (Lesefehler, z.B. physikalisch..)
  badbit (Integrität des Datenstroms korrupt)
  gesetzt ist. **`eof()`** überprüft nur das *eofbit*!

- Analog überprüfen **`fail()`** und **`bad()`** die beiden anderen Bits!

- Vorschau auf das nächste Zeichen, *ohne* es zu lesen:

```
int peek();
// ist äquivalent zu: c = cin.get(); cin.putback(c);
```

Manipulatoren

- Das Verändern der Ausgabe über Format-Flags ist recht kompliziert...
- Wie könnte man folgendes Verhalten erzielen?:

```
cout << oct << zahl << endl;  // Ausgabe der Zahl als Oktalzahl
```

- **oct** könnte eine eigene Klasse sein, die einen globalen **operator<<** besitzt → sehr aufwändig ...
- Es existieren (u.a.) folgende Methoden in der Klasse **ostream**:

```
ostream& operator<<(ostream& (*fp)(ostream&))
{
  *fp(*this); // Funktionsaufruf
  return *this;
}
ostream& operator<<(ios& (*fp)(ios&));
// ...
ostream& endl(ostream&);
// Funktion zur Klasse ios
ios& oct(ios&);
```

- Dadurch können beliebige Funktionen als Manipulator benutzt werden, so lange sie der geforderten Signatur entsprechen!

Dateien und Ströme

- **endl** könnte z.B. so implementiert sein:

```
ostream& endl(ostream& os) {
  os.put('\n');
  os.flush();
  return os;
}
```

- Es existieren viele vordefinierte Manipulatoren (wobei das Einbinden von **<iostream>** automatisch auch **<ios>** beinhaltet!):

Dateien und Ströme

Name	Bedeutung	Typ
endl	neue Zeile ausgeben	ostream&
ends	Nullzeichen ('\0') ausgeben	ostream&
flush	Puffer leeren	ostream&
ws	Zwischenraumzeichen aus der Eingabe entfernen	istream&

iostream-Manipulatoren

Name	Bedeutung
boolalpha	true/false alphabetisch ausgeben oder lesen
noboolalpha	true/false numerisch (1/0) ausgeben oder lesen
showbase	Basis anzeigen
noshowbase	keine Basis anzeigen
showpoint	nachfolgende Nullen ausgeben
noshowpoint	keine nachfolgenden Nullen ausgeben
showpos	+ bei positiven Zahlen anzeigen
nowshowpos	kein + bei positiven Zahlen anzeigen
skipws	Zwischenraumzeichen ignorieren
noskipws	Zwischenraumzeichen berücksichtigen
uppercase	E,X statt e,x
nouppercase	e,x statt E,X
unitbuf	Puffer nach jeder Ausgabe leeren
nounitbuf	Ausgabe puffern
adjustfield:	
internal	zwischen Vorzeichen und Wert auffüllen
left	linksbündige Ausgabe
right	rechtsbündige Ausgabe
basefield:	
dec	dezimal
oct	oktal
hex	hexadezimal
floatfield:	
fixed	Gleitkomma-Format
scientific	Exponential-Format

ios-Manipulatoren

iomanip-Manipulatoren

Name	Bedeutung
resetiosflags(ios::fmtflags M)	Flags entsprechend der Bitmaske M zurücksetzen
setiosflags(ios::fmtflags M)	Flags entsprechend M setzen
setbase(int B)	Basis 8, 10 oder 16 definieren
setfill(char c)	Füllzeichen festlegen
setprecision(int n)	Fließkommaformat
setw(int w)	Weite setzen (entspricht width())

Dateien und Ströme

- Verwendung von Manipulatoren:

```
cout << setw(6) << setfill('0') << 999 << ' ';  // 000999
cout << setprecision(8) << 1234.56789 << endl;  // 1234.5679
```

- Wieso können die Manipulatoren auch Parameter haben? Sie entsprechen damit doch gar nicht der vereinbarten Deklaration!?!
- Lösung über Funktoren: Objekte, die sich wie Funktionen verhalten!
- Beispiel: Manipulator, der N Leerzeilen ausgibt:

```
#ifndef MANIPULA_H
#define MANIPULA_H
#include<iostream>
class Leerzeilen {
  public:
    Leerzeilen(int i = 1) : anzahl(i) {}
    std::ostream& operator()(std::ostream& os) const {  // Benutzung wie ein Parameter-
      for(int i = 0; i < anzahl; ++i) { os << '\n'; }   // loser Manipulator !!
      os.flush();
      return os;
    }
  private:
    int anzahl;
};  // Nächste Folie geht's weiter ...
```

Dateien und Ströme

```
inline std::ostream& operator<<(std::ostream& os, const Leerzeilen& leerz) {
  return leerz(os); // Funktoraufruf
}
#endif
// MANIPULA_H
```

- Die Benutzung könnte jetzt so aussehen:

```
#include "manipula.h"
int main() {
  std::cout << Leerzeilen(25) << "Ende" << std::endl;  // Konstruktoraufruf!!
  //oder
  Leerzeilen lz(10);
  std::cout << lz << "Ende" << std::endl;
}
```

Bei der ersten Benutzung passiert folgendes:

- Konstruktor **`Leerzeilen(25)`** wird aufgerufen
- **`operator<<(cout, Leerzeilen(25))`** wird aufgerufen
- **`leerz.operator()(cout)`** wird aufgerufen
- Eine Referenz auf den übergebenen **`ostream`** wird zurückgegeben.

Dateien und Ströme

Fehlerbehandlung

- Es gibt den Aufzählungstyp **ios_base::iostate**, der so definiert ist (die tatsächlichen Bitwerte sind implementationsabhängig):

```
enum iostate {
  goodbit = 0x00,   // alles ok (kein echtes Bit, nur Überprüfung auf 0
  eofbit  = 0x01,   // Ende des Streams
  failbit = 0x02,   // letzte Ein-/Ausgabe war fehlerhaft
  badbit  = 0x04    // ungültige Operation, grober Fehler
};
```

Folgende Methoden fragen diese Bits ab:

Funktion	Ergebnis
`iostate rdstate()`	aktueller Status
`bool good()`	wahr, falls »gut«, d.h. `rdstate() == 0`
`bool eof()`	wahr, falls Dateiende
`bool fail()`	wahr, falls `failbit` oder `badbit` gesetzt
`bool bad()`	wahr, falls `badbit` gesetzt
`void clear()`	Status auf `goodbit` setzen
`void clear(iostate s)`	Status auf `s` setzen
`void setstate(iostate)`	einzelne Statusbits setzen

Dateien und Ströme

Typumwandlung von Streams zu bool

- Abfragen einer erfolgreichen Dateiöffnung und Überprüfung auf EOF kann man so realisieren:

```
ifstream quellfile;
quellfile.open("text.dat");
if(!quellfile) {          // Umgewandelt in quellfile.operator!()
  cerr << "Datei kann nicht geöffnet werden!";
  exit(-1);
}
int c;
while(quellfile.get(c)) { Umgewandelt in quellfile.get(c).operator void*()
  zielfile.put(c);
}
```

- Realisierung über (Typumwandlungs-)operatoren:

```
// Konversion eines Objekts in einen void-Zeiger
ios::operator void* () const {
  if(fail()) return static_cast<void * >(0);
  else       return static_cast<void * >(this);
}

bool ios::operator!() const {  // überladener Negationsoperator
  return fail();
}
```

Dateien und Ströme

Arbeiten mit Dateien

- Beim Öffnen von Dateien wird ein Modus angegeben (aus `ios_base`):

Modus	Bedeutung
`app`	beim Schreiben Daten an die Datei anhängen
`ate`	nach dem Öffnen an das Dateiende springen
`binary`	keine Umwandlung verschiedener Zeilenendekennungen
`in`	zur Eingabe öffnen
`out`	zur Ausgabe öffnen
`trunc`	vorherigen Inhalt der Datei löschen

- Bestimmte Modi sind voreingestellt (z.B. `ios::in` beim `istream`)
- Das Angeben einen neuen Modus überschreibt den Aktuellen, d.h. bei binärer Eingabe muss z.B. `ios::in | ios::binary` angegeben werden.
- Folgende Modi-Kombinationen sind möglich:

Dateien und Ströme

ios_base Flag combination					stdio equivalent
binary	in	out	trunc	app	
	+				"r"
		+			"w"
		+	+		"w"
		+		+	"a"
	+	+			"r+"
	+	+	+		"w+"
+	+				"rb"
+		+			"wb"
+		+	+		"wb"
+		+		+	"ab"
+	+	+			"r+b"
+	+	+	+		"w+b"

- Es ist z.B. nicht zugelassen, die **in/out/app**-Bits gleichzeitig zu setzen.
- Oder das **trunc**-Bit gibt nur Sinn bei gleichzeitigem setzten von **out**.

Dateien und Ströme

Positionierung bei Dateien

- Manchmal möchte man eine Datei nicht nur sequentiell lesen oder schreiben, sondern wahlfrei auf sie zugreifen.
- Dazu gibt es Methoden, die die Endung 'g' (für 'get') oder 'p' (für 'put') haben.
- Die relative Bezugsposition kann sein:
  - `ios::beg` relativ zum Dateibeginn
  - `ios::cur` relativ zur aktuellen Position
  - `ios::end` relativ zum Dateiende

Rückgabetyp	Funktion	Bedeutung
`ios::pos_type`	`tellg()`	aktuelle Leseposition
`ios::pos_type`	`tellp()`	aktuelle Schreibposition
`istream&`	`seekg(p)`	absolute Position `p` aufsuchen
`istream&`	`seekg(r, Bezug)`	relative Position `r` aufsuchen (zur Bezugsposition siehe Text)
`ostream&`	`seekp(p)`	absolute Position `p` aufsuchen
`ostream&`	`seekp(r, Bezug)`	relative Position `r` aufsuchen

Dateien und Ströme

Beispiel:

```
ifstream einIfstream;
einIfstream.open("seek.dat", ios::binary|ios::in);
einIfstream.seekg(9);           // absolute Leseposition 9 suchen (Zählung ab 0)
char c;
einIfstream.get(c);             // an Pos. 9 lesen, get schaltet Position um 1 weiter
einIfstream.seekg(2, ios::cur); // 2 Positionen weitergehen
ios::pos_type position = einIfstream.tellg(); // akt. Position merken
// ...
einIfstream.seekg(-4, ios::end);// 4 Positionen vor dem Ende
// ...
einIfstream.seekg(position, ios::beg);// zur gemerkten Position gehen
                                      // seekg(position) hätte genauso funktioniert!!
```

- Die Zeiger zum Lesen und Schreiben sind unabhängig, können also auf verschiedenen Positionen zeigen!

Lesen und Schreiben in derselben Datei

- Es gibt eine Klasse **fstream**, die von **istream** und **ostream** abgeleitet ist. Es wird derselbe Pufferspeicher zum Lesen und Schreiben benutzt.
- Die folgende Folie zeigt ein Beispiel:

Dateien und Ströme

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

int main() {
  // Datei anlegen
  fstream filestream( "fstream2.dat" , ios::out | ios::trunc);
  filestream.close(); // leere Datei existiert jetzt
  int i;              // Hilfsvariable
  filestream.open( "fstream2.dat" , ios::in | ios::out); // Lesen und Schreiben

  for(i = 0; i < 20; ++i) {  // schreiben ...
    filestream << i << ' ';
  }
  filestream << endl; // Leerzeile anfügen

  filestream.seekg(0); // lesen ...
  while(filestream.good()) {
    filestream >> i;
    if(filestream.good()) { cout << i << ' ' ; }
    else {
      cout << endl << "Dateiende erreicht (oder Lesefehler)" ;
    }
  }
  cout << endl; // Ergebnis: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
  filestream.clear(); // EOF-Status löschen und Inhalt teilweise überschreiben
  filestream.seekp(5);// Position 5 zum Schreiben suchen
  filestream << "neuer Text " ; // ab Pos. 5 überschreiben
  filestream.seekg(0); // Anfang zum Lesen suchen
  char buf[100]; filestream.getline(buf,100); // Zeile lesen
  cout << buf << endl; // Ergebnis: 0 1 2neuer Text 8 9 10 11 12 13 14 15 16 17 18 19
}
```

Dateien und Ströme

Umleitung auf Strings

- In der Header-Datei **<sstream>** gibt es die Klassen **istringstream** und **ostringstream**, die wie normale Streams funktionieren, aber als Ein- bzw. Ausgabe-'Medium' einen String verwenden.
- Mit den Methoden **str()** und **str(string)** kann dieser String gelesen und gesetzt werden.

Beispiel:

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

int main() {
  string s = "Hallo 2";
  istringstream iss(s);
  string hallo;
  int zahl;
  iss >> hallo >> zahl;

  ostringstream oss;
  oss << hallo << ' ' << zahl;
  string s2 = oss.str();
  cout << s2 << endl;
}
```

Generische Programmierung in C++

C++ Sprachmittel für generische Programmierung:

- Standardbibliothek
- Container
- Algorithmen
- Iteratoren

} Standard-Template-Library (STL)

Generische Programmierung in C++

Große Teile des Konzepts sind durch die bereits eingeführten Templates bekannt. Kurze Wiederholung:

- Programmierung mit generischen Parametern. Generische Parameter sind selbst Typen oder Werte bestimmter Typen.
- Abstrakte Repräsentationen effizienter Algorithmen und Datenstrukturen finden.
- Mit möglichst wenig Annahmen über Datenabstraktionen und Algorithmen auskommen.
- Ggf. spezielle Algorithmen automatisch gegenüber allgemeinen bevorzugen, wenn die Effizienz dadurch verbessert wird.

Vorteil: Vermeiden von Code-Duplikation in statisch getypten Sprachen

Beispiel:

Generische Programmierung in C++

```
// Quadrat einer int-Zahl
int sqr(int a) { return a*a;}
// Quadrat einer double-Zahl
double sqr(double a) { return a*a;}
```

Mit Templates genügt eine einzige Funktion:

```
// Quadrat einer Zahl des Typs T
template<class T>
T sqr(T a) { return a*a; }
```

Anwendung:

```
int i = 3;
double d = 3.635;
int i2 = sqr(i); // Compiler generiert sqr() für T=int
int d2 = sqr(d); // Compiler generiert sqr() für T=double
```

Voraussetzungen für den Datentyp T sind hier nur:

- Der Operator * existiert.
- T-Objekte sind kopierbar.

Die C++ Standardbibliothek, generischer Teil (STL)

Sprachspezifische Klassenbibliothek mit generischen Komponenten (Teilmenge STL). Die STL unterscheidet zwei Ebenen:

1. Standard-Container und -Algorithmen für verschiedene Datentypen → immer dann angebracht, wenn das geforderte Verhalten des Containers bzw. der Algorithmen für alle Datentypen dasselbe sein soll.

   Realisierung durch Templates. Beispiel:

```
template<class T>
class stack {
  // Stack-Methoden und Attribute
};


// Benutzung
stack<int> einIntStack;
stack<string> einStringStack;
stack<XYZ> einXYZstack; // für beliebige XYZ-Objekte
```

Die C++ Standardbibliothek, generischer Teil (STL)

2. Schnittstellenobjekte (Iteratoren) zur Entkopplung von Containern und Algorithmen

   Vorteil: Reduktion der Größe und Komplexität der Library.

   Begründung:

   Annahme: Es werden k Algorithmen (find, copy, ...) für n verschiedene Container (`vector, stack, list`) und m verschiedene Datentypen (`float, int`) benötigt. OHNE STL und Templates sind dann $k \cdot n \cdot m$ Algorithmen notwendig.

   - Templates reduzieren m auf 1.
   - Entkopplung durch Iteratoren reduziert $k \cdot n$ auf $k + n$

Abstrakte und implizite Datentypen

- *Abstrakte* Datentypen (ADT) kapseln Daten und und Funktionen.
- Ein ADT wird ausschließlich über die öffentliche Schnittstelle spezifiziert. Eine Klasse in C++ implementiert einen ADT, die `public`-Methoden definieren die Schnittstelle.
- Ein *impliziter* Datentyp ist ein ADT, der zur Implementierung benutzt wird. Die STL erlaubt für manche ADTs verschiedene Implementierungen.

Beispiel:

```
template<T, class Containertyp = deque<T> >
class stack {
public:
  bool empty () const {
    // öffentliche Methode
    return c.empty();
  }

  // .... weitere Methoden
private:
  Containertyp c;
};
```

Abstrakte und implizite Datentypen

Auswahl der Implementierung:

```
stack<int> IntStackA;       // implizit deque
stack<int, list<int> > IntStackB;
stack<int, vector<int> > IntStackC;
```

STL-Bauelemente

Container

Klassen als Templates zum Verwalten von Objekten Basisklassen:

- `vector`
- `list`
- `deque`

ADTs:

- `stack`
- `queue`
- `priority_queue`
- `set` und `multiset`
- `map` und `multimap`

Die C++ Standardbibliothek, generischer Teil (STL)

- Jeder Container hat *spezifische* Methoden, zum Beispiel `push()`, `pop()` für einen `Stack`. Allen Containern *gemeinsam* sind öffentlich definierte Datentypen (Auswahl, `X` = Containertyp):

Datentyp	Bedeutung
`X::value_type`	`T`
`X::reference`	Referenz auf Container-Element
`X::const_reference`	Referenz auf Element eines const-Containers
`X::iterator`	Typ des Iterators
`X::const_iterator`	Iteratortyp für const-Container
`X::difference_type`	vorzeichenbehafteter integraler Typ
`X::size_type`	für Größenangaben (ohne Vorzeichen)

Implementierung durch

```
template <class T, class Container>
class stack {
public:
  typedef typename Container::value_type  value_type;
  typedef typename Container::size_type  size_type;
```

usw.

Die C++ Standardbibliothek, generischer Teil (STL)

Allen Containern gemeinsam sind auch öffentliche Methoden (Auswahl):

Rückgabetyp Methode	Bedeutung
`iterator begin()`	Anfang des Containers.
`const_iterator begin()`	Anfang eines Containers, Änderung der Elemente ist nicht erlaubt
`iterator end()`	Position *nach* dem letzten Element
`const_iterator end()`	`end()` für Container mit Elementen, die nicht geändert werden
`size_type max_size()`	maximal mögliche Größe des Containers
`size_type size()`	aktuelle Größe des Containers
`bool empty()`	`size() == 0` bzw. `begin() == end()`
`void swap(X&)`	Vertauschen mit Argument-Container
`X& operator=(const X&)`	Zuweisungsoperator

Die C++ Standardbibliothek, generischer Teil (STL)

- Vorteile:

- Schnittstelle für Algorithmen
- Definierter Rahmen für eigene Datenstrukturen und Algorithmen, die mit der C++ Standardbibliothek zusammenarbeiten sollen!

Die C++ Standardbibliothek, generischer Teil (STL)

Iteratoren

... sind Objekte, die auf Elemente eines Containers verweisen!

Operationen:

- `!=` Vergleich
- `*` Dereferenzierung
- `++` einen Schritt weitergehen
- ...

- Einfachster Fall: Iterator = *Zeiger*

- Die Art der Bewegung ist verborgen (Kontrollabstraktion), d.h. ++ auf einem Vektor wirkt anders als auf einem binären Suchbaum oder auf einer verketteten Liste.

Algorithmen

Grundprinzip:

- Generische Algorithmen haben *keine* Kenntnis von den Containern, auf denen sie arbeiten!
- Algorithmen benutzen nur Schnittstellenobjekte (Iteratoren). Dadurch bedingt kann derselbe Algorithmus auf verschiedenen Containern arbeiten.

Die C++ Standardbibliothek, generischer Teil (STL)

Vorhandene Algorithmen (Auszug, später mehr...):

```
for_each            find                      find_if
find_end            find_first_of             adjacent_find
count               mismatch                  equal
search              search_n                  copy
copy_backward       copy_if                   swap
iter_swap           swap_ranges               transform
replace             fill                      fill_n
generate            generate_n                remove
unique              reverse                   rotate
random_shuffle      partition                 sort
partial_sort        nth_element               binary_search
lower_bound         upper_bound               equal_range
includes            set_union                 set_intersect
set_difference      set_symmetric_difference  pop_heap
push_heap           make_heap                 sort_heap
min                 max                       min_element
max_element         lexicographical_compare   next_permutat
prev_permutation    accumulate                inner_product
partial_sum         adjacent_difference
```

Die C++ Standardbibliothek, generischer Teil (STL)

Beispiel:

- Variante 1: noch ohne Benutzung der STL

```
#include<iostream>
typedef int* Iterator;// neuer Typname
using std::cout;
using std::endl;
// Prototyp des Algorithmus
Iterator find(Iterator Anfang, Iterator Ende, const int& Wert);

int main() {
  const int Anzahl = 100;
  int einContainer[Anzahl]; // Container definieren
  Iterator begin = einContainer;
  // Anfang
  Iterator end = einContainer + Anzahl; // Ende+1
  for(int i = 0; i < Anzahl; ++i) // Container füllen
    einContainer[i] = 2*i;
  int Zahl = 0;
  do {
    cout << " gesuchte Zahl eingeben: (-1 = Ende)";
    cin >> Zahl;
    Iterator Position = find(begin, end, Zahl);
    if (Position != end)
      cout << "gefunden an Position " << (Position - begin) << endl;
    else cout << Zahl << " nicht gefunden!" << endl;
  } while(Zahl != -1);
}
```

Die C++ Standardbibliothek, generischer Teil (STL)

Implementierung von `find()`:

```
Iterator find(Iterator Anfang, Iterator Ende, const int& Wert) {

  while(Anfang != Ende      // Zeigervergleich
        && *Anfang != Wert) // Dereferenzierung und Objektvergleich
    ++Anfang;  // weiterschalten

  return Anfang;
}
```

Die C++ Standardbibliothek, generischer Teil (STL)

Variante 2:

Prototyp durch Template ersetzen:

```
template<class Iteratortyp, class T>
Iteratortyp find(Iteratortyp Anfang, Iteratortyp Ende, const T& Wert) {

  while(Anfang != Ende         // Iteratorvergleich
        && *Anfang != Wert)    // Dereferenzierung und Objektvergleich
    ++Anfang;  // weiterschalten

  return Anfang;
}
```

Der Rest des Programms bleibt unverändert!!

Die C++ Standardbibliothek, generischer Teil (STL)

Variante 3: mit der STL

```
#include<vector>
#include<algorithm> // find() !
#include<iostream>
using namespace std;

int main() {
  const int Anzahl = 100;
  vector<int> einContainer(Anzahl); // STL-Container
  for(int i = 0; i < Anzahl; ++i) // Container füllen
    einContainer[i] = 2*i;
  int Zahl = 0; // bis hierher wie vorher
  do {
    cout << " gesuchte Zahl eingeben: (-1 = Ende)";
    cin >> Zahl;
    // Benutzung von Container-Methoden begin(), end():
    vector<int>::iterator Position = find(einContainer.begin(), einContainer.end(), Zahl);
    if (Position != einContainer.end())
      cout << "gefunden an Position " << (Position - einContainer.begin())<< endl;
    else cout << Zahl << " nicht gefunden!" << endl;
  } while(Zahl != -1);
}
```

Die C++ Standardbibliothek, generischer Teil (STL)

Weiteres Beispiel: Wörterbuch

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

int main() {
  typedef map<string, string> MapType;
  typedef MapType::value_type Paar;

  MapType Woerterbuch;
  Woerterbuch.insert(Paar("Firma",  "company"));
  Woerterbuch.insert(Paar("Objekt", "object"));
  Woerterbuch.insert(Paar("Buch",   "book"));
  cout << Woerterbuch["Buch"] << endl; // assoziativer Zugriff

  // Alphabetische Ausgabe deutsch : englisch
  MapType::iterator I = Woerterbuch.begin();
  while(I != Woerterbuch.end()) {
    cout << (*I).first << " : " << (*I).second << endl;
    ++I;
  }
}
```

Iteratoren im Detail

- Iteratoren sind Bindeglieder zwischen Algorithmen und Containern → *Schlüsselkonzept*.
- Sie werden von den Containern zur Verfügung gestellt, weil diese die Information über ihren Aufbau besitzen. Beispiel:

```
// Iteratordefinition mit typedef
template <class T>
class vector {
public:
  typedef T* iterator; // implementationsabhängig!
  typedef const T* const_iterator;
  ...
  iterator begin();
  const_iterator begin() const;
  iterator end();
  const_iterator end() const;
  ...
```

oder ...

Iteratoren im Detail

- Iteratordefinition durch geschachtelte Klasse:

```
template <class T>
class list {
public:
  class iterator : public bidirectional_iterator<T, ...>
    // ...
  }
  ...
  iterator begin();
  iterator end();

  // const_iterator entsprechend
  ...
```

Iteratorkategorien

- Iteratorkategorien sind keine Typen, sondern *Anforderungen*

  Allen gemeinsam: `* != == ++`

- **Input-Iterator**

- nur lesender Zugriff auf Element eines Stroms von Eingabedaten, single pass

```
// Woher ist ein Input-Iterator
Woher = IstreamContainer.begin();
while(Woher != IStreamContainer.end()) {
  Wert = *Woher; // Wert lesen...
  // weitere Berechnungen mit Wert …
  ++Woher;
}
```

Iteratorkategorien

- **Output-Iterator**

- nur schreibender Zugriff auf Element eines Stroms von Ausgabedaten, single pass

```
// Wohin ist ein Output-Iterator
*Wohin++ = Wert;
```

- **Forward-Iterator**

- kann lesen und schreiben, aber nur in Vorwärtsrichtung

- multiple pass ist möglich (Eignung z.B. für einfach-verkettete Liste)

- **Bidirectional-Iterator**

- wie Forward-Iterator, aber vor- und rückwärts (Operator `--`, Eignung z.B. für doppelt-verkettete Liste oder linear im Speicher abgelegte Datenstrukturen (`vector`))

Iteratorkategorien

- **Random-Access-Iterator**

- wie Bidirectional-Iterator, zusätzlich *wahlfreier* Zugriff
  Operator [] für Container, Sprünge, Arithmetik:

```
// Position sei ein Iterator, der auf eine Stelle
// irgendwo innerhalb der Tabelle verweist
n1 = Position - Tabelle.begin(); // Arithmetik
cout << Tabelle[n1] << endl;
// ist gleichwertig mit:
cout << *Position << endl;
if(n1 < n2)
  cout << Tabelle[n1] << "liegt vor " << Tabelle[n2] << endl;
```

Standard-Iterator und Traits-Klassen

- Vorteil der Templates: Auswertung der Typnamen zur Kompilierzeit
- **Ziel**: Typnamen, die zu Iteratoren gehören, ermitteln, ohne sich die Innereien eines Iterators ansehen zu müssen
- Vorschrift: jeder Iterator der C++-Standardbibliothek stellt bestimmte Typnamen öffentlich zur Verfügung

Hilfsmittel: **Traits**-Klassen (engl., etwa Wesenszug, Eigentümlichkeit)

Die Typnamen einer Iteratorklasse werden nach außen exportiert:

```
template<class Iterator>
struct iterator_traits {
  typedef typename Iterator::difference_type difference_type;
  typedef typename Iterator::value_type value_type;
  typedef typename Iterator::pointer pointer;
  typedef typename Iterator::reference reference;
  typedef typename Iterator::iterator_category iterator_category;
};
```

Kann diese Aufgabe nicht direkt von einer Iteratorklasse selbst übernommen werden?

Standard-Iterator und Traits-Klassen

Ja. Aber es gibt Schwachstellen:

- Die Algorithmen der C++-Standardbibliothek sollen auch auf einfachen C-Arrays arbeiten können.
- Die damit arbeitenden Iteratoren sind aber nichts anderes als *Zeiger*, möglicherweise auf Grunddatentypen wie int.
- Ein Typ `int*` kann keine Typnamen zur Verfügung stellen.
  Damit ein generischer Algorithmus dennoch die üblichen Typnamen verwenden kann, wird das obige Template für Zeiger spezialisiert:

```
template<class T>
struct iterator_traits<T*> { // Spezialisierung für Zeiger
  typedef ptrdiff_t difference_type;
  typedef T value_type;
  typedef T* pointer;
  typedef T& reference;
  typedef random_access_iterator_tag iterator_category;
  // (siehe unten)
};
```

Standard-Iterator und Traits-Klassen

- Mit diesen beiden Templates ist es möglich, aus dem Iteratortyp die benötigten Typnamen abzuleiten, und eine Funktion `distance()` kann so geschrieben werden:

```
template<class InputIterator>
typename iterator_traits<InputIterator>::difference_type
distance(InputIterator Erster, InputIterator Zweiter) {

  // Berechnung

}
```

- Nur noch ein Typ ist bei der Instantiierung notwendig.
- Die traits-Klassen erlauben, Datentypnamen wie `difference_type` für komplexe Iteratoren und für Grunddatentypen wie `int*` zu definieren.
- Generische Algorithmen können einheitlich für Iteratorklassen und Grunddatentypen geschrieben werden.

Standard-Iterator und Traits-Klassen

Wie funktioniert dies im Einzelnen? Der Compiler liest den Rückgabetyp von `distance()` und instantiiert das Template `iterator_traits` mit dem betreffenden Iterator. Dabei sind zwei Fälle zu unterscheiden:

- Komplexer Iterator, zum Beispiel Listeniterator:
  Dann ist der gesuchte Typ `iterator_traits<Iteratortyp>::difference_type` identisch mit `Iteratortyp::difference_type`, wie die Auswertung des instantiierten `iterator_traits`-Templates ergibt.
- Schlichter Zeiger, zum Beispiel `int*`:
  Zu einem Zeigertyp können nicht intern Namen wie `difference_type` per `typedef` definiert werden. Die Spezialisierung des `iterator_traits`-Templates für Zeiger sorgt nun dafür, dass nicht auf Namen des Iterators zugegriffen wird, weil in der Spezialisierung die gewünschten Namen direkt zu finden sind, ohne über einen Iterator gehen zu müssen.

`distance()` kann also sehr allgemein geschrieben werden.

Standard-Iterator und Traits-Klassen

Ohne den Traits-Mechanismus müsste es Spezialisierungen für alle benötigten Zeiger geben, nicht nur für Zeiger auf Grunddatentypen, sondern auch für Zeiger auf Klassenobjekte. Deshalb wird in der C++-Standardbibliothek ein Standarddatentyp für Iteratoren angegeben, von dem jeder benutzerdefinierte Iterator erben kann:

```
namespace std {
  template<class Category, class T,
           class Distance = ptrdiff_t,
           class Pointer = T*, class Reference = T&>
  struct iterator {
    typedef Distance difference_type;
    typedef T value_type;
    typedef Pointer pointer;
    typedef Reference reference;
    typedef Category iterator_category;
  };
}
```

Durch die *public*-Erbschaft sind diese Namen in allen abgeleiteten Klassen sicht- und verwendbar.

Markierungen und Identifizierung

Jeder Iterator der STL ist mit einer Markierung versehen, die von eigenen Programmen ebenfalls benutzt werden kann. Die zugehörigen Klassen sind öffentlich und vordefiniert:

```
// Markierungsklassen
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag : public input_iterator_tag {};
struct bidirectional_iterator_tag: public forward_iterator_tag {};
struct random_access_iterator_tag : public bidirectional_iterator_tag {};
```

Die Klassen sind leer. Warum?

Markierungen und Identifizierung

Der Zweck ist nur

- zur Kompilierzeit über die Namen Verträglichkeiten festzustellen oder
- einen möglichst effizienten Algorithmus auszuwählen.

Ein selbst geschriebener Iterator erbt die Eigenschaften vom parametrisierten Standard-Iterator:

```
template <class T>
class istream_iterator : public iterator<input_iterator_tag, T> {
  // ...
};
```

STL Container

- Container dienen der Speicherung von Daten
- Unterschieden werden

Sequenzen: Die Elemente im Container haben eine lineare Ordnung. (Beispiele: vector und list).

Assoziative Container: Erlauben den Zugriff auf Elemente mit Hilfe eines Schlüssels. Dabei werden zwei Unterfälle unterschieden:

- Sortierte Assoziative Container: Die Daten werden bereits beim Einfügen richtig einsortiert → der Container liegt immer in sortierter Reigenfolge vor
- Ungeordente Assoziative Container: Die Daten werden nicht sortiert, sondern mit Hilfe eines Streuspeicherverfahrens abgelegt (später mehr).

STL Container

Übersicht über die STL Container:

Container-Art	Header	Klassen-Template
Sequenzen	<array> <deque> <list> <queue> <stack> <vector>	array deque list queue stack vector
Sortierte Assoziative Container	<map> <set>	map multimap set multiset
Ungeordnete Assoziative Container	<unordered_map> <unordered_set>	unordered_map unordered_multimap unordered_set unordered_multiset
Spezialfälle	<bitset> <vector> <queue>	bitset vector<bool> priority_queue

STL Container

Gemeinsame Eigenschaften

- Alle Container bieten dem Benutzer einen öffentlichen Satz von Datentypen und Methoden an.

Container-Datentypen:

Sei X der Datentyp des Containers (z.B. `vector<int>`), dann existieren:

Datentyp	Bedeutung
`X::value_type`	Typ der gespeicherten Elemente
`X::reference`	Referenz auf Container-Element
`X::const_reference`	dito, aber nur lesend verwendbar
`X::iterator`	Iterator
`X::const_iterator`	nur lesend verwendbarer Iterator
`X::difference_type`	vorzeichenbehafteter integraler Typ
`X::size_type`	integraler Typ ohne Vorzeichen für Größenangaben

STL Container

Container-Methoden:

Rückgabetyp Methode	Bedeutung
`X()`	Standardkonstruktor; erzeugt leeren Container
`X(const X&)`	Kopierkonstruktor
`X(X&&)`	bewegender Konstruktor für temporäre Parameter (R-Werte)
`~X()`	Destruktor; ruft die Destruktoren aller Elemente des Containers auf
`X(il)`	`il` ist eine Initialisierungsliste, siehe Seite 741.
`iterator begin()`	gibt Iterator zurück, der auf das erste Element des Containers, falls vorhanden, verweist; andernfalls wird `end()` zurückgegeben
`const_iterator begin()`	dito, aber mit diesem Iterator kann das Element nicht verändert werden
`const_iterator cbegin()`	dito (siehe Text)
`iterator end()`	Position *nach* dem letzten Element
`const_iterator end()`	dito
`const_iterator cend()`	dito (siehe Text)
`size_type size()`	Größe des Containers = Anzahl der aktuell gespeicherten Elemente = `(end() - begin())`
`size_type max_size()`	maximal mögliche Größe des Containers
`bool empty()`	`(size() == 0)` bzw. `(begin() == end())`
`void swap(X&)`	Vertauschen mit Argument-Container
`X& operator=(const X&)`	Zuweisungsoperator

STL Container

`begin()`, `end()`, `cbegin`, `cend()`

Wie kann man im einfachsten Fall über einen Container iterieren?

```
#include <iostream>
#include <vector>

int main() {
  std::vector<int> v;
  v.push_back(1);  // Elemente befüllen
  v.push_back(2);
  v.push_back(10);
  v.push_back(11);
  v.push_back(12);

  std::vector<int>::const_iterator it;

  for (it = v.begin(); it != v.end(); ++it) {
    std::cout << *it << " ";
  }
  std::cout << std::endl;
  return 0;
}
```

```
1 2 10 11 12
```

Hier wird ein `const-iterator` verwendet, da auf den `vector` nur lesend zugegriffen wird.

STL Container

Wie welche Verbesserungen bietet **C++11** in diesem einfachen Fall?

- Der neue Datentyp `auto` passt sich automatisch dem zugewiesenen Wert an!
- Initialisierungslisten sind möglich!
- Neue Kurzform für `for`-Schleifen (ähnlich wie foreach in Java)

```
int main() {
  std::vector<int> v = {1,2,10,11,12};

  for (auto it = v.cbegin(); it != v.cend(); ++it) {
    std::cout << *it << " ";
  }
  std::cout << std::endl;

  for (auto it : v) {   // Noch kürzer !!!
    std::cout << it << " ";
  }
  std::cout << std::endl;
  return 0;
}
```

```
1 2 10 11 12
1 2 10 11 12
```

Hier werden die neuen Methoden `cbegin()` und `cend()` eingesetzt, die immer einen `const_iterator` zurückliefern!

STL Container

Weitere allgemeine Bemerkungen

- Container (außer unsortierten assoziativen Containern und der `priority_queue` können mit relationalen Operatoren vergleichen werden (lexikographischer Vergleich).
  Unterstützte Operatoren sind `==` `!=` `<` `<=` `>` `>=`
- Container, die einen bidirektionalen Iterator unterstützen (z.B. eine doppelt verkettete Liste: `list`), unterstützen beim Iterator sowohl den `++` als auch den `--` Operator.
  Alternativ kann auch ein 'reverse Iterator' benutzt werden, der mit dem `++` Operator den Container rückwärts durchläuft.

```
// Diese Methoden liefern das LETZTE Element zurück
const_reverse_iterator rbegin()  const
const_reverse_iterator crbegin() const
reverse_iterator rbegin()

// Diese Methoden liefern das Element VOR dem ersten Element zurück
const_reverse_iterator rend()  const
const_reverse_iterator crend() const
reverse_iterator rend()
```

STL Container

Sequenz-Container

- Grundlegende Typen sind `list`, `vector` und `queue`.

Gemeinsame Methoden (zusätzlich zu den schon vorgestellten!):

Rückgabetyp Methode	Bedeutung
`X(n, t)`	Erzeugt eine Sequenz mit `n` Kopien von `t`.
`X(i, j)`	Erzeugt eine Sequenz aus den Elementen des Intervalls `[i, j)`.
`iterator emplace(p, args)`	Fügt ein Objekt, das mit den Parametern `args` konstruiert wird, vor `p` ein (siehe Seite 742). Der zurückgegebene Iterator zeigt auf die eingefügte Kopie.
`iterator insert(p, t)`	Fügt eine Kopie von `t` vor `p` ein. Der zurückgegebene Iterator zeigt auf die eingefügte Kopie.
`void insert(p, n, t)`	Fügt `n` Kopien von `t` vor `p` ein.
`void insert(p, i, j)`	Fügt Kopien der Elemente im Bereich `[i, j)` vor `p` ein. Die Iteratoren `i` und `j` dürfen nicht in den aufnehmenden Container verweisen.
`void insert(p, il)`	`il` ist eine Initialisierungsliste.
`iterator erase(q)`	Löscht das Element, auf das `q` zeigt. Der zurückgegebene Iterator zeigt anschließend auf das `q` folgende Element, wenn es existiert, ansonsten wird ein End-Iterator (`end()`) zurückgegeben.
`iterator erase(p, q)`	Löscht alle Elemente des Bereichs `[p, q)`. Der zurückgegebene Iterator zeigt anschließend auf das `q` folgende Element, wenn es existiert, ansonsten wird ein End-Iterator zurückgegeben.
`void clear()`	Löscht alle Elemente.
`void assign(i, j)`	Alle Elemente löschen, danach Kopie von `[i, j)` einfügen. `i` und `j` dürfen nicht in den Container verweisen.
`void assign(il)`	`il` ist eine Initialisierungsliste.
`void assign(n, t)`	Alle Elemente löschen, danach `n` Kopien von `t` einfügen. Dabei darf `t` keine Referenz auf ein Element des Containers sein.

STL Container

`stl::vector`

- Speichert die Elemente hintereinander im Arbeitsspeicher
- Unterstützt einen '*random-access*'-Iterator.
- Zusätzliche Datentypen:

Datentyp	Bedeutung
`pointer`	Zeiger auf Vektor-Element
`const_pointer`	dito, aber nur lesend verwendbar

- Zusätzliche Methoden:

- **`const_reference front() const`** und **`reference front()`** liefern eine Referenz auf das erste Element.
- **`const_reference back() const`** und **`reference back()`** liefern eine Referenz auf das letzte Element.
- **`const_pointer data() const`** und **`pointer data()`** liefern einen Zeiger auf das erste Element. Es gilt also `data() == &front()`, falls der Vektor nicht leer ist. Der Zeiger erlaubt den Zugriff auf die Daten des Vektors wie bei einem C-Array.

STL Container

- **`const_reference operator[](size_type n) const`** und **`reference operator[](size_type n)`**
  geben eine Referenz auf das n-te Element zurück.
- **`const_reference at(size_type n) const`** und **`reference at(size_type n)`**
  geben eine Referenz auf das *n*-te Element zurück. Der Unterschied zum vorhergehenden `operator[]()` besteht in der Prüfung, ob *n* in einem gültigen Bereich liegt. Falls nicht, d.h. *n* ist ≥ `size()`, wird eine `out_of_range-Exception` geworfen.
- **`void emplace_back(args)`**
  fügt ein Objekt, dessen Konstruktor mit den Parametern args aufgerufen wird, am Ende ein.
- **`void push_back(const T& t)`**
  fügt t am Ende ein.
- **`void pop_back()`**
  löscht das letzte Element.

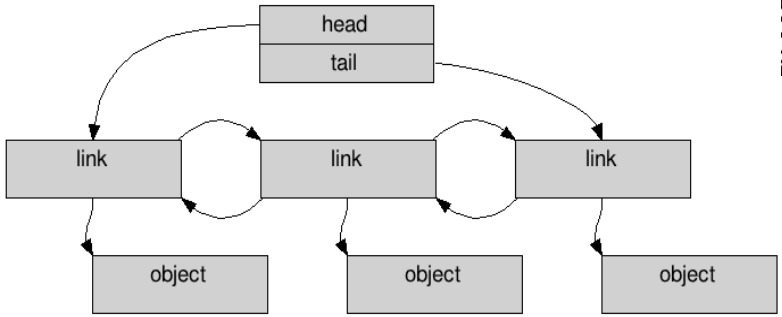
STL Container

- **`void resize(size_type n, T t = T())`**
  Vektorgröße ändern. Dabei werden *n* - `size()` Elemente `t` am Ende hinzugefügt bzw. `size()` - *n* Elemente am Ende gelöscht, je nachdem, ob *n* kleiner oder größer als die aktuelle Größe ist. Die Zeitkomplexität ist O(| size() – n |).
- **`void reserve(size_type n)`**
  Speicherplatz reservieren, sodass der verfügbare Platz (Kapazität) größer als der aktuell benötigte ist. Zweck: Vermeiden von Speicherbeschaffungsoperationen während der Benutzung des Vektors. Die Zeitkomplexität ist O(n).
- **`size_type capacity() const`**
  gibt den Wert der Kapazität zurück (siehe `reserve()`). `size()` ist immer kleiner oder gleich `capacity()`.

STL Container

stl::list

- Implementiert eine doppelt verkettet Liste.
- Unterstützt einen *bidirektionalen* Iterator.
- Zusätzliche Methoden:



- **`const_reference front() const`** und **`reference front()`** liefern eine Referenz auf das erste Element einer Liste
- **`const_reference back() const`** und **`reference back()`** liefern eine Referenz auf das letzte Element einer Liste
- **`void emplace_front(args)`** fügt ein Objekt, dessen Konstruktor mit den Parametern `args` aufgerufen wird, am Anfang ein.
- **`void emplace_back(args)`** fügt ein Objekt, dessen Konstruktor mit den Parametern `args` aufgerufen wird, am Ende ein.

STL Container

- `void push_front(const T& t)` fügt `t` am Anfang ein.
- `void pop_front()` löscht das erste Element.
- `void push_back(const T& t)` fügt t am Ende ein.
- `void pop_back()` löscht das letzte Element.
- `void remove(const T& t)`
  entfernt alle Elemente, die gleich dem übergebenen Element `t` sind. Die Zeitkomplexität ist O(*n*).
- `template<class Praedikat> void remove_if(Praedikat P)`
  entfernt alle Elemente, auf die das Prädikat zutrifft.
  Die Zeitkomplexität ist O(*n*).
- `void resize(size_type neu, T t = T())`
  Listengröße ändern. Dabei werden `neu - size()` Elemente `t` am Ende hinzugefügt bzw. `size()- neu` Elemente am Ende gelöscht, je nachdem, ob `neu` kleiner oder größer als die aktuelle Größe ist. Die Zeitkomplexität ist O(`|n-neu|`).

STL Container

- **`void reverse()`**
  kehrt die Reihenfolge der Elemente in der Liste um (Zeitkomplexität O(*n*)).
- **`void sort()`**
  sortiert die Elemente in der Liste. Die Zeitkomplexität ist O(*n*log*n*). Sortierkriterium ist der für die Elemente definierte Operator `<`.
- **`template<class Compare> void sort(Compare cmp)`**
  wie `sort()`, aber mit dem Sortierkriterium des Compare-Objekts `cmp`. Das Compare-Objekt ist ein Funktionsobjekt (sog. *Functor*).
- **`void unique()`**
  löscht gleiche aufeinanderfolgende Elemente bis auf das erste (Zeitkomplexität O(*n*)). Anwendung auf eine sortierte Liste bedeutet, dass danach kein Element mehrfach auftritt.
- **`template<class binaeresPraedikat> void unique(binaeresPraedikat P)`**
  dito, nur dass statt des Gleichheitskriteriums ein anderes binäres Prädikat genommen wird.

STL Container

- **`void merge(list& L)`**
  Verschmelzen zweier sortierter Listen (Zeitkomplexität O(*n1* + *n2*)). Die aufgerufene Liste `L` ist anschließend leer, die aufrufende enthält nachher alle Elemente.
- **`template<class Compare> void merge(list& L, Compare cmp)`**
  wie vorher, aber für den Vergleich von Elementen wird ein Compare-Objekt genommen.
- **`void splice(iterator pos, list& x)`**
  fügt den Inhalt von Liste `x` vor pos ein. `x` ist anschließend leer.
- **`void splice(iterator p, list&x, iterator i)`**
  Fügt Element `*i` aus `x` vor `p` ein und entfernt `*i` aus `x`.
- **`void splice(iterator pos, list& x, iterator first, iterator last)`**
  fügt Elemente im Bereich [first, last) aus `x` vor `pos` ein und entfernt sie aus `x`. Bei dem Aufruf für dasselbe Objekt (das heißt `&x == this`) wird konstante Zeit benötigt, ansonsten ist der Aufwand von der Ordnung O(`last - first`). `pos` darf nicht im Bereich [first, last) liegen.

STL Container

`stl::deque`

- engl. '*Double ended queue*': Warteschlange, die das Hinzufügen und Entnehmen von Elementen am Anfang und Ende erlaubt.
- Unterstützt einen '*random-access*'- Iterator.
- Zusätzliche Methoden:

- **`const_reference front() const`** und **`reference front()`** liefern eine Referenz auf das erste Element eines Deque-Objekts.
- **`const_reference back() const`** und **`reference back()`** liefern eine Referenz auf das letzte Element eines Deque-Objekts.
- **`const_reference operator[](size_type n) const`** und **`reference operator[](size_type n)`** geben eine Referenz auf das n-te Element zurück.
- **`const_reference at(size_type n) const`** und **`reference at(size_type n)`** geben eine Referenz auf das *n*-te Element zurück. Der Unterschied zum vorhergehenden `operator[]()` wie bei `vector`.

STL Container

- **`void emplace_front(args)`**
  fügt ein Objekt, dessen Konstruktor mit den Parametern `args` aufgerufen wird, am Anfang ein.
- **`void emplace_back(args)`** fügt ein Objekt, dessen Konstruktor mit den Parametern `args` aufgerufen wird, am Ende ein.
- **`void push_front(const T& t)`** fügt `t` am Anfang ein.
- **`void pop_front()`** löscht das erste Element.
- **`void push_back(const T& t)`** fügt `t` am Ende ein.
- **`void pop_back()`** löscht das letzte Element.
- **`void resize(size_type n, T t = T())`**
  Deque-Größe ändern. Dabei werden *n* - `size()` Elemente `t` am Ende hinzugefügt bzw. `size()` - *n* Elemente am Ende gelöscht, je nachdem, ob *n* kleiner oder größer als die aktuelle Größe ist.

STL Container

stl::stack

- Container, der das Ablegen und Entnehmen von Elementen an einer 'Seite' erlaubt.
- Abstrakter Container, der intern `vector`, `list` oder `deque` verwendet. Default-Wert ist `deque`.

Deklaration:

```
template<typename T, class Container = deque<T> > class stack;
```

```
// mit deque realisierter Stack (Voreinstellung)
stack<double> Stack1;
// mit vector realisierter Stack
stack<double, vector<double> > Stack2;
```

STL Container

Methoden (+ relationale Operatoren):

- `stack(const Container& cont = Container())`
  Konstruktor. Ein Stack kann mit einem bereits vorhandenen Container initialisiert werden. Container ist der Typ des Containers. In diesem Fall ist die Zeitkomplexität O(cont.size()).
- `bool empty() const` gibt zurück, ob der Stack leer ist.
- `size_type size() const` gibt die Anzahl der Elemente im Stack zurück.
- `const value_type& top() const` und `value_type& top()` geben das oberste Element zurück.
- `void push(const value_type& x)`
  legt das Element `x` auf dem Stack ab.
- `void pop()`
  entfernt das oberste Element vom Stack.

STL Container

Beispiel:

```
#include<stack>
#include<iostream>
using namespace std;
int main() {
  int numbers[] = {1, 5, 6, 0, 9, 1, 8, 7, 2};
  const int COUNT = sizeof(numbers)/sizeof(int);
  stack<int> einStack;
  cout << "Zahlen auf dem Stack ablegen : " << endl;
  for(int i = 0; i < COUNT; ++i) {
    cout.width(6);
    cout << numbers[i];
    // protokollieren
    einStack.push(numbers[i]);
  }
  cout << endl;
  cout << "Zahlen vom Stack holen ( umgekehrte Reihenfolge ! ) , "
          " anzeigen und löschen : " << endl;
  while(!einStack.empty()) {
    cout.width(6);
    cout << einStack.top(); / / obersten Wert anzeigen
    einStack.pop(); // Wert löschen
  }
  cout << endl;
}
```

STL Container

stl::queue

- Ein Warteschlangen-Container: Erlaubt die Ablage von Objekten auf der einen, und Entnahme der Objekte auf der anderen Seite.
- Abstrakter Container, der intern `list` oder `deque` verwendet. Default-Wert ist `deque`.

Deklaration:

```
template<typename T, class Container = deque<T> > class queue;
```

Methoden (+ relationale Operatoren):

- `queue(const Container& cont = Container())`
- `bool empty() const`
- `size_type size() const`
- `const value_type& front() const` und `value_type& front()`
- `const value_type& back() const` und `value_type& back()`
- `void push(const value_type& x)`
- `void pop()`

STL Container

Stl::priority_queue

- Prioritätsgesteuerter Warteschlangen-Container: Jedes Element hat eine Priorität, und wird schon beim Einfügen in den Container an der entsprechenden Stelle eingefügt.
- Vergleich der Elemente mit dem `<`-Operator (default) oder Functor.
- Abstrakter Container, der intern `vector` oder `deque` verwendet. Default-Wert ist `deque`.
- Zum effizienten Einfügen von mittleren Elementen wird intern ein Heap verwendet.

Deklaration:

```
template<class T, class Container = deque<T>,
         class Compare = less<typename Container::value_type> >
class priority_queue;
```

STL Container

Methoden (+ Methoden von queue):

- `priority_queue(const Compare& cmp = Compare()),`
  `const Container& = Container())`
  Konstruktor. Eine Priority_Queue kann mit einem bereits vorhandenen Container initialisiert werden. Container ist der Typ des Containers. In diesem Fall ist die Zeitkomplexität O(*cont.size()*). `cmp` ist das Funktionsobjekt, mit dem verglichen wird.
  `less<Container::value_type>` wird angenommen, falls kein Typ angegeben ist.

- `template<class InputIterator>`
  `priority_queue(InputIterator first, InputIterator last,`
  `const Compare& cmp = Compare()),`
  `const Container& cont = Container())`
  Dieser Konstruktor unterscheidet sich von dem vorhergehenden, indem die Elemente im Bereich [first, last) eines anderen Containers bei der Konstruktion zusätzlich eingefügt werden. Die Zeitkomplexität ist O(*last – first + cont.size()*).

STL Container

- `bool empty() const`
  gibt zurück, ob die Priority_Queue leer ist.
- `size_type size() const`
  gibt die Anzahl der in der Priority_Queue befindlichen Elemente zurück.
- `const value_type& top() const`
  gibt das erste Element zurück, d.h. das mit der größten Priorität.
- `void push(const value_type& x)`
  fügt das Element x ein. Die Zeitkomplexität ist O(logn).
- `void pop()`
  entfernt das erste Element. Die Zeitkomplexität ist O(logn).

Für die Priority_Queue-Klasse gibt es keine relationalen Operatoren.

STL Container

`stl::array`

- Wrapper für Standard C-Arrays (ab C++11)
- Im Unterschied zu `vector` feste Anzahl von Elementen
- Die Methoden unterscheiden sich allerdings von den Standard-Methoden der Sequenz-Container

Deklaration:

```
template<typename T, size_t N> class array;
```

STL Container

Methoden:

- `void assign(const T& t)`
  weist allen Elementen des Arrays eine Kopie von t zu.
- `const_reference front() const` und `reference front()`
  liefern eine Referenz auf das erste Element.
- `const_reference back() const` und `reference back()`
  liefern eine Referenz auf das letzte Element.
- `const_pointer data() const` und `pointer data()`
  liefern einen Zeiger auf das erste Element. Es gilt also `data() == &front()`, falls das Array nicht leer ist.
- `const_reference operator[](size_type n) const` und `reference operator[](size_type n)`
  geben eine Referenz auf das *n*-te Element zurück.
- `const_reference at(size_type n) const` und `reference at(size_type n)`
  geben eine Referenz auf das n-te Element zurück (s.o.)

STL Container

Beispiel:

```
#include <iostream>
#include <array>

int main() {

  int a_1[5] = { 1, 2, 4, 8, 16 };

  std::array<int, 5> a_2 = { 1, 2, 4, 8, 16 };

  a_1[0] = 13;
  a_2[0] = 13;

  for (auto & i : a_2) {
    i *= 2;
  }

  for (auto i : a_2) {
    std::cout << i << " ";
  }

}
```

```
26 4 8 16 32
```

FH Aachen
Fachbereich 9 Medizintechnik und Technomathematik
Prof. Dr.-Ing. Andreas Terstegge
Straße Nr.
PLZ Ort
T +49. 241. 6009 53813
F +49. 241. 6009 53119
Terstegge@fh-aachen.de
www.fh-aachen.de