Objektorientierte Programmierung mit C++

Ulrich Breymann
Hochschule Bremen

Kurs auf Basis des Buchs
C++ Einführung und professionelle Programmierung, 7. Auflage

Datei vom 6. Januar 2003

Hinweise

Literatur

Dieses Dokument basiert auf meinem Buch Ulrich Breymann: *C++ - Einführung und professionelle Programmierung*. Hanser Verlag, siehe auch
http://www.informatik.hs-bremen.de/~brey/cppbuch.html
Die Kapitelnummerierung entspricht der des Buchs, bis auf das letzte Kapitel, das sich auf das folgende STL-Buch bezieht.

Ulrich Breymann: *Komponenten entwerfen mit der C++ STL*. Addison-Wesley; als pdf-Datei frei erhältlich, siehe
http://www.informatik.hs-bremen.de/~brey/stlb.html

ISO/IEC 14882-1998, *International Standard -Programming Language -C++*, elektronisch über www.ansi.org erhältlich.

Scott Meyers: *Effektiv C++ programmieren*. Addison-Wesley

Scott Meyers: *More Effective C++*. Addison-Wesley

Bjarne Stroustrup: *The C++ -Programming Language*, Addison-Wesley

Roter Text ist i.a. klickbar (z.B. Inhaltsverzeichnis) und führt direkt zur gewünschten Stelle.

Inhalt

1. Einführung

Aufgaben eines Rechners / Programms z.B.:

- eine bestimmte Summe Geld von einem Konto auf ein anderes transferieren
- eine Ampelanlage steuern
- ein Rechteck auf dem Bildschirm zeichnen.

Das *was* und das *womit* gehören stets zusammen.

→ Simulation von Objekten der realen Welt durch Abbildung im Rechner

→ Die Abbildungen heißen selbst wieder *Objekte*.

→ Objekte im Sinne der OOP sind *Abstraktionen* von Dingen der realen Welt.

Eine Menge von Objekten wird beschrieben durch eine *Klasse*. Alle Objekte einer Klasse haben

- gleiche Eigenschaften und
- gleiches Verhalten

Notation für Botschaften:
*Objektname.Anweisung(*gegebenenfalls *Daten)*.

Ampel.Blinken(gelb)
Ampel.Ausschalten(rot)
Ampel.Einschalten(grün)
Rechteck.Zeichnen(Position, Abmessungen)
Rechteck.Verschieben(5cm)

Klassen

Klasse = *Beschreibung* von Objekten

Daten, bestehend aus Namen und Angaben zum Datenformat:

Inhaber (Folge von Buchstaben)
Kontonummer (Zahl)
Betrag (Zahl)
Dispo-Zinssatz (in %)

Operationen:

überweisen (Ziel-Kontonummer, Betrag)
bar_abheben(Betrag)
einzahlen(Betrag)

konkrete *Objekte*: Konten K1 und K2

Beschreibung	Konto K1	Konto K2
Inhaber	Roberts, Julia	Constantine, Eddy
Kontonummer	12573001	54688490
Betrag	-200,30	1222,88
Dispo-Zinssatz	13,75	13,75

Aufträge:
K1.überweisen(54688490, 1000.00)
K2.abheben(22)

→ Objekte als Handelnde

- Die *Beschreibung* eines tatsächlichen Objekts gibt seine innere *Datenstruktur* und die möglichen *Operationen oder Methoden* an, die auf die inneren Daten anwendbar sind.
- Zu *einer* Beschreibung kann es keins oder beliebig viele Objekte (Instanzen) geben.

Prinzip: Datenabstraktion (⇒ ADT)

Kursaufbau

- Daten
  Datentypen, -strukturen
- Operationen
  Kontrollstrukturen, Funktionen
- Klassen
  ADT
  + Vererbung
  + Polymorphismus

2. Grundlegende Begriffe

– das erste Programm

– Datentypen und Operatoren

– Kontrollstrukturen

– zusammengesetzte Datentypen: Array, String

2.1 Das erste Programm

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

int main( ) {
    int summe,a,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;
}
```

C++ : Groß- und Kleinschreibung werden unterschieden!

PaSCaL /FoRTrAn: gRoß OdEr klEIn iSt EGaL!

C++

```
#include<iostream>

int main( )
{

int summe,a,b;

// Lies die Zahlen..
cout « "a..";
cin » a » b;

/* Berechne.. */
summe = a+b;

// Zeige ..
cout « "Summe="
     « summe;

}
```

C

```
#include<stdio.h>

int main( )
{

int summe,a,b;

/* Lies ...*/
printf("%s","a..");
scanf("%d %d",&a,&b);

/* Berechne.. */
summe = a+b;

/* Zeige ..*/
printf("Summe %d",
       summe);
return 0;
}
```

PASCAL

```
PROGRAM main();
BEGIN

summe,a,b,:INTEGER;

(* Lies ... *)
WRITE('a..');
READLN(a,b);

{ Berechne.. }
summe := a+b;

(* Zeige ..*)
WRITE('Summe=',
      summe);

END
.
```

Struktur eines C++ -Programms:

Compiler-Instruktionen (zum Beispiel `#include`)
Deklaration von globalen Objekten
Funktionsprototypen (Unterprogrammschnittstellen)

Implementierung, eigentliche Problemlösung:
`main( ) {`
Folge von Anweisungen und Deklarationen
`}`

Funktions-Definitionen (Unterprogrammcode)

Ein Programm besteht meistens aus vielen Dateien.

2.2 Einfache Datentypen und Operatoren

Datentypen sind definiert durch ihren Wertebereich sowie die mit diesen Werten möglichen Operationen.

2.2.1 Ausdruck

Ein Ausdruck besteht aus einem oder mehreren Operanden, die miteinander durch Operatoren verknüpft sind. Die Auswertung eines Ausdrucks resultiert in einem Wert, der an die Stelle des Ausdrucks tritt. Der einfachste Ausdruck besteht aus einer einzigen Konstanten, Variablen oder Literal. Die Operatoren müssen zu den Operanden passen, die zu bestimmte Datentypen gehören. Beispiele:

Drei einfache Ausdrücke:
17
$1 + 17$
»Textliteral«

$a = 1 + 17$ ist ein zusammengesetzter Ausdruck. Die Operanden 1 und 17 sind Zahl-Literale, die hier ganze Zahlen repräsentieren und die durch den $+$-Operator verknüpft werden. Der resultierende Wert 18 wird dem Objekt a zugewiesen. Der Wert des gesamten Ausdrucks ist a.

2.2.2 Ganze Zahlen

Typische Werte für die Anzahl der Bits:

`short`	16 Bits
`int`	32 Bits (oder 16 Bits)
`long`	64 Bits (oder 32 Bits)

Es gilt: Bits(`short`) $\leq$ Bits(`int`) $\leq$ Bits(`long`)

Ganze Zahlen können auf drei Arten dargestellt werden:

1. Wenn eine Zahl mit einer 0 beginnt, wird sie als *Oktalzahl* interpretiert, zum Beispiel 0377.
2. Wenn eine Zahl mit 0x oder 0X beginnt, wird sie als *Hexadezimalzahl* interpretiert, zum Beispiel 0xAFFE = 0Xaffe = 45054 dezimal.
3. dezimal wie üblich

Operatoren für ganze Zahlen

Operator	Beispiel	Bedeutung
arithmetische Operatoren:		
`+`	`+i`	unäres Plus
`-`	`-i`	unäres Minus
`++`	`++i`	vorherige Inkrementierung
	`i++`	nachfolgende Inkrementierung
`-`	`-i`	vorherige Dekrementierung
	`i-`	nachfolgende Dekrementierung
`+`	`i+2`	binäres Plus
`-`	`i-5`	binäres Minus
`*`	`5*i`	Multiplikation
`/`	`i/6`	Division
`%`	`i % 4`	Modulo
`=`	`i = 3+j`	Zuweisung

Operator	Beispiel	Bedeutung
Kurzform-Operatoren:		
*=	`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`

Operator	Beispiel	Bedeutung
relationale Operatoren:		
`<`	`i < j`	kleiner als
`>`	`i > j`	größer als
`<=`	`i <= j`	kleiner gleich
`>=`	`i >= j`	größer gleich
`==`	`i == j`	gleich
`!=`	`i != j`	ungleich

Operator	Beispiel	Bedeutung
Bit-Operatoren:		
`<<`	`i << 2`	Linksschieben
`>>`	`i >> 1`	Rechtsschieben
`&`	`i & 7`	bitweises UND
`^`	`i^7`	bitweises XOR
`|`	`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`
`|=`	`i |= 3`	`i = i | 3`

2.2.3 Reelle Zahlen

Reelle Zahlen werden (ungenau) dargestellt durch 3 Datentypen

Typ	Bits	Zahlenbereich	Stellen
`float`	32	$\pm 3.4 * 10^{\pm 38}$	7
`double`	64	$\pm 1.7 * 10^{\pm 308}$	15
`long double`	80	$\pm 3.4 * 10^{\pm 4932}$	19

(Bitzahlen implementationsabhängig)

Syntax:

Vorzeichen (optional)
Vorkommastellen
Dezimalpunkt
Nachkommastellen
e oder E und Ganzzahl-Exponent (optional)
Suffix f,F oder l,L (optional, Zahlen ohne Suffix sind `double`)

Es können wegfallen:

- entweder Vorkommastellen oder Nachkommastellen (aber nicht beide)
- entweder Dezimalpunkt oder e (oder E) mit Exponent (aber nicht beide)

Beispiele: -123.789e6f 1.8E6 88.009 1e-03 1.8L

Operatoren für `float`- und `double`-Zahlen

Operator	Beispiel	Bedeutung
`+`	`+d`	unäres Plus
`-`	`-d`	unäres Minus
`+`	`d+2`	binäres Plus
`-`	`d-5`	binäres Minus
`*`	`5*d`	Multiplikation
`/`	`d/6`	Division
`=`	`d = 3+j`	Zuweisung
`*=`	`d *= 3`	`d = d * 3`
`/=`	`d /= 3`	`d = d / 3`
`+=`	`d += 3`	`d = d + 3`
`-=`	`d -= 3`	`d = d - 3`
`<`	`d < f`	kleiner als
`>`	`d > f`	größer als
`<=`	`d <= f`	kleiner gleich
`>=`	`d >= f`	größer gleich
`==`	`d == f`	gleich
`!=`	`d != f`	ungleich

```
#include<iostream>
#include<cmath> // math. Bibliothek einbinden
using namespace std;

// Berechnung mathematischer Ausdrücke
int main() {
    float x;
    cout << "x eingeben:";
    cin >> x;
    cout << " x       = " << x       << endl;
    cout << " fabs(x) = " << fabs(x) << endl;
    cout << " sqrt(x) = " << sqrt(x) << endl;
    cout << " sin(x)  = " << sin(x)  << endl; // Bogenmaß!
    cout << " exp(x)  = " << exp(x)  << endl;
    cout << " log(x)  = " << log(x)  << endl; // Basis e
}
```

Vorrangregeln (Auswahl)

Rang	Operatoren
1	`.` `->` `[ ]` `( )` `++` `--` (postfix)
2	`!` `+` `-` (unär) `++` `--` (präfix) `~` `&` (Adressoperator) `*` (Dereferenzierung) `sizeof`
4	`*` `/` `%`
5	`+` `-`
6	`<<` `>>`
7	`<` `>` `<=` `>=`
8	`==` `!=`
9	`&` (bitweises UND)
10	`^` (bitweises exklusiv-ODER)
11	`|` (bitweises ODER)
12	`&&` (logisches UND)
13	`&&` (logisches ODER)
14	`?:`
15	alle Zuweisungsoperatoren =, +=, <<= usw.
16	`,`

Strukturierung mit Klammern ().

Klammerausdrücke werden zuerst ausgewertet.

ABER: Die *Reihenfolge* der Auswertung mehrerer Klammer- oder Unterausdrücke ist *undefiniert*!

```
int total = 0;
sum = (total=3) + (++total);    // Fehler !

int i = 2;
i = 3*i++;                      // Fehler !
```

2.2.4 Komplexe Zahlen

Komplexe Zahlen bestehen aus dem Real- und dem Imaginärteil, die beide von einem der möglichen Typen für reelle Zahlen sein können (`float`, `double`, `long double`). Der Standarddatentyp für komplexe Zahlen ist deshalb kein Grunddatentyp, sondern zusammengesetzt.

```
#include<iostream>
#include<complex>                    // Header für komplexe Zahlen
using namespace std;
// Beispiele für komplexe Zahlen
// Anstatt double sind auch float und long double möglich.
int main() {
   complex<double> c1;      // komplexe Zahl 0.0 + 0.0i
   complex<double> c2(1.2, 3.4); // (1.2 + 3.4i) erzeugen
   cout << c2 << endl;          // Standard-Ausgabeformat: (1.2,3.4)
   c1 += c2 + c1;               // beispielhafte Rechenoperationen
   c1 = c2 * 5.0;
   double re = c1.real();        // Realteil ermitteln
   cout << re << endl;           // und ausgeben
   cout << c1.imag() << endl;    // Imaginärteil direkt ausgeben
```

2.2.5 Zeichen

3 verschiedene Typen:

`char`	je nach Implementation
`signed char`	mit Vorzeichen (-128..+127)
`unsigned char`	ohne Vorzeichen (0..255)

```
const char stern ='*';
char a;
a = 'x';
a = stern;
```

besondere Zeichenkonstanten

Zeichen	Bedeutung
`\a`	Signalton
`\b`	Backspace
`\f`	Seitenvorschub
`\n`	neue Zeile, Wirkung wie `\r` und `\v`
`\r`	Zeilenrücklauf
`\t`	Tabulator
`\v`	Zeilensprung
`\\`	Backslash
`\'`	'
`\"`	"
	⊗ = Platzhalter:
`\`⊗	⊗ = String aus Oktalziffern, zum Beispiel \377
`\x`⊗, `\X`⊗	⊗ = Zeichenfolge aus Hex-Ziffern, zum Beispiel \xDB

```
i = 66;
c = static_cast<char>(i); // Typumwandlung
cout << c;                // 'B'
c = '1';                  // Ziffernzeichen '1'
i = static_cast<int>(c);  // Position in der ASCII-Tabelle:
cout << i;                // 49
```

Gemischte Verwendung von `char` und `int`:

→ implizite Typwandlung durch Compiler

Typ von Zeichenkonstanten: `char` in C++, `int` in C

Operatoren für Zeichen

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

2.2.6 Logischer Datentyp

Datentyp: `bool`

mögliche Werte: `true false`

gegebenenfalls automatische Typwandlung:
`true` $\longrightarrow 1$
`false` $\longrightarrow 0$

Beispiel:

```
bool grossBuchstabe;
char c;
cin >> c;
grossBuchstabe =  c >='A'  &&  c <='Z';
cout << grossBuchstabe;  // Typwandlung in int
```

Operatoren:

Operator	Beispiel	Bedeutung
`!`	`!i`	logische Negation
`&&`	`a && b`	logisches UND
`||`	`a || b`	logisches ODER
`==`	`d == f`	gleich
`!=`	`d != f`	ungleich
`=`	`a = a && b`	Zuweisung

2.2.7 Referenzen

Referenz = *Verweis* auf ein Objekt oder Aliasname für ein Objekt

```
int i,j;
int &r = i;          // Referenz auf i ( r ist Alias für i)

i  = 100;            // ändert i
r  = 10;             // ändert i
```

- Eine Referenz wird genau wie eine Variable benutzt (der Compiler erledigt den Rest der Arbeit).
- Referenzen müssen bei der Deklaration initialisiert werden.

2.3 Gültigkeitsbereich und Sichtbarkeit

Regeln:

- Namen sind nur *nach der Deklaration* und nur *innerhalb des Blocks* gültig, in dem sie deklariert wurden.
- Sie sind *lokal* bezüglich des Blocks.
- Namen von Variablen sind auch gültig für innerhalb des Blocks neu-angelegte innere Blöcke.
- Die Sichtbarkeit (englisch *visibility*) zum Beispiel von Variablen wird eingeschränkt durch die Deklaration von Variablen gleichen Namens. Für den Sichtbarkeitsbereich der inneren Variablen ist die äußere unsichtbar.

```
{  // Blockbeginn
   int a = 1;
   {  // Blockbeginn
      int b = 2;
      cout << a << b;        // 1 2

      int a = 10;            // erstes a wird unsichtbar
      cout << a << b;        // 10 2
   }  // Blockende

   cout << a;                // 1 (a wieder sichtbar)
   cout << b;                // Fehler!
}  // Blockende
```

2.4 Kontrollstrukturen

Eine Anweisung ist ein Ausdruck gefolgt von einem Semikolon. Kontrollstrukturen dienen zur Steuerung des Programmflusses.

2.4.1 Sequenz (Reihung)

```
a  = b + 1;
a += a;
cout << "\n Ergebnis =" << a;
```

2.4.2 Auswahl (Selektion, Verzweigung)

```
if  (Bedingung)
    Anweisung1
```

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

```
if    (Bedingung {)
      Anweisung_a
      Anweisung_b
      Anweisung_c
}
else  Anweisung_d
```

Der `else`-Zweig kann ebenfalls ein Block sein.

Beispiele:

```
if(x < 100) cout << "x < 100";
else        cout << "x >= 100";

if(a > b) x = a-b;
else x = b-a;

if((c >= 'A') && (c <= 'Z'))
   grossBuchstabe=true;
else
   grossBuchstabe=false;
```

oder:

```
grossBuchstabe =  c >='A' &&  c <='Z';
```

Gleichwertig:

```
if(a) ....          if(a != 0) ....
if(!a) ....         if( a == 0) ....
```

Auswertung des Bedingungsausdrucks von links nach rechts.
Überspringen unnötiger Berechnungen (short cut evaluation)

Vorsicht bei Seiteneffekten!!

```
// Beispiel nur zur Übung! Im allgemeinen sollen
// unübersichtliche Seiteneffekte vermieden werden!

int i = 0, j = 2;      // stimmt's oder nicht?
if(i++ || j++) ++i;    // i==2 j==3

int i = 1, j = 2;
if(i++ || j++) ++i;    // i==3 j==2

int i = 0, j = 2;
if(i++ && j++) ++i;    // i==1 j==2

int i = 1, j = 2;
if(i++ && j++) ++i;    // i==3 j==3
```

Häufige Schreib-/Programmierfehler:

```
if(a = b)                  // Vermutlich anders gemeint!
    cout << "a ist gleich b";

if(a == b);                                  // Fehler!
   cout << "a ist gleich b";

if(x == 1)
   if(y == 1) cout << "x == y == 1 !";
else cout << "x != 1";                          // falsch

int i = -1;
unsigned u = 0;
if(u < i)
     cout << u << '<' << i << '\n';     // falsch
```

Bedingungsoperator ?:

Bedingung `?` *Ausdruck1* `:` *Ausdruck2*

Beispiel:

```
max = a > b ? a : b;
```

Das `if`-Äquivalent dazu ist:

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

2.4.3 Fallunterscheidungen (switch)

```
switch(Ausdruck) {
   case const1  : Anweisungen break;
   case const2  : Anweisungen break;
   case const3  : Anweisungen break;
   // ...
   default      : ErsatzAnweisungen;
}
```

Ausdruck muss den Typ `int` haben.

```
#include<iostream>
using namespace std;
int main( ) { // Erkennung römischer Ziffern
    int a;
    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;
    }
    if(a > 0) cout << "a =  " << a;
    else cout <<"keine römische Ziffer!";
}
```

2.4.4 Schleifen

Schleifen mit while

Syntax:

```
while(Bedingung) Anweisung
```

oder:

```
while(Bedingung) Block
```

Wirkung:

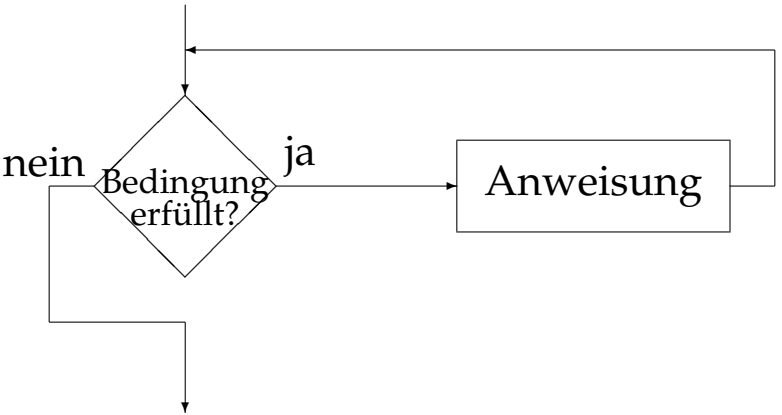


Abb. 2.1: Flussdiagramm für eine `while`-Anweisung

Schleifen können beliebig geschachtelt werden:

```
while(Bedingung1)
   while(Bedingung2)
   {    .....
        while(Bedingung3)
        {    .....
        }
   }
```

Beispiele

- unendliche Schleife: `while(true) Anweisung`
- Anweisung wird nie ausgeführt (unerreichbarer Programmcode):

```
while (false) Anweisung
```

- Summation der Zahlen 1 bis 99:

```
int sum = 0, n = 1, grenze = 99;
while(n <= grenze)  sum += n++;
```

Schleifen mit do while

Wirkung:

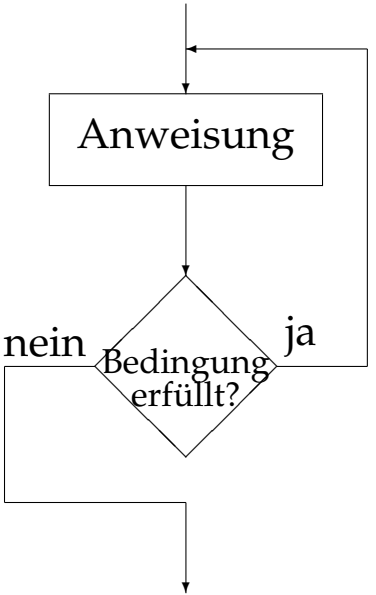


Abb. 2.2: Flussdiagramm für eine `do while`-Anweisung

Syntax:

```
do
        Anweisung
while (Bedingung);
```

oder mit einem Block:

```
do
{       Anweisung1
        Anweisung2
        // ...
} while (Bedingung);
```

typische Anwendung: Eingabe mit Plausibilitätskontrolle:

```
int Prozentsatz;
do {
   cout << "Bitte Prozentsatz eingeben (0..100):";
   cin >> Prozentsatz;
}  while(Prozentsatz < 0  || Prozentsatz > 100);

// weiterrechnen ...
```

Schleifen mit for

Syntax:

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

oder:

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

Bedeutung:

- Durchführung der Initialisierung, zum Beispiel Startwert für eine Laufvariable festlegen. Eine Laufvariable wird als Zähler benutzt, wie `i` in der folgenden Beispielschleife.
- Prüfen der Bedingung
- Falls die Bedingung wahr ist, Anweisung *und* Veränderung ausführen

Beispiel: ASCII-Tabelle von 65..69 ausgeben

```
for(int i = 65; i < 70; ++i)
    cout << i << " " << char (i) << '\n';
```

Regeln für die Gültigkeit von Schleifenvariablen:

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

// kompatible Schreibweise für ältere Compiler:
{
   int i;
   for(i = 0; i < 100; ++i)
   // ...
}
```

Beispiel : Fakultät berechnen

```
cout << "Fakultät berechnen. Zahl >= 0? :";
int n;
cin >> n;
unsigned long fak=1;
for(int i = 2; i <= n; ++i)
     fak *= i;
cout << n <<"!   =   " << fak
     << endl;
```

2.4.5 Kontrolle mit break und continue

`break`	Verlassen des Schleifenkörpers
`continue`	Rest des Schleifenkörpers überspringen

```
#include<iostream>
using namespace std;
int main( ) {   // Menu-Programm
    char c;
    while(true) {
        cout << "Wählen Sie: a,b, x=Ende : ";
        cin >> c;
        if(c == 'a') {
           cout << "Programm a\n";
           continue;            // zurück zur Auswahl
        }
        if(c == 'b') {
           cout << "Programm b\n";
           continue;            // zurück zur Auswahl
        }
        if(c == 'x') break;     // Schleife verlassen
        cout << "Falsche Eingabe! Bitte wiederholen!\n";
    }
    cout << "\n Programmende mit break\n";
}
```

Aufgaben

2.1 Schreiben Sie eine Schleife, die eine gegebene Zahl binär ausgibt.

2.2 Welche Fallstricke sind in den folgenden Schleifen verborgen?

a) `while(i > 0) k = 2*k;`

b) `while(i != 0) i = i - 2;`

c) `while(n != i) { ++i; n = 2*i;}`

2.3 Fünf Leute haben versucht, die Summe der Zahlen von 1 bis 100 zu berechnen. Beurteilen Sie die folgenden Lösungsvorschläge:

(a)
```
int n = 1, sum = 0;
while(n <= 100) {
    ++n;
    sum += n;
}
```

(b)
```
int n = 1, sum = 1;
while(n < 100)
   n += 1;
   sum += n;
```

(c)
```
int n = 100;
int sum = n*(n+1)/2;
```

(d)
```
int n = 1;
while(n < 100) {
    sum = 0;
    n = n+1;
    sum = sum + n;
}
```

(e)
```
int n = 1, sum = 0;
while(n <= 0100) {
    sum += n;
    ++n;
}
```

2.5 Benutzerdefinierte und zusammengesetzte Datentypen

2.5.1 Aufzählungstypen

ungünstig:

```
int farbe;   // rot == 0
             // grün == 1
             // blau == 2
             // gelb == 3
int Wochentag; // Sonntag == 0
               // Montag == 1 usw.
```

Nachteile :

- Die Bedeutung muss im Programm als Kommentar festgehalten werden.
- Zugeordnete Zahlen sind nicht eindeutig: 0 kann rot, aber auch Sonntag bedeuten, und 1 kann für grün oder auch Montag stehen.

- schlechter Dokumentationswert :

  ```
  if(farbe == 2) ...
  ```

  Hieraus ist nicht zu ersehen, welche Farbe gemeint ist. Oder :

  ```
  if(farbe == 5)
  ```

  Der Wert 5 ist undefiniert!

besser: *Aufzählungs- oder Enumerationstyp*

Syntax:

`enum` *[Typname] {Aufzählung} [Variablenliste];*

Beispiel:

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

Beispiel:

```
enum Wochentag   {sonntag, montag, dienstag, mittwoch,
                  donnerstag, freitag, samstag
                 };
```

Wenn ein Datentyp erst einmal bekannt ist, können weitere Variablen definiert werden:

```
Farbtyp gewaehlteFarbe;                  // Definition
Wochentag derFeiertag, einWerktag,       // Definitionen
                    heute=dienstag;      // Definition + Initialisierung
```

2.5.2 Arrays: Der C++ Standardtyp `vector`

Vektor = eindimensionale Tabelle mit Elementen desselben Datentyps

Vordefinierte Klasse der C++ Standardbibliothek, Benutzung als Baustein (black box). Beispiel:

```
vector<int> V(10); // Numerierung 0..9
```

Einige Dienstleistungsfunktionen:

```
cout << V.size()  << endl;          // 10
cout << V[0] << endl;   // erstes Element (Nr. 0!)
```

Es gibt mit [ ] keine Überprüfung der Bereichsüber- oder -unterschreitung! Zugriffe auf nicht existierende Elemente wie `c = V[100]` erzeugen *keine* Fehlermeldung, weder durch den Compiler, noch zur Laufzeit!

MIT Prüfung:

```
cout << V.at(0)    << endl; // alles bestens, dasselbe wie V[0]
// 1000 ist zuviel!
cout << V.at(1000) << endl; // Programmabbruch mit Fehlermeldung
```

Programm mit typischen Vektor-Operationen:

```
#include<iostream>
#include<vector>      // Standard-Vektor einschließen
#include<cstddef>     // size_t
using namespace std;
int main() {
    vector<float> Kosten(12);  // Tabelle
    // Füllen der Tabelle mit beliebigen Daten, dabei
    // Typumwandlung int → float
    for(size_t i = 0; i < Kosten.size(); ++i)
        Kosten[i] = static_cast<float>(150-i*i)/10.0;

    // Tabelle ausgeben
    for(size_t i = 0; i < Kosten.size(); ++i)
        cout << i << ": " << Kosten[i] << endl;

    // Berechnung und Anzeige von ∑_{i=0}^{anzahl-1} Kosten_i
    float sum=0.0;
    for(size_t i = 0; i < Kosten.size(); ++i)
        sum += Kosten[i];
```

```
cout << "Summe = " << sum << endl;
```

```
// Mittelwert anzeigen
cout << "Mittelwert = " << sum/Kosten.size() << endl;

// Maximum anzeigen
float maxi = Kosten[0];
for(size_t i = 1; i < Kosten.size(); ++i)
   if(maxi < Kosten[i]) maxi = Kosten[i];
cout << "Maximum = " << maxi << endl;

// zweite Tabelle sortierteKosten deklarieren und
// mit der ersten initialisieren
vector<float> sortierteKosten = Kosten;

// zweite Tabelle aufsteigend sortieren (Bubble-Sort-Variante)
for(size_t i = 1; i < Kosten.size(); ++i)
   for(size_t j = 0; j < i; j++)
      if(sortierteKosten[i] < sortierteKosten[j]) {
          // Elemente an i und j vertauschen
          float temp = sortierteKosten[i];
          sortierteKosten[i] = sortierteKosten[j];
```

```
                sortierteKosten[j] = temp;
            }
    // und ausgeben
    for(size_t i = 0; i < Kosten.size(); ++i)
        cout << i << ": " << sortierteKosten[i] << endl;
}
```

An der Stelle

```
vector<float> sortierteKosten = Kosten; // Initialisierung
```

wäre auch folgendes möglich gewesen:

```
vector<float> sortierteKosten;          // Objekt anlegen
sortierteKosten = Kosten;               // Zuweisung
```

In C++ ist eine Initialisierung *keine* Zuweisung. Initialisierung und Zuweisung werden in C++ streng unterschieden.

Beides ist trotz desselben Operators (=) leicht zu unterscheiden:

Eine Initialisierung kann nur bei der gleichzeitigen Definition (= Erzeugung) eines Objekts auftreten, eine Zuweisung setzt immer ein schon vorhandenes Objekt voraus.

Variationen zur Suche in einer Tabelle:

```
// Definitionen für alle fünf Fälle
const int n= ...
vector<int> a(n+1);// letztes Element nur für Fall 4
int key=...        // gesuchtes Element
int i;             // Laufvariable
                   // Ergebnis: i == 0..n-1 : gefunden!
                   // i == n : nicht gefunden!
```

1. `while`-Schleife

```
i=0;
while(i < n && a[i] != key) ++i;
```

2. `do while`-Schleife

```
i=-1;
do ++i; while(i < n && a[i] != key);
```

3. `for`-Schleife

```
for(i = 0;i < n; ++i) if(a[i] == key) break;
```

4. (n+1). Element als „Wächter“ (sentinel)

```
i = -1;
a[n] = key;              // garantiert Abbruch der Schleife
while(a[++i] != key);
```

Vektoren sind dynamisch!

```
vector<double> Vd(10);
// ....
Vd.push_back(3.14159254);     // jetzt 11 Elemente!
```

2.5.3 Zeichenketten: Der C++ Standardtyp `string`

Beispiel für die Nutzung des Bausteins:

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

// Programm mit typischen String-Operationen
int main() {
    // String-Objekt einString anlegen und initialisieren.
    // einString kann ein beliebiger Name sein.
    string einString("hallo");

    // String ausgeben
    cout << einString << endl;

    // String zeichenweise ausgeben, ungeprüfter Zugriff:
    for(size_t i = 0; i < einString.size(); ++i)
       cout << einString[i];
    cout << endl;

    // String zeichenweise mit Indexprüfung ausgeben:
```

```
for(size_t i = 0; i < einString.size(); ++i)
   cout << einString.at(i);
cout << endl;

// Kopie des Strings einString erzeugen
string eineStringKopie(einString);
cout << eineStringKopie << endl;      // hallo

// Kopie durch Zuweisung
string diesIstNeu("neu!");
eineStringKopie = diesIstNeu;
cout << eineStringKopie << endl;      // neu!

// Zuweisung einer Zeichenkette in Anführungszeichen
eineStringKopie = "Buchstaben";
cout << eineStringKopie << endl;      // Buchstaben
```

```
    // Zuweisung nur eines Zeichens vom Typ char
    einString = 'X';
    cout << einString<< endl;              // X

    // Strings mit dem +=-Operator verketten
    einString += eineStringKopie;
    cout << einString<< endl;              // XBuchstaben

    // Strings mit dem +-Operator verketten
    einString = eineStringKopie + " ABC";
    cout << einString<< endl;          // Buchstaben ABC

    einString = "123" + eineStringKopie;
    cout << einString<< endl;          // 123Buchstaben

    // Eine Erklärung gibt's erst später, aber folgendes geht nicht:
    einString = "123" + "ABC";             // Fehler!
}
```

2.5.4 Strukturierte Datentypen

zur Bündelung logisch zusammengehöriger Variablen

`struct` *[Typname] {Aufbau} [Variablenliste];*

Beispiel:

```
enum farbtyp {rot, gelb, gruen};
struct punkt {
    int x,y;
    bool sichtbar;
    farbtyp farbe;
} p;                                  // p ist ein punkt
punkt q;                              // q auch

// Zugriff
p.x = 270;    p.y = 20;               // Koordinaten von p
p.sichtbar = false;
p.farbe = gelb;
```

3. Einfache Ein- und Ausgabe

– Standard- Ein- und Ausgabe

– Ein- und Ausgabe mit Dateien

3.1 Standard- Ein- und -Ausgabe

Ein Programm empfängt einen Strom von Eingabedaten, verarbeitet diese Daten, und gibt einen Strom von Ausgabedaten aus.

Vordefiniert sind

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

Eingabe

Eigenschaften des Eingabeoperators >>:

- Führende Whitespaces (Whitespace = Leerzeichen, Tabulatorzeichen '\t', Zeilenendekennung '\n') werden ignoriert.
- Whitespaces werden als Endekennung genutzt.
- Andere Zeichen werden entsprechend dem verlangten Datentyp interpretiert, z.B.:

```
int zahl;
cin >> zahl;        // automatische Wandlung in int
```

Einzelnes Zeichen lesen (auch Whitespace):

```
char c;
cin.get(c);
```

Eine Zeile lesen:

```
string dieZeile;
getline(cin, dieZeile);
```

Eine Zeichenkette bis Whitespace lesen:

```
string Text;
cin >> Text;
```

Ausgabe

automatische Umformung der internen in die ASCII-Darstellung

Formatierungen sind möglich, zum Beispiel:

```
cout << 7;
cout.width(6);
cout << 11;
```

`cin` und `cout` können auf Betriebssystemebene mit $<$ beziehungsweise $>$ umgeleitet werden:
prog1 $<$ *eingabe* $>$ *ausgabe*

3.2 Ein- und Ausgabe mit Dateien

Dateiobjekte sind vom Typ `ifstream` bzw. `ofstream`. Beispiel:

```
// datcopy.cpp kopiert eine Datei
#include<cstdlib> // für exit( )
#include<fstream>
#include<string>
using namespace std;
int main( ) {
    // Definieren und Öffnen der Eingangsdatei
    ifstream quelle;        // Datentyp für Eingabestrom
    string Quelldateinname;
    cout << "Quellfile? ";
    cin >> Quelldateinname;
    quelle.open(Quelldateinname.c_str(), ios::binary|ios::in);
    if(!quelle)  {    // Fehlerabfrage
        cerr << Quelldateinname << " kann nicht geöffnet werden!\n";
        exit(-1);
    }
```

```
    // Definieren und Öffnen der Ausgabedatei
    ofstream ziel;
    string Zieldateinname;
    cout << "Zielfile? ";
    cin >> Zieldateinname;
    ziel.open(Zieldateinname, ios::binary|ios::out);
    if(!ziel) { // muss vorhanden und nicht read-only
                // oder angelegt worden sein
        cerr << Zieldateinname
             << " kann nicht geöffnet werden!\n";
        exit(-1);
    }

    // kopieren
    char ch;
    while(quelle.get(ch)) ziel.put(ch);
}   // Dateien werden automatisch geschlossen.
```

4. Programmstrukturierung

– Funktionen

– Parameter

– modulare Gestaltung

– Funktions-Templates

– inline-Funktionen

4.1 Funktionen

Zweck: Erledigung abgeschlossener Teilaufgaben

4.1.1 Aufbau und Prototypen

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

// Funktionsprototyp (Deklaration)
unsigned long Fakultaet(int);

int main() {
   int n;
   do  {
      cout << "Fakultät berechnen. Zahl >= 0? :";
      cin >> n;
   } while(n < 0);

   cout << "Das Ergebnis ist "
        << Fakultaet(n) << endl; // Aufruf
```

```
   // alternativ mit Zwischenablage des Ergebnisses:
   unsigned long Erg = Fakultaet(n);
   cout << "Das Ergebnis ist " << Erg << endl;
}
// Funktionsimplementation (Definition)
unsigned long Fakultaet(int zahl) {
   unsigned long fak = 1;
   for(int i = 2; i <= zahl; ++i)
      fak *= i;
   return fak;
}
```

- *Deklaration:*
  = Beschreibung für den Compiler, und Hinweis, dass eine Funktion (oder eine Variable/ ein Objekt dieses Aussehens) irgendwo definiert ist. Er ist damit in der Lage, eine Syntaxprüfung vorzunehmen.
- *Definition:*
  veranlasst den Compiler, Speicherplatz zu reservieren und Programmcode zu erzeugen.

Syntax eines Funktions*prototypen*:

Rückgabetyp Funktionsname `(` *Parameterliste* `);`

Rückgabetyp: beliebig, aber kein Array und keine Funktion (aber Zeiger)

```
unsigned long  Fakultaet  (    int n    );
      ⋮            ⋮       ⋮      ⋮       ⋮
  Rückgabetyp   Funktionsname ( Parameterliste );
```

Syntax der Funktions*definition*:

Rückgabetyp Funktionsname (*Formalparameterliste*) *Block*

```
unsigned long  Fakultaet  (      int x      ) {...}
      ⋮            ⋮       ⋮        ⋮        ⋮    ⋮
Rückgabetyp  Funktionsname  (  Formalparameterliste  )  Block
```

Syntax des Funktions*aufrufs*:

Funktionsname (*Aktualparameterliste*);

Compiler: prüft Syntax

Linker : prüft, ob eine Definition vorhanden ist

4.1.2 Gültigkeitsbereiche und Sichtbarkeit in Funktionen

$\Rightarrow$ im Code-Block wie Gültigkeits- und Sichtbarkeitsregeln für Variable

$\Rightarrow$ Ausnahme: Parameterliste

Die Parameterliste

- wirkt nach innen wie eine lokale Deklaration
- stellt nach außen die *Datenschnittstelle* dar

```
#include<iostream>
using namespace std;
int a = 1;                  // überall bekannt: global
void f1( ) {
     int c = 3;             // nur in f1( ) bekannt: lokal
     cout << "f1: c= " << c << endl;
     cout << "f1: globales a= " << a << endl;
}
int main( ) {
    cout << "main: globales a= " << a << endl;
    // cout « f1: c= «< c; ist nicht compilierfähig
    f1( );
}
```

Das Programm erzeugt folgende Ausgabe:

main: globales a= 1
f1: c= 3
f1: globales a= 1

Innerhalb des Blocks deklarierte Variable sind *automatisch!*
Ausnahme : `static`-Variable. Beispiel:

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

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

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

4.2 Schnittstellen zum Datentransfer

Schnittstelle = formale Vereinbarung zwischen Aufrufer und Funktion

Beschreibung durch Funktionsprototyp:

- den Rückgabetyp der Funktion
- den Funktionsnamen
- Parameter, die der Funktion bekannt gemacht werden
- die Art der Parameterübergabe.

Arten des Datentransports: die Übergabe per

- Wert,
- Referenz und
- `return`

4.2.1 Übergabe per Wert

Der Wert wird in die Funktion *hineinkopiert*.

Beispiel 1:

```
#include<iostream>
using namespace std;
int addiere_5(int);       // Deklaration (Funktionsprototyp)
int main( ) {
    int erg, i = 0;
    cout << i << " = Wert von i\n";
    erg=addiere_5(i);
    cout << erg << " = Ergebnis von addiere_5\n";
    cout << i << " = i unverändert!\n";
}
int addiere_5(int x) {      // Definition
     x += 5;
     return x;
}
```

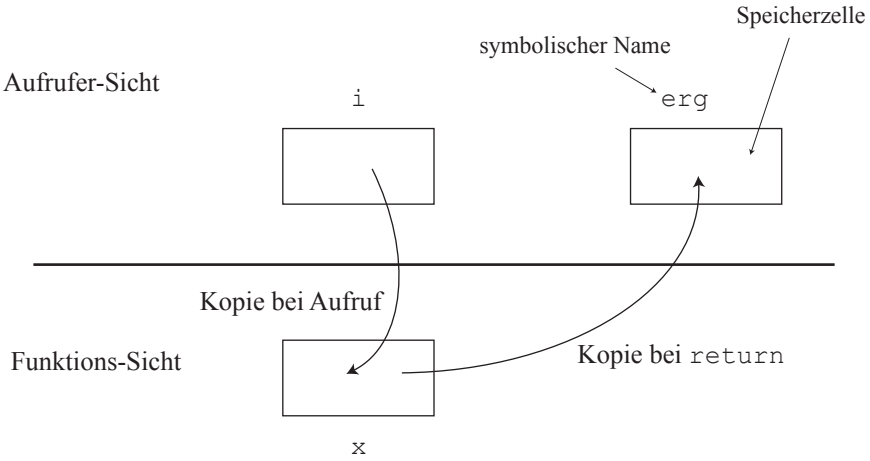


Abb. 4.1: Parameterübergabe per Wert (Bezug: Programmbeispiel)

Beispiel 2:

```
int qsum(long z) {          // Quersumme von z
    if(z == 0) {            // Abbruchbedingung
        int letzteZiffer = z % 10;
        return letzteZiffer+qsum(z/10);
    }
    else return 0;
}
```

4.2.2 Übergabe per Referenz

In der Funktion wird mit *dem Original* gearbeitet.

Eigenschaften:

- Die Angabe des Objektnamens genügt. Der Compiler sorgt für die richtige Zuordnung.
- Laufzeitvorteil bei großen Objekten, weil nicht kopiert wird.
- Der Laufzeitvorteil kann bei nicht zu ändernden Objekten durch `const &` genutzt werden (Ersatz der Übergabe per Wert).

```
void addiere_7(int&); // int& = Referenz auf int

int main( ) {
    int i = 0;
    cout << i << " = alter Wert von i\n";
    addiere_7(i);       // Syntax wie bei Übergabe per Wert
    cout << i << " = neuer Wert von i nach addiere_7\n";
}

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

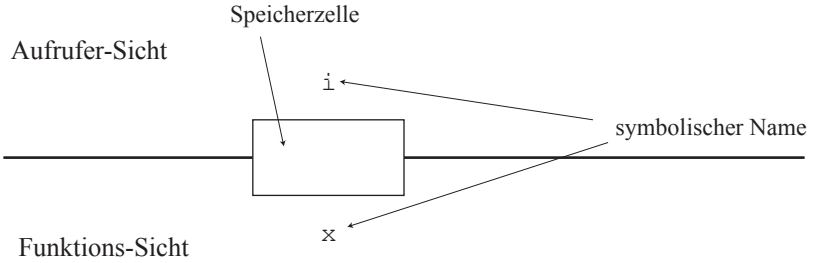


Abb. 4.2: Parameterübergabe per Referenz (Bezug: Programmbeispiel)

4.2.3 Gefahren bei der Rückgabe von Referenzen

Negativ-Beispiel

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

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

int main() {
    int x = 17, y = 4;
    int z = maxwert(x,y);
    //...
}
```

4.2.4 Vorgabewerte und variable Parameterzahl

Vorteil: leichte Erweiterbarkeit von Funktionen, ohne existierenden Code zu verletzen

```
// Aufruf im Programm1
AdressenSortieren(Adressdatei);     // nach Nachnamen
```

Die Sortierung nach Postleitzahlen und Telefonnummern wurde später benötigt und nachträglich eingebaut. Der Aufruf in einer neuen Anwendung könnte wie folgt lauten:

```
// anderes, NEUES Programm2
enum Sortierkriterium {Nachname, PLZ, Telefon};
AdressenSortieren(Adressdatei, PLZ);
```

Das alte Programm1 ist wie bisher mit der gleichen Bibliothek übersetzbar!

Die Parameter mit Vorgabewerten erscheinen in der Deklaration *nach* den anderen Parametern.

Weiteres Beispiel:

```
#include<iostream>
using namespace std;
// Funktionsprototyp 2. Parameter mit Vorgabewert:
void PreisAnzeige(double Preis,
                  const string& Waehrung = "Euro");
// Hauptprogramm
int main() {
     // zwei Aufrufe mit unterschiedlicher Parameterzahl :
     PreisAnzeige(12.35);    // Default-Parameter
     PreisAnzeige(99.99,"US-Dollar");
}
// Funktionsimplementation
void PreisAnzeige(double Preis, const string& Waehrung) {
      cout << Preis << ' ' << Waehrung << endl;
}
```

Ausgabe des Programms:
12.35 Euro
99.99 US-Dollar

4.2.5 Überladen von Funktionen

derselbe Funktionsname für verschiedene Funktionen!

Zweck: bessere Lesbarkeit

```
// Überladen zweier Funktionen
#include<iostream>
using namespace std;

double max(double x, double y) {
    if(x > y) return x;
    else      return y;
}

// zweite Funktion gleichen Namens,
// aber unterschiedlicher Signatur
int max(int x, int y) {
    if(x > y) return x;
    else      return y;
}
```

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

4.2.6 Funktion main

Varianten:

```
int main() {
    ...
    return 0;     // Exit-Code
}

int main() {
   ...            // automatisch: return 0
}
```

`main()` kann null bis drei Parameter haben:

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

`argc` = Anzahl der Kommandozeilenparameter
`argv[ ]` = C-Array mit Kommandozeilenparametern
`env[]` = C-Array der Umgebungsvariablen.

Dabei ist
`argv[argc]` = `""` (leerer String), und
`argv[0]` enthält den Programmaufruf

4.3 Grundsätze der modularen Gestaltung

Sinnvoll: Aufteilung eines großen Programms in einzelne, getrennt übersetzbare Dateien, die zusammengehörige Programmteile enthalten.

Aufbau:

- Header enthalten Konstanten und Schnittstellenbeschreibungen wie
  - Klassendeklarationen,
  - Deklarationen globaler Daten und
  - Funktionsprototypen

  Typisch: Dateien mit *.h*-Endung, außer Standardheader
- Implementations-Files enthalten die Implementation der Klassen und den Programmcode der Funktionen (**.cpp*)
- Main-File enthält das Hauptprogramm `main( )`

Bekanntmachen der Schnittstellen durch Lesen der Header-Files:

```
#include<iostream>
#include"myheader.h"
```

Zwei mögliche Verfahren zur Erzeugung eines lauffähigen Programms:

- Die Steuerung der Übersetzung nur durch `#include`-Anweisungen
- Einbinden von bereits vorübersetzten Programmteilen — besonders sinnvoll bei sehr großen Programmen, von denen einige Teile schon stabil laufen

4.3.1 Steuerung der Übersetzung mit #include

```
// nicht empfehlenswert!
#include "a.cpp"
#include "b.cpp"
int main( ) {
      func_a1( );   // Funktionsaufrufe
      func_a2( );
      func_b( );
}
```

Nur bei *sehr kleinen* Programmen ist dieses Verfahren ausreichend.

4.3.2 Einbinden vorübersetzter Programmteile

I.a. ist es sinnvoll, Schnittstellen (Funktionsprototypen und Klassen) und Implementationen (Programmcode) zu trennen.

```
// a.h
void func_a1();
void func_a2();

// a.cpp
#include "a.h"
void func_a1() { Programmcode zu func_a1 }
void func_a2() { Programmcode zu func_a2 }

// b.h
void func_b();

// b.cpp
#include "b.h"
void func_b() { Programmcode zu func_b }
```

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

Annahme: *a.cpp* und *b.cpp* sind schon übersetzt, die Files *a.o* und *b.o* existieren. In *mainprog.cpp* sei eine Änderung notwendig gewesen. Ablauf:

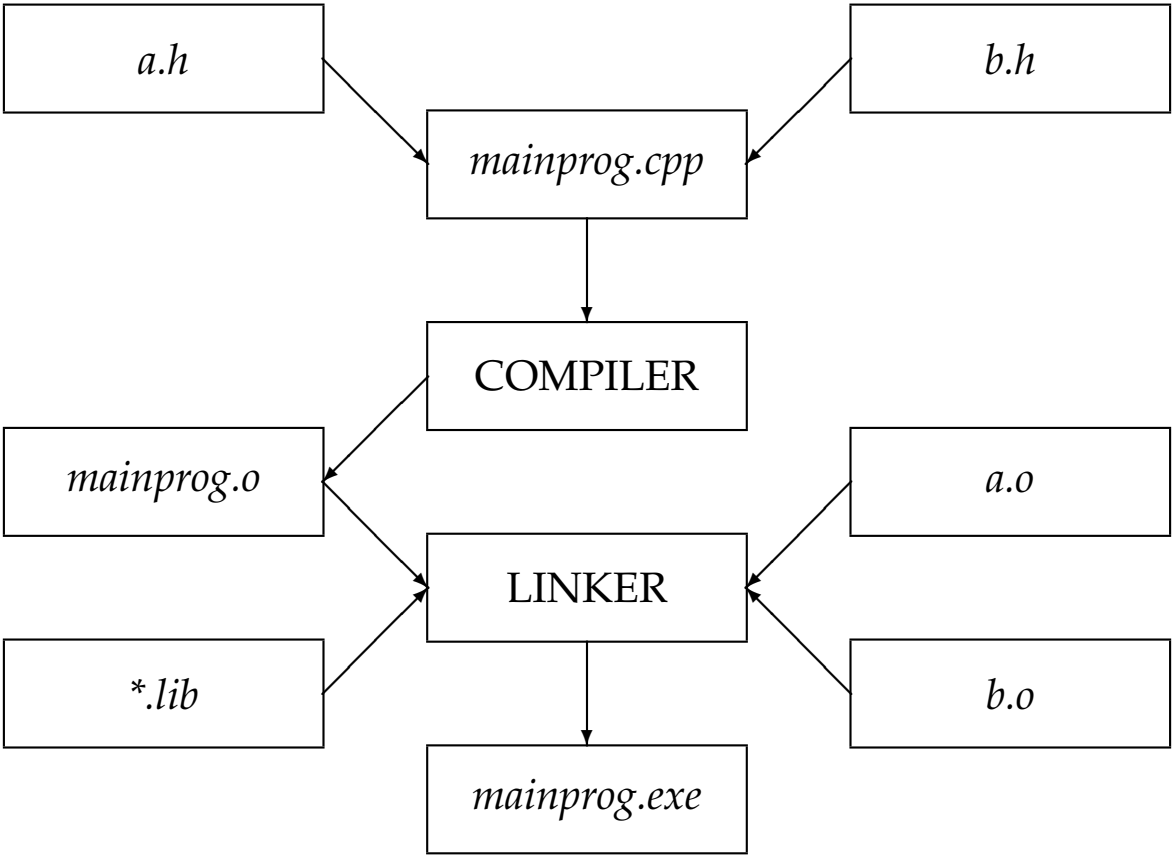


Abb. 4.3: Compilations-Ablauf

Üblich: *make*-Files oder „Projekte“
Die Konzepte

- Trennung von Schnittstellen und Implementation, und
- Gruppierung zusammengehöriger Funktionen und Klassen zu Bibliotheksmodulen

sind Standard in allen größeren Programmierprojekten.

4.3.3 Dateiübergreifende Gültigkeit und Sichtbarkeit

Variablen, die außerhalb `main()` und jeglicher anderer Funktion definiert sind, heißen *global*. Sie sind in *allen* Teilen eines Programms gültig, auch in anderen Dateien. Eine globale Variable muss nur in einem anderen File als `extern` deklariert werden, um dort benutzbar zu sein.

```
// Datei1.cpp
int global;
int main( ) {
    global = 17;
}
// Datei2.cpp
extern int global;
int func1( ) {  global = 123;}
```

Datei2.cpp ist für sich allein übersetzbar. Das Schlüsselwort `extern` sagt dem Compiler, dass eine Variable irgendwo anders definiert ist. Erst beim Linken wird die Referenz aufgelöst.

Beschränkung des Gültigkeitsbereichs auf *eine* Datei mit `static`:

```
// Datei1.cpp (geändert!)
static int global;
int main( ) {
    global = 17;
}
```

Übersetzung von *Datei1.cpp* und *Datei2.cpp*: ok, aber:
Link-Fehler bei *Datei2.o*!

Auf Dateiebene (außerhalb `main()`) definierte *Variable* sind *global* und sind in anderen Dateien benutzbar, wenn sie dort als `extern` deklariert sind.

Bei *Konstanten* (`const`) ist es jedoch anders! Konstanten sind nur im Definitionsfile sichtbar! Sollen Konstante anderen Dateien zugänglich gemacht werden, müssen sie als `extern` deklariert und initialisiert werden:

```
// Datei1.cpp
extern const float pi = 3.14159;

// Datei2.cpp
extern const float pi; // OHNE Wertzuweisung
```

Ohne `extern` in *Datei1.cpp* wäre der Geltungsbereich von `pi` auf *Datei1.cpp* beschränkt.

4.3.4 Übersetzungseinheit, Deklaration und Definition

- Alles, was der Compiler in einem Durchgang liest, ist eine *Übersetzungseinheit*.
- Eine *Deklaration* führt einen Namen in ein Programm ein und gibt dem Namen eine Bedeutung.
- 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.

Folgende Deklarationen sind gleichzeitig Definitionen:

```
int a;                                  // Speicherplatz für a wird angelegt
extern const float pi = 3.14159;        // Speicherplatz für pi wird angelegt
int f(int x) { return x*x;}             // enthält Programmcode
struct meinStrukt {                     // definiert meinStrukt, d.h.
    int c,d;                            // beschreibt die innere Struktur
  };
meinStrukt X;                           // Speicherplatz für X wird angelegt
enum meinEnum { li, re };               // definiert meinEnum
meinEnum Y;                             // Speicherplatz für Y wird angelegt
```

Die folgenden Zeilen sind Deklarationen, aber *keine* Definitionen:

```
extern int a;
extern const float pi;
int f(int);
struct meinStrukt;
enum meinEnum;
```

one definition rule

> Jede Variable, Funktion, Struktur, Konstante und so weiter in einem Programm hat *genau eine* Definition.

Konsequenz:

Inhalt von Header-Dateien (**.h*, **.hpp*)

- Funktionsprototypen (Schnittstellen)

```
void meineFunktion(int einParameter);
```

- reine Deklaration (nicht Definition) globaler Variablen

```
extern int global;
```

- reine Deklaration globaler Konstanten (nicht Definition, das heißt ohne Initialisierung)

```
extern const int globaleKonstante;
```

- Definition von Datentypen wie `enum` oder `struct` (weil die **.cpp*-Dateien die Größe von Objekten dieser Datentypen kennen müssen)

```
struct Punkt {
    int x;
    int y;
};
enum Wochenende {Samstag, Sonntag};
```

Inhalt von Implementations-Dateien (*.cpp, *.cc, *.C)

- Funktionsdefinitionen (Implementation)

```
void meineFunktion(int Parameter) {
    // ... Programmcode
}
```

- Definition globaler Variablen (nur *einmal* im Programm)

```
int global;
```

- Definition und Initialisierung globaler Konstanten (nur *einmal* im Programm)

```
extern const int globaleKonstante = 1;
```

- Definition von Objekten bestimmter Datentypen

```
Punkt einPunkt;

Wochenende einWochenende;
```

Achtung:
Variablen, die ohne das Schlüsselwort `extern` in der Header-Datei auftreten, sind global. Globale Variablen sollten also immer extern deklariert werden, und die Definition sollte nur in einer Übersetzungseinheit vorkommen.

Konstanten, die ohne das Schlüsselwort `extern` in der Header-Datei auftreten, sind nicht global und beziehen sich nur auf die Übersetzungseinheit.

4.3.5 Compilerdirektiven

sind Anweisungen an den Compiler, die den Übersetzungsprozess steuern. Verarbeitung durch Präprozessor.

`#include` Einlesen weiterer Dateien
`#define`, `#ifdef`, `#ifndef`
zur bedingten Compilation. Beispiel:

```
// c.h
#ifndef c_h
#define c_h
void func_c1();
void func_c2();
#endif c_h
```

Bedeutung:
Falls der (beliebige) Name `c_h` nicht definiert ist,
dann definiere `c_h` und akzeptiere alles bis `#endif`.

Die Wirkung des *ersten* Lesens von *c.h* als indirekte Folge von `#include "a.cpp"` in *mainprog.cpp* ist:

- `#ifndef c_h` liefert TRUE, weil `c_h` noch nicht definiert ist
- `#define c_h` definiert `c_h`
- alles bis `#endif` wird eingeschlossen

Die Wirkung des *zweiten* Durchlaufs von *c.h* als indirekte Folge von `#include "b.cpp"` in *mainprog.cpp* ist:

- `#ifndef c_h` liefert FALSE (d.h. 0), weil `c_h` bereits definiert ist
- alles bis `#endif` wird ignoriert.

Makros mit #define

zum Ersetzen von Makros durch Zeichenketten, wobei Parameter erlaubt sind. Die Makrodefinitionen

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

erlauben den Text

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

in einem Programm und würden interpretiert werden als:

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

Die Textersetzung mit `#define` sollte im allgemeinen *nicht* verwendet werden, wenn es Alternativen gibt.

Vergleiche z. B. das Ergebnis des Makros mit dem Ergebnis eines Funktionsaufrufs:

```
int quadrat(int arg) { return arg*arg; }

int x = 4;
int erg1 = QUAD(++x);   // Definition s. oben
int y = 4;
int erg2 = quadrat(++y);
```

Eine weitere übliche Anwendung von `#define` als Textersetzungsmakro mit Parametern ist die gezielte Ein- und Ausblendung von Testsequenzen in einem Programm. Beispiel:

```
#define TEST_EIN
#ifdef TEST_EIN
   #define TESTANWEISUNG(irgendwas) irgendwas
#else
   #define TESTANWEISUNG(irgendwas) /* nichts
                                       */
#endif

// ... irgendwelcher Programmcode
// nur im Test soll bei Fehlern
// eine Meldung ausgegeben werden:
TESTANWEISUNG(if(x < 0)
     cout << "sqrt(negative Zahl)!" << endl;)
y = sqrt(x);
// ... mehr Programmcode
```

Vorteile:

- Nach Testabschluss wird das lauffertige Programm schneller und benötigt weniger Speicher durch die fehlenden Testanweisungen.
- Die Testanweisungen können im Programm zum späteren Gebrauch stehen bleiben. Sie müssen nicht einzeln auskommentiert oder gelöscht werden.

Nur mit den Sprachelementen von C++ :

```
const int TEST_EIN=1;
// ... irgendwelcher Programmcode
// nur im Test soll bei Fehlern
// eine Meldung ausgegeben werden:
if(TEST_EIN)
   if(x < 0) cout << "sqrt(negative Zahl)!"
                  << endl;
y = sqrt(x);
// ... mehr Programmcode
```

Nützlich: Argumentwandlung in Zeichenkette mit #

```
#define PRINT(X) cout « (#X) « "= "« (X) « '\n'
```

Damit kann kurz

```
PRINT(int(xptr)-int(xptr2));
```

geschrieben werden anstatt

```
cout << "int(xptr)-int(xptr2) = "
     << int(xptr)-int(xptr2) << endl;
```

mit dem Ergebnis

int(xptr)-int(xptr2) = 4

auf dem Bildschirm.

Verifizieren logischer Annahmen mit `assert`

assertion = Zusicherung

```
#include<cassert>    // enthält Makrodefinition
const int Grenze = 100;
int Index;

// .... Berechnung von Index

// Test auf Einhaltung der Grenzen:
assert(Index >= 0 && Index < Grenze);
```

Seiteneffekte vermeiden!

```
assert(datei_oeffnen( filename)
                  == erfolgreich);    // Fehler
assert(zeiger = new int[100]);        // Fehler
```

(keine Ausführung bei NDEBUG)

4.4 Funktions-Templates

templates == parametrisierte Datentypen

`template <class` *TypBezeichner*`>` *Funktionsdefinition*

Beispiel: Quicksort:

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

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

```
template<class T>
int kleiner(const T& a, const T& b) {  // Vergleich
    return (a < b);
}
template<class T>
void drucke(const vector<T>& V) {
    for(size_t i = 0; i < V.size(); ++i)
         cout << V[i] <<" ";
    cout << endl;
}
```

```
template<class T>
void quicksort(vector<T>& a, int links, int rechts) {
    int li = links; int re = rechts;
    T el = a[(links+rechts)/2];
    do {
         while(kleiner(a[li],el)) ++li;
         while(kleiner(el,a[re])) --re;
         if(li <  re) tausche(a[li],a[re]);
         if(li <= re) {++li; --re;}
    } while(li <= re);
    if(links < re)  quicksort(a, links, re);
    if(li < rechts) quicksort(a, li, rechts);
}


int main() {
    vector<int> iV(10);
    iV[0]=100; iV[1]=22;  iV[2]=-3; iV[3]=44; iV[4]=6;
    iV[5]= -9; iV[6]=-2;  iV[7]= 1;  iV[8]=8;  iV[9]=9;
```

```
    /* Erst in der folgenden Anweisung werden vom Compiler
       durch den Datentyp des Parameters iV aus den obigen
       Templates die Funktionen quicksort(vector<int>&,
       int, int) und drucke(const vector<int>&) er-
       zeugt, ebenso wie die implizit aufgerufenen Funktionen
       tausche(int&, int&) und kleiner(int&, int&):
     */
    quicksort(iV, 0, iV.size()-1);
    drucke(iV);

    vector<double> dV(7);
    dV[0]=1.09; dV[1]=2.2; dV[2]=79.6; dV[3]=-1.9; dV[4]=2.7;
    dV[5]=100.9; dV[6]=18.8; dV[7]=99.9;
    // Generierung der überladenen Funktionen für double:
    quicksort(dV, 0, dV.size()-1);
    drucke(dV);
}  // Ende von main()
```

Damit liegt ein universelles Sortierprogramm vor, das für verschiedene Datentypen geeignet ist. Man kann natürlich auch die Sortierfunktion der Standardbibliothek nehmen...

Auslagern des Vergleichs erspart die Änderung von `quicksort()`, wenn der Operator $<$ anders definiert werden muss, z.B. Vergleich nach Absolutbetrag:

```
// (#include<cstdlib> für abs() nicht vergessen)
int kleiner(const int& a, const int& b) {
// Vergleich jetzt nach dem Absolutbetrag!
     return abs(a) < abs(b);
}
```

Spezialfälle von überladenen Funktionen werden *nach* den Templates eingefügt.

> Ein Template ist keine Funktionsdefinition im bisherigen Sinn! Die konkrete Definition wird erst bei Bedarf erzeugt.

4.5 inline-Funktionen

`inline`
ist eine Empfehlung an den Compiler, einen Funktionsaufruf direkt durch den Funktionskörper zu ersetzen. Vorteil: Zeitersparnis

Empfehlenswert nur für Funktionen mit *kurzer* Ausführungszeit.

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

`z = quadrat(100);` wird dann ersetzt durch `z = 100*100;`

`inline`-Funktionen sind in Header-Dateien (*.h) zu definieren, nicht in Implementationsdateien (*.cpp), um Linker-Fehler zu vermeiden.

5. Objektorientierung 1

– Abstrakte Datentypen

– Klassen und Objekte

– Konstruktoren

– Destruktoren

5.1 Abstrakte Datentypen

Abstrakter Datentyp = Datentypen + Funktionen

Vergleich (Wiederholung aus der Einführung):
Zugriff auf zwei Koordinaten `X` und `Y` eines Punktes

- Unstrukturierter Zugriff

```
X = 100;
Y = 0;
```

- Strukturierter Zugriff

```
struct Punkt {
   int X, Y;
};

void aendern(Punkt& p, int x, int y) {
     // ... Plausibilitätsprüfung
     p.X = x;
     p.Y = y;
```

```
        // ... Protokollierung
}

Punkt einPunkt;
// Aufruf
aendern(einPunkt, 10, 800);

einPunkt.X = -3000;    // ???
```

- Abstrakter Datentyp

  Kapselung, durch Programmiersprache unterstützt

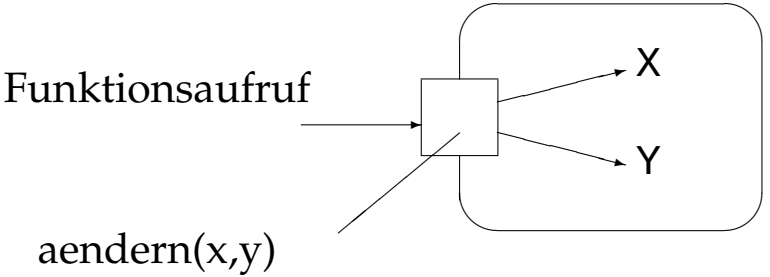


Abb. 5.1: Abstrakter Datentyp

- Die Funktion als öffentliche Schnittstelle gehört zur Datenkapsel.

- Eine direkte Änderung der Daten unter Umgehung der Funktion ist unmöglich.

- Der ADT ist als Software-Baustein verwendbar.

Eigenschaften:

- Der Sinn liegt darin, den richtigen Gebrauch der Daten sicherzustellen.
- Die *tatsächliche* Implementierung der Datenstrukturen ist nach außen nicht sichtbar.
- Logisch zusammengehörige Dinge sind an einem Ort konzentriert.
- Spezifikation ausschließlich durch die öffentliche Schnittstelle des ADT, die die mit ihm möglichen Operationen definert.

Realisierung durch *Klassen* in der OOP.

5.2 Klassen und Objekte

- In C++ ist eine Klasse ein *Datentyp*, genauer: ein ADT.
- Für den Compiler ist eine Klasse eine Beschreibung später zu definierender Objekte.
- Ein Objekt ist die konkrete Ausprägung (Instanz) einer Klasse, es belegt Platz im Speicher.
- Ein Objekt hat eine *Identität* und einen *Zustand*.
- Der Zustand eines Objekts wird durch in der Klasse deklarierte Funktionen (Operationen, Methoden) geändert.

Eine in C++ formulierte Klasse hat eine typische Gestalt:

```
class Klassenname {
  public:
    Typ Elementfunktion1();
    Typ Elementfunktion2();
    // und weitere ...
  private:
    Typ Attribut1;
    Typ Attribut2;
    // und weitere ...
};
```

einfaches Beispiel: Klasse für einen *Ort*. Operationen:

`X()`	= X-Koordinate zurückgeben
`Y()`	= Y-Koordinate zurückgeben
`aendern()`	X- und Y-Koordinaten ändern

Die Klasse wird `Ort1` genannt, weil sie noch Änderungen unterliegt. Sie ist wie folgt definiert:

```
#ifndef ort1_h
#define ort1_h ort1_h
class Ort1 {                              // Version 1
    public:
      int X() const;   // keine Zustandsänderung
      int Y() const;   // keine Zustandsänderung
      void aendern(int x, int y); // x,y = neue Werte
    private:
      int xKoordinate,
          yKoordinate;
};
#endif      // ort1_h
```

Anwendung:

```
// ort1main.cpp
#include"ort1.h"
#include<iostream>
using namespace std;

int main() {                    // Anwendung der Ort1-Klasse
   Ort1 einOrt1;                // Objekt erzeugen
   einOrt1.aendern(100, 200);
   cout << "Der Ort hat die Koordinaten x = "
        << einOrt1.X() << " und y = "
        << einOrt1.Y() << endl;
}
```

Konstruktor = besondere Klassenfunktion zur Objekterzeugung (system-generiert oder selbstgeschrieben)
Ein korrekter Zustand wird (noch) durch `aendern()` hergestellt.

Wie das Objekt diese Dienstleistung erbringt, ist in der Methode `aendern()` versteckt.

Client-Server-Prinzip im Kleinen:

- Client fordert die Dienstleistung (hier: `main()`)
- Server erbringt die Dienstleistung (hier: `einOrt1`)

`aendern()` soll Änderungen der Koordinaten protokollieren:

```
// ort1.cpp
#include"ort1.h"
#include<iostream>
using namespace std;
int Ort1::X() const { return xKoordinate;}
int Ort1::Y() const { return yKoordinate;}
void Ort1::aendern(int x, int y) {
   xKoordinate = x;
   yKoordinate = y;
   cout << "Ort1-Objekt geändert! x = "
        <<  xKoordinate << " y = "
        <<  yKoordinate << endl;
}
```

Erzeugen des lauffähigen Programms durch

- Übersetzen der Datei `ort1.cpp`
- Übersetzen der Datei `ort1main.cpp`
- Linken der Dateien `ort1.o` und `ort1main.o`

5.2.1 inline-Elementfunktionen

1. Deklaration *und* Definition innerhalb der Klasse. Beispiel:

```
// Datei ort1.h, Auszug
class Ort1 {                                  // Version 2
   public:
     int X() const { return xKoordinate;}
     int Y() const { return yKoordinate;}
     void aendern(int x, int y) {
       xKoordinate = x;
       yKoordinate = y;
       std::cout << "Ort1-Objekt geändert! x = "
            <<  xKoordinate << " y = " <<  yKoordinate
            << std::endl;
     }
   private:
     int xKoordinate,
         yKoordinate;
};
```

2. Deklaration und Definition innerhalb der Header-Datei:

```
// datei ort1.h
class Ort1 {                    // Version 3
    public:                     // nur Prototypen
      int X() const;
      int Y() const;
      void aendern(int x, int y); // x,y = neue Werte
    private:
      int xKoordinate,
          yKoordinate;
};

/*  ===== inline - Implementierung ====
 */
inline int Ort1::X() const { return xKoordinate;}
inline int Ort1::Y() const { return yKoordinate;}
inline void Ort1::aendern(int x, int y) {
   // ... wie oben
}
```

3. `inline`-Deklaration außerhalb der Klasse. Nicht empfehlenswert! (Linker kann Probleme bekommen) Beispiel:

```
// in ort1.h
class Ort1  {                    // Version 4
      // ... wie oben
      int X() const;
};

// in ort1.cpp!
inline int Ort1::X() const {     // falsch
    return xKoordinate;
}
```

5.3 Initialisierung und Konstruktoren

Objekte können mit Konstruktoren während der Definition initialisiert, also mit sinnvollen Anfangswerten versehen werden - ein wesentlicher Vorteil der OOP! Konstruktoren haben keinen Return-Typ, auch nicht `void`.

5.3.1 Standardkonstruktor

Der Standardkonstruktor hat keine Argumente (Erweiterung der obigen Klasse):

```
Ort1::Ort1() {                  // neuer Standardkonstruktor
    xKoordinate = 0;            // Koordinaten des Nullpunkts
    yKoordinate = 0;
}
```

Anwendung:

```
Ort1 OrtObjekt;
cout << OrtObjekt.X()       // 0
     << OrtObjekt.Y();      // 0
```

Hinweis: Keine Klammern verwenden!

```
Ort1 nochEinOrtobjekt();        // Fehler! Warum?
```

5.3.2 Allgemeine Konstruktoren

Allgemeine Konstruktoren können
- *Argumente* haben und
- *überladen* werden.

Wenn mindestens ein allgemeiner Konstruktor definiert worden ist, wird vom System kein Standardkonstruktor erzeugt, das heißt, es gibt keinen, wenn man ihn nicht selbst geschrieben hat.

```
// Deklaration nicht vergessen!
Ort1::Ort1(int x, int y) {      // Allgemeiner Konstruktor
    xKoordinate = x;
    yKoordinate = y;
}
```

Aufruf des Konstruktors heißt Definition des Objekts.

```
Ort1 nochEinOrt(70, 90); // Objektdefinition = Konstruktoraufruf
```

Vorgegebene Parameterwerte in Konstruktoren

.... wie bei Funktionen:

```
// in ort1.h
class Ort1 {                          // Version 5
      // ... wie oben
      Ort1(int x, int y=100);
};
```

Anwendung:

```
Ort1 nochEinOrt(70);       // xKoordinate=70, yKoordinate=100!
// Vorgabewert wird überschrieben:
Ort1 nochEinOrt(70, 90);
                  // ... oder allgemeiner Konstruktor?
```

Eindeutigkeit gewährleisten!

Initialisierung mit Listen

Beispiel:

```
// in ort1.h
class Ort1 {                        // Version 6
      // ... wie oben
      Ort1(int x, int y)  // allgemeiner Konstruktor, inline
      : xKoordinate(x), yKoordinate(y) {
            // leerer Block
      }
};
```

Die Reihenfolge der Initialisierung ist:

- Zuerst wird die Liste abgearbeitet. Die Reihenfolge der Initialisierung richtet sich nach der Reihenfolge innerhalb der Klassendeklarationen, nicht nach der Reihenfolge in der Liste. `xKoordinate` wird zuerst initialisiert.
- Danach wird der Codeblock { } ausgeführt.

Weitere Anwendung: Initialisierung von Konstanten

Aktualisierung der Klasse `Ort`:

```
#ifndef ort_h
#define ort_h ort_h
#include<cmath>                    // wegen sqrt()
#include<iostream>
using namespace std;

class Ort {
    public:
      Ort(int einX = 0, int einY = 0)
      : xKoordinate(einX), yKoordinate(einY) {
      }

      int X() const { return xKoordinate;}
      int Y() const { return yKoordinate;}
      void aendern(int x, int y) {
          xKoordinate = x;
          yKoordinate = y;
      }
```

```
    private:
      int xKoordinate,
          yKoordinate;
};
// globale Funktion zur Berechnung der
// Entfernung zwischen zwei Orten
inline double Entfernung(const Ort &Ort1, const Ort &Ort2) {
   double dx = static_cast<double>(Ort1.X() - Ort2.X());
   double dy = static_cast<double>(Ort1.Y() - Ort2.Y());
   return sqrt(dx*dx + dy*dy);
}
inline void anzeigen(const Ort &O) {
    cout << '(' << O.X() << ", " << O.Y() << ')';
}
#endif     // ort_h
```

In der Methode `aendern()` wird jetzt wieder auf die Kontrollausgabe verzichtet. Warum?

- Generell sollten zwei verschiedene Dinge nicht von ein und derselben Methode erledigt werden.
- Eine Ausgabe (oder Eingabe) in einer Methode, die eigentlich eine andere Aufgabe hat, verhindert den universellen Einsatz. Zum Beispiel ließe sich die Methode nicht ohne weiteres in einem System mit graphischer Benutzungsoberfläche verwenden.

5.3.3 Kopierkonstruktor

engl.: *copy constructor, copy initializer*

Deklaration: `X(const X&)` für eine Klasse `X`

(systemgeneriert oder selbstgeschrieben)

```
class Ort {
  public:
    Ort(const Ort& einOrt)          // Kopierkonstruktor
      // Kopie der einzelnen Elemente:
    : xKoordinate(einOrt.xKoordinate),
      yKoordinate(einOrt.yKoordinate) {
        // Anzeige des Aufrufs nur zur Demonstration
        cout << "Kopierkonstruktor aufgerufen\n";
    }
    // ... Rest wie oben
};
```

Arbeitsweise:
rekursive Initialisierung aller Attribute bis zur bitweisen Kopie

Anwendung:

```
Ort einOrt(19,39);
Ort derZweiteOrt = einOrt; // Aufruf
cout << derZweiteOrt.X() << ' '
     << derZweiteOrt.Y();    // 19 39
```

Regel:
Ein Kopierkonstruktor wird nur dann benutzt, wenn ein *neues* Objekt erzeugt wird, aber *nicht* bei Zuweisungen, also Änderung von Objekten.
Bei *Zuweisungen* wird der vom System bereitgestellte *Zuweisungsoperator* benutzt, sofern kein eigener definiert wurde.

Der Kopierkonstruktor wird nicht bei Zuweisungen oder Initialisierungen mit allgemeinen Konstruktoren aufgerufen, bei denen ein temporär erzeugtes Objekt in das neu erzeugte Objekt kopiert wird.

```
Ort O1, O2;                 // keine Initialisierung
O1 = Ort(8,7,1997);         // allgemeiner Konstruktor
O2 = O1;                    // Zuweisung
```

Übergabe von Objekten an eine Funktion per Wert und Rückgabe eines Ergebnisobjekts sind Initialisierung! Beispiel:

```
#include"ort.h"
using namespace std;
// Funktion zum Verschieben des Orts um dx und dy
Ort Ortsverschiebung(Ort derOrt, int dx, int dy) {
    derOrt.aendern(derOrt.X() + dx, derOrt.Y() + dy);
    return derOrt;  // Rückgabe des veränderten Orts
}
int main() {
   Ort einOrt(10, 300);
   Ort verschobenerOrt = Ortsverschiebung(einOrt, 10, -90);
   cout << " alter Ort: ";
   anzeigen(einOrt);
   cout << "\n neuer Ort: ";
   anzeigen(verschobenerOrt);
}
```

Der oben definierte Kopierkonstruktor wird zweimal aufgerufen! Optimierung durch den Compiler möglich

5.3.4 Typumwandlungskonstruktor

Typumwandlungskonstruktor = Spezialfall des allgemeinen Konstruktors. Beispiel: Wandlung von Zeichenketten in Ortsangaben

```
#include<cctype>
#include<string>
class Ort {
 public:
   // Typumwandlungskonstruktor. Format: 2 Folgen von Ziffern
   Ort(const string& str)  {
      unsigned int pos = 0;          // Zifferposition
      for(int j = 0; j < 2; ++j) { // jede Koordinate
         while(pos < str.size()) {  // erste Ziffer suchen
           if(isdigit(str.at(pos))) // Ziffer?
             break;
           else ++pos;
         }
         assert(pos < str.size()); // Ziffer gefunden?
```

```
        // Zahl bilden
        int Koordinate = 0;
        while(pos < str.size() && isdigit(str.at(pos))) {
          Koordinate = 10*Koordinate+str.at(pos)-'0';
          ++pos;
        }
        switch(j) {
          case 0: xKoordinate = Koordinate; break;
          case 1: yKoordinate = Koordinate;
        }
      }
    }
  // ... Rest wie vorher
```

Anwendung:

```
Ort nochEinOrt(string("21 99")); // mögliches Format
anzeigen(nochEinOrt);
cout << endl;
Ort einWeitererOrt("(55, 8)"); // mögliches Format
anzeigen(einWeitererOrt);
```

Umgehung der Typprüfung!

```
string wo("20,400");
Ort hier = Ortsverschiebung(wo, 10, -90);

// besser: explizite Angabe der Typumwandlung (s.u.)
Ort dort = Ortsverschiebung(Ort(wo), 10, -90);
```

Der Compiler erzeugt ein für uns unsichtbares Hilfsobjekt `__temp`:

```
// Erzeugen eines temporären Ort-Objekts:
Ort __temp(wo);   // Typumwandlungskonstruktor
Ort hier = Ortsverschiebung(__temp, 10, -90);
// Hier kann das temporäre Objekt zerstört werden.
```

Verhindern impliziter Typwandlungen mit `explicit`:

```
class Ort {
    public:
      explicit Ort(const string& str)
     // ... Rest wie vorher
};

// .....
Ort O1;
string wo("10, 200");
O1 = wo;            // jetzt ein Fehler!
O1 = Ort(wo);       // erlaubte explizite Typumwandlung
```

5.4 Beispiel: Klasse für rationale Zahlen

5.4.1 Aufgabenstellung

Toolbox zum Rechnen mit rationalen Zahlen

Benutzerschnittstelle

- Benutzer/innen der Toolbox binden *ratio.h* per `#include` ein
- linken mit *ratio.o*
- Die Definition einer rationalen Zahl `r` soll wie eine übliche Variablendeklaration geschrieben werden, zum Beispiel:

```
rational r;
```

Folgende Funktionen sollen von der Toolbox bereitgestellt werden:

`r.eingabe();`	Dialogeingabe der rationalen Zahl `r`
`r.ausgabe();`	Dialogausgabe der rationalen Zahl im Format Zähler/Nenner
`r.definiere( z, n);`	Setzen der Werte für Zähler und Nenner
`r.kehrwert();`	`r` enthält danach den Kehrwert

Definition der Rechenfunktionen:

`r = add(a, b);`	`r = a + b`
`r = sub(a, b);`	`r = a - b`
`r = mult(a, b);`	`r = a * b`
`r = div(a, b);`	`r = a / b`

Kurzformoperationen:

`r.add(a);`	`r += a`
`r.sub(a);`	`r -= a`
`r.mult(a);`	`r *= a`
`r.div(a);`	`r /= a`

Die Operanden `a` und `b` sollen vom Typ `rational`, `int`, oder `long`

`int` sein.

Beschränkungen und Hinweise

- Kürzen nach jeder Operation
- keine Bereichsüberprüfung
- Nenner == 0 → Fehlermeldung

5.4.2 Entwurf

Rechenvorschriften:

$x.z =$ Zähler der rationalen Zahl x
$x.n =$ Nenner der rationalen Zahl x

Kehrwert $1/x$ **bilden :** Vertauschen von Zähler und Nenner

Addition $r = a + b$ **:**

$$r = \frac{a.z * b.n + b.z * a.n}{a.n * b.n}$$

Subtraktion $r = a - b$ **:**

$$r = \frac{a.z * b.n - b.z * a.n}{a.n * b.n}$$

Multiplikation $r = a * b$ **:**

$$r = \frac{a.z * b.z}{a.n * b.n}$$

Division $r = a/b$ **:**

$$r = \frac{a.z * b.n}{a.n * b.z}$$

Möglichkeiten für Operationen mit gemischten Datentypen:

1. Den Methoden können weitere überladen werden, so dass für die zwei Operanden folgende Kombinationen erlaubt sind:

   `(rational, rational)`,

   `(rational, long)`, und

   `(long, rational)`.

   `int`-Werte würden nach `long` konvertiert werden. Pro Rechenoperation würden drei Methoden anstatt einer benötigt, die allerdings teilweise aufeinander zugreifen könnten.

2. Wenn die Anzahl der Methoden klein gehalten werden soll, muss dem Compiler eine Möglichkeit zur Typumwandlung zur Verfügung gestellt werden, wenn er die Daten bei der Wertübergabe in die Rechenfunktionen kopiert. Dies geschieht am günstigsten durch einen Typumwandlungskonstruktor.

Hier: zweite Variante:

```
#ifndef ratio_h
#define ratio_h ratio_h

class rational {
   public:
     rational();
     rational(long z, long n);    // allgemeiner Konstruktor
     rational(long);      // Typumwandlungskonstruktor

     // Abfragen
     long Zaehler() const;
     long Nenner()  const;

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

     // weitere Methoden
     void definiere(long zaehler, long nenner);
```

```
    void eingabe();
    void ausgabe() const;
    void kehrwert();
    void kuerzen();
  private:
    long zaehler, nenner;
};
// inline Methoden
inline rational::rational()
: zaehler(1), nenner(1) {
}
inline rational::rational(long ganzeZahl)
: zaehler(ganzeZahl), nenner(1) {
}
inline rational::rational(long Z, long N)
: zaehler(Z), nenner(N)   {}
inline long rational::Zaehler() const {return zaehler;}
inline long rational::Nenner()  const {return nenner;}
```

```
// globale Funktionsprototypen, werden später durch binäre Operatoren ersetzt
rational add(const rational& a, const rational& b);
rational sub(const rational& a, const rational& b);
rational mult(const rational& a, const rational& b);
rational div(const rational& z, const rational& n);
#endif // ratio_h
```

5.4.3 Implementation

```
void rational::definiere(long z, long n) {
     zaehler = z;
     nenner  = n;
     assert(nenner != 0);
     kuerzen();
}

void rational::eingabe() {
     /* Bildschirmausgabe nur zu Demonstrationszwecken.
        cerr wird gewählt, damit die Abfragen auch dann auf
        dem Bildschirm erscheinen, wenn die Standardausgabe
        in eine Datei zur Dokumentation geleitet wird.
      */
     cerr << "Zähler :";
     cin >> zaehler;
     cerr << "Nenner :";
     cin >> nenner;
     assert(nenner != 0);
     kuerzen();
}
```

```
void rational::ausgabe() const {
     cout << zaehler << "/" << nenner << endl;
}

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.
        Bei Anpassung auf eine Übergabe per Wert muss natür-
        lich auch der Prototyp in ratio.h geändert werden. Die
        Subtraktion kann durch Addition des negativen Argu-
        ments erreicht werden.
      */
     r.zaehler *=-1;
     add(r);
}
```

```
void rational::mult(const rational& r) {
     zaehler = zaehler*r.zaehler;
     nenner  = nenner *r.nenner;
     kuerzen();
}
void rational::div(const rational& n) {
     rational r = n;   // siehe Diskussion bei sub()
     // Division = Multiplikation mit dem Kehrwert
     r.kehrwert();
     mult(r);
}
```

```
long ggt(long x, long y) { // wird zum Kürzen benötigt
    long rest;
    while(y > 0) {
       rest = x % y;
       x=y;
       y=rest;
    }
    return x;
}

void rational::kuerzen() {
    // Vorzeichen merken und Betrag bilden
    int sign = 1;
    if(zaehler < 0) { sign = -sign; zaehler = -zaehler;}
    if(nenner  < 0) { sign = -sign; nenner  = -nenner;}

    long teiler = ggt(zaehler, nenner);
    // Vorzeichen restaurieren
    zaehler = sign*zaehler/teiler;
    nenner = nenner/teiler;
}
```

```
// Es folgen die globalen arithmetische Funktionen
rational add(const rational& a, const rational& b) {
    /* Die eigentliche Berechnung muss hier nicht wiederholt werden, sondern die bereits vorhandenen Funktionen für die Kurzformen der Addition usw. können vorteilhaft wiederverwendet werden. Dazu wird ein mit a initialisiertes temporäres Objekt erzeugt, auf das das Argument b addiert und das dann als Ergebnis zurückgegeben wird. Zum temporären Objekt r siehe auch die Diskussion bei der Elementfunktion sub().
     */
   rational r = a;
   r.add(b);
   return r;
}

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

```
rational mult(const rational& a, const rational& b) {
   rational r = a;
   r.mult(b);
   return r;
}
rational div(const rational& z, const rational& n) {
   rational r = z;
   r.div(n);
   return r;
}
```

5.5 const-Objekte und Methoden

Objekte können wie einfache Variable als *konstant* deklariert werden, z.B.

```
const rational cr;
```

Nicht-verändernde Methoden müssen entsprechend deklariert werden.

```
void rational::ausgabe() const;          // Deklaration

void rational::ausgabe() const {         // Definition
    //.... wie vorher
}
```

Ein konstantes Objekt kann nicht durch `const`- oder andere Funktionen geändert werden, selbst dann nicht, wenn es per Referenz übergeben wird. (Ausnahmen: cast und mutable, siehe angegebene Literatur).

5.6 Faustregeln zur Konstruktion von Schnittstellen

```
// main()-Programm
Geld Kaufpreis(100.00, "Euro");
cout << Kaufpreis.Betrag();       // 100
cout << Kaufpreis.Waehrung();     // Euro
// Ausgabe für Scheckvordrucke:
cout << Kaufpreis.toString();     // eins-null-null
Kaufpreis.neuerBetrag(90.0);      // Betrag ändern
```

Aus der Anwendung ergeben sich vier Prototypen, die im `public`-Bereich der Klasse aufgeführt sind. Die Namen der privaten Attribute sind beliebig gewählt.

```
class Geld {
    public:
      Geld(double einBetrag, const string& eineWaehrung);
      double Betrag() const;
      void neuerBetrag(double);
      const string& Waehrung() const;
      string toString() const;
    private:
      double derBetrag;
      string dieWaehrung;
};
```

Regeln:

1. Die geplante Anwendung bestimmt die Methoden. Beim Entwurf einer Klasse ist es empfehlenswert, sich die Anwendung vorher zu überlegen und *nur solche* Prototypen vorzusehen, die dazu beitragen. (Grund: Alle Methoden müssen irgendwann auch definiert, dokumentiert und getestet werden.)
2. Der Name einer Methode soll gut beschreiben, was die Methode tut.
3. Jede Methode, die den Zustand eines Objekts nicht ändert, erhält den `const`-Qualifizierer.

4. Der Rückgabetyp ist `void`, wenn die Methode zwar etwas tut, aber nichts zurückgibt.

   Andernfalls bestimmt sich der Rückgabetyp aus der Antwort auf die Frage: Ist der zurückgegebene Wert ein Attribut der Klasse?

   - **Ja:** Nächste Frage: Ist der zurückgegebene Wert von einem Grunddatentyp?
     - *Ja:* Rückgabe per Wert. Beispiel: `Betrag()`
     - *Nein:* Rückgabe per `const&`. Beispiel: `Waehrung()`
   - **Nein:** Rückgabe per Wert. Beispiel: `toString()`

5. Die Art der Übergabe eines Objekts in der Parameterliste kann ebenfalls durch die Beantwortung einiger Fragen bestimmt werden. Soll das Objekt beim Aufrufer der Methode verändert werden?

   - **Ja:** Übergabe per Referenz
   - **Nein:** Nächste Frage: Gehört der übergebene Wert zu einem Grunddatentyp?
     - *Ja:* Übergabe per Wert. Beispiel: `neuerBetrag(double)`
     - *Nein:* Übergabe per `const&`. Die Übergabe des Strings im Konstruktor ist ein Beispiel.

6. Wenn Operationen verkettet werden sollen, ist das Objekt selbst per Referenz zurückzugeben. In C++ sind Anweisungen wie `a = b = c;` erlaubt und üblich, ebenso wie die schon bekannte Verkettung von Ausgaben mehrerer Variablen, etwa `cout << a << b << c << endl;`. Beispiel:

```
// Methode der Klasse rational
void add(const rational& r);
```

Die Operation `a.add(b);` ist damit möglich, nicht aber Verkettungen wie

```
a.add(b).add(c);   // a) entspricht: a += b; a += c;
a.add(b.add(c));   // b) entspricht: a += b += c;
```

Die Verkettungen könnten natürlich durch jeweils zwei einzelne Anweisungen ersetzt werden, aber hier geht es darum, zu zeigen, dass und wie Verkettungen mit der Rückgabe des Objekts selbst möglich gemacht werden. Betrachten wir beide Fälle:

(a) Diese Anweisung wird von links abgearbeitet. `a.add(b)` muss dann ein Objekt zurückliefern, auf das dann `add(c)` angewendet wird. Auf einen Rückgabetyp `void` lässt sich keine Funktion anwenden. Dieses zurückgegebene Objekt ist das durch die Addition veränderte Objekt `a`, hier a genannt. Aus `a.add(b)` ergibt sich also a, für das `add(c)` aufgerufen wird. In Einzelschritte zerlegt:

```
a.add(b).add(c);
    a.add(c);
```

Damit ergib sich eine veränderte Implementation (und Deklaration) für `add()`:

```
// veränderter Rückgabetyp:
rational& rational::add(const rational& r) {
     zaehler = zaehler*r.nenner + r.zaehler*nenner;
     nenner  = nenner*r.nenner;
     kuerzen();
     // Rückgabe des Objekts, für das die Methode aufgerufen wird:
     return *this;
}
```

Was bedeuted `*this`? Es ist einfach eine andere Bezeichnung für das Objekt selbst. Es muss in einer Methode so einen universellen Namen geben, weil der beliebige Name, der irgendwann in einem Anwendungsprogramm für ein Objekt vergeben wird, in der Methode unbekannt ist. Das Schlüsselwort `this` wird weiter unten genauer erklärt.

(b) Ein Aufruf der Form `a.add(X)` verlangt, dass das Argument `X` vom Typ `rational` ist. Da `X` identisch mit `b.add(c)` ist und gemeint ist, dass erst `c` auf `b` addiert wird, welches dann auf `a` addiert wird, folgt daraus, dass der Aufruf `b.add(c)` das veränderte `b` zurückgeben muss. Die unter a) angegebene Implementierung löst auch dieses Problem.

Natürlich kann von den Empfehlungen abgewichen werden – wenn es einen Grund dafür gibt. *Kleine* Klassenobjekte kosten nicht viel Zeit bei der Kopie und können daher per Wert übergeben werden statt per Referenz.

Wenn ein Parameter eines Klassentyps nicht verändert werden soll, ist die Übergabe per Wert statt per Referenz auf `const` sinnvoll, wenn er in der Funktion als Variable benötigt wird. Entsprechend der obigen Regel würde die Funktion `Ortsverschiebung()` (Abschnitt 5.3.3) wie folgt geschrieben:

```
                                    // Parameterliste geändert
Ort Ortsverschiebung(const Ort& OriginalOrt, int dx, int dy) {
    Ort derOrt = OriginalOrt;    // neu
    derOrt.aendern(derOrt.X() + dx, derOrt.Y() + dy);
    return derOrt;          // Rückgabe des veränderten Orts
}
```

Die Schreibweise im Abschnitt 5.3.3 spart eine Zeile. Der Rechenaufwand ist in beiden Fällen nahezu gleich, weil der Kopierkonstruktor hier zwar nicht bei der Parameterübergabe, wohl aber bei der Initialisierung der temporären Variablen aufgerufen wird.

5.7 Destruktoren

- Zweck: Aufräumarbeiten für nicht mehr benötigte Objekte
- Wenn Destruktoren nicht vorgegeben werden, werden sie vom System automatisch erzeugt.
- Der häufigste Zweck ist die Speicherfreigabe, wenn der Gültigkeitsbereich eines Objekts verlassen wird.
- Die Reihenfolge des Aufrufs der Destruktoren ist *umgekehrt* wie die der Konstruktoren.
- Destruktoren haben keine Argumente und keinen Rückgabetyp.
- Falls es globale Objekte gibt, wird ihr Konstruktor *vor* der ersten Anweisung von `main()` aufgerufen.
- Innerhalb des äußersten Blocks von `main()` definierte Objekte werden erst nach Verlassen von `main()` freigegeben.
- Wegen der umgekehrten Reihenfolge der Destruktoraufrufe werden globale Objekte zuletzt freigegeben.

```
class Beispiel {
    int zahl;
  public:
    Beispiel(int i = 0);      // Konstruktor
    ~Beispiel();              // Destruktor
};

Beispiel::Beispiel(int i) {   // Konstruktor
    zahl=i;
    cout << "Objekt " << zahl
         << " wird erzeugt.\n";
}

Beispiel::~Beispiel() {       // Destruktor
    cout << "Objekt " << zahl
         << " wird zerstört.\n";
}

// globale Variable, durch Vorgabewert mit 0 initialisiert
Beispiel ein_globales_Beispiel;
```

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

Die Ausgabe des Programms ist:

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.

5.8 Wie kommt man zu Klassen und Objekten? Ein Beispiel

Simulation der Benutzung eines einfachen Getränkeautomaten

In der Kantine stehen Getränkeautomaten, unter anderem einer mit *ObjektCola*. Der Automat sei so eingestellt, dass eine Dose dieses Getränks 2 Euro kostet. Der Automat nimmt einen beliebigen Geldbetrag an. Wenn zu wenig eingeworfen wird, wird das eingeworfene Geld wieder herausgegeben. Wenn zu viel eingeworfen wird, wird eine Dose *ObjektCola* sowie der Restbetrag herausgegeben. Nach Geldeinwurf löst ein Knopfdruck die Prüfung des Geldbetrags und gegebenenfalls die Ausgabe einer Dose aus. Wenn keine Dose mehr vorrätig ist, ist der Automat gesperrt, das heißt, dass jeder eingeworfene Geldbetrag vollständig zurückgegeben wird.

Aufgabe: Das folgende, bewusst einfach gehaltene Szenario soll mit einem Programm simuliert werden:

Der Automat wird anfangs mit 50 Dosen befüllt. Zum Automaten gehen Personen, die eine Dose aus dem Automaten ziehen wollen. Sie tragen unterschiedliche Geldbeträge bei sich. Wenn der Betrag ausreicht, werfen sie Münzen in den Automaten, wobei manche nicht genau zählen und zufällig zu viele oder zu wenige Münzen einwerfen. Anschließend drücken sie auf den Ausgabeknopf mit der Wirkung, dass gegebenenfalls eine Dose und Rückgeld ausgegeben werden. Das Szenario endet, wenn der Automat gesperrt wird, weil er leer ist.

Das Programm soll zur Kontrolle die einzelnen Schritte auf dem Bildschirm dokumentieren. Der Einfachheit halber genügt es, einheitliche Münzen anzunehmen, zum Beispiel 1-Euro-Stücke.

5.8.1 Einige Analyse-Überlegungen

Um die Problemstellung zu verdeutlichen, wird sie aus verschiedenen Blickwinkeln betrachtet, die im Folgenden vorgestellt werden. Es handelt sich dabei nur um *Möglichkeiten*, nicht um den einzig wahren Lösungsansatz (den es nicht gibt).

1. In der Analyse geht es zunächst einmal darum, Prozesse *in der Sprache des (späteren Programm-) Anwenders* zu beschreiben. Dabei sind typische Abläufe, Szenarien genannt, ein gutes Hilfsmittel. Die Aufgabenstellung ist als Szenario formuliert.

2. Im zweiten Schritt wird versucht, beteiligte Objekte, ihr Verhalten und ihr Zusammenwirken zu identifizieren. Dies ist nicht unbedingt einfach, weil spontan identifizierte Beziehungen zwischen Objekten im Programm nicht immer die wesentliche Rolle spielen. Auf einem geeigneten Abstraktionsgrad muss entschieden werden, was zum »System« und was zur »Außenwelt« gehören soll.

(Lösung zusammen mit dem Auditorium entwickeln; vgl. Breymann C++ Kap. 5.8)

6. Intermezzo: Zeiger

– Zeiger und Adressen
– C-Arrays
– C-Zeichenketten
– Dynamische Objekte
– Binäre Ein- und Ausgabe
– Zeiger und Funktionen

6.1 Zeiger und Adressen

Der *Wert* einer Zeigervariablen ist eine *Adresse*.

```
int i = 0;
int *ip;                    // Zeiger auf int
ip =  &i;                   // Adresse von i
*ip = 99;                   // entspricht i=99;
```

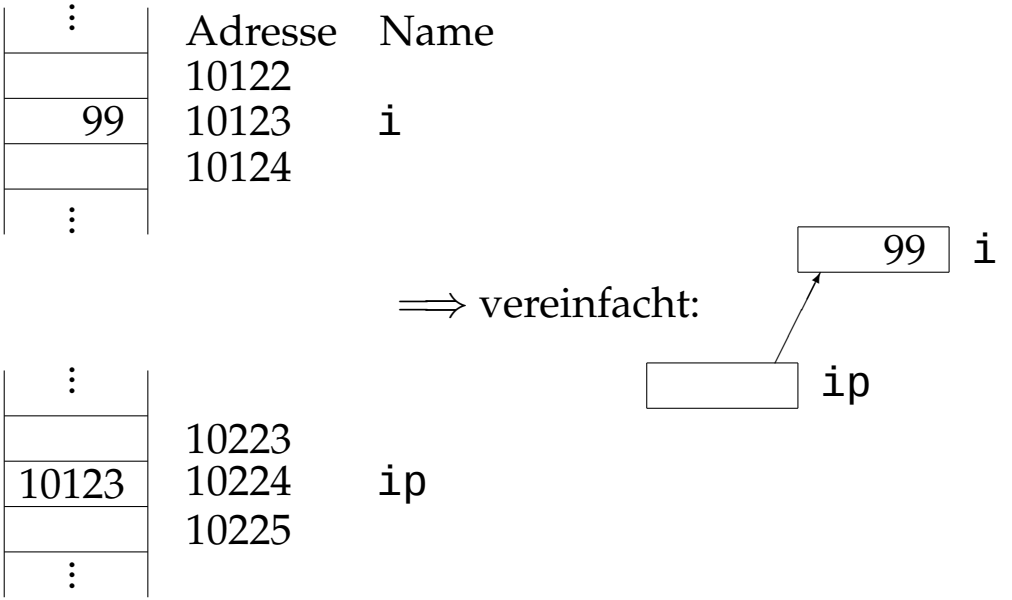


Abb. 6.1: Zeiger

Lesart:

`*ip` hat den Datentyp `int`:

```
   int *ip, i; ↔ int* ip,i;  (Schreibweisen)
≠  int *ip; int *i;          !!!
=  int *ip; int i;
```

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

bewirkt:

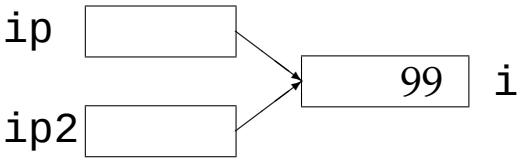


Abb. 6.2: Zwei Zeiger zeigen auf ein Objekt

Äquivalente Anweisungen:

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

Nur mit einem Objekt initialisierte Zeiger dereferenzieren !

```
int *xp;
*xp = -154;        // Fehler!
```

NULL-Zeiger
Eigenschaften:

- Ein mit NULL initialisierter Zeiger zeigt garantiert nicht auf ein Objekt.
- NULL ist als logischer Wert abfragbar.

In C++ wird NULL durch eine 0 implementiert.

6.2 C-Arrays

- primitives Äquivalent zu Vektoren
- „roher“ Speicherbereich
- Nutzung (eingeschränkt) wie Vektoren
- Vektoren basieren auf C-Arrays

Definition:

Datentyp Feldname`[`*Anzahl_der_Elemente*`];`

Anzahl_der_Elemente muss eine Konstante sein oder ein Ausdruck, der ein konstantes Ergebnis hat. Beispiel:

```
const int Anzahl = 5;
int Tabelle[Anzahl];
```

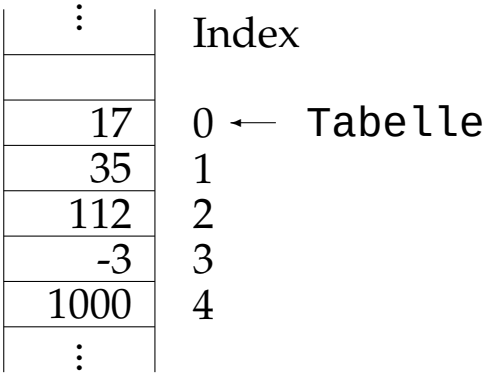


Abb. 6.3: `int`-Array `Tabelle`

Name des Feldes == Startadresse

Nutzung:

```
a[5]        = '?';      // gleichwertig mit Zeigerarithmetik:
*(a+5)      = '?';      // = Wert an der Stelle a + 5

char c = a[9];          // 10. Element

char* cp;       // Zeiger auf char
cp = &c;        // möglich
cp = a;         // cp zeigt auf den Feldbeginn, d.h. *cp==a[0]
a = cp;         // Fehler : a ist kein L-Wert
a = &c;         // Fehler : a ist kein L-Wert
```

Zeiger und Arrays

Zeiger und Arrays sind im Gebrauch sehr ähnlich. Deswegen werden die Unterschiede hier zusammengefasst:

- Ein Zeiger hat einen Speicherplatz, der einen Wert enthält, der als Adresse benutzt werden kann.
- Ein Array besitzt *in diesem Sinne keinen* Speicherplatz. Ein Array ist vielmehr ein symbolischer Name für die Adresse (= den Anfang) eines Bereichs im Speicher. Der Name wird *syntaktisch* wie ein konstanter Zeiger behandelt.

6.2.1 C-Arrays und `sizeof`

C-Arrays sind als »roher Speicher« *keine* Objekte in dem bisherigen Sinn. Es gibt keine Methoden, die verwendet werden könnten:

```
for(size_t i = 0; i < Kosten.size(); ++i) // nicht bei C-Arrays!
    cout << i << ": " << Kosten[i] << endl;
```

Die Größe eines C-Arrays kann nicht von ihm erfragt werden, sie muss vielmehr vorher bekannt oder mit `sizeof` ermittelt worden sein:

```
const int Anzahl = 5;
int Tabelle[Anzahl];                    // C-Array-Definition
// ... Berechnungen
for(size_t i = 0; i < Anzahl; ++i)
    cout << i << ": " << Tabelle[i] << endl;      // oder:
for(size_t i = 0; i < sizeof(Tabelle)/sizeof(Tabelle[0]); ++i)
    cout << i << ": " << Tabelle[i] << endl;
```

`sizeof()` gibt den Platzbedarf eines Objekts in Bytes zurück.

`sizeof` funktioniert jedoch nicht bei C-Arrays, die als Parameter einer Funktion übergeben werden, weil C-Arrays innerhalb der Funktion nur als Zeiger interpretiert werden. Weitere Einzelheiten folgen im Abschnitt zu mehrdimensionalen Arrays. (Weiteres im Abschnitt zu mehrdimensionalen Arrays)

6.2.2 Indexoperator bei C-Arrays

Keine Indexprüfung!

Transformation durch Compiler:

`Tabelle[i]` → `*(Tabelle + i)`

6.2.3 Initialisierung von C-Arrays

```
int feld[ ] = { 1, 777, -500};
const int Anzahl = sizeof(feld) / sizeof(feld[0]);
```

Falls weniger Elemente in der Initialisierungsliste als vorhanden angegeben werden, werden die restlichen Elemente mit 0 initialisiert.

`int feld[3] = {1}` ist identisch mit `int feld[3] = {1, 0, 0}`.

6.3 C-Zeichenketten

C-Zeichenkette = Spezialfall eines Arrays.

C-String = null-terminierte Folge von Zeichen des Typs `char`

Datentyp: `char*`, d.h. Zeiger auf den Anfang der Folge

```
// Beispiel
char * str = "ABC"; // 0 wird ergänzt
cout << str;        // ABC
cout << str[0];     // A
str[0] = 'X';       // Fehler! (nicht portabel)
     // Stringliterale sind (systemabhängig) meistens read-only
```

Zuweisungen sind möglich (Informationsverlust):

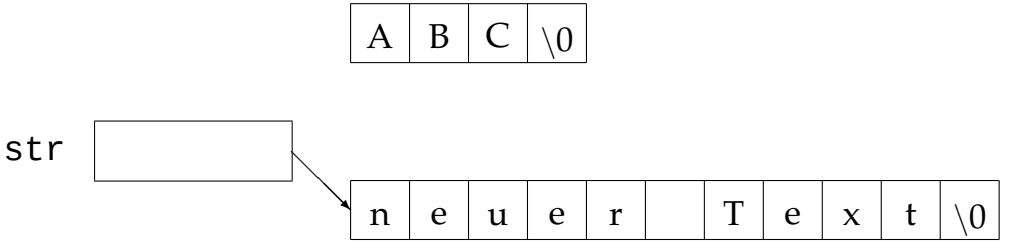


Abb. 6.4: Neuzuweisung eines Strings

Vordefinierte Funktionen: Header <cstring>

```
#include<cstring>
#include<iostream>
using namespace std;
int main() {
    char* text=
      "Bei Initialisierung, Zuweisung, oder Ausgabe"
      " kann ein String wie hier auch aus einzelnen "
      "Teilstrings zusammengesetzt werden.";
    cout << text << endl
         << "enthält " << strlen(text) << " Zeichen\n";
}
```

C-Strings und Initialisierung:

```
// 1.
char* x;
cin >> x;    // Fehler!

// 2.
x = "12345678";
cin >> x;    // Fehler!

// 3.
char Bereich[100];
cin >> Bereich;   // relativ(?) sicher, liest bis zum 1. Zwischenraumzeichen

// 4.
const int zmax = 100;
char Zeile[zmax];
// sicher, liest eine Zeile, aber maximal (zmax - 1) Zeichen:
cin.getline(Zeile, zmax);
```

char-Arrays

```
// Definition und Initialisierung
char str_a [ ] = "noch ein String";
char str_b [9];          // 9 Byte Platz reservieren
                         // d.h. 8 Zeichen 0 bis 7 sowie '\0'
char str_c [9] = "ABC";    // 9 Byte Platz, aber nur die
                          // ersten 4 haben definierte Werte
cin >> str_b;  // Eingabe wie bei Strings. Länge beachten!
// Aliasing, d.h. mehr als einen Namen für nur eine Sache
char * str1 = "Guten Morgen!\n";
char * str2 = str1; // str2 benutzt Platz von str1.
                    // Beweis:
cout << str2;       // Guten Morgen!
// erlaubte Zuweisung oder Änderung
char charArray1[ ] ="hallo\n";
str1 = charArray1; // Zeiger str1 zeigt nun auf Array-Anfang
++str1;            // str1 zeigt jetzt auf das nächste
                   // Zeichen
```

```
// nicht erlaubte Änderungen, weil ein Array kein L-Wert ist:
charArray1 = str1;          // Fehler!
charArray1++;               // Fehler!

// Definition
char buchstaben[3] = "abc"; // Fehler! (kein Platz für '\0')
// tolerante Compiler ignorieren die '3'. Besser:
char buchstaben[4] = "abc";          // oder
char buchstaben[ ] = "abc";          // Compiler zählen lassen, oder
char buchstaben[3] = {'a','b','c'}; // ohne '\0'-Terminierung!
```

Beispiele: Schleifen mit Strings

Stringlänge berechnen

Version 1

```
char *str1 = "pro bonum, contra malum";
char *temp = str1;
int sl = 0;
while(*temp) {++sl; ++temp;}
cout << "Stringlänge von " << str1 << '=' << sl << endl;
```

Version 2

```
char * str2 = "Lieber reich und gesund als arm und krank";
char *temp = str2;
int sl = 0;
while(*temp++) ++sl;
```

Version 3

```
char * str3 = "Morgenstund ist aller Laster Anfang";
int sl = 0;
while(str3[sl++]);
--sl;
```

Version 4

```
char * str4 = "C++";
int sl = -1;
while(str4[++sl]);
```

Version 5

```
char * str5 = "letztes Beispiel zur Stringlängenberechnung";
char *temp = str5;
while(*temp++);
int sl = temp-str5-1;
```

Strings kopieren

```
char* original =
 "Ich verstehe mich! (G. Ch. Lichtenberg, geb. 1742)";
char* duplikat;
```

Die Zuweisung `duplikat = original;` funktioniert nicht!

Definitionen:

```
char * source =
     "Unter allen Tieren steht der Mensch dem "
     "Affen am nächsten.";
char  dest[80];
```

Version 1

```
int i = 0;
while(source[i] != '\0') { dest[i] = source[i]; ++i;}
dest[i] = '\0';
```

Version 2

```
int i = -1;
while(source[++i]) dest[i] = source[i];
dest[i] = '\0';
```

Version 3

```
char *s = source, *d = dest;
while(*s) {*d = *s; ++s; ++d;}
*d = '\0';
```

Version 4

```
char *s = source, *d = dest;
while(*d++ = *s++);
```

6.4 Dynamische Datenobjekte

`new`: Erzeugen von Objekten zur Laufzeit auf dem *Heap*.
`delete`: Löschen von Objekten

```
int *p;                 // Zeiger auf int
p = new int;            // int-Objekt erzeugen

*p = 15;                // Wert zuweisen
cout << *p << endl;     // 15 (*p : „Name“ des Objekts)
```

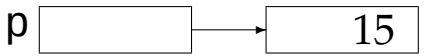


Abb. 6.5: Erzeugen eines `int`-Datenobjekts mit `new`

`*p` == (Wert an der Stelle `p`) kann als Name für das Objekt interpretiert werden.

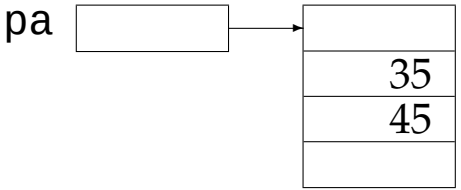


Abb. 6.6: Erzeugen eines `int`-Arrays mit `new[]`

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

Dynamisch erzeugte Struktur

Erzeugung und Löschen wie bei den Grunddatentypen

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

test_struct *sp= new test_struct;
```

Zugriff über Pfeil-Operator `->`:

```
sp->a = 12; // entspricht (*sp).a
sp->b = 17.4;
```

Verkettung:

```
// weiteres Objekt erzeugen
sp->next = new test_struct;

// Zugriff auf Element a des neuen Objekts
sp->next->a = 144;
```

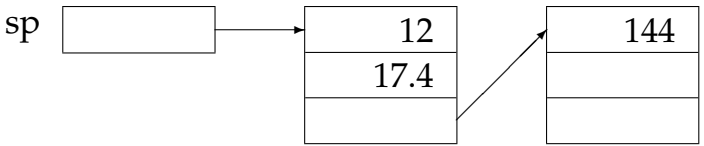


Abb. 6.7: Struktur mit Verweis auf zweite Struktur

6.4.1 Freigeben dynamischer Objekte

In Entsprechung zu den Operatoren `new` und `new [ ]` wird zwischen den Operatoren `delete` und `delete [ ]` unterschieden.

```
// Reihenfolge beachten!
delete sp->next;
delete sp;
```

Einige Regeln:

- `delete` darf *ausschließlich* auf Objekte angewendet werden, die mit `new` erzeugt worden sind.

```
int i;
int *iptr = &i;
delete iptr;          // Fehler! (Absturzgefahr)

iptr = new int;
delete iptr;          // ok!
```

- `delete` darf nur *einmal* auf ein Objekt angewendet werden.

- Der Wert eines Zeigers, auf den `delete` angewendet wurde, ist danach *undefiniert*, er ist also leider nicht gleich NULL. Falls zwei oder mehr Zeiger auf ein Objekt zeigen, bewirkt ein `delete` auf nur einen dieser Zeiger die Zerstörung des Objekts. Folge: »hängende Zeiger« (englisch *dangling pointer*)

- `delete` auf einen NULL-Zeiger angewendet bewirkt nichts und ist unschädlich.

- Wenn ein mit `new` erzeugtes Objekt mit dem `delete`-Operator gelöscht wird, wird automatisch der Destruktor für das Objekt aufgerufen.

- Die Freigabe von Arrays erfordert die Angabe der eckigen Klammern. Ohne `[ ]` gäbe `delete pa;` nur ein einziges Element frei! Der Rest des Arrays wäre danach nicht mehr zugänglich (verwitwetes Objekt).

  Merkregel: `delete [ ]` dann (und *nur* dann) benutzen, wenn die Objekte mit `new [ ]` erzeugt worden sind.

- Mit `new` erzeugte *Objekte* unterliegen *nicht* den Gültigkeitsbereichsregeln für Variable. Sie existieren solange, bis sie mit `delete` gelöscht werden oder das Programm beendet wird, unabhängig von irgendwelchen Blockgrenzen. Dies gilt nicht für die statisch deklarierten *Zeiger* auf diese Objekte, für die die normalen Gültigkeitsbereichsregeln gelten. Konsequenz: ein Zeiger auf ein Objekt sollte mindestens bis zu dessen Löschung existieren.

```
{   // falsch
    int *p = new int;
    *p = 30000;
    // mehr Programmcode
} // »verwitwetes« Objekt, memory leak

{   // richtig
    int *p = new int;
    *p = 30000;
    // mehr Programmcode
    delete p;        // *p wird nicht mehr gebraucht
}
```

oder:

```
int *p;                  // außerhalb des Blocks deklariert
{
   p  = new int;
   *p = 30000;
   // mehr Programmcode
}
cout << *p;              // weitere Verwendung von p
// mehr Programmcode
delete p;
```

Problem der Speicherfragmentierung

Abhilfe: *garbage collection*

6.5 Mehrdimensionale C-Arrays

6.5.1 Statische mehrdimensionale C-Arrays

Gelegentlich hat man es mit mehrdimensionalen Feldern zu tun, am häufigsten mit zweidimensionalen. Eine zweidimensionale Tabelle besteht aus Zeilen und Spalten, eine dreidimensionale aus mehreren zweidimensionalen Tabellen. In C++ wird eine Tabelle linear auf den Speicher abgebildet. Im Fall der zweidimensionalen Tabelle kommen zunächst alle Elemente der ersten Zeile, dann alle Elemente der zweiten Zeile usw.

Ein zweidimensionales Array ist ein Array von Arrays.

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

int main() {
    const int dim1 = 2, dim2 = 3;
    int matrix[dim1][dim2]  = {{1,2,3}, {4,5,6}};

    for(int i = 0; i < dim1; ++i) {
        for(int j = 0; j < dim2; ++j)
            cout << matrix[i][j] << ' ';
        cout << endl;
    }
}
```

Ergebnis des Programms:

1 2 3
4 5 6

Arrays als Funktionsparameter

C-Arrays sind syntaktisch gesehen konstante Zeiger.
Eine Parameterliste (`int Tabelle[]`) entspricht daher (`int* Tabelle`).
Damit kann `sizeof` **nicht** zur Größenermittlung eines Arrays benutzt werden, wie das folgende Beispiel zeigt:

```
// fehlerhaftes Beispiel
void Tabellenausgabe(int Tabelle[]) {    // C-Array-Deklaration
   int AnzahlBytes = sizeof Tabelle;     // Fehler!
                        // Grund: dasselbe wie sizeof(int*)!
   int wieoft = AnzahlBytes/sizeof(int);
   for(int i = 0; i < wieoft; ++i)
      cout << Tabelle[i] << endl;
}
int main() {
   const int Anzahl = 5;
   int Tabelle[Anzahl];              // C-Array-Definition
   // ... Berechnungen
   Tabellenausgabe(Tabelle);         // leider falsch ...
}
```

Folgerung: Die Anzahl der Array-Elemente muss einer Funktion übergeben werden! Dies gilt auch für mehrdimensionale Arrays.

```
// korrigiertes Beispiel
void Tabellenausgabe(int Tabelle[], size_t n) {
   for(int i = 0; i < n; ++i)
      cout << Tabelle[i] << endl;
}
```

2- oder mehrdimensionales Array

```
int feld1[2][3] = {{1,2,3},{4,5,6}};
```

ist ein Array, das 2 Elemente enthält, nämlich zwei Arrays zu je 3 `int`-Werten. Der Typ der zuletzt genannten Arrays ist *Bestandteil des Arraytyps* von `feld1`. Dies gilt entsprechend für Zeiger:

```
// kompatibler Zeiger
int (*pFeld)[3] = feld1;   // zeigt auf Zeile feld1[0]
```

Die Funktion zur Ausgabe dieses Feldes benötigt diese Typinformation in der Parameterliste:

```
void Tabellenausgabe2D(int (*T)[3], int n) {
    // alternativ:  void Tabellenausgabe2D(int T[][3], int n)
  for(int i = 0; i < n; ++i) {
    for(int j = 0; j < 3; ++j)
      cout << T[i][j] << ' ';
    cout << endl;
  }
  cout << endl;
}
```

Mit dieser Typinformation weiß der Compiler, wo jede Zeile anfängt. Innerhalb der Funktion ist `sizeof T[0]` gleich 3 mal `sizeof(int)`. Die Funktion kann für das Array oder den Zeiger gleichermaßen aufgerufen werden:

```
Tabellenausgabe2D(feld1, 2);  // 1 2 3
                              // 4 5 6
++pFeld;                      // zeigt jetzt auf Zeile feld1[1]
Tabellenausgabe2D(pfeld, 1);  // 4 5 6
```

Dieses Schema setzt sich für mehrere Dimensionen fort. Die Information `[3][4]` gehört zum Typ einer dreidimensionalen Matrix `int Matrix3D[2][3][4]`. Die Funktion zur Ausgabe dieser Matrix hat die Schnittstelle

```
void Tabellenausgabe3D(int (*T)[3][4], size_t n) ; // oder
void Tabellenausgabe3D(int T[][3][4], size_t n) ;
```

Funktion für statische zwei-dimensionale Arrays, egal wie groß die Spaltenzahl ist:

```
template<typename Feldtyp>
void Tabellenausgabe2D(Feldtyp 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;
  }
  cout << endl;
}
```

Interpretation von [ ], [ ][ ] usw.

X sei alles, was *vor dem letzten Klammernpaar* steht.
Y sei der Inhalt des letzten Klammerpaares.
Der Compiler wandelt stets X[Y] in *((X)+(Y)) um!

Auf X wird das Verfahren wiederum angewendet, bis alle Indexoperatoren aufgelöst sind, d.h.:

`matrix[i][j]` = `*(matrix[i]+j)` = `*(*(matrix+i)+j)`

`matrix[i]` = Zeiger auf den Beginn der `i`-ten Zeile

Durch die Zeigerarithmetik wird die dahinterstehende Berechnung der tatsächlichen Adresse verborgen, die ja noch die Größe der Datenelemente eines Arrays berücksichtigen muss. Die Position `(matrix + i)` liegt daher (`i` mal `sizeof(matrix[0])`) Bytes von der Stelle `matrix` entfernt.

Aufgaben

6.1 Die Äquivalenz von `*(kosten+i)` und `kosten[i]` ist bekannt. Anstatt `(kosten+i)` könnte man genausogut `(i+kosten)` schreiben, das Ergebnis der Addition wäre das gleiche. Ist es dann richtig, dass die Schreibweise `i[kosten]` äquivalent ist zu `kosten[i]`?

6.2 Geben Sie den für `matrix[2][3]` benötigten Speicherplatz in Byte an, wenn `sizeof(int)` als 4 angenommen wird. An welcher Bytenummer beginnt das Element `matrix[i][j]` relativ zum Beginn das Arrays?

6.3 Schreiben Sie ein Programm zur Multiplikation zweier Matrizen `a[n][m] * b[p][q]`. Das Ergebnis soll in einer Matrix `c[r][s]` stehen. Welche Voraussetzungen gelten für die Zeilen- und Spaltenzahlen n, m, p, q, r, s?

6.4 Schreiben Sie zur Ausgabe von dreidimensionalen Arrays eine Template-Funktion `Tabellenausgabe3D(Feldtyp T, size_t n)` entsprechend dem obigen Muster für zwei Dimensionen.

6.5.2 Dynamisch erzeugte mehrdimensionale Arrays

Mehrdimensionale Arrays mit konstanter Feldgröße

Auf Seite 237 haben Sie die statische Deklaration von mehrdimensionalen Feldern gesehen. Im Abschnitt 6.4 wurden ein Array von `int`-Zahlen und ein Array von Zeigern auf `int` dynamisch erzeugt, also zur Laufzeit des Programms.

Bedeutung von `new`:

```
int *pa = new int[4];
```

`new` gibt einen Zeiger vom Datentyp »Zeiger auf Typ eines Arrayelements« zurück.
D.h. im zweidimensionalen Fall:

```
int (* const p2)[7] = new int [5][7];
```

`new` gibt einen Zeiger vom Datentyp »Zeiger auf eine Zeile mit 7 Elementen« zurück. `p2` ist ein konstanter Zeiger, der auf die Zeile 0 einer zweidimensionalen Matrix mit 5 Zeilen zu je 7 `int`-Zahlen verweist. Sein Typ ist *konstanter Zeiger auf ein int-Array mit 7 Elementen*.

Zugriff auf ein einzelnes Element zum Beispiel

```
p2[1][6] = 66;
```

```
int *pTest    = new int[7];
pTest         = new int [10];   // 7 Elemente nicht mehr zugreifbar!

int *const pC = new int[3];
pC[1]         = 123;            // ok, die Elemente von pC sind
                                // nicht konstant
pC            = new int [33];   // Fehlermeldung des Compilers:
                                // konstantes Objekt pC kann nicht geändert werden!
```

Eine dreidimensionale Matrix ist ein Array von zweidimensionalen Matrizen:

```
int (*const p3)[5][7] = new int [44][5][7];
```

Mehrdimensionale Arrays mit variabler Feldgröße

Mit variabler Feldgröße ist hier gemeint, dass die Größe des Arrays zur Compilierzeit nicht bekannt ist. Die Größe kann dann natürlich nicht als Teil des Datentyps aufgefasst werden!

Lösung: 1. Feld von Zeigern auf eindimensionale Arrays (Zeilen) anlegen, 2. jedem dieser Zeiger eine Zeile mit Spalten zuordnen.

```
int z,s;                    // Zeilen , Spalten
cout << "Zeilen und Spalten eingeben:";
cin >> z >> s;              // erst zur Laufzeit bekannt
```

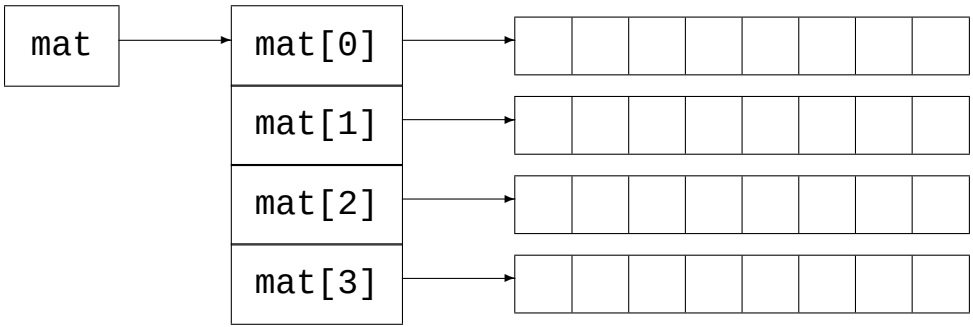


Abb. 6.8: Zweidimensionales dynamisches Array

```
// Feld von Zeigern auf Zeilen anlegen:
// mat ist ein konstanter Zeiger auf Zeiger auf int
int **const mat = new int* [z];

// jeder Zeile Speicherplatz zuordnen:
for(int i = 0; i < z; ++i)
    mat[i] = new int [s];
```

`mat` kann nun wie eine gewöhnliche Matrix benutzt werden. Der Zugriff auf ein Element der Matrix `mat` in Zeile `i` und Spalte `j` wird vom Compiler in die entsprechende Zeigerdarstellung umgewandelt.

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

Der für `mat` mit `new` angelegte Speicherbereich muss nach Gebrauch explizit wieder freigegeben werden, falls er nicht bis zum Programmende bestehen bleiben soll. Wir dürfen jedoch `delete` nicht unmittelbar auf `mat` anwenden, weil dann die einzelnen Zeilen nicht mehr freigegeben werden könnten. Damit ergibt sich folgender Ablauf:

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

6.6 Binäre Ein-/Ausgabe

= unformatierte Ausgabe mit `read()` und `write()`.

Anwendungen:

- Schreiben großer Blöcke
- Einsparen der Formatierung

Schnittstelle für `read()`:

- Adresse des Bereichs, der die gelesenen Daten aufnimmt, als `char*`
- Anzahl der zu transferierenden Bytes

Schnittstelle für `write()`:

- Adresse des Bereichs, der geschrieben werden soll, als `char*`
- Anzahl der zu transferierenden Bytes

Beispiel:

Programm 1: Erzeugen einer Datei *double.dat* mit 20 `double`-Zahlen, die hier durch die Vorschrift 1.1^i berechnet werden.

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

int main( ) {
    ofstream Ziel;
    Ziel.open("double.dat", ios::binary|ios::out);
    if(!Ziel) {
        cerr << "Datei kann nicht geöffnet werden!\n";
        exit(-1);
    }

    // Schreiben von 20 double-Zahlen
    double d = 1.0;
    for(int i = 0;i < 20; ++i, d *= 1.1)
      Ziel.write(reinterpret_cast<const char*>(&d), sizeof(d));
}  // Ziel.close( ) wird vom Destruktor durchgeführt
```

```
// Lesen einer Datei double.dat mit double-Zahlen
#include<cstdlib>
#include<fstream>
using namespace std;

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;
    int i = 1;
    // lesen:
    while(Quelle.read(reinterpret_cast<char*>(&d), sizeof(d)))
         cout << i++ <<"  "<< d << '\n';
}  // Quelle.close() wird vom Destruktor durchgeführt
```

6.7 Zeiger und Funktionen

6.7.1 Parameterübergabe mit Zeigern

= Spezialfall der Übergabe per Wert

- Änderung des Zeigers in der Funktion beeinflusst nicht den Aufrufer.
- Das Objekt, auf das der Zeiger verweist, kann in der Funktion geändert werden.

Beispiel:

```
#include<iostream>
void upcase(char *);      // Prototyp
int main( ) {
    char str[] = "gross 123";
    cout << str;
    upcase(str);
    cout << str << endl;   // GROSS 123
}
```

```
void upcase(char* s) {
    // In der ASCII-Tabelle sind die Platznummern der
    // Kleinbuchstaben um 'a'-'A' = 32 gegenüber den
    // Großbuchstaben verschoben.
    int Differenz = 'a' - 'A'; // impl. Typumwandlung

    while(*s) {
        if(*s >= 'a' && *s <= 'z')
            *s = *s - Differenz; // impl. Typumwandlung
        else
           switch(*s) {
               case 'ä' : *s = 'Ä'; break;
               case 'ö' : *s = 'Ö'; break;
               case 'ü' : *s = 'Ü'; break;
               default:;
           }
        ++s;
    }
}
```

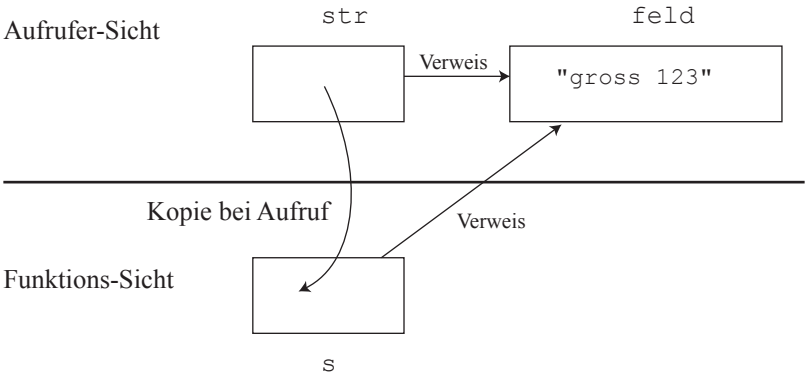


Abb. 6.9: Parameterübergabe per Zeiger (Bezug: Beispielprogramm)

6.7.2 Gefahren bei der Rückgabe von Zeigern

Negativ-Beispiel

```
#include<iostream>

char* murks(char * text) {
    char neu[100];          // Speicherplatz besorgen
    char* n = neu;
    // text wird nach neu kopiert:
    while(*n++ = *text++);
    return neu;             // Fehler!
}

int main() {
    char *sp3= murks("Oh je!");
    cout << sp3;            // nicht existierendes Objekt!
}
```

6.8 Zeiger auf Funktionen

Zeiger auf Funktionen ermöglichen es, erst zur Laufzeit zu bestimmen, welche Funktion ausgeführt werden soll (englisch *late binding, dynamic binding*).

```
#include<iostream>
using namespace std;
int max(int x, int y) {
    if(x > y) return x;
    else return y;
}
int min(int x, int y) {
    if(x < y) return x;
    else return y;
}
```

```
int main() {
    int a = 1700, b = 1000;
    int (*fp)(int, int); // Zeiger auf eine Funktion
    do {
      char c;
      cout << "max(1) oder min(0) ausgeben (sonst=Ende)?";
      cin >> c;
      switch(c) {                    // Zuweisung von max() oder min()
          case '0': fp = &min; break;    // Funktionsadresse zuweisen
          // Ohne den Adressoperator & wandelt der Compiler den
          // Funktionsnamen automatisch in die Adresse um:
          case '1' : fp = max; break;
          default  : fp = 0;
      }
      if(fp) {
          cout << (*fp)(a,b) << endl;   // Aufruf
          cout << fp(a,b) << endl;   // mit Typumwandlung
      }
    } while(fp != 0);
}
```

Callback mit Zeigern auf Funktionen: Standardfunktion `qsort()`

```
void qsort( void *feldname,
            unsigned feldgroesse,
            unsigned feldelementgroesse,
            int (*cmp)(const void *a, const void *b));
```

Der letzte Parameter ist ein Zeiger auf eine Funktion, die

- einen `int`-Wert zurückgibt und
- zwei Zeiger auf `const void` als Parameter verlangt.

Der Name `cmp` steht für »compare« und ist hier nur zu Dokumentationszwecken vorhanden; er kann auch weggelassen werden.

`cmp()` ist vom Anwender vorzugeben / zu schreiben. Anforderung:

Bedingung	Rückgabewert bei	
	aufsteigender Sortierung	absteigender Sortierung
`*a` $<$ `*b`	< 0	> 0
`*a` $>$ `*b`	> 0	< 0
`*a == *b`	0	0

```
#include<iostream>
using namespace std;
#include<cstdlib>    // enthält Prototyp von qsort()

// Definition der Vergleichsfunktion
int icmp(const void *a, const void *b) {
    // Typumwandlung der Zeiger auf void in Zeiger auf int
    // und anschließende Dereferenzierung (von rechts lesen)
    int ia = *static_cast<const int*>(a);
    int ib = *static_cast<const int*>(b);
    // Vergleich und Ergebnisrückgabe ( > 0, = 0, oder < 0)
    if(ia == ib) return 0;
    return ia > ib? 1 : -1;
}
int main() {
    int ifeld[] = {100,22,3,44,6,9,2,1,8,9};
    // Die Feldgröße ist die Anzahl der Elemente des Feldes.
    // Feldgröße = sizeof(Feld) / sizeof(ein Element)
    int Groesse = sizeof(ifeld)/sizeof(ifeld[0]);
```

```
    // Aufruf von qsort():
    qsort(ifeld, Groesse, sizeof(ifeld[0]), icmp);

    // Ausgabe des sortierten Feldes:
    for(int i = 0; i < Groesse; ++i)
        cout << " " << ifeld[i];
    cout << endl;
}
```

Innerhalb von `qsort()` steht an irgendeiner Stelle etwa sinngemäß:

```
if((*icmp)(feld+i, feld+j) < 0) {   // Callback
 ...
}
else     // ...usw.
```

Einsatz eines Callbacks

→ immer dann, wenn ein Server eine Dienstleistung erbringen soll und dazu eine Funktion oder Methode des Clienten benötigt.

6.9 `this`-Zeiger

- `this` ist ein Schlüsselwort.
- `this` ist innerhalb der Methoden einer Klasse ein Zeiger auf das Objekt selbst.
- `*this` bezeichnet das Objekt (Alias).

7. Objektorientierung 2

– eine String-Klasse

– Klassenfunktionen

– Klassentemplates

7.1 Eine String-Klasse

Übungsbeispiel

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

class mstring {
  public:
   mstring();                              // Standardkonstruktor
   mstring(const char *);                  // allg. Konstruktor
   mstring(const mstring&);                // Kopierkonstruktor
   ~mstring();                             // Destruktor
   mstring& assign(const mstring&);        // Zuweisung eines mstring
   mstring& assign(const char *);          // Zuweisung eines char*
```

```
    const char& at(std::size_t position) const; // Zeichen holen
    char& at(std::size_t position);       // Zeichen holen,
                        // die Referenz erlaubt Ändern des Zeichens
    std::size_t length() const { return len;}  // = Anzahl der Zeichen
    const char* c_str() const { return start;} // C-String zurückgeben
    friend void anzeigen(std::ostream&, const mstring&); // siehe Text
  private:
    char *start;                  // Zeiger auf den Anfang
    size_t len;                   // Länge
};

void anzeigen(std::ostream&, const mstring&);// s.u.
#endif // mstring_h, Rest siehe Buch Kap. 7.1
```

Erläuterung zum Kopierkonstruktor:

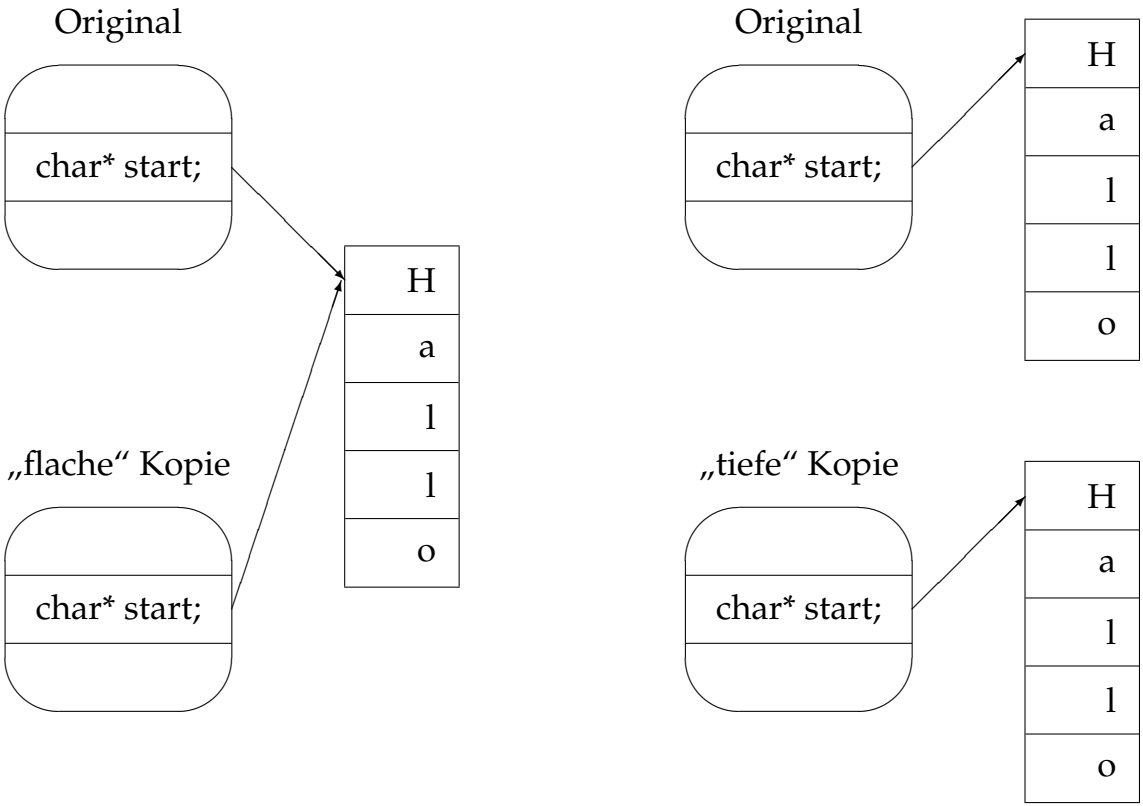


Abb. 7.1: »flache« und »tiefe« Kopie eines Objekts

7.1.1 `friend`-Funktionen

Optionen für eine Funktion `anzeigen()`:

1. Elementfunktion `void anzeigen(ostream &os)`

   Verwendung:

   ```
   einString.anzeigen(std::cout);
   einString.anzeigen(std::cerr);
   ```

   Innerhalb der Methode kann direkt auf `start` zugegriffen werden, weil `start` vom Grunddatentyp `char*` ist und der Ausgabeoperator `<<` für alle Grunddatentypen definiert ist:

   ```
   // Version 1
   void mstring::anzeigen(std::ostream &os) const {
       os << start;
   }
   ```

2. Globale Funktion

   `void anzeigen(std::ostream &os, const mstring &m)`

   Dank `c_str()` kein direkter Zugriff auf privaten Daten.

```
// Aufruf:
anzeigen(cout, einString);

// Implememention:
// Version 2
void anzeigen(std::ostream &os, const mstring &m) {
        os << m.c_str();
}
```

3. `friend`-Funktion

   Falls es aus Laufzeitgründen nicht erwünscht ist, für jedes Zeichen einen Funktionsaufruf zu spendieren, andererseits aber keine Elementfunktion benutzt werden soll (hier nicht sinnvoll):

   `friend`-Funktion

   - Eine `friend`-Funktion ist *keine* Methode der Klasse.
   - Sie darf aber auf private Daten und Methoden zugreifen.
   - Die Erlaubnis dazu wird durch eine `friend`-Deklaration in der Klasse erteilt.

```
// Version 3
void anzeigen(ostream &os, const mstring &m) {
    os << m.start;
}
```

Empfehlung: `friend` vermeiden. Nur falls unbedingt notwendig, einsetzen (Begründung?)

7.2 Klassenspezifische Daten und Funktionen

Klassenspezifische Daten sind Daten, die *nur einmal* für eine Klasse existieren.

Sie sind nicht an einzelne Objekte, sondern an alle Objekte einer Klasse gleichzeitig gebunden.

Dies können Verwaltungsdaten wie die Anzahl der Objekte sein, oder auch Bezugsdaten, die für alle Objekte gelten.

Klassenspezifische Funktionen führen Aufgaben aus, die an eine Klasse, nicht aber an ein Objekt gebunden sind.

Beispiel

Die Klasse `nummeriertesObjekt` hat die Aufgabe, jedem Objekt eine unverwechselbare Seriennummer mitzugeben und über die aktuelle Anzahl aktiver Objekte Buch zu führen.

- Die `static`-Funktion `Anzahl()` soll die Anzahl der Objekte dieser Klasse zurückgeben und ist daher objektunabhängig.
- Die Funktion `Seriennummer()` bezieht sich im Gegensatz dazu nur auf ein einzelnes Objekt.
- Die öffentliche Variable `Testmodus` dient dazu, während der Laufzeit eines Programms den Testmodus für bestimmte Programmabschnitte aus- oder einzuschalten, um auf der Standardausgabe Entstehung und Vergehen aller Objekte zu dokumentieren.

```
#ifndef numobj_h
#define numobj_h

class nummeriertesObjekt { // noch nicht vollständig!
    public:
      nummeriertesObjekt();
      nummeriertesObjekt(const nummeriertesObjekt&);
      ~nummeriertesObjekt();
      unsigned long Seriennummer() const {
        return SerienNr;
      }
      static int Anzahl() { return anzahl;}
      static bool Testmodus;

    private:
      static int anzahl;  // int statt unsigned
      static unsigned long maxNummer;
      const unsigned long SerienNr;
};
#endif       // Ende von numobj.h
```

```
// Implementation der Klasse nummeriertesObjekt
#include<iostream>
#include"numobj.h"
#include<cassert>

// Initialisierung und Definition der klassenspezifischen Variablen:
int           nummeriertesObjekt::anzahl    = 0;
unsigned long nummeriertesObjekt::maxNummer = 0L;
bool          nummeriertesObjekt::Testmodus = false;
```

Zur Initialisierung der `static`-Attribute genügt die Angabe von Typ, Klasse und Variablenname.

Der Standardkonstruktor initialisiert die objektspezifische Konstante `SerienNr` in der Initialisierungsliste und aktualisiert die Anzahl aller Objekte:

```
// Standardkonstruktor
nummeriertesObjekt::nummeriertesObjekt()
:  SerienNr(++maxNummer) {
     anzahl++;
     if(Testmodus) {
        if(SerienNr == 1)
           cout << "Start der Objekterzeugung!\n";
        cout << "  Objekt Nr. "
             << SerienNr << " erzeugt" << endl;;
     }
}
```

Der Kopierkonstruktor hat hier eine besondere Bedeutung. Bisher diente er dazu, ein exaktes Duplikat eines Objekts bei der Initialisierung zu erzeugen. Das darf hier nicht sein. Warum?

```
// Kopierkonstruktor
nummeriertesObjekt::nummeriertesObjekt(
                const nummeriertesObjekt& X)
:  SerienNr(++maxNummer) {
     anzahl++;
     if(Testmodus)
         cout << "  Objekt Nr. " << SerienNr
              << " mit Nr. "  << X.Seriennummer()
              << " initialisiert" << endl;;
}
```

Die Klassendeklaration ist noch nicht vollständig: Was geschieht bei der Zuweisung eines Objekts, z.B.

```
nummeriertesObjekt NumObjekt1;
nummeriertesObjekt NumObjekt2;
// ... irgendwelcher Programmcode
NumObjekt2 = NumObjekt1;                    // ?
```

Antwort?

(Lösung des Problems wird verschoben auf das Kapitel „Überladen von Operatoren“)

Der Destruktor vermerkt, dass es nun ein Objekt weniger gibt. Diagnosechance: Ein versehentliches zusätzliches `delete` kann vom Destruktor festgestellt werden.

```
// Destruktor
nummeriertesObjekt::~nummeriertesObjekt() {
    --anzahl;
    if(Testmodus)  {
       cout << "  Objekt Nr. " << SerienNr << " gelöscht" << endl;
       if(anzahl == 0)
          cout << "letztes Objekt gelöscht!" << endl;
       if(anzahl < 0)
          cout << " FEHLER! zu oft delete aufgerufen!"
               << endl;;
    }
    else assert(anzahl >= 0);
}  // Ende von numobj.cpp
```

Konstruktor und Destruktor dokumentieren Werden und Vergehen der Objekte, sofern der Testmodus eingeschaltet ist.

Klassenspezifische Funktionen können auch objektgebunden aufgerufen werden. Aber:
Der klassenbezogene Aufruf einer Funktion oder die klassenbezogene Benennung eines Attributs ist dem objektgebundenen Aufruf vorzuziehen, weil der objektgebundene Aufruf die `static`-Eigenschaft verschleiert.

```
// Demonstration von nummerierten Objekten
#include "numobj.h"
#include<iostream>
using namespace std;

int main() {
    // Testmodus für alle Objekte der Klasse einschalten
    nummeriertesObjekt::Testmodus=true;

    nummeriertesObjekt dasNumObjekt_X;
    cout << "Die Serien-Nr. von dasNumObjekt_X ist: "
         << dasNumObjekt_X.Seriennummer() << endl;
```

```
// Anfang eines neuen Blocks
{    nummeriertesObjekt dasNumObjekt_Y;

     // schlechter Stil: objektgebundener Aufruf:
     cout << dasNumObjekt_Y.Anzahl()
          << " Objekte aktiv" << endl;
     // *p wird dynamisch erzeugt:
     nummeriertesObjekt *p = new nummeriertesObjekt;
     // schlechter Stil: objektgebundener Aufruf:
     cout << p->Anzahl()
          << " Objekte aktiv" << endl;
     delete p;                    // *p wird gelöscht
     // guter Stil: klassenbezogener Aufruf:
     cout << nummeriertesObjekt::Anzahl()
          << " Objekte aktiv" << endl;
     delete p; // Fehler: ein delete zuviel!
}              // Blockende: dasNumObjekt_Y wird gelöscht

cout << " Kopierkonstruktor: " << endl;
nummeriertesObjekt dasNumObjekt_X1 = dasNumObjekt_X;
```

```
    cout << "Die Serien-Nr. von dasNumObjekt_X ist:"
         << dasNumObjekt_X.Seriennummer() << endl;

    cout << "Die Serien-Nr. von dasNumObjekt_X1 ist:"
         << dasNumObjekt_X1.Seriennummer() << endl;

    // Zuweisung wird wegen const SerienNr
    // vom Compiler verboten
     dasNumObjekt_X1 = dasNumObjekt_X;   // Fehler
}                  // dasNumObjekt_X wird gelöscht
```

7.2.1 Klassenspezifische Konstante

- Klassenspezifische Variable müssen außerhalb der Klassendefinition definiert und initialisiert werden.
- Dies gilt nicht für klassenspezifische Konstanten, für die der Compiler keinen Platz anlegen muss, weil er direkt ihren Wert einsetzen kann.
- In C++ ist diese Ausnahme jedoch auf integrale und Aufzählungstypen beschränkt.

```
class KlasseMitKonstanten {
   enum RGB {rot = 0x0001, gelb = 0x0002, blau = 0x0004};
   static const unsigned int maximaleZahl = 1000;
   // Verwendung zum Beispiel:
   static int CArray[maximaleZahl];
   // ..
```

Diese Konstanten werden *innerhalb* der Klassendefinition initialisiert. Auch wenn das Schlüsselwort `static` fehlt (siehe `enum`), sind sie für alle Objekte einer Klasse gleich, also klassen- und nicht objektspezifisch.

7.3 Klassentemplates

Templates für Klassen == »parametrisierte Datentypen«

Benutzung immer dann, wenn das *Verhalten* eines ADT für verschiedene Datentypen gleich sein soll, z.B.:

- Liste für `int`-Zahlen
- Liste für `double`-Zahlen
- Liste für Adressen ...

7.3.1 Ein Stack-Template

```
// ein einfaches Stack-Template
#ifndef simstack1_t
#define simstack1_t
#include<cassert>
```

In der ersten Variante wird die Größe des Stacks in der Klasse festgelegt:

```
template <class T>
class simpleStack {
  public:
    static const unsigned int maxSize = 20;
    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

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

    void push(const T &x);         // x auf den Stack legen
    // Vorbedingung für push: Stack ist nicht voll
   private:
    unsigned int anzahl;
    T array[maxSize];              // Behälter für Elemente
};
```

```
// noch fehlende Methoden-Implementierungen

template <class T>
const T& simpleStack<T>::top() const {
    assert(!empty());
    return array[anzahl-1];
}

template <class T>
void simpleStack<T>::pop() {
    assert(!empty());
    --anzahl;
}

template <class T>
void simpleStack<T>::push(const T &x) {
    assert(!full());
    array[anzahl++] = x;
}
#endif      // simstack1_t
```

- Der Datentyp T steht für einen beliebigen Datentyp als *Platzhalter*.
- Bei der Definition der Methoden außerhalb der Klasse muss der Datentyp in spitzen Klammern zusätzlich zum Namen der Klasse angegeben werden (<T>).
- Innerhalb der Klassendefinition wird der Typ T bei den Prototypen der Methoden vorausgesetzt, wenn er nicht angegeben ist.
- Bei der Benutzung in einem Programm wird ein konkreter Datentyp angegeben.

Anwendungsbeispiel:

```
#include<iostream>
#include"simstack1.t"

int main() {
    simpleStack<int> einIntStack;

    // Stack füllen
    int i=100;
    while(!einIntStack.full())
          einIntStack.push(++i);
    cout << "Anzahl : "
         << einIntStack.size() << endl;

    // oberstes Element anzeigen
    cout << "oberstes Element: "
         << einIntStack.top() << endl;

    cout << "alle Elemente entnehmen und anzeigen: "
         << endl;
    while(!einIntStack.empty()) {
        i = einIntStack.top();
        einIntStack.pop();
```

```
    cout << i << '\t';
}
cout << endl;

// ein Stack für double-Zahlen
simpleStack<double> einDoubleStack;

// Stack mit (beliebigen) Werten füllen
double d=1.00234;
while(!einDoubleStack.full()) {
     d = 1.1*d;
     einDoubleStack.push(d);
     cout << einDoubleStack.top() << '\t';
}

// Fehler, da Stack voll:
// einDoubleStack.push(1099.986);

cout << "\n4 Elemente des Double-Stacks entnehmen:"
     << endl;
for(i = 0; i < 4; ++i) {
    cout << einDoubleStack.top() << '\t';
    einDoubleStack.pop();
```

```
    }
    cout << endl;
    cout << "Restliche Anzahl : "
         << einDoubleStack.size() << endl;

    cout << "clear Stack" << endl;
    einDoubleStack.clear();
    cout << "Anzahl : "
         << einDoubleStack.size() << endl;

    // einDoubleStack.pop(); // Fehler, da Stack leer
}
```

Nicht nur Grunddatentypen als Argumente:

```
simpleStack<rational> einStackFuerRationaleZahlen;
simpleStack<Datum> einStackFuerDaten;
```

7.3.2 Stack mit statisch festgelegter Größe

Schönheitsfehler des `simpleStack`: fest eingestellte Größe `maxSize`

Alternative 1: dynamische Beschaffung von Speicherplatz

Alternative 2: statische Konfiguration mit Templates
Die Größe eines Stacks gehört dann zum Datentyp, wird also zur Compilierzeit festgelegt. Die geringfügigen Änderungen sind

- Streichen der Konstante `maxSize`,
- Erweiterung der Template-Parameter um `maxSize` und Anpassung der darauf aufbauenden Definitionen:

```
// einfaches Stack-Template, 2. Version
#ifndef simstack2_t
#define simstack2_t
#include<cassert>

template <class T, int maxSize>
class simpleStack {
    // ... genau wie oben
};
```

```
// noch fehlende Implementierungen, int-Parameter wird nicht benutzt
template <class T, int m>
const T& simpleStack<T, m>::top() const {
    assert(!empty());
    return array[anzahl-1];
}

template <class T, int m>
void simpleStack<T, m>::pop() {
    assert(!empty());
    --anzahl;
}

template <class T, int m>
void simpleStack<T, m>::push(const T &x) {
    assert(!full());
    array[anzahl++] = x;
}
#endif      // simstack2_t
```

Nur die Deklarationen in einem Anwendungsprogramm sind ebenfalls zu modifizieren, die Benutzung bleibt sonst gleich:

```
// ein int-Stack mit max. 100 Elementen:
simpleStack<int,100> einIntStack;

// Stack füllen
int i;
while(!einIntStack.full()) {
    cin >> i;
    einIntStack.push(i);
}

// ein char-Stack mit max. 9 Elementen
simpleStack<char,9> einCharStack;
// ...
```

Weil die Größe `size` nicht dem Objekt, sondern dem Template übergeben wurde, ist die Größe eines `simpleStack` schon zur Übersetzungszeit bekannt, so dass `simpleStack`-Objekten statisch Speicherplatz zugeteilt wird, also ohne Rückgriff auf den dynamischen Speicher. Daraus ergibt sich ein Laufzeitvorteil.

8. Vererbung

- Eigenschaften, die einer Menge von Dingen gemeinsam sind, können als verallgemeinertes Konzept betrachtet werden, das besonders behandelt wird.
- Es gibt geringe Unterschiede zwischen diesen Dingen.
- Die Vererbung ist hierarchisch organisiert.

Vererbung = „*ist-ein*“-Beziehung

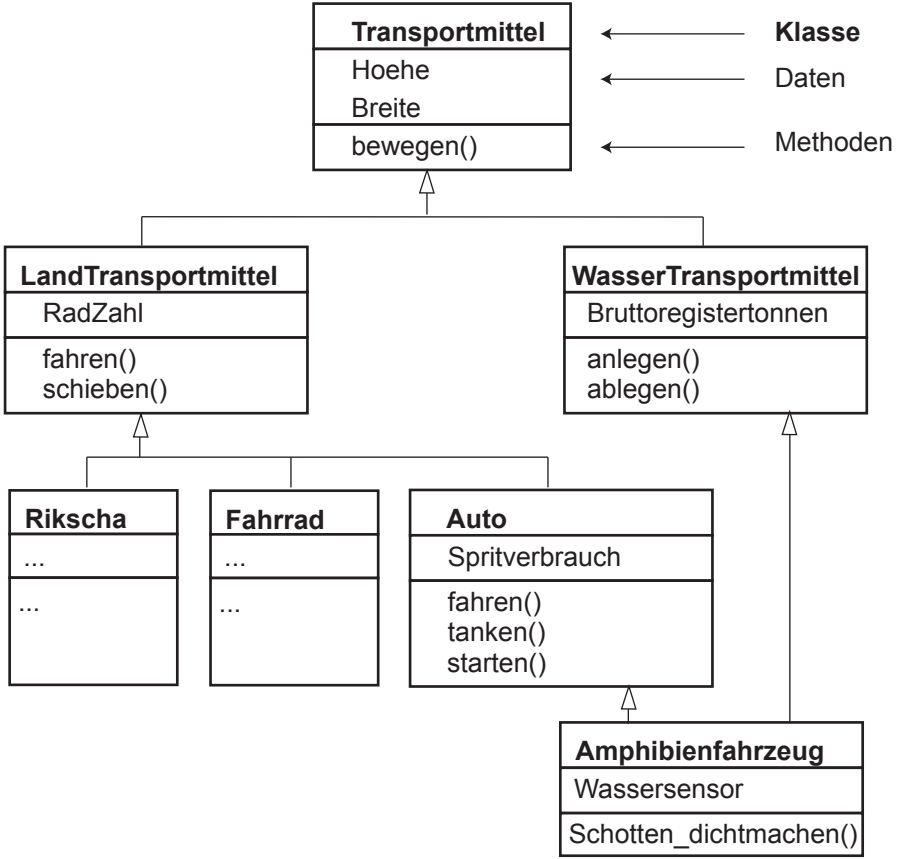


Abb. 8.1: Vererbung von Daten und Methoden

Die Unterklasse *erbt* von der Oberklasse

- die Eigenschaften (Daten) und
- das Verhalten (die Methoden).

Syntax :

```
class Klassenname : public Oberklassenname
```

`: public` = „ist eine Art“

```
// eine Oberklasse
class Transportmittel {
   public:
      bewegen();
    private:
      double hoehe, breite;
      double maxGeschwindigkeit;
};
```

abgeleitete Klasse:

```
class LandTransportmittel
   : public Transportmittel  { // erben
    public:
      fahren();
      schieben();
    private:
      int RadAnzahl;
};
```

etc. etc.

Jedes Objekt ObjA vom Typ A (=abgeleitet) enthält ein (anonymes) Objekt (Subobjekt) vom Typ der Oberklasse.

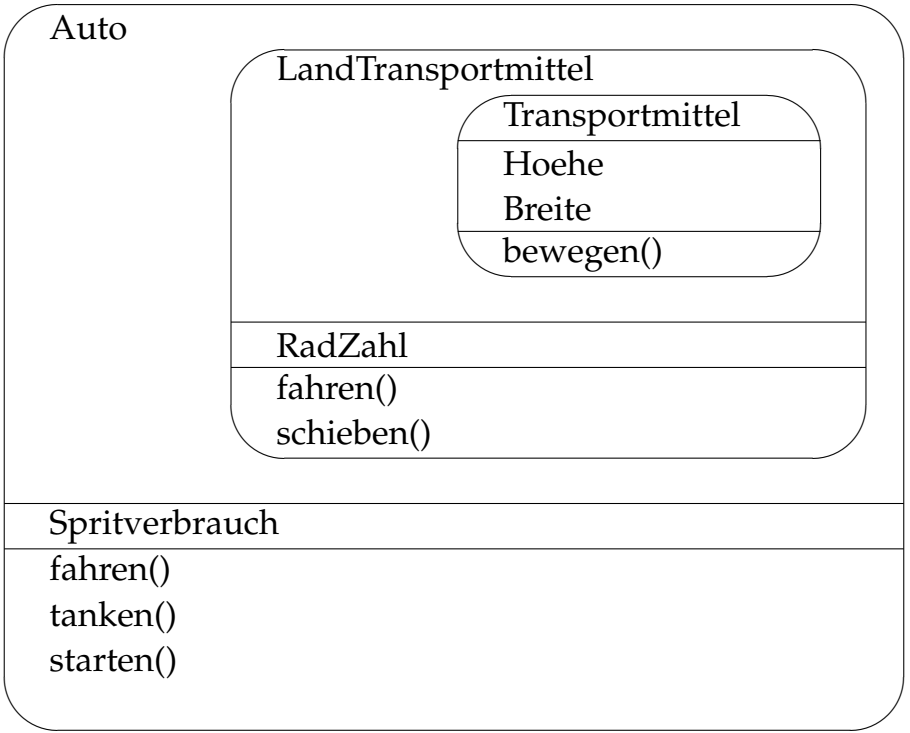


Abb. 8.2: Einschluss von Subobjekten

- Jede öffentliche Elementfunktion der Oberklasse O kann auf ein Objekt des Typs A angewendet werden.
- Die Klasse A kann Erweiterungen der Daten und zusätzliche Methoden enthalten, die keinen Bezug zur Oberklasse haben.
- Zusätzliche Methoden können in A deklariert werden, die im Namen mit Elementfunktionen der Oberklasse übereinstimmen.
- Eine Klasse ist ein *Datentyp* in C++. Eine abgeleitete Klasse A kann als *Subtyp* der Oberklasse O aufgefasst werden.

```
O einO;
A einA;
einO   = einA;       // erlaubt
O* po  = &einA;

einA   = einO;       // nicht möglich
A* pa  = &einO;
```

Klasse GraphObj

```
#ifndef graphobj_h
#define graphobj_h
#include"ort.h"

class GraphObj {                    // Version 1
  public:
    GraphObj(const Ort &einOrt)     // allg. Konstruktor
    : Referenzkoordinaten(einOrt) {} // kein Std.-Kon.
    // Bezugspunkt ermitteln
    Ort Bezugspunkt() const { return Referenzkoordinaten;}
    // alten Bezugspunkt ermitteln und gleichzeitig neuen wählen
    Ort Bezugspunkt(const Ort &nO){
        Ort temp = Referenzkoordinaten;
        Referenzkoordinaten = nO;
        return temp;
    }
    // Koordinatenabfrage
    int X() const { return Referenzkoordinaten.X(); }
```

```
    int Y() const { return Referenzkoordinaten.Y(); }
    // Standardimplementation:
    double Flaeche() const {return 0.0;}
  private:
    Ort Referenzkoordinaten;
};
```

Die Entfernung zwischen 2 GraphObj-Objekten ist hier als Entfernung ihrer Bezugspunkte definiert (überladene Funktion, vgl. Abschnitt 5.3.2)

```
inline double Entfernung( const GraphObj &g1,
                          const GraphObj &g2) {
  return Entfernung(g1.Bezugspunkt(), g2.Bezugspunkt());
}
#endif    // graphobj_h
```

Klasse Strecke

```
#ifndef strecke_h
#define strecke_h
#include"graphobj.h"

class Strecke : public GraphObj {    // erben von GraphObj
   public:
     Strecke(const Ort &Ort1, const Ort &Ort2)
     : GraphObj(Ort1),      // Initialisierung des Subobjekts
       Endpunkt(Ort2)       // Initialisierung des Attributs
     { }                    // leerer Code-Block

     double Laenge() const {
       return Entfernung(Bezugspunkt(), Endpunkt);
     }

   private:
     Ort Endpunkt; // zusätzlich: 2. Punkt der Strecke
       // (der erste ist GraphObj::Referenzkoordinaten)
};
#endif // strecke_h
```

8.1 Vererbung und Initialisierung

Initialisierung im Code-Block:

```
Strecke(const Ort &Ort1, const Ort &Ort2)
: GraphObj(Ort1)      // Initialisierung des Subobjekts
{
   Endpunkt= Ort2;
}
```

Die Initialisierung mit einer Initialisierungsliste ist generell vorzuziehen, weil das Objekt in *einem* Schritt mit den richtigen Werten initialisiert wird. Die Initialisierungsliste darf enthalten:

- Elemente der Klasse selbst, aber keine vererbten Elemente,
- Konstruktoraufrufe der Oberklassen

8.2 Zugriffsschutz

Abstufung von Zugriffsrechten auf Daten und Elementfunktionen

- `public`

  Elemente und Methoden unterliegen keiner Zugriffsbeschränkung.

- `protected`

  Elemente und Methoden sind in der eigenen und in einer abgeleiteten Klasse zugreifbar, nicht aber in anderen Klassen oder außerhalb der Klasse.

- `private`

  Elemente und Methoden sind ausschließlich innerhalb der Klasse zugreifbar (sowie natürlich für `friend`-Klassen und Funktionen).

Vererbung von Zugriffsrechten

Regeln:

- `private`-Elemente sind in einer abgeleiteten Klasse nicht zugreifbar.
- In allen anderen Fällen gilt, dass das jeweils restriktivere Zugriffsrecht gilt.

Häufigster Fall der `public`-Vererbung:

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

„`struct s {`“ : Abkürzung für „`class s { public:`“

```
class Oberklasse {
     private :                                  // Voreinstellung
        int Oberklasse_priv;
        void private_Funktion_Oberklasse();
     protected:
        int Oberklasse_prot;
     public:
        int Oberklasse_publ;
        void Funktion_Oberklasse();
};
```

```
// Oberklasse wird mit der Zugriffskennung public vererbt
class abgeleiteteKlasse : public Oberklasse {
        int abgeleiteteKlasse_priv;
     public:
        int abgeleiteteKlasse_publ;
        void Funktion_abgeleiteteKlasse() {
              // nicht zugreifbar: Oberklasse_priv = 1;
              // in einer abgeleiteten Klasse zugreifbar:
              Oberklasse_prot = 2;
              // generell zugreifbar
              Oberklasse_publ = 3;
        };
};
```

```
int main() {
        int m;
        abgeleiteteKlasse Objekt;
        m = Objekt.Oberklasse_publ;
        // nicht zugreifbar:
    // m = Objekt.Oberklasse_prot;
        // nicht zugreifbar:
    // m = Objekt.Oberklasse_priv;

        m = Objekt.abgeleiteteKlasse_publ;
        // nicht zugreifbar:
    // m = Objekt.abgeleiteteKlasse_priv;

        Objekt.Funktion_abgeleiteteKlasse();

        // Aufruf geerbter Funktionen
        Objekt.Funktion_Oberklasse();
        // nicht zugreifbar:
    // Objekt.private_Funktion_Oberklasse();
}
```

8.3 Code-Wiederverwendung

In den abgeleiteten Klassen können Methoden der Oberklasse wiederverwendet werden, zum Beispiel die Methoden `Bezugspunkt()` und `Flaeche()`. Beispiele:

```
Strecke S1(Ort(0,0), Ort(100,20));

cout << "\n Fläche der Strecke S1 = "
     << S1.Flaeche()        // geerbte Methode
     << endl;

Ort einOrt(100, 50);
Strecke S2(einOrt, O(99, 263));
```

```
// Fall 1) Bezugspunkt() ist geerbt
cout << "= Entfernung der Bezugspunkte: "
     << Entfernung(S1.Bezugspunkt(), S2.Bezugspunkt())
     << endl;

// Fall 2)
cout << "Entfernung der Strecken S1, S2 = "
     << Entfernung(S1, S2) << endl;
```

1) Aufruf von

```
double Entfernung(const Ort &p1, const Ort &p2);
```

2) ?

8.4 Überschreiben von Funktionen in abgeleiteten Klassen

- bisher:
  Überschreiben von gleichnamigen Funktionen *mit unterschiedlicher Schnittstelle*
- neu:
  Überschreiben von *Elementfunktionen* in abgeleiteten Klassen für Funktionen
  - mit derselben Bedeutung, aber
  - mit unterschiedlichem Mechanismus

Dieselbe Bedeutung erfordert dieselbe Schnittstelle!

```
#ifndef rechteck_h
#define rechteck_h
#include"graphobj.h"

class Rechteck : public GraphObj {
   public:
     Rechteck(const Ort &p1, int h, int b)
     : GraphObj(p1), hoehe(h), breite(b)  {}

     double Flaeche() const {
         return double(hoehe) * breite; // int-Überlauf vermeiden
     }

   private:
     int hoehe, breite;
};
#endif // rechteck_h
```

Am Beispiel der Flächenberechnung mit der Funktion `Flaeche()` sehen wir das Prinzip des Überschreibens:

```
Rechteck R0(Ort(0,0), 20, 50);
cout << "R0.Flaeche = "
     <<  R0.Flaeche() << endl;    // 1000
cout << R0.GraphObj::Flaeche();   // null!
```

8.5 Polymorphismus in abgeleiteten Klassen

Polymorphismus = Vielgestaltigkeit

in der objektorientierten Programmierung:

> Erst zur Laufzeit eines Programms wird die zu dem jeweiligen Objekt passende Realisierung einer Operation ermittelt.

(*dynamisches* (oder spätes) *Binden*)

$\Rightarrow$ *virtuelle Funktionen*

Virtuelle Funktionen haben *dieselbe* Schnittstelle (Signatur und Rückgabetyp) für alle abgeleiteten Klassen (andernfalls bräuchte man sie nicht).

8.5.1 Virtuelle Funktionen

Wirkung der `virtual`-Deklaration einer Funktion:
Zeigern und Referenzen vom Basisklassentyp, die auf ein Objekt eines abgeleiteten Typs zeigen, wird die Information über den Objekttyp mitgegeben.

Vergleich nicht-virtuell / virtuell:

Verhalten einer NICHT-virtuellen Funktion

```
class GraphObj {
      // ....
      double Flaeche() const { return 0;} // nicht v.
};
// ...
class Rechteck : public GraphObj {
     // ....
     double Flaeche() const {        // nicht virtuell
         return double(hoehe) * breite;
     }
};
GraphObj GrO(Ort(20,20));
Rechteck R(Ort(100,100), 20,50); // (x, y), Höhe, Breite
GraphObj *GrOptr;                  // Zeiger auf GraphObj
GrOptr = &GrO;                  // Zeiger auf GrO richten
cout << "GrOptr->Flaeche() ="
     << GrOptr->Flaeche() << endl;    // 0
GrOptr = &R;             // Zeiger auf Rechteck richten
```

```
cout << "GrOptr->Flaeche() ="
     << GrOptr->Flaeche() << endl;    // 0
cout << "R.Flaeche()       ="
     << R.Flaeche()       << endl;    // 1000
```

Verhalten einer virtuellen Funktion

```
class GraphObj {
      // ....
      virtual double v_Flaeche()  {  return 0;}
};
// ...

class Rechteck : public GraphObj {
     // ....
      virtual double v_Flaeche() {
           return double(hoehe) * breite;
      }
};
```

Virtuelle Funktionen sind auch in allen nachfolgend abgeleiteten Klassen virtuell.

Das obige Beispiel wird jetzt mit der Funktion `v_Flaeche()` in genau der gleichen Art und Weise wiederholt:

```
GrOptr = &GrO;                  // Zeiger auf GrO richten
cout << "GrOptr->v_Flaeche() ="
     << GrOptr->v_Flaeche() << endl; // 0

GrOptr = &R;            // Zeiger auf Rechteck richten
cout << "GrOptr->v_Flaeche() ="
     << GrOptr->v_Flaeche() << endl; // 1000 !
```

Wesentliche Merkmale virtueller Funktionen:

- Virtuelle Funktionen dienen zum Überschreiben bei gleicher Signatur und gleichem Rückgabetyp.
- Der Aufruf einer nicht-virtuellen Elementfunktion hängt *vom Typ des Zeigers* ab, über den die Funktion aufgerufen wird, während der Aufruf einer virtuellen Elementfunktion *vom Typ des Objekts* abhängt, auf das der Zeiger verweist.
- Eine als virtuell deklarierte Funktion definiert eine *Schnittstelle* für alle abgeleiteten Klassen, auch wenn diese zum Zeitpunkt der Festlegung der Basisklasse noch unbekannt sind.
- Der vorstehende Punkt gilt auch für Destruktoren. Wenn es überhaupt virtuelle Funktionen in einer Klasse gibt, sollte der Destruktor als `virtual` deklariert werden. Die Definition der Klasse `GraphObj` muss um die Zeile

  ```
  virtual ~GraphObj() {};
  ```

  erweitert werden. Einzelheiten folgen.

Aus diesen Punkten lässt sich eine wichtige Regel ableiten:

> Nicht-virtuelle Funktionen einer Basisklasse sollen *nicht* in abgeleiteten Klassen überschrieben werden!

8.5.2 Abstrakte Klassen

Klassen ohne Objekte? Ja!

In vielen Fällen sollte die Basisklasse einer Hierarchie sehr allgemein sein und Code enthalten, der wahrscheinlich nicht geändert werden muss. Es ist dann oft nicht notwendig oder gewünscht, dass Objekte dieser Klassen angelegt werden.

Diese *abstrakten Klassen* dienen ausschließlich als *Basisklassen*.

Abstrakte Klassen benutzen mindestens eine *rein virtuelle* Funktion.

Objekte und konkrete Implementierungen werden nur zu den abgeleiteten Klassen erzeugt. Durch die rein virtuelle Funktion wird gewährleistet, dass stets die zum Objekttyp passende Methode aufgerufen wird.

Definieren einer abstrakten Klasse heißt also nichts anderes, als ein gemeinsames Protokoll für alle abgeleiteten Klassen zu definieren.

Eine rein virtuelle Funktion wird durch Ergänzung mit „= 0“ deklariert:

```
virtual int rein_virtuelle_func(int) = 0;
```

Es gibt Kreise, Rechtecke, Linien, ... aber:

Es gibt kein graphisches Objekt „an sich“!

→ Kandidat für abstrakte Klasse!

Umwandlung von `GraphObj` in eine abstrakte Klasse:

1. `Flaeche()` als rein-virtuelle Funktion deklarieren (s.u.)
2. Die bisherige Definition der Funktion ersatzlos streichen.

Anderes Beispiel: Erweiterung der Klasse `GraphObj` um eine Funktion `zeichnen()`. Die *Implementation* ist natürlich für Kreise und Rechtecke verschieden, der Aufruf jedoch (=die Schnittstelle) ist stets der gleiche!(Der Einfachheit halber nur Textausgabe.)

```
#ifndef graphobj_h
#define graphobj_h
#include<ort.h>
#include<iostream>

class GraphObj {                          // Version 2
  public:
    GraphObj(const Ort &einOrt)    // allg. Konstruktor
    : Referenzkoordinaten(einOrt) {}

    virtual ~GraphObj() {}    // virtueller Destruktor
```

```
    // Bezugspunkt ermitteln
    Ort Bezugspunkt() const { return Referenzkoordinaten;}
    // alten Bezugspunkt ermitteln und gleichzeitig neuen wählen
    Ort Bezugspunkt(const Ort &nO) {
        Ort temp = Referenzkoordinaten;
        Referenzkoordinaten = nO;
        return temp;
    }
    int X() const { return Referenzkoordinaten.X(); }
    int Y() const { return Referenzkoordinaten.Y(); }
    virtual double Flaeche() const = 0; // rein virtuell
    virtual void zeichnen()  const = 0; // rein virtuell
  private:
    Ort Referenzkoordinaten;
};
// Standardimplementierung der rein virtuellen Methode zeichnen()
inline void GraphObj::zeichnen() const {
     std::cout << "Zeichnen: ";
}
```

Die Entfernung zwischen 2 GraphObj-Objekten ist hier als Entfernung ihrer Bezugspunkte (überladene Funktion) definiert.

```
inline double Entfernung( const GraphObj &g1,
                          const GraphObj &g2) {
     return Entfernung(g1.Bezugspunkt(),
                       g2.Bezugspunkt());
}
#endif    // graphobj_h
```

Die Klassen `Strecke` und `Rechteck` müssen die rein virtuellen Methoden implementieren. Andernfalls wären die Klassen ebenfalls abstrakt, und es könnte keine Instanzen von ihnen geben.

```
#ifndef strecke_h
#define strecke_h
#include"graphobj.h"

class Strecke : public GraphObj {  // erben von GraphObj
   public:
     Strecke(const Ort &Ort1, const Ort &Ort2)
     : GraphObj(Ort1),    // Initialisierung des Subobjekts
       Endpunkt(Ort2) {   // Initialisierung des Attributs
```

```
    }                          // leerer Code-Block
    double Laenge() const {
        return Entfernung(Bezugspunkt(), Endpunkt);
    }
  // Definition der rein virtuellen Methoden
  virtual double Flaeche() const { return 0.0;}
  virtual void zeichnen() const {
      GraphObj::zeichnen();
      std::cout << "Strecke von ";
      anzeigen(Bezugspunkt());
      std::cout << " bis ";
      anzeigen(Endpunkt);
      std::cout << std::endl;
  }
 private:
   Ort Endpunkt;     // zusätzlich: 2. Punkt der Strecke
       // (der erste ist GraphObj::Referenzkoordinaten)
};
#endif // strecke_h
```

```
#ifndef rechteck_h
#define rechteck_h
#include"graphobj.h"

class Rechteck : public GraphObj {
  public:
     Rechteck(const Ort &p1, int h, int b)
     : GraphObj(p1), hoehe(h), breite(b)  {}
    int Hoehe()  const {return hoehe;}  // für Quadrat
    int Breite() const {return breite;}

    // Definition der rein virtuellen Methoden
    virtual double Flaeche() const {
        return double(hoehe)*breite;
    }

    virtual void zeichnen() const {
        GraphObj::zeichnen();
        std::cout << "Rechteck (h x b = "
             << hoehe << " x "
```

```
            << breite
            << ") an der Stelle ";
        anzeigen(Bezugspunkt());
        std::cout << std::endl;
    }
  private:
    int hoehe, breite;
};
#endif // rechteck_h
```

```
#ifndef quadrat_h
#define quadrat_h
#include"rechteck.h"

class Quadrat : public Rechteck {  // siehe Text
  public:
    Quadrat(const Ort &O, int seite)
    : Rechteck(O, seite, seite)  {}

    // Definition der rein virtuellen Methoden
    // Bezugspunkt(), Flaeche(), Hoehe() werden geerbt

    virtual void zeichnen() const {
        GraphObj::zeichnen();
        std::cout << "Quadrat (Seitenlaenge = " << Hoehe()
             << ") an der Stelle ";
        anzeigen(Bezugspunkt());
        std::cout << std::endl;
    }
};
#endif // quadrat_h
```

Das Hauptprogramm ruft die Methoden der graphischen Objekte polymorph auf. Entscheidend ist nicht der (statische) Typ des Zeigers, sondern der polymorphe oder dynamische Typ, dass heißt, der Typ des Objektes, auf das der Zeiger verweist. Dasselbe gilt für Referenzen.

```
#include"strecke.h"
#include"quadrat.h"   // schließt rechteck.h ein

int main() {
    // GraphObj G; // Fehler!
            // Instanzen abstrakter Klassen gibt es nicht

    Rechteck R(Ort(0,0), 20, 50);
    Strecke  S(Ort(1,20), Ort(200,0));
    Quadrat  Q(Ort(122, 99), 88);

    // Feld mit Basisklassenzeigern, initialisiert mit
    // den Adressen der Objekte und 0 als Endekennung
    GraphObj* GraphObjZeiger[] = {&R, &S, &Q, 0};
```

```
    int i = 0;
    while(GraphObjZeiger[i]) // Ausgabe der Fläche aller Objekte
           cout << "Fläche = "
                << GraphObjZeiger[i++]->Flaeche()
                << endl;
    i = 0;
    while(GraphObjZeiger[i])      // Zeichnen aller Objekte
           GraphObjZeiger[i++]->zeichnen();
    cout << "Auch Referenzen sind polymorph:\n";
    GraphObj &R_Ref = R, // Der statische Typ ist derselbe,
             &S_Ref = S,
             &Q_Ref = Q;
    R_Ref.zeichnen();    // der dynamische nicht.
    S_Ref.zeichnen();
    Q_Ref.zeichnen();
}
```

Das Beispiel ist sehr leicht um beliebige graphische Klassen erweiterbar (zum Beispiel Kreis, Ellipse, Polygon...), ohne dass die Anweisung „Zeichnen aller Objekte“ überhaupt geändert werden muss.

8.5.3 Virtuelle Destruktoren

Ein virtueller Destruktor sorgt für die exakte Freigabe dynamischer Objekte, auch wenn der auf sie gerichtete Zeiger vom Datentyp einer Oberklasse ist.

```
class Basis {
   // ....
  public:
    Basis();
    virtual ~Basis();   // virtueller Destruktor!
};

Basis::~Basis() {
   cout << " Basis-Destruktor aufgerufen!\n";
}
```

```
class Abgeleitet : public Basis {
    // ....
  public:
    Abgeleitet();
    ~Abgeleitet();
};

Abgeleitet::~Abgeleitet() {
   cout << " Abgeleitet-Destruktor aufgerufen!\n";
};
```

```
int main () {
    Basis*      pb  = new Basis();
    Abgeleitet* pa  = new Abgeleitet();
    Basis*      pba = new Abgeleitet();
    delete pb;    // ok
    delete pa;    // ok
    delete pba;   // ok nur mit virtuellem Destruktor!
}
```

(Implizite) Aufrufreihenfolge:

```
delete pb;  pb->~Basis();

delete pa;  pa->~Abgeleitet();
            pa->~Basis();

delete pba; pba->~Abgeleitet();  // (*)
            pba->~Basis();
```

Ohne das Schlüsselwort `virtual` würde (*) entfallen → Speicherleiche!

Virtueller Destruktor — wann?

... immer wenn Basisklassenzeiger oder -referenzen auf dynamisch erzeugte Objekte benutzt werden.

8.6 Mehrfachvererbung

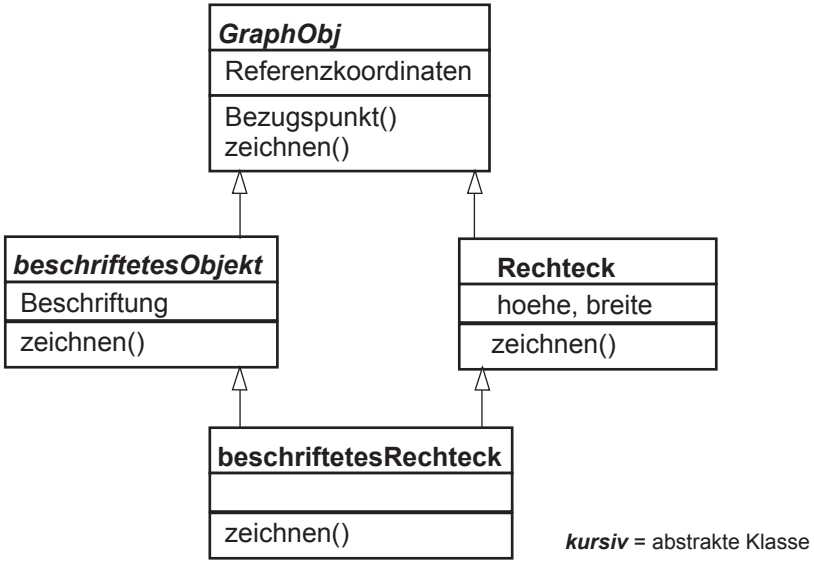


Abb. 8.3: Vererbungsstruktur graphischer Objekte

```
#ifndef beschrif_h
#define beschrif_h
#include"graphobj.h"
#include<string>
class beschriftetesObjekt : public GraphObj { // erben
  public:
    beschriftetesObjekt(const Ort &O, const string &m)
    : GraphObj(O), Beschriftung(m) {}
    virtual void zeichnen() const {
        GraphObj::zeichnen();
        cout << "Beschriftung bei ";
        anzeigen(Bezugspunkt());
        cout <<  Beschriftung << endl;
    }
  private:
    string Beschriftung;
};
#endif // beschrif_h
```

Ein eigener Destruktor ist nicht notwendig.

```
#ifndef bes_r_h
#define bes_r_h
#include"beschrif.h"
#include"rechteck.h"

// Mehrfachvererbung
class beschriftetesRechteck
   : public beschriftetesObjekt, public Rechteck {
  public:
    beschriftetesRechteck(const Ort &O, int h, int b,
                          const string &m)
    : beschriftetesObjekt(O, m), Rechteck(O, h, b) {
    }

    // Definition der rein virtuellen Methoden
    virtual double Flaeche() const {
        // Definition ist notwendig, damit die Klasse
        // nicht abstrakt ist (durch Vererbung über
        // beschriftetesObjekt und GraphObj)
        return Rechteck::Flaeche();
    }
```

```
    virtual void zeichnen()  const {
        Rechteck::zeichnen();
        beschriftetesObjekt::zeichnen();
    }
};
#endif // bes_r_h
```

mögliches Hauptprogramm:

```
// Auszug aus main.cpp:

    Rechteck R(Ort(0,0), 20, 50);

    beschriftetesRechteck BR(Ort(1,20), 60, 60,
                    string("Mehrfachvererbung"));
    R.zeichnen();
    BR.zeichnen();

    beschriftetesRechteck *zBR =
                 new beschriftetesRechteck(
                     Ort(100,0), 20, 80,
                     string("dynamisches Rechteck"));

    zBR->zeichnen();
```

8.6.1 Namenskonflikte

```
cout << "Rechteck-Position: ";
anzeigen(R.Bezugspunkt());
cout << "beschriftetes-Rechteck-Position: "
anzeigen(BR.Bezugspunkt());        // Compiler-Fehlermeldung!
```

Zweideutigkeit beseitigen:

```
anzeigen(BR.Rechteck::Bezugspunkt());    // eindeutig
```

Durch verschiedene Bezugspunkte im Konstruktor wird nachgewiesen, dass `beschriftetesRechteck` *zwei* `GraphObj`-Objekte besitzt:

```
// absichtlich veränderter Konstruktor
    beschriftetesRechteck(const Ort &O, int h, int b,
                          const string &m)
    : beschriftetesObjekt(O, m),
      Rechteck(Ort(100,100), h, b) {          // !!
    }
// jetzt verschiedene Werte:
anzeigen(BR.Rechteck::Bezugspunkt());
anzeigen(BR.beschriftetesObjekt::Bezugspunkt());
```

Weil zwei Subobjekte vom Typ `GraphObj` vorliegen, ist wegen der Nicht-Eindeutigkeit die Zuweisung eines Zeigers nicht möglich:

```
int main() {
    Rechteck R1(Ort(0,0), 20, 50);
    Rechteck R2(Ort(0,100), 10, 40);
    beschriftetesRechteck RB(Ort(1,20), 60, 60,
                  string("Mehrfachvererbung"));

    // Feld mit Basisklassenzeigern, initialisiert mit
    // den Adressen der Objekte, 0 als Endekennung
    GraphObj* GraphObjZeiger[] = {&R1, &R2, 0};    // ok

    // Fehler:
    // GraphObj* GraphObjZeiger[]={&R1,&R2,&RB,0};

    // Zeichnen aller Objekte
    int i = 0;
    while(GraphObjZeiger[i])
          GraphObjZeiger[i++]->zeichnen();
}
```

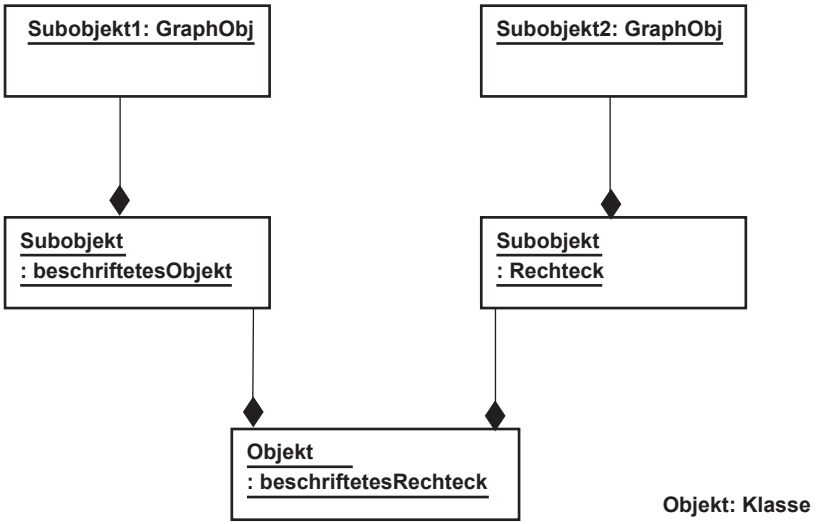


Abb. 8.4: *enthält*-Beziehungen bei nicht-virtueller Vererbung

Auf welches Subobjekt soll der Zeiger `GraphObjZeiger[2]` verweisen?

8.6.2 Virtuelle Basisklassen

Vermeiden duplizierter Subobjekte vom Basisklassentyp.
Notwendige Änderungen:

```
// rechteck.h
class Rechteck : virtual public GraphObj {
    // ... Rest wie vorher
};

// beschrif.h
class beschriftetesObjekt : virtual public GraphObj {
    // ... Rest wie vorher
};

// bes_r.h
class beschriftetesRechteck
    : public beschriftetesObjekt, public Rechteck {
};
```

Änderung im Konstruktor:

```
beschriftetesRechteck(const Ort &O, int h, int b, string &m)
   : GraphObj(O),
      beschriftetesObjekt(O, m),
      Rechteck(O, h, b) {
   }
}
```

Mit diesen Änderungen sind Aufrufe wie

```
cout << "beschriftetesRechteck-Position: ";
anzeigen(BR.Bezugspunkt());
```

möglich und unproblematisch, weil nun genau *ein* Basisklassenobjekt für BR existiert.

```
int main() {                                    // geändert
    Rechteck R1(Ort(0,0), 20, 50);
    Rechteck R2(Ort(0,100), 10, 40);
    beschriftetesRechteck RB(Ort(1,20), 60, 60,
                  string("virtuelle Mehrfachvererbung"));

    // Feld mit Basisklassenzeigern, initialisiert mit
    // den Adressen der Objekte, 0 als Endekennung
    // jetzt ok!
    GraphObj* GraphObjZeiger[] = {&R1, &R2, &RB, 0};

    // Zeichnen aller Objekte
    int i = 0;
    while(GraphObjZeiger[i])
          GraphObjZeiger[i++]->zeichnen();
}
```

8.6.3 Virtuelle Basisklassen und Initialisierung

- In einer Klassenhierarchie kann es mehrere Initialisierer für eine Basisklasse geben.
- Falls wir jedoch virtuelle Basisklassen haben, wird *nur ein* Subobjekt dieser Basisklasse in Objekten einer abgeleiteten Klasse angelegt.
- Welcher von diesen soll entscheiden?

Antwort:

Basisklasseninitialisierer, *der bei dem Konstruktor eines vollständigen Objekts angegeben ist*.

Falls dieser nicht existiert:
Standardkonstruktor der virtuellen Basisklasse

```
class Basis {
   public:
     Basis() { cout << "Basis-Std.-Konstruktor\n"; }
     Basis(char* a)  { cout << a << endl; }
};
class Links : virtual public Basis {
   public:
      Links(char* a)
      // : Basis(a) // siehe unten
      { }
};
class Rechts : virtual public Basis {
   public:
      Rechts(char* a) : Basis(a) {}
};
```

```
class Unten: public Links, public Rechts {
   public:
      Unten(char* a) :
      // Basis(a), // siehe unten
      Links(a), Rechts(a)  {}
};

int main() {
   Unten x("Unten");  // Basis-Std.-Konstruktor
   Links li("Links"); // Basis-Std.-Konstruktor
}
```

```
class Basis {
   public:
     Basis() { cout << "Basis-Std.-Konstruktor\n"; }
     Basis(char* a)  { cout << a << endl; }
};
class Links : virtual public Basis {
   public:
      Links(char* a)
      : Basis(a)              // vgl. oben
      { }
};
class Rechts : virtual public Basis {
   public:
      Rechts(char* a) : Basis(a) {}
};
```

```
class Unten: public Links, public Rechts {
   public:
      Unten(char* a) :
         Basis(a),              // vgl. oben
      Links(a), Rechts(a)  {}
};

int main() {
  Unten x("Unten");  // Unten
  Links li("Links"); // Links
}
```

`Rechts::Basis(a)` wird weiterhin ignoriert.

8.7 Vererbung und andere Beziehungen

8.7.1 Vererbung

Problem: Ist ein Quadrat ein Rechteck?

Eine abgeleitete Klasse kann als Subtyp der Oberklasse aufgefasst werden.

Ein Objekt einer abgeleiteten Klasse kann damit stets an die Stelle eines Objekts der Oberklasse treten – es sind ja alle Methoden der Oberklasse vorhanden, wenn auch möglicherweise überschrieben (Liskov'sches Substitutionsprinzip)

```
class Quadrat  : public Rechteck {
   // ...
};
```

Klasse `Rechteck` mit Methode zum Ändern der Seiten:

```
class Rechteck {
    public:
       virtual void HoeheAendern(int n) {hoehe  = n;}
       virtual void BreiteAendern(int n){breite = n;}
       // ...
    private:
       int hoehe;
       int breite;
};
```

→ kann aus OO-Sicht im Quadrat nicht erlaubt sein!

Einhaltung des „Erbvertrags“:

```
class Quadrat : public Rechteck {    // empfehlenswert?
    public:
       virtual void HoeheAendern(int neu) {
           Rechteck::HoeheAendern(neu);
           Rechteck::BreiteAendern(neu);
       }
       virtual void BreiteAendern(int neu)  {
           HoeheAendern(neu);
       }
       // ... (Konstruktor usw.)
};
```

Ohne Vererbung (Delegation):

```
class Quadrat {                                  // empfehlenswert?
    public:
       Quadrat(const Ort& O, int Seite)
        // privates Rechteck initialisieren:
       : R(O, Seite, Seite) {
       }

       virtual void SeiteAendern(int neu) {
           R.HoeheAendern(neu);
           R.BreiteAendern(neu);
       }

       // ... viele weitere Funktionen, die Methoden der
       // Klasse Rechteck benutzen
     private:
       Rechteck R;
};
```

Vergleichbare Problemstellungen:

- Ist ein Kreis eine Ellipse?
- Ist eine sortierte Liste eine Liste?

8.7.2 Der Teil und das Ganze

„Teil-Ganzes“-Beziehung heißt auch „Aggregation“

```
class Tastatur {
      public:
        // ...
      private:
        // aggregierte Objekte:
        Gehaeuse einGehaeuse;
        Kabel einKabel;
        Taste Tastenfeld[102];
};
```

8.7.3 Assoziation

Eine Assoziation beschreibt eine Verbindung von einem Objekt zu einem oder mehreren anderen. Gemeinsamkeiten können vorhanden sein, spielen aber keine Rolle. Einige Programmfragmente dienen als Beispiel:

```
class Person;    // Vorwärtsdeklaration

class Firma {
   public:
    void einstellen(Person*);
    // ...
   privat:
    Liste<Person*> Mitarbeiter;
    // ...
};

class Person {
   public:
    // ... Konstruktoren usw.

    void lerntKennen(Person& P) {
```

```
    Bekanntenkreis.Hinzufuegen(&P);
  }

  String wieHeisstDu()  { return Name;}

  /* Die Assoziation kann kurzfristig durch die Parameter-
     übergabe eines Objekts begründet werden, sie kann aber
     auch länger andauern, indem sie im Objekt registriert
     wird, wie die folgende Methode zeigt.
   */
  void findetArbeitBei(Firma& F) {
       Arbeitgeber = &F;
       F.einstellen(this);
  }

  void kuendigt() {
       Arbeitgeber->informieren(this);
       Arbeitgeber = 0;
  }

private:
  String Name;
  Firma *Arbeitgeber;
```

```
    Liste<Person*> Bekanntenkreis;
};
// Anwendung
Firma IS_Bremen_GmbH;
Person Walter, Thomas, Sonja;
Thomas.kuendigt();
Thomas.findetArbeitBei(IS_Bremen_GmbH);
Walter.lerntKennen(Sonja);
```

8.7.4 „Benutzt“-Beziehung

Ein Objekt benutzt ein anderes Objekt, um seine Aufgabe zu erfüllen.

Ein Heizregler benutzt einen Temperaturmessfühler. Wie im vorhergehenden Beispiel sind Zeiger zur Formulierung in C++ geeignet:

```
class Heizregler {
    public:
      // ... Konstruktoren

      bool warmGenug() {
         return (dasThermometer->wieWarm() > 20); // Grad
      }
      // ...

    private:
      Temperaturfuehler *dasThermometer;
};
```

Alternative:

Modellierung mit Referenzen (können nie undefiniert sein)

9. Überladen von Operatoren

Um wieviel einfacher ist es, in einem Programm mit Matrizen `a`, `b` und `c`

```
matrix a = b + c/2;
```

zu schreiben anstelle verschachtelter Schleifen!

Überladen = den Operatoren zusätzliche, problemabhängige Bedeutungen geben

bekanntes Beispiel: << Bitshift, aber auch Ausgabeoperator

Das Überladen von Operatoren funktioniert ähnlich wie das Überladen von Funktionen.

Syntax der Operatordefinition:

ReturnDatentyp `operator`$\otimes$`(`*Argumentliste*`) { ... }`

für eine `globale`-Funktion oder

ReturnDatentyp Klassenname`::operator`⊗`(`*Argument*`) { ... }`

für eine Elementfunktion.

⊗ steht für eins der möglichen C++-Operatorsymbole.

Operator = Funktion mit besonderem Namen!

Element-Funktion	Syntax	Ersetzung durch
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()`

T ist Platzhalter für einen Datentyp

1. Der Aufruf wird entsprechend der Tabelle in einen Funktionsaufruf umgewandelt. Es macht keinen Unterschied, wenn man die Umwandlung selbst schon vornimmt und zum Beispiel
   `s = operator+(x,y)` schreibt statt `s = x + y`.
2. Der *Funktionsname* einer Operatorfunktion besteht aus dem Schlüsselwort `operator` und dem angehängten Operatorzeichen.
3. Es können die üblichen C++ -Operatoren wie =, ==, += usw. überladen werden, nicht jedoch `.  .*  ::  ?:` und andere Zeichen wie $ usw.
4. Eine Definition von neuen Operatoren ist *nicht* möglich, auch nicht durch Kombination von Zeichen.
5. Die Operatoren `new` und `delete` sowie `new []` und `delete []` können überladen werden.
6. Die vorgegebenen Vorrangregeln können nicht verändert werden.
7. Wenigstens ein Argument der Operatorfunktion muss ein `class`-Objekt sein, oder die Operatorfunktion muss eine Elementfunktion sein.

9.1 Rationale Zahlen - noch einmal

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

9.1.1 Arithmetische Operatoren

Welche Operatorform ist zu wählen?

Addition ist *binär*
→ nur Zeilen 1 und 4 der Tabelle

drei Fälle für gemischte Datentypen:

1. Addition zweier rationaler Zahlen: $z = x + y$
   Umformulierung entsprechend der Tabelle:
   A) `z = x.operator+(y)`
   B) `z = operator+(x,y)`

2. Addition einer rationalen und einer ganzen Zahl, zum Beispiel
   $z = x + 3$
   Umformulierung:
   A) `z = x.operator+(3)`
   B) `z = operator+(x,3)`

3. Addition einer ganzen und einer rationalen Zahl, zum Beispiel

   $z = 3 + y$

   Umformulierung:

   A) `z = 3.operator+(y)` ???

   B) `z = operator+(3,y)`

Vergleich ergibt Lösung B) als einzig mögliche, wenn die Symmetrie erhalten bleiben soll:

```
rational operator+(const rational&, const rational&);
```

Ohne Typumwandlungskonstruktor

Um gemischte Datentypen zu erlauben, fügen wir zwei Deklarationen hinzu, mit denen Operationen wie $z = x + 3$ oder $z = 3 + y$ bearbeitet werden können:

```
rational operator+(long, const rational&);
rational operator+(const rational&, long);
```

Die Implementierung könnte wie folgt aussehen:

```
rational operator+( const rational& a,
                    const rational& b) {
     rational t;
     t.zaehler = a.Zaehler()*b.Nenner() + b.Zaehler()*a.Nenner();
     t.nenner  = a.Nenner()*b.Nenner();
     t.kuerzen();
     return t;
}
```

```
rational operator+(long a, const rational& b) {
     rational t;
     t.definiere(a, 1); // t wird zu (a/1)
     return t+b;        // Aufruf von +(rational, rational)
}

rational operator+(const rational& a, long b) {
    return b+a;        // Aufruf von +(long, rational)
                       // durch vertauschte Reihenfolge
}
```

Mit Typumwandlungskonstruktor

Wenn ein Typumwandlungskonstruktor vorhanden ist, könnten alle Prototypen und Implementationen des Operators entfallen, die einen `long`-Parameter enthalten.

Implementierung mit Ausnutzen der Kurzformoperatoren

Zunächst wird += implementiert:

Deklaration als Elementfunktion:

```
// in class rational nachtragen:
rational& operator+=(const rational&);
```

Definition:

```
rational& rational::operator+=(const rational& b) {
     zaehler = zaehler*b.nenner + b.zaehler*nenner;
     nenner  = nenner*b.nenner;
     kuerzen();
     return *this;
}
```

`operator+()` kann dann mit Hilfe von `+=` implementiert werden:

```
rational operator+( const rational& a,
                    const rational& b) {
    rational temp = a;
    return temp  += b;       // temp.operator+=(b)
}
```

oder noch kürzer

```
rational operator+( const rational& a,
                    const rational& b) {
    return rational(a) += b;
}
```

9.1.2 Ausgabeoperator <<

```
rational r;
cout << r;
```

Interpretation als
a) `cout.operator<<(r);`, oder als
b) `operator<<(cout,r);` ???

```
ostream& operator<<(ostream& s, const rational&);
```

Referenz als Rückgabetyp erlaubt Verkettung:

```
cout << "r = " << r << "r1= " << r1;
```

ist identisch mit

```
(((cout << "r = ") << r) << "r1= ") << r1;
```

Implementierung z. B.:

```
ostream& operator<<(std::ostream& ausgabe,
                    const rational& r) {
     ausgabe << r.Zaehler() << "/" << r.Nenner();
     return ausgabe;
}
```

9.2 Eine Klasse für Vektoren

(ähnlich zur Standardbibliothek)

Unterschiede zu einem normalen Array:

- Der Arrayzugriff über den Index-Operator [ ] ist *sicher* (im GGs. zur Standardbibliothek, vgl at()). Eine Indexüberschreitung wird zur Laufzeit als Fehler gemeldet, es wird keine undefinierte Adresse angesprungen.
- Es wird ein Zuweisungsoperator definiert, so dass `v1=v2;` geschrieben werden kann mit dem Ergebnis `v1[i]==v2[i]` für alle `i` im Bereich 0 bis (Länge von `v2`)-1.
- Die Klasse kann leicht um weitere Funktionen erweitert werden wie Bildung eines Skalarprodukts, Finden des größten Elements, Addition zweier Vektoren und andere.

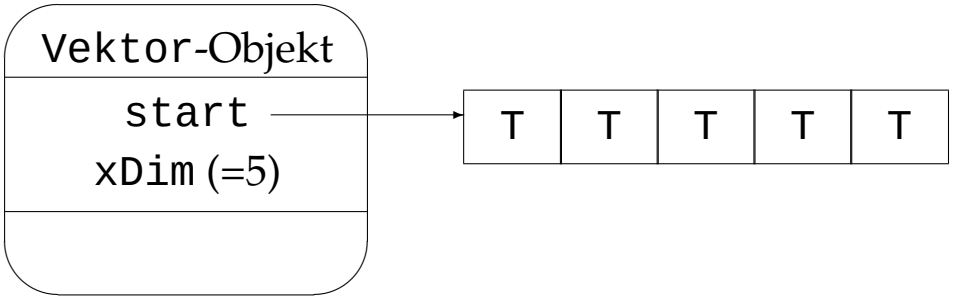


Abb. 9.1: Ein Objekt der Klasse `Vector`

```
// dynamische Vektor-Klasse
#ifndef vektor_t
#define vektor_t
#include<cassert>

template<class T>
class Vektor {
  public:
    Vektor(int x = 0);                  // Standardkonstruktor
    Vektor(const Vektor<T>& v);   // Kopierkonstruktor

    virtual ~Vektor() { delete [] start;}
```

```
int  Groesse() const {return xDim;}

void GroesseAendern(int);     // dynamisch ändern

// Indexoperator inline
T& operator[](int index) {
     assert(index >= 0  &&  index < xDim);
     return start[index];
}

// Indexoperator für Vektoren mit unveränderlichen Elementen
const T&  operator[](int index) const {
     assert(index >= 0  &&  index < xDim);
     return start[index];
}
```

```
    // Zuweisungsoperator
    Vektor<T>& operator=(const Vektor<T>&);

    // Initialisierung des Vektors
    void init(const T&);

    // Zeiger auf Anfang und Position nach dem Ende
    // für nicht-konstante und konstante Vektoren
    T* begin() { return start;}
    T* end()   { return start + xDim;}
    const T* begin() const { return start;}
    const T* end()   const { return start + xDim;}

  protected:  // für Zugriff abgeleiteter Klassen
    int  xDim;                  // Anzahl der Datenobjekte
    T   *start;                 // Zeiger auf Datenobjekte
};
```

noch fehlende Implementierungen:

```
// Konstruktor
template<class T>
inline Vektor<T>::Vektor(int x = 0)
: xDim(x), start(new T[x]) {
}
// Kopierkonstruktor
template<class T>
inline Vektor<T>::Vektor(const Vektor<T> &v) {
    xDim  = v.xDim;
    start = new T[xDim];
    for(int i = 0; i < xDim; ++i)
        start[i] = v.start[i];
}
```

```
template<class T>
inline void Vektor<T>::init(const T& wert) {
     for(int i = 0; i < xDim; ++i)
         start[i] = wert;
}
```

9.2.1 Index-Operator [ ]

Die Deklaration des Indexoperators zeigt, dass nicht ein Objekt, sondern eine *Referenz* auf ein Objekt des Datentyps `T` zurückgegeben wird. Warum?

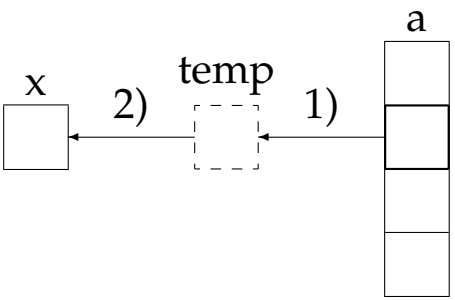
Vergleich zweier Möglichkeiten:

1. Rückgabe per Wert

Die Deklaration würde dementsprechend lauten:

```
T operator[](int index);
```

a) x=a[1];



b) a[1]=x;

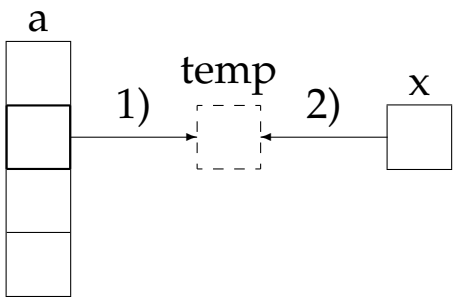


Abb. 9.2:) Zuweisung von x

Zuweisung von Vektorelementen per Wert. Reihenfolge:
1. Erzeugen einer temporären Kopie
2. Zuweisung von x

2. Rückgabe per Referenz

Die Deklaration enthält nun den richtigen Rückgabetyp:

```
T& operator[](int index);
```

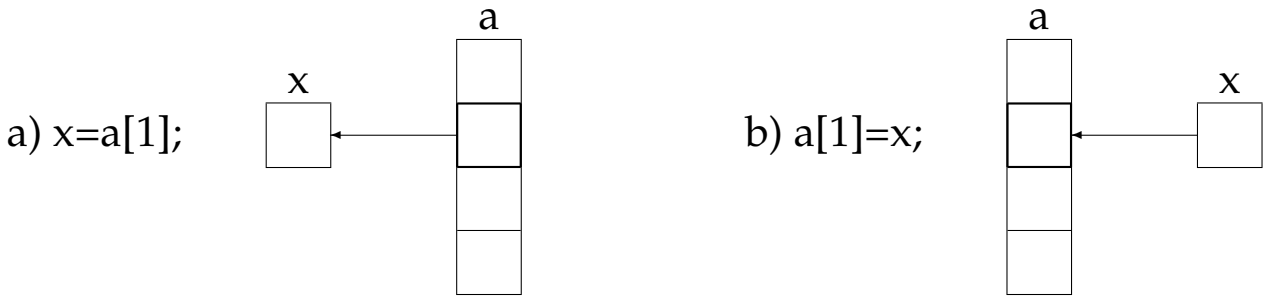


Abb. 9.3: Zuweisung von Vektorelementen per Referenz

Möglicher Einsatz:

```
Vektor<double> v(5);    // Vektor mit 5 Elementen

// Element als Linkswert
v[0] = 1.00; v[1] = 1.234;  v[2] = 2.22;
// Element rechts
double x = v[2];

// Test auf Indexüberschreitung
v[99] = 400.2;
x = v[100];

// Definition und Initialisierung eines anderen Vektors
Vektor<double> vc=v;     // mit Kopierkonstruktor

// Element links und rechts
v[0] = vc[1];
```

Die Schreibweise unterscheidet sich in nichts von der Benutzung eines normalen Arrays!

9.2.2 Zuweisungsoperator

Ein besonderer Zuweisungsoperator ist besonders bei dynamischen Anteilen notwendig.

Wenn ein Objekt A auf ein Objekt B zugewiesen werden soll, d.h. B = A, ist folgende Reihenfolge einzuhalten:

- Reservieren von Speicher für B in der Größe von Objekt A
- Inhalt A nach B kopieren
- Freigabe des von B belegten Speichers
- Aktualisieren der Verwaltungsinformationen

Sonderfall: Zuweisung *auf sich selbst* (unwahrscheinlich, aber erlaubt)

```
template<class T>
inline Vektor<T> &Vektor<T>::operator=(const Vektor<T>& v) {
    if (this != &v) {     // Zuweisung identischer Objekte vermeiden
       T * temp = new T[v.xDim];    // neuen Platz beschaffen
       for (int i = 0; i < v.xDim; ++i)
           temp[i] = v.start[i];
       delete[] start;          // Speicherplatz freigeben
       xDim  = v.xDim;          // Verwaltungsinformation aktualisieren
       start = temp;
    }
    return *this;
}
```

Dynamisches Ändern der Vektorgröße

Nachdem die Wirkungsweisen von Kopierkonstruktor und Zuweisungsoperator bekannt sind, macht es nun keine Schwierigkeiten, eine Methode zum Ändern der Größe eines Vektors zu schreiben:

```
template<class T>
void Vektor<T>::GroesseAendern(int neueGroesse) {
    // neuen Speicherplatz besorgen
    T *pTemp = new T[neueGroesse];
    // die richtige Anzahl von Elementen kopieren
    int kleinereZahl =
              (neueGroesse > xDim) ? xDim : neueGroesse;
    for(int i = 0; i < kleinereZahl; ++i)
        pTemp[i] = start[i];
    delete [] start;    // alten Speicherplatz freigeben
    // Verwaltungsdaten aktualisieren
    start = pTemp;
    xDim  = neueGroesse;
}
#endif // vektor_t
```

9.2.3 Mathematische Vektoren

`Vektor` ist für beliebige Objekte. `mathVektor` ist maßgeschneidert für Objekte der Datentypen `int`, `float`, `complex`, `rational`... (Realisieren mathematischer Operationen, z. B. `*=`)

Vererbung: ein `mathVektor` *ist ein* `Vektor`

```
#ifndef mvektor_t
#define mvektor_t
#include<iostream>
#include"vektor.t"

template<class T>
class mathVektor : public Vektor<T> {
   public:
      mathVektor(int = 0);

      // Operatoren
      mathVektor& operator*=(T);
      // weitere Operatoren und Funktionen ...
};
```

Die Implementation des Konstruktors ist entsprechend einfach. Der Konstruktor initialisiert nur ein Basisklassensubobjekt vom Typ `Vektor` in der richtigen Größe, sonst ist nichts weiter zu tun:

```
template<class T>
mathVektor<T>::mathVektor(int x)
: Vektor<T>(x) {
}
```

Ein eigener Kopierkonstruktor, ein eigener Zuweisungsoperator und ein eigener Destruktor sind nicht notwendig (warum?). Der Indexoperator wird von der Basisklasse geerbt.

9.2.4 Multiplikations-Operator

Anwendung `v1 *= 1.23;`

```
template<class T>
mathVektor<T>& mathVektor<T>::operator*=(T zahl) {
    for(int i = 0; i < xDim; ++i)  // privat?
          // elementweise Multiplikation:
        start[i] = start[i]*zahl;
    return *this;
}
```

`xDim` und `start` könnten auf den ersten Blick als private Attribute der Oberklasse `Vektor` entworfen worden und deswegen nicht zugreifbar sein!

Abhilfe:

1. In der Oberklasse `private` durch `protected` ersetzen. Dann haben alle von `Vektor` abgeleiteten Klassen Zugriff auf diese Variablen.

2. Die zweite Möglichkeit besteht darin, nur geerbte, öffentliche Methoden zu verwenden:

```
// Version 2
template<class T>
mathVektor<T>& mathVektor<T>::operator*=(T zahl)  {
    for(int i = 0; i < Groesse(); ++i)
        operator[](i) = operator[](i)*zahl;
    return *this;
}
```

Diese Möglichkeit ist trotz inlining etwas langsamer, weil der Index geprüft wird. Bei 1. geschieht der Zugriff direkt über `start`.

Jetzt kann die Multiplikation eines beliebig großen Vektors sehr einfach geschrieben werden, zum Beispiel:

```
mathVektor<double> v1(100), v2(200);
//...
// alle Elemente von v1 multiplizieren
v1 *= 1.234567;
```

Aber was ist mit `v1 = v2 * 1.234567;` oder

```
v1 = 1.234567 * v2;
```
?

```
// * Operator für den Fall v1 = zahl*v2;
template<class T>
mathVektor<T> operator*(T zahl, const mathVektor<T> &v) {
    mathVektor<T> temp = v;
    return temp *= zahl;  // Aufruf von operator*=()
}
// * Operator für den Fall v1 = v2*zahl;
// (vertauschte Reihenfolge der Argumente)
template<class T>
mathVektor<T> operator*(const mathVektor<T> &v, T zahl) {
    mathVektor<T> temp = v;
    return temp *= zahl;  // Aufruf von operator*=()
}
```

Die Länge eines Vektors kann sich durch Zuweisung ändern:

```
v1 = 1.234567 * v2;
```

(9.3 Zuweisungsoperator und Vererbung wird übersprungen)

9.4 Inkrement-Operator ++

Präfix- oder Postfix-Form. Beispiel: Datum-Klasse

```
class Datum {
  public:
    Datum();
    Datum(int t, int m, int j);
    void setzeDatum(int t, int m, int j);
    void aktuell();                  // Systemdatum setzen
    bool Schaltjahr() const;

    Datum& operator++();             // Tag hochzählen, präfix
    const Datum operator++(int);// .... und postfix
    int Tag()   const;               // lesende Methoden
    int Monat() const;
    int Jahr()  const;
  private:
     int tag, monat, jahr;
};
```

Um bereits *vor* der Konstruktion eines Datums Tag, Monat und Jahr überprüfen zu können, zum Beispiel nach einer Dialogeingabe, wird eine globale Funktion zur Überprüfung bereitgestellt.

```
// global
bool korrektesDatum(int t, int m, int j)
```

Die Implementierung der Operatormethoden ist am Ende zu finden:

```
#include"datum.h"
#include<ctime>
#include<cstdlib>
#include<cassert>

// Der Standardkonstruktor initialisiert das
// Datum mit dem Systemdatum
Datum::Datum()  { aktuell();}

// Um Code-Duplikation zu vermeiden, wird eine
// Methode aufgerufen
Datum::Datum(int t, int m, int j) { setzeDatum(t, m, j);}

int Datum::Tag()   const { return tag;  }
```

```
int Datum::Monat() const { return monat;}
int Datum::Jahr()  const { return jahr; }

void Datum::setzeDatum(int t, int m, int j) {
    tag = t;
    monat = m;
    jahr = j;
    // Damit nur korrekte Datum-Objekte möglich sind:
    assert(korrektesDatum(tag, monat, jahr));
}
```

```
void Datum::aktuell() {    // Systemdatum eintragen
    // time_t, time(), tm, localtime() sind im
    // Header <ctime> deklariert
    time_t now = time(NULL);
    tm *z = localtime(&now);
           // z ist ein Zeiger auf eine vordefinierte
           // Struktur des Typs tm
    jahr  = z->tm_year + 1900;
    monat = z->tm_mon+1;      // localtime liefert 0..11
    tag   = z->tm_mday;
}
bool Datum::Schaltjahr() const {
     return  jahr % 4==0 && jahr % 100 || jahr % 400==0;
}
```

Tag weiterschalten mit Operator ++:

```
Datum& Datum::operator++() {   // Präfix (kein int-Argument)
    do {
        tag++;
        if(tag > 31) {
            tag = 1;
            monat++;
            if(monat > 12) {
                monat = 1;
                jahr++;
            }
        }
    } while(!korrektesDatum(tag, monat, jahr));
    return *this;
}
```

Der Postfix-Inkrementierungsoperator folgt einem *allgemeinen* Muster:

```
const Datum Datum::operator++(int) {
   Datum temp = *this;
   ++*this;                    // Präfix ++ aufrufen
   return temp;
}
```

Hinweis: `++*this;` ist dasselbe wie `operator++();`

Der Aufruf in einem Programm wird wie folgt interpretiert:

```
++dat1;          Aufruf wie dat1.operator++();
dat1++;          Aufruf wie dat1.operator++(0);
```

9.5 Typumwandlungsoperator

bekannte Beispiele:

```
float f;
int i = 1234;
f = i;                              // implizit
f = static_cast<float>(i);          // explizit
```

sowie Typumwandlung mit einem Konstruktor

hier: Definition von Typumwandlungsoperatoren für selbstgeschriebene Klassen, um ein Objekt der Klasse in einen anderen Datentyp abzubilden.

Typumwandlungsoperatoren sind Elementfunktionen und haben weder eine Parameterliste noch einen Rückgabetyp.

Struktur:

```
operator Datentyp ();
```

Beispiel: `Datum`-Objekt in einen String umwandeln:

```
// datum.h
#include<string>

class Datum {
    // .... wie vorher
    operator string();
};
```

```
// in datum.cpp:
Datum::operator string() {
    string temp("tt.mm.jjjj");
    temp[0] = char(tag/10)+'0';
    temp[1] = char(tag%10)+'0';
    temp[3] = char(monat/10)+'0';
    temp[4] = char(monat%10)+'0';

    int pos = 9;                    // letzte Jahresziffer
    int j = jahr;

    while(j > 0)     {
       temp[pos] = j % 10 + '0';   // letzte Ziffer
       j = j/10;                   // letzte Ziffer abtrennen
       --pos;
    }
    return temp;
}
```

Es ist sowohl die implizite als auch die explizite Umwandlung möglich, wie die folgenden Zeilen zeigen.

```
Datum d;
string t = d;                 // implizit oder
cout << t << endl;                    // Ausgabe
string t = (string)d;         // explizit
cout << t << endl;                    // Ausgabe
```

Empfehlung: Um die Typprüfung des Compilers nicht unnötig zu umgehen, sollte man Typumwandlungsoperatoren nur in begründeten Fällen einsetzen!

Alternativ bieten sich Methoden an. In diesem Beispiel wäre eine Methode namens `toString()` anstelle des Typumwandlungsoperators die bessere Lösung, weil dann eine implizite Typumwandlung von vornherein unmöglich ist. Andere Möglichkeit: `explicit` (vgl. 5.3.4)

10. Fehlerbehandlung

Übliche Strategien:

1. Programmabbruch
2. Parameter an Aufrufer übergeben

```
ergebnis1 = f(par1, par2, fehler);
if(fehler) {
    switch (fehler) {
       case 1: ergebnis1 = -1; break;
       case 2: ergebnis1 =  0; break;
       default: cout << "Fehler in f()!";
                exit(-2);
    }
}
// ...
ergebnis2 = g(par3, fehler);
if(fehler)
   exit(-3);
// ... usw.
```

3. `errno` setzen
4. Fehlerbehandlungsfunktion aufrufen
5. nichts tun :-(
6. Ausnahmebehandlung!

10.1 Ausnahmebehandlung

Trennung von Fehlererkennung und -behandlung, insofern vergleichbar mit 4. oben

Fehlererkennung von

- Division durch Null,
- Bereichsüberschreitung eines Arrays,
- Syntaxfehler bei Eingaben,
- Zugriff auf eine nichtgeöffnete Datei,
- Fehlschlag der Speicherbeschaffung oder
- Nichteinhaltung der Vorbedingung einer Methode

Wege zur *Fehlerbehandlung*:

- Den Fehler beheben und den Programmlauf fortsetzen oder, falls das nicht gelingt, das Programms mit einer aussagefähigen Meldung abbrechen *oder*
- den Fehler an das aufrufende Programm melden, das ihn dann ebenfalls auf eine dieser zwei Arten bearbeiten muss.

Alternativ zur 4. Strategie stellt C++ die Ausnahmebehandlung (exception handling) bereit. Ablauf:

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.

Syntaktische Struktur:

```
try {
      func();
      /* Falls die Funktion func() einen Fehler entdeckt, wirft
         sie eine Ausnahme aus (throw), wobei ein Objekt über-
         geben werden kann, um die geeignete Fehlerbehand-
         lung anzustoßen. Oder es wird in den Anweisungen ein
         Fehler festgestellt, der zum Auswerfen einer Exception
         führt, etwa so:
       */
      // weitere Anweisungen ...
      if(EsIstEinFehlerPassiert)
         throw Exception();
}

catch(Datentyp1 e) {    // Syntax für Grunddatentypen, z.B. const char*
      // durch ausgeworfenes Objekt e ausgewählte Fehlerbehandlung
      // ...
}

catch(const Datentyp2& e) {    // Klassenobjekte per Referenz übergeben
      // durch ausgeworfenes Objekt e ausgewählte Fehlerbehandlung
      // ...
}
```

```
// gegebenenfalls weitere catch-Blöcke
// ...
```

Eine misslungene Fehlerbehandlung sollte „nach oben“ gemeldet werden!

```
catch(...) {    // Ellipse ... für nicht spezifizierte Fehler
     cerr << "nicht behebbarer Fehler!" << endl;
     throw;    // Weitergabe an nächsthöhere Instanz
}

// Hier Fortsetzung des Programms nach Fehlerbearbeitung!
// ...
```

10.1.1 Exception-Spezifikation in Deklarationen

Um Ausnahmen, die eine Funktion auswerfen kann, in den Schnittstellen bekannt zu machen, können sie im Funktionsprototyp als sog. Exception-Spezifikation deklariert werden. Dabei sind drei Fälle zu unterscheiden:

1. `void Zahlen_lesen_und_ausgeben();`
   kann beliebige Ausnahmen auswerfen.
2. `void Zahlen_lesen_und_ausgeben() throw()`
   verspricht, *keine* Ausnahmen auszuwerfen.
3. `void Zahlen_lesen_und_ausgeben()`
   `throw(DateiEnde, const char*)`
   verspricht, nur die deklarierten Ausnahmen auszuwerfen.

10.1.2 Exception-Hierarchie in C++

Die Basisklasse `exception` hat die folgende öffentliche Schnittstelle:

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

Davon leiten sich weitere ab, wie z.B.

```
namespace std {
  class logic_error : public exception {
    public:
      logic_error(const string& argument);
  };
}
```

Eigene Exception-Klassen sollten von den Exception-Klassen der Standardbibliothek abgeleitet werden.

11. Generische Programmierung in C++

ausführliche Literatur zum Download siehe
http://www.informatik.hs-bremen.de/~brey/stlb.html

Übersicht

C++ Sprachmittel für GP:

- Standardbibliothek
- Container
- Algorithmen
- Iteratoren

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:

```
// 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.

11.1 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
```

2. Schnittstellenobjekte 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 $k + n$

11.1.1 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;
};
```

Auswahl der Implementierung:

```
stack<int> IntStackA;                    // implizit deque
stack<int, list<int> > IntStackB;
stack<int, vector<int> > IntStackC;
```

11.1.2 Bauelemente

Container

Klassen als Templates zum Verwalten von Objekten
Basisklassen:

- `vector`
- `list`
- `deque`

ADTs:

- `stack`
- `queue`
- `priority_queue`
- `set` und `multiset`
- `map` und `multimap`

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.

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

Vorteile:

- Schnittstelle für Algorithmen
- Definierter Rahmen für eigene Datenstrukturen und Algorithmen, die mit der C++ Standardbibliothek zusammenarbeiten sollen

Iteratoren

= Objekte, die auf Elemente eines Containers verweisen

Operationen:

- `!=` Vergleich
- `*` Dereferenzierung
- `++` einen Schritt weitergehen
- ...

einfachster Fall: Iterator = Zeiger

Die Art der Bewegung ist *verborgen* (Kontrollabstraktion)

++ auf einem Vektor wirkt anders als auf einem binären Suchbaum!

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.

Vorhandene Algorithmen:

```
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_intersecti
set_difference      set_symmetric_difference    pop_heap
push_heap           make_heap                   sort_heap
min                 max                         min_element
max_element         lexicographical_compare     next_permutati
prev_permutation    accumulate                  inner_product
partial_sum         adjacent_difference
```

11.1.3 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);
}
```

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;
}
```

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.

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);
}
```

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;
  }
}
```

11.2 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 ...

```
// Iteratordefinition durch geschachtelte Klasse
template <class T>
class list {
public:
    class iterator
     : public bidirectional_iterator<T, ...>  {
         // ...
     }
    ...
    iterator begin();
    iterator end();
    // const_iterator entsprechend
    ...
```

11.2.1 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;
    // weitere Berechnungen mit Wert ...
    ++Woher;
}
```

- 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)

- 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;
```

11.2.2 Standard-Iterator und Traits-Klassen

Vorteil der Templates: Auswertung der Typnamen zur Compilierzeit

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?

Ja. Schwachstelle:

- 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 Tynamen 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)
};
```

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.

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.

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; // siehe unten
  };
}
```

Durch die `public`-Erbschaft sind diese Namen in allen abgeleiteten Klassen sicht- und verwendbar.

11.2.3 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?

Der Zweck ist nur

- zur Compilierzeit über die Namen Verträglichkeiten festzustellen oder
- einen möglichst effizienten Algorithmus auszuwählen

Ein selbstgeschriebener Iterator erbt die Eigenschaften vom parametrisierten Standard-Iterator:

```
template <class T>
class istream_iterator
  : public iterator<input_iterator_tag, T> {
  // ...
};
```

11.3 Zusammenfassung der STL-Eigenschaften

- Auswertung der Schnittstellen zur Compile-Zeit
- Entkopplung von Algorithmen und Containern mit Iteratoren
- Die Algorithmen arbeiten auch auf einfachen C-Arrays (Iterator = Zeiger)
- spezialisierte Algorithmen für bestimmte Container
- Öffentliche Typ- und Methodenschnittstellen als Vorgabe für eigene Erweiterungen
- dokumentierte und garantierte Zeitkomplexität
- keine Unterstützung von Subtyp-Polymorphismus!

– The End –

Mit dem bisherigem Kursprogramm ist die Basis für ein erfolgreiches Weiterstudium gelegt, auch mit Hilfe von Literatur (siehe 2. Anfangsseite).