C++: Etwas mehr als C

Ein Überblick über die Programmierung mit *C++*

Version 1.00

Klaus Wachtler

7. Januar 2006

Inhaltsverzeichnis

Kapitel 1

Einleitung

Dieser Text ist als Einführung und Arbeitsunterlage für das Programmieren mit *C++* gedacht. Er entstand aus meinen Unterlagen zu Seminaren und Vorlesungen an der Fachhochschule Hannover im Fachbereich Informatik[1].

Die vorliegende Version ist noch nicht vollständig (im wesentlichen fehlt die Beschreibung der Standardbibliothek); im Endzustand soll allerdings weitere Literatur in den meisten Fällen überflüssig werden.

Form und Inhalt sind:

Breidingstr. 17
29614 Soltau
Tel.: 05191/70271
email: AnyWare@Wachtler.de
http://www.wachtler.de

Für Verbesserungsvorschläge, Fehlerkorrekturen und ähnliches bin ich jederzeit dankbar.

Für weitere Kopien des Buches im jeweils aktuellen Stand bitte ebenfalls bei obiger Adresse nachfragen.

Am Ende des Buches befindet sich ein Stichwortverzeichnis.

Der Text wurde mit TEX und LATEX gesetzt.

[1] http://www.inform.fh-hannover.de

Der **Inhalt** dieses Skripts beschränkt sich auf den standardisierten Sprachumfang. Nicht standardisierte Erweiterungen werden weitestgehend ignoriert.

Zudem wird hier nicht dargestellt, wie man auf einem konkreten System Programme schreibt und erstellt. Dazu ist vielleicht das ergänzende Skript [KW:Tools] hilfreich.

Voraussetzungen:

Vorausgesetzt werden:

- Grundlegende Programmierkenntnisse
- Die Programmiersprache *C*

Wenn diese Voraussetzungen fehlen, ist es sinnvoll zuerst weitere Grundlagenliteratur zu konsultieren, beispielsweise [KW:0 ist false].

Als Einsteigersprache ist *C++* noch weniger geeignet als *C*, da der Sprachumfang von *C++*das komplette *C* beinhaltet – mit allen Vor- und Nachteilen. Das heißt, daß die Beherrschung von *C* als Handwerkszeug auch zur Programmierung in *C++* nötig ist. Deshalb halte ich es für blauäugig, *C++* für Programmierer zu empfehlen, die der Sprache *C* nicht mächtig sind, denn: mit *C++* kann man viele *C*-Probleme umgehen, aber man muß diese dazu kennen!

Insgesamt ist der Lernaufwand bis zum mittelmäßigen Verständnis bei *C++* noch höher als bei *C*. Das steht aber bei kleineren Programmen in keinem Verhältnis zum Nutzen. Zusammen mit der Tatsache, daß die meisten Erweiterungen von *C++* erst bei größeren Projekten zum Tragen kommen, die idealerweise auf mehrere Leute oder sogar auf mehrere Firmen verteilt sind, kann man erkennen, daß man *C++* besser gleich wieder vergißt, wenn man nur kleinere Projekte bearbeiten will. Die Vorteile von *C++* kommen erst bei großen Projekten so weit zum Tragen, daß der Lernaufwand gerechtfertigt ist.

Dementsprechend setzt dieser Text voraus, daß die Sprache *C* nicht nur vom Namen her bekannt ist, sondern wirklich zumindest in den Grundzügen beherrscht wird. Es wird hier weder auf *C* selbst eingegangen, noch auf die Standardbibliothek von ANSI-C. Dazu findet man in [KW:0 ist false] Näheres.

Was ist *C++*? *C++* ist eine relativ moderne Programmiersprache, die zur Zeit (neben *C*, Java, und einigen anderen) viel verwendet wird. Ihr Hauptanwendungsgebiet ist ursprünglich die systemnahe Entwicklung (Betriebssysteme, Compilerbau, Dienstprogramme) als Nachfolger von *C*. Ebenso wie *C* ist *C++* aber sehr flexibel und für viele andere Anwendungen gut geeignet, beispielsweise Numerik, Spieleprogrammierung und so weiter.

C++ ist im wesentlichen eine Erweiterung von *C*. Dementsprechend werden in *C++* geschrieben Programme klassisch kompiliert (in Maschinensprache übersetzt) und gelinkt (Programmteile vor der Ausführung mit ebenfalls kompilierten Bibliotheksfunktionen zusammengebunden), im Gegensatz zu Interpretersprachen.

Wo ist *C++* definiert? Die ersten Entwürfe stammen von Bjarne Stroustrup, also von AT&T. Von dieser Firma stammen auch jeweils lauffähige Compiler mit der Bezeichnung *cfront* und verschiedenen Versionsnummern (anfangs auf der Basis von *C++*→*C*-Konvertern).

Weitere Compilerhersteller halten sich oft an die damit vorgegebenen Definitionen. Allerdings beherrschen natürlich nicht alle im Umlauf befindlichen *C++*-Compiler alle neuen Entwicklungen, sondern übernehmen die Konzepte immer erst mit etwas Verzögerung.

Dadurch gibt es eigentlich nicht ein *C++*, an das sich alle halten. Vielmehr ist die Entwicklung noch im Gange. Aber seit der ANSI/ISO-Normung 1996 zeichnet sich ein Umfang ab, mit dem man gut arbeiten kann, und der inzwischen von allen neueren Compilern verstanden wird.

Leider hängt besonders die Entwicklung der zugehörigen Bibliotheken hinter der Normung hinterher.

Die im Folgenden beschriebenen Beispiele sollten aber mit allen aktuellen Systemen lauffähig sein.

Welche *C++*-Dialekte können einem Programmierer derzeit begegnen? Das sind im wesentlichen folgende Versionen:

1. AT&T *C++* (cfront) Version 2.0, entspricht der ersten Auflage von *Bjarne Stroustrup: Die C++ Programmiersprache* [Strou I]

2. (geplanter) ANSI-Standard (Gremium X3J16), entspricht AT&T cfront Version 2.1 bis 3.1 und der zweiten Auflage von *Bjarne Stroustrup: Die C++ Programmiersprache* [Strou II].

Mit AT&T cfront 3.0 erscheinen *templates*, mit 3.1 eine Definition von Ausnahmebehandlungen.

3. Endgültiger ANSI-Standard; entspricht [Strou IV]. Im ANSI-Standard werden zur Laufzeit verwendbare Typinformationen (RTTI: run time type information) vorgesehen; es gibt *templates* für Klassen und Funktionen, Ausnahmebehandlung, `namespace`, sowie eine Standardbibliothek mit Containern, darauf operierenden Iteratoren und Algorithmen, Strings, Ein-/Ausgabe und weiteren Klassen.

Welches *C++* wird hier behandelt? In diesem Text wird im Zweifelsfall das neuere *C++*, also der Stand in [Strou IV], vorgestellt.

Alle Beispiele sind mit dem Gnu-*C++*-Compiler, Version 2.95.3 auf einem Linuxrechner getestet (sowie großteils unter Windows 2000 mit Visual C++ 6.0 von Microsoft).
Der Gnu-*C++*-Compiler ist kostenlos für viele verschiedene Betriebssysteme (Unix in seinen vielen Abarten, DEC, MSDOS und seinen Nachfolgern, etc.) erhältlich.
Wie alle *C++*-Compiler benötigt er sehr viel Arbeitsspeicher beim Übersetzen eines Programms. Unter anderem deshalb ist MSDOS ein denkbar ungeeignetes System zur ernsthaften Programmentwicklung; Windows nicht unbedingt besser.
Ideal ist ein halbwegs modernes Unix. Für private Gehversuche ebenso wie für ernsthaftes Programmieren ist damit Linux eine ideale Basis; in allen Linuxdistributionen ist ein aktueller Gnu-*C++*-Compiler enthalten.

Nicht alle zur Zeit aktuellen Compiler unterstützen die neueren Entwicklungen. Aber alle guten Compiler konvergieren sehr schnell auf den ANSI-Standard zu, insbesondere der Gnu-*C++*.

Kapitel 2

Grundsätzliche Änderungen gegenüber *C*

Eine der Hauptideen, die hinter dem Übergang von *C* nach *C++* stecken, ist die Unterstützung von großen Projekten, möglichst ohne die Vorteile von *C* aufzugeben (insbesondere die Laufzeiteffektivität und die breite Basis an existierenden Programmen).

Bei der Entwicklung großer Projekte hat *C* etliche Schwächen, die bestenfalls durch enorme Selbstdisziplin des Programmierers und durch Gruppendisziplin in Teams zu umgehen sind.

Durch die Möglichkeit, in *C++* Programme objektorientiert anzulegen, läßt sich die Effizienz und die Sicherheit des Entwickelns drastisch steigern.

Was heißt in diesem Zusammenhang „objektorientiert"? In *C++* meint man damit im wesentlichen die Zusammenfassung von Datentypen und der zugehörigen Funktionen in sogenannten *Klassen*. Im Zusammenhang damit sind *Vererbung* und *Polymorphie* zu sehen.

Eine *Klasse* ist dabei eine Erweiterung der bekannten `struct`. Darin werden nicht nur Daten deklariert, sondern auch die zugehörigen Funktionen (*Elementfunktionen*, *Methoden*) einschließlich Initialisieren, Löschen, Kopieren und anderer Operationen auf den Objekten. Eine Klasse ist also die Definition eines Datentyps inklusive der Definition oder der Deklaration der Operationen (Memberfunktionen). Ein Objekt dieser Klasse, also eine mithilfe der Klasse vereinbarte Variable oder allokierter Speicherplatz, wird als *Instanz* der Klasse bezeichnet.

Durch die Möglichkeit, an eine Datenstruktur gleich die zugehörigen Funktionen anzuhängen, ist es recht einfach, die interne Datenstruktur vor dem eigentlichen Anwender der Daten zu verbergen und den Zugriff auf die Daten nur über Funktionen zuzulassen. Dies ermöglicht spätere Änderungen der Funktionsweise einer Klasse, ohne die Anwendung der Klasse ändern zu müssen, solange die Schnittstelle der Funktionen gleich bleibt (oder nur erweitert wird).

Vererbung bedeutet, daß man zur Schaffung einer Klasse (also die Weiterentwicklung einer aus *C* bekannten `struct`) ihre Eigenschaften nicht komplett neu definieren muß, wenn eine ähnliche Klasse bereits existiert. In einem solchen Fall kann man die neue Klasse aus der bereits existierenden (der sogenannten *Basisklasse*) ableiten. Die abgeleitete Klasse erbt alle Eigenschaften (Daten- und Funktionsdeklarationen) der Basisklasse. Nur Zusätze oder Änderungen gegenüber der Basisklasse müssen explizit beschrieben werden. So kann man –von einfachen Basisklassen ausgehend– eine Hierarchie von mehr oder weniger komplexen Klassen aufbauen und Erweiterungen relativ leicht implementieren, ohne bestehende Klassen manipulieren zu müssen.

Polymorphie von Funktionen und Operatoren bedeutet, daß Aufrufe von Funktionen beziehungsweise die Wirkungsweise von Operatoren vom Zusammenhang abhängen, in dem sie auftreten. So kann beispielsweise die Addition von komplexen Zahlen oder der Sinus einer komplexen Zahl anders definiert werden, als die entsprechenden Operationen mit reellen Zahlen. Der Compiler kann bei mehreren gleichnamigen Funktionen, die sich in ihrer Parameterliste (in der Reihenfolge der Parametertypen oder in der Anzahl der Parameter) unterscheiden, je nach den aktuellen Parametern die richtige Funktion aussuchen. Man kann also eine jede Funktion mit einer weiteren überladen, die sich in der Parameterliste unterscheidet. Ebenso können Operatoren mit Funktionen überladen werden, wenn sich die Parameter der Funktion beziehungsweise die Operanden von den bisherigen Definitionen unterscheiden. *Nicht möglich* ist es allerdings, Operatoren mit einer anderen Anzahl von Operanden zu überladen. Ebenfalls nicht möglich ist es, die Prioritäten der Operatoren untereinander zu verändern. Weiterhin ist es ausgeschlossen, die Operatoren von eingebauten Basisdatentypen zu überladen.

Objektorientiertes Programmieren bedeutet also, nicht Funktionen mit Daten zu füttern (funktions- oder aktionsorientiert), sondern stattdessen die Daten (also die Objekte) direkt zu benennen, und über sie die zugehörigen Aktionen zu veranlassen, weil ja die Funktionen über die Definition der Klassen direkt an die Objekte gebunden sind. Da ein Funktionsaufruf damit an ein bestimmtes Objekt gebunden ist, wird dafür auch oft die Vorstellung bemüht *dem Objekt eine Nachricht zu senden*. Ich verwende allerdings weiterhin die Bezeichnung *Funktionsaufruf* beziehungsweise *Methodenaufruf*, weil es technisch gesehen genau das ist.

Für größere Projekte ist es sinnvoll, Klassen in eigenen *Klassenbibliotheken*[1] zusammenzufassen. Insbesondere ist es sehr zweckmäßig, für alle Objekte einer Klasse gegebenenfalls eine automatische Initialisierung beim Erzeugen und automatische Aufräumfunktionen beim Freigeben von Objekten definieren zu können. Dies erhöht die sichere Anwendung von Klassen drastisch. Der Anwender, der ein Objekt einer Klasse schaffen will, vereinbart einfach eine Variable oder schafft ein Objekt mit `new` (analog zum `malloc()` in einem *C*-Programm). Damit wird ohne sein Zutun die zugehörige Initialisierung des Objekts durchgeführt (der Konstruktor wird aktiviert), die der Programmierer der Klasse dafür vorgesehen hat. Entsprechend wird beim Verlassen des Gültigkeitsbereichs der Variablen oder beim Freigeben des Speichers mit `delete` (als Gegenstück zu `new`) die Funktion zum Aufräumen des Objekts aufgerufen.

C++* ist im wesentlichen eine Erweiterung von *C*.** Das heißt, daß fast jedes sauber geschriebene *C*-Programm auch ein gültiges *C++*-Programm ist und sich gleich verhält, egal, ob es mit einem *C*- oder einem *C++*-Compiler übersetzt wird. Davon gibt es wenige Ausnahmen, die im Abschnitt **3 Änderungen gegenüber *C auf Seite 17 beschrieben werden.

Dadurch bleiben auch die systemnahen Möglichkeiten von *C* und die mögliche Laufzeiteffizienz erhalten.

Daß *C++* alles erlaubt, was auch in *C* üblich ist, heißt aber nicht, daß dies auch sinnvollerweise so verwendet werden soll. Insbesondere wird in *C* der Präprozessor wesentlich intensiver für Makros genutzt, als typischerweise in *C++*. Dies kommt daher, daß er eigentlich sehr leistungsfähig ist und viele Programmiererleichterungen ermöglicht, die in *C++* aber wesentlich eleganter über die eigentliche Sprachsyntax geschaffen werden. Da die entsprechenden *C++*-Konstruktionen nicht nur durch Textersatz vor dem Kompilieren greifen, sondern dem Compiler selbst bekannt sind, sind wesentlich sichereres Entwickeln der Quelltexte und höhere Effizienz des erzeugten Codes möglich.

[1]Oft wird hierfür der Begriff „Objektbibliothek“ verwendet. Dies ist aber unglücklich, da in diesen Bibliotheken keine Objekte (also die Instanzen) stehen, sondern nur die Klassen (also die Deklarationen).

Kapitel 3

Änderungen gegenüber *C*

Leider sind gegenüber *C* nicht nur Erweiterungen dazugekommen, sondern an einigen Stellen verhalten sich *C*-Programme etwas anders, wenn sie mit einem *C++*-Compiler übersetzt werden.

3.1 Logischer Datentyp `bool`

Es gibt (endlich) einen Datentyp für logische Werte (*wahr* oder *falsch*): `bool`.

Dazu kommen zwei vordefinierte Konstanten `true` und `false`.

Logische Ausdrücke (also das Ergebnis von Vergleichsoperatoren) liefern in *C++* nicht mehr `int`-Ergebnisse, sondern `bool`.

Man kann einen Ausdruck dieses neuen Typs problemlos in `int` und zurück verwandeln, sodaß ein Vergleich mit 0 jederzeit wie gewohnt möglich ist.

Zudem ist es möglich, an eine `bool`-Variable erst den Wert `false` zuzuweisen, und die Variable mit dem Operator `++` auf `true` zu setzen.

3.2 Typkontrolle

Die Typkontrolle bei Parametern, Zuweisungen und Vergleichen ist in *C++* in vielen Punkten wesentlich strenger als in K&R-*C* und etwas strenger als in ANSI-*C*. Bei sauber geschriebenen ANSI-*C*-Programmen sollte dies kein Problem sein. In Zweifelsfällen sind die Nörgeleien eines *C++*-Compilers manchmal auch bei ANSI-*C*-Programmen hilfreiche Hinweise auf mögliche Fehler.

Fehlende Funktionsdeklarationen werden als Fehler oder zumindest als Warnung gemeldet.

3.3 Explizite Typumwandlung

In *C* wird der Typ eines Ausdrucks vor der Weiterverwendung in seiner Umgebung explizit gewandelt (*cast*), indem man den gewünschten Typ in Klammern davor schreibt (`(long)i` wandelt den Wert von `i` nach `long`). In *C++* kann man alternativ auch den Typ vor einen geklammerten Ausdruck schreiben, der darauf hin konvertiert wird (`long(i)`). Die *C++*-Version ist manchmal leichter zu lesen, weil die Klammern deutlich den Ausdruck begrenzen, der gewandelt wird.

Diese neue Schreibweise ist nur möglich, wenn der zu erzielende Typ aus einem Wort besteht. Ansonsten muß man ihn wieder klammern und kommt damit zur alten Schreibweise zurück.

Wesentlich deutlicher kann man den Quelltext mit den neuen cast-Operatoren schreiben (**4.14 run time type information (RTTI)** auf Seite 131).

3.4 `char`-Konstanten

In *C* werden `char`-Konstanten generell –außer in Feldern– als `int`-Konstanten behandelt.

In *C++* behalten Sie dagegen den Typ `char`, werden aber natürlich bei Bedarf durch implizite Typkonversion in Ausdrücken umgewandelt.

Dies ist in *C++* dann von Bedeutung, wenn Funktionen überladen werden. Dabei existieren dann mehrere Funktionen mit demselben Namen, aber unterschiedlichen

Typen der Argumente (sie unterscheiden sich also in der *Signatur*). Dann wird gegebenenfalls bei einer `char`-Konstanten auch die Funktion mit einem `char`-Argument ausgewählt.

Das folgende Beispiel ist schon ein Vorgriff auf die Verwendung der Standardausgabedatei (`cout`) in *C++*. Man kann mit dem Operator `<<` (unter anderem für `cout` definiert in `iostream`) Werte in ihrem jeweiligen Standardformat ausgeben.

```
// Time-stamp: "(15.11.01 16:10) couta.cpp [Klaus Wachtler (aw38)]"

#include <iostream>

using namespace std;

int main()
{
  cout << 'A' << "\n";
  return 0;
}
```

Dieses Programm gibt das Zeichen $\boxed{\texttt{A}}$ aus, nicht dagegen seinen Wert im Zahlensystem.

Will man dagegen nicht das Zeichen, sondern seinen Wert, dann muß man explizit den Typ wandeln (*cast*), siehe **3.3 Explizite Typumwandlung** auf Seite 18:

```
// Time-stamp: "(15.11.01 16:03) cout65.cpp [Klaus Wachtler (aw38)]"

#include <iostream>

using namespace std;

int main()
{
    cout << int('A') << "\n";
    return 0;
}
```

3.5 Leere Parameterlisten

Sowohl in *C* als auch in *C++* kann man explizit angeben, daß eine Funktion gar keine Parameter bekommt (`f( void )`), oder mit einer variablen Anzahl von Parametern aufgerufen werden kann (beispielsweise `f( ... )`).

Aber die Angabe einer leeren Parameterliste (`f()`) unterscheidet sich in *C* und *C++*:

In *C* bedeutet eine leere Parameterliste in einer Funktionsdeklaration, daß die Funktion eine unbekannte Anzahl irgendwelcher Parameter erwartet (das entspricht `f( ... )`).

In *C++* dagegen drückt man damit aus, daß die Funktion *keine* Parameter erwartet. `int getchar();` ist in *C++* also gleichbedeutend mit `int getchar( void );`. Damit hat eine leere Parameterliste die Wirkung von `f( void )`.

3.6 Defaultparameter

Wenn eine *C++*-Funktion für einzelne Parameter oft den gleichen Wert bekommt, und nur manchmal ein anderer Wert angegeben wird, dann kann man dafür auch einen Default-Wert setzen. Die Parameter mit einem solchen vordefinierten Wert müssen die letzten sein, und dürfen keine Lücke aufweisen.

Folgende Beispiele sind legal:

```
int f( char *s, int i = 0 );
int g( char *s, int i = 0, int j = 12 );
int h( char *s = "" );
int i( char * = "" );      // Bei Deklaration kein Name nötig
...
f( "abc" );                // i ist 0.
f( "abc", 12 );            // i ist 12.
g( "abc" );                // i ist 0, j ist 12.
g( "abc", 2 );             // i ist 2, j ist 12.
g( "abc", 2, 30 );         // i ist 2, j ist 30.
```

Nicht zulässig sind dagegen folgende Beispiele:

```
int f( int i = 0, int j ); // falsch:
                           // Default-Parameter müssen von hinten
                           // her lückenlos sein!

int g( int i = 0, ... );   // falsch:
                           // Default-Parameter und variable
                           // Parameterlisten vertragen sich nicht!

int i( char *= "" );       // falsch:
                           // *= ist ein Token, deshalb Syntaxfehler!
                           // Hier muß also ein Leerzeichen eingefügt
                           // werden ("*=" -> "* =").
```

Hier ein vollständiges Beispiel:

```
// Time-stamp: "(15.11.01 16:10) defargch.cpp [Klaus Wachtler (aw38)]"
//
// demonstriert default-Argumente.

#include <iostream>

using namespace std;

void h ( const char *s = "abc" )
{
  cout << s << endl;
}

int main( int nargs, char *args[] )
{
  h();         // gibt "abc" aus
  h( "efg" ); // gibt "efg" aus

  return 0;
}
```

... und die zugehörige Ausgabe:

```
abc
efg
```

3.7 Einfachere Schreibweise bei `enum`, `struct` und `union`

Alle Typnamen, einschließlich der `struct`-, `union`- und `enum`-Bezeichner, werden vom Compiler in einem „Namensraum“ untergebracht und müssen deshalb auch untereinander eindeutig sein. Dafür kann man bei weiteren Vereinbarungen nach der ersten Deklaration auch das Schlüsselwort `struct`, `union` bzw. `enum` weglassen.

Beispiel:

```
struct person
{
char *vorname;
char *nachname;
};

struct person a;      /* In C muß es so sein!             */
person b;             // In C++ darf es auch so aussehen.
```

3.8 `void *` und `NULL`

In ANSI-C ist es möglich, an beliebige Zeigervariablen eine Zeigerkonstante vom Typ `void *` ohne expliziten *cast* zuzuweisen.

Dementsprechend ist in ANSI-C die „ungültige“ Zeigerkonstante `NULL` als `(void*)0` definiert (zumindest in stddef.h).

Eine Zuweisung eines `void*` an einen nicht-`void`-Zeiger ist in *C++* nicht mehr zulässig, dafür aber die Zuweisung einer ganzzahligen 0 (-Konstante) an jeden Zeiger. Deshalb muß in *C++* `NULL` einfach als Wert `0` definiert sein. (Die Zuweisung einer anderen `int`-Konstante ist dagegen ohne *cast* nach wie vor nicht erlaubt; der Wert `0` nimmt hier eine Sonderstellung ein.)

In [Strou IV] ist dafür ein nettes Beispiel zu finden:

```
char *p =
           !    !  !    !  !      !
           ! !  !  !    !  !      !
           !  ! !  !    !  !      !
           !    !  !!!!!!  !!!!!  !!!!1;
```

Die daraus resultierende Empfehlung mancher Leute, wegen drohender Verwechslungsgefahr gleich auf `NULL` zu verzichten, und stattdessen 0 oder einen als 0 definierten Wert (`NIL` oder ähnliches) zu verwenden, halte ich für übertrieben. Wenn man weiß, daß `NULL` kein `void *` mehr ist, kann man es recht gefahrlos verwenden. `NULL` hat sich als „ungültiger Zeiger“ durchgesetzt; es gibt m.E. keinen vernünftigen Grund, das zu ändern.

3.9 `main()` gibt implizit 0 zurück

Im Gegensatz zu ANSI-C gibt die Hauptfunktion `main()` den Wert 0 zurück, wenn nichts anderes explizit durch eine `return`-Anweisung angegeben wird.

Anmerkung: Visual C++ 6.0 weiß davon noch nichts.

3.10 Neue Schlüsselworte

Zu den aus *C* bekannten Schlüsselworten kommen noch etliche hinzu, so daß folgende Namen nicht frei verwendet werden können:

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

Eine kurze Erklärung zu allen Schlüsselworten wird in **D Glossar** auf Seite 295 gegeben.

Kapitel 4

Erweiterungen gegenüber *C*

4.1 Kommentare

In *C++* leiten zwei direkt aufeinanderfolgende Schrägstriche (die nicht innerhalb eines `/*...*/`-Kommentars oder in einem String stehen) einen Kommentar ein, der bis zum Rest der Zeile geht.

Die alte Version der Kommentare gilt weiterhin.

Wenn man die neuen Kommentare vorwiegend verwendet, dann kann man mit den alten `/*...*/`-Kommentaren probeweise Quelltextteile stillegen. Besser geeignet ist dafür aber weiterhin der Präprozessor mit `#if...#endif` oder `#ifdef...#endif`, weil es so keinen Konflikt mit Kommentaren gibt, und die Stilllegungen schachtelbar sind.

Siehe dazu auch **A.4 Allgemeiner Stil des Quelltextes** auf Seite 190.

4.2 Ein-/Ausgabe

In *C++* steht mit dem Standard-*streams*-Konzept eine Alternative zu den Funktionen `printf()`, `scanf()` und ihren Verwandten zur Verfügung.

Dabei muß es sich gar nicht um Dateien handeln, sondern mit identischen Operationen kann man sogenannte *string streams* (das sind im Speicher liegende Objekte) ebenso beschreiben und lesen wie Dateien.

Die Standardstreams sind in *C++* geschrieben und vom Programmierer modifzierbar. Insbesondere kann man Erweiterungen schreiben, um eigene Datentypen ebenso einfach ein- und ausgeben zu können, wie die vordefinierten Datentypen. Dies wird in diesem Abschnitt aber nicht vorgestellt, sondern nur die einfache Behandlung der existierenden Datentypen; ein Beispiel dafür findet sich in **4.12.9 Überladen von Operatoren** auf Seite 110 und in **5.9.1 Ein-/Ausgabe mit eigenen Klassen** auf Seite 181. Solche Modifikationen setzen ein gewisses Verständnis des Klassenbegriffs voraus, der erst im Abschnitt **4.12 Klassen** auf Seite 53 folgt. Deshalb wird die Ein- und Ausgabe eigener Objekte erst später vorgestellt.

Ebenso wie in *C* ist die Ein- und Ausgabe nicht Bestandteil der eigentlichen Sprachdefinition, sondern wird über eine eigene Bibliothek abgewickelt. Trotzdem wird dieses Thema gleich hier behandelt. So sind die Beispiele in den folgenden Abschnitten verständlicher und eigene Gehversuche überhaupt erst möglich.

Der wesentliche Vorteil der *streams* liegt darin, daß damit die fehleranfälligen Parameterlisten der *C*-Funktionen vermieden werden. Nachteilig ist natürlich, daß man das neue Konzept erst lernen muß, da es sich grundlegend von den alten Funktionsaufrufen unterscheidet.

Für die Verwendung der Ein-Ausgabe benötigt man in *C++* die zugehörigen Vereinbarungen in `<iostream>`.

Grundlage jeder Dateioperation ist ein Objekt vom Typ `ostream` (Ausgabe), `istream` (Eingabe) bzw. `iostream` (Ein- und Ausgabe mit ein und derselben Datei). Jedes solche Objekt entspricht einer geöffneten Datei.

So wie in *C* mehrere Dateien bei Programmstart automatisch geöffnet werden (`stdin`, `stdout`, `stderr`), existieren in *C++* entsprechende Objekte global: `cin`, `cout` und `cerr` wie die *C*-Dateien, sowie zusätzlich `clog`. `clog` entspricht `cerr`, ist aber im Gegensatz dazu gepuffert.

Im Übrigen sind die Funktionalitäten von Streams nicht auf Dateien beschränkt; beispielsweise kann man identisch in String-ähnliche Gebilde (*string streams*) schreiben oder daraus lesen.

Damit sollten die folgenden Beispiele verständlich sein; mehr Informationen zu Ein- und Ausgabestreams steht in **5.9 Streams** auf Seite 181.

4.3 Anonyme union

Wenn man in *C* eine `struct` vereinbart, in der ein `union` enthalten ist, dann braucht die `union` einen eigenen Namen innerhalb der `struct`, über den ihre Elemente angesprochen werden.

Beispiel: Eine etwas schlicht veranlagte Partnervermittlung möchte ihre Kunden möglichst platzsparend speichern, und möchte bei männlichen und weiblichen Kunden nur das für die erfolgreiche Vermittlung wichtigste Merkmal anlegen. Dies könnte so aussehen:

```
/* Time-stamp: "31.10.04 09:44 union1c.c klaus@wachtler.de"
 */

#include <stdio.h>

struct kunde
{
  char
    *vorname,
    *nachname;
  union
  {
    int     koerbchengroesse;
    double  einkommen;
  } dummy_union;
};

int main()
{
  struct kunde
    frau,
    mann;

  frau.dummy_union.koerbchengroesse = 'B';
  mann.dummy_union.einkommen = 1500.00;

  return 0;
}
```

In *C++* ist diese Regel gelockert: Man kann den Namen der `union` weglassen (*anonyme union*) und kann dann die Elemente der `union` wie Elemente der `struct` ansprechen:

```
// Time-stamp: "(15.11.01 12:03) union1.cpp [Klaus Wachtler (aw38)]"

struct kunde
{
  char
    *vorname,
    *nachname;
  union
  {
    int     koerbchengroesse;
    double  einkommen;
  };
};

int main()
{
  kunde
    frau,
    mann;

  frau.koerbchengroesse = 'B';
  mann.einkommen = 1500.00;
}
```

Das ist in *C++* möglich, weil die Elemente der anonymen `union` im selben Namensraum liegen, wie die der `struct`. Dies heißt aber umgekehrt, daß die Elementnamen eindeutig unterscheidbar sein müssen.

Allerdings ist es nicht möglich, innerhalb der anonymen `union` eine ebenfalls anonyme `struct` anzulegen; hier ist dann doch wieder eine zusätzlicher Name erforderlich:

```
// Time-stamp: "(15.11.01 12:03) union2.cpp [Klaus Wachtler (aw38)]"

enum haarfarbe_e
{
  blond, rot, dunkel
};
```

```
struct kunde
{

  char
    *vorname,
    *nachname;

  union
  {
    struct
    {
      int               koerbchengroesse;
      enum haarfarbe_e   haarfarbe;
    } weiblich;

    double   einkommen;
  };

};

int main()
{
  kunde
    frau,
    mann;

  frau.weiblich.koerbchengroesse = 'B';
  frau.weiblich.haarfarbe = blond;
  mann.einkommen = 1500.00;
}
```

4.4 Freie Position von Deklarationen

In *C++* müssen Deklarationen in Funktionen nicht mehr am Anfang eines Blocks vor den ausführbaren Anweisungen stehen, sondern nur vor ihrer ersten Verwendung. Das kann sehr übersichtlich sein, wenn beispielsweise Schleifenzähler oder andere nur sehr lokal verwendete Variablen direkt bei ihrer Verwendung definiert werden:

```
// Time-stamp: "(15.11.01 16:10) lokal.cpp [Klaus Wachtler (aw38)]"
```

```
#include <iostream>

using namespace std;

int main()
{
  cout << "sosiedassaus!" << endl;

  for( int i=0; i<10; i++ )
    {
      cout << "i ist " << i << endl;
    }

  return 0;
}
```

Leider herrscht bei bei Microsoft eine beachtliche Ignoranz gegenüber allen Normen vor, auch was den Gültigkeitsbereich einer so definierten Variable betrifft: Der VC++ (zumindest bis Version 6.0) läßt `i` auch über die Schleife hinaus leben, während [Strou IV] klar sagt:

> **6.3.3.1 Deklarationen in for-Anweisungen**
>
> Eine Variable kann im Initialisierungsteil einer `for`-Anweisung deklariert werden. Wenn dieser Initialisierer eine Deklaration ist, bleibt die eingeführte Variable (oder die Variablen) bis zum Ende der `for`-Anweisung gültig. Beispiel:
>
> ```
> void f( int v[], int max )
> {
>     for (int i=0; i<max; i++) v[i] = i*i;
> }
> ```
>
> Falls der Endwert eines Index nach dem Verlassen der `for`-Schleife noch benötigt wird, muß die Indexvariable außerhalb der `for`-Schleife deklariert werden…

Damit ist zwar beim GCC und allen anderen modernen Compilern, nicht aber mit dem VC++ folgendes standardkonform möglich:

```
  //...
```

```
for( int i=0; i<10; i++ )
{
  // ...
}
//...
for( int i=0; i<5; i++ )
{
  // ...
}
```

Für den VC++ müßte das Beispiel so aussehen:

```
//...
for( int i=0; i<10; i++ )
{
  // ...
}
//...
for( i=0; i<5; i++ )
{
  // ...
}
```

Um Quelltext jetzt portabel zu halten, bis [Strou IV] auch in Redmond in den Bücherläden steht, hat man zwei Möglichkeiten:

- Man schreibt die Definition vor alle Schleifen:

  ```
  //...
  int i = 0;
  for( i=0; i<10; i++ )
  {
    // ...
  }
  //...
  for( i=0; i<5; i++ )
  {
    // ...
  }
  ```

Das hat den Nachteil, daß die Variable `i` jetzt auch über die letzte Schleife hinaus bis zum Ende des umgebenden Blocks gilt. Das widerspricht dem Streben, Deklarationen möglichst lokal zu halten.

- Man schließt jede Schleife in einen Block ein, der die Gültigkeit von `i` dann begrenzt:

```
//...
{
    for( int i=0; i<10; i++ )
    {
      // ...
    }
}
//...
{
    for( int i=0; i<5; i++ )
    {
      // ...
    }
}
```

  Das ist unschön und umständlich.

- Man beschwert sich als Kunde bei Microsoft.[1]

4.5 Freier Speicher `new` und `delete`

In *C* verwendet man eine der Funktionen `malloc()`, `calloc()` und `realloc()`, um Speicher aus dem *Heap* zu allokieren, und bis zu einem passenden Aufruf von `free()` nach Gutdünken zu verwenden.

In *C++* hat man diese Möglichkeit auch noch. Es gibt aber einen Ersatz dafür, der meistens auch genutzt werden sollte: `new` und `delete`.

Zum Allokieren eines Objekts oder eines Feldes davon verwendet man jetzt
`new` *Objekttyp*
um ein Objekt vom Typ *Objekttyp* zu allokieren, oder

[1] Das ist wahrscheinlich sinnlos, weil man entweder keine Antwort bekommt, oder nur die Aussage, daß *Bjarne Stroustrup sich nicht an den MS-Standard hält*.

`new` *Objekttyp*`[`*Anzahl*`]`
um ein Feld mit *Anzahl* Elementen jeweils vom Typ *Objekttyp* zu allokieren.

Je nach dem, ob man ein Element oder ein Feld allokiert hatte, kann man den Speicher mit `delete` oder mit `delete[]` wieder freigeben.

Beispiel im *C*-Stil:

```
// Zeiger auf ein Element
int *p1;
// Zeiger auf viele Elemente
int *pviele;
// allokiert Platz für 1 int, und liefert Zeiger darauf
p1 = (int*)malloc( sizeof(int) );
if( p1==NULL ) { /* Fehlerbehandlung, Programmabbruch */  }
// allokiert Platz für 500 int, und liefert Zeiger darauf
pviele = (int*)malloc( 500*sizeof(int) );
if( pviele==NULL ) { /* Fehlerbehandlung, Programmabbruch */  }

// Speicher verwenden...
*p1 = 25;
pviele[0] = 1;
// ...
pviele[499] = 500;

// Speicher für 1 Element freigeben
free( p1 );
// Speicher für Feld freigeben
free( pviele );
```

Im *C++*-Stil sieht das selbe so aus:

```
// Zeiger auf ein Element
int *p1;
// Zeiger auf viele Elemente
int *pviele;
// allokiert Platz für 1 int, und liefert Zeiger darauf
p1 = new int;
// allokiert Platz für 500 int, und liefert Zeiger darauf
pviele = new int[500];
```

```
// Speicher verwenden...
*p1 = 25;
pviele[0] = 1;
// ...
pviele[499] = 500;

// Speicher für 1 Element freigeben
delete p1;
// Speicher für Feld freigeben
delete[] pviele;
```

Dabei muß man folgende Unterschiede unbedingt kennen:

- `new` liefert einen Zeiger auf den allokierten Speicher zurück, der bereits den passenden Typ hat (also bereits nach `(`*Objekttyp*`*)` konvertiert ist).

  Man muß demnach nicht explizit casten.

- Wenn `malloc()` (ebenso wie seine Abarten `calloc()` und `realloc()`) den geforderten Speicher nicht beschaffen kann, dann wird als Rückgabewert `NULL` geliefert.

  `new` dagegen liefert immer einen gültigen Zeiger, oder wirft im Fehlerfall eine Ausnahme (vom Typ `bad_alloc`, siehe **4.13 Ausnahmebehandlung** auf Seite 123 und **5.3 Ausnahmen der Standardbibliothek** auf Seite 143).

  Dadurch erspart man sich die Mühe, nach jedem Allokieren den Rückgabewert prüfen zu müssen. Gegebenenfalls muß man stattdessen mit einem `try`-Block die Ausnahme abfangen.

- Mit `new` ist es nicht möglich, ein bereits allokiertes Feld im Nachhinein in der Größe zu verändern, so wie es `realloc()` kann.

- `malloc()` und seine Verwandten sind Bibliotheksfunktionen; `new` und `delete` sind dagegen Operatoren der Sprache. Dadurch kennt der Compiler und damit auch der erzeugte Programmcode den jeweiligen Datentyp.

- `malloc()` und seine Verwandten beschaffen nur den Speicherplatz, ohne zu wissen welchen Typ das Objekt (oder die Objekte?) hat (haben).

  `new` dagegen kennt den Typ, und kann deshalb den beschafften Speicherplatz gleich sinnvoll initialisieren, indem für Klassenobjekte der zugehörige Konstruktor aufgerufen wird (siehe **4.12.3 Konstruktoren, Destruktoren** auf Seite 59).

Ebenso kann `free()` nur den Speicherbereich wieder freigeben, während `delete` und `delete[]` gegebenenfalls für jedes Objekt den passenden Destruktor aufrufen.

Was man weiterhin beachten muß:

- Man darf mit `new` allokierten Speicher nicht mit `free()` freigeben, ebenso wenig darf man mit `malloc()` (oder seinen Verwandten) allokierten Speicher mit `delete` oder `delete[]` freigeben.
- Hat man Platz für ein einzelnes Objekt allokiert (`new` *Objekttyp*), dann darf man zum Freigeben ausschließlich `delete` (ohne `[]`!) verwenden. Ebenso umgekehrt: hat man ein Feld allokiert (`new` *Objekttyp*`[`*Anzahl*`]`, auch wenn *Anzahl* gleich eins ist), dann darf man nur mit `delete[]` den Speicher wieder freigeben.
- Egal wie man den Speicher beschafft und freigibt: Nach der Freigabe darf man nicht mehr darauf zugreifen! Und zwar weder über Zeiger auf oder in den Speicherbereich, noch über **4.8 Referenzen** auf Seite 40.

4.6 scope resolution operator ::

Mit dem *scope resolution operator* `::` kann man zwischen einer globalen und einer lokalen Verwendung ein- und desselben Namens unterscheiden. Normalerweise bezieht sich ein Name im Zweifelsfall auf die innerste Vereinbarung. Mit dem Operator `::` wird dagegen das entsprechende globale Objekt angesprochen:

```
// Time-stamp: "(15.11.01 16:10) scope.cpp [Klaus Wachtler (aw38)]"

#include <iostream>

using namespace std;

int variable = 10;

int main()
{
  double variable = 3.1415;
```

```
  cout << variable << " " << ::variable << endl;

  if( (1<2) )
    {
      char   variable = 'A';

      cout << variable << " " << ::variable << endl;
    }

  return 0;
}
```

Mit `variable` wird immer die Vereinbarung des innersten Blocks ausgewählt, mit `::variable` dagegen immer die globale. Alle eventuell dazwischenliegenden Vereinbarungen sind nicht ansprechbar. Die Ausgabe dieses Programms lautet also:

```
3.1415 10
A 10
```

Im Zusammenhang mit Klassen hat der Operator :: noch eine erweiterte Bedeutung, siehe dazu **4.12 Klassen** auf Seite 53; ebenso im Zusammenhang mit Namensbereichen, siehe **4.7 Namensbereiche (`namespace`)** auf Seite 36.

4.7 Namensbereiche (`namespace`)

Je größer ein Projekt wird, und je mehr Teile dazu aus unterschiedlichen Quellen dafür zusammengeschraubt werden, desto größer wird die Wahrscheinlichkeit von Namenskollisionen, insbesondere bei global gültigen Namen (globale Variablen, Typen, Funktionen).

Dies ist leider nicht immer zu vermeiden.

Um das Problem in den Griff zu bekommen, gibt es in *C++* Namensbereiche (*name spaces*).

Sinnigerweise heißt das zugehörige Schlüsselwort `namespace`.

Dahinter steckt die Idee, jeden verwendeten Namen einem Namensbereich zuzuordnen, in dem der Name eindeutig sein muß. Damit werden Kollisionen zwischen Namen nur noch innerhalb des Namensbereichs möglich, anstatt innerhalb des ganzen Programms.

Beispielsweise sind alle Namen der Standardbibliothek (auch die Namen der alten ANSI-C-Funktionen und -Typen) im Namensbereich `std` deklariert.

In einem anderen Namensbereich können die selben Namen nochmals vergeben werden.

Zu jedem Zeitpunkt kann es einen oder mehrere voreingestellte Namensbereiche geben; die darin deklarierten Namen können direkt verwendet werden solange sie nicht kollidieren. Allen anderen Namen wird ihr jeweiliger Namensbereich vorangestellt (getrennt mit einem *scope operator* `::`; siehe **4.6 scope resolution operator ::** auf Seite 35).

Einen voreingestellten Namensbereich kann man mit

    `using namespace` *Name des Namensbereichs*`;`

einführen.

Weiterhin kann man gezielt einzelne Namen einem Namensbereich zuordnen mit:

    `using` *Name des Namensbereichs*`::`*Name*`;`

Eine `using`-Anweisung gilt bis zum Ende des umgebenden Blocks; beziehungsweise bis zum Ende des Quelltextes, wenn sie außerhalb einer Funktion steht.

Dies wird alles in dem folgenden kleinen Programm gezeigt:

```
// Time-stamp: "10.01.04 13:18 namespacedemo.cpp klaus@wachtler.de"

namespace Microsoft
{
  enum charakter_t { ganz_mies, mies };  // Datentyp
  charakter_t chef;                      // globale Variable Microsoft::chef
  charakter_t knecht[10000];             // globales Feld Microsoft::knecht
}

namespace Oracle
{
  enum charakter_t { ganz_mies, mies, geht_so };  // Datentyp
  charakter_t chef;                      // globale Variable Oracle::chef
  charakter_t knecht[2000];              // globales Feld Oracle::knecht
```

```
  charakter_t pfoertner;                 // globale Variable Oracle::pfoertner
}




int main( int nargs, char **args )
{
  Microsoft::chef = Microsoft::ganz_mies; // voll qualifiziert
  Oracle::chef = Oracle::mies;            // ebenfalls

  using namespace Microsoft;
  chef = ganz_mies; // entspricht Microsoft::chef = Microsoft::ganz_mies
  knecht[0] = mies; // entspricht Microsoft::knecht[0] = Microsoft::mies

  // falsch, weil Oracle nicht mit using namespace geöffnet wurde:
  //   pfoertner = geht_so;

  // Aber nach einem using namespace geht es:
  using namespace Oracle;  // jetzt sind sowohl Microsoft als auch Oracle offen
  pfoertner = geht_so;     // sowohl pfoertner als auch geht_so sind eindeutig

  // falsch, da nicht eindeutig:
  // Microsoft::chef oder Oracle::chef?
  // Ebenso Microsoft::ganz_mies oder Oracle::ganz_mies?
  //   chef = ganz_mies;

  {
    // Man kann aber für einzelne Namen eine Voreinstellung wählen:
    using Microsoft::chef;
    using Microsoft::ganz_mies;

    // jetzt klappt es:
    chef = ganz_mies;
  }

  // jetzt wieder falsch, da die beiden using Microsoft::chef
  // und using Microsoft::ganz_mies nur im Block stehen, der
  // bereits verlassen wurde:
  //   chef = ganz_mies;


} // main( int nargs, char **args )
```

Durch wiederholte `namespace` *Name d. Namesp.*`{...}`-Blöcke kann man zu einem bereits existierenden Namensbereich weitere Namen hinzufügen. Ein Entfernen bereits enthaltener Namen ist nicht möglich:

```
namespace  EinNamespace
{
    int i;
}

// ...

namespace  EinNamespace
{
    int j;
}

// ...

EinNamespace::i = 25;
EinNamespace::j = 13;
```

Hat man einen Namensbereich bereits definiert, dann kann man mit einer Art Zuweisung dafür noch einen anderen Namen vergeben:

```
namespace  EinFurchtbarLangerName
{
    int i;
}

namespace EFLN = EinFurchtbarLangerName;

// ...

EinFurchtbarLangerName::i = 25;
EFLN::i                   = 25; // identisch!
```

4.8 Referenzen

In *C* kann man Speicher entweder als vereinbarte Variable oder über einen Zeiger ansprechen. In *C++* kommt noch eine dritte Möglichkeit dazu, nämlich eine Referenz. Das ist ein neuer Name oder Ausdruck für eine Speicherstelle, also für eine vorhandene Variable oder eine Stelle in allokierten Speicher.

Eine Referenz auf einen Speicher kennzeichnet man mit einem vorangestellten `&`.

Beispiel:

```
// Time-stamp: "(13.11.01 20:55) ref0.cpp [Klaus Wachtler (aw38)]"
//
// Einfaches Beispiel fuer eine Referenz.

int main()
{
  int     a;     // eine Variable
  int    &b = a; // keine neue Variable, sondern nur ein
                 // neuer Name fuer a.

  a = 10;        // a erhaelt den Wert 10
  b = 12;        // a erhaelt den Wert 12
}
```

Eine Referenzvariable muß mit dem Speicherobjekt initialisiert werden, die sie ansprechen soll. Eine spätere Änderung ist dann nicht mehr möglich. Dabei ist es prinzipiell egal, ob das Speicherobjekt eine lokale oder eine globale Variable ist, oder frei allokiert wurde (`malloc()` etc. im *C*-Stil, oder `new` im *C++*-Stil).

Eine angenehme Anwendung von Referenzen ist das Umgehen von lästigen Zeigeroperationen. Beispielsweise würde in *C* eine Funktion, die zwei Variablen vertauscht, etwa so aussehen:

```
// Time-stamp: "(13.11.01 20:56) refpar.c [Klaus Wachtler (aw38)]"
 *
 * C-Variante der Referenzparameter.
 */

#include <stdio.h>
```

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

  *a = *b;
  *b = tmp;
}

int main()
{
  int
    v1 = 10,
    v2 = 5;

  printf( "%d %d\n", v1, v2 );
  tausche( &v1, &v2 );
  printf( "%d %d\n", v1, v2 );

  return 0;
}
```

Mit Referenzen kann man dieses Beispiel in *C++* etwas eleganter ausdrücken:

```
// Time-stamp: "31.10.04 09:46 refpar.cpp klaus@wachtler.de"
//
// Beispiel fuer Referenzparameter.

#include <iostream>

void tausche( int &a, int &b )
{
  int tmp = a;

  a = b;
  b = tmp;
}

int main()
{
  int
    v1 = 10,
```

```
    v2 = 5;

  std::cout << v1 << " " << v2 << std::endl;
  tausche( v1, v2 );
  std::cout << v1 << " " << v2 << std::endl;

  return 0;
}
```

Hier sind weder in der Funktion `tausche()` explizite Zeigeroperationen nötig, noch müssen beim Aufruf die Adressen von `v1` resp. `v2` übergeben werden. Trotzdem werden in `tausche()` `a` und `b` als Referenzen, also neue Namen von `v1` und `v2` behandelt. Vertauscht werden also nicht lokale Parameter `a` und `b`, sondern tatsächlich `v1` und `v2`!

Da eine Referenz eine Speicherstelle bezeichnet, darf man ihr auch etwas zuweisen. Ein Ausdruck vom Typ „Referenz auf ...“ darf also auf der linken Seite einer Zuweisung stehen (ein sogenannter *lvalue*), mit `++` oder `--` manipuliert werden, usw.. Beispiel:

```
// Time-stamp: "(15.11.01 16:10) reffun.cpp [Klaus Wachtler (aw38)]"
//
// Beispiel fuer Referenzparameter.

#include <iostream>

using namespace std;

// feldzugriff() liefert eine Referenz auf das feld[index].
int &feldzugriff( int feld[], int index )
{
  return feld[index];
}

int main()
{
  int
    i,
    array[5] = { 0 };

  array[2] = 25;                 // array[2] = 25
  feldzugriff( array, 3 ) = 35; // array[3] = 35
```

```
  feldzugriff( array, 4 )++;    // array[4]++

  for( i=0; i<5; i++ )
    {
      cout << array[i] << endl;
    }

  return 0;
}
```

Dieses Programm gibt aus:

```
0
0
25
35
1
```

Achtung, schöne Fehlermöglichkeiten! Referenzen kann man sich für viele Anwendungsfälle als Zeiger mit vereinfachter Schreibweise vorstellen. Damit kann man aber die Fehler, die man mit Zeigern gerne und oft praktiziert, auch eleganter mit Referenzen ausdrücken. Ebenso, wie man mit Zeigern auf Speicher zugreifen kann, auf den man gar nicht zugreifen darf, kann man mit Referenzen schöne Fehler produzieren:

- Zeiger `p` noch nicht initialisiert, aber schon eine Referenz mit `*p` initialisiert: **Falsch!**
- Speicher allokiert, Referenz darauf initialisiert, dann den Speicher wieder freigegeben, und die Referenz munter weiterverwenden: **Falsch!**
- Speicher allokiert, Referenz darauf initialisiert, dann den Speicher mit `realloc()` an eine andere Stelle verschoben, und die Referenz auf den inzwischen ungültigen Speicher weiter verwenden: **Falsch!**
- eine Referenz auf eine automatische Variable anlegen, nach dem Verlassen des umgebenden Blocks der automatischen Variable die Referenz weiterverwenden: **Falsch!**

Der Programmierer muß darauf achten, daß eine Referenz nur auf Speicher verweist, auf den man auch noch zugreifen darf. Beispielsweise ist es ziemlich debil, wenn eine Funktion eine Referenz auf eine eigene lokale Variable zurückgibt; nach dem Rücksprung aus dieser Funktion kann mit der Referenz niemand etwas anfangen, weil die zugehörige Variable nicht mehr existiert.

4.9 inline-Funktionen

In *C++* kann man eine Funktion als `inline` deklarieren. Das bedeutet, daß der Compiler -wenn er will- dafür gar keine eigene Funktion erzeugt, sondern den entsprechenden Maschinencode direkt an der Aufrufstelle einfügt.

Dadurch kann ein Programm schneller werden, aber unter Umständen auch größer (wenn die „Funktion" an vielen Stellen aufgerufen wird). Der Compiler wird deshalb nicht beliebig komplexe und lange Funktionen tatsächlich *inline* kompilieren, sondern nur kurze, einfache. Es steht jedem Compiler frei, selbst zu entscheiden, welche Funktionen er so behandelt.

Die Tatsache, daß unter Umständen gar keine echte Funktion existiert, hat natürlich zur Folge, daß man die Adresse einer `inline`-Funktionen nicht bestimmen kann.

Außerdem muß der Compiler in jedem betroffenen Modul die vollständige Definition einer solchen Funktion sehen können, weil er sie ja sonst nicht expandieren kann. Daher kommt –bei einer Aufteilung in eine Deklarationsdatei und eine Definitionsdatei– eine normale Funktionsdefinition zwar in die Definitionsdatei, die Definition einer `inline`-Funktion dagegen in die Deklarationsdatei, weil die ja von allen Anwendern der Funktion eingebunden wird.

4.10 Überladen von Funktionen

In *C++* ist es möglich, mehrere Funktionen mit ein und demselben Namen zu versehen. Dies heißt *Überladen von Funktionen.*

Die Auswahl, welche der gleichnamigen Funktionen in einem konkreten Fall gemeint ist, geht für den Compiler dann (hoffentlich) aus dem Zusammenhang hervor: nämlich aus der Parameterliste des Aufrufs (der Typ des Rückgabewerts wird dagegen **nicht** zur Unterscheidung herangezogen, auch wenn die überladenen Funktionen unterschiedliche Typen zurückgeben können!).

Hinter einem bestimmten Funktionsnamen können sich also mehrere verschiedene

Funktionen verstecken (Polymorphie).

Beispiel:

```
// Time-stamp: "(15.11.01 16:10) ueberlf.cpp [Klaus Wachtler (aw38)]"
//
// zeigt das Ueberladen von Funktionen.

#include <iostream>
#include <cstring>
#include <cstdlib>

using namespace std;

int summe( int s1 )
{
  cout << "summe( int " << s1 << " ) = ";
  return s1;
}

int summe( int s1, int s2 )
{
  cout << "summe( int " << s1 << ", int " << s2 << " ) = ";
  return s1 + s2;
}

double summe( double s1 )
{
  cout << "summe( double " << s1 << " ) = ";
  return s1;
}

double summe( double s1, double s2 )
{
  cout << "summe( double " << s1 << ", double " << s2 << " ) = ";
  return s1 + s2;
}

double summe( double s1, int s2 )
{
  cout << "summe( double " << s1 << ", int " << s2 << " ) = ";
  return s1 + s2;
```

```
}

char *summe( char *s1, const char *s2 )
{
  cout << "summe( char * \"" << s1
       << "\", const char * \"" << s2 << "\" ) = ";
  return strcat( s1, s2 );
}

char *summe( char *s1, int n )
{
  char  *p = (char *)malloc( strlen(s1) + 1 );
  int    i;

  cout << "summe( char * \"" << s1
       << "\", int " << n << " ) = ";

  strcpy( p, s1 );
  for( i=0, *s1 = '\0'; i<n; i++ )
    {
      strcat( s1, p );
    }

  return s1;
}

int main( int nargs, char *args[] )
{
  char
    string1[100] = "So so!",
    string2[100] = " Aber, Aber!";

  cout << summe( 10 ) << endl;
  cout << summe( 1.34 ) << endl;
  cout << summe( 10, 20 ) << endl;
  cout << summe( 1.34, 20 ) << endl;
  cout << summe( 3.1412856, 2.71828 ) << endl;
  cout << summe( string1, " Ja ja!" ) << endl;
  cout << summe( string2, 3 ) << endl;
}
```

An der Ausgabe sieht man ganz schön, daß –trotz des wiederholten Funktionsnamens–

jeder Funktionsaufruf zu einer anderen Funktion gehört. Der Compiler wählt unter den überladenen Funktionen diejenige aus, deren Parameterliste „am besten" zu den Parametern eines Aufrufs paßt:

```
summe( int 10 ) = 10
summe( double 1.34 ) = 1.34
summe( int 10, int 20 ) = 30
summe( double 1.34, int 20 ) = 21.34
summe( double 3.14129, double 2.71828 ) = 5.85957
summe( char * "So so!", const char * " Ja ja!" ) = So so! Ja ja!
summe( char * " Aber, Aber!", int 3 ) =  Aber, Aber! Aber, Aber! Aber, Aber!
```

Obwohl *C++* das nicht erzwingen kann, ist es natürlich sinnvoll, nur Funktionen eines Namens zu überladen, die auch sinngemäß ähnlich mit ihren Parametern reagieren. So kann es durchaus sinnvoll sein, in der Funktion

```
char *summe( char *s1, const char *s2 )
```

zwei Strings aneinanderzuhängen und diese Operation mit dem Namen `summe` zu garnieren. Aber in

```
char *summe( char *s1, int n )
```

wird ein String mehrfach hintereinandergehängt. Dieser Funktion den Namen `summe` zu geben, ist ziemlich schwachsinnig und fördert nicht unbedingt die Lesbarkeit eines Programms.

4.11 Funktionsschablonen (template-Funktionen)

Über das Überladen von Funktionen hinaus kann man in *C++* noch einen Schritt weiter gehen, und sehr ähnliche Funktionen, die man überladen will, zu einer Schablone zusammenzufassen.

In der Praxis tritt oft das Problem auf, eine Funktion mehrfach programmieren zu müssen, um ein und denselben Algorithmus auf verschiedene Typen von Argumenten anzuwenden.

Ein typisches Beispiel ist das Aussuchen des kleineren Werts aus zwei Argumenten. Gegebenenfalls muß man zu diesem Zweck mehrere Funktionen definieren, die jeweils `int`-, `unsigned int`-, `double`-Argumente usw. erhalten und entsprechende Rückgabewerte liefern.

Neben dem Aufwand, alle Versionen auszuformulieren, hat man dann noch das Problem, die jeweils passende Funktion mit dem richtigen Namen aufzurufen. (Letzteres ließe sich gegebenenfalls durch Überladen lösen.)

Schön wäre es, wie in *FORTRAN* generische Funktionen zu haben, die sich im Typ der Argumente unterscheiden und dementsprechend passende Ergebnisse liefern, aber trotzdem alle gleich heißen und zudem nicht mehrfach definiert werden müssen (wie in Abschnitt **4.10 Überladen von Funktionen** auf Seite 44 gezeigt), sondern nur einmal –unabhängig vom tatsächlichen Typ– hingeschrieben werden. Der Compiler muß dann die passende Version auswählen.

In *C* kann man sich ein Makro definieren, um den kleineren zweier Werte zu finden:

```
// Time-stamp: "(14.11.01 20:39) min.c [Klaus Wachtler (aw38)]"

#include <stdio.h>

#define MIN(a,b)  ( (a)<(b) ? (a) : (b) )

int main()
{
  int
    i = 5,
    j = 7;
  double
    d = 3.14,
    e = 2.7182818;

  printf( "MIN( i, j ) ist %d\n", MIN( i, j ) );
  printf( "MIN( d, e ) ist %f\n", MIN( d, e ) );

  return 0;
}
```

Damit hat man so etwas ähnliches wie eine generische Funktion definiert; leider ist es aber keine Funktion, auch wenn `MIN` in vielen Fällen sehr ähnlich verwendet werden kann.

Diese Definition hat gegenüber einer echten Funktion Vor- und Nachteile:

+ Das Makro ist schneller, da zur Laufzeit keine Parameter verwaltet werden müssen
+ Man kann das Makro ohne weiteren Aufwand für verschiedene (skalare) Datentypen verwenden und trotzdem immer mit dem selben Namen aufrufen
– Dagegen ist das Makro unsicherer als ein Funktionsaufruf, da etwa bei `MIN(i++,j++)` unübersichtliche Seiteneffekte auftreten, weil einer der beiden Parameter zweifach ausgewertet wird.

C++ bietet nun die Möglichkeit, generische Funktionen (wie sie zumindest festverdrahtet von FORTRAN bekannt sind), selbst zu definieren.

Das Gegenstück zum `MIN`-Makro könnte in *C++* so aussehen:

```
// Time-stamp: "(15.11.01 16:10) min.cpp [Klaus Wachtler (aw38)]"

#include <iostream>

using namespace std;

template<class T> T min( T a, T b )
{
  return a<b ? a : b;
}

int main()
{
  int
    i = 5,
    j = 7;
  double
    d = 3.14,
    e = 2.7182818;

  cout << "min( i, j ) ist " << min( i, j ) << "\n";
  cout << "min( d, e ) ist " << min( d, e ) << "\n";
```

```
  return 0;
}
```

Hier ist `min` als Schablone für Funktionen definiert, die alle den gleichen Namen (`min`) haben und sich im Typ der Argumente und des Rückgabewerts unterscheiden.

Bei jeder Verwendung in einem konkreten Zusammenhang setzt der Compiler einen Aufruf der passend generierten tatsächlichen Funktion ein. So werden die gefährlichen Nebeneffekte von Makros vermieden. Bei Bedarf kann man sich auch Definitionen der `min`-Funktion schreiben, die keine skalaren Daten vergleichen, sondern beispielsweise eigene Datentypen wie Listen oder Strings. Durch das Schlüsselwort `inline` wird die Funktion –sofern sie nicht zu umfangreich ist– gar nicht als eigene Funktion zur Laufzeit aufgerufen, sondern der Compiler erzeugt den entsprechenden Code an Ort und Stelle anstatt eines Funktionsaufrufs. Dadurch wird –genau wie in der Version als Makro– das Programm schneller, da zur Laufzeit keine Parameter angelegt werden müssen. Trotzdem sorgt der Compiler dafür, daß keine Seiteneffekte auftreten, beispielsweise durch die erwähnte mehrfache Auswertung von Parametern.

Die Verwendung solcher Schablonen ist nicht auf einen freien Parameter beschränkt, sondern man kann mehrere (mit Kommata getrennt) in die Dreiecksklammern schreiben. Weiterhin müssen diese Parameter gar keine Datentypen sein, sondern können auch Werte sein.

Genau genommen sind folgende Templateparameter zulässig:

- alle an dieser Stelle bekannten Datentypen (sowohl die fest eingebauten einfache Datentypen als auch alle Klassen)
- jeder konstante Ausdruck
- die Adresse eines Objekts im Speicher, oder einer Funktion mit externer Bindung (also nicht `static` deklariert)
- ein (nicht überladener) Zeiger auf ein Element

Insbesondere Variablen sind nicht als Parameter zulässig.

Da alle Parameter, die kein Typ sind, als Konstanten gelten, ist es nicht zulässig, sie im Template ändern zu wollen.

Ebenso wie normale Funktionen lassen sich Templatefunktionen überladen.

Zu beachten ist, daß sich die verschiedenen generierten Funktionen auch in den Aufrufparametern unterscheiden müssen, damit der Compiler die erzeugten überladenen Funktionen auseinanderhalten kann. Deshalb müssen sich die Templateparameter auf die Typen der Parameterliste auswirken. Ohnehin gilt, daß alle Parameter der Templatedefinition auch verwendet werden müssen.

4.11.1 Templatefunktion für mehrere Module

Was passiert nun, wenn man sein Programm auf mehrere Quelltexte verteilt? In einem Modul steht die Definition der Templatefunktion, hier kann der Compiler beim Übersetzen alle nötigen Versionen erzeugen; er weiß aber nicht, welche benötigt werden. In den anderen Quelltexten steht nur eine Deklaration; der Compiler weiß hier, welche Versionen kompiliert werden müssen, sieht aber die dazu nötige Definition nicht und behandelt die benötigte Funktion deshalb als `extern`. Aus diesem Grund muß man in dem Modul mit der Definition alle benötigten Versionen explizit deklarieren (zum Beispiel im obigen Beispiel als `int min( int, int );` usw.). Dann sieht der Compiler, was der Programmierer braucht und kann entsprechende Versionen in diesem Modul erzeugen. Dagegen ist es nicht immer empfehlenswert, die Definition über eine Headerdatei allen Modulen direkt zur Verfügung zu stellen: dann würden von ein und derselben Funktion eventuell mehrere Exemplare erstellt und zusammengebunden. Dies ist theoretisch nicht zulässig; auf manchen Systemen aber möglich. So wird das Programm nicht mehr portabel sein und außerdem unnötig groß.

Wie man es dreht und wendet, eine optimale Strategie für alle Fälle gibt es nicht. Man hat hier letztlich folgende Lösungsmöglichkeiten:

1. Man erklärt die `template`-Funktion zu `inline`, und macht ihre Definition allen Modulen soweit nötig zugänglich, beispielsweise über eine Headerdatei.

2. Man verzichtet auf `inline` (weil vielleicht die Funktion zu umfangreich ist), und macht trotzdem die Definition allen anderen Modulen zugänglich.

   Je nach System kann das zu langen Übersetzungszeiten und/oder großen Programmen führen. Daher ist diese Möglichkeit nicht generell empfehlenswert.

   Zudem kann man sich (ebenso wie bei der ersten Möglichkeit) Probleme einhandeln durch Definitionen vor dem jeweiligen `#include`, was zu unterschiedlichen Definitionen der Funktion in den verschiedenen Modulen führen kann.

3. Man definiert die Templatefunktion in genau einem Modul, und zwingt in eben diesem Modul den Compiler durch passende Deklarationen dazu, alle

nötigen Versionen zu generieren, beispielsweise:

```
template<class T> T min( const T &a, const T &b )
{
  return a<b ? a : b
}
int min( int, int );
double min( double, double );
char min( char, char );
```

In allen anderen Modulen reichen dann die Deklarationen.

Diese Vorgehensweise funktioniert gut, setzt aber voraus daß man beim Schreiben desjenigen Moduls, in dem die Definition der `template`-Funktion enthalten ist, bereits alle zu erzeugenden Versionen kennt. Dies ist leider nicht immer der Fall.

4. Wenn nichts anders hilft, dann gibt es noch das Schlüsselwort `export`.

   Man definiert die `template`-Funktion in genau einem Modul, und setzt ein `export` davor:

   ```
   export template<class T> T min( const T &a, const T &b )
   {
     return a<b ? a : b
   }
   ```

   In den anderen Modulen wird nur noch die Deklaration (beispielsweise über eine Headerdatei) bekannt gemacht (ohne `export`).

   Der Compiler muß nun zusehen, wie er damit klar kommt. Das kann mehr Zeit beim Übersetzen kosten, aber es funktioniert in jedem Fall.

4.11.2 Spezialisierung einer Funktionsschablone

Wenn man eine Funktionsschablone definiert hat, aber für eine bestimmte Kombination der Parameter eine andere Definition haben möchte, dann kann man für diese Kombination eine spezialisierte Version des `template` schreiben:

```
// Time-stamp: "(15.11.01 16:10) spectemplatef.cpp [Klaus Wachtler (aw38)]"

#include <iostream>

using namespace std;
```

```
// prüft, ob wert im Bereich [0..N-1] liegt, und
// liefert dementsprechend "ja" oder "nö" zurück:
template <class T, int N> const char *pruefe( T wert )
{
  return ( T(0)<=wert && wert<N ) ? "ja" : "nö";
}

// spezielle Version für T==unsigned und N==10:
template<> const char *pruefe<unsigned,10>( unsigned wert )
{
  return ( wert<10 ) ? "jau" : "nö";
}

int main( int nargs, char **args )
{
  // Aufruf der allgemeinen (generischen) Funktion:
  cout << pruefe<int,10>( 5 ) << endl;        //  5<10     "ja"
  cout << pruefe<int,10>( 15 ) << endl;       // 15<10     "nö"
  cout << pruefe<double,60>( 50.0 ) << endl;  // 50.0<60.0 "ja"

  // Aufruf der speziellen Version:
  cout << pruefe<unsigned,10>( 5u ) << endl;  // 5u<10u    "jau"

} // main( int nargs, char **args )
```

4.12 Klassen

4.12.1 Allgemeines zu Klassen

Die *Klasse* ist der Kern von *C++*; daher auch der ursprüngliche Name von *C++*, nämlich *C mit Klassen*.

Eine Klasse ist eine Erweiterung der aus *C* bekannten `struct` (oder `union`), also ein **Datentyp**. Die wesentlichen Erweiterungen sind einerseits die Möglichkeit, innerhalb der Klasse Funktionen unterzubringen, eine Klasse von einer bereits vorhandenen abzuleiten, sowie den Zugriff auf die Elemente der Klasse zu regeln.

Eine Klasse wird mit den aus *C* bekannten Schlüsselworten `union` oder `struct` vereinbart. Anstelle von `struct` kann man gleichwertig auch das neue Schlüsselwort `class` verwenden.

Die Funktionen, die innerhalb der Klasse vereinbart werden, heißen *Elementfunktionen* oder *member functions*. Oft werden sie auch als *Methoden* bzw. *methods* bezeichnet.

Zur Zugriffsregelung kann man innerhalb der `struct` oder `class` an den gewünschten Stellen `public:`, `private:` oder `protected:` einfügen. Alle folgenden Elemente der Klasse sind dann dementsprechend von außerhalb der Klasse benutzbar (`public:`) oder nur von den Elementfunktionen der Klasse (`private:`). Nur für die Weitervererbung an abgeleitete Klassen (siehe Abschnitt **4.12.7 Ableiten von Klassen** auf Seite 73) ist das Schlüsselwort `protected:` von Belang. Außer für abgeleitete Klassen ist `protected:` gleichbedeutend mit `private:`.

Solange man nichts anderes angibt, sind die Elemente einer mit `struct` vereinbarten Klasse öffentlich (wie mit `public:` vereinbart). Die Elemente einer mit `class` vereinbarten Klasse sind dagegen per Voreinstellung gekapselt (wie mit `private:` vereinbart). Dies ist in *C++* der einzige Unterschied zwischen `struct` und `class`.

Die Vereinbarung einer Struktur beziehungsweise Klasse ist eine Deklaration, die nur die Form der Daten und (über die Elementfunktionen) die darauf zulässigen Operationen beschreibt. Man schafft also einen Datentyp, analog zu `struct` oder `enum`. Die Definition von Objekten, also das Anlegen der Datenstrukturen im Speicher, geschieht dann über normale Variablenvereinbarung oder im dynamischen Speicher mit dem Operator `new` (**4.5 Freier Speicher `new` und `delete`** auf Seite 32). Von der Klasse, dem Datentyp, schafft man also (gegebenenfalls mehrere) konkrete Objekte, die dann auch *Instanzen* genannt werden.

Dazu ein kleines Beispiel: Ein kleines *C++*-Programm soll eine Klasse vereinbaren, die zwei Zahlen enthält. Die beiden Zahlen sollen von außerhalb nicht direkt manipulierbar sein, sondern nur über eine Funktion, die beide Elemente setzt sowie eine weitere Funktion, welche die Summe der beiden Elemente liefert.

In *C++* könnte das so aussehen:

```
// Time-stamp: "(15.11.01 16:10) klasse0.cpp [Klaus Wachtler (aw38)]"
//
// kleine Demonstration einer Klasse

#include <iostream>
```

```
using namespace std;

// Vereinbarung einer Klasse:
struct ersteklasse     // statt "struct" waere genausogut
                       // "class" moeglich
{
private:               // zahl1 und zahl2 sollen nicht von ausserhalb
                       // der Klasse direkt verwendbar sein, daher private
  int     zahl1;
  int     zahl2;

public:                // Die Elementfunktionen sind von ausserhalb auf-
                       // rufbar sein, daher public

  void setzezahlen( int z1, int z2 )
  {
    zahl1 = z1;
    zahl2 = z2;
  }

  int summe( void )
  {
    return zahl1 + zahl2;
  }

};
// Hiermit ist die Klasse mit dem Namen ersteklasse vereinbart.

// Kleines Testprogramm dazu:

int main( int nargs, char *args[] )
{
  // von der Klasse ersteklasse werden zwei Instanzen a und b erzeugt:
  ersteklasse
    a,
    b;

  // a wird auf { 2, 3 } und b auf { 10, 12 } gesetzt:
  a.setzezahlen( 2, 3 );
  b.setzezahlen( 10, 12 );

    // Die Summe a.zahl1+a.zahl2 ausgeben:
```

```
  cout << "a hat den Wert " << a.summe() << "\n";
  // dto. mit b:
  cout << "b hat den Wert " << b.summe() << "\n";

  return 0;
}
```

Die zugehörige Programmausgabe lautet:

```
a hat den Wert 5
b hat den Wert 22
```

Dazu einige Anmerkungen:

- Die gezeigte Klasse steht für sich allein, und ist nicht von einer anderen Klasse abgeleitet.
- Eine Klassendefinition muß mit einem Semikolon (`;`) abgeschlossen werden.
- Da die Elemente `zahl1` und `zahl2` als `private` deklariert sind, ist von außerhalb der Klasse kein direkter Zugriff darauf möglich.

  Beispielsweise ist innerhalb der Funktion `main()` folgendes nicht möglich:

  ```
  // a wird auf { 2, 3 } und b auf { 10, 12 } gesetzt:
  a.zahl1 = 2;       // alles unzulaessig, weil zahl1 und
  a.zahl2 = 3;       // zahl2 in der Klasse ersteklasse
  b.zahl1 = 10;      // als private vereinbart sind!
  b.zahl2 = 12;

  // Die Summe a.zahl1+a.zahl2 ausgeben:
  // In dieser Form ebenfalls unzulaessig, da zahl1
  // und zahl2 private sind:
  std::cout << "a hat den Wert " << a.zahl1+a.zahl2 << "\n";
  // dto. mit b:
  std::cout << "b hat den Wert " << b.zahl1+b.zahl2 << "\n";
  ```

- Eine Methode bezieht sich immer auf das Objekt, über das es aufgerufen wird.

  Wenn die Funktion `setzezahlen()` als `a.setzezahlen( 2, 3 )` aufgerufen wird, dann kann man innerhalb der Funktion mit `zahl1` und `zahl2` die Elemente `a.zahl1` und `a.zahl2` ansprechen.

- Es ist durchaus möglich, einige Datenelemente als `public:`, andere als `private:` zu deklarieren.

  Generell ist es aber besser, freien Zugriff auf die Datenelemente möglichst einzuschränken und nur über Elementfunktionen zu ermöglichen. Dies erleichtert die Fehlersuche in Programmen erheblich, da dann nur noch die Elementfunktionen als Schuldige in Frage kommen, wenn Datenelemente merkwürdige Werte annehmen. Weiterhin sind Programmänderungen wesentlich sicherer, wenn es eine klare Schnittstelle gibt.

- Wenn in einer Klasse nur Elementfunktionen als `public:` deklariert werden, legt man mit den Funktionen **alle** Operationen fest, die von außerhalb der Klasse mit den Daten möglich sind. Dies ist bei größeren Projekten oft sehr zweckmäßig.

- Die Elementfunktionen müssen nicht innerhalb der Klasse definiert werden; dort reicht auch eine Deklaration.

  Natürlich muß nichtsdestotrotz die Definition irgendwo im Programm nachgeholt werden. Dies kann man zur Aufteilung in übersichtliche Module nutzen; dazu folgt gleich ein Beispiel.

  Beispiel für eine Methodendefinition in der Klasse:

```
class A
{
private:
  // Daten...
  int    wert;
public:
  // Memberfunktionen:
  int holeWert()     // Die Methode wird hier gleich DEFINIERT
  {
    return wert;
  }
};
```

  So dagegen wird die Methode nur *deklariert*:

```
class A
{
private:
  // Daten...
  int    wert;
public:
```

```
  // Memberfunktionen:
  int holeWert();    // Die Methode wird hier nur DEKLARIERT
};
```

In diesem Fall ist die Definition der Methode an anderer Stelle noch nötig. Dies kann prinzipiell in jedem Quelltext des Programms möglich sein; allerdings muß der Compiler an dieser Stelle die Klasse `A` kennen. Da es ja viele Klassen mit einer Methode namens `holeWert()` geben kann, ist es nötig die Funktion mit dem Klassennamen und dem *scope resolution operator* `::` eindeutig zu zuordnen:

```
// class A muß hier bekannt sein,
// dadurch ist A::HoleWert() auch deklariert!

// Memberfunktionen der Klasse A:
int A::holeWert()     // Die Methode wird hier DEFINIERT
{
  return wert;
}
```

- Elementfunktionen sind manchmal sehr kurz, so daß der Funktionsaufruf teilweise mehr Rechenzeit kostet als die Funktion selbst. Dann sollte die Elementfunktion als `inline` (siehe Abschnitt **4.9 inline-Funktionen** auf Seite 44) vereinbart werden.
- Wenn eine Elementfunktion in der Klassendeklaration nicht nur deklariert, sondern auch gleich definiert wird, dann ist sie automatisch `inline`

4.12.2 Der Zeiger `this`

Innerhalb von Elementfunktionen kann man die Elemente eines jeden Objekts mit ihrem Elementnamen ansprechen. Damit ist das jeweilige Element der aktuellen Instanz einer Klasse gemeint, obwohl bei der Definition der Elementfunktionen (zur Übersetzungszeit) noch gar keine Instanzen existieren. Manchmal ist es aber innerhalb von Elementfunktionen nötig, nicht nur einzelne Elemente einer Instanz anzusprechen, sondern die ganze Instanz selbst. Dafür gibt es (aber nur innerhalb der Elementfunktionen) den Zeiger `this`. Dieser zeigt zur Laufzeit auf die aktuelle Instanz, also das auf konkrete Objekt im Speicher. Jede Elementfunktion kann den Zeiger `this` ohne weitere Deklaration verwenden (es sei denn, die Funktion ist als `static` deklariert, siehe **4.12.5 Statische und nichtstatische Methoden** auf Seite 71).

Ein Beispiel zur Verwendung von `this` ist in **4.12.9 Überladen von Operatoren** auf Seite 110 zu sehen. Dort wird für die Klasse `Cardinal` der Zuweisungsoperator `=` mit einer eigenen Funktion überladen. Diese soll eine Referenz auf den zugewiesenen Wert liefern und kann dies mit der Anweisung `return *this` tun.

4.12.3 Konstruktoren, Destruktoren

Bei der Erzeugung eines jeden Objekts zur Laufzeit (für statische Variablen beim Programmstart, für automatische Variablen beim Eintreten in den zugehörigen Block, außerdem beim expliziten Aufruf von `new`, siehe **4.5 Freier Speicher `new` und `delete`** auf Seite 32) wird in *C++* –ebenso wie bereits in *C*– ausreichend Speicherplatz für alle vereinbarten Elemente des Objekts zur Verfügung gestellt, nämlich `sizeof(...)` Bytes. Bei allen globalen sowie bei automatischen initialisierten Variablen wird der neue Speicher außerdem mit 0 oder dem entsprechenden Initialisierungswert beschrieben. Beim Freigeben des Objekts (Verlassen des Blocks bei automatischen Variablen, `delete`, oder Programmende) wird dieser Speicher wieder freigegeben.

Manchmal sind aber zwischen der Anforderung (z.B. Variablenvereinbarung) und dem erstmaligen Verwenden des Speichers noch weitere Initialisierungen nötig (Öffnen von Dateien, Allokieren von weiterem Speicher, auf den Zeiger in der eben vereinbarten `struct` zeigen sollen, etc.). Ebenso kann es nötig sein, nach dem letzten Verwenden einer Variablen noch irgendwelche Aufräumarbeiten durchzuführen, beispielsweise Speicher freizugeben oder Dateien zu schließen.

Dies resultiert in *C* immer in Anweisungen desjenigen, der die entsprechende Datenstruktur kreiert, an den Anwender der Struktur in der Art: „Vor dem ersten Verwenden unbedingt die Funktion `init_blabla() aufrufen`, und nach dem letzten Verwenden auf jeden Fall noch ...“ (Beispiel: vor dem Schreiben und Lesen einer Datei `fopen()` aufrufen, danach `fclose()`).

Das ist zum einen lästig für den Anwender, und andererseits extrem fehleranfällig, weil die entsprechenden Tätigkeiten unbedingt synchron zum Geltungsbereich der entsprechenden Objekte erfolgen müssen, der Compiler aber dabei keine Hilfestellung leisten kann. Ein typischer Fall ist die Verwendung von `FILE *`. Wenn man dabei nicht die richtige Reihenfolge von `fopen()`, `fclose()` und der restlichen Verwendung einhält, dann hat man verloren.

In *C++* hat man dafür eine wesentlich elegantere Möglichkeit: Neben dem erwähnten automatischen Beschaffen und Freigeben von Speicher kann man sich zu jeder Klasse bei Bedarf einen *Konstruktor* und/oder einen *Destruktor* definieren. Der

Konstruktor wird dann bei jedem Erzeugen eines Objekts (aber nicht beim Beschaffen des entsprechenden Speichers mit `malloc()`, `calloc()` und `realloc()`!) aufgerufen, der Destruktor entsprechend beim Verlassen des jeweiligen Geltungsbereichs.

Der Name des Konstruktors ist identisch mit dem Namen der entsprechenden Klasse; der Destruktor hat dazu noch ein vorangestelltes `~`. Beide werden ohne einen bestimmten Rückgabewert definiert.

Anstatt *Konstruktor* wird auch oft der Begriff *ctor* (als Kurzform des englischen *constructor*) verwendet; anstatt *Destruktor* entsprechend *dtor* (für *destructor*).

Konstruktoren kann man auch überladen, um eine Initialisierung mit oder ohne Parametern zu ermöglichen. Ein Beispiel folgt in **4.12.9 Überladen von Operatoren** auf Seite 110.

Wie ein Konstruktor die Elemente der Klasse initialisieren kann, ist in **4.12.7.3 Initialisierung der Klassenelemente** auf Seite 85 beschrieben.

4.12.3.1 Aufruf und Auswahl von Konstruktoren

Im Gegensatz zu anderen Methoden können weder der Konstruktor noch der Destruktor aus normalem Programmcode heraus explizit aufgerufen werden.

Der Aufruf genau eines Konstruktors findet automatisch statt, wenn die Lebensdauer eines Objekts im Speicher beginnt:

- für automatische Variablen: wenn der Programmfluß die Definition erreicht
- für globale Variablen: bei Programmstart
- für lokale statische Variablen: wenn der Programmfluß erstmals die Definition erreicht
- für neu allokierten Speicher: wenn mit `new` Speicher für ein Objekt beschafft wird (**4.5 Freier Speicher `new` und `delete`** auf Seite 32)
- einmal für jedes Element: wenn mit `new[]` Speicher für ein Feld von Objekten beschafft wird.

Dabei wird zuerst automatisch ausreichend Speicher beschafft für alle Elemente des Objekts (also mindestens `sizeof(`*Objekt*`)` Bytes); dann erst erfolgt der Konstruktoraufruf.

Innerhalb des Konstruktors kann also auf alle Elemente zugegriffen werden. Die Memberfunktionen können alle angesprochen werden.

Da man den Konstruktor ebenso wie andere Funktionen und Methoden mit verschiedenen Parameterlisten überladen kann, kann man einem Objekt beim Erzeugen gleich passende Parameter mit auf den Weg geben. Dies geschieht bei Variablen durch Angabe der Parameter beim Definieren in Klammern hinter dem Variablennamen, bei `new` dagegen hinter dem Typnamen:

```
class Auto
{
private:

  int    AnzahlRaeder;
  double Motorleistung;

public:

  // Defaultkonstruktor:
  Auto()
  {
    Motorleistung = 0.0;
    AnzahlRaeder = 4;
  }

  // überladener Konstruktor:
  Auto( double LeistungKW, int nRad = 4 )
  {
    Motorleistung = LeistungKW;
    AnzahlRaeder = nRad;
  }

};

int main( int nargs, char **args )
{

  Auto   Golf; // 4 Räder, keine Leistung
  Auto   Ferrari( 250.0 ); // 4 Räder, 250kW
  Auto   UnfallFerrari( 0.0, 3 ); // 3 Räder, keine Leistung (mehr)

  Auto *pLKW = new Auto( 300.0, 10 ); // dicker Brummer
```

```
  // ...
}
```

4.12.3.2 Defaultkonstruktor

Der Konstruktor ohne Parameter ist der *parameterlose Konstruktor*.

Wenn der Programmierer der Klasse überhaupt keinen Konstruktor zur Verfügung stellt, dann wird vom Compiler ein parameterloser Konstruktor generiert. Dieser heißt dann *Defaultkonstruktor*, und initialisiert alle Elemente jeweils mit ihrem parameterlosen Konstruktor beziehungsweise ihrem Defaultkonstruktor.

Es wird kein Defaultkonstruktor generiert, wenn mindestens ein anderer (mit oder ohne Parameter) definiert ist. Das bedeutet, daß kein Objekt ohne Parameter erzeugt werden kann, wenn nur Konstruktoren mit Parametern definiert werden.

4.12.3.3 Konstruktoren mit Defaultparametern

Ebenso wie bei anderen Funktionen (siehe **3.6 Defaultparameter** auf Seite 20) kann man Defaultparameter angeben, um weniger Funktionen definieren zu müssen.

In dem obigen Beispiel könnte man die beiden Konstruktoren auch noch zu einem zusammenfassen:

```
class Auto
{
private:

  int    AnzahlRaeder;
  double Motorleistung;

public:

  // Konstruktor:
  Auto( double LeistungKW = 0.0, int nRad = 4 )
  {
    Motorleistung = LeistungKW;
    AnzahlRaeder = nRad;
```

```
  }

};
// ...
```

4.12.3.4 Aufruf eines Destruktors

Entsprechend wird der Destruktor automatisch aufgerufen, wenn die Lebensdauer eines Objekts endet:

- für automatische Variablen: wenn der Programmfluß das Blockende nach der Definition erreicht
- für globale Variablen: bei Programmende
- für lokale statische Variablen: ebenfalls bei Programmende
- für neu allokierten Speicher: wenn mit `delete` Speicher für ein Objekt freigegeben wird; sonst bei Programmende
- einmal für jedes Element: wenn mit `delete[]` Speicher für ein Feld von Objekten freigegeben wird; sonst bei Programmende

Während des Destruktoraufrufs ist der für das Objekt automatisch allokierte Speicher noch vorhanden; danach wird er automatisch freigegeben.

Ein Destruktor kann keine Parameter haben, und kann deshalb auch nicht überladen werden.

4.12.3.5 Defaultdestruktor

Ähnlich wie bei den Konstruktoren wird der Compiler auch einen *Defaultdestruktor* generieren, wenn man nicht explizit einen eigenen definiert.

Dieser Defaultdestruktor ruft für alle Elemente deren Destruktor auf.

4.12.3.6 `malloc()` und seine sauberen Freunde

Achtung: Konstruktoren und Destruktoren werden zwar aufgerufen für Speicherobjekte, die mit `new` (oder `new[]`) beschafft beziehungsweise mit `delete` (oder `delete[]`) freigegeben werden.
Aber für Objekte, die man versucht mit `malloc()`, `realloc()`, oder `calloc()` zu beschaffen, oder mit `free()` freizugeben, wird weder ein Konstruktor noch ein Destruktor aufgerufen!

Dasselbe gilt für `alloca()`, das meist unter Unix und auf jedem GNU-System zur Verfügung steht: es wird kein Konstruktor und kein Destruktor aufgerufen!

4.12.3.7 `explicit`

Normalerweise bemüht sich der Compiler, bei Funktionsaufrufen mit einer gegebenen Parameterliste eine möglichst passende Funktion zu finden. Wenn es keine Funktion mit einer exakt passenden Signatur gibt, dann werden alle in Frage kommenden überladenen Funktionen betrachtet, und versucht mit einer Reihe einfacher Typkonversionen (`char` zu `int` und umgekehrt, `int` zu `long` und umgekehrt, `int` zu `double` und so weiter) eine passende Auswahl zu treffen. Wenn diese Auswahl zu genau einer der überladenen Funktionen führt, dann wird diese genommen, ansonsten wird eine Fehlermeldung ausgegeben.

Mit Methoden und auch mit Konstruktoren wird normalerweise ebenso verfahren.

Gerade bei Konstruktoren kann diese automatische Konversion aber unerwünscht sein. Beispielsweise kann ein fiktiver Typ `meinstring` mit einer ganzen Zahl initialisiert werden; dies gibt dann die anfängliche Anzahl von (Leer-)Zeichen an; mit einem anderen String oder einer `const char*`-Konstante dagegen gibt man den anfänglichen Inhalt vor:

```
meinstring   s1( 10 );     // 10 Leerzeichen
meinstring   s2( "abc" );  // 3 Zeichen: 'a', 'b', 'c'
```

Weitere Konstruktoren sollen nicht existieren; insbesondere ein Konstruktor, der ein `char` entgegen nimmt, soll nicht vorhanden sein. Ein Benutzer könnte jetzt (ohne vorher genau in seinen Unterlagen zu lesen) versuchen, ein `meinstring`-Objekt mit einem `char` zu initialisieren:

```
meinstring   s1( 'a' );     // gewünscht: ein Zeichen a
```

Die Vorstellung, damit einen String mit einem `a` erzeugt zu haben, ist aber Illusion: weil kein Konstruktor mit einem `char`-Parameter existiert, ist derjenige mit einem `int`-Parameter für den Compiler so verlockend und eindeutig, daß der

Wert `'a'` in eine `int` konvertiert wird (auf einem Rechner mit ASCII-Zeichensatz ergibt das den Wert 97), und es wird ein String `meinstring( 97 )` erzeugt!

Um solche subtilen Fehler zu vermeiden, kann man beim Erstellen der Klasse mit `explicit` ausdrücken, daß der jeweilige Konstruktor nur mit exakt passenden Parametern aufgerufen werden darf, und ein implizites Konvertieren nicht erfolgen soll:

```
class meinstring
{
  // ...
  explicit meinstring( int anzahl );
  meinstring( const char *initstr );
  // ...
}
```

So wird der Anwender gezwungen, einen Konstruktor mit genau passender Signatur zu verwenden, wenn er `meinstring( int anzahl )` aufgerufen haben möchte.

4.12.4 Statische und nichtstatische Klassenelemente

Mit der Definition einer Klasse erzeugt man normalerweise einen Datentyp, für dessen Datenelemente erstmal noch kein Platz geschaffen wird. Erst beim Anlegen von Variablen dieses neuen Typs (global, oder lokal, statisch oder automatisch oder auf dem Heap mit `new`, siehe **4.5 Freier Speicher `new` und `delete`** auf Seite 32) wird –wie mit Variablen einfachen Typs auch– zur Laufzeit des Programms Platz im Speicher belegt.

Der Speicher für Datenelemente wird also nicht durch die Klassendefinition geschaffen, sondern erst durch die Instanziierung, und existiert zur Laufzeit des Programms für jede Instanz, also jedes gerade „lebende“ Objekt der Klasse.

Ähnlich wird in *C* mit:

```
typedef struct
{
  int       i;
  double    d;
} neuer_t;       // hier wird noch kein Platz belegt!
```

```
neuer_t   glob;    // eine Instanz von neuer_t; belegt Platz

void up()
{
  neuer_t  a, b;   // zwei weitere Instanzen von neuer_t; belegt Platz
  // ...
}
```

mit der Typdefinition `typedef struct{...} neuer_t;` kein Speicher belegt; erst mit der Vereinbarung der globalen Variablen `glob` und (solange die Funktion `up()` aktiv ist) mit der Definition der Variablen `a` und `b` wird jeweils Platz für eine `int` und eine `double` belegt.

In *C++* ist es mit Klassen normalerweise ähnlich[2]:

```
class NeuerTyp
{
public:

  int       i;
  double    d;

  void setzeWerte( int neuesi, double neuesd )
  {
    i = neuesi;
    d = neuesd;
  }

};               // hier wird noch kein Platz belegt!


NeuerTyp   glob;    // eine Instanz von NeuerTyp; belegt Platz

void up()
{
  NeuerTyp  a, b;   // zwei weitere Instanzen von NeuerTyp; belegt Platz
  // ...
}
```

[2]Der Einfachheit halber werden die Datenelemente hier auch als `public` vereinbart, auch wenn das in realen Programmen eher nicht wünschenswert ist

In diesem Beispiel gibt es drei Elemente mit dem Namen `i`, nämlich `glob.i`, und (solange `up()` aktiv ist) noch `a.i` und `b.i`; ebenso gibt es das Element `d` einmal je Instanz. Falls `up()` direkt oder indirekt rekursiv aufgerufen wird, kann es zu einem Zeitpunkt auch noch mehr Instanzen von `a` und `b` geben.

Im Gegensatz zu den Datenelementen werden die Methoden zu Maschinencode kompiliert, der zur Laufzeit nur einmal pro Klasse vorhanden ist. Der Code ist also generell statisch. Auch wenn man die Funktion `setzeWerte()` als `glob.setzeWerte(...)` oder `a.setzeWerte(...)` oder `b.setzeWerte(...)` aufruft, so wird doch in allen drei Fällen derselbe Code ausgeführt.

Das drückt sich auch darin aus, daß die Adresse der Funktion immer dieselbe ist, egal über welches Objekt man sie bestimmt (`&(glob.setzeWerte)`, `&(a.setzeWerte)`, `&(b.setzeWerte)`).

Möchte man dieses Verhalten (ein gemeinsames Vorkommen für alle Instanzen) auch für Datenelemente haben, dann kann man diese Elemente innerhalb der Klasse einfach als `static` deklarieren:

```
class NeuerTyp
{
public:

  static int  i;  // jetzt statisch!
  double      d;

  void setzeWerte( int neuesi, double neuesd )
  {
    i = neuesi;
    d = neuesd;
  }

};               // hier wird noch kein Platz belegt!
```

Jetzt gibt es ab Programmstart genau ein Element `i`, egal wieviele Instanzen der Klasse es zu einem bestimmten Zeitpunkt gerade gibt.

Weil `i` faktisch gar nicht mehr an ein bestimmtes Objekt gebunden ist, ist es möglich und auch sinnvoll, es gleich über seinen Klassennamen anzusprechen (`NeuerTyp::i` anstatt `glob.i` oder `a.i` oder `b.i`).

Wie kann man auf ein solches statisches Element zugreifen? Zum einen kann man den Wert ja von außerhalb ansprechen (im Beispiel als `NeuerTyp::i`), falls er im `public:`-Teil der Klassendefinition steht. Dies ist natürlich nicht im Sinne der gewünschten Kapselung, und wird deshalb meistens durch eine `private:`- oder `protected:`-Deklaration unterbunden.

Weiterhin kann jede Methode das Element mit seinem Namen ansprechen, ebenso wie ein nichtstatisches Element.

Zur Initialisierung von statisch deklarierten Elementen gibt es zudem die Möglichkeit, sie wie eine normale statische Variable außerhalb der `class`-Anweisung zu initialisieren; man muß nur den Klassennamen mit angeben, um dem Compiler zu sagen, welche Klasse man meint:

```
int NeuerTyp::i   = 25;
```

Diese Initialisierung darf ebenso wie bei statischen Variablen nur in einem der Quelltexte vorkommen, aus denen das Programm zusammengesetzt wird. Die Initialisierung findet wie bei allen statischen Variablen bei Programmstart statt, bevor `main()` aufgerufen wird.
Die gezeigte Initialisierung ist auch zulässig, wenn das Element im `private:`- oder `protected:`-Bereich einer Klasse steht.
An dieser Stelle ist es übrigens nicht erlaubt, das Element nochmals als `static` zu deklarieren[3].

Diese Art der Initialisierung ist sicher besser, als eine Initialisierung innerhalb des Konstruktors der Klasse, und zwar aus zwei Gründen:

1. Der Konstruktor wird nicht nur genau einmal aufgerufen, sondern wahrscheinlich mehrfach (einmal je Instanz der Klasse). Weil aber ein statisches Element nur einmal existiert in jedem Programmlauf, ist eine mehrfache Initialisierung Zeitverschwendung (und meistens auch unerwünscht, weil sich der Wert ja mit der Zeit meistens ändern soll).
2. Vor dem ersten Erzeugen eines Objekts ist das statische Element nicht mit einem definierten Wert belegt, weil ja noch gar kein Objekt angelegt wurde.

Eine mögliche Verwendung von solchen statischen Elementen ist ein Zähler, wieviele Objekte einer Klasse existieren:

[3] Grund: `static` hat zwei verschiedene Bedeutungen. Innerhalb von Klassen und Funktionen bedeutet es „*diese Variable ist statisch*“, außerhalb (also für globale Namen) verhindert es die Sichtbarkeit eines Namens von anderen Quelltexten aus (siehe [KW:0 ist false]). Letztere Bedeutung hätte es hier, und das wäre natürlich falsch.

```
#include <iostream>

class A
{
private:

  static size_t nInstanzen;  // für die gesamte Klasse: Anzahl Objekte

public:

  // Konstruktor:
  A()
  {
    nInstanzen++;   // eine Instanz mehr
  }

  // Destruktor:
  ~A()
  {
    nInstanzen--;   // eine Instanz weniger
  }


  // liefert die Anzahl der gerade aktiven Objekte:
  static size_t   zeigeAnzahl() { return nInstanzen; }

};

size_t A::nInstanzen = 0;

int main( int nargs, char **args )
{

  A a, b; // ab hier gibt es 2 Objekte

  {
    A c;  // jetzt gibt es 3 Objekte
    cout << A::zeigeAnzahl() << " Objekte" << endl;
  } // jetzt wieder nur 2 Objekte

  cout << A::zeigeAnzahl() << " Objekte" << endl;

} // main( int nargs, char **args )
```

Dieses Programm liefert als Ausgabe:

```
3 Objekte
2 Objekte
```

Eine andere Möglichkeit wäre es, jedem Objekt eine „Seriennummer“ zu geben:

```
class A
{
private:

  static size_t NextSerial; // für gesamte Klasse: nächste freie Nummer

  size_t Serial;            // jedes Objekt: Seriennummer

public:

  // Konstruktor:
  A()
  {
    Serial = NextSerial++;
  }

  // Destruktor:
  ~A()
  {
  }

};

size_t A::NextSerial = 1;
```

So bekommen alle Objekte eine fortlaufende Nummer (`Serial`).

Statische Elemente gehören also zur Klasse, nichtstatische Elemente gehören zu einem Objekt.

4.12.5 Statische und nichtstatische Methoden

Wenn eine Methode ausschließlich auf `static`-Elemente ihrer Klasse zugreift, dann kann (und sollte) man sie als `static` deklarieren:

```
class A
{
private:

  static int i;

public:

  static void setzei( int neuerwert ) { i = neuerwert; }
};

int A::i = 12;

// ...

  A::setzei( 13 );
```

Dann kann man solche Methoden (weil sie ja gar keine Daten benutzen, die zu einer bestimmten Instanz gehören) auch einfach über ihren Klassennamen aufrufen, ohne überhaupt ein Objekt der Klasse zu kennen, hier im Beispiel mit `A::setzei( 13 )`.

Logischerweise kann man in statischen Methoden den Zeiger `this` (**4.12.2 Der Zeiger `this`** auf Seite 58) nicht benutzen.

Statische Methoden kennen nur die statischen Elemente einer Klasse, und brauchen kein Objekt zum Aufruf.

4.12.6 `const`-Methoden

Wenn eine Methode die Elemente der Klasse nur lesend anfaßt, dann kann und sollte sie als `const` deklariert werden; dies geschieht durch die Angabe des Schlüsselworts `const` hinter der schließenden Klammer der Parameterliste:

```
class A
{
private:

  int i;

public:

  int get_i() const { return i; }
  // ...
};
```

Damit garantiert man, daß der Zustand eines Objekts durch den Aufruf der Methode nicht geändert wird.

Dies ist zum einen praktisch, weil der Compiler eine versehentliche Änderung eines Elements als Fehler erkennen kann:

```
  // ...
  int get_i() const { return i++; }  // Fehlermeldung!
  // ...
};
```

Zum anderen sind nur als `const` deklarierte Funktionen anwendbar, wenn ein konkretes Objekt als `const` deklariert ist:

```
void up( const A parameter )
{
  cout << parameter.get_i() << endl;
}
```

Da hier `parameter` als `const` angegeben ist, dürfen nur Methoden darauf angewendet werden, die ebenfalls durch `const` versichern, den Zustand von `parameter` nicht zu ändern.

Eine `const`-Methode kann sowohl auf `const`-Objekte als auch auf nicht-`const`-Objekte angewendet werden; eine nicht-`const`-Methode dagegen nur auf nicht-`const`-Objekte.

Ob eine Methode als `const` deklariert ist oder nicht, ist Teil ihrer Signatur, die sie von anderen überladenen Methoden unterscheidet. Deshalb kann man zwei sonst identische überladene Methoden definieren, die sich nur bezüglich `const` unterscheiden.

Da sich durch `const` eine andere Funktion ergibt, muß man bei einer Definition außerhalb der Klasse dort das Schlüsselwort `const` wiederholen:

```
class A
{
private:

  int i;

public:

  int get_i() const;
  // ...
};

int A::get_i() const { return i; }
```

Soweit die Theorie. Manchmal kommt es aber vor, daß man eine Funktion als `const` deklarieren möchte, um sie auf `const`-Elemente anwenden zu dürfen, aber die Funktion ändert tatsächlich ein Element des Objekts oder einige wenige, ohne daß sich der Zustand des Objekts aus der Sicht des Anwenders ändert (beispielsweise soll ein Zugriffszähler weitergesetzt werden). In diesem Fall ist das Objekt nicht *physisch* konstant; es ist aber aus der Sicht des Anwenders *logisch* konstant. Dann kann man die zu ändernden Elemente der Klasse als `mutable` deklarieren. Das bedeutet, daß diese Elemente auch bei `const`-Objekten nicht wirklich konstant sein müssen, sondern auch von `const`-Methoden geändert werden dürfen. (Eine Alternative zu dieser Vorgehensweise wäre, einen Datentyp zu schaffen, in dem alle veränderlichen Bestandteile enthalten sind, und in dem möglicherweise konstanten Objekt einen Zeiger auf den veränderlichen Teil zu hinterlegen, der im Konstruktor allokiert und im Destruktor freigegeben wird.)

4.12.7 Ableiten von Klassen

Man kann eine neue Klasse auf der Basis einer bereits vorhandenen Klasse bilden.

Die bereits vorhandene Klasse heißt dann *Basisklasse*, die neue Klasse ist eine *Ableitung* der Basisklasse.

Dies ist natürlich dann sinnvoll, wenn die neue Klasse „im Prinzip ähnlich ist“, und nur teilweise andere Eigenschaften oder Erweiterungen haben soll.

Die Datenelemente und Methoden der Basisklasse sind dann in der abgeleiteten Klasse ebenso enthalten, und können ergänzt oder –falls gewünscht– durch andere ersetzt werden.

Beispiel:

```
class Auto
{
protected:

  int    AnzahlRaeder;
  double Motorleistung;

public:

  // Konstruktor:
  Auto( double LeistungKW = 0.0, int nRad = 4 )
  {
    Motorleistung = LeistungKW;
    AnzahlRaeder = nRad;
  }

  // liefert die Anzahl Räder:
  int getRaeder()
  {
    return AnzahlRaeder;
  }
};

class Laster: public Auto
{
protected:

  double Zuladung;

public:

  // Konstruktor:
  Laster( double LeistungKW = 100.0,
```

```
          int nRad = 6,
          double Last = 2000.00
          )
  {
    Motorleistung = LeistungKW;
    AnzahlRaeder = nRad;
    Zuladung = Last;
  }

};

int main( int nargs, char **args )
{
  Auto   Golf; // 4 Räder, keine Leistung
  Auto   Ferrari( 250.0 ); // 4 Räder, 250kW

  // hat 100kW, 6 Räder, 2000.0kg Nutzlast:
  Laster Buessing;

  // hat 350.0kW, 10 Räder, 18000.0kg Nutzlast:
  Laster MAN( 350.0, 10, 18000.0 );

  cout << "Golf hat "  <<  Golf.getRaeder() << " Räder" << endl;
  cout << "MAN hat "  <<  MAN.getRaeder() << " Räder" << endl;
}
```

Die grundsätzliche Vorgehensweise beim Ableiten ist, daß man bei der neuen Klasse hinter dem Klassennamen mit einem Doppelpunkt getrennt die Basisklasse angibt (nach einem `public` oder `private`; zum Unterschied zwischen den beiden Möglichkeiten im nächsten Abschnitt mehr).

Damit enthält die abgeleitete Klasse alle Elemente und Funktionen, die in der Basisklasse bereits angelegt sind, und kann mit ihren Methoden auf alles zugreifen, was in der Basisklasse als `public:` oder `protected:` definiert ist.

Diese Übernahme von Elementen und Funktionen heißt *Vererbung*.

Ein Objekt der so abgeleiteten Klasse ist sowohl ein Objekt der abgeleiteten Klasse, als auch gleichzeitig eines der Basisklasse: *ein Laster ist gleichzeitig auch ein Auto*.

Umgekehrt gilt das nicht; der Golf des obigen Beispiels bekommt die Eigenschaft

Zuladung nicht durch die Ableitung.

In dem obigen Beispiel erbt `Laster` von `Auto` die Elemente `AnzahlRaeder` und `Motorleistung` und definiert sich zusätzlich noch `Zuladung`.

Ebenso wird die Funktion `getRaeder()` geerbt, während der Konstruktor hier durch einen neuen ersetzt wird.

Durch den Vererbungsmechanismus kann ein Laster die Funktion `getRaeder()` der Basisklasse mitnutzen.

Daß ein `Laster` auch gleichzeitig ein `Auto` ist, hat zur Folge daß man problemlos überall wo ein `Auto` stehen darf, auch einen `Laster` verwenden darf:

```
void Machwas( Auto a )
{
  //...
}
//...
  Laster   Tatra;
  Auto     meineKarre;
  Machwas( Tatra );
  meineKarre = Tatra;
```

Der Compiler kann bei Bedarf jederzeit von einer abgeleiteten Klasse zu einer Basisklasse casten (gegebenenfalls auch implizit). Dies geht auch über mehrere Ableitungsstufen hinweg.

Natürlich kann die Funktion `Machwas()` nicht auf das `Laster`-Element `Zuladung` zugreifen, weil der Compiler an dieser Stelle nur von einem `Auto` ausgehen kann.

Ebenso wird bei der Zuweisung eines `Laster` an ein `Auto` das Element `Zuladung` „abgestreift“.

Wenn dieses „Abstreifen“ ein Problem ist, dann steht die Lösung in **4.12.7.4 Virtuelle Methoden** auf Seite 87.

4.12.7.1 `public`, `protected` oder `private` ableiten

Im einführenden Beispiel des vorigen Abschnitts wurde `Laster` von `Auto` *public* abgeleitet, denn die Definition von `Laster` begann folgendermaßen:

```
//...
class Laster: public Auto
{
//...
```

Im Gegensatz dazu könnte man auch *protected* ableiten, also so:

```
//...
class Laster: protected Auto
{
//...
```

oder auch *private*:

```
//...
class Laster: private Auto
{
//...
```

Worin unterscheiden sich diese Varianten?

Sowohl bei einer *public*-Ableitung, *protected*-Ableitung, als auch einer *private*-Ableitung erbt die ableitende Klasse (hier: `Laster`) alle Elemente der Basisklasse, und kann –je nach deren Deklaration als `private:`, `protected:` oder `public:` in der Basisklasse– darauf zugreifen. Insoweit besteht kein Unterschied zwischen den Varianten.

Ein Unterschied kommt erst dann zum Vorschein, wenn Anwender der abgeleiteten Klasse (`Laster`) versuchen, auf `Auto`-Elemente zu zugreifen:

- Durch die *private*-Ableitung werden alle `public`- und alle `protected`-Elemente der Basisklasse für den Anwender der abgeleiteten Klasse nicht mehr sichtbar; sie erscheinen also außerhalb von `Laster` so, als ob alle Elemente in `Auto` als `private` deklariert worden wären. Im Beispiel könnte man demnach für `Laster`-Objekte die Funktion `getRaeder()` nicht mehr aufrufen (außerhalb der Klasse `Laster`). Weiterhin könnte eine von `Laster` abgeleitete Klasse (beispielsweise `Sattelschlepper`) auf kein Element von `Auto` mehr zugreifen.

- Bei einer *protected*-Ableitung tritt ebenfalls für alle Anwender von `Laster` eine Verschiebung der Zugriffsrechte der Basisklasse ein, allerdings etwas milder: alle `public`-Elemente der Basisklasse erscheinen bei der Anwendung der abgeleiteten Klasse wie `protected`. Damit kann über `Laster`-Objekte nicht mehr auf `Auto`-Elemente zugegriffen werden, aber eine von `Laster` abgeleitete Klasse `Sattelschlepper` kann diese Elemente noch nutzen.

- Bei einer *public*-Ableitung werden die Zugriffsrechte der Basisklasse unverändert übernommen für alle Anwender der abgeleiteten Klasse.

  Bei einer *public* geerbten Schnittstelle bleibt also für weiter davon abgeleitete Klassen die gesamte `public`- und `protected`-Schnittstelle der Basisklasse erhalten.

(In jedem Fall sind in den abgeleiteten Klassen aber alle Elemente der Basisklassen enthalten, auch wenn nur beschränkt darauf zugegriffen werden kann. Eine Klasse kann also durch Ableiten nicht schrumpfen, sondern nur erweitert werden.)

In den meisten Fällen wird man *public* ableiten. Allerdings kann es sein, daß eine neue Klasse zwar eine Basisklasse sinnvoll intern verwenden kann, aber zu seinem Anwender wiederum eine derart stark geänderte Schnittstelle hat, daß ein Durchgriff auf die Basisklasse nicht mehr sinnvoll ist. Dann wird man *private* ableiten. (Man sollte sich dann allerdings die Frage stellen, ob wirklich noch eine „IST EIN“-Beziehung zwischen den Klassen plausibel ist, oder nicht eher eine „HAT EIN“-Beziehung; im letzteren Fall sollte man gar nicht ableiten, sondern statt dessen ein Element vom Typ der zugrunde liegenden Klasse einfügen.)

Wenn man den Durchgriff auf einzelne Elemente dennoch ermöglichen möchte, dann kann man das mit einer Deklaration in der abgeleiteten Klasse etwa in der Art tun:

```
//...
class Laster: private Auto
{
public:
  Auto::getRaeder;
/...
```

Angenommen, man hat eine kleine Klasse `A` mit einem `public`-, einem `protected`- und einem `private`-Teil, dann kann eine Methode der Klasse auf alle Elemente zugreifen:

```
class A
{
private:
  int privA;

protected:
  int protA;

public:
  int publA;

  virtual void methode( int wert )
  {
    privA = wert; // ok
    protA = wert; // ok
    publA = wert; // ok
  }
};
```

Davon werden drei Klassen (`BprivA`, `BprotA`, `BpublA`) abgeleitet, je eine `private`, eine `protected` und eine `public`; sowie von diesen dreien wieder je eine `CpubB`... (und zwar `public`). In den dabei enthaltenen Methoden sind die unzulässigen Zugriffe auskommentiert:

```
// von A private erben:
class BprivA: private A
{
public:
  virtual void methode( int wert )
  {
    //privA = wert; // geht nicht, weil private
    protA = wert; // ok
    publA = wert; // ok
  }
};
```

```
class CpublBprivA: public BprivA
{
public:
  virtual void methode( int wert )
  {
    //privA = wert; // geht nicht, weil private
    //protA = wert; // geht nicht, weil private von A geerbt
    //publA = wert; // geht nicht, weil private von A geerbt
  }
};


// von A protected erben:
class BprotA: protected A
{
public:
  virtual void methode( int wert )
  {
    //privA = wert; // geht nicht, weil private
    protA = wert; // ok
    publA = wert; // ok
  }
};


class CpublBprotA: public BprotA
{
public:
  virtual void methode( int wert )
  {
    //privA = wert; // geht nicht, weil private
    protA = wert; // ok
    publA = wert; // ok
  }
};


// von A public erben:
class BpublA: public A
{
public:
  virtual void methode( int wert )
  {
```

```
    //privA = wert; // geht nicht, weil private
    protA = wert; // ok
    publA = wert; // ok
  }
};

class CpublBpublA: public BpublA
{
public:
  virtual void methode( int wert )
  {
    //privA = wert; // geht nicht, weil private
    protA = wert; // ok
    publA = wert; // ok
  }
};
```

Die Zugriffsmöglichkeiten über Objekte der einzelnen Klassen sieht man hier (unzulässige Zugriffe sind wieder auskommentiert):

```
int main( int nargs, char **args )
{
  A einA;
  //einA.privA = 12;, // geht nicht, weil private
  //einA.protA = 12;, // geht nicht, weil protected
  einA.publA = 12; // ok



  BprivA einBprivA;
  //einBprivA.privA = 12; // g.n., w. priv. && priv. von A geerbt
  //einBprivA.protA = 12; // g.n., w. prot. && priv. von A geerbt
  //einBprivA.publA = 12; // g.n., w. priv. von A geerbt

  CpublBprivA einCpublBprivA;
  //einCpublBprivA.privA = 12; // g.n., w. priv. && priv. von A geerbt
  //einCpublBprivA.protA = 12; // g.n., w. prot. && priv. von A geerbt
  //einCpublBprivA.publA = 12; // g.n., w. priv. von A geerbt
```

```
  BprotA einBprotA;
  //einBprotA.privA = 12; // g.n., w. priv. && prot. von A geerbt
  //einBprotA.protA = 12; // g.n., w. prot. && prot. von A geerbt
  //einBprotA.publA = 12; // g.n., w. prot. von A geerbt

  CpublBprotA einCpublBprotA;
  //einCpublBprotA.privA = 12; // g.n., w. priv. && prot. von A geerbt
  //einCpublBprotA.protA = 12; // g.n., w. prot. && prot. von A geerbt
  //einCpublBprotA.publA = 12; // g.n., w. prot. von A geerbt




  BpublA einBpublA;
  //einBpublA.privA = 12; // geht nicht, weil private
  //einBpublA.protA = 12; // geht nicht, weil protected
  einBpublA.publA = 12; // ok

  CpublBpublA einCpublBpublA;
  //einCpublBpublA.privA = 12; // g.n., weil private
  //einCpublBpublA.protA = 12; // g.n., weil protected
  einCpublBpublA.publA = 12; // ok
}
```

Die wesentlichen Zugriffsmöglichkeiten lassen sich auch in einer Tabelle darstellen. Dabei wird wieder von der obigen Klasse `A` ausgegangen:

Zugriff auf über:	`privA`	`protA`	`publA`
Methode in `A`:	ja	ja	ja
Objekt vom Typ `A`:	nein	nein	ja
Falls `B: private A` abgeleitet und `C: public B`			
Methode in `B`:	nein	ja	ja
Objekt vom Typ `B`:	nein	nein	nein
Methode in `C`:	nein	nein	nein
Objekt vom Typ `C`:	nein	nein	nein
Falls `B: protected A` abgeleitet und `C: public B`			
Methode in `B`:	nein	ja	ja
Objekt vom Typ `B`:	nein	nein	nein
Methode in `C`:	nein	ja	ja
Objekt vom Typ `C`:	nein	nein	nein
Falls `B: public A` abgeleitet und `C: public B`			
Methode in `B`:	nein	ja	ja
Objekt vom Typ `B`:	nein	nein	ja
Methode in `C`:	nein	ja	ja
Objekt vom Typ `C`:	nein	nein	ja

4.12.7.2 Aufruf des Konstruktors der Basisklasse

Im gezeigten Beispiel werden die Elemente `AnzahlRaeder` und `Motorleistung` sowohl im Konstruktor von `Auto` als auch im Konstruktor von `Laster` direkt gesetzt.

Dies ist in vielen Fällen unerwünscht oder gar nicht möglich, weil

- die Initialisierung wesentlich aufwendiger sein kann als nur eine Zuweisung wie hier, und dann umfangreicher Code mehrfach geschrieben werden muß (was Arbeit macht und Fehler induziert, sowie Änderungen erschwert)
- der Programmierer der abgeleiteten Klasse sich vielleicht gar nicht für die Interna der Basisklasse interessieren will (oder darf)
- ein direkter Zugriff auf die Elemente der Basisklasse durch `private:` unterbunden ist

Implizit wird vor dem Code des `Laster`-Konstruktors ohnehin der Defaultkonstruktor der Basisklasse `Auto` ausgeführt. Man kann gezielt auch einen anderen Konstruktor wählen, indem man die Klasse `Laster` etwas ändert:

```
//...
  // Konstruktor:
  Laster( double LeistungKW = 0.0,
          int nRad = 6,
          double Last = 100.00
          )
    : Auto( LeistungKW, nRad )
  {
    Zuladung = Last;
  }
//...
```

In einer abgeleiteten Klasse kann man nach der Parameterliste des Konstruktors (mit einem Doppelpunkt getrennt) einen Konstruktor der Basisklasse mit den zugehörigen Parametern angeben, und verzichtet stattdessen auf die entsprechenden direkten Initialisierungen im Funktionsrumpf.

Wenn man den Konstruktor in der Klasse nur deklariert, und an anderer Stelle definiert, dann gibt man den Basiskonstruktor bei der Implementation an (also außerhalb der Klasse).

4.12.7.3 Initialisierung der Klassenelemente

Im letzten Abschnitt wurde gezeigt, wie die Elemente der zugrundeliegenden Basisklasse über einen gezielten Konstruktoraufruf initialisiert werden können.

Ähnliches gilt für die eigenen Elemente einer Klasse, wenn diese wiederum Objekte sind (also nicht eine „IST EIN"-Beziehung vorliegt, sondern eine „HAT EIN"-Beziehung).

Beispiel: Ein `Computer` soll aus einer CPU vom Typ `Prozessor`, einer HD vom Typ `Festplatte` und einer Arbeitsspeicherkapazität `SpeicherMB` vom Typ `int` bestehen.
Dann könnte das etwa so aussehen:

```
class Prozessor
{
private:

  int TaktrateMHz;
  int Busbreite;

public:

  Prozessor( int TaktMHz = 8, int Bus = 16 )
  {
    TaktrateMHz = TaktMHz;
    Busbreite = Bus;
  }
};

class Festplatte
{
private:

  int KapazitaetMB; // MB Speicherplatz

public:

  Festplatte( int KapMB )
  {
    KapazitaetMB = KapMB;
  }
```

```
};

class Computer
{
private:

  Prozessor  CPU;
  Festplatte HD;
  int        SpeicherMB; // MB Arbeitsspeicher

public:

  Computer( int TaktMHz, int Bus, int KapMB, int RAM )
    : CPU( TaktMHz, Bus ), // Initialisierung CPU
      HD( KapMB )          // Initialisierung HD
  {
    SpeicherMB = RAM;      // Initialisierung SpeicherMB
  }
};
//...
  Computer  MeinPC( 1400, 32, 18000, 512*1024 );
```

Man gibt also die Initialisierungen der einzelnen Komponenten ähnlich an wie den Aufruf von Basiskonstruktoren; allerdings nicht mit dem Typnamen, sondern dem Elementnamen stattdessen.

Wenn mehrere Initialisierungen nötig sind, trennt man die einzelnen mit Kommata.

Wenn beides nötig ist (Initialisierung der Basisklasse und Initialisierung von Elementen), dann gibt man halt auch beides an:

```
//...
class Computer: Elektronikbaugruppe  // jetzt abgeleitet!
{
private:

  Prozessor  CPU;
  Festplatte HD;
  int        SpeicherMB; // MB Arbeitsspeicher

public:
```

```
  Computer( int TaktMHz, int Bus, int KapMB, int RAM )
    : Elektronikbaugruppe( TaktMHz ), // Init. Basisklasse
      CPU( TaktMHz, Bus ), // Initialisierung CPU
      HD( KapMB )          // Initialisierung HD
  {
    SpeicherMB = RAM;      // Initialisierung SpeicherMB
  }
};
//...
```

Dabei wird immer erst die Initialisierung der Basisklasse(n) ausgeführt, dann die der Elemente (in der angegebenen Reihenfolge). Erst anschließend erfolgt die Abarbeitung des Funktionsrumpfes des Konstruktors.

4.12.7.4 Virtuelle Methoden

Für die kommenden Beispiele wird angenommen, man habe sich die in Bild 4.1 gezeigte Klassenhierarchie aufgebaut.

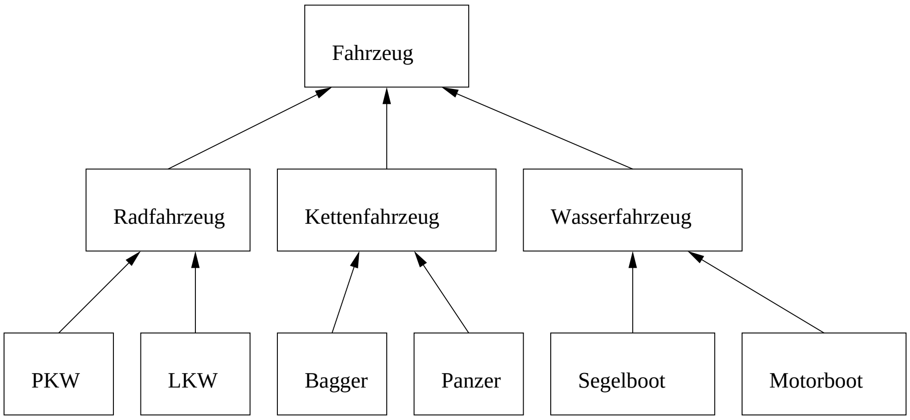

Abbildung 4.1: Klassenhierarchie `Fahrzeug`

Ausgehend von einer allgemeinen Klasse `Fahrzeug` wird relativ grob unterteilt in `Radfahrzeug`, `Kettenfahrzeug`, und `Wasserfahrzeug`.

Diese Klassen werden noch weiter verfeinert; beispielsweise werden Radfahrzeuge in `PKW` und `LKW` unterteilt.

Jede Klasse soll die Fähigkeit haben, mit der Methode `zeigdich()` ihren Inhalt zur Standardausgabe (`cout`) zu schreiben.

Nach dem bisher gesagten könnte die Hierarchie (in Auszügen) etwa so implementiert werden (es werden nur die Klassen `Fahrzeug`, `Radfahrzeug` und `PKW` gezeigt, sowie ein kleines Testprogramm):

```
// Time-stamp: "31.10.03 21:09 fahrzeuge0.cpp klaus@wachtler.de"

#include <iostream>
#include <string>

using namespace std;

// Basisklasse für alle Fahrzeuge:
class Fahrzeug
{
protected:
  string     Typbezeichnung;

public:

  // Konstruktor
  Fahrzeug( string Typbezeichnung = "?" )
  {
    this->Typbezeichnung = Typbezeichnung;
  }

  void zeigdich() const
  {
    cout << " Fahrzeug::zeigdich():";
    cout << Typbezeichnung << endl;
  }

};

// Vom Typ Fahrzeug wird Radfahrzeug abgeleitet:
class Radfahrzeug: public Fahrzeug
{
protected:

  int AnzahlRaeder;

public:
```

```
  // Konstruktor
  Radfahrzeug( string Typbezeichnung = "?",
               int AnzahlRaeder = 4
            )
    : Fahrzeug( Typbezeichnung )
  {
    this->AnzahlRaeder = AnzahlRaeder;
  }

  void zeigdich() const
  {
    cout << " Radfahrzeug::zeigdich():";
    cout << Typbezeichnung
         << ", "
         << AnzahlRaeder
         << " Räder"
         << endl;
  }

};

// Von Radfahrzeug wird ein PKW abgeleitet:
class PKW: public Radfahrzeug
{
protected:

  int Sitzplaetze;

public:

  // Konstruktor:
  PKW( string Typbezeichnung = "?",
       int AnzahlRaeder = 4,
       int Sitzplaetze = 4
       )
    : Radfahrzeug( Typbezeichnung, AnzahlRaeder )
  {
    this->Sitzplaetze = Sitzplaetze;
  }

  void zeigdich() const
  {
    cout << " PKW::zeigdich():";
```

```
    cout << Typbezeichnung
         << ", "
         << AnzahlRaeder
         << " Räder"
         << ", "
         <<  Sitzplaetze
         << " Sitzplätze"
         << endl;
  }
};



// nur zum Probieren:


void tuwas( Fahrzeug f )
{
  f.zeigdich();
}

int main( int nargs, char **args )
{
  Fahrzeug EinHoovercraft( "die gute Wahl, Hoover" );
  Radfahrzeug EinRadlader( "Bagger auf Rädern", 4 );
  PKW Golf( "VW Golf" );
  PKW A6( "Audi A6", 4, 5 );

  cout << "Direkte Ausgabe:" << endl;

  EinHoovercraft.zeigdich();
  EinRadlader.zeigdich();
  Golf.zeigdich();
  A6.zeigdich();

  cout << "Ausgabe mit tuwas():" << endl;

  tuwas( EinHoovercraft );
  tuwas( EinRadlader );
  tuwas( Golf );
  tuwas( A6 );

} /* main( int nargs, char **args ) */
```

Die Ausgabe davon sieht folgendermaßen aus:

```
Direkte Ausgabe:
 Fahrzeug::zeigdich():die gute Wahl, Hoover
 Radfahrzeug::zeigdich():Bagger auf Rädern, 4 Räder
 PKW::zeigdich():VW Golf, 4 Räder, 4 Sitzplätze
 PKW::zeigdich():Audi A6, 4 Räder, 5 Sitzplätze
Ausgabe mit tuwas():
 Fahrzeug::zeigdich():die gute Wahl, Hoover
 Fahrzeug::zeigdich():Bagger auf Rädern
 Fahrzeug::zeigdich():VW Golf
 Fahrzeug::zeigdich():Audi A6
```

In dem Testprogramm werden Variablen von den Typen `Fahrzeug`, `Radfahrzeug` und `PKW` definiert, und ihr Inhalt direkt ausgegeben,, sowie über eine Funktion, die ein `Fahrzeug` erhält.

Die direkte Ausgabe (beispielsweise `A6.zeigdich();`) ist befriedigend; es werden alle Elemente ausgegeben:

- für ein `Fahrzeug` wird mit der Funktion `Fahrzeug::zeigdich()` nur die Bezeichnung ausgegeben
- für ein `Radfahrzeug` werden mit der Funktion `Radfahrzeug::zeigdich()` die Bezeichnung sowie die Anzahl Räder ausgegeben,
- für ein `Radfahrzeug` werden mit der Funktion `PKW::zeigdich()` die Bezeichnung, die Anzahl Räder ausgegeben, sowie die Anzahl der Sitzplätze ausgegeben.

Die Ausgabe über die Funktion `tuwas( Fahrzeug f )` dagegen ist dürftig: es wird immer nur die Bezeichnung ausgegeben; egal, welches Fahrzeug man übergibt.

Das Problem liegt natürlich darin, daß zum Aufruf der Funktion jedes Fahrzeug vom Compiler implizit in ein temporäres Objekt vom Typ `Fahrzeug` gecastet wird; dabei gehen alle zusätzlichen Elemente von `Radfahrzeug` und `PKW` verloren. In der Funktion wird dann für diese temporäre abgemagerte Kopie die Funktion `Fahrzeug::zeigdich()` aufgerufen.

Wünschenswert wäre es dagegen, wenn die Funktion `tuwas( Fahrzeug f )` ein komplettes Fahrzeug ausgeben könnte, ohne die zusätzlichen Elemente eines Objekts einer abgeleiteten Klasse abzustreifen.

Dazu muß man zwei Dinge ändern:

1. es muß verhindert werden, daß bei der Übergabe eines von `Fahrzeug` abgeleiteten Objekts die implizite Wandlung in die Basisklasse `Fahrzeug` stattfindet.

   Dies kann man leicht erreichen, indem man den *call by value* (Wertübergabe) durch ein *call by reference* ersetzt. Man muß also entweder eine Referenz des auszugebenden Werts übergeben, oder einen Zeiger darauf; dann wird vom Compiler keine temporäre Kopie vom Typ `Fahrzeug` angelegt. Vielmehr bleibt das komplette Objekt erhalten.

2. Man muß dem Compiler mitteilen, daß die in `Fahrzeug` definierte Funktion `zeigdich()` nur für diese Klasse gilt, und abgeleitete Klassen diese Funktion überschreiben werden.

Für diesen Hinweis gibt es das Schlüsselwort `virtual`. Damit muß man in der Basisklasse (und wahlweise in den Ableitungen) alle Methoden deklarieren, die beim Ableiten überschrieben werden sollen:

```
// Time-stamp: "31.10.03 21:09 fahrzeuge1.cpp klaus@wachtler.de"

#include <iostream>
#include <string>

using namespace std;

// Basisklasse für alle Fahrzeuge:
class Fahrzeug
{
protected:
  string     Typbezeichnung;

public:

  // Konstruktor
  Fahrzeug( string Typbezeichnung = "?" )
  {
    this->Typbezeichnung = Typbezeichnung;
  }

  virtual ~Fahrzeug()
  {
```

```
  }

  virtual void zeigdich() const
  {
    cout << " Fahrzeug::zeigdich():";
    cout << Typbezeichnung << endl;
  }

};

// Vom Typ Fahrzeug wird Radfahrzeug abgeleitet:
class Radfahrzeug: public Fahrzeug
{
protected:

  int AnzahlRaeder;

public:

  // Konstruktor
  Radfahrzeug( string Typbezeichnung = "?",
               int AnzahlRaeder = 4
             )
    : Fahrzeug( Typbezeichnung )
  {
    this->AnzahlRaeder = AnzahlRaeder;
  }

  void zeigdich() const
  {
    cout << " Radfahrzeug::zeigdich():";
    cout << Typbezeichnung
         << ", "
         << AnzahlRaeder
         << " Räder"
         << endl;
  }

};

// Von Radfahrzeug wird ein PKW abgeleitet:
class PKW: public Radfahrzeug
{
```

```
protected:

  int Sitzplaetze;

public:

  // Konstruktor:
  PKW( string Typbezeichnung = "?",
       int AnzahlRaeder = 4,
       int Sitzplaetze = 4
       )
    : Radfahrzeug( Typbezeichnung, AnzahlRaeder )
  {
    this->Sitzplaetze = Sitzplaetze;
  }

  void zeigdich() const
  {
    cout << " PKW::zeigdich():";
    cout << Typbezeichnung
         << ", "
         << AnzahlRaeder
         << " Räder"
         << ", "
         <<  Sitzplaetze
         << " Sitzplätze"
         << endl;
  }
};



// nur zum Probieren:


void tuwas( const Fahrzeug &f )  // Parameter jetzt als Referenz!
{
  f.zeigdich();
}

int main( int nargs, char **args )
{
  Fahrzeug EinHoovercraft( "die gute Wahl, Hoover" );
```

```
  Radfahrzeug EinRadlader( "Bagger auf Rädern", 4 );
  PKW Golf( "VW Golf" );
  PKW A6( "Audi A6", 4, 5 );

  cout << "Direkte Ausgabe:" << endl;

  EinHoovercraft.zeigdich();
  EinRadlader.zeigdich();
  Golf.zeigdich();
  A6.zeigdich();

  cout << "Ausgabe mit tuwas():" << endl;

  tuwas( EinHoovercraft );
  tuwas( EinRadlader );
  tuwas( Golf );
  tuwas( A6 );

} /* main( int nargs, char **args ) */
```

(Wenn mindestens eine Funktion `virtual` deklariert ist, dann muß der Destruktor ebenfalls `virtual` deklariert werden[4].)

Als Dank dafür kann `tuwas( Fahrzeug f )` die jeweils richtige Funktion `zeigdich()` aufrufen, ohne den genauen Typ zu kennen (dies funktioniert auch, wenn zum Zeitpunkt des Kompilierens von `tuwas( Fahrzeug f )` die abgeleiteten Klassen noch gar nicht geschrieben sind!):

```
Direkte Ausgabe:
 Fahrzeug::zeigdich():die gute Wahl, Hoover
 Radfahrzeug::zeigdich():Bagger auf Rädern, 4 Räder
 PKW::zeigdich():VW Golf, 4 Räder, 4 Sitzplätze
 PKW::zeigdich():Audi A6, 4 Räder, 5 Sitzplätze
Ausgabe mit tuwas():
 Fahrzeug::zeigdich():die gute Wahl, Hoover
```

[4]Mit dieser Regel liegt man auf der sicheren Seite; etwas genauer lautet sie laut [Cline: C++ FAQs]: „if any derived class (or any data member and/or base class of any derived class, or any base class of any data member of any derived class, or any data member of any base class of any data member of any derived class and all other recursive combinations of base classes and data members) has (or will ever have) a nontrivial destructor, and if any code anywhere deletes (or will ever delete) that derived class object via a base class pointer, then the base class's destructor needs to be virtual“. Ich bevorzuge die kurze Regel.

```
Radfahrzeug::zeigdich():Bagger auf Rädern, 4 Räder
PKW::zeigdich():VW Golf, 4 Räder, 4 Sitzplätze
PKW::zeigdich():Audi A6, 4 Räder, 5 Sitzplätze
```

Weil die Bindung des Aufrufs von `zeigdich()` an eine Methode des aktuellen Objekts offensichtlich nicht beim Kompilieren stattfindet, sondern erst zur Laufzeit, heißt der Mechanismus *late binding* oder *dynamic binding*. Sofern möglich, wird ein *C++*-Compiler immer *early binding* (oder *static binding*) durchführen; weil *early binding* etwas Rechenzeit kostet. Im Falle von *early binding* kann der Compiler beim Aufrufer bereits die Adresse der aufzurufenden Funktion einsetzen. Dagegen wird für *late binding* für jede Klasse ein Feld mit den Adressen der zugehörigen virtuellen Funktionen angelegt, und jedes Objekt kennt dieses Feld, die sogenannte *vtable*. Bei der Übergabe von Zeigern auf Objekte oder von Objekten per Referenz kann dann über diese vtable die jeweils richtige Methode gefunden werden.

[Strou IV] vertritt übrigens die Ansicht, daß nur solche virtuellen Funktionen *Methoden* heißen. Das ist Geschmackssache, aber in jedem Fall *sollen alle Methoden als* `virtual` *deklariert werden, wenn sie in Ableitungen möglicherweise überladen werden.*

Abgeleitete Klassen müssen eine virtuelle Methode nicht überschreiben, wenn sie sie entweder gar nicht benötigen, oder mit der ererbten Version zufrieden sind.

4.12.7.5 Virtuelle Konstruktoren

Da ein Konstruktor nie direkt aufgerufen wird, sondern nur beim Erzeugen eines neuen Objekts ein entsprechender Aufruf vom Compiler generiert wird, muß und kann ein Konstruktor nie virtuell sein. Denn aus dem in der Variablendefinition beziehungsweise bei `new` verwendeten Klassennamen geht eindeutig der Typ des zu erzeugenden Objekts hervor, und damit für den Compiler auch der zu verwendende Konstruktor. Der Compiler kennt die aufzurufende Funktion und verwendet statische Bindung; eine dynamische Bindung ist nicht nötig und auch nicht zulässig.

Trotzdem hat dieser Abschnitt seinen Sinn, denn es gibt Fälle wo eine Funktionen einen Zeiger oder eine Referenz auf ein Objekt `obj` von einem Typ `T` hat, aber nicht wissen kann, ob das Objekt wirklich ein `T` ist oder etwas davon abgeleitetes. Wenn die Funktion nun eine Kopie von `obj` benötigt, kann sie nicht einfach `new T(obj)` aufrufen; denn dann würde nur Speicher für ein `T` allokiert und der copy-Konstruktor der Basisklasse `T` aufgerufen werden. Wenn aber `obj` einen von `T` abgeleiteten Typ hat, müßte gegebenenfalls mehr allokiert werden, und der Konstruktor der abgeleiteten Klasse verwendet werden.

Dieses Problem tritt regelmäßig auf, wenn eine Klasse (beispielsweise `C`) entworfen wird, die als Container für irgendwelche Elemente vom Typ `T` (oder davon abgeleitete) dienen soll. In `C` wird dann ein Zeiger auf die Nutzdaten vom Typ `T` angelegt werden, der aber auch auf ein von `T` abgeleitetes Objekt zeigen darf. Zum Einfügen in den Container könnte man eine Methode `insert( T &init )` definieren, die von dem übergebenen Objekt `init` eine Kopie anlegen soll. Wie kann jetzt die Methode `C::insert( T &init )` ein neues Objekt beschaffen, das auch dann eine vollwertige Kopie von `init` ist, wenn `init` ein von `T` abgeleitetes Objekt ist? In `C::insert( T &init )` kann ein direkter Aufruf von `new` dies nicht leisten.

Für einen solchen Fall ist in der Klasse `T` eine virtuelle Methode vorzusehen, die selbst das `new` ausführt, und einen Zeiger beziehungsweise eine Referenz auf das erzeugte Objekt liefert. Diese Methode könnte als

```
virtual T&clone( T &init );
```

deklariert sein, und muß in den von `T` abgeleiteten Klassen individuell überschrieben werden.

Dann wird in `C::insert( T &init )` kein `new` mehr direkt verwendet, sondern stattdessen `clone()` aufgerufen:

```
class T
{
public:

  virtual T *clone() const
  {
    return new T( *this );
  }

  // ...

}; // class T

class TT
  : public T
{
public:

  virtual TT *clone() const
  {
    return new TT( *this );
  }
```

```
  // ...

};

class C
{
public:

  C()
  {
    pT = NULL;
  }

  void insert( const T &k )
  {
    delete pT;
    pT = k.clone();
  }

private:

  T  *pT;

  // ...

};

// ...
  T   einT;
  TT  einTT;

  C   c;
  c.insert( einT );  // führt zu T::clone() und damit zu new T
  c.insert( einTT ); // führt zu TT::clone() und damit zu new TT
```

Auch wenn die `clone()`-Methode in Wirklichkeit gar kein Konstruktor ist, sondern einen solchen nur indirekt über `new` aufruft, wird eine solche Funktion oft *virtueller Konstruktor* genannt. Ein anderer dafür manchmal verwendeter Name ist *class factory*, was aber m.E. Unfug ist, weil keine Klassen, sondern Objekte (einer Klasse) erzeugt werden.

Zu diesem Thema ist in **B.1.4 Ableitungen mit virtuellen Konstruktoren** auf Seite 200 eine vollständige Übungsaufgabe zu finden.

4.12.7.6 Konkrete und abstrakte Klassen

Die bisher vorgestellten Klassen sind alle *konkret*; das heißt man kann von diesem Typ auch Objekte erzeugen (als Variablen oder mit `new`).

Es kann allerdings wünschenswert sein, eine Klasse zu schaffen, die ausschließlich zum Ableiten verwendet werden kann, ohne daß von der Basisklasse Objekte erzeugt werden können. Die obige Beispielklasse `Fahrzeug` wäre ein Kandidat dafür; denn außer einen Namen zu speichern kann die Klasse nichts weiter tun. Erst mit den Ableitungen erwacht die Klasse zu Leben.

In einem solchen Fall kann man eine *abstrakte Klasse* erzeugen, indem man mindestens eine der virtuellen Funktionen „totlegt“. Das geschieht für mindestens eine virtuelle Funktion durch Zuweisen eines Nullwerts an die vtable, anstatt die Funktion zu definieren:

```
class Fahrzeug
{
/...
  virtual void zeigdich() const = 0;
/...
```

Eine solche Methode heißt *pure virtual function*

Jetzt können keine `Fahrzeug`-Objekte mehr erzeugt werden, allerdings kann man von der Klasse weitere ableiten. Falls diese alle in der Basisklasse als *pure virtual* deklarierten Methoden mit einer echten Definition überschreiben, dann sind sie *konkret* und man kann davon Objekte erzeugen.

Eine *abstract base class* heißt gelegentlich auch *ABC*. Ein Javaprogrammierer kann sie auch *Schnittstelle* nennen und so nutzen; man hat aber die Möglichkeit, mehr Funktionalität darin unterzubringen als dies in Java in einer Schnittstelle möglich wäre (insbesondere nichtkonstante Daten und vollständige Funktionsdefinitionen).

4.12.8 `friend`: Freunde haben freien Zutritt

Man kann innerhalb einer Klasse eine Funktion oder eine andere Klasse deklarieren, und mit dem Schlüsselwort `friend` dabei zulassen, daß die `friend`-Funktion beziehungsweise alle Elementfunktionen der `friend`-Klasse freien Zugriff auf alle Elemente der aktuellen Klasse haben, gleichgültig ob die Elemente im `private:`-, `protected:`- oder `public:`-Teil stehen.

Ob die `friend`-Deklaration selbst im `private:`-, `protected:`- oder `public:`-Bereich steht, ist belanglos; sie selbst gilt immer als `public`.

Beispiel für eine `friend`-Funktion:

```
class A
{
  friend void meinKumpel( A &objekt );

protected:

  int i;

public:
  //...
};


void meinKumpel( A &objekt )
{
  // obwohl A::i protected ist, ist hier (dank friend)
  // ein Zugriff zulässig:
  objekt.i = 25;
}

int main( int nargs, char **args )
{
  A Variable;

  meinKumpel( Variable );
}
```

4.12.8.1 Mehrfachvererbung

In den bisherigen Beispielen wurde jede Klasse entweder von Grund auf neu geschaffen, oder sie wurde von genau einer Basisklasse abgeleitet.

In *C++* ist es nun aber auch möglich, eine neue Klasse von mehreren Basisklassen abzuleiten (im Gegensatz zu Java beispielsweise).

Es ist allerdings schwer, dafür sinnvolle Beispiele zu finden, die vollständig ausformuliert in wenige Zeilen passen.

Hier wird nur das Prinzip gezeigt. Beispielsweise soll es ein Klasse `Kran` geben, die unter anderem durch ein Eigengewicht und eine maximale Last gekennzeichnet ist, sowie ein Klasse `Fahrwerk`.

Aus diesen beiden wird eine neue Klasse `FahrbarerKran` geschaffen, die sowohl von `Kran` als auch von `Fahrwerk` erbt:

```
class Kran
{
protected:
  double maxLast;
  double Eigengewicht;
  double Ausladung;

public:
  Kran( double maxLast, double Eigengewicht )
  {
    this->maxLast = maxLast;
    this->Eigengewicht = Eigengewicht;
  }

  virtual ~Kran(){}

  virtual double getEigengewicht() { return Eigengewicht; }

  virtual double getmaxLast() { return maxLast; }

  virtual double getAusladung() { return Ausladung; }
};

class Fahrwerk
```

```
{
protected:
  double maxLast;
  double Eigengewicht;

public:
  Fahrwerk( double maxLast,
            double Eigengewicht
            )
  {
    this->maxLast = maxLast;
    this->Eigengewicht = Eigengewicht;
  }

  virtual ~Fahrwerk(){}

  virtual double getEigengewicht() { return Eigengewicht; }

  virtual double getmaxLast() { return maxLast; }
};

class FahrbarerKran: public Fahrwerk, public Kran
{
public:
  FahrbarerKran( double maxLastFahrwerk,
                 double maxLastKran,
                 double EigengewichtFahrwerk,
                 double EigengewichtKran
                 )
    : Fahrwerk( maxLastFahrwerk, EigengewichtFahrwerk ),
      Kran( maxLastKran, EigengewichtKran )
  {
  }

  virtual ~FahrbarerKran(){}

  virtual double getEigengewicht()
  {
    return Kran::Eigengewicht + Fahrwerk::Eigengewicht;
  }

  virtual double getmaxLast()
  {
```

```
    // Die Last darf nicht größer werden als jeder der beiden
    // Werte:
    //    maxLast des Krans
    //    maxLast des Fahrwerks - Gewicht des aufgeschraubten Krans
    return ( Kran::maxLast>Fahrwerk::maxLast-Kran::Eigengewicht
             ? Fahrwerk::maxLast-Kran::Eigengewicht
             : Kran::maxLast
             );
  }
};

int main( int nargs, char **args )
{
  FahrbarerKran  Toefftoeff( 20000.0,  // Last Fahrwerk
                             20000.0,  // Last Kran
                             10000.0,  // Gewicht Fahrwerk
                             11000.0   // Gewicht Kran
                             );

  cout << "Gewicht : " << Toefftoeff.getEigengewicht() << endl;
  cout << "Last : " << Toefftoeff.getmaxLast() << endl;

  return 0;
} // main( int nargs, char **args )
```

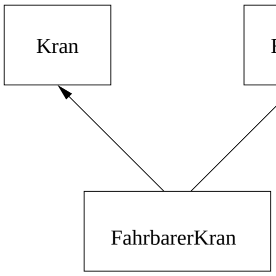

Abbildung 4.2: Klassenhierarchie `FahrbarerKran`

Ein Objekt des Typs `FahrbarerKran` ist sowohl ein `Fahrwerk`, als auch ein `Kran`. Deshalb kann ein solches Objekt (`Toefftoeff`) sowohl an eine Funktion übergeben werden, die ein `Fahrwerk` erwartet, als auch an eine die einen `Kran` erwartet.

Innerhalb von `FahrbarerKran` existieren alle Elemente aus `Fahrwerk` ebenso wie alle Elemente aus `Kran`. Wenn die Namen kollidieren, dann muß mit dem jeweiligen Klassennamen und dem *scope resolution operator* nachgeholfen werden, um klarzumachen, was gemeint ist (`Fahrwerk::maxLast-Kran::Eigengewicht`).

Was passiert aber, wenn durch Mehrfachvererbung eine Basisklasse effektiv mehrfach eingebunden wird? Ein Beispiel wäre, sowohl `Kran` als auch `Fahrwerk` von einer gemeinsamen Basisklasse `Maschine` abzuleiten, die bereits ein Gewicht enthält. Dann könnte in `Kran` und `Fahrwerk` das entsprechende Feld entfallen:

```
class Maschine
{
protected:
  double Eigengewicht;

public:
  Maschine( double Eigengewicht )
  {
    this->Eigengewicht = Eigengewicht;
  }

  virtual ~Maschine(){}

  virtual double getEigengewicht() { return Eigengewicht; }

};

class Fahrwerk: public Maschine
{
protected:
  double maxLast;

public:
  Fahrwerk( double maxLast, double Eigengewicht )
    : Maschine( Eigengewicht )
  {
    this->maxLast = maxLast;
  }

  virtual ~Fahrwerk(){}

  virtual double getEigengewicht() { return Eigengewicht; }

  virtual double getmaxLast() { return maxLast; }
```

```
};



class Kran: public Maschine
{
protected:
  double maxLast;
  double Ausladung;

public:
  Kran( double maxLast, double Eigengewicht )
    : Maschine( Eigengewicht )
  {
    this->maxLast = maxLast;
  }

  virtual ~Kran(){}

  virtual double getEigengewicht() { return Eigengewicht; }

  virtual double getmaxLast() { return maxLast; }

  virtual double getAusladung() { return Ausladung; }
};

class FahrbarerKran: public Fahrwerk, public Kran
{
public:
  FahrbarerKran( double maxLastFahrwerk,
                 double maxLastKran,
                 double EigengewichtFahrwerk,
                 double EigengewichtKran
                 )
    : Fahrwerk( maxLastFahrwerk, EigengewichtFahrwerk ),
      Kran( maxLastKran, EigengewichtKran )
  {
  }

  virtual ~FahrbarerKran(){}

  virtual double getEigengewicht()
  {
```

```
    return Kran::Eigengewicht + Fahrwerk::Eigengewicht;
  }

  virtual double getmaxLast()
  {
    // Die Last darf nicht größer werden als jeder der beiden
    // Werte:
    //    maxLast des Krans
    //    maxLast des Fahrwerks - Gewicht des aufgeschraubten Krans
    return ( Kran::maxLast>Fahrwerk::maxLast-Kran::Eigengewicht
             ? Fahrwerk::maxLast-Kran::Eigengewicht
             : Kran::maxLast
             );
  }
};


int main( int nargs, char **args )
{
  FahrbarerKran  Toefftoeff( 20000.0,  // Last Fahrwerk
                             20000.0,  // Last Kran
                             10000.0,  // Gewicht Fahrwerk
                             11000.0   // Gewicht Kran
                             );

  cout << "Gewicht : " << Toefftoeff.getEigengewicht() << endl;
  cout << "Last : " << Toefftoeff.getmaxLast() << endl;

  return 0;
} // main( int nargs, char **args )
```

In diesem Fall enthält sowohl `Kran` als auch `Fahrwerk` jeweils eine `Maschine`. In `FahrbarerKran` sind also zwei `Maschine`n enthalten!

Wenn man stattdessen wünscht, daß eine Klasse, die von `Kran` und `Fahrwerk` erbt, nur eine `Maschine` ist, dann müssen `Kran` und `Fahrwerk` von `Maschine` virtuell erben:

```
class Fahrwerk: public virtual Maschine
{
  //...
};
```

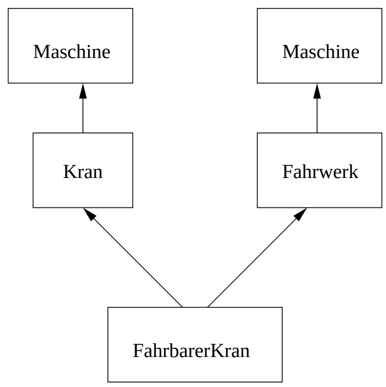

Abbildung 4.3: Klassenhierarchie `FahrbarerKran` mit 2 Maschinen

```
class Kran: public virtual Maschine
{
  //...
};

class FahrbarerKran: public Fahrwerk, public Kran
{
  //...
};
```

Dadurch wird bei einer folgenden Mehrfachvererbung nur eine `Maschine` erscheinen; dementsprechend gibt es auch nur ein gemeinsames `Eigengewicht`:

```
class Maschine
{
protected:
  double Eigengewicht;

public:
  Maschine( double Eigengewicht )
  {
    this->Eigengewicht = Eigengewicht;
  }

  virtual ~Maschine(){}

  virtual double getEigengewicht() { return Eigengewicht; }
```

```
};

class Fahrwerk: public virtual Maschine
{
protected:
  double maxLast;

public:
  Fahrwerk( double maxLast, double Eigengewicht )
    : Maschine( Eigengewicht )
  {
    this->maxLast = maxLast;
  }

  virtual ~Fahrwerk(){}

  virtual double getEigengewicht() { return Eigengewicht; }

  virtual double getmaxLast() { return maxLast; }
};



class Kran: public virtual Maschine
{
protected:
  double maxLast;
  double Ausladung;

public:
  Kran( double maxLast, double Eigengewicht )
    : Maschine( Eigengewicht )
  {
    this->maxLast = maxLast;
  }

  virtual ~Kran(){}

  virtual double getEigengewicht() { return Eigengewicht; }

  virtual double getmaxLast() { return maxLast; }
```

```
  virtual double getAusladung() { return Ausladung; }
};

class FahrbarerKran: public Fahrwerk, public Kran
{
public:
  FahrbarerKran( double maxLast,
                 double Eigengewicht
                 )
    : Maschine( Eigengewicht ),
      Fahrwerk( maxLast, Eigengewicht ),
      Kran( maxLast, Eigengewicht )
  {
  }

  virtual ~FahrbarerKran(){}

  virtual double getEigengewicht()
  {
    return Eigengewicht;
  }

  virtual double getmaxLast()
  {
    // Die Last darf nicht größer werden als jeder der beiden
    // Werte:
    //    maxLast des Krans
    //    maxLast des Fahrwerks - Gewicht des aufgeschraubten Krans
    return Kran::maxLast;
  }
};


int main( int nargs, char **args )
{
  FahrbarerKran  Toefftoeff( 9000.0,   // Last
                             21000.0   // Gesamtgewicht
                             );

  cout << "Gewicht : " << Toefftoeff.getEigengewicht() << endl;
  cout << "Last : " << Toefftoeff.getmaxLast() << endl;

  return 0;
```

```
} // main( int nargs, char **args )
```

Dieses Beispiel ist insofern schlecht konstruiert, als `Maschine::Eigengewicht` mehrfach initialisiert wird.

Aber das wesentliche der virtuellen Vererbung ist ersichtlich: die virtuelle geerbte Basisklasse ist nur noch einfach vorhanden.

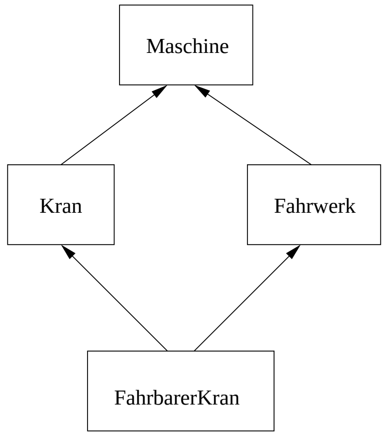

Abbildung 4.4: Klassenhierarchie `FahrbarerKran mit einer Maschine`

4.12.9 Überladen von Operatoren

In *C++* kann man für alle selbstgeschaffenen Klassen die meisten Operatoren überladen.

Im einzelnen sind dies die Rechenoperatoren `+`, `-`, `*`, `/`, `%`, `^`, `&`, `|`, `+`, `!`, `<<`, `>>`, die Vergleichsoperatoren `<`, `>`, `==`, `<=`, `>=`, `!=`, `&&`, `||`, die Zuweisungen `=`, `+=`, `-=`, `*=`, `/=`, `%=`, die Inkrement- und Dekrementoperatoren `++`, `--`, sowie der Feldzugriff `[]`, der Klammeroperator `()`, und die Speicheroperatoren `new`, `delete`, sowie `->`. Schließlich lassen sich auch explizite und implizite Typkonversionen ebenso wie Konstruktoren und Destruktoren überladen.

Durch Überladen der Operatoren mit eigenen Funktionen kann man nicht die Prioritäten der Operatoren untereinander verändern. Ebenfalls kann man nicht die Anzahl der Operanden verändern. Operatoren, die sowohl unär als auch binär existieren (zum Beispiel `-`), können dementsprechend getrennt voneinander zweimal überladen werden mit Funktionen, die jeweils ein oder zwei Argumente haben.

Mehrfaches Überladen mit Funktionen unterschiedlicher Parametertypen ist natürlich erlaubt.

Ein Überladen ist prinzipiell sowohl mit Elementfunktionen, als auch mit normalen Funktionen möglich. Nur die Operatoren `[]`, `()`, `->` und Konversionsfunktionen müssen Elementfunktionen sein. Mindestens einer der beteiligten Operanden muß ein Klassenobjekt sein; Dadurch kann man die Operationen, an denen ausschließlich elementare C-Objekte beteiligt sind, nicht neu definieren.

Eine Nichtelementfunktion, die einen Operator überlagert, hat logischerweise ebensoviele Parameter, wie der entsprechende Operator Operanden hat. (Einzige Ausnahme sind die Operatoren `++` und `--`. Diese können vor oder nach ihrem Operanden stehen (Präfixoperator, Postfixoperator) und liefern dementsprechend den Wert nach oder vor der Änderung zurück. Sowohl beide Versionen von `++` als auch beide Versionen von `--` können getrennt durch eine Funktion überladen werden. Da jetzt aber die beiden Versionen nicht anhand ihrer Parameterliste unterschieden werden könnten, wird für die jeweilige Postfixversion, also für *Operand*`++` und *Operand*`--`, ein zusätzlicher `int`-Parameter übergeben, der nur zur Unterscheidung der Parameterlisten gegenüber der Präfixversion dient, und ansonsten nicht verwendet wird.)

Bei der Definition als Elementfunktion entfällt der erste Parameter; stattdessen wird der erste Operand als zugehöriges Klassenobjekt angesprochen.

Der Name einer Operatorfunktion ist immer `operator`, gefolgt vom gewünschten Operatorzeichen, beispielsweise `operator+` für die Addition. Anzahl und Typen der Parameter ergeben sich aus den Operanden.

Als Beispiel soll eine neue Klasse für Kardinalzahlen, also nichtnegative Zahlen geschaffen werden. Der Einfachheit halber werden die Elementfunktionen gleich innerhalb der Klassendeklaration definiert, auch wenn das normalerweise nicht unbedingt sinnvoll ist (siehe **A.2 Aufteilen auf Quelltexte** auf Seite 185):

```
// Time-stamp: "15.01.04 01:27 card.cpp klaus@wachtler.de"

#include <iostream>
#include <cstdlib>

using namespace std;

// Die Deklaration der Klasse Cardinal.
class Cardinal
```

```
{
  friend ostream &operator<<( ostream &s, Cardinal &c );

private:

  unsigned long wert;

public:

  // parameterloser Konstruktor:
  Cardinal( void )
  {
    cout << "Konstruktor Cardinal( void )" << "\n";
    wert = 0;
  }

  // Konstruktor mit einem ganzzahligen Parameter:
  Cardinal( long int i )
  {
    cout << "Konstruktor Cardinal( long int i=" << i << " )" << "\n";
    if( (i<0) )
      {
        cerr << "Cardinal(): " << i << " ist negativ!\n";
        exit( 2 );
      }
    wert = i;
  }

  // Zuweisung unsigned int an Cardinal:
  Cardinal &operator=( unsigned int i )
  {
    cout << "operator=( unsigned int i=" << i << " )" << "\n";
    wert = i;
    return *this;
  }

  // Zuweisung int an Cardinal:
  Cardinal &operator=( int i )
  {
    cout << "operator=( int i=" << i << " )" << "\n";
    if( (i<0) )
      {
        cerr << "Cardinal(): " << i << " ist negativ!\n";
```

```
        exit( 2 );
      }
    wert = i;
    return *this;
  }

  // Zuweisung Cardinal an Cardinal:
  Cardinal &operator=( Cardinal c2 )
  {
    cout << "operator=( Cardinal c2=" << c2.wert << " )" << "\n";
    wert = c2.wert;
    return *this;
  }

  // Operator Cardinal+Cardinal:
  Cardinal operator+( Cardinal c2 )
  {
    cout << "Objekt " << wert
         << ", operator+( Cardinal c2=" << c2.wert << " )" << "\n";
    return wert + c2.wert;
  }

};

// Operator ostream<<Cardinal (Ausgabe von Cardinal nach ostream).
// Kann nicht als Elementfunktion von class Cardinal geschrieben werden,
// da der erste Parameter nicht vom Typ Cardinal ist:
ostream &operator<<( ostream &s, Cardinal &c )
{
  cout << "operator<<( ostream &s, Cardinal &c ) mit c=" << c.wert << "\n";
  s << c.wert;
  return s;
}


int main( int nargs, char *args[] )
{
  Cardinal
    c = 0,  // Aufr. d. Konstruktors Cardinal( long int i ) mit i=0
    d;      // Aufr. d. Konstruktors Cardinal( void )
  int
    j = 15;
```

```
  c = j;  // Aufr. v. c.operator=( int i ) mit i=j
  d = c;  // Aufr. v. d.operator=( Cardinal c2 ) mit c2=c

  d = c + 1; // Mit einem Konstruktor Cardinal( long int i )
             // wird ein temporäres Objekt erzeugt, dieses mit
             // c.operator+( Cardinal c2 ) mit dem temp. Objekt als c2
             // verknüpft, dann mit d.operator=( Cardinal c2 ) an d
             // zugewiesen.

  cout << c << "\n";
  cout << d << "\n";
}
```

Die Ausgabe dieses Programms sieht so aus:

```
Konstruktor Cardinal( long int i=0 )
Konstruktor Cardinal( void )
operator=( int i=15 )
operator=( Cardinal c2=15 )
Konstruktor Cardinal( long int i=1 )
Objekt 15, operator+( Cardinal c2=1 )
Konstruktor Cardinal( long int i=16 )
operator=( Cardinal c2=16 )
operator<<( ostream &s, Cardinal &c ) mit c=15
15
operator<<( ostream &s, Cardinal &c ) mit c=16
16
```

Zu diesem Programm folgende Anmerkungen:

- Da Operatoren als Elementfunktion implizit den ersten Operanden als aktuelles Objekt vom Typ der Klasse erhalten, kann man mit Elementfunktionen nur solche Operanden überladen, die auch ein Objekt der Klasse als ersten Operanden haben. Deshalb kann in diesem Beispiel die Funktion `ostream &operator<<( ostre` nicht als Elementfunktion der Klasse `Cardinal` vereinbart werden. Theoretisch wäre es möglich, die Funktion als Elementfunktion von `ostream` zu definieren. Da aber `ostream` im Nachhinein nicht geändert werden kann, und geschrieben wurde, als `Cardinal` nicht bekannt war, scheidet diese Möglichkeit praktisch aus.

  Wenn eine solche Funktion trotzdem auf die Elemente der Klasse zugreifen können soll, dann muß sie als `friend` vereinbart werden.

- Ein Operator läßt sich für verschiedene Kombinationen von Operandentypen mehrfach überladen (etwa `Cardinal operator=( unsigned int i )` und `Cardinal operator=( int i )`).
- Es sind auch mehrere Konstruktoren mit unterschiedlichen Parameterlisten erlaubt.

Wenn man den Operator `[]` überlädt, dann sollte die zugehörige Funktion eine Referenz auf eine Variable liefern. Würde die Funktion nur einen Wert liefern, dann wäre der resultierende Ausdruck bei der Verwendung des `[]`-Operators kein *lvalue*, an den etwas zugewiesen werden darf. Ein Ausdruck der Form `objekt[1] = ...` wäre dann nicht möglich.

Dasselbe gilt für andere Operatoren, wenn man deren Ergebnis möglicherweise als *lvalue* verwenden möchte.

Dazu ein kleines Beispiel, mit dem das Überladen des `[]`-Operators gezeigt wird:

```
// Time-stamp: "(15.11.01 16:10) intarr.cpp [Klaus Wachtler (aw38)]"
//
// demonstriert eine Klasse für int-Felder
// mit dynamischer Allokierung:
// Bei jedem Zugriff hinter das bislang allokierte
// Feld wird automatisch neu allokiert.

#include <iostream>
#include <cstdlib>

using namespace std;

class intarr
{
private:

  int       *p;  // Zeiger auf das eigentliche Feld
  size_t     l;  // die allokierte Länge

  void resize( size_t index );

public:
```

```
  intarr( void );
  ~intarr( void );
  int &operator[]( long index );

};

// Der Konstruktor sorgt für jedes intarr-Objekt für
// die Initialisierung (anfangs 0 Feldelemente).
intarr::intarr( void )
{
  p = NULL;
  l = 0;
}

// Der Destruktor sorgt für jedes intarr-Objekt für
// die Freigabe des allokierten Feldes.
intarr::~intarr( void )
{
  if( (p) )     // wurde überhaupt schon etwas allokiert?
    {
      free( p );
    }
}

// Der Operator [] wird mit einer Funktion überladen, die:
// a) ein Unterschreiten des Feldes (index<0) abfängt, und
// b) das Feld mit intarr::resize() ggf. verlängert, und dann
// c) eine Referenz auf das index'te Element des Objekts liefert.
int &intarr::operator[]( long index )
{
  if( (index<0) )          // Unterschreitung abfangen
    {
      cerr << "So geht das nicht!" << endl;
      cerr << "intarr::operator[]( long index ): (index=="
           << index << "), es muss aber (index>=0) sein!" << endl;
      exit( 1 );
    }

  resize( index );         // Das Feld ggf. verlängern

  return p[index];         // weil diese Funktion als Referenz
  // vereinbart ist (int &), wird hier
  // nicht der Wert eines Feldelements
```

```
  // geliefert, sondern eine Referenz darauf.
}

// resize() sorgt dafür, dass ein Feldelement mit der Nummer (index)
// verwendet werden kann.
// Wenn index ausserhalb des allokierten Bereichs liegt,
// wird das Feld mit realloc() entsprechend verlängert.
// Der neue Bereich wird mit (0)en vorbesetzt.
void intarr::resize( size_t index )
{
  if( (index >= l) )
    {
      // der gewünschte Index liegt hinter der
      // momentanen Feldgrenze, also muss das Feld
      // vergrössert werden:
      p = (int *)realloc( p, (++index)*sizeof(*p) );

      // die neugewonnenen Elemente erstmal auf 0 setzen:
      while( l<index )
        {
          p[l++] = 0;
        }
    }
}

// Testprogramm:
// Ein Feld (ohne Angabe von Grenzen!) wird als Variable feld
// vereinbart. Dann werden einige Elemente ( feld[4], feld[3] )
// belegt und feld[0] bis feld[5] dann ausgegeben.
// Zuletzt wird durch einen negativen Index ein Unterlauf des
// Feldes und damit ein Programmabbruch provoziert.

int main( int nargs, char *args[] )
{
  intarr     feld;   // Ein Objekt der Klasse wird erzeugt
  // (ohne Elemente)

  feld[4] = 44;      // feld[0] bis feld[4] werden allokiert,
  // feld[4] wird mit 44 beschrieben, der Rest mit 0
  feld[3] = 3;       // feld[3] wird überschrieben

  for( int i=0; i<=5; i++ )
    {
```

```
      // feld[0] bis feld[4] werden ausgegeben,
      // dann wird feld[5] mit realloc() geschaffen, mit 0
      // vorbesetzt und auch ausgegeben:
      cout << "feld[" << i << "] = " << feld[i] << endl;
    }

  feld[-5] = 12; // erzeugt Programmabbruch mit exit()

  return 0;
}
```

Die zugehörige Ausgabe lautet:

```
feld[0] = 0
feld[1] = 0
feld[2] = 0
feld[3] = 3
feld[4] = 44
feld[5] = 0
So geht das nicht!
intarr::operator[]( long index ): (index==-5), es muss aber (index>=0) sein!
```

4.12.10 Klassenschablonen (template-Klassen)

TODO Klassenschablonen, typename

```
// Time-stamp: "07.01.06 04:54 t.cpp klaus@wachtler.de"

#include <iostream>
#include <stdexcept>

template <class T, int N> class Feld
{
private:

  T     f[N];

public:
```

```
  T&operator[]( long index )
  {
    if( index<0 || index>=N )
      {
        throw( std::runtime_error( "so gehts nicht!" ) );
      }
    return f[index];
  }

};


int main( int nargs, char **args )
{
  try
    {
      Feld<int,10>  meinFeld;
      meinFeld[20] = 12;

    }
  catch( std::exception &error )
    {
      std::cerr << error.what() << std::endl;
    }

} // main( int nargs, char **args )
```

TODO: Spezialisierungen möglich! [Cline: C++ FAQs] 25.03

TODO: Defaultparameter:

```
// Time-stamp: "07.01.06 05:01 templdefltclass.cpp klaus@wachtler.de"

#include <iostream>
#include <stdexcept>

using namespace std;

template <class T, int N=10> class Feld
{
private:
```

```
  T     f[N];

public:

  T&operator[]( long index )
  {
    if( index<0 || index>=N )
      {
        throw( std::runtime_error( "so gehts nicht!" ) );
      }
    return f[index];
  }

};

int main( int nargs, char **args )
{
  try
    {
      Feld<int,10>  meinFeld;
      meinFeld[20] = 12;

    }
  catch( std::exception &error )
    {
      std::cerr << error.what() << std::endl;
    }

  try
    {
      Feld<int>  meinFeld;
      meinFeld[15] = 12;
      std::cout << meinFeld[5] << std::endl;

    }
  catch( std::exception &error )
    {
      std::cerr << error.what() << std::endl;
    }

} // main( int nargs, char **args )
```

4.12.11 Cheshire Cats: Information Hiding für Paranoide

Das unter objektorientierten Menschen so beliebte „information hiding“ ist in *C++* ähnlich umgesetzt wie in anderen objektorientierten Sprachen auch: man deklariert Teile einer Klasse als `public`, `protected` und `private` (und –anders als in Java– auch entsprechend, von welchen Klassen man ableitet). Damit gibt der Ersteller einer klasse vor, was nur in der Klasse benutzt werden darf und was der Benutzer beim Ableiten oder Instanziieren *verwenden* darf. Eine so erteilte Zugriffsbeschränkung wird zwar als *information hiding* bezeichnet, ist aber in Wirklichkeit gar kein Verstecken von Informationen: der Benutzer hat ja offensichtlich Lesezugriff auf die von ihm mit `#include` eingebundenen Headerdateien, in denen die `private`- und `protected`-Elemente der Klasse ja alle aufgeführt sind.

Auch wenn die so realisierte Zugriffsbeschränkung hinreichend ist, um sauberes Programmieren zu fördern, ist es gelegentlich noch nicht ausreichend.

Beispielsweise möchte ein Anbieter einer kommerziellen Bibliothek, der viel Entwicklungsaufwand in das Design steckte, sicher nicht seinen Konkurrenten die Arbeit beim Nachbau erleichtern, indem er alle Datenelemente und intern verwendeten Methoden in der Headerdatei auflistet; am besten noch gut kommentiert.

Die Definition der Methoden (also der wirklich ausgeführte Code) ist recht leicht vor dem Benutzer zu verstecken, indem in der Klasse nur die Deklarationen aufgeführt werden, und die Definitionen dagegen außerhalb der Klasse in einem eigenen Quelltext. Letzterer wird für das jeweilige Zielsystem kompiliert und dem Anwender nur als Objektdatei ausgeliefert.

Aber auch die Deklaration der Methoden und der Datenelemente ist bereits recht informativ.

Was der Anwender wirklich braucht, ist ja nur der öffentliche Teil (`public`) der Schnittstelle (und falls er Ableitungen der Klasse erstellen soll, auch noch den `protected`-Teil, was im folgenden aber ignoriert wird; es ist analog zu behandeln).

Um jetzt alles zu verbergen, was der Benutzer nicht unbedingt braucht, gibt es einen Mechanismus, der gelegentlich als *Cheshire Cats* bezeichnet wird:

Der Ersteller einer Klasse (beispielsweise `A`) baut in Wirklichkeit zwei Klassen:

1. eine interne Klasse `A_intern` mit der vollen Funktionalität in einem eigenen Quelltext A_intern.cpp und einer eigenen Deklarationsdatei A_intern.h. Diese Klasse wird dem Anwender nur kompiliert zur Verfügung gestellt; weder der

Quelltext A_intern.cpp noch die Deklarationen in A_intern.h werden aus der Hand gegeben.

2. Eine weitere Klasse `A`, mit der der Anwender letztlich arbeitet, wird ebenfalls in einem Quelltext (A.cpp) und einer Headerdatei A.h erstellt.

   Letztere wird dem Benutzer zur Verfügung gestellt, und enthält ausschließlich öffentliche Elemente und Methoden (oder auch `protected`, falls der Anwender ableiten darf). Zudem ist in der Klasse eine Referenz auf ein Element der geheimen Klasse `A` eingefügt, und im Konstruktor wird ein solches Element allokiert und im Destruktor wieder freigegeben. Dazu muß der Compiler beim Übersetzen der Anwenderprogramme nicht mehr wissen, als daß es eine Klasse mit dem Namen `A_intern` gibt.

   Die Implementation der Klasse `A` in der Datei A.cpp wird dem Anwender nicht im Quelltext zur Verfügung gestellt, sondern ebenfalls nur als kompilierte Objektdatei.

   Neben dem beschriebenen Allokieren und Freigeben des Elements der Klasse `A_intern` (was sowohl in der Headerdatei als auch in der Definitionsdatei erfolgen kann) enthält diese Klasse nur alle öffentlichen Methoden aus der geheim zu haltenden internen Klasse. Diese Methoden rufen jeweils nur die gleichnamige Methode aus der internen Klasse auf und reichen dabei ihre Parameter und die Funktionsergebnisse durch.

Was der Anwender also bekommt, ist nur noch eine Headerdatei A.h mit dem öffentlichen Teil (für den Compiler ebenso wie als Dokumentation, soweit diese nicht extern erfolgt), sowie die Kompilate der beiden Klassen.

Zum Begriff „Cheshire Cats“ habe ich zwei Stellen im Internet gefunden, mit denen die Herkunft aber auch nicht so recht klar wird:

- http://www.geocities.com/Heartland/Prairie/6520/legendary.html Cheshire Cat: In Chapter Six of Lewis Carroll's Alice in Wonderland (1865), a large cat is encountered grinning from ear to ear. When Alice asks why he is grinning, the only answer she gets is ¨It's a Cheshire cat and that's why.¨ There is no explanation why cats from that particular English county should be prone to smiling. A clue comes with the final disappearance of the cat, when it slowly vanishes, starting with the end of its tail and ending with the broad grin, which remains some time after the rest of the animal is gone. It is this disembodied grin that some authorities claim explains the source of Carroll's image. There used to be a special kind of Cheshire cheese that had a grinning feline face marked on one end. The rest of the cat was omitted by the cheesemaker, giving the impression that all but the grin had vanished.

Carroll may have seen these cheeses, but he may have taken his reference from an even earlier source. The Cheshire cheesemakers saw fit to add a grinning cat to their product because it was an abbreviation of ¨grin like a Cheshire Caterling¨, which was current about five centuries ago. Caterling was a lethal swordsman in the time of Richard III, a protector of the Royal Forests who was renowned for his evil grin, a grin that became even broader when he was despatching a poacher with his trusty sword. Caterling soon became shortened to ¨Cat¨ and anyone adopting a particularly wicked grin was said to be ¨grinning like a Cheshire Cat.¨ Carroll probably knew of this phrase, but because he refers to the grin outlasting the rest of body, it is more than likely his real influence was the cheese rather than the swordsman.

In reality, the true origin of the saying is much older; it was merely borrowed and made famous by him.

- http://website.lineone.net/ ssleightholm/dict/glossary/cheshire.htm Lewis Carroll did not, as is widely believed, create the Cheshire Cat. The phrase ¨to grin like a Cheshire cat¨ has been around for ages; it is

  An old simile popularized by Lewis Carroll:

  ¨please would you tell me,¨ said Alice a little timidly,... ¨why your cat grins like that?¨ ¨It's a Cheshire cat,¨ said the Duchess, ¨and that's why.¨ Alice in Wonderland, (1865), ch. vi.

  The phrase has never been satisfactorily explained, but it has been said that Cheshire cheese was once sold moulded like a cat that appeared to be grinning. The waggish explanation is that the cats know that Cheshire is a county palatine [properly the dominion of an earl palatine over which he had quasi-royal jurisdiction] and find the idea a source of amusement. (source: Brewer´s Dictionary of Phrase and Fable)

4.13 Ausnahmebehandlung

Ein wesentlicher Gesichtspunkt beim Konstruieren von Programmen ist die Frage, wie eine angemessene Fehlerbehandlung erfolgen soll.

Wenn das *Auftreten* eines Fehlers in einem Programmteil erfolgt, in dem auch die *Behandlung* sinnvoll möglich ist, dann hat man in keiner der herkömmlichen Programmiersprachen nennenswerte Probleme.

Schwieriger wird es, wenn das Auftreten eines Fehlers und eine sinnvolle Behandlung im Programm weit auseinanderliegen, und die betroffenen Programmteile nichts voneinander wissen, weil sie womöglich von verschiedenen Personen oder gar von verschiedenen Firmen stammen.

Beispielsweise könnte vom Hauptprogramm aus bald nach dem Programmstart eine Funktion `init()` aufgerufen werden, die alle Initialisierungen machen soll (Konfigurationsdateien lesen, Benutzeroberfläche initialisieren, Aufbau einer Verbindung zu einer Datenbank etc.). Einige Unterprogrammebenen tiefer tritt dabei ein Fehler auf: möglicherweise in einer zugekauften Bibliothek während der Initialisierung einer Datenbank. Beim Auftreten des Fehlers kann nicht sinnvoll reagiert werden (Wenn überhaupt wäre nur die Ausgabe einer Fehlermeldung sinnvoll, aber wohin? Ob das Programm unter einer grafischen Benutzeroberfläche läuft, wenn ja unter welcher, kann der Entwickler der Datenbankbibliothek nicht wissen. Wie sollte also ein Fehler ausgegeben werden? Mehr als eine Fehlerausgabe kann aber hier gar nicht stattfinden). Innerhalb von `init()` könnte sinnvoller reagiert werden, indem beispielsweise eine Fehlermeldung angemessen dem Benutzer gezeigt wird, und anschließend von Benutzer nach dem Namen einer anderen Datenbank gefragt wird, um damit einen zweiten Versuch zu unternehmen. Wenn also in `init()` auf einen Fehler reagiert werden kann, wie kommt dann die Information über den aufgetretenen Fehler dahin?

Der klassische Weg wäre, in jedem Unterprogramm das Auftreten von Fehlern vorzusehen, und das Unterprogramm mit einem speziellen Rückgabewert zu beenden. Das wirft aber Probleme auf:

- Jeder potentielle Fehler muß durch jede Unterprogrammebene durchgereicht werden, indem jede Teiloperation auf Fehler untersucht wird, und gegebenenfalls innerhalb des jeweiligen Unterprogramms alle offenen Vorgänge (allokierter Speicher, offene Dateien etc.) angemessen beendet werden.
- Als spezieller Rückgabewert beim Auftreten eines Fehlers hat man bei vielen Unterprogrammen einen Wert zur Verfügung; beispielsweise können Funktionen, die einen Zeiger liefern sollen, `NULL` zurückgeben, wenn etwas nicht geklappt hat (so wie es `malloc()` macht).

  Viele Unterprogramme benötigen aber schon ohne das Auftreten von Fehlern das gesamte Spektrum der möglichen Rückgabewerte für ihren Wertebereich, sodaß keine speziellen Werte frei sind. Ein Beispiel dafür ist `double tan( double arg )`; jeder mögliche Rückgabewert kann im Standard auch ein gültiger Wert sein (es sei denn, die konkrete Implementierung bietet etwas wie NAN=*not a number* an, worauf man sich aber nicht verlassen kann).
- Selbst wenn man einen speziellen Rückgabewert verwenden kann, dann kann

damit nur die Information *ein Fehler ist aufgetreten* geliefert werden. Eine genauere Beschreibung des Fehlers ist damit noch nicht möglich, sondern erfordert globale Objekte (wie beispielsweise `errno`), in denen mehr Information untergebracht werden kann.

Dies ist immer unelegant und unflexibel, und wird schnell aufwendig, ohne viel zu erreichen.

- Die Art der Fehlerbehandlung kann praktisch nicht über mehrere Unterprogrammebenen konsistent gehalten werden.

  Allein dies führt zu wirren Programmen, die schwer zu verstehen und anfällig für weitere Fehler sind.

- Da die Fehlerbehandlung einen erheblichen Teil des Quelltextes und der Programmlogik verschlingt (und zwar umso mehr, je kleiner und kompakter die einzelnen Funktionen werden), wird der Code stark aufgebläht. Das führt zu Programmen, die wesentlich größer, umständlicher und schwerer zu verstehen sind als ohne Fehlerbehandlung (und dadurch wiederum auch fehleranfälliger). Das ist besonders bedauerlich, weil der zusätzliche Code großteils nur dazu dient, die (hoffentlich seltenen) Fehler an höhere Unterprogrammebenen weiterzureichen, ohne eine konkrete Funktionalität zu haben.

Eine Lösung aller erwähnten Probleme auf einen Schlag hat man mit dem Konzept der **Ausnahmen** zur Verfügung.

Die Verwendung ist sehr elegant und einfach: Beim Auftreten eines Fehlers wird eine Ausnahme „geworfen“ (mit `throw`).
Beispiel:

```
throw "so gehts nicht!"
```

Dieses Werfen einer Ausnahme bedeutet, daß ein Objekt eines beliebigen Typs hinter `throw` angegeben wird, und damit der aktuelle Block beendet wird (und dabei alle automatischen Variablen sauber zerstört werden), dann der umgebende Block ebenso beendet wird, und so weiter. Alle auf dem Stack liegenden Variablen werden also von unten weg eine nach der anderen aufgeräumt (in der umgekehrten Reihenfolge ihres Erzeugens), und alle so beendeten Unterprogramme werden verlassen, um dann beim aufrufenden Unterprogramm ebenso zu verfahren. Dieser Vorgang heißt auch oft *stack unwinding*.

Das Aufräumen geht solange weiter, bis ein Block erreicht wird, der ausdrücklich eine geworfene Ausnahme „auffängt“ (wenn es keinen solchen Block gibt, dann endet das Aufräumen mit dem Beenden des aktuellen Programms!).

Ein Auffangen von Ausnahmen kann man mit einen `try`-Block erreichen, beispielsweise etwa so:

```
try
{
  auszuführender Code, Funktionsaufrufe etc.
}
catch( int i )
{
  Fehlerbehandlung, falls ein int geworfen wurde
}
catch( double d )
{
  Fehlerbehandlung, falls ein double geworfen wurde
}
catch( const char * text )
{
  Fehlerbehandlung, falls ein const char* geworfen wurde
}
catch( CPKW &f )
{
  Fehlerbehandlung, falls ein CPKW geworfen wurde
}
catch( CFahrzeug &f )
{
  Fehlerbehandlung, falls ein CFahrzeug geworfen wurde
}
catch( ... )
{
  Fehlerbehandlung, falls etwas anderes geworfen wurde
}
Jedenfalls: hier geht es weiter, egal ob Fehler oder nicht
```

Wenn also im Teil *auszuführender Code, Funktionsaufrufe etc.* irgendwo ein Fehler auftritt (möglicherweise mehrere Unterprogrammebenen tiefer), dann wird solange der Stack aufgeräumt, bis ein `try`-Block erreicht wird. Hier endet das Aufräumen zumindest vorläufig, und es werden alle `catch`-Zweige nacheinander geprüft, ob einer zum Typ der geworfenen Ausnahme paßt. Falls vorhanden, trifft der `catch( ... )`-Zweig auf alle nicht anderweitig gefangenen Ausnahmen zu. Der Code in dem gefundenen `catch`-Zweig wird ausgeführt, danach geht es in dem *Jedenfalls*-Teil weiter mit der Programmausführung (falls nicht der verwendete `catch`-Zweig mit einem weiteren `throw` oder `return` das verhindert).

Damit ein `catch`-Zweig zutrifft, muß die geworfene Ausnahme entweder exakt den angegebenen Typ haben, oder davon abgeleitet sein (und zwar durchgehend `public`, weil sonst vom `catch` aus die Basisklasse nicht sichtbar ist). Im `catch( CPKW &f )` können also alle Ausnahmen vom Typ `CPKW` gefangen werden, sowie allen davon `public` abgeleiteten Typen. Um das Casten eines abgeleiteten Typs in den angegebenen zu vermeiden, sollten zumindest Klassenobjekte per Referenz gefangen werden (wie oben gezeigt).

Im übrigen werden die `catch`-Zweige in der angegebenen Reihenfolge geprüft, bis der erste zutrifft; alle weiteren werden ignoriert. Ein `catch( ... )` ist also nur am Ende sinnvoll, weil alle danach folgenden Zweige nie erreicht werden können. Trifft keiner der Zweige zu, dann wird die Ausnahme einfach weitergeworfen.

Damit beschränkt sich die Fehlerbehandlung auf zwei Stellen: an einer Stelle wird der Fehler erkannt, und eine Ausnahme geworfen; und an einer anderen Stelle kann auf den Fehler reagiert werden, indem ein `try` mit einem passenden `catch` geschrieben wird.

Besonders schick ist die Tatsache, daß sich alle dazwischen liegenden Unterprogrammebenen überhaupt nicht um ein Weiterleiten des Fehlers bemühen müssen, sondern sich beliebig „dumm“ stellen können!

Durch die Konzentration der Fehlerbehandlung auf das Werfen und Fangen von Ausnahmen wird der gesamte dazwischen liegende Quelltext von jeglicher Fehlerbehandlung befreit, und wird dadurch kompakter und wesentlich leichter lesbar (sowohl für den menschlichen Leser des Quelltextes als auch für den Compiler, der ohne die vielen Fallunterscheidungen zur Fehlerbehandlung besser optimierten Maschinencode erzeugen kann).

Oft kommt es vor, daß man an einer Stelle zwar einen Fehler fangen möchte, aber hier nur bedingt darauf reagieren kann. Dann kann man aus dem `catch` heraus den gefangenen Fehler erneut auswerfen (oder einen anderen Fehler, wenn das sinnvoll erscheint; das heißt dann *Abbilden von Ausnahmen* oder *exception mapping*). Vor dem erneuten Auswerfen kann man den gefangenen Wert natürlich bei Bedarf manipulieren (beispielsweise an einen darin enthaltenen Text etwas anhängen).

Wie kann man aber in einem `catch( ... )` den gefangenen Fehler erneut auswerfen, wenn man gar keinen Namen hat, um das gefangene Objekt anzusprechen? Dafür gibt es die Möglichkeit, `throw` ohne Parameter zu verwenden. Dann wird das gefangene Objekt unverändert weitergeworfen.

Trotz der einfachen Verwendung kann eine beliebige Informationsmenge übertragen werden, indem ein passender Datentyp für das zu werfende Objekt gewählt wird. Das kann wie hier ein einfacher Typ sein. Oft werden aber Klassen nur zu dem

Zweck geschaffen, Objekte davon als Ausnahmen zu werfen (beispielsweise die **5.3 Ausnahmen der Standardbibliothek** auf Seite 143). Durch geschickten Aufbau einer Hierarchie solcher „Fehlerklassen“ kann man verschiedene Fehler voneinander abgeleiteter Klassen in einem `catch` fangen.

Beispielsweise wird von `new` eine Ausnahme vom Typ `bad_alloc` geworfen, wenn das Allokieren von Speicher gescheitert ist. Damit spart man sich viel Schreibarbeit und das Programm wird deutlich aufgeräumter, weil (im Gegensatz zum `NULL`-Ergebnis bei `malloc()`) nicht mehr jedes Allokieren von Speicher einzeln geprüft werden muß. Siehe dazu auch **4.5 Freier Speicher `new` und `delete`** auf Seite 32.

Wie gesagt werden automatische Variablen beim Aufräumen des Stacks sauber entfernt. Ein Problem hat man allerdings, wenn zwischen dem Allokieren einer sonstigen Resource (Speicher mit `new` oder `malloc()`, oder Öffnen einer Datei mit `fopen()`) und der zugehörigen Freigabe eine Ausnahme auftritt, die in dieser Ebene nicht gefangen wird. Dann wird ja das zugehörige Freigeben der Resource nicht mehr erreicht. Als Abhilfe kann man bei der Verwendung von Dateien auf `streams` ausweichen (**5.9 Streams** auf Seite 181) anstatt `FILE*` zu verwenden, beziehungsweise allokierten Speicher über **5.4 Autopointer `auto_ptr<T>`** auf Seite 144 verwalten.

In *C++*-Programmen ist es übrigens nicht zulässig, `setjmp()` und `longjmp()` zu verwenden. Damit lassen sich zwar ähnliche Sprünge von einer tieferen Unterprogrammebene zu einer höheren durchführen; ebenso wird hierbei der Stack entsprechend freigegeben. Aber: es werden für die freigegebenen automatischen Variablen keine Destruktoren aufgerufen!

4.13.1 Verwenden von Ausnahmen ohne Fehler

Kann man Ausnahmen auch für etwas anderes als zur Fehlerbehandlung verwenden? Natürlich ja, verbieten kann einem das niemand.

Ein etwas extremes Beispiel, einen String zu kopieren, wäre:

```
char *s = "abc";
char array[20];
try
  {
    char *src = s;
```

```
      char *dst = array;
      while( 1 )
        {
          *dst++ = *src;
          if( 0==*src++ )
            {
              throw "fertig";
            }
        }
    }
  catch( ... )
    {
    }
```

Hier wird die Schleife formal als Endlosschleife geschrieben, und dann nach dem Kopieren mit einem `throw` verlassen (anstatt einem `break`).

Prinzipiell ist es durchaus möglich, Ausnahmen auch für etwas anderes als Fehler zu verwenden: ein Schleifenende kann kaum als Fehler bezeichnet werden.

Generell wird aber von solchen Spielereien mehr oder weniger streng abgeraten, je nachdem wen man fragt.

Ein existierendes Programm zu verstehen, ist in der Praxis schon schwer genug (sogar ein eigenes, aber noch viel mehr wenn es der geistige Auswuchs eines Kollegen ist). Ausnahmen für nicht fehlerhafte Programmzustände zu verwenden, steigert die Verständlichkeit sicher nicht.

Es lassen sich aber durchaus Fälle finden, in denen Ausnahmen zumindest einen gewissen Charme haben. Beispielsweise hat man in *C++* keine Möglichkeit, aus einer Funktion mit einer bestimmten Signatur je nach den Umständen mal einen Wert eines Typs, mal eines anderen Typs zurückzugeben. Indem man aber Ausnahmen von verschiedenen Typen werfen kann, geht es auf einmal doch!

Zumindest kann man damit enorme Kreativität beweisen.

Ein ganz anderer Punkt ist die Laufzeiteffizienz: Üblicherweise kostet die Verwendung von `try`...`catch` nicht nennenswert Laufzeit, solange keine Ausnahmen geworfen werden. Das Werfen von Ausnahmen dagegen liegt zumindest in der Größenordnung eines Funktionsaufrufs (inklusive Rücksprung), kann aber auch noch teurer sein. Dies verursacht dem Hersteller kein schlechtes Gewissen, weil Fehler nicht allzu häufig auftreten sollten im Vergleich zu fehlerfrei laufendem Programmcode.

Mißbraucht man Ausnahmen dagegen für Programmzustände, die eigentlich keine Fehler sind, sondern häufig auftreten, kann man durchaus mit Laufzeiteinbußen rechnen.

4.13.2 Deklaration der von einer Funktion geworfenen Ausnahmen

Man kann (muß aber nicht) hinter der Parameterliste einer Funktionsdeklaration angeben, ob und wenn ja welche Ausnahmen von dieser Funktion geworfen werden:

```
void up( int i, int j ) throw( int, CFahrzeug );
```

Hier wird zugesichert, daß `up()` nur Ausnahmen vom Typ `int` oder `CFahrzeug` (oder davon abgeleiteten Klassen) wirft.

Dagegen garantiert die Deklaration:

```
void up2( int i, int j ) throw();
```

daß die Funktion überhaupt keine Ausnahme wirft.

Der Compiler kann dies zur Übersetzungszeit gar nicht oder zumindest nicht in jedem Fall kontrollieren, weil er nicht alle direkt oder indirekt aufgerufenen Unterprogramme kennt, also auch nicht kontrollieren kann, was darin alles geworfen werden könnte.

Wird eine Funktion an mehreren Stellen deklariert, dann müssen überall die selben Ausnahmen gelistet werden.

Macht man gar keine Angaben über die geworfenen Ausnahmen:

```
void up3( int i, int j );
```

dann darf die Funktion werfen, was sie will.

Will man eine virtuelle Elementfunktion überschreiben, dann darf die Überschreibung die Liste der geworfenen Ausnahmen einengen, aber nicht erweitern.

Wenn für Funktion eine Liste der möglichen Ausnahmen spezifiziert ist, sie aber trotzdem etwas anderes zu werfen versucht, dann wird zur Laufzeit die Funktion `std::unexpected()` aufgerufen, die normalerweise `std::terminate()` aufruft, die wiederum im Standardfall mit `abort()` das laufende Programm beendet.

Man hat mehrere Möglichkeiten, dieses Verhalten zu ändern:

- Die Funktion `std::unexpected()` kann durch einen Aufruf von `void set_unexpected( void(` auf eine eigene Funktion umgebogen werden. Diese eigene Funktion (keine

Parameter, keine Rückgabewerte) sollte je nach Bedarf den Fehler protokollieren, und dann durch einen Aufruf von `terminate()` das Programm beenden.

Die übergebene Funktion darf nicht zurückkehren, und keine Ausnahme werfen.

Anmerkung: Selbst wenn man gleich am Anfang von `main()` *mit* `set_unexpected()` *einen eigenen Handler für diesen Fall installiert, kann es trotzdem schon vor dem Eintritt in das Hauptprogramm zu einer unbehandelten Ausnahme kommen, und zwar wenn eine solche Ausnahme in einem Konstruktor auftritt, der für ein globales Objekt aufgerufen wird. Diese Aufrufe finden vor dem Eintritt in* `main()` *statt!*

- Analog kann man mit einem Aufruf von `void set_terminate( void(*)() )` eine eigene Funktion angeben, die anstelle von `terminate()` aufgerufen wird.

  Auch hier bekommt die übergebene Funktion keine Parameter, liefert keinen Rückgabewert, darf nicht zurückkehren und keine Ausnahme werfen.

- Als zusätzliche zulässige Ausnahme gibt man in der Deklaration der betreffenden Funktion den Typ `bad_exception` (siehe dazu **5.3 Ausnahmen der Standardbibliothek** auf Seite 143) an. Dann wird bei Auftreten einer nicht angegebenen Ausnahme automatisch vom Laufzeitsystem die tatsächlich auftretende Ausnahme auf eine vom Typ `bad_exception` umgesetzt, und diese kann von einem aufrufenden Programmteil wie gewohnt gefangen und behandelt werden. Allerdings geht damit jede Information über den Ort und die Art des aufgetretenen Fehlers verloren.

4.14 run time type information (RTTI)

Zusätzlich zu der „alten“ Möglichkeit in *C*, eine Typkonversion (*cast*) durchzuführen, indem man den gewünschten Typ in Klammern vor einen Ausdruck schreibt, kommen jetzt weitere Varianten hinzu, die wesentlich sicherer zu verwenden sind.

Ein Problem mit der alten cast-Version besteht darin, daß häufig unklar ist, was wirklich passiert.

Beispiele:

- `int i = 3; up( (double)i );` konvertiert eine lokale Kopie von `i` (ein temporäres Objekt) aktiv in den Typ `double`. Dabei wird tatsächlich eine sinnvolle Konversion vorgenommen; der an `up()` übergebene Wert ist `3.0`.

- `int i=3; *(double*)&i = 3.0` dagegen konvertiert nichts!

  Vielmehr wird `&i`, also die Adresse von `i`, als Zeiger auf eine `double` betrachtet (`(double*)&i`). Dann wird dahin, worauf diese Adresse zeigt (an die Stelle `*(double*)&i` im Speicher) ein `double`-Wert (3.0) geschrieben. Auf üblichen Rechnern ist eine `int` 4 Byte groß, eine `double` dagegen 8 oder mehr Byte. Es wird also über die 4 Byte von `i` hinaus weiterer dahinterliegender Speicher mit dem Bitmuster der Gleitkommazahl `3.0` überschrieben; in `i` sowie in dem nicht mehr dazugehörigen dahinterliegenden Speicher steht Schrott.

Der alte cast bewirkt also je nach Umständen manchmal wirklich eine sinnvolle Konversion, manchmal dagegen nur ein Neuinterpretieren eines Bitmusters, im obigen Beispiel einer Adresse. Gerade beim Interpretieren eines Zeigers hat man kaum Kontrolle, ob der cast Sinn macht.

Um die Absicht des Programmierers klarer auszudrücken, sollte der bisherige cast nur noch möglichst wenig verwendet werden. Stattdessen bieten sich die folgenden neuen Operatoren an:

- `dynamic_cast<T>` dient dazu, ein sicheres *down cast* durchzuführen. Damit ist gemeint, einen Zeiger auf ein Objekt einer Basisklasse in einen Zeiger auf ein Objekt einer davon abgeleiteten Klasse zu konvertieren (oder analog mit Referenzen).

  Die Richtung im Begriff *down cast* bezieht sich auf die übliche Darstellungsweise wie in Abbildung 4.1, bei der die Basisklasse oben gezeigt wird, und die davon abgeleiteten Klassen jeweils nach unten angehängt werden. `dynamic_cast<T>` konvertiert also einen Zeiger (oder eine Referenz), der als Zeiger (oder Referenz) auf eine Basisklasse deklariert ist, in Richtung der abgeleiteten Klasse `T`.

  Die Konvertierung gelingt, wenn folgende Voraussetzungen erfüllt sind:

  1. der zu konvertierende Wert ist ein Zeiger oder eine Referenz vom Typ einer (direkten oder indirekten) Basisklasse von `T`
  2. der zu konvertierende Wert oder Zeiger verweist tatsächlich auf ein Objekt vom Typ `T` oder einer davon durchgehend `public` abgeleiteten Klasse

  Wenn die Konvertierung gelingt, dann wird ein Zeiger vom Typ `T*` beziehungsweise eine Referenz vom Typ `T&` auf das ursprüngliche Objekt geliefert; das Objekt selbst wird dabei nicht verändert.

Im Fehlerfall wird bei Aufruf mit einem Zeiger der Wert `NULL` geliefert; bei einem Aufruf mit einer Referenz dagegen wird eine Ausnahme vom Typ `bad_cast` geworfen (siehe **5.3 Ausnahmen der Standardbibliothek** auf Seite 143).

In den beteiligten Klassen muß mindestens eine virtuelle Methode vorhanden sein.

Beispiel:

```
#include <iostream>

using namespace std;

class CA
{
public:
  virtual ~CA(){}
};

class CB
: public CA
{
public:
  virtual ~CB(){}
};

class CC
  : public CB
{
public:
  virtual ~CC(){}
};

void demo_dynamic_cast_Zeiger()
{
  {
    // das klappt (down cast):
    CC cc;
    CA *ptr = &cc;
    cout << "ptr = " << (void*)(dynamic_cast<CC*>(ptr)) << endl;
  }
  //{
  //  // das klappt NICHT (up cast):
```

```
  //  CA ca;
  //  CC *ptr = &ca;
  //  cout << "ptr = " << (void*)(dynamic_cast<CA*>(ptr)) << endl;
  //}
}

void demo_dynamic_cast_Referenz()
{
  {
    // das klappt (down cast):
    CC cc;
    CA &ref = cc;
    cout << "cc  = " << (void*)&(cc) << endl;
    cout << "ptr = " << (void*)&((dynamic_cast<CC&>(ref))) << endl;
  }
  //{
  //  // das klappt NICHT (up cast):
  //  CA ca;
  //  CC &ptr = ca;
  //  cout << "ca  = " << (void*)&(ca) << endl;
  //  cout << "ptr = " << (void*)&((dynamic_cast<CA&>(ref))) << endl;
  //}
}

int main( int nargs, char **args )
{
  demo_dynamic_cast_Zeiger();
  demo_dynamic_cast_Referenz();

  return 0;
} // main( int nargs, char **args )
```

- `static_cast<T>` erzwingt eine Konvertierung im Sinne des ursprünglichen cast (also `(T)`*wert*). Dabei führt der Compiler eine Konvertierung durch, soweit er kann (beispielsweise `double` nach `int` und ähnliches). Dies macht Sinn für die Konvertierung verschiedener einfacher Datentypen untereinander, zwischen Zeigern auf Objekte einer Klassenhierarchie, oder zwischen `enum`-Werten und integralen (also ganzzahligen) Werten.

  Ein `static_cast<T>` berücksichtigt `const`-Deklarationen und die Zugriffsrechte gemäß `public`, `private` und `protected`.

  Wenn die Konvertierung nicht möglich ist, tritt ein Übersetzungsfehler auf.

- `reinterpret_cast<T>` bewirkt, daß das Bitmuster eines gegebenen Ausdrucks einfach als ein anderer Typ angesehen wird. Der Compiler führt tatsächlich keine Konversion durch!

Angewendet werden kann das sinnvoll für Zeiger, wenn man sich über die Folgen im klaren ist. Ansonsten sollte man die Finger von explosiven oder stromführenden Gegenständen fernhalten.

- `const_cast<T>()` konvertiert nicht etwa irgendein Objekt in ein äquivalentes `const`-Objekt, wie der Name glauben machen könnte. Ganz im Gegenteil wird von einem `const`-Objekt das `const`-Attribut abgestreift.

  Der Typ `T` muß eine Referenz oder ein Zeiger sein; das abgestreifte `const` bezieht sich auf das referenzierte bzw. verwiesene Objekt.

  Beispiel:

```
  int  i;

  const int &iref = i;
  const int *iptr = &i;

  i     = 10;      // zulässig, da i nicht const ist
//iref  = 10;      // nicht zulässig, da iref eine const-Referenz ist
//*iptr = 10;      // nicht zulässig, da iptr ein const-Zeiger ist
  const_cast<int&>(iref)  = 10; // ok
  *const_cast<int*>(iptr) = 10; // ok
```

  Es ist nicht möglich, mit einem `const_cast` eine Konstante zu einer (nicht-`const`-) Variablen zu machen:

```
  const int j = 0;
//const_cast<int>(j)  = 10;  // geht nicht, da j nicht Ref. oder Zeiger
```

  Mit `const_cast` wird also nur das `const` einer Referenz oder eines Zeigers auf ein Objekt abgestreift, welches wiederum nicht `const` sein darf. Im obigen Fall wird der Fehler vom Compiler erkannt.

  In dem Beispiel:

```
  const int  ci=0;

  const int &ciref = ci;
  const int *ciptr = &ci;

//i     = 10;      // zulässig, da i nicht const ist
//iref  = 10;      // nicht zulässig, da iref eine const-Referenz ist
```

```
//*iptr = 10;      // nicht zulässig, da iptr ein const-Zeiger ist
  const_cast<int&>(ciref)  = 10; // ok
  *const_cast<int*>(ciptr) = 10; // ok

  std::cout << " ci     = " << ci << std::endl;
  std::cout << " ciref  = " << ciref << std::endl;
  std::cout << " *ciptr = " << *ciptr << std::endl;
```

wird mit dem `const_cast` wiederum nur das `const` der Referenz beziehungsweise des Zeigers abgestreift, aber das verwiesene Objekt selbst ist ebenfalls `const`, was durch `const_cast` nicht geändert wird. Damit ist der modifizierende Zugriff auf `ci` über die Referenz und den Zeiger zwar ebenfalls nicht zulässig, aber der Compiler wird das nicht erkennen können. Dann ist nicht definiert, was passiert.

Der GNU-Compiler (g++ 3.4.3, Linux 2.6) erzeugt folgende Ausgabe:

```
 ci     = 0
 ciref  = 10
 *ciptr = 10
```

Obwohl sowohl `ci` als auch `ciref` und `*ciptr` sich auf dieselbe Variable beziehen sollten, haben sie bei der Ausgabe unterschiedliche Werte!

Dies liegt daran, daß die `const`-Deklaration nicht unbedingt ein nicht veränderbares Objekt erzeugt, sondern nur besagt: „Der Programmierer will das Objekt nicht ändern“ (ein Versprechen an den Compiler, das er durch den Zugriff über die Referenz und den Zeiger gleich wieder bricht). Der Compiler verläßt sich aber darauf, daß `ci` nicht geändert wird, und erzeugt die Ausgabe von `ci` nicht durch Lesen des `const`-Objekts aus dem Speicher, sondern gleich durch Einsetzen des Werts in den Code zum Aufruf von `cout <<`...; aus seiner sollte es ja gleichwertig sein (aber schneller, als den vermeintlich bekannten Wert aus dem Speicher zu lesen). An der Ausgabe von `ciref` und `ciptr` erkennt man jedoch, daß `ci` tatsächlich geändert wurde.

Andere Compiler können das anders handhaben; möglicherweise sind alle drei ausgegebenen Werte gleich, oder das Programm bricht zur Laufzeit ab, wenn `ci` in einem tatsächlich schreibgeschützten Speicherbereich liegt.

Um die Ausgabe wenigstens konsistent zu machen, muß man `ci` als `volatile` deklarieren (um dem Compiler zu sagen, daß sich trotz des `const` der Wert durch irgendwelche Maßnahmen ändern kann, die sich seiner Kontrolle entziehen):

```
const volatile int  ci=0;
```

```
    const volatile int &ciref = ci;
    const volatile int *ciptr = &ci;

    const_cast<int&>(ciref)  = 10; // ok
    *const_cast<int*>(ciptr) = 10; // ok

    std::cout << " ci     = " << ci << std::endl;
    std::cout << " ciref  = " << ciref << std::endl;
    std::cout << " *ciptr = " << *ciptr << std::endl;
```

Jetzt ist auch bei der Ausgabe von `ci` der Wert 10 zu sehen, weil sich der Compiler nicht mehr auf den unveränderten Wert verläßt, sondern zur Ausgabe extra auf die Variable zugreift. Siehe dazu auch die Diskussion zu `const` und `volatile` in [KW:0 ist false] (Abschnitt **Daten**, dort **Attributangaben**).

Weiterhin gehört zur Gruppe der RTTI-Operatoren noch `typeid()`. Damit kann zur Laufzeit der Typ eines Ausdrucks oder eines Typnamens in Erfahrung gebracht werden. Ähnlich wie `sizeof()` kann `typeid()` mit einem Typ oder einem Ausdruck aufgerufen werden. Geliefert wird ein Objekt vom Typ `type_info` (deklariert in `<typeinfo>`), das allerdings nicht gespeichert werden kann (weil der Konstruktor `private` deklariert ist).

Man kann aber zwei Objekte davon vergleichen, sowie mit der Methode `name()` einen String liefern lassen, der eine textuelle Beschreibung des Typs liefert (der Text ist aber systemabhängig).

Wenn `typeid()` für einen `NULL`-Zeiger aufgerufen wird, wird eine Ausnahme vom Typ `bad_typeid` geworfen (**5.3 Ausnahmen der Standardbibliothek** auf Seite 143).

Beispiel:

```
#include <iostream>
#include <typeinfo>

using namespace std;

class A
{
};

int main( int nargs, char **args )
```

```
{
  A a;
  cout << ( typeid( A )==typeid( a ) ? "gleich" : "nicht gleich" )
       << endl;
  cout << " Name = " << typeid( A ).name() << endl;

  return 0;
}
```

Kapitel 5

Die Standardbibliothek

5.1 Überblick

Ähnlich wie in *C* ist die Definition der Syntax nur die halbe Miete; ein wesentlicher Teil der Funktionalität des genormten Sprachumfangs steckt in der zugehörigen Bibliothek.

Dies entlastet den eigentlichen Compiler, der auch ohne Ein-/Ausgabeoperationen, komplexe Datentypen und ähnlichem schon umfangreich und aufwendig zu bauen ist.

Der Umfang der Standardbibliothek ist in *C++* allerdings wesentlich größer als in *C*. ANSI-C enthält gut 130 relativ einfache Funktionen, verteilt auf 15 Headerdateien. Für *C++* kommen nochmals einige Dutzend Headerdateien hinzu, die untereinander teilweise recht eng verzahnt sind.

Apropos Headerdateien: Weil man sich bei der Normung nicht auf eine einheitliche Dateiendung für die Headerdateien einigen konnte, nahm man den kleinsten gemeinsamen Nenner: keine Endung.

Man schreibt also nicht wie von *C* gewohnt ein `.h` in die `#include`-Anweisung; Beispiel:

```
#include <iostream>
```

(und **nicht**:

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

). Da diese endungslose Variante der Headerdateien lange diskutiert wurde, waren die meisten Hersteller schon vorgeprescht und hatten in der Erwartung von .h als Standardendung in den vorläufigen Systemen solche Dateien angeboten. Viele existierende Programme verwenden demnach diese Dateien, weshalb sie auf den meisten Systemen vorhanden sind (ohne daß dies im endgültigen Standard so vorgesehen wäre).

So kann man in aller Regel beide Versionen verwenden; allerdings gibt es einen Unterschied (siehe **4.7 Namensbereiche (namespace)** auf Seite 36):

- #include <irgendwas.h> führt die darin enthaltenen Deklarationen außerhalb aller Namensbereiche ein, während
- #include <irgendwas> die Deklarationen in den Namensbereich std stellt. Deshalb muß man, um auf die Namen zugreifen zu können, entweder ein using namespace std; verwenden, oder die einzelnen Namen mit einem vorangestellten std:: kennzeichnen.

In der Standardbibliothek von *C++* werden nicht nur Funktionen definiert, sondern mit den neuen Sprachmöglichkeiten gleich universell verwendbare Datentypen geschaffen (insbesondere Container) und die darauf möglichen Operationen definiert. Ein Container repräsentiert dabei eine Menge anderer Objekte, beispielsweise als Liste oder Feld (Vektor).

Dieser Teil der Standardbibliothek ist die sogenannte *standard template library* (STL).

Solche Container (**5.5.2 Container** auf Seite 155) stellen beispielsweise eine Liste oder ein Feld eines anderen beliebigen Datentyps dar. Der zugrundeliegende Datentyp jedes einzelnen Elements ist nicht Gegenstand der STL, sondern vom Anwender der STL beizusteuern. Die STL implementiert also die allgemeinen Funktionen beispielsweise von Listen, und der Anwender füllt die leere Hülle (den Container) mit einem eigenen Typ. Dazu wird von *templates* reger Gebrauch gemacht.

Jeder dieser Container hat natürlich Möglichkeiten zum Einfügen, Löschen, und so weiter. Ein Teil der Schnittstelle ist einheitlich, so daß einige elementare Funktionalitäten mit mehreren Containern auf die selbe Art erfolgen kann.

Dies wird unter anderem für Iteratoren (siehe **5.5.3 Iteratoren** auf Seite 179)

genutzt; dies sind Objekte, mit denen man ein einzelnes Element der Gesamtmenge anfassen kann, und von einem Element zum nächsten wandern kann, ähnlich wie man in *C* einen Zeiger in ein Feld richten kann, und dann durch Hoch- oder Herunterzählen von einem Element zu einem anderen wandern kann. Nicht alle Container, aber zumindest die grundlegenden, ermöglichen Iteratoren.

Algorithmen (siehe **5.5.4 Algorithmen** auf Seite 181) sind ebenfalls nicht Teil der Container, sondern operieren als `templates` mit den (meisten) Containern identisch. Dazu rufen sie Methoden, hauptsächlich die *Iteratoren*, der jeweiligen Container auf.

Mit Algorithmen kann man Operationen wie Finden, Sortieren, oder selbstdefinierte Funktionalitäten auf einzelne Objekte oder (Teil-)Mengen oder alle Elemente des Containers durchführen.

Da weder Iteratoren noch Algorithmen die Container kennen, auf denen sie operieren (sondern nur als `templates` einige Funktionen aufrufen, die von den Containern zur Verfügung gestellt werden müssen), kann man auch eigene Container entwerfen, und die Iteratoren und Algorithmen darauf anwenden.

Neben der STL enthält die Standardbibliothek noch einige weitere gute Geister:

Strings (siehe **5.6 Strings** auf Seite 181) sind ein robuster und komfortabler Ersatz für die in *C* gängigen nullterminierten Zeichenketten.

Mit *streams* gibt es eine objektorientierte Alternative zu den gepufferten Ein-/Ausgabefunktionen aus *C* (`fopen()` etc.); siehe **5.9 Streams** auf Seite 181.

Zudem werden in der Standardbibliothek **Ausnahmen** definiert, die von Bibliotheksfunktionen geworfen werden können, aber auch für eigene Zwecke verwendbar sind (siehe **5.3 Ausnahmen der Standardbibliothek** auf Seite 143).

Die **Fehlerbehandlung** in der C++-Bibliothek ist nicht konsistent. Dies liegt zum einen daran, daß die einzelnen Teile aus unterschiedlichen Quellen stammen (die STL beispielsweise stammt weitgehend von Hewlett Packard) und beim Zusammenführen nicht stark verändert wurden, um einheitlicher auszusehen; zum anderen erfüllen die einzelnen Teile sehr unterschiedliche Anforderungen, sodaß eine einheitliche Fehlerbehandlung gar nicht sinnvoll ist.

Die STL ist durchgehend auf Laufzeiteffizienz getrimmt und macht so gut wie keine Fehlerüberprüfungen. Der Programmierer muß wieder wissen, was er tut und Fehlerüberprüfungen selbst machen, wo er sie für sinnvoll hält.

Streamein- und -ausgaben und `string` sind dagegen Beispiele für möglichst sicheres Programmieren und dementsprechend ist hier die interne Prüfung stärker und es werden bei Bedarf Ausnahmen geworfen.

Alle von ANSI-C übernommenen Funktionen (aus math.h beziehungsweise cmath, stdlib.h/cstdlib usw.) behalten dagegen ihre Fehlerbehandlung bei, um kompatibel zu bleiben und C-Programmierern das Leben zu erleichtern. Es werden Fehler also generell über den Rückgabewert von Funktionen und gegebenenfalls durch den Wert der globalen Variablen `errno` signalisiert.

Soweit die C++-Bibliothek über ANSI-C hinausgeht, werden Ausnahmen verwendet, die von `exception` abgeleitet sind; siehe **5.3 Ausnahmen der Standardbibliothek** auf Seite 143.

5.2 Funktionen aus ANSI-C

Da *C++* als Erweiterung von *C* entworfen wurde, kann man wie gewohnt die Headerdateien aus *C* einbinden und die zugehörigen Funktionen aufrufen:

```
#include <string.h>
// ...
char *p = ...
strcpy( p, "blabla" );
```

Als Alternative dazu kann man aber auch eine Headerdatei verwenden, die dem aus *C* gewohnten Namen entspricht, aber mit einem vorangestellten `c` und ohne Dateiendung; beispielsweise:

```
#include <cstring>
```

Die so deklarierten Funktionen stehen (wie alles in den Headerdateien der Standardbibliothek Definierte) im Namensbereich `std`, und können also entweder so:

```
std::strcpy( p, "blabla" );
```

oder so:

```
using namespace std;
strcpy( p, "blabla" );
```

aufgerufen werden:

Siehe dazu auch **4.7 Namensbereiche (`namespace`)** auf Seite 36.

Da die *C*-Funktionen in *C++* identisch verwendet werden können und in [KW:0 ist false] beschrieben sind, werden sie hier nicht wiederholt.

Zu beachten ist allerdings, daß nicht alle *C*-Funktionen in *C++* sinnvoll verwendet werden können:

- `setjmp()` und `longjmp()` haben in *C++* nichts verloren.
- `abort()` sollte durch `terminate()` ersetzt werden
- `bsearch()` und `qsort()` funktionieren nur, wenn die Elemente des übergebenen Feldes aufeinanderfolgend im Speicher steht. In *C++* ist es durchaus möglich, Container zu bauen, auf dessen Elemente man mit dem Feldzugriffsoperator `[]` zugreifen kann, obwohl die Elemente nicht aufeinanderfolgend zu finden sind. Dann kann man diese beiden Funktionen nicht verwenden. Weiterhin sortiert `qsort()` Elemente um, ohne sich um deren Typ zu kümmern. Das geht dann schief, wenn die zu sortierenden Elemente nicht einfach im Speicher verschoben werden dürfen (weil sie beispielsweise interne Zeiger pflegen müssen, oder weil anderweitig Referenzen darauf verweisen); `qsort()` wird niemals Elemente verschieben, indem an der neuen Stelle das alte Element freigegeben wird und ein Copy-Konstruktor den Speicher initialisiert anhand des dahin zu kopierenden Elements.
- `malloc()`, `calloc()`, `realloc()` und `free()` sind ungeeignet für Objekte von Klassen, weil für allokierten Speicher keine Konstruktoren und für freigegebenen Speicher keine Destruktoren aufgerufen werden.
- Analog sind `memchr()`, `memcmp()`, `memcpy()`, `memmove()` und `memset()` mit Vorsicht zu genießen, wenn der betroffene Speicher zu Objekten gehört.

5.3 Ausnahmen der Standardbibliothek

In den Headerdateien `<exception>` und `<stdexcept>` sind bereits einige Klassen vorgesehen, die zum Werfen von Ausnahmen geeignet sind, und teilweise auch von der Standardbibliothek selbst verwendet werden können:

- `exception` ist die Basis aller hier definierten Ausnahmen
  - `ios_base::failure`
  - `logic_error` ist die Basis für logische Fehler (die schon vor der Laufzeit erkannt werden könnten):
    - `domain_error`
    - `invalid_argument`
    - `length_error`
    - `out_of_range`
  - `bad_cast` wird geworfen, wenn ein `dynamic_cast` scheitert (**4.14 run time type information (RTTI)** auf Seite 131)
  - `bad_typeid` wird geworfen, wenn `typeid(NULL)` aufgerufen wird (**4.14 run time type information (RTTI)** auf Seite 131).
  - `runtime_error` ist die Basis für Fehler, die erst zur Laufzeit erkannt werden können:
    - `range_error`
    - `overflow_error`
    - `underflow_error`
    - `bad_alloc` wird geworfen, wenn ein `dynamic_cast<T>` mit einer Referenz scheitert (**4.14 run time type information (RTTI)** auf Seite 131)
  - `bad_exception` wird möglicherweise geworfen, wenn in der Deklaration einer Funktion (oder Methode) garantiert wird, daß nur bestimmte Ausnahmen auftreten dürfen, zur Laufzeit aber dann eine andere Ausnahme geworfen wird (siehe **4.13.2 Deklaration der von einer Funktion geworfenen Ausnahmen** auf Seite 130).

Jedes Objekt dieser Klassen muß mit einem `string` (oder einem `const char*`) initialisiert werden, der als Fehlerbeschreibung dient.

Den enthaltenen Text kann man mit der Methode `what()` erfahren.

5.4 Autopointer `auto_ptr<T>`

Als Motivation für das eigentliche Thema dieses Abschnitts werden ein paar gefährliche, aber durchaus übliche Programmfragmente gezeigt. Durch die starke

Verdichtung auf das Wesentliche sind die Probleme hoffentlich leicht zu erkennen; aber in realem Quelltext werden sie oft nicht erkannt, wodurch Speicherlecks entstehen (allokierter Speicher wird nicht freigegeben) oder es wird illegal auf ungültigen Speicher zugegriffen (weil ein Objekt beispielsweise bereits vor dem Zugriff freigegeben wurde). Letzteres führt –falls man Glück hat– zu einem Programmabbruch, oder zu fehlerhaften Daten.

Relativ harmlos erscheint es meistens, in einem Block einen Zeiger auf ein dort bekanntes Objekt zu richten (beispielsweise auf neu allokierten Speicher), über den Zeiger etwas mit dem Objekt zu machen, und praktisch zeitgleich mit dem Ende der Lebensdauer des Zeigers auch das Objekt zu zerstören, auf das der Zeiger verweist:

```
{
  Cwassweissich *ptr = new Cwassweissich;

  tuwas( ptr );
  // ...

  delete ptr;   // hier endet die Lebensdauer des allokierten Objekts
}               // hier endet die Lebensdauer von ptr
```

Der Programmierer dieses Quelltextes gibt den allokierten Speicher sauber frei, bevor der Block verlassen wird.

Probleme treten regelmäßig auf, wenn der Programmfluß nicht so schön linear verläuft wie in dem obigen Beispiel, oder wenn die Lebensdauer des Zeigers und des darauf verwiesenen Objekts unterschiedlich sind.

Beispiele für heikle Fälle:

- Zwischen dem Allokieren und Freigeben von Speicher wird der Programmfluß durch ein `return` unterbrochen:

  ```
  {
    Cwassweissich *ptr = new Cwassweissich;

    tuwas( ptr );
    if( einProblem ) return;
    // ...
  ```

```
    delete ptr;
  }
```

Hier wird leicht übersehen, den Speicher im `if`-Zweig freizugeben.

Analoges gilt für ein `break`:

```
  while( schoeneswetter )
  {
    Cwassweissich *ptr = new Cwassweissich;

    tuwas( ptr );
    if( einProblem ) break;
    // ...

    delete ptr; // ggf. wegen break nicht ausgeführt!
  }
```

In beiden Fällen würde die Lösung natürlich darin bestehen, im `if`-Zweig den Speicher freizugeben. Bei mehreren Zeigern in einem etwas verwickelten Unterprogramm wird das aber schnell unübersichtlich und fehleranfällig.

- Der Programmteil zwischen dem Allokieren und Freigeben löst eine Ausnahme aus, was dem Programmierer möglicherweise gar nicht bewußt ist:

```
  {
    Cwassweissich *ptr = new Cwassweissich;
    Cwassweissich objekt;

    tuwas( ptr ); // hier tritt eine Ausnahme auf
    // ...

    delete ptr; // ggf. wegen Ausnahme nicht ausgeführt!
  }
```

Durch den Mechanismus des *stack unwinding* (siehe **4.13 Ausnahmebehandlung** auf Seite 123) werden durch die nicht gefangene Ausnahme alle Objekte auf dem Stack aufgelöst, also hier `objekt` ebenso wie der Zeiger `ptr`. Aber das, worauf der Zeiger verweist (nämlich der mit `new` allokierte Speicher), liegt nicht auf dem Stack und wird hier auch nicht freigegeben!

Der gezeigte Block wird dann verlassen (bis auf dem Stack weiter oben ein passendes `try`...`catch` gefunden wird), und der allokierte Speicher wird nicht mehr freigegeben.

Das Problem läßt sich lösen, indem man um jeden solchen Programmteil, in dem allokiert wird, ein eigenes `try` baut:

```
{
  Cwassweissich *ptr = NULL;
  try
  {
    ptr = new Cwassweissich;
    Cwassweissich objekt;

    tuwas( ptr ); // hier tritt eine Ausnahme auf
    // ...
    delete ptr;
  }
  catch(...)
  {
    delete ptr;
    throw;
  }
}
```

Dadurch wird zwar das ursprüngliche Problem gelöst, aber schön ist das nicht: der Quelltext wird unnötig aufgebläht, das `delete` ist jetzt an zwei Stellen nötig, und die Lebensdauer des Zeigers ist jetzt ungleich der Lebensdauer des allokierten Objekts, womit wir schon beim nächsten kritischen Fall wären:

- Eine unterschiedliche Lebensdauer eines Zeigers und des darauf verwiesenen Objekts führt immer wieder zu Fehlern.

  Ein offensichtliches Beispiel, bei dem der Zeiger länger lebt als das Objekt:

```
{
  Cwassweissich *ptr;

 // Achtung!
 // Hier existiert der Zeiger ptr schon, aber man darf über
 // *ptr auf nichts zugreifen!

  ptr = new Cwassweissich;  // ab jetzt ist *ptr verwendbar

  tuwas( ptr );
  // ...
  delete ptr;  // jetzt ist *ptr tabu

 // Achtung!
 // Hier existiert der Zeiger ptr noch, aber man darf über
```

```
     // *ptr auf nichts zugreifen!

    } // jetzt ist auch ptr verschwunden!
```

- Noch lustiger wird es, wenn man gezwungen ist, in einem Unterprogramm etwas zu allokieren, was der Aufrufer verwenden soll, aber dann auch selbst wieder freigeben muß:

```
Cwassweissich *up()
{
  Cwassweissich *ret = new Cwassweissich;   // up() allokiert
  // ...
  return ret;
}

void Aufrufer()
{
  Cwassweissich *ptr = up();

  tuwas( ptr );
  // ...

  delete ptr;                               // Aufrufer gibt frei
}
```

  Die Freigabe beim Aufrufer ist natürlich anfällig dafür, vergessen zu werden oder möglicherweise mehrfach versucht zu werden.

Alle beschriebenen Probleme (außer dem letzten) lassen sich überraschend einfach lösen, wenn man den Zeiger in eine eigene Klasse „verpackt“:

```
class Cwassweissich_ptr_t
{
public:

  // Konstruktor
  Cwassweissich_ptr_t()
    : ptr( NULL )
  {
  }

  // Konstruktor mit einem Zeiger auf Cwassweissich:
```

```
  Cwassweissich_ptr_t( Cwassweissich *rechteSeite )
    : ptr( rechteSeite )
  {
  }

  // Destruktor
  ~Cwassweissich_ptr_t()
  {
    delete ptr;
  }

  // Zuweisung Cwassweissich_ptr_t = Cwassweissich*
  Cwassweissich_ptr_t & operator=( Cwassweissich *rechteSeite )
  {
    // bei mehreren Zuweisungen nacheinander die
    // vorherigen Zeiger freigeben:
    delete ptr;

    // neuen Zeiger merken:
    ptr = rechteSeite;

    return *this;
  }

  // Damit man ein *Cwassweissich_ptr_t anstelle eines
  // *Cwassweissich verwenden kann, bauen wir eine
  // passende Typkonvertierung:
  operator Cwassweissich*() const
  {
    return ptr;
  }

private:

  // der eigentliche Zeiger auf den allokierten Speicher:
  Cwassweissich * ptr;

}; // class Cwassweissich_ptr_t
```

Die Verwendung ändert sich geringfügig, weil statt eines `Cwassweissich*` jetzt ein `Cwassweissich_ptr_t` verwendet werden muß, wodurch ein manuelles Freigeben überflüssig wird (und auch gar nicht mehr möglich ist):

```
{
  Cwassweissich_ptr_t ptr = new Cwassweissich;

  tuwas( ptr );              // Ausnahme? Kein Problem!
  if( einProblem ) return;  // return? Kein Problem!
  // ...

 // spätestens hier endet die Lebensdauer von ptr,
 // allokierter Speicher wird im Destruktor automatisch
 // freigegeben!
}
```

Durch diesen Kunstgriff meisterlicher Hand stören Unterbrechungen im linearen Programmfluß nicht mehr weiter:

- Falls in `tuwas()` eine Ausnahme geworfen wird, wird das Objekt `ptr` vom Stack entfernt, und dabei sein Destruktor `~Cwassweissich_ptr_t()` aufgerufen.

  Und dabei wird wie gewünscht mit einem `delete` der mit `new` allokierte Speicher freigegeben.

- Falls die Funktion mit einem `return` verlassen wird, passiert exakt dasselbe.

Durch das Einbetten des Zeigers in ein automatisches Objekt auf dem Stack wird also die Lebensdauer eines Heap-Objekts (`Cwassweissich`) an die Lebensdauer des Stackobjekts (`Cwassweissich_ptr_t`) gebunden, und die Speicherverwaltung wird –ohne weiteres Zutun des Aufrufers– wieder konsistent.

Das letzte der geschilderten Probleme (Allokieren in einer Funktion, nötige Freigabe beim Aufrufer) läßt sich mit wenig Mehraufwand lösen. Wenn in einer Funktion ebenfalls ein `Cwassweissich_ptr_t` erzeugt und als Rückgabewert an den Aufrufer geliefert wird, müssen an der Klasse `Cwassweissich_ptr_t` nur wenige Änderungen gemacht werden:

- An einem allokierten Objekt (vom Typ `Cwassweissich`) muß immer genau ein `Cwassweissich_ptr_t` (von möglicherweise mehreren gleichzeitig existierenden) den *Besitz* haben. Dazu existiert ein zusätzliches Element vom Typ `bool`; beispielsweise soll dieses `owner` heißen.

  Die Idee hinter diesem Klassenelement ist, daß es zwar möglicherweise mehrere `Cwassweissich_ptr_t`-Objekte zu einem allokierten Speicher gibt, aber die Methoden so konstruiert sind, daß immer genau eines auch Besitzer ist.

- `Cwassweissich_ptr_t` benötigt einen copy-Konstruktor, der gegebenenfalls den Besitz `owner` von der rechten Seite (dem Initialisierer) auf die linke Seite (das zu initialisierende Objekt) überträgt, und beim Initialisierer löscht.
- Ebenso definiert man im `operator=()` einen Übergang des Besitzes.
- der Destruktor eines `Cwassweissich_ptr_t` gibt den für das zugehörige `Cwassweissich` allokierten Speicher nur dann frei, wenn es auch den Besitz daran hat. Weil immer genau ein `Cwassweissich_ptr_t` Besitzer des allokierten Speichers ist, werden auch genau einmal `delete` und dadurch der Destruktor aufgerufen.

Mit einem so erweiterten `Cwassweissich_ptr_t` kann man in einem Unterprogramm ein `Cwassweissich` allokieren, und als `Cwassweissich_ptr_t` verpackt zurückgeben:

```
Cwassweissich_ptr_t up()
{
  Cwassweissich_ptr_t ret = new Cwassweissich;   // up() allokiert
  // ...

  // hier wird ein temporäres Cwassweissich_ptr_t-Objekt
  // geschaffen, was den Besitz am allokierten Speicher übernimmt.
  // Deshalb gibt der Destruktor von ret keinen Speicher frei!
  return ret;
}

void Aufrufer()
{
  // bei der Initialisierung geht der Besitz am allokierten
  // Speicher an ptr über.
  // Anschließend wird das temporäre Objekt freigegeben,
  // und der allokierte Speicher nicht freigegeben, weil
  // das temporäre Objekt keinen Besitz mehr hat.
  Cwassweissich_ptr_t ptr = up();

  tuwas( ptr );
  // ...

 // spätestens hier endet die Lebensdauer von ptr,
 // allokierter Speicher wird im Destruktor automatisch
 // freigegeben, weil ptr den Besitz am allokierten Speicher
 // hat!
}
```

Für die Rückgabe aus dem Unterprogramm wird der Compiler ein temporäres Objekt vom Typ `Cwassweissich_ptr_t` schaffen und mit einem copy-Konstruktor initialisieren; dabei geht der Besitz auf das temporäre Objekt über.

Der Aufrufer wird das Ergebnis des Funktionsaufrufs an eine automatische Variable vom Typ `Cwassweissich_ptr_t` zuweisen; dabei geht durch den `operator=()` der Besitz an diese über.

Mit dem Ende der Lebensdauer der automatischen Variable wird dann auch der allokierte Speicher freigegeben.

Soweit so gut. Mit dem bisherigen Wissen kann man also alle geschilderten Probleme leicht umgehen, wenn man sich für jeden Zeiger auf irgendeinen Typ die Mühe macht, und eine passende Verpackung baut und statt des Zeigers verwendet.

Weil diese Verpackung aber außer dem geschilderten Mechanismus mit `delete` und dem Besitz am allokierten Speicher nichts machen muß, und insbesondere nichts über den verpackten Datentyp wissen muß, bietet sich dafür eine `template`-Klasse an (siehe **4.12.10 Klassenschablonen (template-Klassen)** auf Seite 118).

Und weil die Problematik sehr gängig ist und die beschriebene Lösung sehr hilfreich, gibt es eine solche `template`-Klasse bereits in der STL (in `<memory>`).

Die Klasse heißt `auto_ptr<T>` und hat genau das oben beschriebene Verhalten.

Beispielverwendung:

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

// ...

auto_ptr<Cwassweissich> up()
{
  auto_ptr<Cwassweissich> ret( new Cwassweissich );   // up() allokiert
  // ...

  return ret;
}
```

```
void Aufrufer()
{
  auto_ptr<Cwassweissich> ptr = up();

  tuwas( ptr );
  // ...

} // hier automatische Freigabe mit delete!
```

Achtung! Auch hier gibt es wieder eine Fallgrube. Durch die Tatsache, daß beim Anlegen einer Kopie der Besitz auf die Kopie übergeht, muß man das Erzeugen temporärer Kopien durch den Compiler vermeiden. Diese werden beispielsweise beim Aufruf von Unterprogrammen angelegt, wenn ein Parameter als *Wert* übergeben wird (*call by value*). Mit dem Löschen dieses temporären Objekts würde auch der allokierte Speicher freigegeben werden.

Beispiel:

```
void up( Cwassweissich parameter )
{
  // ...
}

// ...

  auto_ptr<Cwassweissich> a( new Cwassweissich ); // a hat Besitz

  // für die Übergabe wird eine temporäre Kopie von a erzeugt,
  // dafür wird der copy-Konstruktor aufgerufen und der Besitz
  // am allokierten Speicher geht auf die temporäre Kopie über.
  // Die Kopie wird in up() als Parameter verwendet.
  up( a );
  // Mit dem Ende der Funktion up() wird die temporäre Kopie
  // freigegeben, und der oben mit new allokierte Speicher wird
  // freigegeben, weil sie den Besitz daran hat.

  // Ab hier darf nicht mehr auf a zugegriffen werden!
```

Aus diesem Grund dürfen `auto_ptr<T>`-Objekte niemals als Wert an ein Unterprogramm übergeben werden, sondern nur als Referenz!

Zum **Erzeugen** eines `auto_ptr<T>`-Objekts existieren folgende Konstruktoren:

- ein parameterloser Konstruktor; der erzeugte auto_ptr<T> enthält kein Objekt.

- ein Konstruktor, der einen Zeiger auf ein mit new allokiertes Objekt übergeben bekommt; der erzeugte auto_ptr<T> enthält das Objekt und hat den Besitz daran.

- ein copy-Konstruktor, der das enthaltene Objekt übernimmt und den Besitz daran überträgt. Der Typ des zur Initialisierung übergebenen auto_ptr darf auch eine vom Typ des zu initialisierenden Objekts abgeleitete Klasse sein (oder es muß eine andere automatische Typumwandlung bestehen).

Daneben existiert noch wie bereits oben erwähnt eine Zuweisung, die ein auf der linken Seite eventuell vorhandenes Objekt freigibt (falls die linke Seite den Besitz daran hat), dann das Objekt von der rechten auf die linke Seite kopiert, und dann den Besitz überträgt (falls auf der rechten Seite vorhanden). Der Typ des auf der rechten Seite angegebenen auto_ptr darf auch eine vom Typ Objekts auf der linken Seite abgeleitete Klasse sein (oder es muß eine andere automatische Typumwandlung bestehen).

Mit einer Memberfunktion get() kann man einen Zeiger auf das verwaltete Objekt bekommen (oder NULL, wenn kein solches existiert).

release() löscht den Besitz eines auto_ptr, ohne das Objekt zu zerstören. Rückgabewert ist ein Zeiger auf das enthaltene Objekt, oder NULL wenn kein solches existiert.

Mit operator* und operator-> kann man auf das verwiesene Objekt oder seine Elemente zugreifen.

5.5 Standard Template Library (STL)

Die Standard Template Library (STL) bietet einen Satz von template-Klassen sowie zugehörige Funktionen an, mit denen man aus fast beliebigen Datentypen komplexere Gebilde bauen kann.

5.5.1 Vergleichsoperationen `rel_ops`

In <utility> sind zusätzliche Vergleichsoperationen als `template`-Operatorfunktionen für `!=`, `<=`, `>` und `>=` definiert, die intern nur `==` und `<` verwenden. Die neuen Operatoren sind innerhalb von `std` im darin eingebetteten Namensbereich `rel_ops` definiert.

Damit genügt es, für eigene Klassen nur `==` und `<` zu definieren; alle weiteren in den Containern verwendeten Vergleiche können darauf zurückgeführt werden.

5.5.2 Container

Container werden verwendet, um mehrere Elemente eines beliebigen Datentyps in einem Objekt zusammenzufassen, ähnlich einem Feld von Werten. Alle Werte in einem Container sind dabei vom selben Typ.

Dabei werden beim Einfügen generell **Kopien** der einzufügenden Objekte angelegt, sodaß die einzufügenden Objekte nicht dieselbe Lebensdauer haben müssen wie der Container. Wenn man für einen Container im konkreten Einzelfall einen Referenzsemantik benötigt, dann kann man das auch erreichen; siehe [Josuttis: Stdbib.] 5.10.

Alle in einem der Container verwendeten Klassen müssen folgende Mindestanforderungen erfüllen:

- Es muß ein korrekter copy-Konstruktor existieren (und am besten effizient implementiert sein).
- die Objekte müssen mit einer Zuweisung kopierbar sein.
- Für einige Container ist ein parameterloser Konstruktor nötig (Defaultkonstruktor oder ein selbst definierter).
- Es muß natürlich ein Destruktor aufrufbar sein (also `public`).
- Einige Container benötigen einen Vergleich auf Gleichheit (`operator==()`), die assoziativen Container benötigen zusätzlich den `operator<()`.
- Die hier erwähnten und damit von der STL verwendeten Funktionen sollten keine Ausnahmen werfen, da die STL diese nicht fängt und damit der innere Zustand der Containerklassen inkonsistent wird.

- Zeiger als Containerelemente sind in aller Regel ungeeignet, weil die internen Vergleiche auf *gleich* oder gegebenenfalls *kleiner* die Zeiger (also Adressen) vergleichen, und nicht die darauf verwiesenen Werte.

Es gibt folgende vollwertigen Container:

- **5.5.2.1** `vector` auf Seite 158 stellt ein eindimensionales Feld dar mit einem schnellen Zugriff auf einzelne Elemente über ihre Position (Index)
- **5.5.2.2** `queue` auf Seite 170 ist eine Art Warteschlange: an einem Ende wird angefügt (wenn sich neue Kunden am Schalter anstellen wollen, und sich schön brav am Ende anstellen), am anderen Ende werden Elemente entnommen (wenn jeweils ein Kunde am Anfang der Schlange fertig ist); also ein *FIFO* (*first in, first out*)
- **5.5.2.3** `priority_queue` auf Seite 170 sammelt Elemente ähnlich einer `queue`, aber die Entnahme erfolgt anhand der Priorität, die jedem Element zugeordnet wird (also eine bakschischgesteuerte Warteschlange)
- **5.5.2.4** `deque` auf Seite 170 ist eine *double ended queue*, also eine Warteschlange, an deren beiden Enden Elemente angefügt oder entfernt werden können
- **5.5.2.5** `list` auf Seite 170 ist eine (doppelt) verkettete Liste; an jeder Stelle kann angefügt oder entnommen werden
- **5.5.2.6** `stack` auf Seite 170 ermöglicht das Ablegen von Werten auf einem Stapel, und Entnehmen in der umgekehrten Reihenfolge (*LIFO*, *last in, first out*)
- **5.5.2.7** `map` auf Seite 170 ermöglicht die Speicherung von Paaren aus *Schlüssel* (*key*) und einem *Wert* (*value*), sowie die Suche anhand eines gegebenen Schlüssels, wobei jeder Schlüssel in der `map` nur einfach vorkommen kann
- **5.5.2.8** `multimap` auf Seite 174 ist identisch, läßt aber ein mehrfaches Vorkommen einzelner Schlüssel zu
- **5.5.2.10** `set` auf Seite 179 ist eine Ansammlung von einfach vorkommenden Werten, nach denen wieder gesucht werden kann (wenn man die Werte als Schlüssel betrachtet, weil nach ihnen gesucht werden kann, dann ist ein `set` wie eine `map`, aber ohne Werte an den Schlüsseln)
- **5.5.2.11** `multiset` auf Seite 179 ist wie `set`, aber mehrfaches Einfügen ist möglich

`map`/`multimap` und `set`/`multiset` werden dabei als *assoziative Container* bezeichnet, weil ihre Elemente über einen Schlüssel „assoziiert“ werden können. Die anderen Container bilden *sequentielle Container*; sie beinhalten eine Reihenfolge ihrer Elemente.

`vector`, `deque`, `list`, `map`, `multimap`, `set` und `multiset` haben insoweit eine identische Schnittstelle, daß *Iteratoren* (siehe **5.5.3 Iteratoren** auf Seite 179) und *Algorithmen* (siehe **5.5.4 Algorithmen** auf Seite 181) darauf angewendet können.

Die restlichen (`stack`, `queue` und `priority_queue`) sind dafür nicht geeignet. Sie sind aus den grundlegenden Containern hergeleitet, und heißen *Container-Adapter*.

Daneben gibt es noch Container, die nicht den vollen Satz an Funktionalität haben, und dafür auf ihren Einsatzzweck optimiert sind:

- **5.5.2.12** `basic_string` auf Seite 179 ist für eine Folge von Elementen (etwa eine Zeichenfolge als Textstring) gedacht.
- **5.5.2.13** `bitset` auf Seite 179 dient der kompakten Speicherung großer Mengen logischer Werte (`bool`), ohne für jeden `true`/`false`-Wert ein Byte (oder je nach Rechner auch 2 oder 4 Byte) zu verbrauchen.
- **5.5.2.14** `valarray` auf Seite 179 ist für die numerische Verarbeitung von Feldern.

Vereinbart sind diese Container jeweils in einer Headerdatei gleichen Namens, außer `priority_queue`, `multimap` und `multiset`; diese sind in `<queue>`, `<map>` beziehungsweise `<set>` mit enthalten.

5.5.2.1 `vector`

In einem `vector` wird eine (fast) beliebige Menge von Elementen so gespeichert, daß der Zugriff über einen Index (im Bereich von 0 bis (*Anzahl der Elemente*-1)) effektiv möglich ist. Dies kann über den Operator `[]` erfolgen, sodaß die Verwendung einem üblichen (eindimensionalen) Feld sehr ähnlich ist.

Allerdings ist die Anzahl der in einem `vector` gespeicherten Elemente zur Laufzeit einstellbar.

Dabei muß man allerdings beachten, daß die Feldgröße nicht durch bloße Verwendung eines Index wächst, sondern nur durch eine Anfangsgröße beim Erzeugen, durch explizites Einfügen oder Verlängern gesetzt wird (im Gegensatz zu der in **B.2.6 Felder beliebigen Typs mit variablen Grenzen** auf Seite 245 vorgestellten Lösung).

Beim Zugriff auf einzelne Elemente mit dem Operator `[]` wird –ebenso wie bei gewohnten Feldern– **keine** Feldgrenzenüberprüfung durchgeführt. Alternativ kann man mit der Methode `at( size_t index )` eine Referenz auf das Element an der Position `index` erhalten; dabei wird die Einhaltung der aktuellen Feldgrenzen geprüft und im Fehlerfall eine Ausnahme `out_of_range` geworfen (siehe **5.3 Ausnahmen der Standardbibliothek** auf Seite 143).

Ein kleines Beispiel für die Verwendung von `vector` erzeugt ein Feld von `int`-Werten:

```
// Time-stamp: "23.12.02 15:50 vector1.cpp klaus@wachtler.de"
//
// Demonstriert die Grundfunktionlität von vector<class T>.
//
// Y:\>cl /GX /W0 vector1.cpp & vector1
// v[0] = 10
// v[1] = 11
// v[2] = 0
// v[3] = 0
// v[4] = 0
// v[5] = 12
// v[6] = 13
// Element an der Stelle 10 = 49
// (hat geklappt)
// Fehler in main() entdeckt:
// invalid vector<T> subscript
```

```
// Element an der Stelle 10 = -5
// (hat geklappt)

#include <iostream>
#include <vector>
#include <stdexcept>

using namespace std;

int main( int nargs, char **args )
{

  vector<int>  v( 5 );  // 5 Elemente als Anfangsgröße

  try
  {
    v.push_back( 12 );    // ein (sechstes) Element anhängen
    v.push_back( 13 );    // ein (siebtes) Element anhängen

    v[0] = 10;      // ein Element verwenden (ohne Feldgrenzenüberprüfung)
    v.at( 1 ) = 11; // ein Element verwenden (mit Feldgrenzenüberprüfung)

    // alle Elemente ausgeben:
    for( unsigned i=0; i<v.size(); i++ )
    {
      cout << "v[" << i << "] = " << v[i] << endl;
    }

    // (vermutlich) ungestraft hinter das Feldende fassen:
    cout << "Element an der Stelle 10 = " << v[10] << endl;
    cout << "(hat geklappt)" << endl;

    // Ebenfalls hinter das Feldende fassen; aber hier kommt eine Ausnahme:
    cout << "Element an der Stelle 10 = " << v.at( 10 ) << endl;
    cout << "(hat geklappt)" << endl;
  }
  catch( exception &fehler )
  {
    cerr << "Fehler in main() entdeckt:" << endl;
    cerr << fehler.what() << endl;
  }
  catch( ... )
  {
```

```
    cerr << "Unbekannter Fehler in main() entdeckt" << endl;
  }
  try
  {
    // Feld verlängern, dann kommt auch keine Ausnahme!
    // auf 11 Elemente (0..10) verlängern, neue Werte stehen auf -5:
    v.resize( 11, -5 );
    cout << "Element an der Stelle 10 = " << v.at( 10 ) << endl;
    cout << "(hat geklappt)" << endl;
  }
  catch( exception &fehler )
  {
    cerr << "Fehler in main() entdeckt:" << endl;
    cerr << fehler.what() << endl;
  }
  catch( ... )
  {
    cerr << "Unbekannter Fehler in main() entdeckt" << endl;
  }

}
```

Die Ausgabe dieses Programms ist:

```
v[0] = 10
v[1] = 11
v[2] = 0
v[3] = 0
v[4] = 0
v[5] = 12
v[6] = 13
Element an der Stelle 10 = 134529592
(hat geklappt)
Fehler in main() entdeckt:
vector [] access out of range
Element an der Stelle 10 = -5
(hat geklappt)
```

Ein weiteres Beispiel zeigt neben der Erzeugung des Vektorinhalts auch die üblichen Möglichkeiten, über die vorhandenen Elemente zu iterieren:

```
// Time-stamp: "30.10.04 10:52 vec.cpp klaus@wachtler.de"
//
// demonstriert std::vector
//
// 25.10.2004 kw   unter Linux entworfen und getestet, mit VC6 nicht
//                 kompilierbar
//
// 27.10.2004 kw   Mit VC6 (cl 12.00.8168) getestet unter Windows 2000
//                 Übersetzen: cl /EHsc vec.cpp
//
// 29.10.2004 kw   Getestet mit MS VC++ 7.1 (VCToolkit),
//                 cl-Version 13.10.3077 unter Windows 2000
//                 Übersetzen: cl /EHsc vec.cpp
//
//                 Außerdem mit devcpp (g++ 3.3.1 mingw) getestet
//                 unter Windows 2000.
//                 Übersetzen mit g++ -Wall vec.cpp -o vec.exe
//
//                 Nach wie vor unter Linux mit g++ 3.3.4 lauffähig
//                 (g++ -Wall vec.cpp -o vec)
//
// 30.10.2004 kw   demo_foreach_fo() eingefügt
//

#include <iostream>
#include <vector>
#include <stdexcept>
#include <algorithm>


// Gibt einen beschreibenden (text) und anschließend einen vector
// Element für Element aus:
template<typename T> void zeigevector( const char *text, std::vector<T> v )
{
  std::cout << "vector\n" << text << std::endl;
  for( size_t i=0; i<v.size(); i++ )
  {
    std::cout << "[" << i << "] = " << v[i] << std::endl;
  }
  std::cout << std::endl;
}

std::vector<int> demo_simple()
```

```
{
  size_t   i;

  // vector aus int-Werten:
  std::vector<int>   vint(5);  // Anfangslänge 5

  for( i=0; i<vint.size(); i++ )
  {
    vint[i] = i*i;
  }

  zeigevector( "vint anfangs", vint );

  try
  {
    // Das Element an der Stelle 5 ist unzulässig!
    // Mit [5] kann trotzdem zugegriffen werden (auch auf die Gefahr
    // hin, daß das Programm Unsinn macht):
    std::cout << "vint[5]    = " << vint[5] << std::endl;

    // Hier wird dagegen zwar ebenfalls versucht, auf ein unzulässiges
    // Element zuzugreifen; aber bei at() wird dies zur Laufzeit
    // erkannt und eine Ausnahme geworfen:
    std::cout << "vint.at(5) = " << vint.at(5) << std::endl;

  }
  catch( std::exception &Fehler )
  {
    std::cerr << "Fehler: <" << Fehler.what() << "> in demo_simple()\n";
  }
  catch( ... )
  {
    std::cerr << "unbekannter Fehler in demo_simple()\n";
  }

  try
  {
    // Wenn man ein Element anhängt, geht alles klar:
    vint.push_back( 99 ); // verlängert um ein Element
    std::cout << "vint[5]    = " << vint[5] << std::endl;
    std::cout << "vint.at(5) = " << vint.at(5) << std::endl;

  }
```

```
  catch( std::exception &Fehler )
  {
    std::cerr << "Fehler: <" << Fehler.what() << "> in demo_simple()\n";
  }
  catch( ... )
  {
    std::cerr << "unbekannter Fehler in demo_simple()\n";
  }

  // Den vector etwas verlängern:
  vint.resize( 10 );
  for( i=5; i<vint.size(); i++ )
  {
    vint[i] = i*i;
  }

  // neuen Inhalt ausgeben:
  zeigevector( "vint nach Verlängern", vint );

  return vint;
}

// zeigt alle Elemente, Zugriff mit []
void demo_index( std::vector<int>  vint )
{
  std::cout << "demo_index:" << std::endl;
  // normale Schleife über alle Elemente:
  size_t  i;
  for( i=0; i<vint.size(); i++ )
  {
    std::cout << "vint[" << i << "] = " << vint[i] << std::endl;
  }
  std::cout << std::endl;
}

// zeigt alle Elemente, Zugriff mit at()
void demo_at( std::vector<int>  vint )
{
  std::cout << "demo_at:" << std::endl;
  // normale Schleife über alle Elemente:
  size_t  i;
  for( i=0; i<vint.size(); i++ )
  {
```

```
    std::cout << "vint.at( " << i << " ) = " << vint.at( i ) << std::endl;
  }
  std::cout << std::endl;
}

// zeigt alle Elemente, Zugriff mit einem Iterator
void demo_iterator( std::vector<int>  vint )
{
  std::cout << "demo_iterator:" << std::endl;
  // Iteration über alle Elemente:
  std::vector<int>::iterator  i;
  for( i=vint.begin(); i!=vint.end(); i++ )
  {
    std::cout << "*i" << " = " << *i << std::endl;
  }
  std::cout << std::endl;
}

// zeigt alle Elemente, Zugriff mit for_each und einer Funktion
// gibaus():

void gibaus( int i )
{
  std::cout << "i" << " = " << i << std::endl;
}


void demo_foreach_fkt( std::vector<int>  vint )
{
  std::cout << "demo_foreach_fkt:" << std::endl;
  // Schleife aus <algorithm> über alle Elemente; was mit jedem Objekt
  // zu geschehen hat, wird von der Funktion gibaus() definiert:
  std::for_each( vint.begin(), vint.end(), gibaus );
  std::cout << std::endl;
}

// zeigt alle Elemente, Zugriff mit for_each und einem Funktionsobjekt
// der Klasse gib:

// Ein Objekt dieser Klasse kann erstens mit einer int als Parameter
// aufgerufen werden (und gibt dann das Element aus), und zweitens
// werden diese Aufrufe mitgezählt und die Anzahl kann mit
// getAnzahlAufrufe() abgefragt werden.
```

```
class gib
{
public:

  // Konstruktor
  gib()
    : anzahlAufrufe( 0 )
  {}

  // Mit diesem Operator kann ein Objekt o dieser Klasse wie eine
  // Funktion mit einem int-Parameter aufgerufen werden, z.B.:
  //    o( 5 );
  void operator()( const int &aktuellesObjekt )
  {
    std::cout << "aktuellesObjekt = "
              << aktuellesObjekt
              << std::endl;
    anzahlAufrufe++;
  }

  // liefert zurück, wieoft operator()( const int &aktuellesObjekt )
  // schon für das aktuelle Objekt (*this) aufgerufen wurde:
  int getAnzahlAufrufe()
  {
    return anzahlAufrufe;
  }

private:

  int   anzahlAufrufe;

}; // class gib...

void demo_foreach_fo( std::vector<int>  vint )
{
  std::cout << "demo_foreach_fo" << std::endl;

  // Schleife aus <algorithm> über alle Elemente.
  // Was mit jedem Objekt zu geschehen hat, wird von dem
  // Funktionsobjekt fo definiert:
  gib  fo; // Funktionsobjekt, das als fo( int ) verwendet werden kann
           // (also wie eine Funktion, durch den operator() in gib).
           // Die Anzahl der Aufrufe wird intern mitgezählt.
```

```
  // Wenn man den Rückgabewert von for_each() nicht verwendet, kann
  // man anschließend nicht mehr die Anzahl der in fo gespeicherten
  // Aufrufe feststellen, weil fo an for_each mit call by referene
  // übergeben wird, also eine Kopie angelegt wird. In dieser Kopie
  // werden dann die Aufrufe gezählt; der Zähler in fo bleibt auf 0
  // stehen. Glücklicherweise liefert for_each() aber seine Kopie
  // wieder als Rückgabewert zurück; daher die Zuweisung an fo:
  fo = std::for_each( vint.begin(), vint.end(), fo );

  std::cout << std::endl;

  // Wieviele Elemente wurden ausgegeben?
  std::cout << "Es wurden " << fo.getAnzahlAufrufe()
            << " Elemente ausgegeben." << std::endl;
}

int main( int nargs, char **args )
{
  std::vector<int>   v;
  v = demo_simple();

  demo_index( v );
  demo_at( v );
  demo_iterator( v );
  demo_foreach_fkt( v );
  demo_foreach_fo( v );

  return 0;
} // main( int nargs, char **args )
```

Dazu ein paar Anmerkungen. Die Iteration über alle Elemente erfolgt:

- in `demo_index()` mit einem Feldindex, der von `0` bis zur Nummer des letzten Elements hochgezählt wird; auf das gewünschte Element wird zugegriffen mit den eckigen Klammern (`[...]`) wie bei einem einfachen Feld auch (ohne Überprüfung der Feldgenzen)
- in `demo_at()` ähnlich wie in `demo_index()`, aber anstatt mit `[...]` wird das jeweilige Element mit `at(...)` ausgewählt; dabei wird die Einhaltung der Feldgrenzen überprüft

- in `demo_iterator()` durch Verwendung eines Iterators (siehe auch **5.5.3 Iteratoren** auf Seite 179)

- in `demo_foreach_fkt()` mit der (`template-`) Funktion `std::for_each` aus `<algorithm>`, der eine Sequenz übergeben wird (für den gesamten Container also `begin()` und `end()`) sowie eine Funktion `gibaus()`[1], welche angibt, was mit jedem Element zu geschehen hat (siehe **5.5.4 Algorithmen** auf Seite 181).

- in `demo_foreach_fo()` ebenfalls mit `std::for_each`, allerdings nicht durch Übergabe einer Funktion, sondern eines Funktionsobjekts stattdessen. Dabei wird ausgenutzt, daß `std::for_each` ohnehin nur ein `template` ist und letztlich vom Compiler mit einer Art Textersetzung ausgewertet wird.

  Innerhalb dieses `template` wird der dritte Parameter verwendet, indem ihm das jeweils aktuelle Element (hier: eine `int`) des Vektors wie bei einem Funktionsaufruf übergeben wird. Dadurch muß dieser dritte Parameter an `std::for_each` nicht unbedingt wirklich eine Funktion sein, sondern man kann auch ein Objekt übergeben, für das der `operator()( int )` passend überladen ist.

  Diese Vorgehensweise sieht auf den ersten Blick ziemlich umständlich aus, eröffnet aber leicht weitere Möglichkeiten. Im Beispiel ist in dem übergebenen Objekt noch ein Zähler eingebaut, in dem die bisherige Anzahl der Aufrufe gespeichert wird. Nach dem Ende der Schleife kann dann die Anzahl der Aufrufe und damit der ausgegebenen Elemente festgestellt werden.

  Funktionsobjekte haben also die Möglichkeiten, neben ihrer eigentlichen Funktionalität (hier: die Ausgabe eines Elements) noch weitere Informationen zu tragen.

  Siehe auch **C.4 Funktionsobjekte** auf Seite 291.

Die Ausgabe des Programms ist dementsprechend:

```
vector
vint anfangs
[0] = 0
[1] = 1
[2] = 4
[3] = 9
[4] = 16
```

[1]Genaugenommen wird die Adresse der Funktion übergeben

```
vint[5]    = 0
Fehler: <vector [] access out of range> in demo_simple()
vint[5]    = 99
vint.at(5) = 99
vector
vint nach Verlängern
[0] = 0
[1] = 1
[2] = 4
[3] = 9
[4] = 16
[5] = 25
[6] = 36
[7] = 49
[8] = 64
[9] = 81

demo_index:
vint[0] = 0
vint[1] = 1
vint[2] = 4
vint[3] = 9
vint[4] = 16
vint[5] = 25
vint[6] = 36
vint[7] = 49
vint[8] = 64
vint[9] = 81

demo_at:
vint.at( 0 ) = 0
vint.at( 1 ) = 1
vint.at( 2 ) = 4
vint.at( 3 ) = 9
vint.at( 4 ) = 16
vint.at( 5 ) = 25
vint.at( 6 ) = 36
vint.at( 7 ) = 49
vint.at( 8 ) = 64
vint.at( 9 ) = 81

demo_iterator:
*i = 0
```

```
*i = 1
*i = 4
*i = 9
*i = 16
*i = 25
*i = 36
*i = 49
*i = 64
*i = 81

demo_foreach_fkt:
i = 0
i = 1
i = 4
i = 9
i = 16
i = 25
i = 36
i = 49
i = 64
i = 81

demo_foreach_fo
aktuellesObjekt = 0
aktuellesObjekt = 1
aktuellesObjekt = 4
aktuellesObjekt = 9
aktuellesObjekt = 16
aktuellesObjekt = 25
aktuellesObjekt = 36
aktuellesObjekt = 49
aktuellesObjekt = 64
aktuellesObjekt = 81

Es wurden 10 Elemente ausgegeben.
```

5.5.2.2 `queue`

5.5.2.3 `priority_queue`

5.5.2.4 `deque`

5.5.2.5 `list`

5.5.2.6 `stack`

5.5.2.7 `map`

Eine Map enthält Wertepaare (siehe **C.3 pair** auf Seite 291).

Die Typen des ersten und zweiten Elements in einem Paar können unterschiedlich sein.

Das erste Element im Paar wird als *Schlüssel* betrachtet, das zweite ist der zugehörige Wert. Eine Map könnte also Namen als Schlüssel und die zugehörigen Telefonnummern als Wert enthalten

Wertepaare werden komplett eingefügt (also immer ein Schlüssel und ein Wert zusammen).

Der Zugriff auf die Elemente erfolgt typischerweise entweder, indem anhand eines Schlüssel der zugehörige Wert gesucht wird (z.B. anhand eines Namens die passende Telefonnummer), oder indem alle Elemente durchlaufen werden und bei jedem Durchlauf ein Paar aus Schlüssel und Wert zum Vorschein kommt.

Eine Map ist also im Prinzip die Übersetzung der Werte einer Menge (Schlüssel, z.B. Namen) in jeweils einen Wert einer anderen Menge (Telefonnummer).

Für die Suche nach einem vorhandenen Element kann man entweder die Methode `find()` benutzen, oder den `[]`-Operator wie bei einem Feldzugriff; allerdings ist der Suchschlüssel nicht unbedingt ganzzahlig, sondern natürlich vom Typ des Schlüssels.

Es ist vielleicht überraschend (weil sich Felder ganz anders verhalten), daß man sowohl mit `[]` als auch mit `find()` auf Elemente zugreifen kann, die bisher gar nicht existieren. Dann werden die Elemente umgehend angelegt (mit ihrem parameterlosen beziehungsweise ihrem Defaultkonstruktor).

Will man testen, ob ein Element existiert, geht man einen Umweg über einen Iterator, den man mit `end()` vergleichen kann. Wenn der Iterator gleich `end()` ist, dann verweist er auf kein gültiges Objekt.

Dazu ein kleines Beispielprogramm:

```
// Time-stamp: "05.01.06 11:24 map.cpp klaus@wachtler.de"
//
// demonstriert eine std::map
//
// 25.10.2004 kw   unter Linux entworfen und getestet, mit VC6 nicht
//                 kompilierbar
//
// 27.10.2004 kw   An VC6 (cl 12.00.8168) angepaßt unter Windows 2000:
//                  - zeigealter() gibt nicht mehr p.first aus,
//                    sondern p.first.c:str()
//                  - map bekommt als dritten Parameter ein
//                    Vergleichsargument für string (Typ stringcmp)
//                 Übersetzen: cl /EHsc map.cpp
//
// 29.10.2004 kw   Getestet mit MS VC++ 7.1 (VCToolkit),
//                 cl-Version 13.10.3077 unter Windows 2000
//                 Übersetzen: cl /EHsc map.cpp
//
//                 Außerdem mit devcpp (g++ 3.3.1 mingw) getestet
//                 unter Windows 2000.
//                 Übersetzen mit g++ -Wall map.cpp -o map.exe
//
//                 Nach wie vor unter Linux mit g++ 3.3.4 lauffähig
//                 (g++ -Wall map.cpp -o map)
//

#include <functional>
#include <algorithm>
#include <iostream>
#include <map>
#include <cstring>
#include <stdexcept>
#include <utility>
```

```
void zeigealter( const std::pair<std::string,int> p )
{
  std::cout << p.first.c_str() // c_str() ist nur für VC6 nötig; Linux
                               // bzw. g++ 3.3.2 kann strings direkt
                               // ausgeben (p.first).
            << " ist " << p.second
            << " Jahre alt"
            << std::endl;
}

// Für VC6 + VC7 nötig (sonst können strings in der map nicht verglichen
// werden):
class stringcmp
{
public:
  // Diese Funktion soll true liefern, wenn s1 kleiner als s2 ist.
  // Mit VC6 kann man leider strings nicht direkt vergleichen; deshalb
  // kommt die gute alte Funktion strcmp() zum Zuge:
  bool operator()( const std::string &s1, const std::string &s2 ) const
  {
    return strcmp( s1.c_str(), s2.c_str() ) < 0;
  }
}; // class stringcmp...

// Unter Linux (g++ 3.3.2) kann der dritte Parameter (stringcmp) der
// map entfallen:
//typedef std::map< const std::string, int > container;
typedef std::map< const std::string, int, stringcmp > container;

int main( int nargs, char **args )
{
  try
  {
    container    alter;

    // effektives Einfügen:
    alter.insert( std::make_pair(std::string( "Klaus" ), 42 ) );
    alter.insert( std::make_pair(std::string( "Otto" ), 12 ) );
    alter.insert( std::make_pair(std::string( "Fritz" ), 18 ) );
    alter.insert( std::make_pair(std::string( "Kuno" ), 56 ) );
    alter.insert( std::make_pair(std::string( "Gandalf" ), 99 ) );
```

```
    // geht auch zum Einfügen, aber langsamer:
    alter["Hund"] = 3;

    // Ausgabe aller Elemente der map:
    std::for_each( alter.begin(), alter.end(), zeigealter );

    // Zugriff auf ein vorhandenes Element über find():
    std::cout << "Fritz ist " << (alter.find("Fritz"))->second  << " alt"
              << std::endl;

    // Ebenso mit operator[]:
    std::cout << "Fritz ist " << alter["Fritz"]  << " alt"
              << std::endl;
    // "Fritzchen" existiert nicht; aber das ergibt keinen Fehler,
    // sondern es wird ein neues Element angelegt (mit dem Alter 0);
    // in dieser Form ist der Zugriff ziemlich unsinnig:
    std::cout << "Fritzchen ist " << alter["Fritzchen"]  << " alt"
              << std::endl;

    // Besser: Feststellen, ob es Fritzl gibt:
    container::iterator fritzl = alter.find("Fritzl");
    if( fritzl!=alter.end() )
    {
      // Wenn ja: ausgeben
      std::cout << "Fritzl ist "
                << fritzl->second  << " alt"
                << std::endl;
    }
    else
    {
      // Wenn nein: natürlich nicht ausgeben
      std::cout << "Fritzl gibt es nicht!" << std::endl;
    }
  }
  catch( std::exception &Fehler )
  {
    std::cerr << "Fehler: <" << Fehler.what() << "> in main()\n";
  }
  catch( ... )
  {
    std::cerr << "unbekannter Fehler in main()\n";
  }
```

```
  return 0;
} // main( int nargs, char **args )
```

Die Ausgabe davon:

```
Fritz ist 18 Jahre alt
Gandalf ist 99 Jahre alt
Hund ist 3 Jahre alt
Klaus ist 42 Jahre alt
Kuno ist 56 Jahre alt
Otto ist 12 Jahre alt
Fritz ist 18 alt
Fritz ist 18 alt
Fritzchen ist 0 alt
Fritzl gibt es nicht!
```

5.5.2.8 `multimap`

Eine `multimap` ist einer `map` (**5.5.2.7** `map` auf Seite 170) sehr ähnlich.

Allerdings ist es hier erlaubt, Elemente mit dem selben Suchschlüssel (und bei Bedarf auch vollkommen identisch, also selber Schlüssel und selber Wert) mehrfach einzufügen.

Dazu ein kleines Beispielprogramm:

```
// Time-stamp: "02.11.04 00:14 mumap.cpp klaus@wachtler.de"
//
// demonstriert eine std::multimap
//
// 01.11.2004 kw   aus map.cpp übernommen
//

#include <functional>
#include <algorithm>
#include <iostream>
#include <map>
#include <cstring>
#include <stdexcept>
```

```
#include <utility>


void zeigealter( const std::pair<std::string,int> p )
{
  std::cout << p.first.c_str() // c_str() ist nur für VC6 nötig; Linux
                               // bzw. g++ 3.3.2 kann strings direkt
                               // ausgeben (p.first).
            << " ist " << p.second
            << " Jahre alt"
            << std::endl;
}

// Für VC6 + VC7 nötig (sonst können strings in der map nicht verglichen
// werden):
class stringcmp
{
public:
  // Diese Funktion soll true liefern, wenn s1 kleiner als s2 ist.
  // Mit VC6 kann man leider strings nicht direkt vergleichen; deshalb
  // kommt die gute alte Funktion strcmp() zum Zuge:
  bool operator()( const std::string &s1, const std::string &s2 ) const
  {
    return strcmp( s1.c_str(), s2.c_str() ) < 0;
  }
}; // class stringcmp...

// Unter Linux (g++ 3.3.2) kann der dritte Parameter (stringcmp) der
// multimap entfallen:
//typedef std::multimap< const std::string, int > container;
typedef std::multimap< const std::string, int, stringcmp > container;

int main( int nargs, char **args )
{
  try
  {
  }
  catch( std::exception &Fehler )
  {
    std::cerr << "Fehler: <" << Fehler.what() << "> in main()\n";
  }
  catch( ... )
  {
```

```
    std::cerr << "unbekannter Fehler in main()\n";
  }

  return 0;
} // main( int nargs, char **args )
```

Die Ausgabe davon:

5.5.2.9 hash_map

TODO

Nicht im Standard, aber in SGI-STL vorhanden

```
http://www.heise.de/foren/go.shtml?t=1&T=thread%20mfgkw&sres=1&msg_id=5132619&forum_
19. Februar 2004 13:43
Re: Hash Tables mit C++
Klaus Wachtler

Erik Meusel schrieb am 19. Februar 2004 13:07

> Klaus Wachtler schrieb am 19. Februar 2004 9:56
> ...
> Danke für Dein Beispiel im anderen Posting, mir fehlte bisher im
> Thread einfach der Hinweis darauf, wie man eine Hashfunktion angeben
> kann, das ist alles. ;)
> ...

Dann hilft vielleicht noch folgendes:
Für eine eigene Hashfunktion kann man sich aussuchen, ob man
die template class hash spezialisiert (was man für einen gegebenen
Datentyp des Schlüssels nur einmal machen kann in einem
Programm), oder ob man eine "normale" Funktion schreibt, und dem
Konstruktor der hash_map oder hash_set übergibt.
Dadurch könnte man für mehrere Hashtabellen gleichen Schlüsseltyps
verschiedene Hashfunktionen verwenden.
```

```
Beispiel 1 (Spezialisierung hash<double>; damit man hash
spezialisieren
kann, habe ich als Schlüsseltyp double genommen, auch wenn es
eigentlich
dämlich ist):
#include <iostream>
#include <exception>
#include <cstddef>
#include <ext/hash_map>

using namespace std;
using namespace __gnu_cxx;


struct eqdouble
{
  bool operator()(double d1, double d2) const
  {
    return  d1 == d2;
  }
};

// Typ eines Funktionsobjekts als Parameter an template
namespace __gnu_cxx
{
  template<> struct hash<double>
  {
    size_t operator()( double d ) const
    {
      return  (size_t)(fabs( d ));
    }
  };
};

int main( int nargs, char **args )
{
  hash_map<double,int,hash<double>,eqdouble>  Map;

  Map[3.14] = 3;
  Map[2.7] = 2;
  Map[2.8] = 15;

  cout << "2.7 -> " << Map[2.7] << endl;
```

```
  hash_map<double,int,hash<double>,eqdouble>::iterator   i;
  double key = 2.9;
  if( (i=Map.find( key ))!=Map.end() )
  {
    cout << i->first << " -> " << i->second << endl;
  }
  else
  {
    cout << key << " nicht gefunden!" << endl;
  }
} // main( int nargs, char **args )
```

Beispiel 2 (das selbe, aber mit einer echten Funktion):

```
#include <iostream>
#include <exception>
#include <cstddef>
#include <ext/hash_map>

using namespace std;
using namespace __gnu_cxx;


struct eqdouble
{
  bool operator()(double d1, double d2) const
  {
    return  d1 == d2;
  }
};

// echte Funktion als Hash-Funktion:
size_t hash_d( double d )
{
  return  (size_t)(fabs( d ));
}

int main( int nargs, char **args )
{
  hash_map<double,int,size_t(*)(double),eqdouble>  Map(10,hash_d);

  Map[3.14] = 3;
```

```
  Map[2.7] = 2;
  Map[2.8] = 15;

  cout << "2.7 -> " << Map[2.7] << endl;

  hash_map<double,int,size_t(*)(double),eqdouble>::iterator    i;
  double key = 2.9;
  if( (i=Map.find( key ))!=Map.end() )
  {
    cout << i->first << " -> " << i->second << endl;
  }
  else
  {
    cout << key << " nicht gefunden!" << endl;
  }

} // main( int nargs, char **args )


Beides mit g++ 3.3.2 probiert.

mfgkw
```

5.5.2.10 set

5.5.2.11 multiset

5.5.2.12 basic_string

5.5.2.13 bitset

5.5.2.14 valarray

5.5.3 Iteratoren

TODO Strou4 587

Ein *Iterator* ist ein Objekt, das zu einem gegebenen Container ein Objekt daraus repräsentiert (oder auch keines möglicherweise).

Einen Iterator kann man manipulieren, indem man ihm einen Wert zuweist (beispielsweise das erste Element des Containers), und indem man ihn um ein Element vor- oder zurücksetzt.

Damit erfüllt ein Iterator in etwa die selbe Aufgabe wie ein Zeiger auf ein Feldelement: auch dieses kann man auf ein Element richten, sowie mit `++` oder `--` über die Elemente des Feldes wandern lassen. Tatsächlich können Iteratoren als Zeiger auf den zugrundeliegenden Datentyp eines Containers implementiert werden, wobei natürlich `operator++()` und `operator--()` so überschrieben werden müssen, daß sie den nicht auf ein im Speicher davor oder dahinter liegendes Objekt richten, sondern dem Container entsprechend das benachbarte Element finden.

Iteratoren sind in `<iterator>` definiert. Wenn man Iteratoren allerdings auf die Standardcontainer anwendet, dann braucht dafür kein `#include` geschrieben werden, weil die Headerdatei des jeweiligen Containers (`<vector>` etc.) ohnehin `<iterator>` einbindet.

Folgende Operationen sind mit Iteratoren möglich:

- Erzeugen eines Iterators.

  Beispiel:

  ```
  vector<int>    meinFeld;
  vector<int>::iterator  position;
  ```

  Hier wird ein Feld mit dem Namen `meinFeld` definiert, sowie ein Iterator `position`.

- Zuweisen eines Iterators an einen anderen: Damit repräsentiert der Iterator, der auf der linken Seite der Zuweisung steht, das selbe Objekt wie der Iteratorausdruck der rechten Seite.

  Beispiel:

Man kann Iteratoren allerdings auch auf eigene Container anwenden, wenn sie ein paar Mindestanforderungen erfüllen:

- eine Memberfunktion `begin()` liefert einen Iterator auf das erste Objekt
- eine Memberfunktion `end()` liefert einen ungültigen Iterator, der als „hinter dem letzten Element“ zu verstehen ist

- ein Datentyp `iterator`

  Wenn der Iterator beispielsweise ein Zeiger auf den zugrundeliegenden Datentyp sein soll, dann könnte die Definition so aussehen:

- TODO was noch?

5.5.4 Algorithmen

5.6 Strings

5.7 Numerik

5.8 Bitsets

5.9 Streams

`istream`, `ostream`, `iostream`

`istringstream`, `ostringstream`, `stringstream` (`include <sstream>`)

5.9.1 Ein-/Ausgabe mit eigenen Klassen

5.10 Internationalisierung

5.11 Allokatoren

5.12 Sonstiges

Anhang A

Empfehlungen

Der bisherige Text zeigt im Wesentlichen, was man in *C++* *darf* und was man bleiben lassen *muß*.

Es gibt jedoch einige Punkte, die man beachten *sollte*, auch wenn dies von der Sprache nicht direkt erzwungen wird.

Im Folgenden werden solche Regeln aufgeführt, wobei es teilweise natürlich um eine Frage des persönlichen Programmierstils geht. Insofern sollte man die aufgeführten Empfehlungen nicht sklavisch einhalten; entweder durch die Umstände im Einzelfall oder durch einen generell anderen Programmierstil mag es durchaus Gründe für ein anderes Verhalten geben. Überdenkenswert sind die Punkte aber in jedem Fall.

A.1 Auch Teile von *C++* können hilfreich sein

Man muß nicht den gesamten Umfang von *C++* beherrschen, um sinnvoll damit arbeiten zu können.

C++ hat viele Aspekte, die für sich genommen sinnvoll sein können, aber man muß in einem Programm ja nicht alles einbauen, was die Sprache bietet.

Beispielsweise kann man durchaus die Ausnahmebehandlung sinnvoll nutzen, ohne die *standard template library* (STL) zu kennen, oder man kann die STL nutzen, ohne *run time type information* (RTTI) verstanden zu haben.

Bei kleinen Projekten braucht man sicher nicht alles, was die Sprache bietet. Erst bei zunehmender Projektgröße wird man mehr verstehen müssen.

Insofern braucht man sich nicht gleich zu Anfang um alles zu kümmern. Ein solide Basis in *C* und ein gewisses Grundverständnis für *C++* reichen aus, um die Aspekte zu finden, mit denen man eine konkrete Aufgabe lösen kann. Wenn man sich dann in diese Teile der Sprache einarbeitet, reicht das um die Sprache angemessen zu nutzen.

Dazu ein Kommentar von Bjarne Stroustrup aus einem Interview in [ct 9/94 Hueskes]:

> *c't: Aber die Komplexität der Programmiersprache erhöht doch die Einarbeitungszeit.*
>
> Stroustrup: Viele fortgeschrittene und neue Errungenschaften von C++ ermöglichen das Erstellen besserer Bibliotheken. Der Einsteiger braucht sie nicht zu verstehen.
>
> C++ wurde so entworfen, daß es Schritt für Schritt erlernbar ist und beim Lernen bereits Nutzen bringt. Es ist keine Sprache, der man schnell entwächst - auch ich entdecke noch interessante neue Aspekte und Techniken.

A.2 Aufteilung auf mehrere Quelltexte

Da ein Programm in aller Regel nicht in einem Stück geschrieben wird, bietet es sich an, die Definition einer Klasse in eine eigene Datei zu schreiben und in allen Programmteilen, welche die Klasse verwenden, eine passende Headerdatei mit `#include ...` einzufügen.

Solche einzelnen Module sind übersichtlicher, besser wartbar, und leichter wieder zu verwenden.

Prinzipiell gilt dieses Kapitel nicht nur für *C++*, sondern ebenso für *C*. Dort wird aber natürlich nicht eine Klasse in einem solchen Modul abgelegt, sondern thematisch zusammengehörige Funktionen und Datentypen; eben genau das, was man in *C++* zu einer Klasse zusammenfaßt.

Das eben genannte Beispiel mit der Klasse `ErsteKlasse` könnte also so aufgeteilt werden:

Jede Klasse (im Beispiel nur `ErsteKlasse`) sollte für sich eine funktionale Einheit bilden, und genau einem Quelltext zur Definition (ErsteKlasse.cpp) und einem zur Deklaration (ErsteKlasse.h) entsprechen.

Der Dateiname sollte (von der Endung abgesehen) möglichst eng aus dem Klassennamen abgeleitet sein. Die unter MSDOS und ähnlichen Systemen bekannten Beschränkungen des Dateinamens auf 8 Zeichen (zuzüglich Endung) sind inzwischen Geschichte; sogar die aktuellen Windowsversionen vertragen hier mehr. Leider gibt es aber nach wie beschränkte Systeme, die nicht zwischen Groß- und Kleinschreibung unterscheiden. Trotzdem ist man gut geraten, wenn man die Groß-/Kleinschreibung der Klassennamen auch in den `include`-Anweisungen und den Dateinamen konsequent einhält, weil immer wieder zumindest Teile eines Projekts auf fortschrittlichere Systeme portiert werden müssen, wo dann die Schreibweise beachtet wird.

ErsteKlasse.h enthält alle Deklarationen, die für die Definition und Verwendung der Klasse benötigt werden:

```
// Time-stamp: "(13.11.01 13:18) ErsteKlasse.h [Klaus Wachtler (aw38)]"
//
// Vereinbarung (Deklaration) der Klasse ErsteKlasse:

#ifndef _ERSTEKLASSE_H_
#define _ERSTEKLASSE_H_
```

```
class ErsteKlasse
{
private:

    int     zahl1;
    int     zahl2;

public:

    void setzezahlen( int z1, int z2 );
    int summe( void );

};

#endif // defined _ERSTEKLASSE_H_
```

Die Elementfunktionen werden innerhalb der `class`-Anweisung jetzt nicht mehr definiert, sondern nur deklariert.

Damit ein mehrfaches `#include`n nicht zu Problemen führt, wird am Anfang jeder Headerdatei ein Makro definiert, das aus dem Dateinamen abgeleitet wird (hier: `_ERSTEKLASSE_H_`), und alles mit einem `#ifndef _ERSTEKLASSE_H_` eingerahmt. Dadurch wird der gesamte Inhalt der Datei nur beim ersten `#include`n wirksam.

Eine weitere Datei (ErsteKlasse.cpp) nimmt die Definition der Elementfunktionen auf (Implementation der Klasse):

```
// Time-stamp: "(13.11.01 13:19) ErsteKlasse.cpp [Klaus Wachtler (aw38)]"
//
// kleine Demonstration einer Klasse
//
// In dieser Datei werden die Elementfunktionen der Klasse
// definiert.

#include "ErsteKlasse.h"   // Hiermit ist die Klasse mit dem
                           // Namen ErsteKlasse deklariert.


void ErsteKlasse::setzezahlen( int z1, int z2 )
{
    zahl1 = z1;
    zahl2 = z2;
```

```
}

int ErsteKlasse::summe( void )
{
    return zahl1 + zahl2;
}
```

Nach den üblichen Anfangskommentaren werden in den Quelltext ziemlich zu Anfang die zugehörigen Deklarationen mit `#include "ErsteKlasse.h"` eingefügt. Weil die Elementfunktionen innerhalb der `class`-Anweisung nur noch deklariert werden, weiß der Compiler nicht automatisch, daß die Funktionen `setzezahlen()` und `summe()` zu der Klasse `ErsteKlasse` gehören. Daher muß man ihm dies mit dem *scope resolution*-Operator `::` mitteilen.

Da das „Anwendungsprogramm" (hier: klasse1t.cpp) eher klein ist, bildet es nur ein Modul ohne eigene Headerdatei:

```
// Time-stamp: "(15.11.01 16:10) klasse1t.cpp [Klaus Wachtler (aw38)]"

// Kleines Testprogramm zu der Klasse ErsteKlasse

#include <iostream>

using namespace std;

#include "ErsteKlasse.h"   // Hiermit ist die Klasse mit dem
// Namen ErsteKlasse deklariert.

int main( int nargs, char *args[] )
{
  // von der Klasse ErsteKlasse werden zwei Instanzen a und b erzeugt:
  ErsteKlasse
    a,
    b;

  // a wird auf { 2, 3 } und b auf { 10, 12 } gesetzt:
  a.setzezahlen( 2, 3 );
  b.setzezahlen( 10, 12 );

    // Die Summe a.zahl1+a.zahl2 ausgeben:
  cout << "a hat den Wert " << a.summe() << "\n";
  // dto. mit b:
```

```
  cout << "b hat den Wert " << b.summe() << "\n";

  return 0;
}
```

In echten Programmen hat man als Anwendung meist mehrere Module. Jedes dieser Module holt sich gegebenenfalls über `#include "ErsteKlasse.h"` die nötigen Vereinbarungen und kann `ErsteKlasse` wie einen eingebauten Datentyp verwenden, indem es Objekte des Typs (Instanzen der Klasse) vereinbart. Die kompilierten Module werden dann mit dem Kompilat von ErsteKlasse.cpp zusammengelinkt.

Hat man weitere Programmteile, die für sich genommen zu einem Modul zusammenpassen (ohne eine Klasse zu bilden), dann schreibt man auch hier zu jedem Modul einen Quelltext mit den Definitionen und eine Headerdatei mit den Deklarationen.

Damit wird eine sehr günstige Aufteilung des Quelltextes erreicht: die anwendenden Programmteile sehen nur Deklarationen, ohne die Repräsentation der Elementfunktionen zu kennen oder die geschützten Elemente direkt anfassen zu können. Andererseits kann in der Definition der Elementfunktionen die Implementation der Methoden geändert werden, ohne die Schnittstelle zu ändern. Dadurch bleiben die anwendenden Programmteile von solchen Änderungen unberührt; sie müssen gegebenenfalls nicht neu kompiliert oder gar geändert werden. So sind Funktion und Verwendung der Klassen vollkommen trennbar.

Zu beachten ist aber, daß `inline`-Funktionen dem Compiler dann bekannt sein müssen, wenn er ihren „Aufruf“ übersetzt. Solche Elementfunktionen gehören deshalb nicht in das Implementationsmodul (hier ErsteKlasse.cpp), sondern in die Deklarationsdatei (ErsteKlasse.h). Elementfunktionen werden (wenn sie klein und damit für `inline` geeignet sind) dann oft gleich in den Block der zugehörigen `class` geschrieben.

A.3 Präprozessor

Neben den unabdingbaren `#include`-Anweisungen sollte der Präprozessor nicht übermäßig verwendet werden.

Eine sicher sinnvolle Anwendung ist die Verwaltung verschiedener Versionen eines Quelltextes, beispielsweise für verschiedene Systemumgebungen oder verschiedene Kunden mit `#ifdef` beziehungsweise `#if`.

Wenn ein Programmierer von *C* zu *C++* wechselt, ist er sicher versucht, den Präprozessor für viele Dinge einzusetzen, die in *C++* aber besser über Sprachkonstrukte realisiert werden:

- Die Definition von Konstanten (`#define PI 3.141592`) ist wesentlich besser mit `const`-deklarierte Variablen (`const double pi=3.141592;`).
- Funktionsähnliche Makros (`#define MAX(a,b)...`) sind gefährlich, und sollten durch `template`-Funktionen (siehe **4.11 Funktionsschablonen (template-Funktionen)** auf Seite 47) ersetzt werden.

  Um die Laufzeitvorteile der tatsächlich schnellen Makros mit `template`-Funktionen einzuholen, kann man diese als `inline` deklarieren.

A.4 Allgemeiner Stil des Quelltextes

Zu diesem Punkt verbindliche Regeln zu geben ist sicher sinnlos, weil jeder seinen eigenen Stil finden muß.

Aber im Zweifelsfall sollte man seinen Stil (beziehungsweise den der Firma) dann auch möglichst konsequent einsetzen, um ein leichtes Lesen der Quelltexte und damit möglichst wenig Fehler zu ermöglichen.

- Die Kommentierung sollte einheitlich sein. Ein wilder Wechsel zwischen den verschiedenen Kommentierungsarten (`/*`...`*/`, `//`...) fördert nur Verwirrung.

  Viele Leute nehmen für die eigentlichen Kommentare `//`, und nutzen dann `/*`...`*/`, um vorübergehend ganze Bereiche eines Quelltextes stillzulegen. Das halte ich nicht für elegant, weil es erstens nicht leicht zu erkennen ist, was denn nun auskommentiert ist oder nicht, und zweitens das Stillegen nicht geschachtelt werden kann.

  Besser finde ich es, Teile des Quelltextes mit `#if 0` oder ähnlichem stillzulegen, oder noch besser mit `//` in jeder Zeile, falls das mit einem vernünftigen Editor leicht möglich ist (mit dem EMACS ist es sehr leicht möglich):

  ```
  tuwas();
  // stillgelegt! tunix();
  // stillgelegt! for( int i=0; i<10; i++ )
  // stillgelegt!   {
  // stillgelegt!      tunix();
  // stillgelegt!   }
  // stillgelegt! tunix();
  tuwasanderes();
  ```

  Dabei kann neben den eigentlichen Kommentarzeichen auch gleich noch erklärender Text untergebracht werden.

  Geschachteltes Stilllegen ist ebenfalls leicht zu machen:

  ```
  // Schatz, heute nicht.  tuwas();
  // Schatz, heute nicht.  // stillgelegt! tunix();
  // Schatz, heute nicht.  // stillgelegt! for( int i=0; i<10; i++ )
  // Schatz, heute nicht.  // stillgelegt!   {
  // Schatz, heute nicht.  // stillgelegt!      tunix();
  // Schatz, heute nicht.  // stillgelegt!   }
  // Schatz, heute nicht.  // stillgelegt! tunix();
  // Schatz, heute nicht.  tuwasanderes();
  ```

Manche Leute nehmen `//` für „kleine“ Kommentare in einer Zeile, und `/*...*/` für mehrzeilige Kommentare. Ich sehe keinen tieferen Sinn darin, aber wer's mag!

- Die Kommentare ebenso wie Namen von Bezeichnern sollten in einer Landessprache gehalten werden.

  Dafür ist deutsch für meinen Geschmack eigentlich besser geeignet als englisch, weil man leicht zwischen selbstvergebenen Namen einerseits und (englischen) Namen aus Bibliotheken sowie *C++*-Schlüsselwörtern andererseits unterscheiden kann.

  Soll der Quelltext für ein internationales Publikum geeignet sein, ist natürlich Englisch die einzige akzeptable Sprache.

- Die Anweisungen innerhalb von Blöcken sollten gegenüber ihrer Umgebung angemessen (vielleicht 4 Spalten?) eingerückt werden.

  Ich bevorzuge dabei die Version, daß öffnende und schließende Klammern immer übereinander stehen:

```
int main()
{
  for( int i=0; i<10; i++ )
    {
      while( k )
        {
          printf( "Hallo!\n" );
          k--;
        }
    }
}
```

  In letzter Zeit verwenden manche (vor allem Java-belastete) Leute eine andere Art, die Klammern zu positionieren:

```
int main(){
    for( int i=0; i<10; i++ ){
        while( k ){
            printf( "Hallo!\n" );
            k--;
          }
      }
}
```

  Ich finde das sehr unübersichtlich (aber mir muß es ja nicht gefallen).

A.5 Bezeichner lokal halten

Zumindest Variablen und (soweit möglich) Konstanten sollten so lokal wie möglich vereinbart werden. Das bedeutet:

- nichts global vereinbaren, wenn es nicht einen wirklich guten Grund dafür gibt. Wer viele globale Variablen braucht, sollte dringend am Konzept feilen. Eine Alternative zu globalen Variablen könnten `static`-Elemente einer Klasse sein oder statische lokale Variablen einer Funktion beziehungsweise Methode.

- Variablen, die man nur in einem begrenzten Bereich benötigt, sollten auch nicht wesentlich darüberhinaus sichtbar sein.

  Diese Regel führt sowohl zu einer besseren Verständlichkeit des Quelltextes, als auch zu schnellerem Übersetzen und besserem Optimieren durch den Compiler.

  Gegebenenfalls kann man durch zusätzliche Blöcke einen kleineren Namensraum schaffen.

  Auch das wiederholte Verwenden einer Variablen für unterschiedliche Zwecke sollte vermieden werden, beispielsweise ein Schleifenzähler für mehrere nacheinander folgende Schleifen; besser ist es jeder Schleife einen eigenen Zähler zu spendieren. Siehe dazu auch **4.4 Freie Position von Deklarationen** auf Seite 29.

- Analog sollte man Datentypen (`enum`, `class`, `typedef` etc.) soweit möglich innerhalb einer logisch zugehörigen Klasse einbetten, um die Namen nicht global sichtbar zu machen.

  Dies trifft sowohl für „echte“ Datentypen zu als auch für Typen, die nur zum Werfen von Ausnahmen verwendet werden.

  Eine alternative Möglichkeit, die global sichtbaren Namen zu reduzieren, hat man durch **4.7 Namensbereiche (`namespace`)** auf Seite 36.

A.6 Namensgebung

Man sollte aussagekräftige Namen vor dem Hintergrund ihrer Sichtbarkeitsdauer wählen:

- für einen Schleifenzähler, der nur eine Sichtbarkeit von wenigen Quelltextzeilen hat, ist der Name `i` vollkommen in Ordnung (auch wenn `iElement` oder ähnliches deutlicher sein kann).
- Ein Name, der in einem großen Bereich sichtbar ist, muß dagegen wesentlich klarer aussagen, wofür er steht (`aktuelleLeseposition` sagt wohl mehr aus als ein knappes `pos`).
- Wer lange Namen scheut, weil er sie immer wieder von Hand schreiben muß, der hat den falschen Editor.
- Wenn ein Name bereits an einen Namensbereich (oder an eine Klasse) gebunden ist, dann braucht man die Aussage des Namensbereichs nicht noch einmal wiederholen.

  Wenn beispielsweise eine Klasse `Fahrzeug` einen Namen enthält, dann braucht man die Methode zum Setzen oder Abfragen nicht `FahrzeugName()` nennen; durch ein einfaches `Name()` verliert man hier nichts, weil der Name automatisch an ein Fahrzeug gebunden ist:
  ```
  Fahrzeug meinErstesAuto( "Seifenkiste" );
  cout << meinErstesAuto.Name() << endl;
  ```

Es gibt Leute, die Klassennamen immer mit einem `C` beginnen. Das wird schnell lästig.

Ein Vorschlag:

- Makros immer durchgehend groß (als offensichtliche Warnung, daß es sich um Makros handelt)
- Klassennamen immer mit einem Großbuchstaben beginnen, danach gemischt
- Variablennamen mit Kleinbuchstaben beginnen, danach gemischt (wenn mindestens ein Großbuchstabe enthalten ist, muß man keine Kollision mit einem der vielen Schlüsselwörter befürchten, siehe **3.10 Neue Schlüsselworte** auf Seite 24).
- Namen von abgeleiteten Klassen so wählen, daß man dem Namen die Ableitung leicht ansehen kann (`class FzgPKW: public Fahrzeug{...};`)

A.7 Resourcen

TODO

Betrifft: allokierter Speicher, offene Dateien, Semaphoren

da frei geben, wo auch allokiert wird,

Wenn in Konstruktor initialisiert, dann in Destruktor freigeben

Zeiger vermeiden, wenn möglich stattdessen Autopointer verwenden (**5.4 Autopointer** `auto_ptr<T>` auf Seite 144).

A.8 Sprünge vermeiden

TODO

lokale Sprünge: nur in wenigen Ausnahmefällen!

globale Sprünge: kein `setjmp()` und `longjmp()` in *C++* verwenden! Stattdessen immer Ausnahmen verwenden. Dabei die Hinweise in **5.4 Autopointer** `auto_ptr<T>` auf Seite 144 beachten!

Anhang B

Übungen

B.1 Aufgaben

B.1.1 Zuweisung einfacher Objekte

Es ist eine einfache Klasse `Person` zu schreiben, die aus mindestens einem Vor- und einem Nachnamen für eine Person besteht.

Es sind mindestens zwei Methoden zu definieren:

- eine Methode `setze( const char *vn, const char *nn )` setzt den Vor- und den Nachnamen eines Objekts
- `zeige()` gibt den aktuellen Inhalt eines Objekts zur Standardausgabe aus.

Die Klasse ist in drei verschiedenen Versionen zu implementieren, wobei sich die interne Darstellung der Namen (Vor- und Nachname) unterscheidet:

1. in der primitivsten Version sollen die Namen als `char`-Felder mit fester Länge vorliegen; damit wird für kurze Namen Platz verschwendet und Namen ab einer gewissen Länge sind nicht möglich
2. die für den Ersteller der Klasse bequemste, aber auch zur Laufzeit uneffektivste Version soll den in C++ vorhandenen Datentyp `string` verwenden, der automatisch für das Allokieren von ausreichend Speicher sorgt

3. die dritte Version soll selbst für das Allokieren sorgen, indem in der Klasse nur ein Zeiger auf die Namen steht, und bei Bedarf mit `new` oder `malloc()` Platz beschafft wird.

   Hierbei ist aber darauf zu achten, daß nicht mehr benötigter Speicher auch sauber freigegeben wird.

Die Klasse soll mit folgendem Testprogramm ausprobiert werden:

```
// Time-stamp: "08.12.02 13:14 test_person.cpp klaus@wachtler.de"
//
// Testprogramm für class Person aus der C++-Übung A

//#include <cstdlib>
//#include <cstddef>
//#include <cstdio>
//#include <cstring>

#include <iostream>

#include <exception>


#include "Person.h"

using namespace std;

int main( int nargs, char **args )
{
  try
  {
    // zwei Personen vereinbaren und initialisieren:
    cout << "(definieren)" << endl;
    Person a( "Karl", "Moik", 72 ), b( "Otto", "Waalkes", 55 );

    // Beide Personen ausgeben:
    a.zeige(); b.zeige();

    // a zu einer Kopie von b machen:
    cout << "(zuweisen)" << endl;
    a = b;
```

```
    // Beide Personen ausgeben:
    a.zeige(); b.zeige();

    // b neu setzen:
    cout << "(b setzen)" << endl;
    b.setze( "Eddi", "Stotter", 88 );

    // Beide Personen ausgeben:
    a.zeige(); b.zeige();
  }
  catch( exception &Fehler )
  {
    cerr << "Ausnahme in main() gefangen:\n"
         << Fehler.what() << endl;
  }
  catch( ... )
  {
    cerr << "unbekannte Ausnahme in main()" << endl;
  }


  return 0;
} // main( int nargs, char **args )
```

(Der Anwender der Klasse wird also nicht damit belästigt, wie die Implementation intern aussieht.)

Themen:

- Zeiger als Klassenelemente
- Allokieren und Freigeben
- Überladen von Operatoren

Lösung siehe **B.2.1 Zuweisung einfacher Objekte** auf Seite 205.

B.1.2 Generische Funktionen

Im ersten Schritt sind `template`-Funktionen gefordert mit den Namen `min` und `max`, die für 2, 3 oder 4 Parameter beliebigen (aber jeweils gleichen) Typs den Kleinst- oder Größtwert liefern.

Beispiel:
```
min( 5, 2 )
```
soll den Wert `2` liefern,
```
min( 3.14, 7.2, 1.3 )
```
dagegen soll `1.3` liefern.

Im zweiten Schritt soll eine eigene Klasse geschrieben werden, und es soll möglich sein, einzelne Objekte dieser Klasse mit den obigen Funktionen `min()` und `max()` zu verwenden.

Themen:

- `template`-Funktionen-Funktionen
- Spezialisierung von `template`-Funktionen
- Überladen von Operatoren
- eigene Operatoren (`operator`...)

Lösung siehe **B.2.2 Generische Funktionen** auf Seite 213.

B.1.3 `bsearch`, `qsort` selbst bauen

1. Es soll die Funktion `bsearch()` aus der Standardbibliothek nachgebildet werden.

   Beispielsweise soll die Funktion:

   ```
   void msearch( void *suchschluessel,
                 void *feld,
                 size_t anzahl,
                 size_t elementgroesse,
                 int (*vergleichsfunktion)( const void *p1,
                                            const void *p2
                                          )
                 );
   ```

   gebaut werden, die sich wie `bsearch()` verhält.

   Ob die neue Funktion tatsächlich *binäre Suche* verwendet, ist zweitrangig.

2. Analog dazu ist `qsort()` der Standardbibliothek nachzubilden.

   Hierin müssen zum Sortieren Feldelemente umkopiert werden; aber innerhalb der Funktion ist der Datentyp der Elemente unbekannt. Deshalb kann man nicht mit einer Zuweisung (Operator `=`) arbeiten. Als Alternative bietet sich die Funktion `memcpy()` an. Unter Linux oder Unix siehe dazu:

   ```
   man 3 memcpy
   ```

Themen:

- *C*-Standardbibliothek
- Zeiger
- Felder
- Sortieren
- Suchen

Lösung siehe **B.2.3 `bsearch`, `qsort` selbst bauen** auf Seite 226.

B.1.4 Ableitungen mit virtuellen Konstruktoren

B.1.4.1 Basisklasse `Kiste`

Es ist eine Klasse zu entwerfen, die aus mindestens:

- Länge
- Breite
- Höhe
- Material

besteht, und von der weitere Klassen abgeleitet werden können.

Ein Objekt der Klasse soll zu einem `ofstream` ausgegeben werden können.

B.1.4.2 Klassen für Möbel

Von `Kiste` sollen weitere Klassen abgeleitet werden:

- `Schubladenschrank` (enthält eine Anzahl einzusetzender Schubladen)
- `Kleiderschrank` (enthält die Anzahl seiner Türen)

B.1.4.3 `Kistenbaum`

Es soll ein Baum aus `Kiste`-Objekten (oder von `Kiste` abgeleiteten Klassen) gebaut werden können,

Jeder Knoten eines Baums enthält neben einem `Kiste`-Objekt noch:

- ein (variables) Feld von Unterobjekten vom Typ `Kiste` (oder davon abgeleitet)
- die Möglichkeit, sich selbst und seine Unterobjekte mit einer Methode `run()` auszugeben.

B.1.4.4 Planung

Aus `Kistenbaum` soll die folgende Hierarchie aufgebaut werden:

- `Raum` besteht aus einer Kiste, und enthält:

  2 Kleiderschränke

  1 Schubladenschrank

Die ganze Planung ist mit `Raum.run()` auszugeben.

Themen:

- Ableiten
- „virtuelle Konstruktoren" (siehe **4.12.7.5 Virtuelle Konstruktoren** auf Seite 96)
- Überladen von Methoden

Lösung siehe **B.2.4 Ableitungen mit virtuellen Konstruktoren** auf Seite 233.

B.1.5 Felder beliebigen Typs mit konstanten Grenzen

Es ist eine `template`-Klasse `feld` zu schreiben, mit der man ein Feld anlegen kann.

Der Typ der Feldelemente soll beliebig sein (aber zur Übersetzungszeit bekannt), ebenso die Anzahl der Elemente.

Beispielverwendung:

```
class CAuto
{
   // ...
};

int main()
{
   feld<int,10>   meinfeld;
   // jetzt kann man meinfeld[0] ... meinfeld[9] verwenden...

   feld<CAuto,40>  FeldVonAutos
   // jetzt kann man FeldVonAutos[0] ... FeldVonAutos[39] verwenden...


}
```

Themen:

- `template`-Klassen
- Überladen von Operatoren
- Feldzugriff

Lösung siehe **B.2.5 Felder beliebigen Typs mit konstanten Grenzen** auf Seite 242.

B.1.6 Felder beliebigen Typs mit variablen Grenzen

Gefordert ist eine `template`-Klasse, beispielsweise mit dem Namen `VarArr`. Diese Klasse soll die automatische Verwaltung von dynamischen Feldern realisieren.

Die Feldgrenzen sollen zur Laufzeit durch Ansprechen von Elementen des Feldes automatisch erweitert werden.

Beispielverwendung:

```
#include <VarArr.h>
...

VarArr<int>  intArr;

intArr[3] = 33;         // Feld hat jetzt nur ein Element [3]
intArr[2] = 22;         // Feld hat jetzt zwei Elemente ([2], [3])
intArr[-1] = 111;       // Feld hat jetzt 5 Elemente ([-1] bis [3])
intArr[0] = 0;
intArr[1] = 11;
for( int i=-1; i<=3 )
{
   cout << "Element " << i << " ist " << intArr[i] << endl;
}
```

Mit `VarArr<int>  intArr;` wird ein Feld von `int`-Werten mit dem Namen `intArr` vereinbart, ohne die gewünschten Feldgrenzen angeben zu müssen.

Dann kann beliebig auf Elemente des Feldes zugegriffen werden; dabei soll im Hintergrund dafür gesorgt werden, daß die angesprochenen Feldelemente auch existieren.

Des Pudels Kern ist natürlich die geschickte Definition von `operator[]`. Diese Funktion wird bei jedem Feldzugriff (mit den Zeichen `[]`) aufgerufen, und kann bei Bedarf in Ruhe Speicher nachallokieren,umkopieren, oder was sonst nötig ist, bevor sie eine Referenz auf das gewünschte Element liefert.

Zu beachtende Randbedingungen:

1. Die Klasse `VarArr` soll sowohl für eingebaute Datentypen (`int`, `double` etc.) als auch für eigene Datentypen (`class ...`) sinnvoll zu verwenden sein.

2. Um Sortier- und Suchfunktionen wie `qsort()` oder `bsearch()` verwenden zu können, wäre es sinnvoll wenn die Elemente tatsächlich in der Reihenfolge ihrer Indices im Speicher stehen (nicht zwingend).

3. Neben den Mindestanforderungen kann man noch sinnvolle Memberfunktionen definieren, um beispielsweise die aktuellen Feldgrenzen zu setzen oder abzufragen.

   Beispiele:

   - `int l()` liefert die aktuelle Länge des Feldes (0 bei einem neuen Feld, sonst Obergrenze-Untergrenze+1)
   - `int l( int neuelaenge )` setzt die aktuelle Feldlänge: die Obergrenze ist danach der übergebene Wert minus 1, die Untergrenze ist 0.
   - `int l( int untergrenze, int obergrenze )` setzt die Unter- und Obergrenze
   - `int ug()` und `int og()` liefern die Unter- und Obergrenze, `int ug( int untergrenze )` und `int og( int obergrenze )` setzen die Grenzen entsprechend.
   - `SetzeDefault( T &defaultwert )` setzt einen Defaultwert, der in Zukunft für unbenutzte Elemente verwendet wird. Dies kann sinnvoll sein, wenn im Feld unbenutzte Lücken vorkommen:

     ```
     VarArr<char*> strings;
     strings.setDefault( NULL );
     strings[-1] = "abc";
     strings[+1] = "def";
     ```

     Hier werden die Elemente `strings[-1]` und `strings[+1]` explizit gesetzt; `strings[0]` enthält implizit durch den vorhergehenden Aufruf von `SetzeDefault()` den Wert `NULL`.
   - Kopieren eines Feldes in ein (typgleiches) anderes mit `operator=()`
   - Initialisierung eines Feldes mit einem anderen (typgleichen)

Themen:

- `template`-Klassen
- Überladen von Operatoren
- Feldzugriff

Lösung siehe **B.2.6 Felder beliebigen Typs mit variablen Grenzen** auf Seite 245.

B.2 Lösungen

B.2.1 Zuweisung einfacher Objekte

Die Klasse `Person` ist recht einfach, so daß es sich nicht lohnt, dafür eine eigene CPP-Datei zu schreiben; die Methoden werden gleich in der Klasse und damit in der Headerdatei untergebracht.

Um die Unterschiede zwischen den drei Versionen klarer zu machen, sind in einem Quelltext alle geforderten Varianten untergebracht, und werden mit Präprozessordirektiven aktiviert (durch Aktivieren genau einer der drei Zeilen der Art `#define AKTUELLEVERSION`...):

```
// Time-stamp: "08.12.02 14:18 Person.h klaus@wachtler.de"
//
// Deklarationen (und Definitionen) für die class Person aus der
// C++-Übung A
//
// Musterlösung 08.12.2002 Klaus Wachtler

#ifndef __PERSON_H__
#define __PERSON_H__


// Welche Versionen gibt es?
#define FELDVERSION    1
#define STRINGVERSION  2
#define ZEIGERVERSION  3


// Welche Version soll es sein?
//#define AKTUELLEVERSION  FELDVERSION
//#define AKTUELLEVERSION  STRINGVERSION
#define AKTUELLEVERSION  ZEIGERVERSION


#include <iostream>


#if AKTUELLEVERSION==FELDVERSION
#include <cstring>
#include <string>
```

```
#include <stdexcept>
#define LAENGENAMEN    50 // maximale Länge inkl. '\0'
#elif AKTUELLEVERSION==STRINGVERSION
#include <string>
#elif AKTUELLEVERSION==ZEIGERVERSION
#include <cstring>
#include <cstdio>
#else
#error "AKTUELLEVERSION nicht definiert"
#endif // AKTUELLEVERSION...

class Person
{
public:

#if AKTUELLEVERSION==FELDVERSION

  // Konstruktor
  Person( const char *vn="",
          const char *nn="",
          unsigned short nLenze=0u
          )
  {
    // setze() macht eh das Selbe:
    setze( vn, nn, nLenze );
  }

  // (kein Destruktor nötig)

  // setzt den Inhalt komplett neu:
  void setze( const char *vn="",
              const char *nn="",
              unsigned short nLenze=0u
              )
  {
    // Länge der Namen prüfen, ggf. Ausnahme werfen:
    if( strlen( vn )>=LAENGENAMEN
        ||
        strlen( nn )>=LAENGENAMEN
        )
    {
      // Namen zu lang, abbrechen!
      throw std::out_of_range( "Name zu lang in Person::setze()" );
```

```
    }

    // ok, beide Namen können kopiert werden:
    strcpy( vorname, vn );
    strcpy( nachname, nn );

    alter = nLenze;
  }


#elif AKTUELLEVERSION==STRINGVERSION

  // Konstruktor
  Person( const char *vn="",
          const char *nn="",
          unsigned short nLenze=0u
          )
    : vorname( vn ),
      nachname( nn ),
      alter( nLenze )
  {
  }

  // (kein Destruktor nötig)

  // setzt den Inhalt komplett neu:
  void setze( const char *vn="",
              const char *nn="",
              unsigned short nLenze=0u
              )
    {
      vorname = vn;  // string läßt sich mit Zuweisung kopieren
      nachname = nn;
      alter = nLenze;
    }

#elif AKTUELLEVERSION==ZEIGERVERSION

  // Konstruktor
  Person( const char *vn="",
          const char *nn="",
          unsigned short nLenze=0u
          )
```

```
{
  // setze() macht eh das Selbe, allerdings müssen vorname und
  // nachname zu NULL gesetzt werden, weil sonst delete[] mit deren
  // zufälligen bisherigem Inhalt aufgerufen wird, ein delete[]NULL
  // dagegen schadet nichts:
  vorname = NULL;
  nachname = NULL;
  setze( vn, nn, nLenze );
}

// Destruktor
virtual ~Person()
{
  // im ctor oder setze() allokierten Speicher wieder freigeben:
  delete[] vorname;  vorname = NULL;
  delete[] nachname; nachname = NULL;
}

// setzt den Inhalt komplett neu:
void setze( const char *vn="",
            const char *nn="",
            unsigned short nLenze=0u
            )
{
  // Beide bisherigen Namen freigeben:
  delete[] vorname;  vorname = NULL;
  delete[] nachname; nachname = NULL;

  // Mit passender Länge neu allokieren:
  vorname = new char[strlen( vn ) + 1];
  nachname = new char[strlen( nn ) + 1];

  // new klappt immer, beide Namen können kopiert werden:
  strcpy( vorname, vn );
  strcpy( nachname, nn );


  alter = nLenze;
}

// Überschreiben der Funktion operator=() noch nicht
// aktiviert; siehe Beschreibung der Musterlösung
```

```
//  // Zuweisung Person = Person
//  Person & operator=( const Person &rechteSeite )
//  {
//    // Beide bisherigen Namen freigeben:
//    delete[] vorname;  vorname = NULL;
//    delete[] nachname; nachname = NULL;
//
//    // Anhand der rechten Seite der Zuweisung neu allokieren
//    // und kopieren:
//    setze( rechteSeite.vorname,
//           rechteSeite.nachname,
//           rechteSeite.alter
//           );
//
//    return *this;
//  }

#else
#error "AKTUELLEVERSION nicht definiert"
#endif // AKTUELLEVERSION...

  // Ausgabe einer Person:
  void zeige()
  {
    // Ausgabe nach cout klappt mit char[], char* und string:
    std::cout << vorname << " "
              << nachname << ", "
              << alter
              << std::endl;
  }

private:

  // Die Namen:

#if AKTUELLEVERSION==FELDVERSION

  char          vorname[LAENGENAMEN];  // Feste Länge!
  char          nachname[LAENGENAMEN];

#elif AKTUELLEVERSION==STRINGVERSION

  std::string   vorname;  // wird automatisch verwaltet
```

```
  std::string   nachname;

#elif AKTUELLEVERSION==ZEIGERVERSION

  char         *vorname;  // nur Zeiger, rechtzeitig allokieren
  char         *nachname; // und wieder freigeben!

#else
#error "AKTUELLEVERSION nicht definiert"
#endif // AKTUELLEVERSION...

  // Das Alter:
  unsigned short alter;  // nicht mehr als 65535 Jahre möglich


}; // class Person...

#endif // ifdef __PERSON_H__
```

Bei der Verwendung von char-Feldern treten keine Probleme auf (abgesehen von der üblichen Platzverschwendung und der harten maximalen Länge der Namen natürlich).

Nach dem Initialisieren haben die beiden Objekte `a` und `b` die gewünschten Werte, nach der Zuweisung sind sie gleich, und nach dem erneuten Setzen hat `b` den gesetzten Wert:

```
klaus@lap2:~/skript_cpp > g++ -Wall test_person.cpp -o test_person
klaus@lap2:~/skript_cpp > test_person
(definieren)
Karl Moik, 72
Otto Waalkes, 55
(zuweisen)
Otto Waalkes, 55
Otto Waalkes, 55
(b setzen)
Otto Waalkes, 55
Eddi Stotter, 88
```

In der Version mit C++-strings funktioniert die Klasse ebenfalls wie erwartet, und erzeugt exakt dieselbe Ausgabe wie die Version mit Feldern (siehe oben).

Die Version mit Zeigern verhält sich dagegen anders (Beispiel unter Linux getestet, auf anderen Systemen kann es etwas anders aussehen, aber jedenfalls nicht wie bei den obigen Versionen):

```
klaus@lap2:~/skript_cpp > g++ -Wall test_person.cpp -o test_person
klaus@lap2:~/skript_cpp > test_person
(definieren)
Karl Moik, 72
Otto Waalkes, 55
(zuweisen)
Otto Waalkes, 55
Otto Waalkes, 55
(b setzen)
Eddi Stotter, 55
Eddi Stotter, 88
Segmentation fault
```

Daran sind zwei Punkte sehr befremdlich:

1. Nach der Zuweisung sind die Objekte `a` und `b` zwar erwartungsgemäß gleich (`Otto Waalkes, 55`), aber nach dem Setzen von `b` auf einen neuen Wert haben sich Vor- und Nachname in `a` ebenfalls geändert!

2. Am Ende des Programms (Ende von `main()`) stürzt das Programm ab (`Segmentation fault`).

Was ist passiert? Das Problem liegt daran, daß mit der Zuweisung `a=b` in `main()` alles, was innerhalb der Klasse definiert ist (`vorname`, `nachname`, `alter`) exakt von `b` nach `a` kopiert wird. Das bedeutet, daß auch die Zeiger umkopiert werden. Dadurch werden beide Zeiger `a.vorname` und `a.nachname` durch `b.vorname` und `b.nachname` ersetzt. Dagegen werden die eigentlichen Namen (also die Zeichen in den mit `new` allokierten Speicherbereichen) nicht kopiert.

Weil jetzt die Zeiger in `a` auf dieselben Stelle im Speicher zeigen wie die in `b`, wirkt sich eine Änderung der Namen in `b` auch auf `a` aus. (Mit dem Neusetzen der Namen in `b` wird freigegeben und neu allokiert; theoretisch könnte dabei ein Speicher an einer anderen Stelle im Speicher reserviert werden; dann würden die Zeiger in `a` nicht auf denselben Speicher zeigen, sondern auf bereits freigegebenen Speicher. Das scheint hier aber nicht der Fall zu sein, weil die neuen Namen nicht wesentlich länger sind als die alten und deshalb der mit `delete` freigegebene Speicher mit dem `new` gleich wieder reserviert wird).

Dies erklärt die Tatsache, daß sich eine Änderung der Namen in `b` auf die Namen von `a` auswirkt. Dieser Effekt ist sicher nicht erwünscht.

Aber warum stürzt das Programm zum Schluß ab? Dies liegt daran, daß am Ende der Lebensdauer von beiden Variablen jeweils ihr Destruktor aufgerufen wird, zuerst der von `b`. Hierin wird der Speicher für die beiden Namen freigegeben; danach wird der Destruktor von `a` aufgerufen. Weil durch die Zuweisung aber in `a` dieselben Zeiger stehen wie in `b`, wird versucht, den bereits im ersten Destruktoraufruf freigegebenen Speicher nochmals freizugeben. Dies führt zum Absturz. Andererseits kann der vor der Zuweisung für `a` allokierte Speicher nie vom Programm freigegeben werden, weil kein Zeiger mehr darauf existiert.

Nachdem die Probleme durch die vom Compiler unsinnig durchgeführte Zuweisung verursacht werden (auch wenn sie erst später in Erscheinung treten), liegt die Lösung darin, die Zuweisung zu verbessern.

Dazu kann man sich eine Funktion `operator=()` definieren, mit der man definiert, was bei einer Zuweisung genau passieren soll (siehe auch **4.12.9 Überladen von Operatoren** auf Seite 110).

In der Musterlösung `Person.h` ist dies bereits (auskommentiert) enthalten. Wenn man die Kommentarzeichen vor der Funktion `operator()` entfernt, erhält man eine stabile Programmversion, die wie erwartet funktioniert und exakt die selbe Ausgabe ergibt wie die beiden ersten Versionen (mit Feldern beziehungsweise `string`).

Anstatt einfach die Zeiger zu kopieren, wird der Speicher der linken Seite freigegeben, in passender Länge neu allokiert, und dann werden die Zeichen der Namen mit `strcpy()` umkopiert.

B.2.2 Generische Funktionen

Der erste Teil der Aufgabe ist relativ schlicht, solange es um einfache Datentypen geht.

```
// Time-stamp: "(29.11.01 18:32) loes_minmax1.cpp [Klaus Wachtler (aw38)]"
//
// erste Musterlösung für die Aufgabe, min() und max() als
// Templatefunktionen zu implementieren, mit einem kleinen
// Testprogramm für einfache Datentypen (int, double).

#include <cstdio>

using namespace std;


// die geforderten min()- und max()-Funktionen:
template<class T>
const T &min( const T &a, const T &b )
{
  return a<b ? a : b;
}

template<class T>
const T &min( const T &a, const T &b, const T &c )
{
  return min( min( a, b ), c );
}

template<class T>
const T &min( const T &a, const T &b, const T &c, const T &d )
{
  return min( min( a, b ), min( c, d ) );
}

template<class T>
const T &max( const T &a, const T &b )
{
  return a<b ? b : a;
}

template<class T>
```

```
const T &max( const T &a, const T &b, const T &c )
{
  return max( max( a, b ), c );
}

template<class T>
const T &max( const T &a, const T &b, const T &c, const T &d )
{
  return max( max( a, b ), max( c, d ) );
}


// Testprogramm dafür:
void testmin( int a, int b )
{
  printf( "min(%d,%d) = %d\n", a, b, min(a,b) );
}

void testmin( int a, int b, int c )
{
  printf( "min(%d,%d,%d) = %d\n", a, b, c, min(a,b,c) );
}

void testmin( int a, int b, int c, int d )
{
  printf( "min(%d,%d,%d,%d) = %d\n", a, b, c, d, min(a,b,c,d) );
}

void testmax( int a, int b )
{
  printf( "max(%d,%d) = %d\n", a, b, max(a,b) );
}

void testmax( int a, int b, int c )
{
  printf( "max(%d,%d,%d) = %d\n", a, b, c, max(a,b,c) );
}

void testmax( int a, int b, int c, int d )
{
  printf( "max(%d,%d,%d,%d) = %d\n", a, b, c, d, max(a,b,c,d) );
}
```

```
void testmin( double a, double b )
{
  printf( "min(%g,%g) = %g\n", a, b, min(a,b) );
}

void testmin( double a, double b, double c )
{
  printf( "min(%g,%g,%g) = %g\n", a, b, c, min(a,b,c) );
}

void testmin( double a, double b, double c, double d )
{
  printf( "min(%g,%g,%g,%g) = %g\n", a, b, c, d, min(a,b,c,d) );
}

void testmax( double a, double b )
{
  printf( "max(%g,%g) = %g\n", a, b, max(a,b) );
}

void testmax( double a, double b, double c )
{
  printf( "max(%g,%g,%g) = %g\n", a, b, c, max(a,b,c) );
}

void testmax( double a, double b, double c, double d )
{
  printf( "max(%g,%g,%g,%g) = %g\n", a, b, c, d, max(a,b,c,d) );
}

int main( int nargs, char **args )
{
  testmin( 2, 4 );
  testmin( -2, -4 );
  testmin( 20, 14 );
  testmin( 20, 14, 18 );
  testmin( 20, 18, 14 );
  testmin( 18, 20, 14, 5 );

  testmin( 2.1, 4.4 );
  testmin( -2.7, -4.6 );
```

```
  testmin( 20.1, 14.3 );
  testmin( 20.3, 14.9, 18.4 );
  testmin( 20.2, 18.1, 14.8 );
  testmin( 18.1, 20.1, 14.1, 5.0 );

  testmax( 2, 4 );
  testmax( -2, -4 );
  testmax( 20, 14 );
  testmax( 20, 14, 18 );
  testmax( 20, 18, 14 );
  testmax( 18, 14, 20, 5 );

  testmax( 2.1, 4.4 );
  testmax( -2.7, -4.6 );
  testmax( 20.1, 14.3 );
  testmax( 20.3, 14.9, 18.4 );
  testmax( 20.2, 18.1, 14.8 );
  testmax( 18.1, 20.1, 14.1, 5.0 );


  return 0;
} // main( int nargs, char **args )
```

Eine Variante mit variablen Parameterlisten verträgt sich leider nicht mit eigenen Klassenobjekten (und damit auch nicht mit `template`-Funktionen), sodaß man die Funktionen mit jeweils 2, 3, und 4 Parametern ausformulieren muß.

Für alle einfachen numerischen Datentypen ist die Vergleichsfunktion mit `<` definiert, so daß dafür auch eine passende `min()`- und `max()`-Funktion aus dem `template` generiert werden kann (siehe **4.11 Funktionsschablonen (template-Funktionen)** auf Seite 47).

Problematischer wird es, wenn auch eigene Datentypen ins Spiel kommen.

In der bisherigen Form kann man das `template` nicht mehr verwenden, solange Objekte der eigenen Klasse nicht mit `<` verglichen werden können.

Als Beispiel dient eine Klasse, mit der man die wesentlichen Eigenschaften eines Studenten beschreiben kann. Zum Vergleich auf kleiner/größer soll die Intelligenz dienen.

Als Ausweg aus dem Dilemma, keinen Vergleich mit `<` durchführen zu können, könnte man auf die Idee kommen, eine kleine Hilfsfunktion einzuführen:

```
// Die Vergleichsfunktionen:
bool kleiner( const Stud &a, const Stud &b )
{
  return a.getIQ()<b.getIQ();
}
```

In der `template`-Funktion muß man dann entsprechend diese Funktion aufrufen anstatt des Operators `<`:

```
// Time-stamp: "(29.11.01 19:00) loes_minmax2.cpp [Klaus Wachtler (aw38)]"
//
// Musterlösung für die Aufgabe, min() und max() als
// Templatefunktionen zu implementieren, mit einem kleinen
// Testprogramm für eigene Datentypen.
// Um die eigenen Datentypen verwenden zu können, wird eine
// Vergleichsfunktion bool kleiner( Stud a, Stud b ) eingebaut,
// die den Vergleich liefert.

#include <iostream>
#include <string>

using namespace std;


// Die Klasse Stud:
class Stud
{
private:
  string name;
  int IQ;
  int semesteranzahl;

public:
  Stud( string name="?", int IQ = 50, int semesteranzahl = 20 )
  {
    this->name           = name;
    this->IQ             = IQ;
    this->semesteranzahl = semesteranzahl;
```

```
  }

  string getName() const
  {
    return name;
  }

  int getIQ() const
  {
    return IQ;
  }

  int getSemester() const
  {
    return semesteranzahl;
  }

};

// Die Vergleichsfunktionen:
bool kleiner( const Stud &a, const Stud &b )
{
  return a.getIQ()<b.getIQ();
}

// die geforderten min()- und max()-Funktionen:
template<class T>
const T &min( const T &a, const T &b )
{
  return kleiner(a,b) ? a : b;
}

// Versionen mit 3 und 4 Parametern wie gehabt...


template<class T>
const T &max( const T &a, const T &b )
{
  return kleiner(a,b) ? b : a;
}

// Versionen mit 3 und 4 Parametern wie gehabt...
```

```
int main( int nargs, char **args )
{
  Stud  schlau( "Hinz", 95, 3 ), doofi( "Kunz", 40, 38 );

  Stud derschlauere, derdoofe;

  derschlauere = max( schlau, doofi );
  derdoofe = min( schlau, doofi );

  cout << derschlauere.getName() << " ist schlau." << endl;
  cout << derdoofe.getName() << " nicht ganz so." << endl;

  return 0;
} // main( int nargs, char **args )
```

Diese Variante kommt gut mit den Objekten der Klasse `Stud` klar; leider nicht mehr mit einfachen Datentypen wie `int`, solange man nicht für jeden benutzten Datentyp eine passend überladene Funktion `kleiner()` zur Verfügung stellt. Das hebt leider den Nutzen einer generischen `min()`- oder `max()`-Funktion auf, wenn man dann für jeden Typ die Vergleichsfunktion liefern muß.

Eine andere Variante wäre, die ursprüngliche `min()`- und `max()`-Funktion für einfache Datentypen zu wählen, und von der `template`-Funktion für schwierigere Datentypen jeweils eine Spezialisierung zu schreiben:

```
// Time-stamp: "(29.11.01 19:17) loes_minmax3.cpp [Klaus Wachtler (aw38)]"
//
// Musterlösung für die Aufgabe, min() und max() als
// Templatefunktionen zu implementieren, mit einem kleinen
// Testprogramm für eigene Datentypen.
// Um die eigenen Datentypen verwenden zu können, wird für
// die template-Funktionen eine Spezialisierung für Stud
// geschrieben.

#include <iostream>
#include <string>

using namespace std;


// Die Klasse Stud:
```

```
class Stud
{
private:
  string name;
  int IQ;
  int semesteranzahl;

public:
  Stud( string name="?", int IQ = 50, int semesteranzahl = 20 )
  {
    this->name           = name;
    this->IQ             = IQ;
    this->semesteranzahl = semesteranzahl;
  }

  string getName() const
  {
    return name;
  }

  int getIQ() const
  {
    return IQ;
  }

  int getSemester() const
  {
    return semesteranzahl;
  }

};

// die geforderten min()- und max()-Funktionen:

// für die meisten Typen:
template<class T>
const T &min( const T &a, const T &b )
{
  return a<b ? a : b;
}

// Spezialisierung für Stud:
template<>
```

```
const Stud &min<Stud>( const Stud &a, const Stud &b )
{
  return a.getIQ()<b.getIQ() ? a : b;
}


// für alle Typen:
template<class T>
const T &min( const T &a, const T &b, const T &c )
{
  return min( min( a, b ), c );
}

template<class T>
const T &min( const T &a, const T &b, const T &c, const T &d )
{
  return min( min( a, b ), min( c, d ) );
}

// für die meisten Typen:
template<class T>
const T &max( const T &a, const T &b )
{
  return a<b ? b : a;
}

// Spezialisierung für Stud:
template<>
const Stud &max<Stud>( const Stud &a, const Stud &b )
{
  return a.getIQ()<b.getIQ() ? b : a;
}


// für alle Typen:
template<class T>
const T &max( const T &a, const T &b, const T &c )
{
  return max( max( a, b ), c );
}

template<class T>
const T &max( const T &a, const T &b, const T &c, const T &d )
```

```
{
  return max( max( a, b ), max( c, d ) );
}


int main( int nargs, char **args )
{
  Stud  schlau( "Hinz", 95, 3 ), doofi( "Kunz", 40, 38 );

  Stud derschlauere, derdoofe;

  derschlauere = max( schlau, doofi );
  derdoofe = min( schlau, doofi );

  cout << derschlauere.getName() << " ist schlau." << endl;
  cout << derdoofe.getName() << " nicht ganz so." << endl;

  return 0;
} // main( int nargs, char **args )
```

Jetzt kann man mit `min()` und `max()` sowohl die eingebauten Datentypen, als auch Elemente von `Stud` verwenden.

Am elegantesten ist allerdings die Lösung, bei der für die Klasse `Stud` definiert wird, was als *kleiner* anzusehen ist. Dazu überlädt man einfach den Operator `<` (siehe dazu auch **4.12.9 Überladen von Operatoren** auf Seite 110), indem man in der Klasse eine Funktion `operator<()` definiert.

Durch diese Vorgehensweise kann man `min()` und `max()` (ebenso wie alle anderen `template`-Funktionen, die den Operator `<` voraussetzen) ohne Spezialisierung aufrufen:

```
// Time-stamp: "(29.11.01 19:26) loes_minmax4.cpp [Klaus Wachtler (aw38)]"
//
// Musterlösung für die Aufgabe, min() und max() als
// Templatefunktionen zu implementieren, mit einem kleinen
// Testprogramm für eigene Datentypen.
// Um die eigenen Datentypen verwenden zu können, wird für
// Stud der Operator < überschrieben.

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

```
using namespace std;


// Die Klasse Stud:
class Stud
{
private:
  string name;
  int IQ;
  int semesteranzahl;

public:
  Stud( string name="?", int IQ = 50, int semesteranzahl = 20 )
  {
    this->name           = name;
    this->IQ             = IQ;
    this->semesteranzahl = semesteranzahl;
  }

  string getName() const
  {
    return name;
  }

  int getIQ() const
  {
    return IQ;
  }

  int getSemester() const
  {
    return semesteranzahl;
  }

  bool operator<( const Stud &rechteSeite ) const
  {
    return this->IQ<rechteSeite.IQ;
  }

};

// die geforderten min()- und max()-Funktionen:
```

```
// für alle Typen, die den Operator < kennen:
template<class T>
const T &min( const T &a, const T &b )
{
  return a<b ? a : b;
}

// für alle Typen:
template<class T>
const T &min( const T &a, const T &b, const T &c )
{
  return min( min( a, b ), c );
}

template<class T>
const T &min( const T &a, const T &b, const T &c, const T &d )
{
  return min( min( a, b ), min( c, d ) );
}

// für alle Typen, die den Operator < kennen:
template<class T>
const T &max( const T &a, const T &b )
{
  return a<b ? b : a;
}

// für alle Typen:
template<class T>
const T &max( const T &a, const T &b, const T &c )
{
  return max( max( a, b ), c );
}

template<class T>
const T &max( const T &a, const T &b, const T &c, const T &d )
{
  return max( max( a, b ), max( c, d ) );
}


int main( int nargs, char **args )
```

```
{
  Stud  schlau( "Hinz", 95, 3 ), doofi( "Kunz", 40, 38 );

  Stud derschlauere, derdoofe;

  derschlauere = max( schlau, doofi );
  derdoofe = min( schlau, doofi );

  cout << derschlauere.getName() << " ist schlau." << endl;
  cout << derdoofe.getName() << " nicht ganz so." << endl;

  return 0;
} // main( int nargs, char **args )
```

B.2.3 bsearch, qsort selbst bauen

Um nicht mit den Namen der Standardfunktionen zu kollidieren, werden in der folgenden Musterlösung andere Funktionsnamen verwendet, nämlich `msearch()` und `msort()`.

```
// Time-stamp: "(31.10.01 19:28) msearchmsort.cpp [Klaus Wachtler (aw38)]"
//
// Lösung der Beispielaufgabe "Neuprogrammieren von bsearch() und qsort()".
//
// Diese Version läuft nur unter dem gcc, weil zum Allokieren von
// Speicherplatz die Funktion alloca() verwendet wird. Dies ist kein
// ANSI-Standard!


#include <stdlib.h>
#include <stdio.h>

// sucht in einem sortierten Feld (*base)[] nach einem
// Element, für das die Funktion (*compar)() Gleichheit mit dem
// Schlüssel (*key) verspricht, und liefert einen Zeiger auf
// das gefundene Element, oder NULL
// (entspricht dem Original-bsearch() nach ANSI-C)
void *msearch( const void *key,
               const void *base,
               size_t nmemb,
               size_t size,
               int (*compar)( const void *, const void * )
               )
{
  switch( nmemb )
    {
    case 0:
      return NULL;

    case 1:
      return ( (*compar)( base, key )==0 ? (void *)base : NULL );

    default :

      // Teilen, und suchen lassen:
      {
```

```
        size_t n1 = nmemb/2;
        const void  *p1 = (const char*)base+(n1*size);
        int    v1 = (*compar)( p1, key );
        if( v1==0 )
          {
            // Schlüssel genau getroffen
            return(void *) p1;
          }
        else if( v1<0 )
          {
            // base[n1] kleiner als Schlüssel;
            // also in der oberen Hälfte suchen.
            // base[n1] braucht schon nicht mehr betrachtet
            // zu werden.
            return msearch( key,
                            (const char*)p1+size,
                            nmemb - n1 - 1,
                            size,
                            compar
                            );
          }
        else
          {
            // base[n1] größer als Schlüssel;
            // also in der unteren Hälfte suchen.
            // base[n1] braucht schon nicht mehr betrachtet
            // zu werden.
            return msearch( key,
                            base,
                            n1,
                            size,
                            compar
                            );
          }
      }

    } // switch

} // msearch()



// Aus c't 3/92, S. 263ff. entnommen:
```

```
// {Bezugsdatenstruktur a : ARRAY [1 .. n] und temp}
//
// PROCEDURE quicksort ( links, rechts : integer);
// VAR k, i, j : integer;
// BEGIN
//    IF rechts > links THEN BEGIN
//       i := links -1; j := rechts; k := a[rechts].key;
//       REPEAT
//          REPEAT i := i +1 UNTIL a[i].key > k;
//          REPEAT j := j -1 UNTIL a[j].key <= k;
//          IF i >= j THEN
//          BEGIN
//             temp := a[i]; a[i] := a[r]; a[r] := temp;
//          END
//          ELSE
//          BEGIN
//             temp := a[i]; a[i] := a[j]; a[j] := temp;
//          END;
//       UNTIL i >= j;
//    quicksort( links, i-1 );
//    quicksort( i+1, rechts);
//   END;
// END;


void msort( const void *base,
            size_t nmemb,
            size_t size,
            int (*compar)( const void *, const void * )
            )
{
  // VAR k, i, j : integer;
  int i, j;
  void *key_p = alloca( size );

  // IF rechts > links THEN BEGIN
  if( nmemb>1 )
    { // Feldlänge größer als 1

      // i := links -1; j := rechts; k := a[rechts].key;
      i = -1; j = nmemb - 1;
      memcpy( key_p, (char*)base + size*j, size );
```

```
// REPEAT
do
  {

    // REPEAT i := i +1 UNTIL a[i].key > k;
    while( compar( (char*)base + size*++i, key_p )>0 );

    // REPEAT j := j -1 UNTIL a[j].key <= k;
    while( compar( (char*)base + size*--j, key_p )<=0 );

    // IF i >= j THEN
    if( i>=j )
      {
        // temp := a[i]; a[i] := a[r]; a[r] := temp;
        // ?
        // 31.10.2001 Klaus Wachtler:
        // Was hier vertauscht werden soll, weiß ich nicht.
        // Die Variable r ist bisher nicht erwähnt; ich nehme
        // an, daß nichts vertauscht werden muß.
        // Mein Beispielfeld jedenfalls wird richtig
        // sortiert.
        // Was wirklich gemeint ist, muß ich noch klären.
      }
    // ELSE
    else
      {
        // temp := a[i]; a[i] := a[j]; a[j] := temp;
        void *temp_p = alloca( size );
        memcpy( temp_p, (char*)base + size*i, size );
        memcpy( (char*)base + size*i, (char*)base + size*j, size );
        memcpy( (char*)base + size*j, temp_p, size );
      }


  }
while( i<j ); // UNTIL i >= j;

// quicksort( links, i-1 );
msort( base, i-1, size, compar );

// quicksort( i+1, rechts);
msort( (char*)base + size*(i+1), nmemb - i - 1, size, compar );
```

```
    } // Feldlänge größer als 1


} // msort( const void *base,...


typedef struct
{
  double d;
  char   s[100];
} s_t;


// vergleicht zwei Elemente vom Typ s_t (auf welche
// die beiden übergebenen Zeiger zeigen) anhand ihrer
// darin enthaltenen Gleitkommawerte d:
int vergl_f( const void *p1, const void *p2 )
{
  if( ((const s_t*)p1)->d > ((const s_t*)p2)->d )
    {
      return 1;
    }
  else if( ((const s_t*)p1)->d < ((const s_t*)p2)->d )
    {
      return -1;
    }
  else
    {
      return 0;
    }
}


// zu sortierendes Beispielfeld:
s_t f[] =
{
  { 3.141592654, "pi" },
  { 2.718281828, "e" },
  { 53.00000000, "w" },
  { 1.000000000, "eins" },
  { 2.000000000, "zwoo" },
  { 13.00000000, "drölf" },
};
```

```
// Anzahl der Feldelemente:
const int l = sizeof(f)/sizeof(f[0]);


// Testprogramm:
int main( int nargs, char **args )
{
  qsort( f,
         l,
         sizeof(f[0]),
         vergl_f
         );

  //for( int i=0; i<l; i++ )
  //  {
  //    printf( "f[%2d]=%g\n", i, f[i].d );
  //  }


  while( 1 )
    {
      double wert;
      printf( "Bitte einen Wert: " );
      if( scanf( "%lf", &wert )!=1 )
        {
          break;
        }

      s_t key = { wert, "" };
      s_t *p = (s_t *)msearch( &key,
                               f,
                               l,
                               sizeof(f[0]),
                               vergl_f
                               );

      if( p )
        {
          printf( "Gefunden: %f %s\n", p->d, p->s );
        }
      else
```

```
        {
          printf( "nichts gefunden!\n" );
        }

    }

  return 0;

} /* main( int nargs, char **args ) */
```

B.2.4 Ableitungen mit virtuellen Konstruktoren

Im folgenden eine Musterlösung zu den Teilaufgaben aus **B.1.4 Ableitungen mit virtuellen Konstruktoren** auf Seite 200.

```
// Time-stamp: "(05.12.01 14:45) kiste.cpp [Klaus Wachtler]"
//

#include <iostream>
#include <exception>
#include <vector>
#include <string>

using namespace std;


//
//  *   *  **          *
//  *  *               *
//  * *         ***    *     ***
//  **     *   *      ***   *   *
//  * *    *    ***    *    ****
//  *  *   *       *   *    *
//  *   *   *   ***     *    ***
//
//

class CKiste
{
public:

  CKiste( string Name = "?",
          double b = 0.0,
          double h = 0.0,
          double t = 0.0,
          string Material = "Luft"
          )
  {
    this->Name     = Name;
    this->b        = b;
    this->h        = h;
    this->t        = t;
```

```
    this->Material = Material;
  }

  CKiste( const CKiste &rechteSeite )
  {
    this->Name     = rechteSeite.Name;
    this->b        = rechteSeite.b;
    this->h        = rechteSeite.h;
    this->t        = rechteSeite.t;
    this->Material = rechteSeite.Material;
  }

  virtual CKiste *clone() const
  {
    return new CKiste( *this );
  }

  virtual ~CKiste()
  {
  }

  CKiste & operator=( const CKiste &rechteSeite )
  {
    this->Name     = rechteSeite.Name;
    this->b        = rechteSeite.b;
    this->h        = rechteSeite.h;
    this->t        = rechteSeite.t;
    this->Material = rechteSeite.Material;

    return *this;
  }

  virtual ostream &zeigdich( ostream &s ) const
  {
    s << Name
      << ", " << b
      << ", " << h
      << ", " << t
      << ", " << Material;

    return s;
  }
```

```
  // gibt alle Teile aus:
  friend ostream &operator<<( ostream &s, const CKiste &k );

protected:

  string   Name;
  double   b, h, t; // Breite, Höhe, Tiefe
  string   Material;

private:


}; // class CKiste

ostream &operator<<( ostream &s, const CKiste &k )
{
  return k.zeigdich( s );
}


//
//   ***         *            *                       *
//  *   *        *            *                       *
//  *            *            *           ***         *
//   ***    ***  ***   *   *  ****       *       ***  ***
//      *  *     *  *  *   *  *   *       ***   *     *  *
//  *   *  *     *  *  *   *  *   *  **      *  *     *  *  **
//   ***    ***  *  *   ****  ****   **   ***    ***  *  *  **
//
//

class CSchubladenschrank
  : public CKiste
{
public:

  CSchubladenschrank( string Name = "?",
                      double b = 0.0,
                      double h = 0.0,
                      double t = 0.0,
                      string Material = "Luft",
                      int ns = 0
                      )
```

```
    : CKiste( Name, b, h, t, Material )
  {
    this->AnzahlSchubladen = ns;
  }

  CSchubladenschrank( const CSchubladenschrank &rechteSeite )
    : CKiste( (CKiste)rechteSeite )
  {
    this->AnzahlSchubladen = rechteSeite.AnzahlSchubladen;
  }

  virtual CSchubladenschrank *clone() const
  {
    return new CSchubladenschrank( *this );
  }

  virtual ~CSchubladenschrank()
  {
  }

  CSchubladenschrank & operator=( const CSchubladenschrank &rechteSeite )
  {
    *(CKiste*)this = (CKiste)rechteSeite;
    this->AnzahlSchubladen = rechteSeite.AnzahlSchubladen;

    return *this;
  }


  virtual ostream &zeigdich( ostream &s ) const
  {
    ( CKiste(*this) ).zeigdich( s );
    s << ", Schubladen " << AnzahlSchubladen;
    return s;
  }

  // gibt alle Teile aus:
  friend ostream &operator<<( ostream &s, const CSchubladenschrank &k );

protected:

  int AnzahlSchubladen;
```

```
private:


}; // class CSchubladenschrank

// ostream &operator<<( ostream &s, const CSchubladenschrank &k )
// {
//   return k.zeigdich( s );
// }


//
//  *   *  *          **      *                             *
//  *  *   *                  *                             *
//  * *    *    ***           *   ***           ***         *
//  **     *   *   *  *    ****  *   *  ****   *       ***  ***
//  * *    *   ****   *   *   *  ****   *   *   ***   *     *  *
//  *  *   *   *      *   *   *  *      *          *  *     *  *  **
//  *   *   *   ***    *   ***    ***   *       ***    ***  *  *  **
//
//

class CKleiderschrank
  : public CKiste
{
public:

  CKleiderschrank( string Name = "?",
                      double b = 0.0,
                      double h = 0.0,
                      double t = 0.0,
                      string Material = "Luft",
                      int nt = 0
                      )
    : CKiste( Name, b, h, t, Material )
  {
    this->AnzahlTueren = nt;
  }

  CKleiderschrank( const CKleiderschrank &rechteSeite )
    : CKiste( (CKiste)rechteSeite )
  {
    this->AnzahlTueren = rechteSeite.AnzahlTueren;
```

```
  }

  virtual CKleiderschrank *clone() const
  {
    return new CKleiderschrank( *this );
  }

  virtual ~CKleiderschrank()
  {
  }

  CKleiderschrank & operator=( const CKleiderschrank &rechteSeite )
  {
    *(CKiste*)this = (CKiste)rechteSeite;
    this->AnzahlTueren = rechteSeite.AnzahlTueren;

    return *this;
  }

  virtual ostream &zeigdich( ostream &s ) const
  {
    ( (CKiste)*this ).zeigdich( s );
    s << ", Türen " << AnzahlTueren;
    return s;
  }

  // gibt alle Teile aus:
  friend ostream &operator<<( ostream &s, const CKleiderschrank &k );

protected:

  int AnzahlTueren;

private:


}; // class CKleiderschrank

// ostream &operator<<( ostream &s, const CKleiderschrank &k )
// {
//   return k.zeigdich( s );
// }
```

```
//
//  ****
//  *   *
//  *   *   ***
//  ****       *  *   *  *** **
//  *   *   ****  *   *  *  *  *
//  *   *  *   *  *   *  *  *  *
//  ****    ****   ****  *  *  *
//
//

class CKistenbaum
{
public:

  CKistenbaum()
  {
    pKiste = NULL;
  }

  void setzeKiste( const CKiste &k )
  {
    delete pKiste;
    pKiste = k.clone();
  }

  void setzeUO( const CKistenbaum &uo )
  {
    Unterobjekte.push_back( new CKistenbaum() );
    Unterobjekte[Unterobjekte.size()-1]->setzeKiste( (*uo.pKiste) );
  }

  virtual ~CKistenbaum()
  {
    delete pKiste;
  }

  void run()
  {
    if( pKiste )
      {
```

```
        cout << *pKiste << endl;
      }
    for( size_t i=0; i<Unterobjekte.size(); i++ )
      {
        Unterobjekte[i]->run();
      }
  }

  friend ostream &operator<<( ostream &s, CKistenbaum &k );


protected:

  CKiste  *pKiste;

  vector<CKistenbaum*>   Unterobjekte;

private:


}; // class CKistenbaum


ostream &operator<<( ostream &s, CKistenbaum &k )
{
  if( k.pKiste )
    {
      s << *(k.pKiste);
    }
  return s;
}

int main( int nargs, char **args )
{
  // Wurzel des Baums:
  CKistenbaum wurzel;
  wurzel.setzeKiste( CKiste( "Wurzel", 500, 360, 400, "Luft" ) );


  // Unterobjekte einfügen:

  CKistenbaum tmp;
```

```
  tmp.setzeKiste( (CSchubladenschrank( "Holzschrank",
                                       80,
                                       85,
                                       60,
                                       "Holz",
                                       5
                                       )
                  )
                );
  wurzel.setzeUO( tmp );

  tmp.setzeKiste( (CKleiderschrank( "dretüriger Kleiderschrank",
                                    180,
                                    200,
                                    60,
                                    "Span",
                                    3
                                    )
                  )
                );
  wurzel.setzeUO( tmp );

  tmp.setzeKiste( (CSchubladenschrank( "sieben Schubladen",
                                       80,
                                       85,
                                       40,
                                       "Blech",
                                       7
                                       )
                  )
                );
  wurzel.setzeUO( tmp );

  // alles ausgeben:
  wurzel.run();


  return 0;
} // main( int nargs, char **args )
```

B.2.5 Felder beliebigen Typs mit konstanten Grenzen

Im folgenden eine Musterlösung zur Aufgabe, Felder beliebigen Typs zu vereinbaren, deren Elementanzahl zur Übersetzungszeit frei ist, aber zur Laufzeit dann konstant.

Die Lösung basiert auf einer `template`-Klasse mit einem Typparameter und der Anzahl der Elemente.

```
// Time-stamp: "(12.12.01 21:25) ConstArr.cpp [Klaus Wachtler (aw38)]"
//
//

#include <iostream>
#include <cstdio>

using namespace std;

// Wenn FELDGRENZENPRUEFEN definiert ist, dann wird bei jedem
// Zugriff der Wert des Index geprüft:
#define FELDGRENZENPRUEFEN

template<class T, int N> class Feld
{
public:

  // Konstruktor
  Feld()
  {
  }

  // Konstruktor mit einem Wert, der anfangs an jedes
  // Element zugewisen werden soll:
  Feld( const T &Anfangswert )
  {
    for( int i=0; i<N; i++ )
      {
        array[i] = Anfangswert;
      }
  }

  // copy-Konstruktor
  Feld( const Feld &rechteSeite )
```

```
  {
    for( int i=0; i<N; i++ )
      {
        array[i] = rechteSeite.array[i];
      }
  }

  // Die Verwendung von [index] liefert eine
  // Referenz auf das Element in array[index]:
  T &operator[]( int index )
  {

#ifdef FELDGRENZENPRUEFEN

    if( index<0 || index>=N )
      {
        static char puffer[200];
        sprintf( puffer,
                 "Feldgrenzenüberschreitung in Feld<class T,int N>:\n"
                 "%d liegt nicht im Bereich %d...%d-1",
                 index,
                 0,
                 N
                 );
        throw( (const char*)puffer );
      }

#endif // ifdef FELDGRENZENPRUEFEN

    return array[index];
  }

private:

  // das eigentliche Feld mit den Elementen
  T array[N];


}; // class Feld...


// Testprogramm für Feld<int,10>:
```

```
int main( int nargs, char **args )
{
  try
    {
      Feld<int,10>  feld(-99);

      // Elemente belegen:
      for( int i=0; i<5; i++ )
        {
          feld[i] = i*i;
        }

      // Elemente ausgeben:
      for( int i=0; i<10; i++ )
        {
          cout << "feld[" << i << "] = " << feld[i] << endl;
        }

      // "versehentlicher" Zugriff neben das Feld:
      cout << "feld[" << 10 << "] = " << feld[10] << endl;
    }
  catch( const char *Fehler )
    {
      cerr << "Fehler in main():\n" << Fehler << endl;
    }
  catch( ... )
    {
      cerr << "Unbekannter Fehler in main()\n";
    }
  return 0;
} // main( int nargs, char **args )
```

B.2.6 Felder beliebigen Typs mit variablen Grenzen

Die Funktionsweise der Musterlösung sollte aus den Kommentaren hervorgehen; im Anschluß findet sich ein kleines Testprogramm.

```
//=============================================-*-c++-*-==============
//
// Time-stamp: "20.06.03 14:15 vararr.h klaus@wachtler.de"
//
//====================================================================
//
// Die Klasse stellt eine template-Klasse für variable Felder
// zur Verfügung.
//
//==================================================================//
//                                                                  //
// Copyright:                                                       //
// ==========                                                       //
//                                                                  //
//           ///| |||\    || ||||    //                             //
//          ////| |||\\   || ||||   //                              //
//         ////|| ||||\\  || ||||  //                               //
//        //// || |||| \\ || |||| //                                //
//       ////==|| ||||  \\|| ||||//                                 //
//      ////   || ||||   \\| |||//                                  //
//     ////    || ||||    \| ||//                                   //
//                           |//                                    //
//         ||||    /|    //  //  ///| ||||=====//  ////=======      //
//         ||||   /||   //  //  ////| ||||    //  ////              //
//         ||||  //||  //  //  ////|| ||||   //  ////               //
//         |||| // || //  //  //// || ||||  //  ////                //
//         ||||//  ||//  //  ////==|| ||||=((  ((((=======          //
//         |||//   |//  //  ////   || ||||  \\  \\\\                //
//         |||/    |/  //  ////    || ||||   \\  \\\\=======        //
//                                                                  //
// (C) 1997-2003 AnyWare                                            //
//                                                                  //
// Klaus Wachtler                                                   //
// Breidingstr. 17                                                  //
// 29614 Soltau                                                     //
//                                                                  //
// Tel. (49)-5191-70 2 71                                           //
```

```
// eMail AnyWare@Wachtler.de                                              //
//                                                                        //
//========================================================================//
//
// Geschichte:
// ===========
//
// Datum:   Name: Vers. Änderung:
//
// 16.05.1997 kw    0.1   erster Entwurf aus [1] intarr.cc entnommen
//
// 12.06.1997 kw          Fehlerbehandlung auf CError umgestellt
//
// 03.07.1997 kw    0.1b  freigegeben
//
// 15.07.1997 kw    0.1c  freigegeben
//
// 17.07.1997 kw          Fehlerkorrektur:
//                        Die Übergabe eines VarArr-Feldes an eine
//                        Funktion erzeugte ein neues (temporäres)
//                        Objekt, dabei wird ein default-Konstruktor
//                        (statt:
//                          VarArr<T>( const VarArr<T> & init )
//                        ) aufgerufen.
//                        Dieser allokiert das Feld (p) nicht neu,
//                        sondern kopiert nur den Zeiger.
//                        beim Verlassen der Funktion wird der
//                        Destruktor aufgerufen, und das allokierte
//                        Feld (p) freigegeben (obwohl das zugehörige
//                        ursprüngliche Objekt noch existiert!)
//                        Beim Freigeben des ursprünglichen Objekts
//                        wird der zugehörige Destruktor nochmals
//                        (p) freigeben.
//                        Lösung:
//                          VarArr<T>( const VarArr<T> & init )
//                        korrekt definiert.
//
//                        Bei dieser Gelegenheit auch gleich:
//                          VarArr<T> & operator=( const VarArr<T> )
//                        definiert.
//
// 18.07.1997 kw          - Erweiterung auf negative Indizes
//                        - Allokieren grösserer Blöcke statt
```

```
//                              Verlängerung auf gerade benötigte Länge.
//
// 19.07.1997 kw              Fehlerbehandlung mit CError wieder entfernt:
//                            ohne feste untere Grenze gibt es auch keine
//                            Benutzerfehler mehr (Setzen von l(), ug(),
//                            und og() auf unsinnige Werte erklärt das Feld
//                            einfach für freigegeben).
//
// 22.07.1997 kw              Speicherfehler geknackt (Bei Adreßberechnung
//                            von p0_neu, wenn Elementgröße kein Vielfaches
//                            von 4 ist)
//
// 23.07.1997 kw      0.1d    freigegeben
//
// 24.07.1997 kw              - Kommentare korrigiert
//                            - copy-Konstruktor: Kopieren mit "=" ersetzt
//                              durch entsprechenden Teil aus operator=(),
//                              da sonst eine Endlosschleife beim Zuweisen
//                              entsteht.
//
// 28.01.1998 kw      0.2     einige vorhandene Funktionen als "... const"
//                            definiert, sowie die Funktion:
//                              T & operator[]( int index ) const
//                            neu definiert.
//                            Damit kann man mit einem Feldzugriff (mit [])
//                            auch in Objekte fassen, die als const ver-
//                            einbart sind. Bei einer Feldgrenzenüber-
//                            schreitung wird dann nicht das Feld
//                            vergrößert, sondern eine Ausnahme ausgeworfen
//                            (CErrorVarArrFeldgrenzenConst).
//                            Damit ist jetzt auch ein #include "cerror.h"
//                            nötig.
//
// 30.01.1998 kw              Feldgrenzenüberprüfung korrigiert;
//                            die Überprüfung kann mit
//                               #define VARARR_FELDGRENZENKONTROLLIEREN
//                            eingeschaltet werden.
//
// 08.05.1998 kw              Bei DEBUG ist etwas mehr public
//
// 23.05.1998 kw      0.3     Neuerung:
//                            Wenn VOR dem #includen von vararr.h
//                            ein VARARR_NULL_POINTER #definiert
```

```
//                          wird, dann wird beim Verlängern eines
//                          Feldes kontrolliert, ob der Name des
//                          zugrundeliegenden Typs der Feldelemente
//                          auf "*" endet.
//                          Wenn ja, scheint es ein Zeigertyp zu sein,
//                          und der gesamte neu allokierte Speicher wird
//                          mit Nullen vorbelegt.
//
// 01.03.1999 kw            Einbau von Kleinst- und Größtwerten,
//                          bei deren Überschreitung eine assert()-
//                          Verletzung generiert wird (MINASSERT,
//                          MAXASSERT)
//
// 14.03.1999 kw            So eingestellt, daß im Normalfall keine
//                          weiteren Dateien nötig sind
//                          (also VARARR_NULL_POINTER und
//                          VARARR_FELDGRENZENKONTROLLIEREN aus)
// 20.10.1999 kw
// 27.05.2000 kw            Anpassungen an Linux/gcc
//                          Fehler mit Defaultwerten eliminiert;
//                          in dieser Form jetzt für Formelinterpreter
//                          unter Linux geeignet
//
// 08.09.2002 kw            Mechanismus mit MINASSERT/MAXASSERT entfernt
//                          (war nur zum Debuggen interessant),
//                          sowie die Nullpointer (mit
//                          VARARR_NULL_POINTER, nicht portabel)
//
// 09.09.2002 kw            in operator[] einen Schnelltest eingefügt,
//                          ob resize() überhaupt aufgerufen werden muß
//
//=====================================================================
//
// Prinzipiell können damit alle Objekte in dynamischen Feldern
// verwaltet werden.
// Voraussetzungen:
// - bei Basisdatentypen: keine
// - bei Klassen:
//   1) die Objekte müssen mit new und new[] allokierbar sein,
//      mit delete und delete[] wieder gelöscht werden können.
//      Also: Saubere Konstruktoren (ohne Parameter, und mit
//      const&-Initialisierung) und Destruktor
//   2) die Objekte müssen mit operator= kopierbar sein.
```

```
//      Der Standardoperator = kopiert nur byteweise. Wenn die
//      Objekte also beispielsweise mit malloc() oder new beschafften
//      Speicher haben, dann braucht man auch einen sauberen
//      Kopieroperator = (der ggf. ein Objekt sauber aufsetzt).
//
//======================================================================
//
// Verwendung:
// ===========
//
// Der Anwender vereinbart ein Feld von beispielsweise int-Werten
// etwa so:
//   #include <vararr.h>
//   VarArr<int> feld;
//
// Dann kann er auf beliebige Feldelemente zugreifen, z.B.:
//   feld[5]  = 10;
//   feld[-1] = 3;
//   feld[1]  = feld[5]+feld[-1];
//   cout << feld[1];
//
// Dabei wird die Länge des Feldes dynamisch so angepaßt, daß
// die angesprochenen Werte auch existieren.
//
//
// Abfrage der aktuellen Feldgröße:
//
// Mit den Elementfunktionen ug(), og(), und l() kann man die
// untere bzw. obere maximale Feldgrenze und die aktuelle Länge
// erfragen.
// Bei einem frisch vereinbarten Feld liefert ug() den Wert INT_MAX,
// og() den Wert INT_MIN, und l() den Wert 0.
//
//
// Setzen der Feldgröße:
//
// Mit l( int lneu ) kann man die aktuelle Gesamtlänge setzen.
// Ein positives lneu setzt dabei die untere Grenze auf 0, und die
// obere Grenze auf lneu-1, so daß nachfolgende Aufrufe von l() wieder
// og()-ug()+1 liefern.
// Negative Werte oder 0 setzen das Feld zurück, sodaß die Länge
// wieder 0 ist.
// Dabei wird kein Speicher freigegeben, siehe [Funktionsweise].
```

```
//
// Mit ug( int ugneu ) und og( int ogneu ) kann man die jeweilige
// Feldgrenze von Hand setzen. Nachfolgende Aufrufe von l() liefern
// dann eine entsprechend korrigierte Länge.
// Ist ugneu größer als die bisherige obere Grenze, dann wird das Feld
// als leer betrachtet (wie mit l(0)).
// Ebenso, wenn ogneu kleiner ist als die bisherige untere Grenze.
// Bei einem vorher leeren Feld (neu vereinbart, oder mit l(0)
// geleert) setzt wird mit der oberen Grenze auch gleichzeitig die
// untere gesetzt, und umgekehrt. Bei einem leeren Feld setzt also
// og(10) auch gleichzeitig die Untergrenze auf 10. Das Element [10]
// ist dann das einzige Element im Feld. ug() und og() liefern dann
// den Wert 10, l() liefert 1.
//
// ug(int), og(int), und l(int) liefern jeweils den neuen Wert des
// entsprechenden Parameters zurück (ihr Argument, oder 0 für l(int),
// INT_MAX für ug(int), und INT_MIN für og(int)).
//
//
// Unbenutzte Lücken im Feld:
//
// Es wird intern NICHT kontrolliert, welche Elemente belegt sind.
// Nur die Grenzen werden mitgeführt und gegebenenfalls angepaßt.
// Das heißt, daß Elemente, die nie belegt wurden, gelesen werden
// können, aber ihr Inhalt ist undefiniert (es sei denn, es ist
// für die Elemente ein Konstruktor definiert, der eine definierte
// Vorbelegung vornimmt).
//
// Bei Bedarf kann man mit SetDefault() einen Wert für zukünftig
// entstehende Lücken setzen; bis an diese Elemente etwas anderes
// zugewiesen wird, haben sie dann wenigstens einen definierten Inhalt
// (beispielsweise macht es Sinn, für ein Feld von Zeigern NULL als
// Standardwert zu verwenden).
//
// Bereits bestehende Lücken werden von SetDefault() nicht
// beeinflußt.
//
//
// Debug-Ausgabe:
//
// Durch #definieren von DEBUGVARARR vor dem #includen von vararr.h
// ereicht man eine ausführliche Ausgabe der internen Vorgänge.
//
```

```
//=====================================================================
//
// Funktionsweise:
// ===============
//
// Es wird beim ersten Zugriff auf ein Element ein Stück Speicher
// nur für dieses Feldelement allokiert.
// Bei jedem weiteren Zugriff (lesend oder schreibend) wird
// geprüft, ob der Zugriff in dem allokierten Bereich erfolgt, oder
// außerhalb. Gegebenenfalls wird das Feld in die richtige Richtung
// vergrößert. Dabei wird aber nicht nur bis zum aktuell benötigten
// Element vergrößert, sondern um REALLOC_MEHR Elemente mehr.
// Dadurch werden die Fälle, in denen man neu allokieren und alle
// alten Elemente umkopieren muß, wesentlich seltener.
// Nachteil ist natürlich, daß immer ein paar unbenutzte Elemente
// in der Gegend herumliegen.
//
// Es gibt zwei Feldbereiche mit eigenen Ober- und Untergrenzen:
// - Den Bereich, den der Benutzer sieht. Er reicht vom untersten
//   benutzten Element, bis zum höchsten benutzten Element.
//   Dieser Bereich wird von _ug, _og, und _l beschrieben.
// - Der allokierte Bereich ist meist größer, und reicht von
//   _pallok bis (_pallok+_lallok-1).
//
// Es werden intern zwei Zeiger auf das allokierte Feld gepflegt,
// nämlich _p0 und_pallok.
// _p0 zeigt auf das Element [0] (auch wenn die untere Grenze
// positiv ist, oder die obere Grenze negativ; dann ist _p0[0] nicht
// gültig; _p0 selbst aber wohl).
// _pallok zeigt auf das allokierte Feld.
//
// Anhand von _pallok==NULL wird erkannt, daß das Feld noch nicht
// allokiert wurde.
//
// Bei jeder Feldvergrößerung wird nicht nur bis zur aktuell
// geforderten Länge vergrößert, sondern gleich ein ordentliches Stück
// mehr genommen. Aber wieviel? Die Verlängerung ist der größere
// der beiden Werte (REALLOC_MIN1) und (REALLOC_MIN2).
// Um einerseits kleine Felder effektiv verwenden zu können, sollte
// REALLOC_MIN1 ein kleiner konstanter Wert sein, z.B. (5).
// Um andererseits bei großen Feldern das aufwendige Neuallokieren
// zu begrenzen, sollte REALLOC_MIN2 längenabhängig sein, also mit
// der Feldlänge mitwachsen, z.B.: (l_allok/2).
```

```
// Einzige Ausnahme: der erste Zugriff mit beliebigem Index läßt nur
// das geforderte Element allokieren, da VarArr nicht wissen kann, ob
// das Feld in Zukunft nach oben oder nach unten wachsen wird.
//
// Durch dieses großzügige Allokieren muß _pallok nun nicht identisch
// sein mit _p0[ug()].
// Der Benutzer sieht Feldelemente von [_ug] bis [_og], der
// allokierte Bereich reicht von _pallok bis (_pallok+_lallok-1).
// Diese beiden Bereiche müssen nicht identisch sein, aber:
// Der Benutzerbereich liegt immer innerhalb des allokierten Bereichs.
//
// Beispiel:
// REALLOC_MIN1 sei 3, und REALLOC_MIN2 sei (_lallok/2).
//
// Mit der Definition:
//   VarArr<int> feld;
// wird ein neues Feld geschaffen.
// feld sieht dann so aus:
//
//   _l      == 0
//   _lallok == 0
//   _og     == INT_MIN
//   _ug     == INT_MAX
//   _p0     == NULL
//   _pallok == NULL
//
// Durch die Zuweisung:
//   feld[0] = 12;
// wird nur Platz für feld[0] allokiert. _p0 und _pallok zeigen auf
// diesen Wert.
//
// feld hat dann beispielsweise  |   Der Speicher könnte so aussehen:
// folgenden Inhalt:             |
//
//   _l      == 1
//   _lallok == 1                        |                         |
//                                  5004 +-------------------------+
//   _og     == 0                        |      (int)feld[0]       |
//   _ug     == 0                        |      ==                 |
//                                       |      (int)12            |
//   _p0     == 5000 -----+              |                         |
//   _pallok == 5000 -----+------> 5000  +-------------------------+
//                                       |                         |
```

```
// Durch den Zugriff:                     |                         |
//   feld[-2] = 13;
// passiert folgendes:
// - das Feld muß nach unten verlängert werden. Da REALLOC_MIN1 aber
//   3 ist, müssen über das ohnehin nötige feld[-2] hinaus noch zwei
//   Elemente mehr allokiert werden (insgesamt also [-4] bis [-1]).
//   Es wird also ein Feld mit fünf Elementen allokiert, und das
//   alte Feld freigegeben (nach dem Kopieren des einzigen Elements)
// - _pallok zeigt auf das niedrigste allokierte Element feld[-4]
// - _p0 zeigt auf die neue Position von feld[0]
// - _ug wird auf -2 gesetzt
// - _l wird auf 3 gesetzt (weil für den Benutzer die Feldlänge
//   immer _og-_ug+1 ist)
// - _lallok wird auf 5 gesetzt, da das Feld von einem auf
//   fünf Elemente verlängert wird.
//
// Die Situation könnte dann so aussehen:
//
//                                        |                         |
//                                  6016  +-------------------------+
//                                        |      (int)feld[0]       |
//                                        |      ==                 |
//                                        |      (int)12            |
//                                        |                         |
//                      +---------> 6012  +-------------------------+
//                      |                 |      (int)feld[-1]      |
//                      |                 |      ==                 |
//                      |                 |      ??                 |
//                      |                 |                         |
//                      |           6008  +-------------------------+
//                      |                 |      (int)feld[-2]      |
//                      |                 |      ==                 |
//                      |                 |      (int)13            |
//                      |                 |                         |
//                      |           6004  +-------------------------+
//                      |                 |      (int)feld[-3]      |
//                      |                 |      ==                 |
//   _l      == 3       |                 |      ??                 |
//   _lallok == 4       |                 |                         |
//                      |           6000  +-------------------------+
//   _og     == 0       |                 |      (int)feld[-4]      |
//   _ug     == -2      |                 |      ==                 |
//                      |                 |      ??                 |
```

```
//   _p0     == 6012 --+                 |                        |
//   _pallok == 5996 ------------> 5996  +------------------------+
//
// Aus Effizienzgründen wird allokierter Speicher für das eigentliche
// Feld nie mehr freigegeben,
// bis der Destruktor des gesamten Feldes aufgerufen wird.
// Also wird auch beim Aufruf von l(0) nichts freigegeben, sondern es
// werden nur _l, _ug, und _og gesetzt. _p0 bleibt auf das bisherige
// Element [0] gerichtet.
//
// Ausnahme: wenn das Feld für leer erklärt wird (z.B. durch l(0)),
// und danach als erstes auf ein Element außerhalb des allokierten
// Bereichs zugegriffen wird, dann wird der gesamte allokierte Bereich
// freigegeben und nur Platz für das aktuell nötige Element allokiert.
//
// Wenn _pallok!=NULL, dann gilt:
// - ( _pallok )               zeigt auf das unterste allok. Element
// - ( _pallok + _lallok )     zeigt HINTER das allokierte Feld
// - ( _pallok + _lallok - 1 ) zeigt auf das letzte allokierte Element
// - ( _p0 )                   zeigt auf das Element [0]
// - ( _p0 + _ug )             zeigt auf das unterste Benutzerelement
// - ( _p0 + _og )             zeigt auf das oberste Benutzerelement
//
// Dazu sind private member-Funktionen definiert:
// - _p_letzt() liefert Zeiger auf das letzte allokierte Element
// - _p_ug()    liefert Zeiger auf das unterste Benutzerelement
// - _p_og()    liefert Zeiger auf das oberste Benutzerelement
// Achtung:
// Diese darf man nur aufrufen, wenn _pallok!=NULL ist!!!
//
//======================================================================
//
// Benötigt:
// =========
//
// ---
//
//======================================================================
//
// Umgebung:
// =========
//
// getestet mit:
```

```
//
//  + gcc/Linux
//  + Visual C++ 4.2 - 6.0, Windows NT 4.0/SP3
//
//======================================================================
//
// Literatur:
// ==========
//
// [1]      Klaus Wachtler: C++
//
// [2]      Gedruckte Dokumentation duh.ps, und
//          Online-Hilfe duh.hlp
//
//======================================================================


#ifndef _VARARR
#define _VARARR

#include <iostream>
#include <cstdlib>
#include <cstdio>
#include <climits>



// Mit VARARR_FELDGRENZENKONTROLLIEREN kann man eine
// Feldgrenzenüberprüfung einschalten.
// Diese wird in
//    T & operator[]( int index ) const
// wirksam, also beim Zugriff auf ein Element eines const-Objekts.
// In diesem Fall darf das Feld nicht implizit verlängert werden,
// weil ja "const". Eine Referenz auf ein Element darf dann
// auch nicht geliefert werden, also wird eine exception geworfen.
// Wenn VARARR_FELDGRENZENKONTROLLIEREN nicht definiert ist,
// dann findet keine Kontrolle statt. Vielmehr wird dann eine
// unbrauchbare Referenz auf irgendwas im Speicher geliefert.
// Viel Spaß damit!
// Seit 08.09.2002 wird in diesem Fall eine Ausnahme vom Typ
// std::out_of_range geworfen (vorher war es eine Ausnahme von einem
// eigenen Typ CError).
```

```
#define VARARR_FELDGRENZENKONTROLLIEREN

#define nDEBUGVARARR


#ifdef VARARR_FELDGRENZENKONTROLLIEREN
#include <stdexcept>
#endif /* ifdef VARARR_FELDGRENZENKONTROLLIEREN */

#ifndef REALLOC_MIN1
#define REALLOC_MIN1 (5)
#endif /* ndef REALLOC_MIN1 */

#ifndef REALLOC_MIN2
#define REALLOC_MIN2 (_lallok/2)
#endif /* ndef REALLOC_MIN2 */

// Es werden REALLOC_MEHR Elemente allokiert, als mindestens nötig.
// Mindestens nötig wäre immer 1 (für das aktuelle).
#ifndef REALLOC_MEHR
#define REALLOC_MEHR                                                  \
 ( ( REALLOC_MIN1>REALLOC_MIN2 ? REALLOC_MIN1 : REALLOC_MIN2 ) - 1 )
#endif /* ndef REALLOC_MEHR */

#ifndef assert
#include <assert.h>
#endif /* ifndef assert */

template<class T> class VarArr
{
private:

  int        _l;        // momentane Länge (für Benutzer)
  int        _lallok;   // allokierte Länge

  int        _ug;       // untere Feldgrenze (für Benutzer)
  int        _og;       // obere Feldgrenze (für Benutzer)

  T        * _p0;       // Zeiger auf Element [0] (auch dann, wenn
                        // [0] nicht im Feld enthalten ist)
  T        * _pallok;   // Zeiger auf das eigentliche Feld

  T        * _default;  // NULL oder Zeiger auf default-Wert
```

```
  // Vorbelegung für leeres Feld;
  void reset()
  {
#ifdef DEBUGVARARR
    cerr << "[-] VarArr<T>::reset()" << endl;
#endif /* DEBUGVARARR */

    _l
      = _lallok
      = 0;
    _og = INT_MIN;
    _ug = INT_MAX;
    _p0
      = _pallok
      = _default
      = NULL;
  }

  // gibt ein belegtes Feld frei (es wird nicht tatsächlich
  // freigegeben; vielmehr werden _ug, _og und _l gesetzt):
  void loesche()
  {
#ifdef DEBUGVARARR
    cerr << "[-] VarArr<T>::loesche()" << endl;
#endif /* DEBUGVARARR */

    if( _default && _lallok>0 )
      {
        for( int i=0; i<_lallok; i++ )
          {
            _pallok[i] = *_default;
          }
      }
    else
      {
      }

    _l  = 0;
    _og = INT_MIN;
    _ug = INT_MAX;
```

```
    // Testweise geben wir alles frei:
    delete[] _pallok; _pallok = NULL;   // Speicher freigeben
    reset();             // Initialisieren
    //.

  }

  // liefert Zeiger auf das letzte allokierte Element
  // (_pallok muß !=NULL sein!!!!!!!)
  T *_p_letzt() const
  {
#ifdef DEBUGVARARR
    cerr << "[-] T * VarArr<T>::_p_letzt()" << endl;
#endif /* DEBUGVARARR */

    assert( _pallok!=NULL );
    return ( _pallok + _lallok - 1 );
  }

  // liefert Zeiger auf das unterste Benutzerelement
  // (_pallok muß !=NULL sein!!!!!!!)
  T *_p_ug() const
  {
#ifdef DEBUGVARARR
    cerr << "[-] T * VarArr<T>::_p_ug()" << endl;
#endif /* DEBUGVARARR */

    assert( _pallok!=NULL );
    return ( _p0 + _ug );
  }

  // liefert Zeiger auf das oberste Benutzerelement
  // (_pallok muß !=NULL sein!!!!!!!)
  T *_p_og() const
  {
#ifdef DEBUGVARARR
    cerr << "[-] T * VarArr<T>::_p_og()" << endl;
#endif /* DEBUGVARARR */


    assert( _pallok!=NULL );
```

```
    return ( _p0 + _og );
  }


  // sorgt dafür, daß das Element [index_neu] vorhanden ist.
  void resize( int index_neu )
  {
#ifdef DEBUGVARARR
    cerr << "[-] VarArr<T>::resize( int index_neu = "
         << index_neu << ")" << endl;
    cerr << "Default "
         << ( _default==NULL ? "nicht " : ""  )
         << "gesetzt!\n";
#endif /* DEBUGVARARR */

    if( !_pallok )
      {
        // noch gar kein Speicher allokiert!
        // Aber das kann man nachholen:
#ifdef DEBUGVARARR
        cerr << "[-] - bisher noch kein Speicher allokiert!" << endl;
#endif /* DEBUGVARARR */

        _l       = 1;
        _lallok  = 1;
        _pallok  = new T[1];
        if( _default )
          {
            *_pallok = *_default;
          }
        else
          {
          }

        _p0      = _pallok - index_neu;
        _ug
          = _og
          = index_neu;

        return;
      }
    else
      { // Feld ist schon allokiert.
```

```
#ifdef DEBUGVARARR
        cerr << "[-] - Feld ist schon allokiert." << endl;
#endif /* DEBUGVARARR */

        if( _l<=0 )
          { // Feld ist zwar allokiert, aber wieder freigegeben.
            // Wenn der verlangte Wert feld[index_neu] im allokierten
            // Bereich liegt, dann wird dieses eine Element als neues
            // _ug und _og betrachtet; die Länge ist dann 1.
#ifdef DEBUGVARARR
            cerr << "[-] - _l<=0, also Feld freigegeben." << endl;
#endif /* DEBUGVARARR */

            if( ( _p0 + index_neu ) >= _pallok
                && ( _p0 + index_neu ) <= _p_letzt()
                )
              { // feld[index_neu] liegt im bereits vorhandenen Feld.
#ifdef DEBUGVARARR
                cerr << "[-] - index_neu liegt im allokierten "
                     << "Bereich." << endl;
#endif /* DEBUGVARARR */

                _l = 1;
                _ug
                  = _og
                  = index_neu;

                if( _default )
                  {
                    for( int i=0; i<_lallok; i++ )
                      {
                        _pallok[i] = *_default;
                      }
                  }
                else
                  {
                  }

                return ;
              } // feld[index_neu] liegt im bereits vorhandenen Feld.
            else
              {
                // Feld ist allokiert, aber wieder freigegeben,
```

```
                // und das erste neue Element [index_neu] liegt
                // außerhalb des allokierten Bereichs.
                // Dann das Feld freigeben, und neu anfangen:
#ifdef DEBUGVARARR
                cerr << "[-] - index_neu liegt ausse"
                     << "rhalb des allokierten Bereichs.\n"
                     << "[-]   Also delete[], und neu "
                     << "allokieren!" << endl;
#endif /* DEBUGVARARR */

                delete[] _pallok; _pallok = NULL;
                reset();             // Initialisieren
                resize( index_neu ); // (rekursiv) allokieren

                return;
              }
          } // Feld ist zwar allokiert, aber wieder freigegeben.

        // Kontrollieren, ob die Grenzen zur Länge passen:
        assert( (_l==( _og - _ug + 1 )) );

        // Hier ist ein Feld vorhanden, und die Länge ist
        // mindestens 1.

        // reicht die allokierte Länge?
        if( &_p0[index_neu] > _p_letzt() )
          { // Feld muß nach oben verlängert werden!
#ifdef DEBUGVARARR
            cerr << "[-] - Feld muss nach oben verlaengert werden!"
                 << endl;
#endif /* DEBUGVARARR */

            // neue Länge:
            int lallok_neu
              // Adresse des letzten allokierten Elements, wenn
              // das Feld an der gleichen Stelle bleiben würde:
              = ( &_p0[index_neu + REALLOC_MEHR]
                   // minus Basisadresse ergibt (neue Länge-1)
                   - _pallok
                   // plus 1 ergibt neue Länge (Zaunlatten!)
                   + 1
                   );
#ifdef DEBUGVARARR
```

```
            cerr << "[-] - REALLOC_MIN1 = " << REALLOC_MIN1;
            cerr << ", REALLOC_MIN2 = " << REALLOC_MIN2 << endl;
            cerr << "[-]   REALLOC_MEHR = " << REALLOC_MEHR
                 << ", lallok_neu = " << lallok_neu << endl;
#endif /* DEBUGVARARR */

            assert( lallok_neu>_lallok );

            // neues Feld:
            T    * pallok_neu = new T[lallok_neu];

            if( _default )
              {
                for( int i=_lallok; i<lallok_neu; i++ )
                  {
                    pallok_neu[i] = *_default;
                  }
              }
            else
              {
              }


            // Zeiger auf neues Element [0]:
            // Das neue Element [0] soll im neuen Feld den gleichen
            // Offset haben, wie das alte [0] im alten Feld.
            // Das heißt: ( _p0 - _pallok ) == ( p0_neu - pallok_neu )
            // Oder:      p0_neu == ( _p0 - _pallok + pallok_neu  )

            // Auch hier muß man sicherstellen, daß nur Differenzen
            // von Zeigern gebildet werden, die sich auf EINEN
            // allokierten Speicherblock beziehen (siehe unten):
            // T    * p0_neu = ( _p0 - _pallok + pallok_neu  );
            T    * p0_neu = &pallok_neu[(_p0 - _pallok)];


            // Änderung 27.05.2000 kw:
            // dies war der alte Code:

//            // Damit kann man die Elemente kopieren:
//            for( int i=_ug; i<=_og; i++ )
//              {
//#ifdef DEBUGVARARR
```

```
//                cerr << "[-] - Kopiere Element " << i << endl;
//#endif /* DEBUGVARARR */
//                p0_neu[i] = _p0[i];
//              }

            // Fehler darin ist, daß nur die Elemente des Bereichs
            // umkopiert werden, die der Benutzer sieht
            // ((_ug)...(_og)).
            // Wenn jedoch (_default) gesetzt ist, dann muß man
            // alle bisher allokierten Elemente umkopieren, weil
            // sonst die Elemente, die durch Mehrallokieren
            // außerhalb von (_ug)...(_og) liegen, verlorengehen.
            // Diese "Vorratselemente" müssen ja auch mit in das
            // neu allokierte Feld übernommen werden.
            // Weil dies aber wiederum überflüssig ist, wenn
            // (_default) nicht gesetzt ist, gibt es hier eine
            // Fallunterscheidung:
            if( _default )
              { // _default, also alle allokierten Elemente kopieren
                for( int i=0; i<_lallok; i++ )
                  {
#ifdef DEBUGVARARR
                    cerr << "[-] - Kopiere all. Element " << i << endl;
#endif /* DEBUGVARARR */
                    pallok_neu[i] = _pallok[i];
                  }
              } // _default, also alle allokierten Elemente kopieren
            else
              { // KEIN _default, also nur von _ug bis _og kopieren
                for( int i=_ug; i<=_og; i++ )
                  {
#ifdef DEBUGVARARR
                    cerr << "[-] - Kopiere Element " << i << endl;
#endif /* DEBUGVARARR */
                    p0_neu[i] = _p0[i];
                  }
              } // KEIN _default, also nur von _ug bis _og kopieren

            // Das alte Feld kann weg:
            delete[] _pallok;

            _pallok = pallok_neu;
            _og     = index_neu;
```

```
            _p0     = p0_neu;
            _lallok = lallok_neu;
            _l      = _og - _ug + 1;
#ifdef DEBUGVARARR
            cerr << "[-] - _ug = " << _ug << endl;
            cerr << "[-]   _og = " << _og << endl;
            cerr << "[-]   _l  = " << _l  << endl;
#endif /* DEBUGVARARR */

            return;
          } // Feld muß nach oben verlängert werden!
        else if( &_p0[index_neu] < _pallok )
          { // Feld muß nach unten verlängert werden!
#ifdef DEBUGVARARR
    cerr << "[-] - Feld muss nach unten verlaengert werden!" << endl;
#endif /* DEBUGVARARR */

            // neue Länge:
            int lallok_neu =
              // erste Adresse hinter dem allokierten Feld:
              &_pallok[_lallok]
              // minus Adresse des ersten allokierten Elements, wenn
              // das Feld nach unten verlängert werden könnte:
              - &_p0[index_neu-REALLOC_MEHR];
#ifdef DEBUGVARARR
            cerr << "[-] - REALLOC_MIN1 = " << REALLOC_MIN1;
            cerr << ", REALLOC_MIN2 = " << REALLOC_MIN2 << endl;
            cerr << "[-]   REALLOC_MEHR = " << REALLOC_MEHR
                 << ", lallok_neu = " << lallok_neu << endl;
#endif /* DEBUGVARARR */

            assert( lallok_neu>_lallok );

            // neues Feld:
            T    * pallok_neu = new T[lallok_neu];

            if( _default )
              {
                for( int i=0; i<(lallok_neu-_lallok); i++ )
                  {
                    pallok_neu[i] = *_default;
                  }
              }
```

```
else
  {
  }

// Zeiger auf neues Element [0]:
// Das neue Element [0] soll im neuen Feld den gleichen
// Abstand vom Ende haben, wie das alte [0] im alten Feld.
// Das heißt:
// ( _pallok + _lallok - _p0 )
//            ==
// ( pallok_neu + lallok_neu - p0_neu )
// Oder:
// p0_neu
//   ==
// ( pallok_neu+lallok_neu-( _pallok+_lallok-_p0 ) )
// Oder:
// p0_neu
//   ==
// ( pallok_neu+lallok_neu+( -_pallok-_lallok+_p0 ) )
// Oder:
// p0_neu
//   ==
// ( pallok_neu+lallok_neu-_pallok-_lallok+_p0 )
//
//T    * p0_neu =
//  (&pallok_neu[lallok_neu]
//   - &_pallok[_lallok]
//   )
//  + _p0;

// Diese Version geht schief (bei sizeof(T)!=n*8 kommen
// Adressen heraus, die nicht Vielfache von sizeof(T)
// sind!!!!!!!!!!!!!!!!!!!!!!!
//T * p0_neu = pallok_neu+lallok_neu-_pallok-_lallok+_p0;

// Vielmehr muß man den Ausdruck so zusammenfassen, daß
// nur Differenzen von Adressen gebildet werden, die in
// einem allokierten Block liegen.
// Haben die allokierten Blöcke nämlich einen Abstand, der
// ungleich einem Vielfachen der Elementgröße ist, dann
// kommen Rundungsfehler ins Spiel (weil die Differenzen
// von Adressen immer durch sizeof(...) geteilt werden.
```

```
            T    * p0_neu =
              &pallok_neu[lallok_neu] - ( &_pallok[_lallok] - _p0 );


#ifdef DEBUGVARARR
                cerr << "[-] - p0_neu, pallok_neu = "
                     << p0_neu << ", " << pallok_neu << endl;
#endif /* DEBUGVARARR */

            // Änderung 27.05.2000 kw:
            // dies war der alte Code:

//            // Damit kann man die Elemente kopieren:
//            for( int i=_ug; i<=_og; i++ )
//              {
//
//#ifdef DEBUGVARARR
//                cerr << "[-] - Kopiere Element " << i << endl;
//#endif /* DEBUGVARARR */
//
//                p0_neu[i] = _p0[i];
//              }

            // Fehler darin ist, daß nur die Elemente des Bereichs
            // umkopiert werden, die der Benutzer sieht
            // ((_ug)...(_og)).
            // Wenn jedoch (_default) gesetzt ist, dann muß man
            // alle bisher allokierten Elemente umkopieren, weil
            // sonst die Elemente, die durch Mehrallokieren
            // außerhalb von (_ug)...(_og) liegen, verlorengehen.
            // Diese "Vorratselemente" müssen ja auch mit in das
            // neu allokierte Feld übernommen werden.
            // Weil dies aber wiederum überflüssig ist, wenn
            // (_default) nicht gesetzt ist, gibt es hier eine
            // Fallunterscheidung:
            if( _default )
              { // _default, also alle allokierten Elemente kopieren
                for( int i=0; i<_lallok; i++ )
                  {
#ifdef DEBUGVARARR
                    cerr << "[-] - Kopiere all. Element " << i << endl;
#endif /* DEBUGVARARR */
                    pallok_neu[i+lallok_neu-_lallok] = _pallok[i];
```

```
                 }
             } // _default, also alle allokierten Elemente kopieren
           else
             { // KEIN _default, also nur von _ug bis _og kopieren
               for( int i=_ug; i<=_og; i++ )
                 {
#ifdef DEBUGVARARR
                   cerr << "[-] - Kopiere Element " << i << endl;
#endif /* DEBUGVARARR */
                   p0_neu[i] = _p0[i];
                 }
             } // KEIN _default, also nur von _ug bis _og kopieren

           // Das alte Feld kann weg:
           delete[] _pallok;
           _pallok = pallok_neu;
           _ug     = index_neu;
           _p0     = p0_neu;
           _lallok = lallok_neu;
           _l      = _og - _ug + 1;

#ifdef DEBUGVARARR
           cerr << "[-] - _ug = " << _ug << endl;
           cerr << "[-]   _og = " << _og << endl;
           cerr << "[-]   _l  = " << _l  << endl;
#endif /* DEBUGVARARR */

           return;
         } // Feld muß nach unten verlängert werden!
       else
         { // Feld ist vorhanden, Länge ist größer 0,
           // keine Verlängerung nötig, &Element[index_neu] liegt im
           // Bereich [_pallok,_p_letzt()]

           // Dann muß man nur sehen, ob index_neu im Bereich
           // _ug..._og liegt; gegebenenfalls _ug oder _og anpassen:

#ifdef DEBUGVARARR
           cerr << "[-] - Feld ist vorhanden, " << endl;
           cerr << "[-]   Laenge ist groesser 0," << endl;
           cerr << "[-]   keine Verlaengerung noetig." << endl;
           cerr << "[-]   " << endl;
#endif /* DEBUGVARARR */
```

```
            if( index_neu<_ug )
              {
                _ug = index_neu;
                _l = _og - _ug + 1;

#ifdef DEBUGVARARR
                cerr << "[-] - _ug auf " << _ug
                     << " gesetzt." << endl;
                cerr << "[-]   _l  auf " << _l
                     << " gesetzt." << endl;
#endif /* DEBUGVARARR */
              }
            if( index_neu>_og )
              {
                _og = index_neu;
                _l = _og - _ug + 1;

#ifdef DEBUGVARARR
                cerr << "[-] - _og auf " << _og
                     << " gesetzt." << endl;
                cerr << "[-]   _l  auf " << _l
                     << " gesetzt." << endl;
#endif /* DEBUGVARARR */
              }
            return;
          } // Feld ist vorhanden, Länge ist größer 0,
      } // Feld ist schon allokiert.
  } // void resize( int index_neu )

public:

  // Konstruktor:
  VarArr()
  {

#ifdef DEBUGVARARR
    cerr << "[-] VarArr<T>:: VarArr()" << endl;
#endif /* DEBUGVARARR */

    _default = (T *)NULL;

    reset();
```

```
  }

  // 17. 7.97 kw: der fehlte noch, um temporäre Objekte schaffen zu
  // können:
  // Initialisieren mit Referenz auf const-Objekt:
  VarArr<T>( const VarArr<T> & init )
  {
#ifdef DEBUGVARARR
    cerr << "[-] VarArr<T>::VarArr<T>( const VarArr<T> & init )"
         << endl;
#endif /* DEBUGVARARR */

    // alles löschen:
    reset();

    if( init._default )
      {
        _default = new T( *(init._default) );
      }

    // Initialisierer zuweisen:
    // 24. 7.97 kw:
    // mit operator= zuweisen klappt, nicht, weil zu dessen Aufruf
    // ein temporäres Objekt mit VarArr<T>( const VarArr<T> & init )
    // angelegt, wird; darin wird wieder operator= aufgerufen, dazu
    // wird...
    // Klassische Rekursion; man lernt nie aus.


    // hat das Feld einen Inhalt?
    if( init._pallok && init._l>0 )
      {
        // JA!

#ifdef DEBUGVARARR
    cerr << "[-] - Feld hat Inhalt." << endl;
#endif /* DEBUGVARARR */

        // Kontrollieren, ob die Grenzen zur Länge passen:
        assert( (init._l==( init._og - init._ug + 1 )) );

        // Diese Werte kann man direkt übernehmen:
        _l      = init._l;
```

```
        _ug     = init._ug;
        _og     = init._og;

        // Platz allokieren (nur soviel, wie nötig):
        _lallok = _l;
        _pallok = new T[_lallok];

        // _p0 auf Element [0] zeigen lassen:
        _p0 = _pallok - _ug;

        // Die belegten Elemente von der rechten Seite nach links
        // kopieren;
        for( int i=_ug; i<=_og; i++ )
          {
#ifdef DEBUGVARARR
            cerr << "[-] - Kopiere Element " << i << endl;
#endif /* DEBUGVARARR */

            _p0[i] = (init._p0)[i];
          }
      }
  }

  // Der Operator [] wird mit einer Funktion überladen, die
  // a) ein Unter- oder Überschreiten des Feldes durch Verlängern
  //    erlaubt,
  // b) eine Referenz auf das (index)-te Objekt liefert.
  T & operator[]( int index )
  {
#ifdef DEBUGVARARR
    cerr << "[-] T & VarArr<T>::operator[]( int index = "
         << index << " )" << endl;
#endif /* DEBUGVARARR */

    // 09.09.2002 kw: hier ein Schnelltest, ob resize() überhaupt
    //                nötig ist. Diese Tests werden zwar in resize()
    //                nochmals gemacht, aber in den meisten Fällen
    //                wird die Funktion gar nicht aufgerufen werden
    //                müssen (solange die meisten Zugriffe im
    //                existierenden Feld stattfinden).
    //                Damit sollte es so im Schnitt etwas schneller
    //                gehen:
    if( index>_og || index<_ug || !_pallok )
```

```
    {
      resize( index );   // Ggf. das Feld verlängern lassen
    }

    return _p0[index];   // Referenz auf (index)-tes Objekt liefern
  }

  // Der Operator [] wird mit einer Funktion überladen, die
  // a) ein Unter- oder Überschreiten des Feldes überwacht,
  //    und notfalls eine exception schmeißt,
  // b) eine Referenz auf das (index)-te Objekt liefert.
  // Im Gegensatz zum "normalen" Feldzugriff
  // T & operator[]( int index ) wird das Feld nicht manipuliert!
  // Deshalb ist diese Version auch für const-Objekte geeignet (und
  // eventuell schneller).
  T & operator[]( int index ) const
  {
#ifdef DEBUGVARARR
    cerr << "[-] T & VarArr<T>::operator[]( int index = "
         << index << " ) const" << endl;
#endif /* DEBUGVARARR */

#ifdef VARARR_FELDGRENZENKONTROLLIEREN
    if( index < _ug || index > _og )
      {
        throw( std::out_of_range( "VarArr<>::operator[](int index) const" ) );
      }
#endif // VARARR_FELDGRENZENKONTROLLIEREN

    return _p0[index];     // Referenz auf (index)-tes Objekt liefern
  }

  // Die Funktionen l(), ug(), und og() liefern die Anzahl der
  // belegten Elemente, den kleinsten belegten Index, und den
  // größten belegten Index.

  long int l() const
  {
#ifdef DEBUGVARARR
    cerr << "[-] VarArr<T>::l()" << endl;
#endif /* DEBUGVARARR */

    return _l;
```

```
  }

  long int ug() const
  {
#ifdef DEBUGVARARR
    cerr << "[-] VarArr<T>::ug()" << endl;
#endif /* DEBUGVARARR */

    return _ug;
  }

  // Destruktor:
  ~VarArr()
  {
#ifdef DEBUGVARARR
    cerr << "[-] VarArr<T>::~ VarArr()" << endl;
#endif /* DEBUGVARARR */

    if( _pallok )
      {
        delete[] _pallok; _pallok = NULL;
      }

    reset();

    if( _default )
      {
        delete _default;
      }
  }

  // 17. 7.97 kw
  // Zuweisung
  VarArr<T> & operator=( const VarArr<T> init )
  {
#ifdef DEBUGVARARR
    cerr << "[-] VarArr<T>::operator=( const VarArr<T> init )"
         << endl;
#endif /* DEBUGVARARR */

    // Ggf. altes Feld freigeben:
    if( _pallok )
      {
```

```
        delete[] _pallok; _pallok = NULL;
      }

    reset();

    // hat das Feld einen Inhalt?
    if( init._pallok && init._l>0 )
      {
        // JA!


#ifdef DEBUGVARARR
    cerr << "[-] - Feld hat Inhalt." << endl;
#endif /* DEBUGVARARR */

        // Kontrollieren, ob die Grenzen zur Länge passen:
        assert( (init._l==( init._og - init._ug + 1 )) );

        // Diese Werte kann man direkt übernehmen:
        _l      = init._l;
        _ug     = init._ug;
        _og     = init._og;

        // Platz allokieren (nur soviel, wie nötig):
        _lallok = _l;
        _pallok = new T[_lallok];

        // _p0 auf Element [0] zeigen lassen:
        _p0 = _pallok - _ug;

        // Die belegten Elemente von der rechten Seite nach links
        // kopieren;
        for( int i=_ug; i<=_og; i++ )
          {
#ifdef DEBUGVARARR
            cerr << "[-] - Kopiere Element " << i << endl;
#endif /* DEBUGVARARR */

            _p0[i] = (init._p0)[i];
          }
      }
    return *this;
  }
```

```
  long int og() const
  {
#ifdef DEBUGVARARR
    cerr << "[-] VarArr<T>::og()" << endl;
#endif /* DEBUGVARARR */

    return _og;
  }

  // Mit einem Parameter setzen die Funktionen l(int), ug(int),
  // und og(int) die Anzahl der
  // belegten Elemente (ab 0), resp. den kleinsten belegten Index,
  // resp. den größten belegten Index.

  long int l( int lneu )
  {
#ifdef DEBUGVARARR
    cerr << "[-] long int VarArr<T>::l( int lneu = "
         << lneu << " )" << endl;
#endif /* DEBUGVARARR */

    if( lneu>0 )
      {
        // neuer Bereich ist [0,(lneu-1)].
        // Dazu muß (lneu-1) existieren:
#ifdef DEBUGVARARR
        cerr << "[-] - lneu>0" << endl;
#endif /* DEBUGVARARR */

        resize( lneu - 1 );
        resize( 0 );
        _ug = 0;
        _og = lneu - 1;
        _l = lneu;
      }
    else
      {
#ifdef DEBUGVARARR
    cerr << "[-] - lneu<=0" << endl;
#endif /* DEBUGVARARR */
        // Feld soll gelöscht werden.
        loesche();
```

```
      }

    return _l;
  }

  long int ug( int ugneu )
  {
#ifdef DEBUGVARARR
    cerr << "[-] long int VarArr<T>::ug( int ugneu = "
         << ugneu << " )" << endl;
#endif /* DEBUGVARARR */

    if( _l<=0 )
      {
#ifdef DEBUGVARARR
        cerr << "[-] - Feld ist leer, neu anlegen: [ugneu]"
             << endl;
#endif /* DEBUGVARARR */

        // Feld ist bisher leer!
        // Dann das Element [ugneu] schaffen:
        resize( ugneu );
        _og = _ug = ugneu;
        _l  = 1;
      }
    else if( ugneu>_og )
      {
#ifdef DEBUGVARARR
        cerr << "[-] - ugneu>_og, Feld soll freigegeben werden"
             << endl;
#endif /* DEBUGVARARR */

        // Feld soll freigegeben werden:
        loesche();
      }
    else
      {
#ifdef DEBUGVARARR
        cerr << "[-] - ugneu<=_og, F"
             << "eld soll am unteren Ende angepaßt werden"
             << endl;
#endif /* DEBUGVARARR */
```

```
        // Feld soll am unteren Ende angepaßt werden:
        resize( ugneu );
        _ug = ugneu;
        _l  = _og - _ug + 1;
      }
#ifdef DEBUGVARARR
    cerr << "[-] - _ug ist jetzt " << _ug << endl;
    cerr << "[-] - _og ist jetzt " << _og << endl;
    cerr << "[-]   _l  ist jetzt " << _l  << endl;
#endif /* DEBUGVARARR */

    return _ug;
  }

  long int og( int ogneu )
  {
#ifdef DEBUGVARARR
    cerr << "[-] long int VarArr<T>::og( int ogneu = "
         << ogneu << " )" << endl;
#endif /* DEBUGVARARR */

    if( _l<=0 )
      {
#ifdef DEBUGVARARR
        cerr << "[-] - Feld ist leer, neu anlegen: [ogneu]"
             << endl;
#endif /* DEBUGVARARR */

        // Feld ist bisher leer!
        // Dann das Element [ogneu] schaffen:
        resize( ogneu );
        _og = _ug = ogneu;
        _l  = 1;
      }
    else if( ogneu<_ug )
      {
#ifdef DEBUGVARARR
        cerr << "[-] - ogneu<_ug, Feld soll freigegeben werden"
             << endl;
#endif /* DEBUGVARARR */


        // Feld soll freigegeben werden:
```

```
        loesche();
      }
    else
      {
#ifdef DEBUGVARARR
        cerr << "[-] - ogneu>=_ug, Feld soll am oberen"
             << " Ende angepasst werden" << endl;
#endif /* DEBUGVARARR */

        // Feld soll am oberen Ende angepaßt werden:
        resize( ogneu );
        _og = ogneu;
        _l  = _og - _ug + 1;
      }
#ifdef DEBUGVARARR
    cerr << "[-] - _ug ist jetzt " << _ug << endl;
    cerr << "[-] - _og ist jetzt " << _og << endl;
    cerr << "[-]   _l  ist jetzt " << _l  << endl;
#endif /* DEBUGVARARR */

    return _og;
  }

  // setzt den Defaultwert, mit dem neue Feldbereiche gefüllt
  // werden sollen
  void
  SetDefault( const T &Default )
    {
      if( _default )
        {
          delete _default;
        }

      _default = new T( Default );
    }

  void
  SetNoDefault()
    {
      if( _default )
        {
          delete _default;
          _default = (T*)NULL;
```

```
        }
    }

}; // { Ende von template<class T> class VarArr }

// vergleicht zwei VarArr<T> nach Länge und Inhalt.
// true, wenn beide gleich sind.
// Benötigt für die Klasse T den operator==().
template <class T> static bool
CmpVarArr( VarArr<T> &a1, VarArr<T> &a2 )
{
  if( a1.ug()!=a2.ug() ||  a1.og()!=a2.og() )
    {
      return false;
    }
  else
    {
      for( int i=a1.ug(); i<=a1.og(); i++ )
        {
          if( !(a1[i]==a2[i]) )
            {
              return false;
            }
        }
    }
  return true;
}

#endif /* _VARARR */
```

Hier das zugehörige Testprogramm:

```
// Time-stamp: "09.09.02 19:43 test.cpp klaus@wachtler.de"
//
// Kleines Demoprogramm für VarArr

#define NDEBUG  // um diverse assert() in vararr.h stillzulegen

#include "vararr.h" // oder <vararr.h>

#include <iostream>
```

```
#include <string>


//////////////////////////////////////////////////////////////////////////////

// demonstriert ein einfaches Feld:
void demo_simple_int()
{
  // Feld vereinbaren:
  VarArr<int>     feld;

  // Beliebige Elemente initialisieren und verwenden:
  feld[1] = 11;
  feld[0] = 10;
  feld[-5] = 15;

  std::cout << "\ndemo_simple_int:" << std::endl;
  std::cout << "feld[-5] = " << feld[-5] <<std::endl;
  std::cout << "feld[0]  = " << feld[0] <<std::endl;
  std::cout << "feld[1]  = " << feld[1] << std::endl;
}

//////////////////////////////////////////////////////////////////////////////

// demonstriert ein einfaches Feld:
void demo_simple_double()
{
  // Feld vereinbaren:
  VarArr<double>     feld;

  // Beliebige Elemente initialisieren und verwenden:
  feld[1] = 11.11;
  feld[0] = 10.00;
  feld[-5] = 15.55;

  std::cout << "\ndemo_simple_double:" << std::endl;
  std::cout << "feld[-5] = " << feld[-5] <<std::endl;
  std::cout << "feld[0]  = " << feld[0] <<std::endl;
  std::cout << "feld[1]  = " << feld[1] << std::endl;
}

//////////////////////////////////////////////////////////////////////////////
```

```
// demonstriert ein einfaches Feld mit einem default-Wert für nicht
// definierte Elemente:
void demo_default_int()
{
  // Feld vereinbaren:
  VarArr<int>     feld;

  // einen Wert als Standardwert setzen:
  feld.SetDefault( 12 );

  // Beliebige Elemente initialisieren und verwenden:
  feld[1] = 11;
  feld[0] = 10;
  feld[-5] = 15;

  std::cout << "\ndemo_default_int:" << std::endl;

  // Alle Elemente vom kleinsten benutzten bis zum höchsten benutzten
  // ausgeben (obwohl nicht alle explizit initialisiert wurden!):
  for( int i=feld.ug(); i<=feld.og(); i++ )
  {
    std::cout << "feld[" << i << "] = " << feld[i] <<std::endl;
  }

}

///////////////////////////////////////////////////////////////////////////

// demonstriert ein Feld einer eigenen Klasse:
class Student
{
  friend std::ostream & operator<<( std::ostream &os, const Student &obj );

public:
  Student( std::string name = "", int iq = 30 )
    : name(name), iq(iq) {}
  std::string getName() { return name; }
  int getIQ() { return iq; }

private:
  std::string name;
  int         iq;
};
```

```
std::ostream & operator<<( std::ostream &os, const Student &obj )
{
  return (os<<obj.name<<","<<obj.iq);
}

void demo_student()
{
  VarArr<Student>  feld;

  std::cout << "\ndemo_student:" << std::endl;

  feld[3] = Student( "Hans-Peter", 12 );
  feld[5] = Student( "Birgit", 103 );

  // Alle Elemente vom kleinsten benutzten bis zum höchsten benutzten
  // ausgeben (obwohl nicht alle explizit initialisiert wurden!):
  for( int i=feld.ug(); i<=feld.og(); i++ )
  {
    std::cout << "feld[" << i << "] = " << feld[i] <<std::endl;
  }

}

//////////////////////////////////////////////////////////////////////////////

// vergleicht anhand einer Mischung aus lesenden und schreibenden
// Zugriffen die Laufzeiten für echte Felder, VarArr<> und vector<>
// aus der STL.
//
// Auf meinem Laptop (PIII, 1.1GHz) ergibt das mit dem g++ 2.95.3 (Linux):
//  klaus@lap2:~/vararr > g++ -Wall  test.cpp -O0 && a.out
//  demo_zeit:
//  belegen echtes Feld:     0.03500000 usec/Element
//  verwenden echtes Feld:   0.05725000 usec/Zugriff
//  belegen var. Feld:       0.20700000 usec/Element
//  verwenden var. Feld:     0.06125000 usec/Zugriff
//  verwenden var. Feld:     0.06700000 usec/Zugriff (const)
//  belegen vector-Feld:     0.13700000 usec/Element
//  verwenden vector-Feld:   0.06800000 usec/Zugriff (ungeprüft)
//  klaus@lap2:~/vararr > g++ -Wall  test.cpp -O3 && a.out
//  demo_zeit:
//  belegen echtes Feld:     0.03000000 usec/Element
```

```
//  verwenden echtes Feld:   0.01530000 usec/Zugriff
//  belegen var. Feld:       0.13200000 usec/Element
//  verwenden var. Feld:     0.02635000 usec/Zugriff
//  verwenden var. Feld:     0.02335000 usec/Zugriff (const)
//  belegen vector-Feld:     0.08700000 usec/Element
//  verwenden vector-Feld:   0.01555000 usec/Zugriff (ungeprüft)
// bzw. unter W2k mit VC++ 6.0:
//  Y:\>cl /GX test.cpp  & .\test
//  demo_zeit:
//  belegen echtes Feld:     0.02800000 usec/Element
//  verwenden echtes Feld:   0.04226000 usec/Zugriff
//  belegen var. Feld:       0.18930000 usec/Element
//  verwenden var. Feld:     0.05382500 usec/Zugriff
//  verwenden var. Feld:     0.05308000 usec/Zugriff (const)
//  belegen vector-Feld:     0.37950000 usec/Element
//  verwenden vector-Feld:   0.05518000 usec/Zugriff (ungeprüft)
//  Y:\>cl /Ox /GX test.cpp  & .\test
//  demo_zeit:
//  belegen echtes Feld:     0.02800000 usec/Element
//  verwenden echtes Feld:   0.03675500 usec/Zugriff
//  belegen var. Feld:       0.15120000 usec/Element
//  verwenden var. Feld:     0.03560000 usec/Zugriff
//  verwenden var. Feld:     0.03900500 usec/Zugriff (const)
//  belegen vector-Feld:     0.16730000 usec/Element
//  verwenden vector-Feld:   0.03675000 usec/Zugriff (ungeprüft)


// Anzahl der Durchläufe der verschiedenen Varianten (um die
// Genauigkeit von clock() zu erhöhen); je nach Rechner müssen die
// Werte ggf. angepaßt werden:
#define NDURCHLAUF_ARRAY   100000000
#define NDURCHLAUF_VARARR  100000000
#define NDURCHLAUF_VECTOR  100000000

// gewünschte Feldlänge:
#define N_ELEMENTE         10000000

#include <ctime>
#include <vector>

void demo_zeit()
{
  clock_t start, ende;
```

```
size_t  i;

std::cout << "\ndemo_zeit:" << std::endl;

//
// "echtes" Feld:
//
// erstmals belegen (static, weil unter VC++ sonst der Stack nicht
// reicht):
static int    echtes_feld[N_ELEMENTE];
start = clock();
for( i=0; i<N_ELEMENTE; i++ )
{
  echtes_feld[i] = (int)i;
}
ende = clock();
printf( "belegen echtes Feld:   %12.8f usec/Element\n",
        (double)( ende - start ) // clock ticks
        /CLOCKS_PER_SEC          // Umrechnen auf sec
        /N_ELEMENTE              // Zeit je Element
        *1.E6                    // Umrechnen auf mikro sec
        );

// verwenden:
start = clock();
for( i=0; i<NDURCHLAUF_ARRAY; i++ )
{
  echtes_feld[i%N_ELEMENTE] = (int)i + echtes_feld[(i-55)%N_ELEMENTE];
}
ende = clock();
printf( "verwenden echtes Feld: %12.8f usec/Zugriff\n",
        (double)( ende - start ) // clock ticks
        /CLOCKS_PER_SEC          // Umrechnen auf sec
        /NDURCHLAUF_ARRAY        // Zeit je Element
        /2.0                     // zwei Feldzugriffe je Durchlauf
        *1.E6                    // Umrechnen auf mikro sec
        );


//
// VarArr:
//
// erstmals belegen:
```

```
VarArr<int>    vararr_feld;
start = clock();
for( i=0; i<N_ELEMENTE; i++ )
{
  vararr_feld[i] = (int)i;
}
ende = clock();
printf( "belegen var. Feld:     %12.8f usec/Element\n",
        (double)( ende - start ) // clock ticks
        /CLOCKS_PER_SEC          // Umrechnen auf sec
        /N_ELEMENTE              // Zeit je Element
        *1.E6                    // Umrechnen auf mikro sec
        );

// verwenden (normales VarArr):
start = clock();
for( i=0; i<NDURCHLAUF_VARARR; i++ )
{
  vararr_feld[i%N_ELEMENTE] = (int)i + vararr_feld[(i-55)%N_ELEMENTE];
}
ende = clock();
printf( "verwenden var. Feld:   %12.8f usec/Zugriff\n",
        (double)( ende - start ) // clock ticks
        /CLOCKS_PER_SEC          // Umrechnen auf sec
        /NDURCHLAUF_VARARR       // Zeit je Element
        /2.0                     // zwei Feldzugriffe je Durchlauf
        *1.E6                    // Umrechnen auf mikro sec
        );

// verwenden (const VarArr):
const VarArr<int> &vararr_feld_const = vararr_feld;
start = clock();
for( i=0; i<NDURCHLAUF_VARARR; i++ )
{
  vararr_feld_const[i%N_ELEMENTE]
    = (int)i + vararr_feld_const[(i-55)%N_ELEMENTE];
}
ende = clock();
printf( "verwenden var. Feld:   %12.8f usec/Zugriff (const)\n",
        (double)( ende - start )  // clock ticks
        /CLOCKS_PER_SEC           // Umrechnen auf sec
        /NDURCHLAUF_VARARR        // Zeit je Element
        /2.0                      // zwei Feldzugriffe je Durchlauf
```

```
        *1.E6                     // Umrechnen auf mikro sec
        );


  //
  // vector aus der STL:
  //
  // erstmals belegen:
  std::vector<int>    vector_feld; // Anm.: Man könnte hier auch gleich die
                                   // vermutliche Länge angeben, um
                                   // späteres teures Allokieren vermeiden
                                   // zu helfen.
                                   // Um aber wirklich dynamisches
                                   // Verhalten zu simulieren, stellen wir
                                   // uns dumm und wissen die Länge nicht.
  start = clock();
  for( i=0; i<N_ELEMENTE; i++ )
  {
    vector_feld.push_back( (int)i );
  }
  ende = clock();
  printf( "belegen vector-Feld:   %12.8f usec/Element\n",
          (double)( ende - start ) // clock ticks
          /CLOCKS_PER_SEC          // Umrechnen auf sec
          /N_ELEMENTE              // Zeit je Element
          *1.E6                    // Umrechnen auf mikro sec
          );

  // verwenden (ohne Feldgrenzenüberprüfung):
  start = clock();
  for( i=0; i<NDURCHLAUF_VECTOR; i++ )
  {
    vector_feld[i%N_ELEMENTE] = (int)i + vector_feld[(i-55)%N_ELEMENTE];
  }
  ende = clock();
  printf( "verwenden vector-Feld: %12.8f usec/Zugriff (ungeprüft)\n",
          (double)( ende - start ) // clock ticks
          /CLOCKS_PER_SEC          // Umrechnen auf sec
          /NDURCHLAUF_VECTOR       // Zeit je Element
          /2.0                     // zwei Feldzugriffe je Durchlauf
          *1.E6                    // Umrechnen auf mikro sec
          );
```

```
}

//////////////////////////////////////////////////////////////////////////////

// demonstriert ein zweidimensionales Feld:
void demo_2dim_int()
{
  // Feld von Feld vereinbaren:
  VarArr< VarArr<int> >     feld;
  // Leerzeichen hier^ wichtig!


  // Beliebige Elemente initialisieren und verwenden:
  feld[1][-1]  = 11;
  feld[1][0]   = 10;
  feld[1][1]   = 11;
  feld[0][3]   = 3;
  feld[0][4]   = 4;
  feld[0][5]   = 5;
  feld[0][6]   = 6;
  feld[0][8]   = 8;
  feld[0][9]   = 9;
  feld[-5][10] = -510;
  feld[-5][11] = -511;
  feld[-5][12] = -512;
  feld[-5][13] = -513;

  // alle Werte ausgeben:
  std::cout << "\ndemo_2dim_int:" << std::endl;
  for( int i=feld.ug(); i<=feld.og(); i++ )
  {
    for( int j=feld[i].ug(); j<=feld[i].og(); j++ )
    {
      std::cout << "feld[" << i << "][" << j << "] = "
                << feld[i][j]
                << std::endl;
    }
  }
}

//////////////////////////////////////////////////////////////////////////////
```

```
int main( int nargs, char **args )
{
  demo_simple_int();    // einfaches Feld
  demo_simple_double(); // einfaches Feld
  demo_default_int();   // Feld mit default-Wert
  demo_student();       // Feld eigene class für Elemente
  demo_zeit();          // Zeitvergleich C-Array, VarArr, vector
  demo_2dim_int();      // 2-dimensionales Feld

  return 0;
}

/////////////////////////////////////////////////////////////////////////////
```

Anhang C

Was noch fehlt:

C.1 Zeiger auf überladene Funktionen

Wie bekommt man die Adresse einer überladenen Funktion? Bei mehreren überladenen Funktionen gleichen Namens (beispielsweise `f`) wird diejenige verwendet, deren Typ zur Verwendung von `&f` paßt. Demnach ist eine Verwendung, aus der der geforderte Typ nicht hervorgeht, unzulässig (wie `void *p = &f;` etwa).

```
#include <iostream>


int up( int a, int b )
{
 std::cout << "int-Version: a=" << a << ", b=" << b << std::endl;
 return 1;
}

int up( double a, double b )
{
 std::cout << "double-Version: a=" << a << ", b=" << b << std::endl;
 return 2;
}

int main( int nargs, char **args )
{
```

```
int(*p)(int, int);

// p hat den Typ "Zeiger auf Funktion, die zwei int bekommt, und
// eine int liefert". Nur deshalb wird für &up die Adresse der
// ersten up genommen.
// Das ist eine eklatante Verletzung der von mir schon im Umlauf
// gebrachten (eigentlich eisernen Regel), daß die Verwendung eines
// Ausdrucks keinen Einfluß auf seine Auswertung hat (in C gilt
// diese Regel auch uneingeschränkt!).
// Es sollte also der Typ des Ausdrucks links einer Zuweisung
// niemals einen Einfluß auf die Auswertung der rechten Seite haben;
// hier ist es aber ausnahmsweise anders:
p = &up;

// Weil p (laut seiner Deklaration) auf eine Funktion zeigt, die
// zwei int bekommt, wird beim Aufruf mit doubles implizit nach int
// gecastet; es wird in beiden Fällen das erste up korrekt
// aufgerufen:
(*p)( 12, 13 );
(*p)( 12.8, 13.9 );


// Jetzt machen wir etwas Versautes!
// Die Adresse von up wird jetzt mit einem cast zu einer
// (int(*)( double, double )) gemacht. Dieser cast bewirkt
// tatsächlich etwas, nämlich daß die Adresse der zweiten
// up-Funktion genommen wird (die mit den double als Parameter).
// Dieser cast beeinflußt also auf die Auswertung von &up.
// Als
//    p = (int(*)( double, double ))&up;
// wäre die Zuweisung nicht zulässig (Compilerfehler, weil der Typ
// der zugewiesen wird, nicht mit dem Typ der Variablen p
// zusammenpaßt).
// Das Ergebnis wird aber mit dem linken cast (int(*)( int, int ))
// in einen "Zeiger auf Funktion, die zwei int bekommt und eine int
// liefert" gecastet; die Zuweisung wird damit erzwungen.
// Dieser cast bewirkt jetzt nur, daß der Compiler die Zuweisung
// akzeptiert und hat sonst keine Auswirkung.
p = (int(*)( int, int ))  (int(*)( double, double ))&up;


// Jetzt steht in p die Adresse des zweiten up, einer
// "Funktion, die zwei double bekommt und eine int liefert".
```

```
 // Der Compiler geht aber wegen der Deklaration von p davon aus, daß
 // p auf eine "Funktion, die zwei int bekommt und eine int
 // liefert" zeigt. Schade!
 std::cout << "Achtung, Schweinerei! Das geht schief!" << std::endl;
 // Die beiden Parameter werden als int an das zweite up übergeben,
 // und die Bitmuster der beiden int (und des dahinter liegenden
 // Speichers, wenn double mehr Bytes belegt als int!) als double
 // verwendet:
 (*p)( 12, 13 );
 // Analog: die beiden double werden implizit nach int gecastet, und
 // die Bitmuster der ints vom zweiten up als double verwendet:
 (*p)( 12.8, 13.9 );

 return 0;
} // main( int nargs, char **args )
```

C.2 Compiler: gcc, cl

C.3 pair

C.4 Funktionsobjekte

C.5 keine Ausnahmen in Destruktoren

Aufruf von anderen Methoden mit `try` kapseln! [Cline: C++ FAQs] 359

C.6 Tabelle aller Operatoren

dazu auch alternative Schlüsselwörter!

C.7 Verweise auf Tools-Buch

C.8 `extern "C"`

`asm`

Default-Template-Parameter

In Destruktoren am besten keine Ausnahmen werfen.

Destruktoren immer virtuell definieren, damit im Zweifel der richtige aufgerufen wird (aber nicht pur virtuell).

Zuweisungsoperator und Copy-Konstruktor nicht vergessen, auf jeden Fall für Objekte mit allokierten Resourcen!

Zuweisungsoperator, Copy-Konstruktor, und Vergleichsoperatoren (`==`, `!=`, `<` etc.) sollten sinnvoll zusammenpassen.

Operatoren, deren Ergebnisse als *lvalues* dienen können (alle Zuweisungen, `operator[]`, `operator*` etc.) müssen eine Referenz liefern.

Die Funktion von überladenen Operatoren sinnvoll ändern, gegensätzliche Operatoren (`<` und `>=` beispielsweise) sollten das auch bleiben. Zusammenfassende Operatoren sollten so funktionieren wie die Einzeloperatoren (also sollte ein `a+=b` das selbe Ergebnis liefern wie ein `a = a + b`).

Binäre Operationen als `friend` besser, da der Compiler dann beide Parameter implizit konvertieren kann (sonst nur den zweiten).

Es gibt Leute, die von Referenzen als Parameter und Rückgabewerten weitgehend abraten. Namensgebung? NULL liefern!

Soweit möglich `const &` statt *call by value*: schneller, besser optimierbar

Ableiten: Zum Überschreiben muß die exakte Signatur verwendet werden, sonst Verdecken (inkl. `const`)!

intern:

```
class A_Intern
{
 public:
   // Alles, was der Anwender sehen darf...
 private:
   // Alles, was der Benutzer nicht sehen darf...
};
```

öffentlich:

```
class A_Intern;
class A
{
 private:
    A_Intern  &intern;
 public:
    A() intern( *new A_Intern )
    ~A() { delete &intern; }
};
```

Anhang D

Glossar

TODO: *C*-Bibliotheksnamen einsetzen

TODO: *C++*-Bibliotheksnamen einsetzen

In diesem Abschnitt werden in alphabetischer Reihenfolge wichtige Begriffe vorgestellt, und bei Bedarf wird auf weitere Erklärungen verwiesen.

Damit wird ein Mittelweg beschritten zwischen einem Stichwortverzeichnis (in dem alle Begriffe ohne Erklärung und Verweise auf zugehörigen Text stehen) und einer genauen Beschreibung (die für eine erste Information zu unübersichtlich ist).

Begriffe aus folgenden Gruppen werden in der gezeigten Form aufgeführt:

`beispielname` *(Schlüsselwort* C *und* C++*)*
: So wird ein Schlüsselwort aus *C* und *C++* beschrieben.

`beispielname` *(Schlüsselwort* C++*)*
: So wird ein Schlüsselwort beschrieben, das zu *C++* gehört (aber nicht zu *C* nach ANSI-Standard, auch wenn vielleicht manche Implementationen das Wort bereits verwenden)

`beispielname` **oder** ***Begriff*** (C- *und* C++-*Bibliothek)*
: Ein `beispielname` oder ein *Begriff* aus der Standardbibliothek von *C* ebenso wie von *C++*.

`beispielname` **oder** ***Begriff*** (C++-*Bibliothek)*
: Ein `beispielname` oder ein *Begriff* aus der Standardbibliothek von *C++*.

`beispielname` **oder *Begriff***
sonstiger *Begriff*

Hier jetzt die Liste der Begriffe:

Ableiten
Ableiten ist der Vorgang, aus einer Klasse eine andere, ähnliche zu bilden. Die abgeleitete Klasse erbt von der bereits vorhandenen Basisklasse die wesentlichen Eigenschaften (außer dem Namen); diese kann man dann in der abgeleiteten Klasse ändern der ergänzen.

Siehe **2 Grundsätzliche Änderungen gegenüber *C*** auf Seite 13.

Allokieren
Mit Allokieren bezeichnet man allgemein das Anfordern einer Resource des Betriebssystems zur Laufzeit eines Programms.

Meist ist damit das Allokieren von Speicher gemeint, der dann vom Betriebssystem aus einem meist *heap* genannten Speicherbereich zur Verfügung gestellt wird.

Andere allokierbare Resourcen könnten Plattenplatz, Semaphoren, Rechenzeit auf einem Prozessor, oder ähnliche Dinge sein.

`and` *(Schlüsselwort* C++*)*
Mit dem Schlüsselwort `and` hat man eine alternative Schreibweise für den *logischen UND-Operator* `&&`.

Folgende Zeilen sind also gleichwertig:

```
if( (i<5) && (j>10) ) // ...
if( (i<5) and (j>10) ) // ...
```

Siehe [KW:0 ist false].

`and_eq` *(Schlüsselwort* C++*)*
`and_eq` ist eine alternative Schreibweise für `&=`, also für das *bitweise AND* mit gleichzeitiger Zuweisung.

Siehe [KW:0 ist false].

ANSI
ANSI ist ein US-amerikanisches Normungsinstitut; etwa vergleichbar mit dem deutschen DIN.

`asm` *(Schlüsselwort* C++*)*
Mit dem Schlüsselwort `asm` kann man in einem nachfolgenden Klammerpaar einen String angeben, der als Assemblercode verstanden werden soll. Der genaue Inhalt eines solchen Strings ist systemabhängig.

Für portablen Code ist `asm` zwangsläufig ungeeignet.

`auto` *(Schlüsselwort* C *und* C++*)*
Mit `auto` kann man ausdrücken, daß eine lokale nichtstatische Variable eine sogenannte *automatische Variable* sein soll. Das ist sie aber sowieso, und für alle anderen Deklarationen ist `auto` nicht zulässig. Damit ist dieses Schlüsselwort schlicht und ergreifend überflüssig.

Siehe [KW:0 ist false].

`bitand` *(Schlüsselwort* C++*)*
Mit dem Schlüsselwort `bitand` hat man eine alternative Schreibweise für den *bitweisen UND-Operator* `&`.

Folgende Zeilen sind also gleichwertig:

```
i = j & 0x000f;
i = j bitand 0x000f;
```

Siehe [KW:0 ist false].

`bitor` *(Schlüsselwort* C++*)*
Mit dem Schlüsselwort `bitor` hat man eine alternative Schreibweise für den *bitweisen ODER-Operator* `|`.

Folgende Zeilen sind also gleichwertig:

```
i = j | 0x000f;
i = j bitor 0x000f;
```

Siehe [KW:0 ist false].

`bool` *(Schlüsselwort* C++*)*
`bool` ist der logische Datentyp; siehe **3.1 Logischer Datentyp** `bool` auf Seite 17.

`break` *(Schlüsselwort* C *und* C++*)*
Mit `break` kann man einerseits jede Schleife (`for`, `while`, `do`...`while`) schlagartig beenden, und andererseits in einem `switch` am Ende eines `case`-Zweiges vermeiden, daß der Programmfluß in das nächste `case` läuft. Siehe [KW:0 ist false].

`case` *(Schlüsselwort* C *und* C++*)*
Mit `case` werden die einzelnen Zweige einer `switch`-Anweisung markiert.

Siehe [KW:0 ist false].

cast
Mit cast bezeichnet man den Vorgang, ein Objekt eines Datentyps entweder in ein Objekt eines anderen Typs zu überführen, oder das vorhandene Objekt nur als ein Objekt des anderen Typs zu interpretieren.

Der cast kann explizit vom Programmierer erzwungen werden, oder implizit vom Compiler durchgeführt werden.

Siehe [KW:0 ist false], **3.3 Explizite Typumwandlung** auf Seite 18, und **4.14 run time type information (RTTI)** auf Seite 131.

`catch` *(Schlüsselwort* C++*)*
Mit `catch` kann man den Typ einer zu fangenden Ausnahme und den gegebenenfalls auszuführenden Code angeben; siehe **4.13 Ausnahmebehandlung** auf Seite 123.

`char` *(Schlüsselwort* C *und* C++*)*
`char` ist ein Basisdatentyp, der eine kleine ganze Zahl beziehungsweise ein einfaches Zeichen aufnehmen kann.

Siehe [KW:0 ist false].

`class` *(Schlüsselwort* C++*)*
Mit `class` kann man einen eigenen Datentyp, eine Klasse, vereinbaren. Siehe **4.12 Klassen** auf Seite 53.

class factory
Mißverständliche Bezeichnung für eine Methode, die ein Objekt ihrer Klasse erzeugt und zurückliefert; siehe **4.12.7.5 Virtuelle Konstruktoren** auf Seite 96.

Compiler
Ein Compiler ist ein Programm, welches ein Programm von einer Programmiersprache (beispielsweise *C++*) in eine andere (meist eine konkrete Maschinensprache) übersetzt.

`compl` *(Schlüsselwort* C++*)*
Als alternativen Namen für den Operator `~` kann man `compl` verwenden.

Siehe [KW:0 ist false].

`const` *(Schlüsselwort* C *und* C++*)*
Mit `const` kann man angeben, daß ein Objekt nicht vom Programm aus geändert werden darf. Beim Definieren oder Deklarieren von Variablen oder

Parameteren bezieht sich diese Eigenschaft auf eben dieses Objekt; siehe [KW:0 ist false].

Wenn `const` hinter der Parameterliste einer Methodendeklaration steht, dann ist damit gemeint, daß die Methode ihr zugrundeliegendes Objekt (also `*this`) nicht verändert; siehe **4.12.6 `const`-Methoden** auf Seite 71.

`const_cast` *(Schlüsselwort* C++*)*
Mit `const_cast` kann man von einem Ausdruck das `const`-Attribut entfernen; siehe **4.14 run time type information (RTTI)** auf Seite 131.

`continue` *(Schlüsselwort* C *und* C++*)*
Mit `continue` kann man den Rumpf einer Schleife (`for`, `while`, `do`...`while`) beenden, und mit der Reinitialisierung (bei der `for`-Schleife) beziehungsweise der Bedingung (bei den anderen Schleifen) weitermachen.

Siehe [KW:0 ist false].

`default` *(Schlüsselwort* C *und* C++*)*
Damit definiert man in einer `switch`-Anweisung den Programmteil, der ausgeführt werden soll wenn keiner der angegebenen `case`-Zweige zutrifft.

Siehe [KW:0 ist false].

`delete` *(Schlüsselwort* C++*)*
Mit `delete` kann man den Speicher freigeben, den man mit `new` *Datentyp* allokiert hat.

Mit `delete[]` kann man den Speicher freigeben, den man mit `new` *Datentyp*`[`*Anzahl*`]` allokiert hat.

Siehe **4.5 Freier Speicher `new` und `delete`** auf Seite 32.

`do` *(Schlüsselwort* C *und* C++*)*
Mit `do` beginnt man eine (nicht abweisende) `do`...`while`-Schleife.

Siehe [KW:0 ist false].

`double` *(Schlüsselwort* C *und* C++*)*
`double` ist ein Basisdatentyp zur Darstellung einer (doppelt genauen) Gleitkommazahl.

Siehe [KW:0 ist false].

`dynamic_cast` *(Schlüsselwort* C++*)*
siehe **4.14 run time type information (RTTI)** auf Seite 131.

`else` *(Schlüsselwort* C *und* C++*)*
Damit beginnt die Alternative einer `if`-Anweisung, also der Programmteil, der ausgeführt werden soll wenn die Bedingung im `if` nicht zutrifft.

Siehe [KW:0 ist false].

`enum` *(Schlüsselwort* C *und* C++*)*
: Mit `enum` kann man Aufzählungen ausdrücken; Ergebnis ist zum einen eine Menge von ganzzahligen Konstanten, und gegebenenfalls ein neuer Datentyp, dem diese Konstanten angehören.

  Siehe [KW:0 ist false], sowie **3.7 Einfachere Schreibweise bei `enum`, `struct` und `union`** auf Seite 22.

`explicit` *(Schlüsselwort* C++*)*
: Mit einem `explicit` vor einem Konstruktor kann man einen Aufruf mit anderen als den angegebenen Parametern (und einer impliziten Konversion durch den Compiler) verhindern.

  Siehe **4.12.3.7 `explicit`** auf Seite 64.

`export` *(Schlüsselwort* C++*)*
: Wenn es nicht praktikabel ist, den Code einer `template`-Funktion in jedem Modul zur Verfügung zu haben, in dem eine der generischen Funktionen verwendet wird, dann kann man mit `export` weiterkommen. Siehe **4.11.1 Templatefunktion für mehrere Module** auf Seite 51.

`extern` *(Schlüsselwort* C *und* C++*)*
: Damit kann man eine Konstante, Variable, oder Funktion deklarieren, und zudem angeben, daß die Definition möglicherweise in einem anderen Modul enthalten ist.

  Siehe [KW:0 ist false].

  Weiterhin kann man mit `extern "C"` Deklarationen erzeugen, die nicht nach *C++*-Konventionen behandelt werden, sondern nach *C*-Konventionen. Siehe **C.8 `extern ¨C¨`** auf Seite 292.

`false` *(Schlüsselwort* C++*)*
: `false` ist eine der beiden Konstanten vom Typ `bool`; siehe **3.1 Logischer Datentyp `bool`** auf Seite 17.

`float` *(Schlüsselwort* C *und* C++*)*
: `float` ist ein Basisdatentyp zur Darstellung einer (einfach genauen) Gleitkommazahl.

  Siehe [KW:0 ist false].

`for` *(Schlüsselwort* C *und* C++*)*
: Mit `for` kann man sehr flexible Zählschleifen bauen.

  Siehe [KW:0 ist false].

`fortran`
: `fortran` wird von manchen Implementationen verwendet, um eine Funktion

zu deklarieren, wenn das Bilden der Parameterliste und der Aufruf der Funktion nach Fortran-Konvention erfolgen soll.

Dadurch kann man Fortran-Unterprogramme und -Funktionen aus *C* heraus aufrufen, und umgekehrt.

`fortran` ist kein offizielles Schlüsselwort, wird aber von vielen Systemen verwendet.

Free Software Foundation

Die Free Software Foundation (FSF) ist eine Stiftung in den USA, die freie Software fördert.

`friend` *(Schlüsselwort* C++*)*

Mit `friend` kann man in einer Klasse eine andere Klasse oder Funktion so deklarieren, daß letztere auf alle Elemente (egal ob `private`, `protected` oder `public`) der aktuellen Klasse zugreifen darf.

Siehe **4.12.8 `friend`: Freunde haben freien Zutritt** auf Seite 100.

`goto` *(Schlüsselwort* C *und* C++*)*

Mit `goto` kann man einen lokalen Sprung innerhalb einer Funktion auslösen (man kann es aber auch bleiben lassen).

Siehe [KW:0 ist false].

FSF

FSF ist die Abkürzung für *Free Software Foundation*.

GNU

Ein ehrgeiziges (aber trotzdem von Erfolg gekröntes) Projekt der FSF ist GNU. Darin wird ein komplettes unixähnliches Betriebssystem entwickelt, das technisch mindestens so gut sein soll wie kommerzielle Unixvarianten, aber vollständig aus frei verfügbarer Software besteht.

Ein heute üblicherweise „Linux“ genanntes System besteht tatsächlich aus einem Kernel, dem eigentlichen Linux, sowie aus einer Unmenge Software aus dem GNU-Projekt. Deshalb sollte das komplette System eigentlich nicht „Linux“ heißen, sondern „GNU/Linux“.

Von der FSF wird aber auch die Entwicklung eines eigenen Kernels betrieben (das „Hurt“-Projekt). Ob damit allerdings gegen das weit verbreitete Linux anzukommen ist, erscheint fraglich.

Aus dem GNU-Projekt entstammt auch die *Gnu compiler collection* (*gcc*), in der etliche Compiler enthalten sind, unter anderem für *C*, *C++*, Fortran, Pascal, Ada, Java, und andere.

Weiterhin ist in die Rubrik GNU auch der EMACS einzuordnen, der möglicherweise einzig wahre Editor.

`if` *(Schlüsselwort* C *und* C++*)*
Mit `if` baut man Fallunterscheidungen.

Siehe [KW:0 ist false].

`inline` *(Schlüsselwort* C++*)*
Als `inline` deklarierte Funktionen können vom Compiler komplett überall dort eingesetzt werden, wo der Programmierer einen Aufruf der Funktion schreibt. Dadurch kann sich die Ausführungszeit unter Umständen verbessern.

Siehe **4.9 inline-Funktionen** auf Seite 44.

`int` *(Schlüsselwort* C *und* C++*)*
Der Basisdatentyp `int` stellt eine ganze Zahl dar.

Siehe [KW:0 ist false].

ISO
ISO ist eine internationale Institution zur Erstellung von Normen.

Die ANSI-Normen zu *C* und *C++* wurden als ISO-Normen übernommen.

Konstante
Als Konstante bezeichnet man Objekte im Speicher, deren Wert sich während des Programmlaufs nicht ändern kann (beispielsweise die Stringkonstante `"abc"` oder die ganzzahlige Konstante `25`), aber auch Variablen, die man mit dem Schlüsselwort `const` vor einer versehentlichen Manipulation schützt. Letzteres ist keine wirkliche Konstante, weil sich ihr Wert theoretisch ändern kann, ohne daß der Compiler dafür direkt Code erzeugt hat; siehe dazu das Schlüsselwort `volatile`.

Linker
Ein Linker setzt mehrere Teilprogramme, die in Maschinensprache vorliegen (also Ausgaben eines Compilers) zu einem lauffähigen Programm zusammen.

`long` *(Schlüsselwort* C *und* C++*)*
Als Abart von `int` kann man der Datentyp `long` möglicherweise einen größeren Zahlenbereich darstellen.

Siehe [KW:0 ist false].

Maschinensprache
Maschinensprache ist die Programmiersprache, die ein bestimmter Prozessor direkt verarbeiten kann.

Methode
Eine „Methode“ ist eine Funktion, die einer Klasse zugeordnet ist, (*member function*) und die Daten der Klasse („Elemente“) bearbeiten kann.

Microsoft

Microsoft ist ein Softwarehersteller aus den USA.

`mutable` *(Schlüsselwort* C++*)*
Mit `mutable` kann man Klassenelemente deklarieren, die von `const`-Methoden (im Gegensatz zu „normalen“, nicht `mutable` deklarierten Elementen) verändert werden dürfen.

Siehe **4.12.6 `const`-Methoden** auf Seite 71.

`namespace` *(Schlüsselwort* C++*)*
Mit `namespace` kann man den Gültigkeitsbereich von Namen steuern; siehe **4.7 Namensbereiche (`namespace`)** auf Seite 36.

`new` *(Schlüsselwort* C++*)*
Mit `new` kann man Speicher allokieren, und gegebenenfalls dafür Konstruktoren aufrufen.

Siehe **4.5 Freier Speicher `new` und `delete`** auf Seite 32.

`not` *(Schlüsselwort* C++*)*
Anstelle des `!`-Operators (logische Verneinung) kann man auch `not` schreiben.

Siehe [KW:0 ist false].

`not_eq` *(Schlüsselwort* C++*)*
`not_eq` ist eine alternative Schreibweise für `!=`, also für den Vergleich auf *Ungleichheit*.

Siehe [KW:0 ist false].

objektorientiert
Als „objektorientiert“ bezeichnet man eine Vorgehensweise zum Erstellen von Programmen (und -systemen), bei der die reale Welt abgebildet wird, indem die realen Objekte in Form von Klassen beschrieben werden, und zur Laufzeit eines Programms dann von solchen Klassen Instanzen gebildet werden (die dann jeweils einem Objekt der realen Welt entsprechen).

Eine Klasse beschreibt nicht nur die zum Abbilden nötigen Daten, sondern an gleicher Stelle auch die darauf anwendbaren Funktionen („Methoden“).

`operator` *(Schlüsselwort* C++*)*
Das Schlüsselwort `operator` ist Bestandteil aller Funktionsnamen, die Operatoren überladen.

Siehe **4.12.9 Überladen von Operatoren** auf Seite 110.

`or` *(Schlüsselwort C++)*

`or` ist eine alternative Schreibweise für den Operator `||` (*logisches ODER*).

Siehe [KW:0 ist false].

`or_eq` *(Schlüsselwort C++)*

`or_eq` ist eine alternative Schreibweise für `|=`, also für das *bitweise ODER* mit gleichzeitiger Zuweisung.

Siehe [KW:0 ist false].

`pascal`

`pascal` wird von manchen Implementationen verwendet, um eine Funktion zu deklarieren, wenn das Bilden der Parameterliste und der Aufruf der Funktion nach Pascal-Konvention erfolgen soll.

Dadurch kann man Pascal-Unterprogramme und -Funktionen aus *C* heraus aufrufen, und umgekehrt.

`pascal` ist kein offizielles Schlüsselwort, wird aber von vielen Systemen verwendet.

Polymorphie

Mit *Polymorphie* bezeichnet man die Fähigkeit bestimmter Dinge, sich je nach den gegebenen Umständen unterschiedlich darstellen zu können.

Das können überladene Funktionen sein, die je nach ihren aktuellen Parametern unterschiedlichen Programmcode ausführen (**4.10 Überladen von Funktionen** auf Seite 44), oder Objekte im Speicher, die über virtuelle Funktionen ein unterschiedliches Verhalten zeigen (**4.12.7.4 Virtuelle Methoden** auf Seite 87).

Siehe auch **2 Grundsätzliche Änderungen gegenüber *C*** auf Seite 13.

Präprozessor

Der Präprozessor ist ein Programm, das vor dem eigentlichen Übersetzen durch den *C*- oder *C++*-Compiler den Quelltext manipulieren kann.

Siehe [KW:0 ist false].

`private` *(Schlüsselwort C++)*

Mit `private` kann man in einer Klasse einen Abschnitt einleiten, in dem alle dort definierten oder deklarierten Elemente nur den Methoden der Klasse zugänglich sind; siehe **4.12.1 Allgemeines zu Klassen** auf Seite 53.

Weiterhin kann man durch eine `private`-Ableitung einer Basisklasse erreichen, daß alle Elemente der Basisklasse durch die abgeleitete Klasse betrachtet unzugänglich sind (also als `private` erscheinen); siehe **4.12.7.1 `public`, `protected` oder `private` ableiten** auf Seite 77.

`protected` *(Schlüsselwort C++)*
Mit `protected` kann man in einer Klasse einen Abschnitt einleiten, in dem alle dort definierten oder deklarierten Elemente nur den Methoden der Klasse sowie den Methoden abgeleiteter Klassen zugänglich sind; siehe **4.12.1 Allgemeines zu Klassen** auf Seite 53.

Weiterhin kann man durch eine `protected`-Ableitung einer Basisklasse erreichen, daß alle `public`-Elemente der Basisklasse durch die abgeleitete Klasse betrachtet `protected` erscheinen; siehe **4.12.7.1 `public`, `protected` oder `private` ableiten** auf Seite 77.

`public` *(Schlüsselwort C++)*
Mit `public` kann man in einer Klasse einen Abschnitt einleiten, in dem alle dort definierten oder deklarierten Elemente frei zugänglich sind; siehe **4.12.1 Allgemeines zu Klassen** auf Seite 53.

Weiterhin kann man durch eine `public`-Ableitung einer Basisklasse erreichen, daß alle Elemente der Basisklasse durch die abgeleitete Klasse betrachtet genau so zugänglich sind, wie sie in der Basisklasse vereinbart wurden; siehe **4.12.7.1 `public`, `protected` oder `private` ableiten** auf Seite 77.

`register` *(Schlüsselwort C und C++)*
Wenn man eine lokale Variable als `register` deklariert (zusätzlich zu ihrem Typ), dann gibt man dem Compiler damit einen Hinweis, daß die Programmausführung schneller werden könnte, wenn er diese Variable in einem Register hält.

Bei modernen, gut optimierenden Compilern ist das aber nicht mehr hilfreich.

Siehe [KW:0 ist false].

`reinterpret_cast` *(Schlüsselwort C++)*
Eine explizite Anweisung an den Compiler, den Speicher, den ein Objekt belegt, als einen anderen Typ zu betrachten.

Siehe **4.14 run time type information (RTTI)** auf Seite 131.

`return` *(Schlüsselwort C und C++)*
Das ist ein Rücksprung aus einer Funktion (Unterprogramm).

Siehe [KW:0 ist false].

`short` *(Schlüsselwort C und C++)*
Ganzzahliger Datentyp, der mit einer gewissen Wahrscheinlichkeit weniger Platz für jedes Element benötigt als `int`.

Siehe [KW:0 ist false].

`signed` *(Schlüsselwort* C *und* C++*)*
: Mit `signed` modifiziert man einen ganzzahligen Datentyp derart, daß er ausdrücklich vorzeichenbehaftet ist.

  Siehe [KW:0 ist false].

`sizeof` *(Schlüsselwort* C *und* C++*)*
: Ein Operator, der die Größe eines Ausdrucks oder eines Typs im Speicher liefert.

  Siehe [KW:0 ist false].

`static` *(Schlüsselwort* C *und* C++*)*
: `static` ist ein Modifizierer für Deklarationen.

  Je nach Verwendung hat `static` verschiedene Bedeutungen:

  - lokale Variablen werden durch `static` statisch (statt automatisch), und existieren während des gesamten Programmlaufs exakt einmal.
    Siehe [KW:0 ist false].
  - Die Sichtbarkeit globaler Funktionen wird durch `static` auf das aktuelle Modul beschränkt.
    Siehe [KW:0 ist false].
  - Als `static` deklarierte Elemente einer Klasse existieren nicht einmal je Instanz der Klasse, sondern im gesamten Programm genau einmal; egal wieviele Instanzen der Klasse gebildet werden.
    Siehe **4.12.4 Statische und nichtstatische Klassenelemente** auf Seite 65.
  - Als `static` deklarierte Methoden dürfen nur `static`-Elemente der Klasse ansprechen; dafür können sie auch ohne Instanzen der Klasse aufgerufen werden.
    Siehe **4.12.5 Statische und nichtstatische Methoden** auf Seite 71.

`static_cast` *(Schlüsselwort* C++*)*
: führt eine Typumwandlung durch, ähnlich wie der übliche cast `(`*Typ*`)`*Ausdruck*.

  Siehe **4.14 run time type information (RTTI)** auf Seite 131.

Stroustrup
: Bjarne Stroustrup ist der Erfinder von *C++*.

`struct` *(Schlüsselwort* C *und* C++*)*
: Mit `struct` kann man neue Datentypen aus bereits vorhandenen Typen bilden.

  Siehe [KW:0 ist false].

`switch` *(Schlüsselwort* C *und* C++*)*
Auswahlentscheidung anhand eines ganzzahligen (*integralen*) Ausdrucks.

Siehe [KW:0 ist false].

`template` *(Schlüsselwort* C++*)*
Mithilfe von `template` lassen sich Schablonen für generische Funktionen (siehe **4.11 Funktionsschablonen (template-Funktionen)** auf Seite 47) und Klassen (siehe **4.12.10 Klassenschablonen (template-Klassen)** auf Seite 118) definieren.

`this` *(Schlüsselwort* C++*)*
Über den Zeiger `this` findet eine (nichtstatische) Methode das Objekt, über das sie aufgerufen wurde.

Siehe **4.12.2 Der Zeiger `this`** auf Seite 58.

`throw` *(Schlüsselwort* C++*)*
Mit `throw` wirft man eine Ausnahme; siehe **4.13 Ausnahmebehandlung** auf Seite 123.

`true` *(Schlüsselwort* C++*)*
`true` ist eine der beiden Konstanten vom Typ `bool`; siehe **3.1 Logischer Datentyp `bool`** auf Seite 17.

`try` *(Schlüsselwort* C++*)*
`try` leitet einen Programmteil ein, in dem eine Ausnahme erwartet wird, auf die mit `catch` reagiert werden soll; siehe **4.13 Ausnahmebehandlung** auf Seite 123.

`typedef` *(Schlüsselwort* C *und* C++*)*
`typedef` vereinbart einen neuen Namen für einen Datentypen.

Siehe [KW:0 ist false].

`typeid` *(Schlüsselwort* C++*)*
Mit dem Operator `typeid` kann man zur Laufzeit den Typ eines Objekts in Erfahrung bringen; siehe run time type information (RTTI)rtti.

`typename` *(Schlüsselwort* C++*)*
Mit `typename` sagt man, daß ein nachfolgender Bezeichner ein Typ sein muß; siehe **4.12.10 Klassenschablonen (template-Klassen)** auf Seite 118.

`union` *(Schlüsselwort* C *und* C++*)*
Mit einer `union` kann man ein und dieselbe Speicherstelle als verschiedene Datentypen ansprechen.

Siehe [KW:0 ist false], sowie **3.7 Einfachere Schreibweise bei `enum`, `struct` und `union`** auf Seite 22 und **4.3 Anonyme `union`** auf Seite 27.

`unsigned` *(Schlüsselwort* C *und* C++*)*
: Mit `unsigned` modifiziert man einen ganzzahligen Datentyp derart, daß er ausdrücklich **nicht** vorzeichenbehaftet ist (mithin nur nichtnegative Werte annehmen kann).

Siehe [KW:0 ist false].

`using` *(Schlüsselwort* C++*)*
: Zum einfacheren Zugriff auf Namen, die in einem `namespace` deklariert wurden, kann man `using` benutzen; siehe **4.7 Namensbereiche (`namespace`)** auf Seite 36.

Vererbung
: Mit Vererbung bezeichnet man den Vorgang, daß beim Ableiten einer Klasse von einer anderen (Basisklasse) Eigenschaften übernommen werden; hauptsächlich die bereits in der Basisklasse definierten Elemente, Methoden, sowie die zugehörigen Zugriffsrechte.

Siehe **2 Grundsätzliche Änderungen gegenüber *C*** auf Seite 13.

`virtual` *(Schlüsselwort* C++*)*
: Durch `virtual` kann man den Compiler darauf hinweisen, daß eine Methode in einer abgeleiteten Klasse möglicherweise überschrieben wird; siehe **4.12.7.4 Virtuelle Methoden** auf Seite 87.

Zudem kann man im Falle von **4.12.8.1 Mehrfachvererbung** auf Seite 101 unterdrücken, daß Elemente einer mehrfach indirekt beerbten Basisklasse in der abgeleiteten Klasse mehrfach auftreten.

Virtueller Konstruktor
: Bezeichnung für eine Methode, die ein Objekt ihrer Klasse erzeugt und zurückliefert; siehe **4.12.7.5 Virtuelle Konstruktoren** auf Seite 96

`void` *(Schlüsselwort* C *und* C++*)*
: Als Rückgabetyp von Funktionen angegeben, sagt `void`, daß die Funktion **nichts** zurückgibt. Siehe [KW:0 ist false].

In einer Parameterliste drückt man mit `void` aus, daß die Funktion keine Parameter erwartet. Siehe [KW:0 ist false] und **3.5 Leere Parameterlisten** auf Seite 20.

Weiterhin ist der Datentyp `void*` ein Zeiger auf irgendetwas im Speicher. Siehe [KW:0 ist false].

Schließlich kann man mit `void` auch das Ergebnis eines Ausdrucks explizit verwerfen; das wird aber kaum genutzt, weil man das Ergebnis ja ohnehin nicht verwenden muß (außer, um in einem `?:`-Ausdruck, der nichts liefern soll, die beiden Teilausdrücke formal typgleich zu machen).

`volatile` *(Schlüsselwort* C *und* C++*)*
Bei einem als `volatile` deklarierten Objekt rechnet der Compiler damit, daß sich der Wert ändern kann, ohne daß der vom Compiler generierte Code darauf Einfluß hat (beispielsweise durch asynchron auftretende Signale oder parallel laufende *threads*).

Dadurch darf ein Compiler keine Optimierungen durchführen, die sich auf einen unveränderlichen Wert des Objekts verlassen.

Siehe [KW:0 ist false].

`wchar_t` *(Schlüsselwort* C++*)*
In einem Element vom Typ `wchar_t` kann man ein Zeichen speichern; im Gegensatz zu einem normalen `char` lassen sich aber mehr als die gewohnten 256 Zeichen darstellen.

`while` *(Schlüsselwort* C *und* C++*)*
Eine abweisende Schleife wird mit `while` gebaut.

Siehe [KW:0 ist false].

`xor` *(Schlüsselwort* C++*)*
Mit dem Schlüsselwort `xor` hat man eine alternative Schreibweise für den *bitweisen Exklusiv-ODER-Operator* `^`.

Folgende Zeilen sind also gleichwertig:

```
i = j ^ 0x000f;
i = j xor 0x000f;
```

Siehe [KW:0 ist false].

`xor_eq` *(Schlüsselwort* C++*)*
`xor_eq` ist eine alternative Schreibweise für `^=`, also für das *bitweise Exklusiv-ODER* mit gleichzeitiger Zuweisung.

Siehe [KW:0 ist false].

Literaturverzeichnis

[Strou I] Bjarne Stroustrup:
Die C++ Programmiersprache
1. Auflage
Addison-Wesley, 1985

[Strou II] Bjarne Stroustrup:
Die C++ Programmiersprache
2. Auflage
Addison-Wesley, 1992

[Strou IV] Bjarne Stroustrup:
Die C++ Programmiersprache
4. Auflage
Addison-Wesley, 2000

[Cline: C++ FAQs] Marshall Cline, Greg Lomow, Mike Girou:
C++ FAQs
2nd Edition
Addison-Wesley, 1999
Eine Sammlung vieler kleinerer Aspekte zu C++, CORBA und COM

[Josuttis: Stdbib.] Nicolai Josuttis::
Die C++-Standardbibliothek
Eine detaillierte Einführung in die vollständige ANSI/ISO-Schnittstelle
1. Auflage
ISBN 3-8273-1023-7
Addison-Wesley, 1996

[Josuttis: Stdlib.99] Nicolai M. Josuttis:
The C++ Standard Library
A Tutorial and Reference

Addison-Wesley 1999
ISBN 0201379260

[RRZN] *Die Programmiersprache C++ für C-Programmierer*
RRZN-Klassifikationsschlüssel SPR.C 2
3. Auflage
Revision B Regionales Rechenzentrum für Niedersachsen/Universität Hannover
Juni 1993

[KW:0 ist false] Klaus Wachtler:
0 ist false, alles andere true
Selbstverlag Klaus Wachtler
Breidingstraße 17
29614 Soltau
http://www.wachtler.de
August 2001

[KW:Tools] Klaus Wachtler:
A Fool with a Tool is still a Fool
Überblick über einige wichtige Programmiertools
Selbstverlag Klaus Wachtler
Breidingstraße 17
29614 Soltau
http://www.wachtler.de
August 2001

[ct 9/94 Hartinger] Roland Hartinger:
Syntax-Puzzle
Die jüngsten C++-Erweiterungen
c't 9/1994, S. 184
Korrektur in c't 10/1994, S. 12
Kurzabriß der neueren Sprachelemente (außer templates): Ausnahmebehandlung, RTTI, Namensräume (name spaces)

[ct 9/94 Hueskes] Ralf Hüskes:
Geduldsspiel
Der kommende C++-Standard
c't 9/1994, S. 180
Zur Standardisierung von C++, Interview mit Bjarne Stroustrup

[ct 11/94 Langner] Angelika Langner:
Instantware
Die neuen Standardklassen für C++

c't 11/1994, S. 272

[Eckel: TIC++ 1] Bruce Eckel:
Thinking in C++
Volume 1
http://www.mindview.net/Books/DownloadSites

[Eckel: TIC++ 2] Bruce Eckel:
Thinking in C++
Volume 2
http://www.mindview.net/Books/DownloadSites

Index