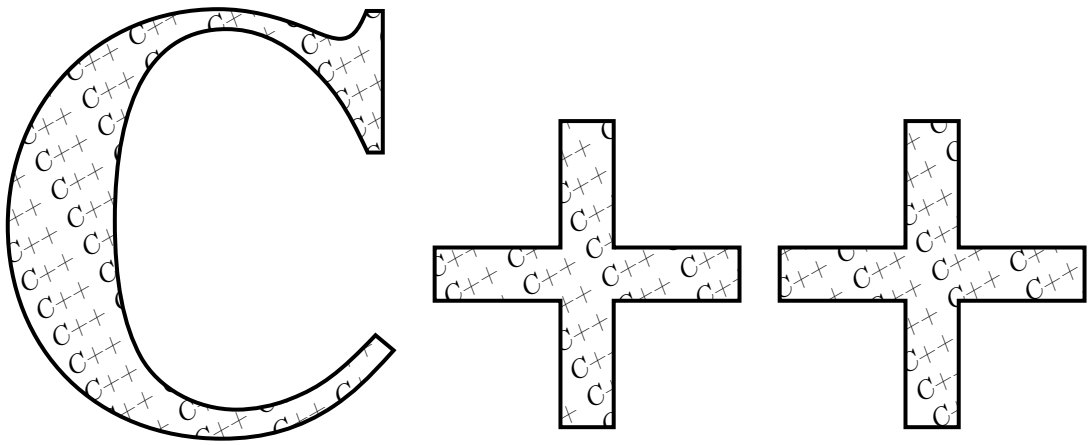
Programmierung mit C++ 2

Prof. Dr.-Ing. Fritz Mehner

Fachhochschule Südwestfalen
Fachbereich Informatik und Naturwissenschaften

Version 1.5

Stand 23. Juni 2013



Inhaltsverzeichnis

Vorwort

Die Sprache *C++*

Die Programmiersprache ***C++*** wurde in den frühen 1980-er Jahren von Bjarne Stroustrup in den Bell Research Laboratories als objektorientierter Nachfolger für die Programmiersprache ***C*** entwickelt. Nach zahlreichen Änderungen im Laufe des Entwicklungsprozesses wurde die Sprachdefinition 1998 mit dem Standard ISO-/IEC 14882 abgeschlossen. Die Sprache ist unter anderem in dem Buch [Str00] beschrieben (Lehrbuch und Referenz). Die dritte und letzte Ausgabe des ***C++***-Standards wurde 2011 veröfffentlich und bringt wichtige Erweiterungen. Diese Version der Sprache wird kurz als ***C++***11 bezeichnet.

C++ ist eine hybride Sprache: die objektorientierten Eigenschaften sind über den ***C***-Unterbau gelegt und können, müssen jedoch größtenteils nicht, verwendet werden. Natürlich sind die objektorientierten Eigenschaften gerade diejenigen, die den Einsatz dieser Sprache interessant machen! Der ***C***-Unterbau ist weitgehend rückwärtskompatibel zu ***C***.

C++ ist nicht mehr, wie ***C***, in erster Linie Systemimplementierungssprache, sondern zielt auf den Einsatz in großen Software-Projekten. Hierbei sind Korrektheit, Robustheit, Erweiterbarkeit und Wiederverwendbarkeit wesentliche Ziele, die durch den Einsatz der objektorientierten Programmiermethode erreicht werden sollen.
Kapitel 2 führt in die Grundbegriffe der objektorientierten Programmierung ein. In den nachfolgenden Kapiteln werden die wichtigsten Techniken der objektorientierten Programmierung behandelt (Operatorüberladung, Vererbung, Fehlerbehandlung, Einsatz von Templates).

Ein ***C++***-Kurs vom vorliegenden Umfang kann und will nur Grundkenntnisse vermitteln. Eine umfassende Behandlung der Sprache ist wegen ihres Umfanges, der zahlreichen Besonderheiten und der Fülle der möglichen Programmiertechniken hier nicht möglich. Die erworbenen Kenntnisse sollen aber geeignet sein, den Einstieg in einfache Programmierprojekte zu erlauben und das Wissen durch Selbststudium, Projektarbeit oder weitere Kurse zu erweitern und zu vertiefen.

Eigenständiges Üben durch Nachvollziehen des Lehrstoffes und durch Lösen von Aufgaben ist, wie beim Erlernen einer jeden Programmiersprache, auch hier unabdingbar.
Eine wesentliche Quelle für Anregungen und neue Einsichten ist das Studium fremder Programmtexte. Es ist außerdem ausgezeichnet dazu geeignet, die eigenen Fähigkeiten beim passiven Codeverständnis zu überprüfen. Studienmaterial findet sich reichlich in Programmierbüchern und im Internet.

Literatur

Zur Vertiefung des Stoffes und zur Begleitung des Praktikums sind in der Regel Bücher und Nachschlagewerke erwünscht und erforderlich. Im Literaturverzeichnis (Seite 85) sind einige Empfehlungen aufgeführt. Für den Einstieg sind [IH04] und [KM03] geeignet. Das Buch „The C++ Programming Language“ des Spracherfinders Bjarne Stroustrup ist das Referenzwerk

und eher bei fortgeschrittenen Kenntnissen geeignet ([Str00]). Wer einen Style Guide sucht möchte, greift zu [MBG04]. Zur Vertiefung der Kenntnisse ist [Mey05] zu empfehlen.

Dateinamen

Für *C++*-Dateien wurde durchgängig die Dateinamenerweiterung `.cc` gewählt, Header-Dateien erhalten die Erweiterung `.hh`. Damit können diese Dateien vom Leser, aber insbesondere auch von Werkzeugen (Editoren, IDEs, Compiler) als *C++*-Dateien erkannt und behandelt werden. Erweiterungen, wie etwa `.cpp`, `.cxx`, `.hpp`, `.hxx` sind ebenfalls gebräuchlich. Die Erweiterungen `.C` und `.H` sollten vermieden werden, da manche Plattformen nicht zwischen Groß- und Kleinbuchstaben in Dateinamen unterscheiden.

Zur Darstellung

Programmcode, Programmausgaben, Programm- und Dateinamen, Schlüsselwörter von Programmiersprachen und Menüeinträge erscheinen in `Schreibmaschinenschrift mit fester Zeichenbreite`.

`switch`	Schlüsselwort (halbfett)
`x = 2.0-y;`	Anweisung (normal)
`// *** Kommentar ***`	Kommentar (kursiv)

Die Nachkommastellen reeller Zahlen werden, entsprechend der Darstellung in den meisten Programmiersprachen, durch einen *Dezimalpunkt* abgetrennt, also zum Beispiel `x = -7.11`.

1. Nicht-objektorientierte Erweiterungen von *C++*

1.1. Namensbereiche

In großen Projekten kann der Zwang, gleiche Benennungen für global sichtbare Funktionen, Variablen und so weiter zu vermeiden, zu einem erheblichen Aufwand bei der Vergabe und beim Abgleich von Namen führen. Als Abhilfe und als weiteres Strukturierungsmittel für global sichtbare Namen wurden in ***C++*** Namensräume eingeführt. In unterschiedlichen Namensräumen dürfen nun gleiche Bezeichnungen auftreten, ohne daß (bei richtiger Handhabung) Namenskonflikte auftreten.
Die Gliederung großer Projekte wird im Rahmen dieser Einführung nicht weiter betrachtet. Es muß jedoch bei allen ***C++***-Programmen beachtet werden, daß zu Beginn der Datei die Anweisung

```
using namespace std;
```

enthalten ist. Diese Anweisung ordnet alle nachfolgenden Angaben dem Namensraum `std` zu. Zu diesem Standard-Namensraum gehören insbesondere alle Bibliotheksfunktionen.

1.2. Der Datentyp `bool`

In ***C*** fehlt leider ein Datentyp zur Darstellung logischer oder boolescher Größen. In ***C++*** ist hierfür der Datentyp **`bool`** mit den zwei Konstanten **`true`** und **`false`** vorhanden. Der Compiler führt gegebenenfalls auch Typkonvertierungen von anderen Zahlentypen in den Zieltyp **`bool`** aus.

```
bool  b1, b2;
b1  = true;     // Zuweisung einer Konstanten
b2  = false;    // Zuweisung einer Konstanten
  ...
b1  = 5;        // entspricht true
b2  = 0;        // entspricht false
```

Boolesche Werte werden intern als Ganzzahlen abgelegt: 0 für `false` und 1 für `true`. Nach der obigen Zuweisung `b1 = 5` enthält `b1` also den Wert 1.

1.3. Vereinbarungen

In ***C++*** ist es nicht mehr zwingend erforderlich, alle Vereinbarungen an den Blockanfang zu stellen. Ein Block kann also auch mit einer ausführbaren Anweisung beginnen. Die Erhöhung der Freizügigkeit bei der Vereinbarung ist vorteilhaft und kann dazu verwendet werden, Größen in der Nähe ihrer ersten Verwendung zu vereinbaren. Vereinbarungen, die von Bedingungen

abhängig sind, werden unter Umständen gar nicht ausgeführt. Das kann die Programmausführung beschleunigen. Die durchgängige Vereinbarung am Blockanfang hat hingegen den Vorteil, daß der Leser des Abschnittes sofort eine Übersicht über alle verwendeten Größen erhält.
In besonderen Fällen ist der eingeschränkte Gültigkeitsbereich zu beachten. Eine Variable, die im **if**-Zweig einer Verzweigung vereinbart ist, ist im **else**-Zweig nicht bekannt und umgekehrt ebenso. Im Bedarfsfall muß eine entsprechende Variable zweimal vereinbart werden. Das trifft zum Beispiel für die Variable `i` der folgenden Schleifensteuerungen zu:

```
if( auf_ab )
  for( int i=0; i<n; i++ )
    a[i]  = i;
else
  for( int i=0; i<n; i++ )
    a[i]  = n-1-i;
```

1.4. Konstante Objekte

Mit dem Schlüsselwort **const** werden Objekte ausgezeichnet, deren Wert nach der Anfangswertzuweisung in der Vereinbarung nicht mehr veränderbar sein soll. In den folgenden Zeilen wird die Variable `Nmax` zur globalen Größenfestlegung von Feldern verwendet, die Größe `constFaktor` dient im Hauptprogramm als globale Konstante:

```
const int Nmax = 1000;                          // max. Puffergröße
  ...
int main ( )
{
  double        puffer1[Nmax];                  // 1. Puffer
  const double  constFaktor = 2.3*tan(1.5);     // konstanten Faktor berechnen
  ...
}
```

Die globale **const**-Größe ist hier an die Stelle einer entsprechenden `define`-Anweisung getreten. Der Vorteil einer **const**-Größe ist, daß sie eine typgebundene Variable darstellt, deren Vereinbarung und Gebrauch der Typprüfung durch den Compiler unterliegt. Eine `define`-Anweisung stellt hingegen nur eine Textersetzung zur Verfügung. In *C++* sollte daher weitestgehend auf `define`-Größen verzichtet werden. Stattdessen verwendet man typsichere **const**-Größen.

1.5. Die Operatoren `new` und `delete`

Zur dynamischen Belegung und Freigabe von Speicherplatz werden in *C* die Funktionen `calloc`, `malloc` und `free` verwendet (header-Datei `stdio.h`). In *C++* stehen hierfür zwei Operatorpaare zur Verfügung:

Operator	Bedeutung
`new` Datentyp	Speicherplatz für ein Objekt des angegebenen Datentyps beschaffen; Zeiger zurückgeben
`delete` Zeiger	Speicherplatz eines Objektes freigeben
`new` Datentyp`[n]`	Speicherplatz für n Objekte des angegebenen Datentyps beschaffen (Feld); Zeiger zurückgeben
`delete[]` Zeiger	Speicherplatz eines Feldes freigeben

Diese Operatoren sollten in ***C++***-Programmen ausschließlich zur dynamischen Speicherbelegung verwendet werden. Der erste Operator **new** *Datentyp* belegt Speicher für ein Objekt vom Typ *Datentyp* und liefern als Rückgabewert einen Zeiger vom Typ (*Datentyp*∗). Dem Ausdruck kann in runden Klammern ein Anfangswert für das gerade eingerichtete Objekt folgen. Fehlt der Anfangswert, dann ist der Wert des Objektes nicht definiert. Belegter Speicher wird mit Hilfe des Operators **delete** freigegeben.

```
double *x1 = new double;              // der Wert *x1 ist nicht definiert
double *x2 = new double(47.11);       // der Wert *x2 ist 47.11
  ...
delete x1;                            // Speicher von x1 freigegeben
delete x2;                            // Speicher von x2 freigegeben
```

Zur Belegung und Freigabe von Speicherplatz für Felder wird das zweite Operatorpaar verwendet. Hier folgt bei **new** der Typangabe die Anzahl der zu belegenden Plätze in eckigen Klammern:

```
int *feld = new int[1024];            // Feld belegen; 1024 Elemente
  ...
delete[] feld;                        // Feld freigegeben
```

Die Initialisierung eines Feldes mit Null gleich bei der Beschaffung ist möglich:

```
int *feld2 = new int[100]();          // Feld beschaffen; mit 0 initialisieren
```

Die runden Klammern müssen, im Gegensatz zu einfachen Datentypen, leer bleiben.

Für die Freigabe eines Feldes wird ausschließlich der zugehörige Operator **delete[]** verwendet. Die Verwendung des Operators ohne die eckigen Klammern kann bei der Freigabe von Feldern zu einem Speicherleck[1] führen! Die fehlerfreie Verwaltung des Speichers und die richtige Handhabung der Zeiger bleibt wie in ***C*** vollständig dem Programmierer überlassen. Zugriffe auf nicht vorhandenen Speicher, Freigabe von bereits freigegebenem Speicher, Überschreiben von Zeigern und so weiter haben unvorhersehbare Folgen, bis hin zum Programmabsturz!

Die Speicherbeschaffung kann natürlich auch fehlschlagen. Das Programm wirft in diesem Fall ein sogenanntes Ausnahme (engl. exception) aus und bricht mit einer Fehlermeldung ab. Die richtige Fehlererkennung und -behandlung erfordert ein weiteres ***C++***-Konzept. Dessen Behandlung wird in Kapitel 5 erfolgen.

1.6. Referenzen

Die Sprache ***C++*** führt zusätzlich zu den bereits aus ***C*** bekannten Zeigern einen eingeschränkten Zeigertyp, die sogenannte Referenz, ein.

> Eine Referenz ist ein *konstanter Zeiger* über den *ohne Inhaltsoperator* auf den Inhalt zugegriffen werden kann.

Da der Zeigerwert nach der Vereinbarung nicht mehr geändert werden kann, muß eine Referenz bei der Vereinbarung einen Anfangswert erhalten. Eine Referenz ist damit ein zweiter Name (Alias) für ein bereits vorhandenes Objekt.

[1] Speicher, der weder freigegeben, noch verwendet werden kann. Werden in einem langlaufenden Programm wiederholt Speicherlecks erzeugt, kann das zur vollständigen Aufzehrung des Hauptspeichers führen.

Referenzen werden häufig als Parameter und als Rückgabewerte von Funktionen und überladenen Operatoren verwendet. Da der Inhaltszugriff ohne Dereferenzierung stattfinden kann, entfällt die sonst übliche, etwas lästige und schlecht lesbare Zeigerschreibweise.

Die Vereinbarung einer Referenzen geschieht durch das Zeichen `&` vor dem Variablennamen (Zeichen `*` bei Zeigern). Die folgenden Zeilen zeigen zwei Beispiele:

```
int  a = 0;      // einfache int-Variable
int &b = a;      // Referenz auf a
int &c = b;      // Referenz auf b

cout << a;       // der Wert 0 wird ausgeben
c++;             // a wird erhöht
cout << a;       // der Wert 1 wird ausgeben
```

Das nächste Beispiel zeigt die Verwendung von Referenzen bei Funktionen. Die Funktion `swap` tauscht die Werte ihrer Argumente und muß dazu die Argumente mittels Adreßübergabe erhalten (Zeiger oder Referenzen).

```
void  swap ( int &a, int &b )
{
  int h = a;         // Zugriffe ohne Inhaltsoperator
      a = b;
      b = h;
}

int main ( )
{
  int x = 7, y = 9;
  ...
  swap( x, y );      // Aufruf ohne Adreßoperator
  ...
}
```

Das folgende Beispiel zeigt die Verwendung einer Referenz als Rückgabewert. Die Funktion `quad` quadriert den Wert ihres Parameter in der aufrufenden Umgebung und gibt eine Referenz auf diesen Parameter zurück.

```
double & quad ( double &a )
{
  a *= a;        // Wert des Parameters quadrieren
  return a;      // Referenz auf den Parameter zurückgeben
}
```

Da Argument und Rückgabewert eine Referenz gleichen Typs sind, kann die Funktion geschachtelt aufgerufen werden:

```
double  a = 3, b;
  ...
b = quad( quad( a ));   // a = b = 81
  ...
```

Wegen des Referenzzugriffs (Zeigerzugriff) haben beide Größen anschließend den Wert 81 ($= 3^{2^2}$).

Die Vorteile bei der Verwendung von Referenzen:

- Einige Schreibweisen sind einfacher als bei der Verwendung von Zeigern.

- Referenzen enthalten immer eine gültige Adresse und sind deshalb in der Verwendung sicherer als Zeiger. Zeiger können nicht initialisiert sein oder durch fehlerhafte Adreßrechnungen oder Konvertierungen ungültig geworden sein.

Die Vorteile von Zeigern und damit die Nachteile von Referenzen:

- Referenzen sind im Programmtext oft nicht ohne weiteres von einfachen Variablen zu unterscheiden. Zeiger sind meist sofort als solche zu erkennen.
- Zeiger sind Adreßvariablen, deren Inhalt geändert werden kann (zum Beispiel durch Adreßrechnung). Damit erlauben Zeiger Dinge, die mit Referenzen nicht möglich sind.

1.7. Die iostream-Bibliothek

Manipulator	**Bedeutung**	Std.	beeinflußt
`dec`	dezimale Ausgabe	√	ganze Zahlen
`hex`	hexadezimale Ausgabe		
`oct`	oktale Ausgabe		
`setbase(int n)`	Zahlenbasis setzen (8, 10 oder 16)	10	
`showbase`	`0` vor Oktalzahlen, `0x` vor Hexadez.zahlen		
`noshowbase`		√	
`showpos`	`+` vor positive Zahlen		
`noshowpos`		√	
`scientific`	Darstellung mit Exponent (z.B. `0.321e6`)		reelle Zahlen
`fixed`	Darstellung ohne Exponent (z.B. `123.77`)		
`setprecision(int n)`	Gesamtstellenzahl `n` bei reellen Größen. Zusammen mit `fixed` oder `scientific`: Anzahl der Nachkommastellen	6	
`showpoint`	Dezimalpunkt bei reellen Größen mit ganzzahligem Wert ausgeben		
`noshowpoint`		√	
`uppercase`	`E` anstatt `e`, `X` anstatt `x`		reelle Zahlen, hex. Zahlen
`nouppercase`		√	
`right`	rechtsbündige Ausgabe	√	alle
`left`	linksbündige Ausgabe		
`setw(int n)`	`n` ist die minimale Zeichenanzahl für die nachfolgende Ausgabegröße	0	alle außer `char`
`setfill(int c)`	Füllzeichen `c` anstelle des Leerzeichens verwenden	`' '`	
`endl`	Zeilenvorschub ausgeben; `flush` anwenden		
`flush`	Ausgabepuffer leeren		

Tabelle 1.1.: Die wichtigsten Ausgabemanipulatoren (erfordern die header-Datei `iomanip`)

Für ***C++*** steht ein vollkommen eigenes Ein-/Ausgabesystem zur Verfügung. Zwar können die aus ***C*** bekannten Funktionen `printf`, `scanf` und so weiter weiterhin verwendet werden, es

ist jedoch geraten, in einem C++-Programm auschließlich die neuen Möglichkeiten zu verwenden. Das C++-eigene Ein-/Ausgabesystem ist typsicher, erweiterbar und die Ein-/Ausgabeanweisungen sind bis zu einem gewissen Grad selbstdokumentierend. Die erschöpfende Behandlung dieses Ein-/Ausgabesystems würde den Rahmen dieses Kurses sprengen. Hier sollen nur die Grundzüge erläutert werden. Die Ein-/Ausgabe stützt sich auf Klassen ab, die durch die folgende Anweisung zugänglich gemacht werden:

```
#include <iostream>
```

Für die Standardausgabe steht das Objekt `cout` zur Verfügung. Die Ausgabe einer Größe wird mit Hilfe des Ausgabeoperators `<<` dargestellt (überladener Linksschiebeoperator). In der einfachsten Verwendungsform erfolgt die Ausgabe einer **`double`**-Größe `x` wie folgt:

```
double  x = 3.45;
  ...
cout << x;
```

Die Angabe eines Typkennzeichens (wie etwa `"%f"` bei der Verwendung von `printf`) ist weder erforderlich noch in dieser Form möglich.
Für die Standardeingabe steht das Objekt `cin` zur Verfügung. Die Eingabe einer Größe wird mit Hilfe des Eingabeoperators `>>` dargestellt (überladener Rechtsschiebeoperator). In der einfachsten Verwendungsform erfolgt die Eingabe einer **`double`**-Größe `y` wie folgt:

```
double  y;
  ...
cin >> y;
```

Ein Typkennzeichen ist ebensowenig erforderlich wie die Verwendung des Adreßoperators. Ein- und Ausgaben von Größen unterschiedlichen Typs können in einer Anweisung zusammengefaßt werden. Die Ausgabeanweisung in der Schleife

```
for ( i=0; i<n; i+=1 )
  cout << "\n  a[" << i << "] = " << a[i];
```

erzeugt Ausgaben der folgenden Art:

```
a[0] = 234
a[1] = 237
a[2] = 228
  ...
```

Die auszugebenden Größen (Zeichenketten und Variablenwerte) sind jeweils durch einen Ausgabeoperator getrennt. Die Abarbeitung erfolgt offensichtlich von links nach rechts.

Zur Formatierung der Ausgabe stehen sogenannte Manipulatoren zur Verfügung. Die wichtigsten sind in der Tabelle 1.1 zusammengefaßt.

Das Programm in Liste 1.1 zeigt die Verwendung einiger Ausgabemanipulatoren. Es erzeugt die folgende Ausgabezeilen:

```
2004......
......2004
eine Oktalzahl   :     0400
eine Hexadez.zahl:    0x100
8/7 =     1.1429
```

Liste 1.1: Verwendung von Ausgabemanipulatoren (1.07-1.ea.cc)

```
#include  <iostream>
#include  <iomanip>
#include  <ios>

using namespace std;

  int
main ( int argc, char *argv[] )
{
  int    jahr  = 2004;
  int    zahl  =  256;
  double bruch = 1/7.0;

  cout << left  << setfill('.') << setw(10) << jahr << endl;
  cout << right << setfill('.') << setw(10) << jahr << endl;
  cout << setfill(' ');

  cout << "eine Oktalzahl   : " << oct << setw(8) << showbase << zahl << endl;
  cout << "eine Hexadez.zahl: " << hex << setw(8) << showbase << zahl << endl;

  cout << "1/7 = " << setw(10) << fixed << setprecision(4) << bruch << endl;

  return 0;
}       // ----------  end of function main  ----------
```

Liste 1.2: Dateieingabe und Dateiausgabe (1.07-2.ea.cc)

```
#include  <fstream>                              // file stream objects
#include  <iostream>                             // I/O  stream objects
#include  <string>
#include  <cstdlib>

using namespace std;

  int
main ( int argc, char *argv[] )
{
  int wert;

  string    ifs_file_name = "1.07-2.ea-in.dat"; // input file name
  ifstream  ifs;                                // create ifstream object
  ifs.open( ifs_file_name.c_str() );            // open ifstream
  if (!ifs) {
    cerr << "\nERROR : failed to open input file " << ifs_file_name << endl;
    exit (1);
  }

  string    ofs_file_name = "1.07-2.ea-out.dat";// output file name
  ofstream  ofs;                                // create ofstream object
  ofs.open( ofs_file_name.c_str() );            // open ofstream
  if (!ofs) {
    cerr << "\nERROR : failed to open output file " << ofs_file_name << endl;
    exit (2);
  }

```

```
29    //---------------------------------------------------------------------
30    //  Eingabedatei lesen, Ausgabedatei schreiben
31    //---------------------------------------------------------------------
32    while( ifs >> wert ) {                          // unbekannte Dateilänge
33      //
34      // ggf. wert bearbeiten
35      //
36      ofs << -wert << endl;                         // wert in die Ausgabedatei
37    }
38
39    ifs.close();                                    // close ifstream
40    ofs.close();                                    // close ofstream
41
42    return 0;
43  }       // ----------  end of function main  ----------
```

Für die Eingabe aus Dateien und die Ausgabe in Dateien stehen enbenfalls entsprechende Ein-/Ausgabeobjekte zur Verfügung. In Liste 1.2 wird deren Verwendung gezeigt. Aus der Eingabedatei `in.dat` unbekannter Länge werden ganze Zahlen gelesen, evtl. bearbeitet und in die Ausgabedatei `out.dat` geschrieben. Die Ein-/Ausgabeobjekte für Dateien werden durch die Anweisung

```
#include  <fstream>
```

zugänglich gemacht. Hier die Erläuterung einiger Zeilen:

Zeile 14 Eingabestrom-Objekt `ifs` erzeugen.

Zeile 15 Öffnen der Eingabedatei und Verbinden mit dem Eingabestrom-Objekt `ifs`.

Zeile 16 Überprüfen, ob die Datei offen ist. Im Fehlerfall Ausgabe einer Meldung und Programmabbruch.

Zeile 32 Einlesen der Daten. Das Eingabstrom-Objekt steuert über seinen Wert gleichzeitig die `while`-Schleife. Solange der Wert des `ifs`-Objektes ungleich Null ist, wird der gerade erfolgreich gelesene Wert `wert` bearbeitet, ausgegeben und die Schleife erneut ausgeführt.

Zeilen 39,40 Dateien schließen.

Für den Umgang mit Dateien stehen weitere Steuerungsmöglichkeiten zur Verfügung. Hier muß jedoch auf entsprechende Bücher verwiesen werden.

1.8. Überladen von Funktionsnamen

In *C++* sind, im Gegensatz zu *C*, mehrere Funktionen mit demselben Funktionsnamen, aber mit unterschiedlichen Parameterlisten erlaubt. Man spricht dann von einer *Überladung des Funktionsnamens*. Der Compiler ordnet einem Funktionsaufruf anhand der aktuellen Parameterliste die richtige aufzurufende Funktion zu. Eine entsprechende Funktion muß natürlich vorhanden sein.
Das folgende Beispiel zeigt die Implementierung zweier Funktionen mit dem Namen `swap`. Beide vertauschen die Werte ihrer Aufrufparameter, die erste für **`int`**-Größen, die zweite für **`double`**-Größen. Die Funktionsdefinitionen verwenden Referenzen (Abschnitt 1.6).

```
void  swap ( int &a, int &b ) {       // int-Version
  int    h = a;
```

```
        a = b;
        b = h;
}

void  swap ( double &a, double &b ) { // double-Version
  double h = a;
        a = b;
        b = h;
}
  ...
int     a = 5,    b = 7;
double  x = 9.1,  y = 7.8;
  ...
swap ( a, b );                        // int-Version
swap ( x, y );                        // double-Version
swap ( x, a );                        // Fehler, keine passende Funktion!
```

Durch die Überladung von Funktionsnamen kann eine ausufernde Namensvielfalt vermieden werden. Unabhängig vom Datentyp können nun alle Funktionen die einen Wertetausch durchführen `swap` heißen. Die einzelnen Funktionen müssen natürlich trotzdem alle geschrieben werden.

Ein Blick auf die beiden `swap`-Funktionen zeigt, daß die Ähnlichkeiten im Code hoch sind. Sie unterscheiden sich nur in dem jeweils dreimal auftretenden Schlüsselwort für den jeweiligen Datentyp. In Kapitel 6 wird gezeigt, wie sich die Implementierung derartiger Funktionen durch Schablonen (sogenannte templates) weiter vereinfachen läßt.

Die Funktionsüberladung ist ein wichtiges Konzept, von dem bei der objektorientierten Programmierung an verschiedenen Stellen Gebrauch gemacht wird (Überladung von Methoden einer Klasse, Vererbung, Überladung von Operatoren).

1.9. Überladen von Operatoren

In ***C++*** lassen sich auch die meisten Operatoren überladen (die wichtigsten sind in Tabelle A.2 angegeben). Durch die Überladung erhalten die bereits vorhandenen Operatoren eine *zusätzliche* Bedeutung. Neue, in ***C++*** nicht vorhandene Operatoren können nicht definiert werden.

In ***C++*** ist es möglich, die Operatoren über einen vorgegebenen Funktionsnamen anzusprechen. Im Falle des Additionsoperators heißt der Funktionsname **`operator +`**. Für die anderen Operatoren sind die Funktionsnamen entsprechend: dem Schlüsselwort **`operator`** folgt das jeweilige Operatorsymbol. Diese Name stehen damit zur Funktionsüberladung zur Verfügung.
Liste 1.3 zeigt die Verwendung. Die Vektoren der Länge drei werden als Strukturen dargestellt und können deshalb als Funktionsparameter und als Rückgabewert verwendet werden (Wertübergabe).
Der Additionsoperator verknüpft zwei Operanden zu einem neuen Ergebnisvektor. Die Implementierung als Funktion übernimmt deshalb den linken Operanden einer Addition als ersten Parameter, den rechten als zweiten Parameter (Liste 1.3, Zeile 10). Das Additionsergebnis wird in der lokalen Struktur `hilf` berechnet. Diese ist zwar nach dem Funktionsaufruf verschwunden, ihr Wert wird aber per Wertübergabe zurückgegeben und steht deshalb in einem Rechenausdruck als Operand zur Verfügung oder kann an eine andere Struktur zugewiesen werden. Die beiden Funktionen für die links- und rechtsseitige Multiplikation mit einem Skalar

(**double**-Größe) sind nach der gleichen Idee angelegt. Die Überladung des Ausgabeoperators **operator** << ist ebenfalls möglich. Damit lassen sich Strukturen vom Typ Vek3 so ausgeben, wie das oben für die Standarddatentypen gezeigt wurde. Die Einzelheiten zur Überladung von Ein-/Ausgabeoperatoren werden in Abschnitt 3.4 behandelt.
Das Hauptprogramm zeigt zwei Verwendungsmöglichkeiten. In Zeile 46 ist eine Vektoraddition mit Hilfe des Pluszeichens dargestellt. Der Übersetzer hat erkannt, daß für Strukturen vom Typ Vek3 der Standardoperator + nicht definiert ist und sucht deshalb nach einer passenden überladenen Funktion. Das ist in diesem Fall die Funktion Vek3 **operator** + (Vek3 v1, Vek3 v2). Genau diese Funktion wird für die beiden aktuellen Parameter u und v aufgerufen.
Die Funktion kann auch mit ihrem Funktionsnamen aufgerufen werden. Das geschieht in Zeile 49. Hierbei ist lediglich zu erkennen, daß es sich um dieselbe überladene Funktion wie in Zeile 46 handelt. Ansonsten ist der direkte Aufruf ungebräuchlich. Die Überladung erlaubt ja gerade die Verwendung von Operatoren und soll damit die etwas umständlichen Funktionsaufrufe vermeiden. Die Operatorüberladung bietet weitere interessante Möglichkeiten und wird in Kapitel 3 noch einmal aufgegriffen.

Liste 1.3: Vektorarithmetik mit überladenen Operatoren (1.09-1.vektor-ohne-klasse.cc)

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

struct Vek3                                     // ***  Vektoren der Länge 3  *****
{
  double  x, y, z;
};

Vek3 operator + ( Vek3 v1, Vek3 v2 )            // ***  Vek3-Addition  ************
{
  Vek3  hilf = { v1.x + v2.x,
                 v1.y + v2.y,
                 v1.z + v2.z };
  return  hilf;
}

Vek3 operator * ( Vek3 v1, double s )           // ***  Vek3 * Skalar  ************
{
  Vek3  erg = { s*v1.x, s*v1.y, s*v1.z };
  return erg;
}

Vek3 operator * ( double s, Vek3 v1 )           // ***  Skalar * Vek3  ************
{
  return v1*s;
}

void print_vek ( Vek3 v )                       // ***  Vek3-Ausgabefunktionen  ***
{
  cout  << "[ " << v.x << " , " << v.y << " , " << v.z << " ]";
}

//------------------------------------------------------------------------
//  Hauptprogramm
//------------------------------------------------------------------------
int main ( int argc, char *argv[] )
{
  Vek3    u = { 2, 4, 5 };
  Vek3    v = { 1, 2, 1 };
  Vek3    w;

  w = u + v;                                    // Vektoraddition mittels Operator
  cout  << "\nw = ";  print_vek(w);

  w = operator + ( u, v);                       // Vektoraddition als Funktionsaufruf
  cout  << "\nw = ";  print_vek(w);

  w = 3.0*w*2.0;                                // Skalarmultiplikation links, rechts
  cout  << "\nw = 3.0*w*2.0 = ";  print_vek(w);

  return 0;
}   // ----------  end of function main  ----------
```

1.10. Die Klasse `string` der Standard Library

Die Handhabung von Zeichenketten in *C* und *C++* stützt sich auf den Basisdatentyp **`char`**. Die zugehörigen Funktionen der *C*-Bibliothek stehen in *C++* ebenfalls zur Verfügung. Es bleibt jedoch der Mangel, daß die Handhabung von Zeichenketten auf dieser unteren Ebene mühsam ist. Besonders beim Umgang mit Zeichenketten mit veränderlicher Längen wird dem Benutzer die Speicherverwaltung aufgebürdet (mit Hilfe der Funktionen `malloc` und `free`). In einem größeren Projekt greift man daher zu einer geeigneten Bibliothek.
Zu *C++* gehört eine umfangreiche Bibliothek. Sie enthält eine leistungsfähige Sammlung von Datentypen und Methoden, die gut aufeinander abgestimmt sind. Diese Bibliothek wird in Abschnitt 6.4 vorgestellt, da zu ihrem Verständnis zunächst weitere Sprachkonzepte eingeführt werden müssen. An dieser Stelle werden nur einige Möglichkeiten für den Umgang mit Zeichenketten gezeigt, damit die Verwendung in Beispielen und Übungsaufgaben möglich ist.

Anweisung, Funktion	**Bedeutung**
`a = b`	Wertzuweisung, `a` und `b` sind vom Typ `string`
`a = "aabbcc"`	Zuweisung einer *C*-Zeichenkette
`a.c_str()`	Anhängen einer binären Null (`'\0'`) und Rückgabe als *C*-Zeichenkette
`a[i]`	Zugriff auf das Zeichen an der Position `i`
`a + b`	Verkettung der Zeichenketten `a` und `b` zu einer neuen Zeichenkette
`a += b`	Zeichenkette `b` wird an Zeichenkette `a` angehängt
`a.find('x')`	Index des Zeichens `x` in `a` ermitteln
`a.length()`	Länge der Zeichenkette `a` ermitteln
`a.clear()`	Zeichenkette löschen; `a` enthält anschließend die leere Zeichenkette
`a.empty()`	feststellen, ob `a` die leere Zeichenkette enthält
`a.substr(i,l)`	Teilzeichenkette ab dem Index `i` zurückgeben (Länge `l`)
`a == b` `a != b`	zeichenweiser Vergleich

Tabelle 1.2.: Operationen mit Zeichenketten (Auswahl)

Die Bibliotheksteile, die verwendet werden sollen, müssen über `include`-Anweisungen zugänglich gemacht werden. Für die Zeichenketten-Funktionen erfolgt das durch

```
#include <string>
```

Damit stehen dann ohne weiteres der Datentyp `string` und eine ganze Reihe zugehöriger Methoden zur Verfügung. Auch die Ein-/Ausgabeoperatoren sind bereits überladen. Um die Speicherbeschaffung muß sich der Anwender nicht kümmern: die Zeichenketten sind *dynamisch*.

Das folgende Beispiel zeigt Vereinbarung, Anfangswertzuweisung, Wertzuweisung und eine einfache Ausgabe:

```
string  str1;                                  // enthält eine leere Zeichenkette
string  str2("eine Zeichenkette");             // Zeichenkette mit Anfangswert
string  str3  = ".";                           // Zuweisung einer C-Zeichenkette
string  str4;                                  // enthält eine leere Zeichenkette

str1  = "Dies ist ";                           // Zuweisung einer C-Zeichenkette
cout  << str1 << str2 << str3 << endl << endl;
```

Die Tabelle 1.2 zeigt eine Auswahl an Operatoren und Methoden. Der Aufruf von Methoden (zum Beispiel `length()`) erfolgt durch Anhängen des Aufrufs an das Zeichenketten-Objekt mit Hilfe des Punkt-Operators.

Liste 1.4: Textdatei zeilenweise einlesen und ausgeben (`1.10-3.read-lines.cc`)

```
// #####   HEADER FILE INCLUDES   #################################################
#include <fstream>
#include <iostream>
#include <string>
#include  <cstdlib>

using namespace std;

//----------------------------------------------------------------------
//  main
//----------------------------------------------------------------------
int
main (int argc, char *argv[])
{
  string  buffer;

  if (argc != 2) {                              // 2 command line arguments
    cout << "\n\n\tusage:" << argv[0] << " textfile\n";
    exit(1);
  }

  ifstream  in( argv[1] );                      // open ifstream
  if ( !in ) {
    cerr << "\nERROR : failed to open input file " << argv[1] << endl;
    exit(1);
  }

  //--------------------------------------------------------------------
  //  read lines, write lines
  //--------------------------------------------------------------------
  while ( getline( in, buffer, '\n' ) ) {
    cout  << buffer << endl;
  }

  return 0;
}       // ----------  end of function main  ----------
```

Liste 1.4 zeigt ein vollständiges Programm, dem als Aufrufparameter der Name einer Textdatei mitgegeben wird. Zunächst wird überprüft, ob ein Dateiname angegeben wurde, dann wird diese Datei geöffnet. Die offene Datei wird dem Eingabestrom `in` zugeordnet. Der Aufruf `getline( in, buffer, '\n')` liest vollständige Zeilen aus dem Eingabestrom und legt diese in der Zeichenkette `buffer` ab. Der Parameter `'\n'` gibt an, daß der Zeilenvorschub als Satzende verwendet werden soll. Die Schleife bricht ab, sobald das Dateiende erreicht wurde.

2. Klassen

2.1. Grundbegriffe

Zunächst sind einige neue Begriffe einzuführen.

Eine **Klasse** ist eine benutzerdefinierte Datenstruktur mit zwei Arten von Bestandteilen:

- **Daten.** Die Daten sind in der Regel *privat* und damit vor einem unmittelbaren Zugriff von außen vollständig geschützt.
- **Funktionen.** Die Funktionen sind in der Regel *öffentlich*. Sie bilden die Schnittstelle über die der Zugriff und die Änderung der Daten ausschließlich möglich ist.

Eine Klasse stellt also einen **Abstrakten Datentyp** (ADT, Datenstruktur mit Zugriffsfunktionen) zur Verfügung.

Die Verwendungs- und Eingriffsmöglichkeiten, die die Funktionen einer Klasse zur Verfügung stellen, werden als **Methoden** dieser Klasse bezeichnet.
Eine Klasse ist eine Erweiterung einer Struktur um Funktionskomponenten. Eine Klassendefinition ist, ähnlich wie eine Strukturdefinition, eine Schablone für das Anlegen eines tatsächlichen Datenverbundes im Speicher. Ein tatsächlicher vorhandener Datenverbund der einer gegebenen Klassendefinition entspricht, wird als Objekt dieser Klasse bezeichnet.

Ein **Objekt einer Klasse** ist ein Datenverbund, der einer gegebenen Klassendefinition entspricht.

- Jedes Objekt enthält seine eigenen Datenkomponenten.
- Die Funktionskomponenten sind für alle Objekte einer Klasse nur einmal vorhanden.

In einem Objekt der beschrieben Form können die Datenkomponenten in umfassender Form gekapselt und damit vollständig vor einem direkten Zugriff von außen geschützt werden. Man spricht von **Datenkapselung**.
Bei einer Struktur besteht dieser Schutz nicht. Auch bei einer Struktur besteht die Möglichkeit, entsprechende Funktionen zur Verfügung zu stellen, über die die Zugriffe auf die Datenkomponenten der Struktur erfolgen *sollen*. Da die Datenkomponenten einer Struktur jedoch immer öffentlich und damit nicht weiter geschützt sind, kann der Zugriff über die bereitgestellten Funktionen stets durch einen direkten Zugriff umgangen werden. Besonders bei großen Software-Projekten kann diese Vorgehensweise zu erheblichen Schwierigkeiten führen, da nicht mehr ohne größeren Aufwand erkennbar ist, ob und wo direkt auf Datenstrukturen

zugegriffen wird.
Ein wesentlicher Vorteil einer strengen Kapselung besteht darin, daß bei Änderungen die Anpassung der Daten- und Funktionskomponenten der Klasse genügen. Die Änderungen sind also lokal begrenzt. Im günstigsten Fall muß das Projekt anschließend neu übersetzt und gebunden werden, um die Änderungen im gesamten Programm wirksam werden zu lassen.

Ein wesentliches Merkmal der **objektorientierten Programmierung** (OOP) ist die Verwendung von Klassen als abstrakte Datentypen und der Aufbau eines Software-Systems auf der Grundlage solcher Klassen.
Wird die Möglichkeit der **Vererbung** genutzt, dann lassen sich aus bereits definierten Klassen (Basisdatentypen) weitere Klassen ableiten (Kapitel 4). Diese enthalten in der Regel zusätzliche Daten- und Funktionskomponenten und erweitern somit den Basisdatentyp. Die Bestandteile der Basisklasse können übernommen werden, sodaß eine Neuerstellung von ähnlichem oder gleichem Programmcode vermieden wird. Diese **Wiederverwendbarkeit** von Programmcode ist ein weiteres wesentliches Ziel der objektorientierten Programmierung.

Zu diesen Merkmalen und Eigenschaften kommen weitere hinzu. So ermöglichen objektorientierte Programmiersprachen Programmbestandteile, die auf Objekte unterschiedlicher Klassen zeigen können und sich dann jeweils anders verhalten beziehungsweise andere Eigenschaften besitzen. Dieses Merkmal nennt man **Polymorpie** (griech. polymorphos „vielgestaltig“). Derselbe Methodenaufruf kann also auf unterschiedliche Objekte angewendet werden. Der tatsächliche Objekttyp kann unter Umständen erst zur Laufzeit des Programmes erkennbar sein. Diese Eigenschaft, abhängig von dem zur Laufzeit (dynamisch) ermittelten Objekttyp die zugehörige Methode aufzurufen, wird als **dynamische Bindung** bezeichnet. Bei einem einfachen Funktionsaufruf ist dagegen die aufzurufende Funktion bereits bei der Übersetzung bekannt und der Aufrufcode kann beim Binden fest eingetragen werden. Hier spricht man von **statischer Bindung**.

Das Konzept der objektorientierten Programmierung ist nicht neu. Es wurde bereits in den 1970er-Jahren am Forschungsinstitut Xerox PARC in Palo Alto (Kalifornien) entwickelt. Auch die 1967 eingeführte Programmiersprache *Simula* weist wichtige Züge der Objektorientierung auf. Im Jahr 1981 entstand *Smalltalk*, die erste durchgängig objektorientierte Programmiersprache. Der Durchbruch für die OOP kam aber erst Anfang der 1990er-Jahre, im wesentlichen durch den zunehmenden Einsatz von ***C++***.

2.2. Klassendefinition

Die allgemeine Form einer Klassendefinition lautet:

```
class  Klassenname
{
 public:
                öffentliche Komponenten
 private:
                private Komponenten
};
```

- Die Bezeichnungen **`class`**, **`private`** und **`public`** sind Schlüsselwörter.

- Auf **private Komponenten** (in der Regel die Datenkomponenten) können nur Komponentenfunktionen der eigenen Klasse zugreifen. Öffentliche Datenkomponenten sind möglich, aber in den wenigsten Fällen sinnvoll.
- Auf **öffentliche Komponenten** (in der Regel die Komponentenfunktionen) kann von außen zugegriffen werden. Über die öffentliche Funktionsschnittstellen wird auf die privaten Daten zugegriffen. Private Funktionen sind ebenfalls möglich. Diese können nur von anderen Funktionskomponenten, aber nicht von außen aufgerufen werden.
- Die Reihenfolge der Bereichsmarken **`private`** und **`public`** ist beliebig. Die Marken können auch mehrmals vorkommen.
- Die **Funktionskomponenten** werden entweder durch ihre Prototypen vertreten oder die Funktionen sind mit ihrem vollständigen Code aufgeführt. Die zweite Möglichkeit wird meist nur bei sehr kurzen Funktionen gewählt. Der Code dieser Funktionen ist dann *inline*, das heißt er wird an der Stelle jedes Aufrufes vollständig einkopiert !
- Eine Klasse entspricht in der einfachsten Form (ohne Methoden) einer Struktur. Bei einer Struktur sind alle Komponenten öffentlich, bei einer Klasse sind, wenn nichts anderes angegeben wird, zunächst alle Komponenten privat.

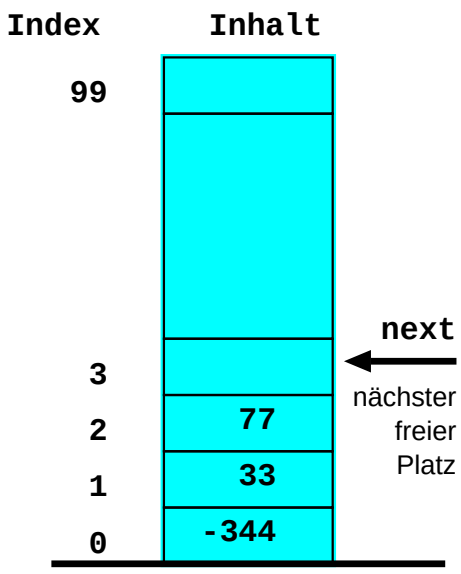


Abbildung 2.1.: Stapel fester Größe, realisiert als Feld

Als Beispiel für eine einfache Klasse soll der abstrakte Datentyp **Stapel** implementiert werden. Dazu sind die Daten- und die Funktionskomponenten festzulegen (siehe auch Abbildung 2.1).

Daten

- Feld mit 100 Plätzen zur Aufnahme der zu stapelnden Werte
- Stapelzeiger `next`; Index des nächsten freien Platzes
- Feldgröße `max`; dient der Feldgrenzenüberwachung

Methoden

- Initialisierung des Stapels (`next = 0`)
- Element auflegen (falls der Stapel nicht voll ist)
- Element herunternehmen (falls der Stapel nicht leer ist)
- Wert des obersten Elementes abfragen (falls der Stapel nicht leer ist)
- Anzahl der Elemente abfragen
- Größe des Stapels abfragen
- Feststellen, ob der Stapel leer ist
- Feststellen, ob der Stapel voll ist

Die Liste 2.1 zeigt die Definition und die Implementierung der Klasse `stack`, die den ADT Stapel mit den genannten Eigenschaften bereitstellt.
Die Zeilen 10-26 enthalten die Klassendefinition. Von den ersten vier Methoden sind nur die Prototypen angegeben. Die nächsten vier Methoden sind wegen ihrer Kürze mit ihrer Implementierung angegeben.

Ab Zeile 31 folgen die Implementierungen der vier ersten Methoden. Die Methode `push` hat folgende Implementierung:

```
void  Stack :: push (int datum) // ***** Element speichern *****
{
  assert( next < max );         // Überlauf ?
  value[next++] = datum;        // Wert speichern
}
```

Zwischen Rückgabetyp und Funktionsname ist der sogenannte *scope resolution operator* oder kurz **Scope-Operator** `Stack ::` gesetzt. Er besteht aus den Klassennamen gefolgt von zwei Doppelpunkten. Der Scope-Operator ordnet die Implementierung der Funktion `push` der Klasse `Stack` zu. Dies ist erforderlich, weil es weitere Funktionen mit gleichem Namen und gleicher Parameterliste geben könnte, die keiner Klasse oder anderen Klassen angehören.

Liste 2.1: Klasse `Stack`, Definition und Implementierung (`2.2-2.stack.cc`)

```
1  #include  <iostream>
2  #include  <cassert>
3  using namespace std;
4
5  const int n_max_stack = 1000;                    // maximale Stapelgröße
6
7  //--------------------------------------------------------------------
8  //  Definition der Klasse Stack
9  //--------------------------------------------------------------------
10 class   Stack
11 {
12   public:
13     void  init      ();                          // Initialisierung
14     void  push      (int datum);                 // Element hinzufügen
15     int   pop       ();                          // Element entfernen
16     int   top       ();                          // Wert des obersten Elementes
17     int   count     () { return next; }          // Anzahl der Elemente
18     int   maxel     () { return max; }           // Stapelgröße
19     bool  empty     () { return next==0; }       // Stapel leer
20     bool  not_empty () { return next!=0; }       // Stapel nicht leer
21
22   private:
23     int value [ n_max_stack ];                   // Feld zur Aufnahme der Datenelem.
24     int next;                                    // Stapelzeiger
25     int max;                                     // maximale Elementzahl
26 };
27
28 //--------------------------------------------------------------------
29 //  Implementierung der Klasse Stack
30 //--------------------------------------------------------------------
31 void  Stack :: init ()                           // INITIALISIERUNG
32 {
33   next = 0;                                      // Index zurücksetzen
34   max = n_max_stack;                             // maximale Elementzahl eintragen
35 }
36
37 void  Stack :: push (int datum)                  // ELEMENT SPEICHERN
38 {
39   assert( next < max );                          // Überlauf ?
40   value[next++] = datum;                         // Wert speichern
41 }
42
43 int   Stack :: pop ()                            // ELEMENT ENTFERNEN
44 {
45   assert( next > 0);                             // Unterlauf ?
46   return value[--next];                          // Index vermindern, Wert zurückgeben
47 }
48
49 int   Stack :: top ()                            // OBERSTES ELEMENT ZURÜCKGEBEN
50 {
51   assert( next > 0 );                            // Unterlauf ?
52   return value[next-1];                          // obersten Wert zurückgeben
53 }
```

Der Aufruf des Makros `assert`,

```
assert( next < max );          // Überlauf ?
```

stellt eine einfache Fehlerüberwachung zur Verfügung. Das Programm bricht mit einer Fehlermeldung ab, wenn die zu überprüfende Bedingung (hier: `next < max`) nicht erfüllt ist. Die Fehlermeldung enthält den Dateinamen der Programmquelle, den Funktionsnamen und die Zeilennummer des Aufrufs. Bei einem getesteten Programm sollten Unterläufe und Überläufe nicht mehr möglich sein; die `assert`-Aufrufe können dann entfernt werden. Weitere Möglichkeiten zur Fehlerbehandlung werden in Kapitel 5 vorgestellt.

Die Liste 2.2 zeigt die Verwendung des Stapels in einem Hauptprogramm.

Liste 2.2: Klasse `Stack`, Verwendung in einem Hauptprogramm (`2.2-2.stack.cc`)

```
1  //--------------------------------------------------------------------------
2  //  Hauptprogramm
3  //--------------------------------------------------------------------------
4    int
5  main ( int argc, char *argv[] )
6  {
7    int   i, n = 8;
8    Stack stapel;                                        // Objekt stapel anlegen
9
10   stapel.init();
11
12   for( i=0; i<=n; i++ )                                // Stapel belegen
13     stapel.push( i+10 );
14
15   cout  << "\n\toberstes Element : " << stapel.top() << "\n";
16
17   cout  << "\n\tStapel nicht leer : " << boolalpha << stapel.not_empty() << endl;
18
19   for( i=n; i>=0; i-- )                                // Stapel abbauen
20     cout  << "\n\t" << i << " : " << stapel.pop();
21
22   cout  << "\n\n";
23
24   return 0;
25 }       // ----------  end of function main  ----------
```

In Zeile 8 wird ein Objekt der Klasse `Stack` mit dem Namen `stapel` angelegt. In Zeile 10 wird das neue Objekt initialisiert. Der **Methodenaufruf** besteht aus dem Objektnamen an den mittels eines Punktes der Funktionsaufruf angehängt ist:

```
stapel.init();
```

Die Schreibweise erinnert an die Komponentenadressierung bei Strukturen.

2.3. Konstruktoren und Destruktoren

Das Beispiel des Stapels in den Listen 2.1 und 2.2 zeigt, daß die Stapel-Objekte zunächst nicht initialisiert sind. Der bestimmungsgemäße Gebrauch erfordert aber die Initialisierung der Variablen `next` mit 0 und die Zuweisung der Stapelgröße an die Variable `max`. Da beide Komponenten privat sind, kann die Initialisierung nur über eine Funktionskomponente erfolgen, in diesem Fall über `init()`.
Es ist leicht einzusehen, daß bei beliebigen Aufgabenstellungen die meisten Objekte nichttrivialer Klassen initialisiert werden müssen. Sollte innerhalb einer Initialisierung dynamisch Speicher beschafft werden (**`new`**, **`delete`**), dann ist vor dem Lebensende eines Objektes (am Ende des Gültigkeitsbereiches) dieser Speicher auch freizugeben. Es wird also zusätzlich zu einer Initialisierungsfunktion noch eine Beseitigunsfunktion erforderlich sein. Beide Funktionen müssen Methoden sein und die Aufrufe sind in den meisten Fällen offensichtlich zwingend. Da die Aufrufe trotzdem vom Programmierer vergessen werden könnten, sind in der Sprache ***C++*** zwei besondere Methoden für diese Aufgaben vorgesehen:

- der **Konstruktor** für die Initialisierung eines Objektes
- der **Destruktor** für Aufgaben bei der Beseitigung eines Objektes.

2.3.1. Der Konstruktor

Die allgemeine Form eines Konstruktors lautet:

```
Klassenname ( Parameterliste ) { ... }
```

Konstruktoren sind besondere Komponentenfunktionen, die den Namen der Klasse besitzen und keinen Rückgabetyp haben. Anzahl und Art der Parameter richten sich nach dem Zweck der Klasse, möglicherweise sind keine Parameter erforderlich. Mehrere Konstruktoren sind möglich. Diese müssen sich dann paarweise durch ihre Parameterlisten unterscheiden. Konstruktoren können beliebige Anweisungen enthalten.
Ein Konstruktor wird automatisch bei der Vereinbarung eines Objektes aufgerufen. Die Stapel-Klasse im letzten Abschnitt kann zum Beispiel wie folgt um einen Konstruktor erweitert werden:

```
class Stack
{
  public:   // ---------- öffentliche Komponenten -------

    Stack ();                 // Konstruktor
    ...
  private:    // -------- private Komponenten -----------
    ...
};

Stack :: Stack ()
{
  next = 0;                   // Index zurücksetzen
  max = n_max_stack;          // maximale Elementzahl eintragen
}
```

Wenn die Klasse `Stack` in dieser Weise ergänzt ist, dann entfallen alle Aufrufe der Methoden `init()` vor dem ersten Gebrauch eines Objektes (zum Beispiel in Liste 2.2). Die Methode

`init()` hat aber weiterhin ihre Berechtigung, da die Löschung eines Stapels im laufenden Programm von der Aufgabenstellung her erforderlich sein kann, um den Stapel anschließend anders zu nutzen.

2.3.2. Der Destruktor

Die Stapel-Klasse soll nun so geändert werden, daß Stapel beliebiger Größe möglich sind. Dazu wird in der Klassendefinition das Feld fester Größe durch eine Zeiger ersetzt. Bei der Vereinbarung eines neuen Stapels wird dem Konstruktor die gewünschte Größe übergeben. Der Konstruktor kann so mit **`new`** Speicher beschaffen und gegebenenfalls weitere Initialisierungen vornehmen.

Die Vereinbarung eines derartigen Stapel-Objektes innerhalb einer Funktion (zum Beispiel auch innerhalb von `main`) begrenzt die Lebensdauer auf diese Funktion. Deshalb wird hier eine weitere Methode sinnvoll, der **Destruktor**, die automatisch am Lebensende eines Objektes aufgerufen wird und die eine Bereinigung durchführt. Diese bestehen bei dem zu implementierenden Stapel natürlich in der Freigabe des im Konstruktor dynamisch belegten Speichers. Liste 2.3 zeigt für die Klasse `Stack` die neu hinzugekommen Teile.

Die allgemeine Form eines Destruktors lautet:

```
~Klassenname ( ) {  ...  }
```

Destruktoren sind besondere Komponentenfunktionen, die den Namen der Klasse mit einer vorangestellten Tilde (~) tragen und keinen Rückgabetyp haben. Destruktoren besitzen keine Parameter.
Es ist nicht die Aufgabe eines Destruktors, den von einem Objekt automatisch belegten Speicher freizugeben; der Destruktor macht lediglich die im Konstruktoraufruf und die gegebenenfalls danach durchgeführten Belegungen von dynamischem Speicher rückgängig.
So hat der Destruktor im Stapel-Beispiel nur die Aufgabe, das im Konstruktor dynamisch belegte Feld wieder freizugeben (die Speicherplätze für die Größen `*value`, `next` und `max` werden auch ohne Destruktor am Lebensende des Objektes beseitigt, sie besitzen die Speicherklasse `auto`):

```
Stack::~Stack ( )            // =====  Destruktor  =====
{
  delete [] value;           // dyn. Speicher freigeben
}
```

Der Destruktor wird automatisch am Lebensende für das jeweilige Objekt aufgerufen. Der explizite Aufruf eines Destruktors ist möglich, aber nicht üblich.

Liste 2.3: Stapel beliebiger Größe (mit Konstruktor, Destruktor)

```
1  class Stack
2  {
3    public:   // ---------- öffentliche Komponenten -------
4
5      Stack       ( int groesse = 1000 );     // Konstruktor
6      ~Stack      ( );                        // Destruktor
7      ...
8    private:    // -------- private Komponenten -----------
9
```

```
10      int   *value;              // Zeiger auf den Stapel
11      int   next;                // Stapelindex
12      int   max;                 // Stapelgröße
13  };
14
15  Stack::Stack ( int groesse )  // =====  Konstruktor =====
16  {
17    if( groesse <=0 )
18      groesse = 1000;            // ggf. Ersatzwert übernehmen
19    value = new int[groesse];    // dyn. Speicher belegen
20    max   = groesse;             // Stapelgröße übernehmen
21    next  = 0;                   // Index initialisieren
22  }
23
24  Stack::~Stack ( )              // =====  Destruktor  =====
25  {
26    delete [] value;             // dyn. Speicher freigeben
27  }
28
29  ...
```

Der Prototyp des Konstruktors in der Klassendefinition in Liste 2.3

```
Stack       ( int groesse = 1000 );     // Konstruktor
```

weist einen **Ersatzwert** auf. Ersatzwerte können, müssen aber nicht verwendet werden. Wird ein Stapel-Objekt ohne Größenangabe eingerichtet, dann wird der Ersatzwert als Stapelgröße verwendet. Bei einem Wert größer Null wird gemäß Implementierung in Liste 2.3 dieser Wert verwendet. Bei einer unsinnigen Größenangabe kleiner oder gleich Null wird hier ebenfalls der Ersatzwert verwendet.
Für die Vereinbarung eines Stapel-Objektes sind mehrere Schreibweisen möglich und üblich:

```
Stack stapel_1;               // ohne Parameter; Länge = 1000

Stack stapel_2 (128);         // mit Parameter; Länge = 128

Stack stapel_3 = Stack(500); // Konstruktoraufruf mit Größenangabe;
                              //  Länge = 500
```

Wenn ein Funktionsparameter ohne Ersatzwert vorhanden ist, dann muß bei einem Funktions- oder Konstruktoraufruf natürlich immer ein Argument angegeben werden.

2.4. Zuweisungsoperator und Kopierkonstruktor

2.4.1. Der Zuweisungsoperator

Das Kopieren von Objekten ist grundsätzlich möglich. Wenn zwei Stapel `stapel1` und `stapel2` gemäß Liste 2.1 (also mit gleichen Feldern fester Größe) vorhanden sind, dann kann ein Stapel mit dem anderen überschrieben werden:

```
stapel2 = stapel1;
```

Unmittelbar nach der Zuweisung sind beide Stapel gleich. Danach können sie natürlich wieder unterschiedlich weiterverwendet werden, da die Speicherplätze voneinander unabhängig sind.

Abbildung 2.2 zeigt zwei Stapel-Objekte mit jeweils einem Feld der Länge 8 vor und nach einer Zuweisung. Die Datenkomponenten des Objektes `stapel1` werden byteweise auf die Datenkomponenten des Objektes `stapel2` kopiert. Dabei entsteht unter anderem eine genaue Kopie des Feldes. Diese Art einer Kopie wird **flache Kopie** genannt.

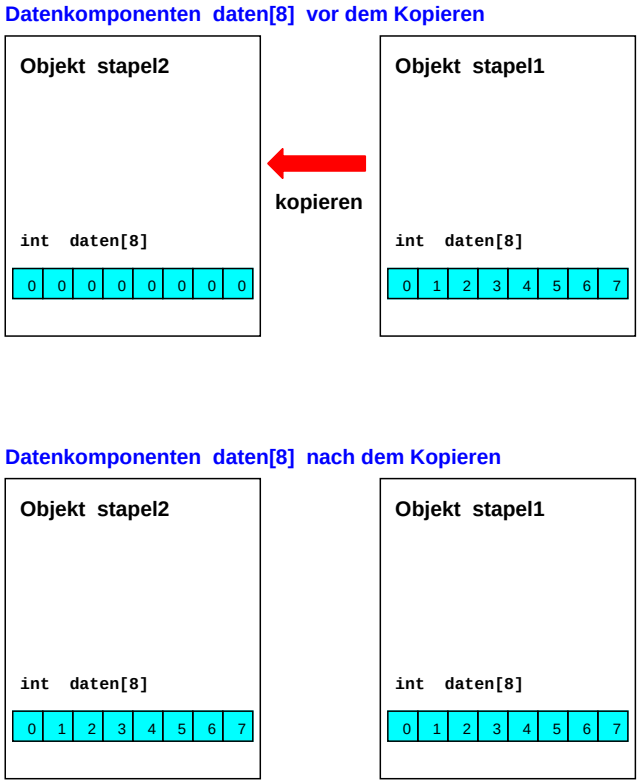


Abbildung 2.2.: Flache Kopie eines Objektes

Wenn die Objekte gemäß Liste 2.3 dynamisch angelegten Speicher enthalten, dann sind die Datenfelder, die die eigentlichen Stapel darstellen, *nicht Bestandteil des Speicherabbildes* des jeweiligen Objektes. Bestandteil sind jeweils nur die Speicherplätze für die Zeiger `value`. Wird nun eine Kopie durch eine Anweisung

```
stapel2 = stapel1;
```

angefertigt, dann entsteht ebenfalls eine flache Kopie. Das heißt in diesem Fall, daß der Inhalt des Zeigers `value` des Objektes `stapel1` auf den Platz des Zeigers im Objekt `stapel2` kopiert wird. Danach ist der ursprüngliche Zeigerwert im Objekt `stapel2` verloren und das dynamisch angelegte Feld ist zwar nicht mehr zugreifbar, der zugehörige Speicher bleibt aber belegt! Weiterhin zeigen nun beide Objekte auf denselben Speicherbereich. Methodenaufrufe für `stapel1` und `stapel2` verändern jetzt denselben Speicher. Der erste Destruktoraufruf wird das dynamisch angelegte Feld in `stapel1` freigeben, der zweite Destruktoraufruf wird versuchen, dasselbe Feld nochmals freizugeben! In der Abbildung 2.3 sind diese Verhältnisse dargestellt. Es ist klar, daß diese üblicherweise nicht erwünscht sind.

Die Lösung besteht darin, daß der Standard-Zuweisungsoperator, der vom Compiler angelegt und bisher automatisch verwendet wurde, durch Funktionsüberladung neu definiert wird. Die dazu nötige Vorgehensweise ist in Abbildung 2.3, unten, dargestellt und läuft in den folgenden Schritten ab:

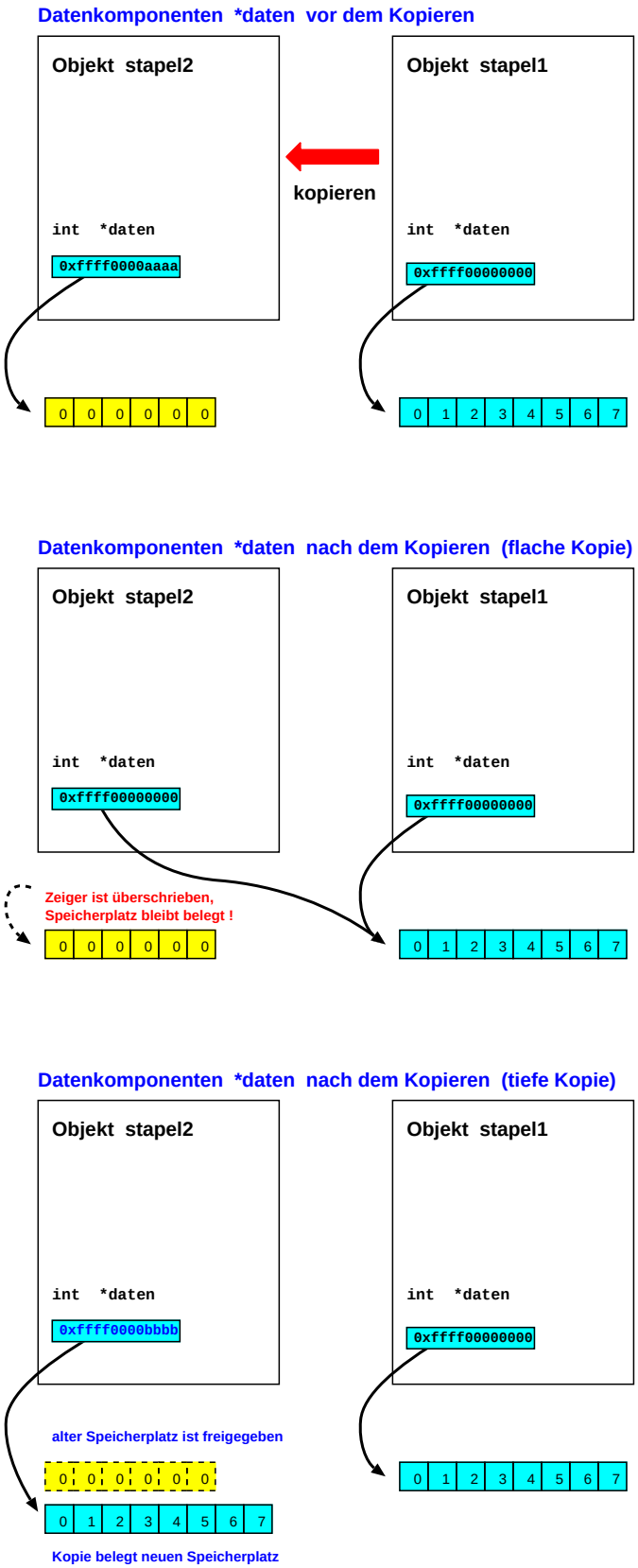


Abbildung 2.3.: Flache und tiefe Kopie eines Objektes im Vergleich

- Korrekte Freigabe des dynamisch belegten Feldes im Zielobjekt (hier `stapel2`).
- Anlegen eines neuen Feldes für das Zielobjekt in der Größe des darauf zu kopierenden Feldes (hier die Feldgröße aus `stapel1`).
- Elementweises Kopieren des Originalfeldes auf das neue Zielfeld und Übernehmen der Feldgröße.

Danach sind wieder zwei voneinander unabhängige Stapel vorhanden, die unmittelbar nach der Zuweisung natürlich gleich sind. Die Destruktoraufrufe beseitigen nun jeweils nur den eigenen dynamisch belegten Speicherplatz.

Liste 2.4: Stapel-Klasse, erweitert um einen Zuweisungsoperator

```
1  class Stack
2  {
3    public:   // ---------- öffentliche Komponenten -------
4
5      Stack       ( int groesse = 1000 );     // Konstruktor
6      ~Stack      ( );                        // Destruktor
7
8      void   operator = (const Stack &obj);   // Zuweisungsoperator
9
10     ...
11
12   private:    // -------- private Komponenten -----------
13
14     int   *value;             // Zeiger auf den Stapel
15     int   next;               // Stapelindex
16     int   max;                // Stapelgröße
17 };
18
19 //---------------------------------------------------------------------
20 // Klasse Stack :  operator =
21 //---------------------------------------------------------------------
22 void Stack::operator = (const Stack &obj)
23 {
24   if ( this != &obj ) {           // nicht auf identische Objekte anwenden!
25     int *save = value;            // Zeiger auf den eigenen dyn. Speicherplatz
26     value = new int[obj.max];     // dyn. Speicherplatz neu belegen
27     max   = obj.max;              // max. Größe der rechten Seite übernehmen
28     next  = obj.next;             // akt. Stapelzeiger übernehmen
29     for ( int i = 0; i < next; i++ )
30       value[i] = obj.value[i];    // Stapel kopieren
31     delete [] save;               // alten dyn. Speicherplatz freigeben
32   }
33    return;
34 }
```

Liste 2.4 zeigt die Erweiterung der Stapel-Klasse um einen Zuweisungsoperator. Der Prototyp in der Klassendefinition lautet

```
void operator = (const Stack &obj);   // Zuweisungsoperator
```

Der Funktionsname ist **operator** = , der einzige Parameter ist eine konstante Referenz auf ein Stapel-Objekt. In der Implementierung werden die oben genannten Schritte umgesetzt (Zeilen 22-34). Die erforderlichen Schritte sind zusätzlich durch folgende **if**-Anweisung geschützt:

```
if (this != &obj) {
  ...
}
```

Diese Bedingung prüft, ob Quellobjekt und Zielobjekt dieses Kopiervorganges ungleich sind. Nur in diesem Fall wird der Kopiervorgang ausgeführt. Der Vergleich wird durchgeführt, indem ein Zeiger des Zielobjektes auf sich selbst, der Zeiger **this**, mit der Adresse `&obj` des Quellobjektes (hier der Funktionsparameter) verglichen wird. Der Zeiger **this** auf das eigene Objekt ist immer vorhanden und kann deshalb ohne weiteres verwendet werden.
Die Reihenfolge, in der der dynamische Speicherplatz gelöscht und neu angelegt wird, ist ebenfalls von Bedeutung. Die Anweisung

```
int *save = value;            // Zeiger auf den eigenen dyn. Speicherplatz
```

rettet zunächst die Adresse auf das eigene dynamisch angelegte Feld, bevor dieses neu angelegt und der Zeiger `value` dabei überschrieben wird. Schlägt die Speicherplatzreservierung mittels **new** fehl, dann wird die Methode mit einer Ausnahme (Kapitel 5) verlassen, ohne daß der Zeiger `value` überschrieben wurde. Das Objekt ist somit noch intakt. Ist die Speicherplatzreservierung erfolgreich, wird der alte Speicher danach freigegeben:

```
delete [] save;               // alten dyn. Speicherplatz freigeben
```

Diese Implementierung ist damit ausnahmensicher (exception save). Sie arbeitet auch dann richtig, wenn auf die Überprüfung auf Selbstzuweisung verzichtet wird.
Nach der Überladung des Zuweisungsoperators wird bei der Ausführung einer Zuweisung

```
stapel2 = stapel1;
```

der überladene Operator **operator** = aufgerufen. Das Zielobjekt steht links, das Quellobjekt steht rechts. Das wird deutlicher, wenn man statt dieser üblichen Schreibweise einen Methodenaufruf verwendet:

```
stapel2.operator= ( stapel1 );
```

Beide Aufrufarten sind völlig gleichwertig. Die letztere ist ungebräuchlich, zeigt aber den tatsächlichen Zusammenhang zwischen der Operatorfunktion **operator** = und der Verwendung des Gleichheitszeichens als Zuweisungsoperator.

Mit Hilfe des gerade überladenen Zuweisungsoperators lassen sich, wie gezeigt, einfache Zuweisungen in der üblichen Schreibweise darstellen. In ***C*** und ***C++*** sind für Standardobjekte aber auch Mehrfachzuweisungen erlaubt. Die Zuweisung desselben Wertes an drei **double**-Größen kann zum Beispiel wie folgt abkürzend dargestellt werden:

```
x = y = z = 23.777;
```

Das ist mit der gerade eingeführten Form des Zuweisungsoperators nicht möglich. Wenn die Methode jedoch eine Referenz auf das gerade veränderte, eigene Objekt zurückgibt, dann sind geschachtelte Aufrufe (wie am Ende von Abschnitt 1.6 dargestellt) auch für den Zuweisungsoperator möglich:

```
Stack& Stack::operator = (const Stack &obj)
{
  if ( this != &obj ) {           // nicht auf identische Objekte anwenden!
    int *save = value;            // Zeiger auf den eigenen dyn. Speicherplatz
    value = new int[obj.max];     // dyn. Speicherplatz neu belegen
    max   = obj.max;              // max. Größe der rechten Seite übernehmen
    next  = obj.next;             // akt. Stapelzeiger übernehmen
    for ( int i = 0; i < next; i++ )
      value[i] = obj.value[i];    // Stapel kopieren
    delete [] save;               // alten dyn. Speicherplatz freigeben
  }
  return  *this;
}
```

Die Referenz auf das eigene Objekt wird in der `return`-Anweisung beschafft:

```
return  *this;
```

Dazu wird der Zeiger dereferenziert und liefert damit zunächst das gesamte eigene Objekt. Der Rückgabetyp verlangt die Rückgabe einer Referenz. Deshalb wird eine Referenz auf das mit `*this` angesprochenen, eigene Objekt zurückgegeben. Nach dieser Änderungen sind Mehrfachzuweisungen der Form

```
stapel3 = stapel2 = stapel1;
```

möglich. Diese Anweisung entspricht den folgenden geschachtelten Aufrufen

```
stapel3.operator= ( stapel2.operator= ( stapel1 ) );
```

Die Methode `operator =`, die auf das Objekt `stapel3` angesetzt wird, liefert jetzt natürlich auch eine Referenz auf dieses Objekt zurück. Da dieser Aufruf aber nicht auf der rechten Seite einer Zuweisung steht, wird diese Referenz nicht weiter berücksichtigt.

Die allgemeine Form eines Zuweisungsoperators für eine Klasse `X` lautet:

```
X& operator = ( const X &x ) {  ...  }
```

2.4.2. Der Kopierkonstruktor

Es gibt weitere Zuweisungsarten, bei denen vollständige Objekte kopiert werden, unter anderem die Anfangswertzuweisungen in einer Vereinbarung:

```
Stack stapel1;
  ...
Stack stapel2 = stapel1;  // mit einer Kopie von stapel1 beginnen
Stack stapel3 (stapel1);  // mit einer Kopie von stapel1 beginnen
```

Hierbei ist zwar sicher, daß die beiden Objekte in jeder Vereinbarung unterschiedlich sind, die Anfangswertzuweisung für `stapel2` wird aber trotz des verwendet Gleichheitszeichens nicht durch den möglicherweise bereits überladenen Zuweisungsoperator durchgeführt!

Für Vereinbarung mit Anfangswertzuweisung wird ein sogenannte **Kopierkonstruktor** aufgerufen, der für jede Klasse zunächst automatisch erzeugt wird. Wenn dieser Kopierkonstruktor nicht überladen ist, dann fertigt er eine flache Kopie an. Im Falle der Stapel-Klasse mit

dynamisch angelegtem Stapel ist das natürlich nicht erwünscht. Der automatisch erzeugte Kopierkonstruktor muß also durch einen eigenen überladen werden, der eine tiefe Kopie anlegt. Die allgemeine Form eines Kopierkonstruktor für eine Klasse `X` lautet:

```
X ( const X &x ) {  ...  }
```

Der Kopierkonstruktor besitzt keinen Rückgabewert. Der einzige Parameter ist eine konstante Referenz auf ein Objekt der eigenen Klasse. Für die Stapel-Klasse sieht der Kopierkonstruktor wie folgt aus:

```
Stack::Stack (const Stack &obj)
{
  value   = new int[obj.max];         // dyn. Speicher belegen
  max     = obj.max;                  // Größe übernehmen
  next    = obj.next;                 // Zeiger übernehmen
  for( int i=0; i<next; i++ )         // Datenfeld kopieren
    value[i]  = obj.value[i];
}
```

Der Kopierkonstruktor wird auch bei anderen Gelegenheiten automatisch verwendet:

```
void  fkt1 ( Stack s ) {              // Wertübergabe : Kopierkonstruktor
  ...
}

Stack fkt2 ( ) {
  Stack hilf;
  ...
  return hilf;                        // Wertrückgabe : Kopierkonstruktor
}
```

Die Funktion `fkt1` erhält beim Aufruf mittels Wertübergabe die Kopie eines vollständigen Stapels. Zur Anfertigung dieser Kopie wird automatisch der Kopierkonstruktor aufgerufen.
Die Funktion `fkt2` verwendet einen Stapel `hilf` als lokale Größe, dessen Wert mittels **return** an den Aufrufer zurückgeben wird. Auch zur Anfertigung dieser Kopie wird automatisch der Kopierkonstruktor aufgerufen.

2.5. Automatisch erzeugte Komponentenfunktionen

Aus den letzten Abschnitten wurde ersichtlich, daß jede Klasse nach ihrer Definition vier automatisch erzeugte Komponentenfunktionen besitzt. Diese werden verwendet, wenn sie nicht vom Programmierer durch eigene Methoden überladen werden:

- Konstruktor
- Kopierkonstruktor
- Destruktor
- Zuweisungsoperator

Auf Grund der Ausführungen in den letzten beiden Abschnitten ist klar, daß man auf eine Überladung dieser vier Ersatzfunktionen nur bei sehr einfachen Klassen verzichten kann.

Bei Verwendung von dynamisch angelegtem Speicher *muß* eine Überladung stattfinden. Die Ersatzfunktionen haben für eine Klasse `X` folgende Schnittstellen:

Konstruktor	`X ( )`
Kopierkonstruktor	`X ( const X &x )`
Destruktor	`~X ( )`
Zuweisungsoperator	`X& operator= ( const X &x )`

2.6. Befreundete Klassen und Funktionen

Die zu Beginn des Kapitels erläuterte Vorgehensweise der vollständigen Abschottung ist bei vielen Anwendungsfällen zu einengend. Es besteht deshalb die Möglichkeit, die Abschottung teilweise aufzuheben. Hierzu können einzelne Funktionen oder ganze Klassen mit Hilfe des Schlüsselwortes **`friend`** als befreundet ausgewiesen werden. Befreundete Funktionen oder Klassen haben dann ohne weiteres Zugang zu den privaten Daten der Klassen, mit denen sie befreundet sind.

In Liste 2.5 werden zwei Klassen definiert. Die Klasse `vektor` stellt Vektoren mit drei Elementen zur Verfügung, die Klasse `matrix` stellt 3×3-Matrizen zur Verfügung. Bei der Multiplikation einer Matrix mit einem Vektor entsteht ein Ergebnisvektor:

$$\vec{w} = M \times \vec{v}$$

Zur schnellen Abwicklung der Multiplikationsvorschrift ist es günstig, wenn die Multiplikationsfunktion sowohl direkten Zugriff auf Vektorkomponenten als auch auf die Matrixkomponenten hat.
In dem Programm in Liste 2.5 wird diese Aufgabe von der Funktion `mat_vek` durchgeführt (Zeile 61-69). Diese Funktion gehört zu keiner der beiden Klassen, ist jedoch durch die in beiden Klassendefinition enthaltene Anweisungen

```
friend vektor  mat_vek ( matrix &m, vektor &v );
```

mit beiden Klassen befreundet. Innerhalb der Funktion kann deshalb auf die privaten Daten der beiden Funktionsargumente einfach mit Hilfe der Punkt-Operators (Zugriff auf Struktur- und Klassenkomponenten) zugegriffen werden (Zeile 66):

```
h.vek[i] += m.mat[i][j] * v.vek[j];
```

Die Zeilen 5 und 6 enthalten sogenannte **Vorwärtsreferenzen**. Diese geben die Klassennamen ohne weitere Einzelheiten bekannt, sodaß innerhalb der nachfolgenden Klassendefinitionen bereits auf Klassen Bezug genommen werden kann, die erst weiter unten definiert werden.

Auch eine gesamte Klasse kann mit einer anderen Klasse befreundet sein. Dazu ist die nachfolgende Schreibweise zu verwenden.

```
class xxx
{
  friend  class yyy;    // befreundete Klasse
  ...
}

class yyy
{
  ...
}
```

Die Methoden der Klasse `yyy` haben nun direkten Zugriff auf die privaten Komponenten der Klasse `xxx`. Diese Beziehung ist einseitig. Wenn die Methoden der Klasse `xxx` Zugriff auf die privaten Komponenten der Klasse `yyy` haben sollen, dann ist eine weiter **`friend`**-Vereinbarung notwendig.

Liste 2.5: Verwendung einer `friend`-Funktion (`2.6-1.mat.cc`)

```
1  #include  <iostream>
2
3  using namespace std;
4
5  class vektor;                                    // Vorwärtsreferenz
6  class matrix;                                    // Vorwärtsreferenz
7
8  // ========================================================================
9  //        Class:  matrix
10 //  Description:  3x3-Matrizen
11 // ========================================================================
12 class matrix
13 {
14   public:
15     matrix ();                                   // Konstruktor
16     friend vektor  mat_vek ( matrix & m, vektor & v );
17
18   private:
19     double  mat[3][3];
20 };
21
22 // ========================================================================
23 //        Class:  vektor
24 //  Description:  Vektoren der Länge 3
25 // ========================================================================
26 class vektor
27 {
28   public:
29     vektor();                                    // Konstruktor
30     friend vektor  mat_vek ( matrix & m, vektor & v );
31
32   private:
33     double  vek[3];
34 };
35
36 //-------------------------------------------------------------------------
37 //       Class:  vektor
38 //      Method:  vektor
39 // Description:  Konstruktor
40 //-------------------------------------------------------------------------
41 vektor :: vektor ()
42 {
43   for( int i=0; i<3; i++ )  vek[i]  = 0.0;
44 }
45
46 //-------------------------------------------------------------------------
47 //       Class:  matrix
48 //      Method:  matrix
49 // Description:  Konstruktor
50 //-------------------------------------------------------------------------
51 matrix :: matrix ()
52 {
53   for( int i=0; i<3; i++ )
54     for( int j=0; j<3; j++ )  mat[i][j]= 0.0;
55 }
```

```
56
57 // ===  FUNCTION  ================================================================
58 //         Name:  mat_vek
59 //  Description:  Matrix x Vektor
60 // ===============================================================================
61 vektor mat_vek ( matrix & m, vektor & v )
62 {
63   vektor h;                                  // Initialisierung durch den Konstruktor
64   for(int i=0;i<3;i++) {
65     for( int j=0; j<3; j++ )
66       h.vek[i] += m.mat[i][j] * v.vek[j];
67   }
68   return h;
69 }
70
71 //-------------------------------------------------------------------------
72 //  Hauptprogramm
73 //-------------------------------------------------------------------------
74 int main () {
75   vektor  v, w;
76   matrix  m;
77
78   w = mat_vek ( m, v );                          // Matrix * Vektor
79
80   return 0;
81 }
```

3. Überladen von Operatoren

In *C++* können nicht nur Funktionen, sondern auch alle Operatoren überladen werden. Am Beispiel des Zuweisungsoperators wurde das bereits gezeigt. Die Überladung wird in der Regel dazu genutzt, einem Operator eine *zusätzliche* Bedeutung zu geben. So ist der Additionsoperator + zunächst nur für Zahlengrößen definiert. Es liegt nahe, den Operator für eine Klasse von Vektoren so zu überladen, daß in der Programmdarstellung ein Pluszeichen für die Vektoraddition geschrieben werden darf, also zum Beispiel

```
Vek3  u, v, w;
  ...
w = u + v;        // Vektoraddition
```

Durch die Überladung weiterer Operatoren lassen sich dann nahezu alle grundlegenden Vektorverknüpfungen in der gewohnten mathematischen Schreibweise darstellen.
Diese Vorgehensweise ist natürlich auf andere Beispiele übertragbar. So könnte ein überladener Additionsoperator in einer Klasse zur Darstellung von Mengen dazu verwendet werden, einer Menge ein weiteres Element hinzuzufügen.

Der Vorteil besteht offensichtlich darin, daß hierdurch sinnfällige, vorlagennahe oder auch gewohnte Darstellungen in einem Programmtext möglich sind. Insbesondere kann bei den zweistelligen Operatoren die Infix-Schreibweise beibehalten werden (Operator steht zwischen den beiden Operanden). Ein weiterer wesentlicher Vorteil ist die verkürzte Darstellung, die zu sehr dichtem Programmcode führt. Die alternative Darstellungsmöglichkeit sind Funktionsaufrufe, die gegebenenfalls geschachtelt werden und dadurch das Textverständnis meistens nicht erleichtern.

Ein Nachteil kann entstehen, wenn in einem Projekt zahlreiche Klassen mit überladenen Operatoren verwendet werden. Wenn der gleiche Operator für mehrere Klassen jeweils auch noch mehrfach überladen ist, dann ist das Verständnis einer Programmstelle durch Betrachten der lokalen Umgebung nicht mehr ohne weiteres möglich. Bei maßvollem Gebrauch dieser Einrichtungen ist das jedoch kein Einsatzhindernis.

Die wesentlichen Merkmale und die Handhabung der Operatorüberladung werden in den folgenden Abschnitten an den arithmetischen Operatoren gezeigt und sollten dann auf andere Operatoren übertragbar sein.

3.1. Einstellige arithmetische Operatoren

Zunächst sollen die einstelligen arithmetischen Operatoren, also die Vorzeichen, überladen werden. Dazu wird eine Klasse für Vektoren mit drei Elementen verwendet. Die wesentlichen Teile sind in Liste 3.1 wiedergegeben.

Liste 3.1: Überladung von Vorzeichenoperatoren (3.1-1-vek3-methoden.cc)

```
1  class Vek3 {
2    public:
3
4      Vek3 ( double _x = 0.0, double _y = 0.0, double _z = 0.0 );  // Konstruktor
5
6      Vek3    operator +  ( );                    // positives Vorzeichen
7      Vek3    operator -  ( );                    // negatives Vorzeichen
8      Vek3    operator +  ( const Vek3 &other );  // Addition
9      Vek3    operator += ( const Vek3 &other );  // Addition mit Zuweisung
10     double  get_x       ( ) const { return x; } // Komponentenzugriff
11     double  get_y       ( ) const { return y; } // Komponentenzugriff
12     double  get_z       ( ) const { return z; } // Komponentenzugriff
13
14   private:
15     double  x, y, z;                            // Vektorkomponenten
16 }; // -----  end of class  Vek3  -----
17
18 Vek3::Vek3 ( double _x, double _y, double _z )  // Konstruktor
19 {
20   x = _x; y = _y; z = _z;
21 }
22
23 Vek3 Vek3::operator + ( )                       // positives Vorzeichen
24 {
25   return *this;
26 }
27
28 Vek3 Vek3::operator - ( )                       // negatives Vorzeichen
29 {
30   return Vek3( -x, -y, -z );                    // Konstruktoraufruf
31 }
```

Implementierung mit Hilfe von Methoden

Ein **positives Vorzeichen** ist für den Rechengang stets überflüssig, wird aber gelegentlich zur Hervorhebung eines positiven Wertes verwendet, also gewissermaßen als Kommentar. Die Implementierung gibt einfach den Wert des Objektes oder Ausdruckes zurück, vor dem der Operator steht, zum Beispiel

```
v3  = +v1;
```

Die rechte Seite `+v1` dieser Zuweisung wird also durch den Wert des Objektes `v1` ersetzt.

Die Implementierung für das **negative Vorzeichen** muß einen neuen Vektor zurückgeben, bei dem alle Komponenten den negativen Wert der Originalkomponenten besitzen, ohne daß der Originalvektor verändert werden darf:

```
v3  = -v1;
```

In der Funktion **`operator -`** wird deshalb ein Hilfsvektor angelegt, dessen Komponenten aus denen des Originalvektors bestimmt werden und der anschließend als Wert zurückgeben werden kann. Die Rückgabe als Wert ist erforderlich, weil der Hilfsvektor in der Operatorfunktion lokal ist und nach dem Aufruf nicht mehr existiert. Bei Rückgabe eines Zeigers darauf würde

dieser ins Leere zeigen! Die einfachste Möglichkeit besteht darin, in der **return**-Anweisung den Konstruktor der Klasse aufzurufen und diesen mit den richtigen Komponentenwerten zu initialisieren:

```
return Vek3( -x, -y ,-z );
```

Dieser Konstruktoraufruf liefert ein namenloses, lokales `Vek3`-Objekt, dessen Kopie sofort über **return** zurückgegeben wird.

Die Funktion **operator** - ist eine Methode der Klasse `Vek3` und kann deshalb nur direkt auf ein Objekt angewendet werden. Dieses Objekt kann nur dasjenige sein, welches rechts vom Vorzeichen steht. Die Funktion kann deshalb keine weiteren Argumente haben. Erhellend ist hier wieder die ungebräuchliche, aber gleichwertige Funktionsschreibweise für die Verwendung des Operators:

```
v3  = v1.operator- ();            // v3 = -v1
```

Die Verwendung des Vorzeichenoperators stellt gewissermaßen eine zulässige Kurzschreibweise für diesen Funktionsaufruf dar.

Implementierung mit Hilfe von `friend`-Funktionen

In manchen Fällen kann oder muß auf eine zweite Möglichkeit der Implementierung zurückgegriffen werden. Die beiden Vorzeichenoperatoren können auch mit Hilfe von **friend**-Funktionen überladen werden. Dazu sind die Funktionen in folgender Weise in der Klassendefinition als befreundet bekannt zu machen:

```
friend Vek3 operator + ( const Vek3 &obj );   // positives Vorzeichen
friend Vek3 operator - ( const Vek3 &obj );   // negatives Vorzeichen
```

Da es sich hier nicht mehr um Methoden handelt, können die Funktionen nicht direkt auf ein Objekt angewendet werden. Deshalb müssen beide einen Parameter besitzen, der eine Referenz auf das Objekt übergibt, auf das der Operator angewendet werden soll. Die Implementierung der beiden Funktionen sieht wie folgt aus:

```
Vek3 operator + ( const Vek3 &obj )
{
  return obj;
}

Vek3 operator - ( const Vek3 &obj )
{
  return Vek3( -obj.x, -obj.y, -obj.z );
}
```

Die Funktion **operator** + reicht den Wert des erhaltenen Objektes einfach mittels **return** weiter.
Die Funktion **operator** - muß wieder einen lokalen Hilfsvektor erzeugen und kann als befreundete Funktion mit einem Komponentenzugriff auf die Vektorkomponenten des Argumentes zugreifen.

3.2. Zweistellige arithmetische Operatoren

Zur Bereitstellung der Grundrechenarten sind die Operatoren `+`, `-`, `*` und `/` für die jeweilige Klasse zu überladen. Für die Vektoren der Klasse `Vek3` sollen Anweisungen der folgenden Art möglich sein:

```
Vek3  u, v, w;
  ...
w = u + v;        // Vektoraddition
```

Zunächst einige Überlegungen zur Implementierung:

- Da das Ergebnis der Addition in der Regel keinem der beiden Vektoren der rechten Seite entsprechen wird, muß das Additionsergebnis zunächst in einem Hilfsvektor erzeugt werden.
- Die Operatorfunktion muß eine Kopie des Wertes dieses Hilfsvektors zurückgeben, da dieser eine lokale Größe ist.
- Wenn der Operator als Methode implementiert wird, dann ist der linke Operand jeweils dasjenige Objekt, auf welches der Operator angewendet wird. Damit muß nur der rechte Operand als Parameter übergeben werden.
- Wenn der Operator als **friend**-Funktion implementiert wird, dann müssen beide Operanden als Parameter übergeben werden.
- Eine Addition darf beide Operanden nicht ändern.

Implementierung als Methode

Aus diesen Überlegungen ergibt sich folgende Implementierung als Komponentenfunktion:

```
Vek3 Vek3::operator + ( const Vek3 &rs )
{
  return  Vek3( x + rs.x,
                y + rs.y,
                z + rs.z );
}
```

Die Vektoraddition wird auf die Addition der Vektorkomponenten zurückgeführt. Der Ergebnisvektor wird aus den Komponenten des linken Operanden (`x`, `y`, `z`) und aus denen des rechten Operanden (`rs.x`, `rs.y`, `rs.z`) gebildet, die in der Parameterliste des Konstruktors addiert werden. Der Konstruktor erzeugt ein namenloses Vektorobjekt, dessen Wert sofort mittels **return** zurückgegeben wird. Das namenlose Vektorobjekt selbst verschwindet sofort wieder, da seine Lebensdauer auf die Funktion **operator** + begrenzt ist.

In der Addition

```
w = u + v;                // Vektoraddition
```

ist `u` das Objekt, auf welches der Operator angewendet wird, `v` wird als Argument übergeben. Die Funktionsschreibweise sieht so aus:

```
w = u.operator+ ( v );    // Vektoraddition
```

Implementierung als `friend`-Funktion

Aus den genannten Überlegungen ergibt sich sofort die folgende Implementierung als **friend**-Funktion:

```
Vek3 operator + ( const Vek3 &ls, const Vek3 &rs )
{
  return  Vek3( ls.x + rs.x,
                ls.y + rs.y,
                ls.z + rs.z );
}
```

In der Addition

```
w = u + v;                // Vektoraddition
```

werden nun die beiden Operanden `u` und `v` als Argumente an die **friend**-Funktion **operator** `+` übergeben. Die Funktionsschreibweise sieht so aus:

```
w = operator+ ( u, v );   // Vektoraddition
```

3.3. Arithmetische Zuweisungsoperatoren

Die arithmetischen Zuweisungsoperatoren

```
+=   -=   *=   /=   %=
```

verbinden eine Rechenoperation mit einer Zuweisung. In der Zuweisung

```
w += u;
```

wird der Wert von `u` zum Wert von `w` addiert. Bei der Implementierung als Methode ist `w` dasjenige Objekt, auf welches diese Methode angewendet wird. Die rechte Seite `u` muß dann als Argument übergeben werden. Bei Implementierung als **friend**-Funktion ist `w` das erste, `u` das zweite Argument.
Der linke Operand wird auf jeden Fall verändert und stellt damit die Zielgröße der Rechnung dar. In beiden Fällen würde somit eine Funktion genügen, die keinen Wert zurückgibt.

Implementierung als Methode

Die Implementierung als Methode sieht wie folgt aus:

```
void  Vek3 :: operator  += ( const Vek3 &rs )
{
  x += rs.x;
  y += rs.y;
  z += rs.z;
}
```

Die rechte Seite (Argument `rs`) hat den Zusatz **const**, da diese Größe auf keinen Fall verändert werden soll.

Implementierung als `friend`-Funktion

Die Implementierung als **`friend`**-Funktion sieht so aus:

```
void operator += ( Vek3 &ls, const Vek3 &rs )
{
  ls.x  += rs.x;
  ls.y  += rs.y;
  ls.z  += rs.z;
}
```

Die linke Seite geht in die Rechnung ein, ist aber auch Zielgröße. Somit muß das Argument `ls` veränderbar sein. Die rechte Seite (Argument `rs`) hat auch hier den Zusatz **`const`**, da sie auf keinen Fall verändert werden soll.

Implementierung mit Rückgabewert

Der Rückgabetyp ist in beiden Implementierung **`void`**, was für die gezeigte, einfache Anwendung dieser Operatoren ausreicht. Mehrfachzuweisungen der Art

```
w  =  v  +=  b;
```

sind damit jedoch nicht möglich. Die Abarbeitung dieser Anweisung wird durch die folgende Klammerung verdeutlicht:

```
w  =  ( v  +=  b );
```

Die Abarbeitung von Zuweisungen geschieht von rechts nach links. Dazu ist zunächst der Wert der Klammer zu bestimmen. Die enthaltene Operation wird durch den Operator `+=` durchgeführt. Beide gezeigten Implementierung haben jedoch keinen Rückgabewert, das heißt die Klammer hätte keinen Wert! Damit ist diese Mehrfachzuweisung nicht ausführbar. Dieser Mangel läßt sich in beiden Fällen beheben.

Hierzu müssen die Operatorfunktionen so geändert werden, daß der Rückgabetyp jeweils ein Referenz auf ein `Vek3`-Objekt ist.
Die Methode gibt mit Hilfe des **`this`**-Zeigers den Wert des Objektes zurück, auf das sie angewendet wurde:

```
Vek3& Vek3 :: operator  += ( const Vek3 &rs )
{
  x += rs.x;
  y += rs.y;
  z += rs.z;
  return  *this;
}
```

Die **`friend`**-Funktion gibt den (nun veränderten) Wert ihres ersten Argumentes, das heißt der linken Seite, zurück:

```
Vek3& operator += ( Vek3 &ls, const Vek3 &rs )
{
  ls.x  += rs.x;
  ls.y  += rs.y;
  ls.z  += rs.z;
  return ls;
}
```

In beiden Fällen erhält nun die Klammer in dem obigen Beispiel den Ergebniswert der in ihr enthaltenen Rechenoperation. Dieser kann nun in der Zuweisung weiterverwendet werden.

3.4. Eingabe- und Ausgabe-Operatoren

Der Eingabeoperator >> und der Ausgabeoperator << können überladen werden. Zweck der Überladung ist die vereinfachte Ausgabe von Objekten mit benutzerdefinierten Datentypen.

Überladung der Ausgabe

Die Ausgabedaten werden an ein Objekt der Klasse `ostream` übergeben. Zu dieser Klasse gehört auch der zu überladende Ausgabeoperator. Für die Klasse `Vek3` könnte der überladene Ausgabeoperator zum Beispiel wie folgt aussehen:

```
ostream & operator << ( ostream &out, const Vek3 &v )
{
  out << "( " << v.x << " , " << v.y << " , " << v.z << " )" ;
  return  out;
}
```

Diese Funktion kann als **`friend`**-Funktion der jeweiligen Klasse (hier `Vek3`) implementiert werden. Der grundsätzliche Aufbau einer derartigen Ausgabefunktion ist stets gleich:

1. Argument	`ostream &out`	Referenz auf ein `ostream`-Objekt
2. Argument	`const Vek3 &v`	Referenz auf ein `Vek3`-Objekt
Rückgabetyp	`ostream &`	Referenz auf ein `ostream`-Objekt

Das zweite Argument ist stets eine Referenz auf ein Objekt der Klasse, für die die Ausgabe überladen wird. Das Innere der Funktion bilden Ausgabeanweisungen, die auf die Datenkomponenten der Objekte zugreifen und deren Werte entsprechend formatiert an ein `ostream`-Objekt (1. Argument) ausgeben. Durch Austausch des 2. Arguments und der Ausgabeanweisungen im Innern kann diese Form für beliebige Klassen angepaßt werden. Die Ausgabeanweisung

```
cout  << a ;
```

erzeugt dann zum Beispiel die Zeile

```
( 3 , 3 , 4 )
```

Die Rückgabe einer Referenz ermöglicht die Verkettung mehrerer Ausgaben in einer Anweisung:

```
cout  << a << b ;
```

Der Operator << ist eigentlich der Linksschieben-Operator, der in ***C++*** für die Ausgabeklassen als Ausgabeoperator verwendet wird. Der Operator << ist linksbindend, das heißt bei Verkettung geschieht die Abarbeitung von links nach rechts. Das entspricht für dieses Beispiel der folgenden Klammerung:

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

Das Ergebnis der Klammer ist eine Referenz auf ein `ostream`-Objekt. Diese kann sofort wieder als erstes Argument für die Ausführung des nächsten Ausgabeoperators dienen.

Überladung der Eingabe

Die Eingabedaten werden von einem Objekt der Klasse `istream` übernommen. Zu dieser Klasse gehört auch der zu überladende Eingabeoperator. Für die Klasse `Vek3` könnte der überladene Eingabeoperator zum Beispiel wie folgt aussehen:

```
istream & operator >> ( istream &in, Vek3 &v )
{
  in >> v.x >> v.y >> v.z ;
  return  in;
}
```

Die wichtigsten Bestandteile dieser Funktion sind

1. Argument	`istream &in`	Referenz auf ein `istream`-Objekt
2. Argument	`const Vek3 &v`	Referenz auf ein `Vek3`-Objekt
Rückgabetyp	`istream &`	Referenz auf ein `istream`-Objekt

Der Operator dieses Beispiels liest einfach drei aufeinanderfolgende Zahlenwerte ein und weist diese den Komponenten eines Vektors (2. Argument) zu. Die folgende Form erwartet zusätzlich eine Umrahmung der drei Werte durch runde Klammern und die Trennung der Werte durch Kommata. Diese Bestandteile der Eingabe werden hier allerdings nicht überprüft.

```
istream & operator >> ( istream &in, Vek3 &v )
{
  char  klammer, komma;
  in >> klammer >> v.x >> komma >> v.y >> komma >> v.z >> klammer;
  return  in;
}
```

Damit können nun Vektoren in derselben Form eingegeben werden, in der sie durch die oben gezeigte Ausgabefunktion ausgegeben werden.
Alle ***C++***-Operatoren können in ähnlicher Weise überladen werden. Für die nicht behandelten Operatoren muß hier auf die Literatur verwiesen werden ([IH04], [Str00].

4. Vererbung

Ein wesentliches Ziel der Objektorientierten Programmierung ist die Erzeugung wiederverwendbarer Programmbestandteile. Ein wichtiger Schritt hierfür ist die im vorhergehenden Kapitel dargestellte Erzeugung von Klassen als Darstellungsform abstrakter Datentypen.
In größeren Anwendungen ist es nahezu unvermeidlich, daß eigene oder fremde Klassen ergänzt oder verändert werden müssen, um sinnvoll weiterverwendet werden zu können. Genauer gesagt kann der Wunsch oder die Notwendigkeit bestehen, eine vorhandene Klasse durch folgende Maßnahmen zu erweitern:

- weitere Datenelemente aufnehmen
- weitere Methoden (Funktionskomponenten) bereitstellen
- vorhandene Methoden überladen

Um eine Neuerstellung zu vermeiden, wurde der Erweiterungsmechanismus der **Vererbung** eingeführt. Die Grundidee besteht darin, eine bereits vorhandene **Basisklasse** unangetastet zu lassen und nur die ihr gegenüber notwendigen Erweiterungen in einer **abgeleiteten Klasse** anzugeben.

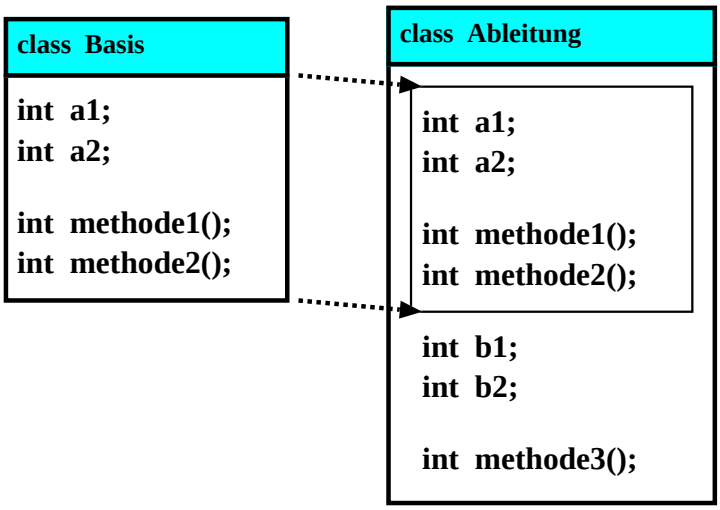


Abbildung 4.1.: Basisklasse und abgeleitete Klasse

Die Abbildung 4.1 zeigt ein einfaches Beispiel. Die Basisklasse `Basis` enthält zwei Datenelemente und zwei Methoden. Die spezialisierte Klasse `Ableitung` soll ebenfalls diese Datenelemente und Methoden enthalten. Zusätzlich werden aber die beiden Datenelemente `b1` und `b2` benötigt. Außerdem ist eine weitere Methode erforderlich. Objekte der Klasse `Ableitung` können die Datenelemente und Methoden der Basisklasse benutzen, ohne diese Teile neu implementieren zu müssen.

Eine derartige Basisklasse könnte zum Beispiel dazu verwendet werden, die grundlegenden Parameter und Verhaltensweisen eines Gleichstrommotors für eine Antriebsauslegung zur Verfü-

gung zu stellen. Einzelne Motorbaureihen würden durch abgeleitete Klassen dargestellt. Diese erweitern die Basisklasse um die typenspezifischen Besonderheiten (Geräteparameter, abweichende Berechnungsvorschriften, und so weiter) der einzelnen Baureihen.

4.1. Abgeleitete Klassen

Um eine neue Klasse `B` von einer vorhandenen Basisklasse `A` abzuleiten ist folgende Schreibweise anzuwenden:

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

Der Zusatz **`public`** `A` hinter dem neuen Klassennamen weist die Klasse `B` als abgeleitete Klasse der Basisklasse `A` aus. Das Schlüsselwort **`public`** legt fest, das über ein `B`-Objekt auf alle **`public`**-Komponenten der Basisklasse zugegriffen werden kann. Damit stehen einem `B`-Objekt sofort alle öffentliche Methoden der Basisklasse zur Verfügung.

Liste 4.1 zeigt eine einfache Vererbung. Die Basisklasse `bahn` stellt die Grundlage für die Darstellung von Bewegungsbahnen dar (zum Beispiel für eine Bearbeitungsmaschine). Von dieser Basisklasse wird dann eine Klasse `kbogen` abgeleitet, die zur Darstellung von Kreisbögen dient.

Jede Bewegungsbahn hat, unabhängig von ihrer tatsächlichen Gestalt, einen Anfangs- und einen Endpunkt. Diese Punkte werden als Ortsvektoren im dreidimensionalen Raum dargestellt. Hierzu werden Objekte der im letzten Kapitel eingeführten Klasse `Vek3` verwendet. Obwohl `Vek3` selbst eine Klasse ist, handelt es sich bei dieser Verwendung von `Vek3`-Objekten als Datenkomponenten *nicht* um eine Anwendung der Vererbung!

Die Klasse `bahn` (Liste 4.1, Zeilen 4-17) besitzt einen Konstruktor, der zwei Vektoren als Anfangs- und Endpunkt an die jeweiligen Datenkomponenten zuweist. Mit Hilfe der Methoden `start` und `ziel` können Anfangspunkt und Endpunkt einzeln gesetzt werden. Die Methode `laenge` ermittelt die Bahnlänge als Streckenlänge zwischen Anfangs- und Endpunkt ausschließlich mittels Vektorrechnung:

```
return (pe-pa).laenge();
```

Die Vektordifferenz `(pe-pa)` stellt ein namenloses `Vek3`-Objekt dar, auf welches die Vektormethode `laenge` angewendet wird.

Die Klasse `kbogen` (Liste 4.1, Zeilen 22-36) wird gemäß der oben dargestellten Schreibweise von der Basisklasse `bahn` abgeleitet:

```
class kbogen : public bahn
{
  ...
};
```

In beiden Klassen werden die Zugriffsrechte für die Datenkomponenten mit dem neuen Schlüsselwort **`protected`** bezeichnet. Diese Komponenten sind privat, jedoch für abgeleitete Klassen direkt zugreifbar. Reine **`private`**-Komponenten wären für abgeleitete Klassen nicht zugreifbar!

Liste 4.1: Basisklasse bahn und abgeleitete Klasse kbogen (bahn1.hh)

```
1  //------------------------------------------------------------------------
2  //  Klasse bahn
3  //------------------------------------------------------------------------
4  class bahn {
5
6    public:
7      bahn    ( void )  {  }
8      bahn    ( const Vek3 &p1, const Vek3 &p2 )  { pa=p1; pe=p2; }
9
10     void    start   ( const Vek3 &p ) { pa = p; }
11     void    ziel    ( const Vek3 &p ) { pe = p; }
12     double  laenge  ( void )          { return (pe-pa).laenge(); }
13
14   protected:
15     Vek3    pa;                                  // Anfangspunkt
16     Vek3    pe;                                  // Endpunkt
17 };
18
19 //------------------------------------------------------------------------
20 //  Klasse kbogen
21 //------------------------------------------------------------------------
22 class kbogen : public bahn {
23
24   public:
25     kbogen  ( void )  {  }
26     kbogen  ( const Vek3 &p1, const Vek3 &p2, const Vek3 &p3 )
27     {
28       pa=p1; pm=p2; pe=p3;
29     }
30
31     void    zpunkt  ( const Vek3 &p ) { pm = p; }
32     double  laenge  ( void );
33
34   protected:
35     Vek3    pm;       // mittlerer Punkt auf dem Kreisbogen
36 };
```

Liste 4.2: Implementierung von kbogen::laenge (bahn1.cc)

```
1  double  kbogen::laenge  ( void )
2  {
3    double  radius, alpha, bogen;              // Hilfsgrößen
4    Vek3    v1, v2;                            // Hilfsgrößen
5
6    v1      = (pa-pm).norm();                  // normierter Vektor (1. Sehne)
7    v2      = (pe-pm).norm();                  // normierter Vektor (2. Sehne)
8    alpha   = acos(v1*v2);                     // Winkel zwischen den Sehnen
9    alpha   = 4.0*atan(1.0) - alpha;           // halber Winkel zwischen den Radien
10                                              //  zum Anfangs- und Endpunkt
11   radius  = 0.5*(pe-pa).laenge()/sin(alpha); // Radius = (halbe Sehne zw. Anfangs-
12                                              //  und Endpunkt)/sin(alpha)
13   bogen   = 2.0*alpha*radius;                // Bogen für den doppelten Winkel
14   return bogen;
15 }
```

Die neue Klasse `kbogen` besitzt als Erweiterung ihrer Basisklasse drei neue Komponenten:

`Vek3 pm;`	weiteres Datenelement
`void zpunkt ( const Vek3 &p )`	Setzen eines Zwischenpunktes
`double laenge ( void )`	Längenberechnung

Zur Darstellung eines Kreisbogens ist ein weiterer Punkt `pm` zwischen Anfangs- und Endpunkt erforderlich. Dadurch ist eine Methode zum Setzen dieses Punktes notwendig.
Für die richtige Berechnung der Bogenlänge kann nicht auf die Methode `laenge` der Basisklasse zurückgegriffen werden. Die Methode `laenge` der Klasse `kbogen` trägt aber den selben Namen und überlädt deshalb die entsprechende Methode der Basisklasse.

Liste 4.3: Verwendung der Klassen `bahn` und `kbogen` (`4.1-1.bahn-test.cc`)

```
1  int main()
2  {
3    double    bahnlaenge;
4    Vek3      p1, p2, p3;
5
6    bahn      bahn1;                  // Aufruf: bahn-Konstr.1
7    kbogen    bogen1;                 // Aufruf: bahn-Konstr.1, kbogen-Konstr.1
8
9    p1  = Vek3( 0, 1, 0 );
10   p2  = Vek3( 0, 0, 0 );
11   p3  = Vek3( 1, 0, 0 );
12
13   bahn1.start( p1 );                              // Methode der Basisklasse
14   bahn1.ziel ( p3 );                              // Methode der Basisklasse
15
16   bogen1.start ( p1 );                            // Methode der Basisklasse
17   bogen1.zpunkt( p2 );                            // Methode der Klasse kbogen
18   bogen1.ziel  ( p3 );                            // Methode der Basisklasse
19
20   bahnlaenge  = bahn1.laenge();                   // Methode der Basisklasse
21   cout  << "\nLänge Gerade       = " << bahnlaenge;
22
23   bahnlaenge  = bogen1.laenge();                  // Methode der Klasse kbogen
24   cout  << "\nLänge Kreisbogen 1 = " << bahnlaenge;
25   cout  << endl;
26
27   kbogen  bogen2(p1,p2,p3);                       // Aufruf: bahn-Konstr.1,
28                                                   //  kbogen-Konstr.2
29   bahnlaenge  = bogen2.laenge();                  // Methode der Klasse kbogen
30   cout  << "\nLänge Kreisbogen 2 = " << bahnlaenge;
31   cout  << endl << endl;
32
33   return 0;
34 }
```

Liste 4.3 zeigt die Verwendung der beiden Klassen. Die Anweisungen

```
bahn1.start( p1 );
bogen1.start ( p1 );
```

rufen jeweils die Methode `start` der Basisklasse auf, weil diese Methode für `kbogen` nicht überladen ist. Die Anweisung

```
bahnlaenge  = bahn1.laenge();
```

ruft die Methode `laenge` der Basisklasse auf, weil `bahn1` ein `bahn`-Objekt ist. Die Anweisung

```
bahnlaenge  = bogen1.laenge();
```

ruft hingegen die Methode `laenge` der abgeleiteten Klasse auf, weil `bogen1` ein `kbogen`-Objekt ist. Das Hauptprogramm liefert folgende Ausgaben:

```
Länge Gerade       = 1.41421
Länge Kreisbogen 1 = 2.22144

Länge Kreisbogen 2 = 2.22144
```

4.2. Initialisierung von Basisklassenkomponenten

Bei der Vereinbarung eines `kbogen`-Objekt (zum Beispiel Liste 4.3, Zeile 27)

```
kbogen  bogen2(p1,p2,p3);
```

wird der zweite Konstruktor der Klasse `kbogen` aufgerufen. Dieser Konstruktor hat folgende Implementierung

```
kbogen  ( const Vek3 &p1, const Vek3 &p2, const Vek3 &p3 )
{
  pa=p1; pm=p2; pe=p3;
}
```

Diese Implementierung übernimmt drei Vektoren und weist diese den entsprechenden Datenkomponenten zu. Ein Teil dieser Aufgabe, die Initialisierung des Anfangs- und Endpunktes, wird aber auch vom Konstruktor der Basisklasse `bahn` geleistet. Dieser Programmcode ist somit doppelt vorhanden, was schon aus Aufwands- und Sicherheitsgründen niemals wünschenswert ist. Wünschenswert ist aber, daß der Basisklassenkonstruktor bei der Initialisierung von Objekten einer abgeleiteten Klasse mitverwendet wird.

Wenn man nun alle Konstruktoren dieses Beispiels mit Ausgaben der Art

```
cout << "\nKlasse bahn   : Konstruktor 1";
```

ausstattet, dann stellt man fest, daß bei der Konstruktion eines `kbogen`-Objektes bereits der 1. Konstruktor der Basisklasse aufgerufen wird. Da dessen Implementierung leer ist (Liste 4.1, Zeile 7), besitzt er keine Wirkung.
Abhilfe bringt erst der ausdrückliche Aufruf des 2. Konstruktors der Basisklasse, verbunden mit der dann möglichen Umgestaltung des 2. Konstruktors der Klasse `kbogen`:

```
kbogen  ( const Vek3 &p1, const Vek3 &p2, const Vek3 &p3 )
          : bahn(p1,p3)
{
  pm = p2;
}
```

Der Aufruf `bahn(p1,p3)` des Basisklassenkonstruktors steht in einer sogenannte **Initialisierungsliste**, die durch einen Doppelpunkt getrennt hinter den Kopf des Konstruktors gestellt wird.
Diese Initialisierungsliste kann weitere Elemente enthalten. So kann in diesem einfachen Fall auch die noch verbleibende Zuweisung `pm = p2` in die Liste übernommen werden:

```
kbogen  ( const Vek3 &p1, const Vek3 &p2, const Vek3 &p3 )
          : bahn(p1,p3), pm(p2)
{ }
```

Wenn einzelne oder alle Datenkomponenten der Basisklasse **private** sind, dann ist ein Konstruktoraufruf in der Initialisierungsliste der abgeleiteten Klasse die einzige Möglichkeit, alle Datenkomponenten der Basisklasse zu initialisieren!

4.3. Virtuelle Funktionen

Die Vererbung in der bisherigen Form kann dazu verwendet werden, eine statische Klassenhierarchie aufzubauen. In einer derartigen Hierarchie könnten zum Beispiel Klassen für Geraden, Kreisbögen und Parabelstücke vorhanden sein (Abbildung 4.2).

Im nächste Schritt soll eine derartige Hierarchie für weitere Kurvenformen so erweiterbar sein, daß nach dem Hinzufügen einer neuen Form bereits bestehender Programmcode nicht mehr geändert werden muß.

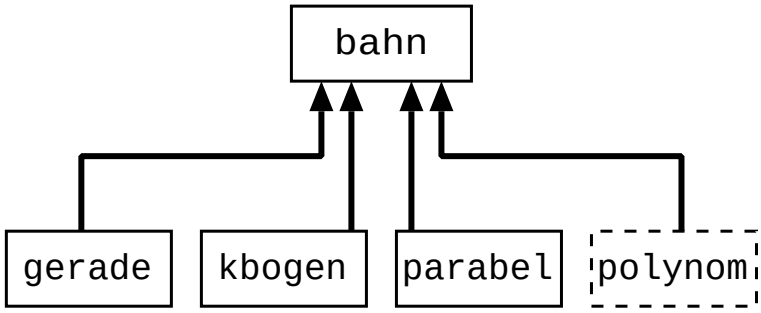


Abbildung 4.2.: Klassenhierarchie

Zur Verdeutlichung soll ein Beispiel betrachtet werden. In Abbildung 4.3 ist die Bewegungsbahn einer Bearbeitungsmaschine dargestellt. Eine derartige Bahn wird für jede Anwendung neu programmiert. Das heißt aber, daß die aufeinanderfolgenden Bahnobjekte (Geraden, Kreisbögen, ...) nicht durch den Programmierer der Steuerung, sondern durch den Anwender der Steuerung erzeugt werden.

Die Bahnobjekte müssen deshalb dynamisch mit Hilfe des Operators **new** angelegt werden. Die Zeiger auf diese Objekte werden dann zum Beispiel in einem Feld verwaltet:

```
bahn    *teilbahn[100];     // Feld von Zeigern auf Teilbahnen
```

Da der jeweilige Typ der im Feld `teilbahn` verzeigerten Teilbahnen bei der Implementierung natürlich noch nicht bekannt sein kann, besitzt das Feld den Datentyp `bahn*` (Zeiger auf Objekte der Basisklasse).

Bei der eigentlichen Bahnprogrammierung entstehen dann durch Bedienereingaben neue Bahnobjekte, die durch Anweisungen der Form

```
teilbahn[index++] = new kbogen( pn1, pn2, pn3 );
  ...
teilbahn[index++] = new parabel( pn1, pn2, pn3 );
```

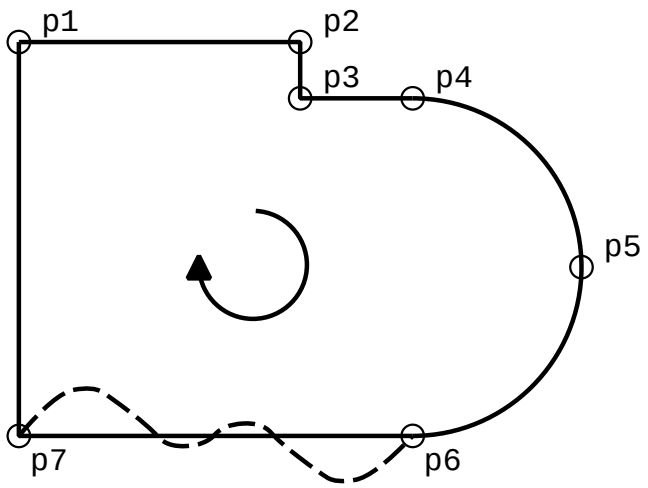


Abbildung 4.3.: Bearbeitungsbahn

in die Bewegungsfolge aufgenommen werden. Weiterhin sind bereits Funktionen vorhanden, die zum Beispiel die Gesamtbahnlänge aus den Längen der Teilbahnen berechnen:

```
double  bahnlaenge ( bahn *b[], int n )
{
  double  sum = 0.0;
  for(int i=0;i<n;i++)
    sum += b[i]->laenge();
  return  sum;
}
```

Die Funktion `bahnlaenge` erhält bei einem Aufruf ein derartiges Feld von Zeigern auf Basisklassenobjekte und ruft für jede Teilbahn in der **for**-Schleife die Längenberechnung auf.

Ohne Änderung an der Basisklasse wird in der Funktion `bahnlaenge` allerdings immer die Längenberechnung der Basisklasse aufgerufen! Die Längenberechnung eines eingezeigerten Kreisbogens wird durch die Berechnung der Sehnenlänge ersetzt und führt damit zum falschen Ergebnis.

Die Ursache liegt in der sogenannte **frühen Bindung**. Der Datentyp des Zeigerfeldes liegt zur Übersetzungszeit der Steuerung bereits fest. Damit liegt auch Klasse fest, zu der die aufzurufende Längenberechnung gehört.

Zur Behebung dieses Mangels wird ein Mechanismus benötigt, der erst zur Laufzeit des Programmes anhand der tatsächlich verzeigerten Teilbahn deren Typ feststellt und die Längenberechnung der zugehörigen Klasse aufruft. Dieser Aufrufmechanismus wird **späte Bindung** genannt.

Diese späte Bindung kann erzwungen werden, indem die Funktion `laenge` der Basisklasse als **virtuelle Funktion** erklärt wird:

```
virtual double  laenge  ( void )
{
  return 0;
}
```

Nach dieser Änderung wird in der Funktion `bahnlaenge` die zu jedem Teilbahntyp gehörige Längenberechnung aufgerufen; die Funktion liefert jetzt ohne weiteres die richtige Länge der

Gesamtbahn.
Das wird auch so bleiben, wenn die Hierarchie der Bahntypen erweitert wird. In Abbildung 4.3 ist angedeutet, daß die Teilbahn von Punkt `p6` nach `p7` durch ein Polynom ersetzt wird. Wenn die Klassenhierarchie (Abbildung 4.2) um eine abgeleitete Klasse `polynom` erweitert wird die ebenfalls eine Funktion `laenge` mitbringt, dann muß bereits vorhandener Programmcode, wie zum Beispiel die Funktion `bahnlaenge`, nicht mehr geändert werden.

Für die Verwendung virtueller Funktionen gelten weitere Regeln:

- Wenn eine abgeleitete Klasse eine virtuelle Funktion der Basisklasse *nicht* überlädt, dann erbt sie die Funktion der Basisklasse.
- Die Funktion der abgeleiteten Klasse, die die virtuelle Funktion der Basisklasse überladen soll, muß dieselbe **Signatur** (Name und Parameterliste) besitzen wie die zu überladende Funktion. Wenn das nicht der Fall ist, findet keine Überladung statt.

Das folgende Beispiel zeigt eine kleine Mißachtung der letzten Regel:

```
class bahn {
  ...
  virtual double  fkt_1 ( int n );
};

class gerade : public bahn {
  ...
          double  fkt_1 ( long n );
};
```

Die Funktion `fkt_1` der Klasse `gerade` überlädt keineswegs die virtuelle Funktion `fkt_1` der Basisklasse `bahn`, weil die Parameterlisten unterschiedlich sind!

4.4. Abstrakte Basisklassen

Im letzten Abschnitt wurde die Methode `bahn::laenge()` definiert:

```
virtual double  laenge  ( void )
{
  return 0;
}
```

Diese Funktion wird stets aufgerufen, wenn in einer abgeleiteten Klasse keine eigene Funktion `laenge()` vorhanden ist.
Wenn in der Basisklasse ausdrücklich auf eine Implementierung der Funktion `bahn::laenge()` verzichtet werden soll, aber gleichzeitig verlangt wird, daß diese Funktion in einer abgeleiteten Klasse durch Überladung vereinbart werden muß, dann wird diese Funktion in der Basisklasse als **rein virtuelle Funktion** vereinbart:

```
class bahn {
  ...
  virtual double  laenge  ( void ) = 0;
};
```

Diese Vereinbarung hat nur noch den Charakter einer Schnittstellendefinition für abgeleitete Klassen. Die Vereinbarung von `bahn`-Objekten ist jetzt nicht mehr möglich. Die Funktion `laenge()` muß in allen abgeleiteten Klassen überladen werden.

Liste 4.4: Abstrakte Basisklasse `bahn` (Ausschnitt der Datei `bahn3.hh`)

```
1  //------------------------------------------------------------------------
2  //  Klasse bahn (abstrakte Basisklasse)
3  //------------------------------------------------------------------------
4  class bahn {
5
6    public:
7      bahn    ( void )  { }
8      bahn    ( const Vek3 &p1, const Vek3 &p2 )  { pa=p1; pe=p2; }
9      virtual ~bahn() { }                          // virtueller Destruktor
10
11     void    start   ( const Vek3 &p ) { pa = p; }
12     void    ziel    ( const Vek3 &p ) { pe = p; }
13
14     virtual double  laenge  ( void ) = 0;       // rein virtuelle Funktion
15
16   protected:
17     Vek3    pa;                                  // Anfangspunkt
18     Vek3    pe;                                  // Endpunkt
19 };
20
21 //------------------------------------------------------------------------
22 //  Klasse gerade
23 //------------------------------------------------------------------------
24 class gerade : public bahn {
25
26   public:
27     gerade  ( void )  { }
28     gerade  ( const Vek3 &p1, const Vek3 &p2 ) : bahn(p1,p2) { }
29
30     double  laenge  ( void );
31 };
32
33 //------------------------------------------------------------------------
34 //  Klasse kbogen
35 //------------------------------------------------------------------------
36 class kbogen : public bahn {
37
38   public:
39     kbogen  ( void )  { }
40     kbogen  ( const Vek3 &p1, const Vek3 &p2, const Vek3 &p3 )
41       : bahn(p1,p3), pm(p2) { }
42
43     void    zpunkt  ( const Vek3 &p ) { pm = p; }
44     double  laenge  ( void );
45
46   protected:
47     Vek3    pm;       // mittlerer Punkt auf dem Kreisbogen
48 };
```

In Liste 4.4 sind die Definitionen dreier Klassen in einer Header-Datei zusammengefaßt. Die Definition der Klasse `bahn` zeigt eine weitere Besonderheit: der Destruktor dieser Klasse ist ebenfalls virtuell. In Abschnitt 4.3 wurde gezeigt, wie auf Objekte abgeleiteter Klassen mit Hilfe von Basisklassenzeigern zugegriffen werden kann. Diese dynamisch erzeugten Objekte müssen natürlich nach Gebrauch durch entsprechende **`delete`**-Aufrufe wieder beseitigt wer-

den. Wegen der Verwendung von Basisklassenzeigern würde ohne weitere Maßnahme in der folgenden Löschfunktion für jedes Objekt der Basisklassendestruktor aufgerufen:

```
void loesche_bahn ( bahn *teilbahn[], int nbahn )
{
  for ( int i = 0; i < nbahn; i += 1 )
    delete teilbahn[i];
}   // -----  end of function loesche_bahn  -----
```

Der Basisklassendestruktor kann jedoch nicht die in den abgeleiteten Klassen hinzugekommenen Datenkomponenten beseitigen! Das wird erreicht, indem auch für die Destruktoraufrufe, durch Hinzufügen des Schlüsselwortes **virtual**, die späte Bindung erzwungen wird. Jetzt werden für alle Objekte abgeleiteter Klassen deren Destruktoren aufgerufen. Nur diese Destruktoren sind in der Lage, die hinzugekommen Datenkomponenten zu beseitigen.

4.5. Zugriffsschutz

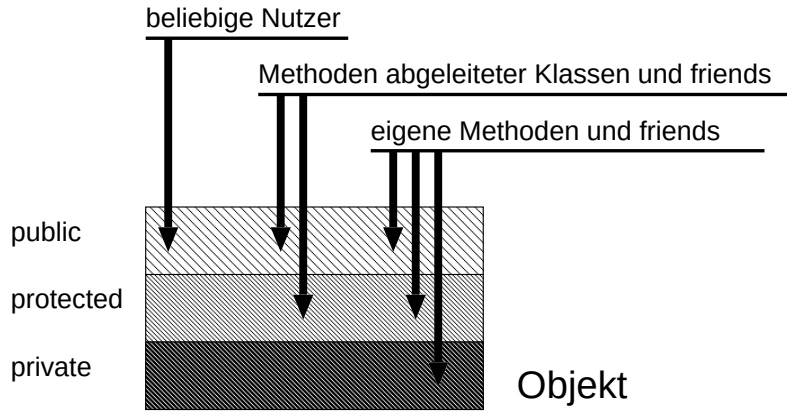


Abbildung 4.4.: Zugriffskontrolle bei einem Objekt

Die Bestandteile einer Klasse können `private`, `protected` oder `public` sein.

`private` Diese Komponenten können nur von den eigenen Methoden und von den friends der Klasse verwendet werden, in der diese `private`-Komponenten deklariert sind.

`protected` Diese Komponenten können von den eigenen Methoden und von den friends der Klasse verwendet werden, in der diese `private`-Komponenten deklariert sind. Außerdem haben die Methoden und die friends abgeleiteter Klassen Zugriff.

`public` Diese Komponenten können von allen anderen Funktionen aus aufgerufen werden.

4.5.1. `public`-Vererbung

Die am häufigsten verwendete Vererbungsart ist die **`public`**-Vererbung (Abschnitt 4.1, Abbildung 4.5). Die **`public`**-Komponenten von Klasse `A` sind Teil der **`public`**-Komponenten von Klasse `B`. Das Entsprechende gilt für die **`protected`**-Komponenten. Die **`private`**-Komponenten von `A` sind für `B`-Methoden nicht zugänglich. Das Gesagte gilt auch dann, wenn die Klasse `A` ihre Bestandteile selbst von einer anderen Basisklasse geerbt hat.

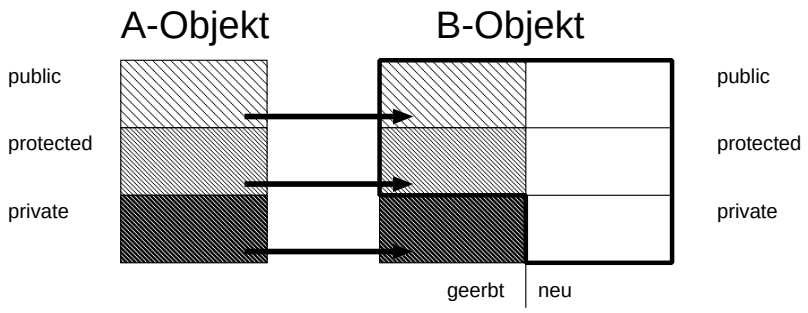


Abbildung 4.5.: `public`-Vererbung

4.5.2. `protected`-Vererbung

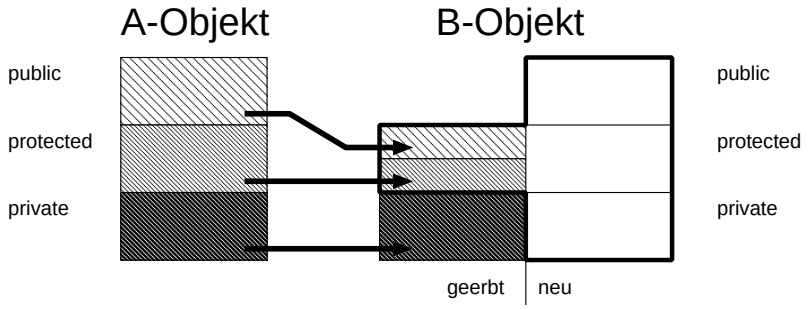


Abbildung 4.6.: `protected`-Vererbung

Bei der **`protected`**-Vererbung wird bei der Definition der abgeleiteten Klasse das Schlüsselwort **`protected`** verwendet:

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

Die **`public`**- und die **`protected`**-Komponenten der Basisklasse `A` kommen dabei in den **`protected`**-Teil der neuen Klasse `B` (Abbildung 4.6). Die öffentliche Schnittstelle der Klasse `A` ist damit nicht mehr Bestandteil der öffentliche Schnittstelle der Klasse `B`. Die **`public`**- und **`protected`**-Komponenten von `A` sind nur noch für `B`-Methoden und **`friend`**-Funktionen erreichbar.

4.5.3. `private`-Vererbung

Bei der **`private`**-Vererbung wird bei der Definition der abgeleiteten Klasse das Schlüsselwort **`private`** verwendet:

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

Die **`public`**-, **`protected`**- und die **`private`**-Komponenten der Basisklasse `A` kommen dabei in den **`private`**-Teil der neuen Klasse `B` (Abbildung 4.7). Die öffentliche Schnittstelle der Klasse `A` ist damit nicht mehr Bestandteil der öffentliche Schnittstelle der Klasse `B`. Bestandteil der öffentliche Schnittstelle der Klasse `B`. Die **`public`**- und **`protected`**-Komponenten von `A` sind nur noch für `B`-Methoden und **`friend`**-Funktionen erreichbar.

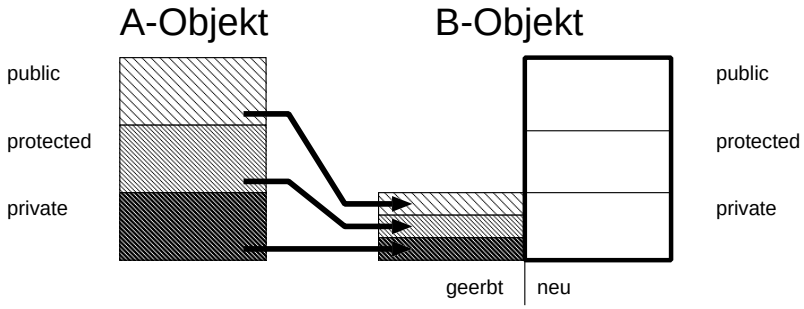


Abbildung 4.7.: `private`-Vererbung

5. Fehlerbehandlung

Es gibt mehrere gebräuchliche Möglichkeiten, Fehler in einer Funktion oder in einer Anwendung zu behandeln. Alle haben Vorteile und Schwächen.

5.1. Rückgabewerte

Der Rückgabewert einer Funktion kann zur Übermittlung einer Zustandsanzeige an den Aufrufer verwendet werden, wenn sonst keine Daten zurückzugeben sind oder diese auf anderem Wege weitergereicht werden (zum Beispiel über Funktionsparameter oder globale Größen).
Eine Funktion zur Lösung eines linearen Gleichungssystems könnte zum Beispiel den Wert 0 zurückgeben, wenn eine numerisch hinreichend genaue Lösung gefunden wurde. War die Koeffizientenmatrix singulär, dann wird eine 1 zurückgeben und so weiter. Der Aufrufer ist für die Überprüfung des Rückgabewertes verantwortlich. Viele Funktionen der ***C***-Bibliothek verwenden den Rückgabewert in dieser Weise. Eine typische Anwendung ist das Schließen einer Datei:

```
FILE  *p;                           /* input-file pointer */
char  *p_file_name = "input.dat";   /* input-file name    */

/*  ...
 */

if( fclose(p) == EOF ) {      /* close input file   */
  fprintf ( stderr, "couldn't close file '%s'; %s\n",
      p_file_name, strerror(errno) );
  exit (EXIT_FAILURE);
}
```

Bei Erfolg gibt die Funktion `fclose` 0 zurück. Anderenfalls wird `EOF` zurückgegeben und die globale Variable `errno` gesetzt, um den Fehler anzuzeigen.

Ein Nachteil bei der Verwendung von Rückgabewerten besteht darin, daß deren Überprüfung durch den Programmierer unterbleiben kann. Wenn sie stattfindet, dann weitet sie den Kontrollfluß an dieser Stelle auf. Ein weiterer Nachteil: Konstruktoren und einige überladene Operatoren haben keine Rückgabewerte und müßten deshalb ohnehin andere Vorgehensweisen verwenden.

5.2. Verwendung von `assert`

Ein Klassiker der Fehlerbehandlung ist die Verwendung des Makros `assert` (Header-Datei `assert.h` in ***C***-Projekten, Header-Datei `cassert` in ***C++***-Projekten). Die Verwendung von `assert`, zum Beispiel in

```
assert( next < max );                       // Überlauf ?
```

überprüft die angegebene Bedingung. Das Programm bricht mit einer Fehlermeldung ab, wenn die Bedingung (hier: `next < max`) nicht erfüllt ist. Die Fehlermeldung enthält den Dateinamen der Programmquelle, den Funktionsnamen und die Zeilennummer des Aufrufs. Bei einem vollständig getesteten Programm sollten diese Überprüfungen nicht mehr notwendig sein. In auslieferungsfähigen Programmversionen kann die Wirkung des Makros aufgehoben werden, wenn vor dem Einbinden von `assert.h` das Makro `NDEBUG` definiert wird.

5.3. Ausnahmen

Es gibt viele Anwendungen, in denen die oben genannten Vorgehensweisen nicht geeignet oder zu wenig anpassungsfähig sind. Manche Anwendungen können oder sollen zum Beispiel nicht einfach abgebrochen werden, während bei anderen nach genauer Fehleruntersuchung noch sinnvolle Gegenmaßnahmen eingeleitet werden können.

C++ bietet die Möglichkeit, an Stellen, an denen Fehler erkannt werden, sogenannte Fehlerobjekte auszuwerfen.
Programmabschnitte in denen Fehler auftreten können, werden zur Überwachung in **`try`**-Blöcken ausgeführt.
Einem **`try`**-Block folgt mindestens ein **`catch`**-Block, in dem der erkannte Fehler behandelt wird. Können in dem überwachten Abschnitt Fehler auftreten die zum Auswerfen unterschiedlicher Fehlerobjekte führen, dann kann für jede Fehlerart ein **`catch`**-Block vorhanden sein.

Einige Methoden der Klasse `Stack` in Liste 2.1 werden mit dem `assert`-Makro überwacht. Das soll nun durch den Ausnahme-Mechanismus ersetzt werden. Dazu werden Fehlernummern eingeführt, die im einfachsten Fall als globale Größen eingerichtet werden können (das ist die einfachste Art von Fehlerobjekten). Die betroffenen Methoden werfen im Fehlerfall einen ganzzahligen Wert vom Typ **`int`** aus. Die Änderungen, die an dem Programm in Liste 2.1 vorzunehmen sind, sehen am Beispiel der Methode `push` wie folgt aus:

```
static const int StackErrorUnderflow  = 1;       // Wert eines Fehlerobjektes
static const int StackErrorOverflow   = 2;       // Wert eines Fehlerobjektes

  // Definition der Klasse Stack
  // Methoden
  //  ...

//------------------------------------------------------------------------
//  push
//------------------------------------------------------------------------
void  Stack :: push ( int datum )
{
  if ( next >= max )
    throw int( StackErrorOverflow );            // Überlauf
  value[next++] = datum;
}
```

Die **`if`**-Anweisung überprüft, ob der Stapel beim Aufruf der Methode `push` bereits voll ist. In diesem Fall wirft die Anweisung **`throw`** ein **`int`**-Objekt aus, das an dieser Stelle erzeugt und mit dem konstanten Wert von `StackErrorOverflow` initialisiert wird (Konstruktoraufruf für ein **`int`**-Objekt). Die Methode `push` wird dann an dieser Stelle sofort verlassen!

Für die weitere Fehlerbehandlung ist der Aufrufer der Methode verantwortlich. Erzeugte Fehlerobjekte können aufgefangen werden, wenn die Methodenaufrufe, die solche Objekte erzeu-

gen können, in einem überwachten Abschnitt aufgerufen werden. Die allgemeine Form dieser Überwachungen sieht wie folgt aus:

```
try
{
  beliebige Anweisungen
}
catch ( Datentyp1 Argumentname )
{
  Behandlung der Ausnahme vom Typ Datentyp1
}
catch ( Datentyp2 Argumentname )
{
  Behandlung der Ausnahme vom Typ Datentyp2
}
//
// evtl. weitere catch-Blöcke ...
//
catch (...)
{
  Behandlung aller anderen Ausnahmen
}
```

Liste 5.1: Auffangen einer Ausnahme (5.3-1.stack+exception.cc)

```
1  //------------------------------------------------------------------------
2  //  Hauptprogramm
3  //------------------------------------------------------------------------
4  int main ( int argc, char *argv[] )
5  {
6    int   n = 10;                                  // Stapelgröße
7    Stack stapel1( n );                            // 1. Stapel
8    int   datum;                                   // Hilfsgröße
9
10   for( int i=0; i<n; i++ )                       // Stapel vollständig belegen
11     stapel1.push( i );
12
13   try
14   {
15     datum = stapel1.pop();                       // Element entnehmen
16     stapel1.push(datum);                         // Element hinzufügen
17     stapel1.push(datum);                         // Element hinzufügen ! Überlauf !
18   }
19   catch ( int &ExceptObj )                       // Ausnahmebehandlung
20   {
21     cout  << "\nstapel1 : Fehlercode = " << ExceptObj << endl;
22     exit( ExceptObj );
23   }
24
25   return 0;
26 }       // ----------  end of function main  ----------
```

Liste 5.1 zeigt die Anwendung in einem Hauptprogramm. Nachdem der Stapel vollständig belegt wurde, werden einige Operationen in einem **try**-Block ausgeführt. Der zweite Aufruf der Methode `push` erzeugt einen Überlauf (Zeile 17). Das von der Methode `push` ausgeworfene Fehlerobjekt wird aufgefangen. Der nachstehende **catch**-Block besitzt eine Parameterliste, die zum Datentyp des Fehlerobjektes paßt. Deshalb führt dieser Block die Ausnahmebehandlung durch. Dies besteht in diesem Beispiel lediglich darin, eine Fehlermeldung auszugeben und das Programm mit der Fehlernummer aus Rückgabewert zu verlassen. Hier wären natürlich beliebig aufwendige Fehlerbehandlungen möglich.
Wenn die Überwachung durch einen **try**-Block fehlt, bricht daß Programm mit einer Fehlermeldung ab. Das ist ein ganz wesentlicher Unterschied zur Verwendung von Rückgabewerten zur Fehleranzeige (Abschnitt 5.1). Der Aufrufer wird nun gezwungen eine Fehlerbehandlung einzurichten, wenn ein Methodenaufruf eine Ausnahme auswerfen *kann*.

5.3.1. Eigene Fehlerklassen

Im letzten Abschnitt wurden Fehlerobjekte vom Typ **int** verwendet. Für umfangreichere Projekte und Fehlerbehandlungen können und sollten jedoch Fehlerobjekte selbst definierter oder bereits vorhandener Fehlerklassen verwendet werden. Damit werden Verwechslungen von Fehlerobjekten mit Nicht-Fehlerobjekten ausgeschlossen.
Die folgende Klasse `StackError` dient nur dazu, auf einfache Weise Fehlerobjekte zu erzeugen, die aus Zeichenketten bestehen und die somit gleich als Meldungstexte verwendet werden können.

```
class StackError {
  public:     StackError ( string msg = "StackError" ) : message(msg)  { }
              virtual ~StackError ( )  { }
              virtual string what ( ) const throw ( )  { return message; }
  protected:  string  message;
}; // ----------  end of class  StackError  ----------
```

Die Klasse besitzt nur drei Methoden und ist offensichtlich als Basisklasse für weitere, abgeleitete Fehlerklassen eingerichtet. Der Konstruktor übernimmt beim Aufruf einen Text als Parameter (oder den Ersatzwert) und weist diesen der einzigen Datenkomponente `message` zu (Initialisierungsliste). Die virtuelle Methode `what` gibt den Text des jeweiligen Fehlerobjekt zurück.
In einer Stapel-Klasse kann diese Fehlerklasse zur Behandlung von Stapelfehlern dienen. So könnte zum Beispiel die Funktion `top` (Abfrage des obersten Datenelementes auf einem Stapel) jetzt wie folgt aussehen:

```
int Stack :: top  () {
  if ( next<=0 )
    throw StackError("class  Stack - top : stack is empty");
  return value[next-1];
}
```

Wenn der Stapel leer ist, dann wird mittels **throw** ein Fehlerobjekt vom Typ `StackError` erzeugt und ausgeworfen. Dazu wird der `StackError`-Konstruktor aufgerufen, dem beim Aufruf eine Fehlerbeschreibung als aktueller Parameter mitgegeben wird. Die anderen Methoden der Klasse `Stack` sind mit entsprechenden Anweisungen zu versehen.
Hier ein überwachter Abschnitt, der gegebenenfalls eine Fehlerbehandlung einleitet:

```
try {
  top = stapel3.top();                          // Zugriff auf das oberste Element
```

```
}
catch ( StackError &ExceptObj ) {              // ggf. Fehlerobjekt fangen
  cerr  << ExceptObj.what() << endl;
}
```

Sollte der Stapel `stapel3` leer sein und trotzdem das oberste Element abgefragt werden, dann wird in `top` ein Fehlerobjekt ausgeworfen und im nachfolgenden **`catch`**-Block abgefangen. Dieser Block hat einen formalen Parameter mit Namen `ExceptObj`. Mit `ExceptObj.what()` kann somit die Methode `what` des erhaltenen Fehlerobjektes aufgerufen werden. Der Aufruf wird hier lediglich dazu benutzt, die Fehlerbeschreibung an eine Ausgabeanweisung weiterzugeben.

Sowohl die Überwachung mit **`try`** als auch das Fangen von Fehlern mit **`catch`** kann geschachtelt werden. Mit Hilfe von **`throw`** können bereits gefangene Fehlerobjekten wieder ausgeworfen werden, um Fehlerbehandlungen auf höheren Ebenen zu bedienen. Die Anweisung

```
throw;
```

innerhalb eines **`catch`**-Blockes reicht die gerade zu behandelnde Ausnahme an den nächsthöheren Block weiter.

Für eine Reihe von immer wiederkehrenden Fehlern gibt es Standard-Fehlerobjekte. Der nächste Abschnitt geht kurz darauf ein.

5.3.2. Standardausnahmen

Die Methoden der Standardbibliothek werfen an verschiedenen Stellen ebenfalls Ausnamenobjekte aus. Die zugrunde liegenden Ausnahmeklassen sind als Klassenhierarchie aufgebaut (Abbildung 5.1). Die Namen der Klassen weisen auf deren Bedeutung hin. Die Behandlung dieser Klassen kann in diesem Kurs nicht erfolgen. Im Folgenden soll aber kurz auf die Verwendung der Standardausnahme `bad_alloc` zur Behandlung von Fehlern bei der dynamischen Speicherbelegung durch **`new`** eingegangen werden.

Schlägt die Beschaffung von Speicherplatz mittels **`new`** fehl, dann wird von diesem Operator ein Fehlerobjekt vom Typ `bad_alloc` ausgeworfen. Dieses kann wie folgt gefangen werden:

```
try {
  // ** Speicherbeschaffung mit new
}
catch ( std::bad_alloc ) {
  // ** Fehlerbehandlung **
}
```

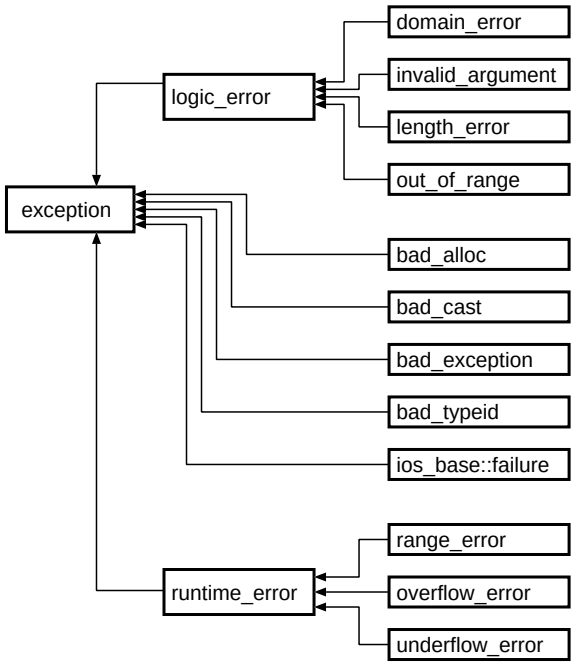


Abbildung 5.1.: Die Hierarchie der Standardausnahmenklassen

Liste 5.2 zeigt die Anwendung. Die Funktion `block2` beschafft bei jedem Aufruf zwei **`int`**-Felder. Die Beschaffung kann wegen Speicherplatzmangels fehlschlagen. Einer der Operatoren **`new`** in `block2` wird dann ein `bad_alloc`-Objekt auswerfen, das gleich im nachfolgenden **`catch`**-Block

behandelt wird. Schlägt das erste **new** fehl, dann ist nichts aufzuräumen. Die **delete**-Anweisung bleibt wirkungslos, weil der Zeiger `feld1` mit Null initialisiert wurde. Schlägt das zweite **new** fehl, dann muß das erste Feld wieder beseitigt werden. Anschließend wird die Ausnahme an den Aufrufer (hier `main`) zur weiteren Behandlung weitergegeben.

Liste 5.2: Auffangen einer Ausnahme vom Typ `std::bad_alloc` (`5.3-3.new-failed.cc`)

```
1  using namespace std;
2
3  #include <iostream>
4  #include <cstdlib>
5
6  //------------------------------------------------------------------------
7  //  Funktion block2
8  //------------------------------------------------------------------------
9  void
10 block2 ( size_t size )
11 {
12   int *feld1  = 0;
13   int *feld2  = 0;
14
15   try {
16     feld1 = new int[size];                    // Speicherplatzbeschaffung
17     feld2 = new int[size];                    // Speicherplatzbeschaffung
18   }
19   catch (...) {                               // Ausnahmebehandlung
20     delete [] feld1;                          // Speicherfreigabe
21     feld1 = 0;
22     throw;                                    // Ausnahme weitergeben
23   }
24   //
25   // ... Speicherplatz verwenden
26   //
27   delete [] feld1;                            // Speicherfreigabe
28   delete [] feld2;                            // Speicherfreigabe
29 }   // -----  end of function block2  -----
30
31 //------------------------------------------------------------------------
32 //  Hauptprogramm
33 //------------------------------------------------------------------------
34 int
35 main ( int argc, char *argv[] )
36 {
37   size_t  size  = 2L*1000*1000*1000;
38
39   try {
40     block2( size );                           // überwachter Aufruf
41   }
42   catch ( std::bad_alloc ) {                  // Ausnahmebehandlung
43     cout  << "*** Speicherplatzbeschaffung fehlgeschlagen ***\n";
44     exit(EXIT_FAILURE);
45   }
46   return EXIT_SUCCESS;
47 }       // ----------  end of function main  ----------
```

5.3.3. Ausnahmefester Code

Ein wesentliches Ziel bei der Verwendung von Ausnahmen ist die Erstellung von ausnahmefestem Code (engl. exception-safe code). Beim Auswerfen einer Ausnahme springt die Programmausführung von der Auswurfstelle in einen behandelnden **`catch`**-Block (falls vorhanden). Dadurch können Obkjekte in einem inkonsistenten Zustand zurückgelassen werden.
Ein Konstruktor, der mit **`new`** mehrere Speicherblöcke initialisieren muß, dabei aber unterbrochen wird, läßt das Objekt in einem nicht definierten Zustand zurück. Ein Destruktor, der bei der Speicherfreigabe und anderen Aufräumarbeiten unterbrochen wird, kann nicht mehr zugreifbare Speicherblöcke (Speicherlecks), offene Dateien oder falsch gesetzte Synchronisationsgrößen zurücklassen. Für eine umfassende Behandlung dieser Aufgabenstellung muß hier auf die Literatur verwiesen werden (zum Beispiel [SDT06], [Mey05]).

Fehlerbehandlung durch Verwendung von Ausnahmen ist auch dann nicht kostenlos, wenn in einem laufenden Programm nie Fehler auftreten. Alle Funktionen müssen Informationen für den exception handler bereitstellen, da in einem Fehlerfall der Aufrufstack abgebaut werden muß (stack unwinding) und bei dieser Gelegenheit die Destruktoren der Speicherobjekte aufgerufen werden müssen, die auf dem Weg zur Fehlerstelle angelegt wurden.

6. Templates

6.1. Funktionstemplates

In Abschnitt 1.8 wurde am Beispiel der Funktion `swap` die Überladung von Funktionsnamen eingeführt. Hier nochmals die **`int`**-Version und die **`double`**-Version:

```
void  swap ( int *a, int *b )                    // int-Version
{
  int     h = *a;
         *a = *b;
         *b =  h;
}

void  swap ( double *a, double *b )              // double-Version
{
  double  h = *a;
         *a = *b;
         *b =  h;
}
```

Die beiden Funktionen unterscheiden sich offensichtlich nur durch das jeweils dreimal verwendete Schlüsselwort für den jeweiligen Datentyp. Weitere Funktionen für zusätzliche Datentypen würden sich ebenfalls genau darin unterscheiden.

Es liegt deshalb nahe, für alle derartigen Funktionen eine Schablone festzulegen, und die tatsächliche Implementierung einer Funktion für einen gegebenen Datentyp (zum Beispiel **`int`**) aus der Schablone abzuleiten. Eine derartige Schablone wird in ***C++*** als **Template** bezeichnet und wie folgt vereinbart:

```
template <class T> void swap  ( T *a, T *b )
{
  T h  = *a;
   *a  = *b;
   *b  =  h;
}   // -----  end of template function swap  -----
```

Diese Definition hat folgende Bestandteile:

`template <class T>`	Schlüsselwort `template`, gefolgt von einem formalen Parameter `T` für den zu verwendenden Datentyp
`void swap`	Rückgabetyp und Name der Funktion
`T *a`	1. Argument: Zeiger auf ein Objekt vom Datentyp `T` (2. Argument entsprechend)
`T h`	Hilfsgröße vom Datentyp `T`

Für Templates gelten einige zusätzliche Festlegungen:

- Templates sind Schablonen für eine Menge möglicher Funktionen. Eine Template-Definition erzeugt selbst keinen Programmcode und wird deshalb in größeren Projekten stets in Header-Dateien geschrieben.
- Erst wenn in einer Anwendung entsprechende Funktionsaufrufe erscheinen, wird der zugehörige Programmcode vom Compiler aus dem Template selbsttätig erzeugt. Für den Aufruf

```
int a, b;
  ...
swap ( &a, &b );
```

  ist dies aus der oben angegebenen Definition möglich, da beide Aufrufparameter denselben Datentyp besitzen. Der Aufruf

```
int     a;
double  x;
  ...
swap ( &a, &x );
```

  führt zu einem Übersetzungsfehler, weil die Aufrufparameter unterschiedliche Datentypen besitzen, im Template aber zwei gleiche Parameter vorgesehen sind.
- Templates die mehrere Datentypen verwenden sind ebenfalls möglich. In diesem Fall werden die vorläufigen Bezeichnungen für diese Datentypen in einer Liste nach dem Schlüsselwort **template** angegeben:

```
template <class T1, class T2>
void swap ( T1 &a, T2 &b )
{
  T1 h  =     a;
     a  = (T1)b;
     b  = (T2)h;
}   // -----  end of template function swap  -----
```

  Bei den Zuweisungen werden hier explizite Typumwandlungen durchgeführt. Das folgende Beispiel ist jetzt ohne weiteres möglich:

```
int     a =    9;
double  x = -2.5;
  ...
swap ( a, x );
```

  Durch die Verwendung von Referenzen in der Parameterliste entfällt die Adreßbildung beim Aufruf. Wegen der Typumwandlung hat die Variable `a` nach dem Aufruf den Wert -2.

6.2. Klassentemplates

Templates sind natürlich auch für ganze Klassen sinnvoll. In Liste 2.1 wird eine Implementierung einer Stapelklasse gezeigt, die für den Datentyp **int** ausgelegt ist. Stapel sind auch für andere Datenobjekte sinnvoll. Die Implementierung für den Datentyp **double** würde sich von der vorliegenden nur an wenigen Stellen durch das andere Schlüsselwort unterscheiden.

Um nun eine Template-Klasse für Stapel zu definieren, muß offenbar der Datentyp an den maßgeblichen Stellen der Definition durch eine Variable ersetzt werden. Im Falle unseres Stapels sieht der Rahmen wie folgt aus:

```
template <class T> class Stack
{
  public:
    ...
  private:
    T     *value;                  // Stapel zur Aufnahme der Daten
    int   next;                    // Stapelzeiger ( nächstes freies Element)
    int   max;                     // Stapelgröße
};        // -----  end of template class  Stack  -----
```

Die Definition beginnt also stets mit dem Schlüsselwort **template**. Darauf folgt in spitzen Klammern die Angabe eines oder mehrerer formaler Parameter, in der Regel vorläufige Namen für zu verwendende Klassen oder Datentypen (hier **class** T). Der Datentyp des Zeigers für die im Stapel zu verwaltenden Daten ist dementsprechend T*.
Die beiden Größen next und max bleiben selbstverständlich vom Typ **int**, da sie unabhängig vom Datentyp der verwalteten Objekte immer einen ganzzahligen Feldindex und dessen Obergrenze darstellen.

Die Parameter und Rückgabetypen, sowie weitere Hilfsgrößen, müssen in der Implementierung der Methoden nun den Datentyp T erhalten, sofern sie sich auf ein im Stapel befindliches Datenobjekt beziehen.
Liste 6.1 zeigt die vollständige Definition der Klasse, Liste 6.2 zeigt die zugehörige Implementierung.

Liste 6.1: Template-Klasse Stack, Definition (6.2-1.stack-template.cc)

```
1  template <class T> class Stack
2  {
3    public:
4      // ====================  LIFECYCLE   ========================================
5      Stack ( int groesse = 1000 );                    // constructor
6      Stack ( const Stack &other );                    // copy constructor
7      ~Stack ();                                       // destructor
8
9      // ====================  OPERATORS   ========================================
10     const Stack& operator = (const Stack &other);    // assignment operator
11
12     // ====================  OPERATIONS  ========================================
13     void  Push          ( T datum );                 // Element hinzufügen
14     T     Pop           ( );                         // Element entfernen
15     void  Init          ( ) { next = 0; };           // Initialisierung
16
17     // ====================  ACCESS      ========================================
18     T     Top           ( );                         // oberstes Element
19     int   Hight         ( ) { return next; };        // Anzahl der Elemente
20     int   MaxHight      ( ) { return max; };         // max. Anzahl der Elemente
21
22     // ====================  INQUIRY     ========================================
23     bool  IsEmpty       ( ) { return next==0; };     // Test, ob Stack leer
24     bool  IsNotEmpty    ( ) { return next!=0; };     // Test, ob Stack nicht leer
25
26   private:
27     T     *value;           // Stapel zur Aufnahme der Daten
28     int   next;             // Stapelzeiger (zeigt auf das nächste freie Element)
29     int   max;              // Stapelgröße
30 };        // ~~~~~  end of class  Stack  ~~~~~
```

Liste 6.2: Template-Klasse `Stack`, Implementierung (`6.2-1.stack-template.cc`)

```
1  template < class T >
2  Stack<T> :: Stack ( int groesse )
3  {
4    assert( groesse > 0 );            // Mindestgröße 1
5    max   = groesse;                  // max. Größe übernehmen
6    value = new T[max];               // dyn. Speicher belegen
7    next  = 0;                        // Zeiger auf den Feldanfang stellen
8  }
9
10 //--------------------------------------------------------------------------
11 // Klasse Stack : Kopierkonstruktor
12 //--------------------------------------------------------------------------
13 template < class T >
14 Stack<T>::Stack ( const Stack &other )
15 {
16   max     = other.max;              // max. Größe übernehmen
17   value   = new T[max];             // dyn. Speicher belegen
18   next    = other.next;             // Zeiger übernehmen
19   for( int i=0; i<next; i++ )       // Datenfeld kopieren
20     value[i]  = other.value[i];
21 }
22
23 //--------------------------------------------------------------------------
24 // Klasse Stack : Destruktor
25 //--------------------------------------------------------------------------
26 template < class T >
27 Stack<T> :: ~Stack  ( )
28 {
29    delete [] value;                 // dyn. Speicher freigeben
30 }
31
32 //--------------------------------------------------------------------------
33 // Klasse Stack :  Push
34 //--------------------------------------------------------------------------
35 template < class T >
36 void Stack<T> :: Push ( T datum )
37 {
38   assert( next < max );             // Überlauf ?
39   value[next++] = datum;
40 }
41
42 //--------------------------------------------------------------------------
43 // Klasse Stack :  Pop
44 //--------------------------------------------------------------------------
45 template < class T >
46 T Stack<T> :: Pop ()
47 {
48   assert( next > 0);                // Unterlauf ?
49   return value[--next];
50 }
51
52 //--------------------------------------------------------------------------
53 // Klasse Stack :  Top
54 //--------------------------------------------------------------------------
55 template < class T >
```

```
56 T Stack<T> :: Top ()
57 {
58   assert( next > 0 );               // Unterlauf ?
59   return value[next-1];
60 }
61
62 //------------------------------------------------------------------------
63 // Klasse Stack :  operator =
64 //------------------------------------------------------------------------
65 template < class T >
66 const Stack<T>&
67 Stack<T>::operator = ( const Stack &other )
68 {
69   if ( this != &other )             // Objekt nicht auf sich selbst kopieren
70   {
71     delete [] value;                // eigenen dyn. Speicher freigeben
72     max   = other.max;              // max. Größe der rechten Seite übernehmen
73     value = new T[max];             // dyn. Speicher neu belegen
74     next  = other.next;             // akt. Stapelzeiger übernehmen
75     for (int i = 0; i < next; i++)
76       value[i] = other.value[i];    // Stapel kopieren
77   }
78   return  *this;
79 }
```

Bei der Implementierung ist jeder Methode der Stapelklasse der Zusatz `template < class T >` voranzustellen. Der Klassenname ist beim Scope-Operator mit Parameterliste anzugeben, hier also `Stack<T>`. Die Implementierung des Kopierkonstruktors (siehe auch Liste 6.2) erhält somit das folgende Aussehen:

```
template < class T >
Stack<T>::Stack (const Stack &other)
{
    ...
}
```

Die folgenden Zeilen zeigen den Anfang eines Hauptprogrammes. Zu Beginn werden drei Stapel in der selben Weise eingerichtet, wie das bei gewöhnlichen Klassen geschieht. Der wesentliche Unterschied besteht darin, daß für den formalen Parameter `T` der Klassendefinition nun ein bereits bekannter Datentyp oder eine bereits definierte Klasse in spitzen Klammern angegeben werden muß.

```
int main ( int argc, char *argv[] )
{
  Stack<double>   stapel1 (500);
  Stack<int>      stapel2;
  Stack<double>   stapel3 = stapel1;
    ...
  return 0;
}       // ----------  end of function main  ----------
```

6.3. Container-Klassen

Eine Container-Klasse ist eine Klasse, die beliebige andere Objekte verwalten kann und für diese Objekte Grundoperationen zur Verfügung stellt. Die Anzahl der Objekte innerhalb einer Container-Klasse sollte durch deren Aufbau nicht beschränkt sein.

Der wichtigste Gedanke dabei ist, immer wieder verwendete Datenstrukturen als Container-Klassen anzulegen: Vektoren, einfach und doppelt verkettete Listen, Mengen, Warteschlangen und andere. Um eine gute Wiederverwendbarkeit zu erzielen und dem Programmierer die Arbeit zu erleichtern, werden Container-Klassen bereits mit einer Reihe Grundoperationen ausgestattet, zum Beispiel

- Zugriff auf das erste und das letzte Element,
- Kopieren eines Containers,
- dynamische Erweiterung,
- Einfügen, Löschen, Überschreiben, Verschieben von Elementen und
- Durchlaufen des gesamten Containers.

Dadurch entstehen *generische Datenstrukturen* (das heißt typunabhängige Datenstrukturen), die für viele Anwendungen nur einmal geschrieben werden müssen. Abhängig von der Art des Containers können auch generische Algorithmen hinzugefügt werden, die die typunabhängigen Teile immer wiederkehrender Bearbeitungsvorgänge, wie zum Beispiel Sortieren, Suchen, Vergleichen und so weiter implementieren. Dem Anwender verbleibt dann nur noch die Anpassung dieser generischen Datenstrukturen und Algorithmen an seine eigentliche Aufgabe.

6.3.1. Eine Container-Klasse für Listen

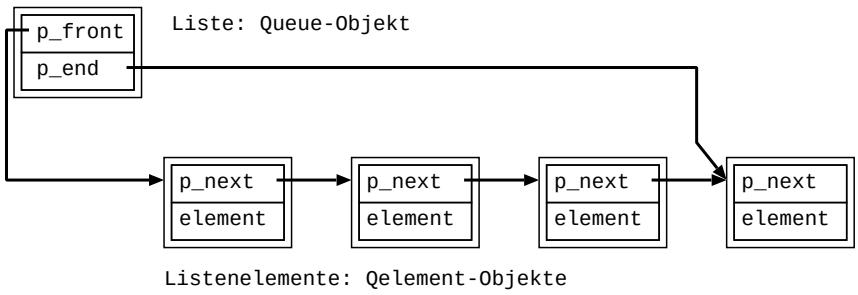


Abbildung 6.1.: Einfach verkettete Liste

In diesem Abschnitt wird die grundsätzliche Vorgehensweise zur Erstellung einer Container-Klasse gezeigt. Für die Implementierung werden Klassentemplates verwendet. Abbildung 6.1 zeigt eine einfach verkettete Liste. Der Listenkopf besteht aus einem Objekt einer Template-Klasse `Queue`, die Listenelemente bestehen aus Objekten einer Template-Klasse `Qelement`.

In der Definition der Template-Klasse `Qelement` in Liste 6.3 sind nur die beiden Datenelemente und der Konstruktor enthalten. Dieser übernimmt lediglich einen Initialisierungswert und setzt den Zeiger auf ein nächstes Element zu Null. In diesem einfachen Beispiel sind auch die Datenkomponenten öffentlich.

Liste 6.3: Template-Klasse `Qelement` (`queue-container.hh`)

```
1  template <class T1> class Qelement
2  {
3    public:
4      T1         value;                          // Datenkomponente
5      Qelement  *p_next;                         // Zeiger aus das nächste Element
6
7      Qelement  ( const T1 &val ) : value(val), p_next(0) { }; // Konstruktor
8  };
```

Die Template-Klasse `Queue` für die Organisation einer verketteten Liste ist etwas umfangreicher (Liste 6.4). Wie in Abbildung 6.1 zu sehen ist, sind als Datenelemente nur zwei Zeiger vorhanden. Der Konstruktor setzt zunächst beide auf Null. In diesem Beispiel sind nur wenige Methoden vorhanden:

empty stellt fest, ob die Liste leer ist.

add hängt ein neues Element am Ende einer Liste an.

remove entfernt das Element am Anfang einer Liste (falls vorhanden).

Destruktor Der Destruktor entfernt solange Elemente, bis die Liste leer ist.

Die Datenelemente sind **protected**, um abgeleiteten Klassen den Zugriff zu ermöglichen. Die **friend**-Vereinbarung am Anfang wird erst im nächsten Abschnitt wichtig.

Liste 6.4: Template-Klasse `Queue` (`queue-container.hh`)

```
1  template <class T1> class Queue
2  {
3    friend  class Qtraverse<T1>;                 // Iterator-Klasse
4
5    public:
6
7    Queue ( ) : p_front(0), p_end(0)  {}         // Konstruktor
8
9    virtual ~Queue ( )                           // Destructor
10   {
11     while ( !empty() ) remove();
12   }
13
14   bool  empty   ( void )                       // wahr, wenn Liste leer
15   {
16     return (p_front == 0);
17   }
18   virtual void  add     ( const T1 &value );   // Element am Ende anhängen
19   virtual T1    remove  ( void );              // Element am Anfang entfernen
20
21   protected:
22
23   Qelement<T1>  *p_front;                      // Zeiger auf das erste Element
24   Qelement<T1>  *p_end;                        // Zeiger auf das letzte Element
25
26 }; // -----  end of template class  Queue  -----
27
28 //---------------------------------------------------------------------
29 //  Queue : add
```

```
30 //----------------------------------------------------------------------
31 template <class T1>
32 void Queue<T1> :: add ( const T1 &value )
33 {
34   Qelement<T1>  *p_element = new Qelement<T1>( value );
35
36   if( empty() )                                  // Liste leer ?
37     p_front       = p_element;                  // p_front zeigt auf 1. Element
38   else
39     p_end->p_next = p_element;                  // am Ende einketten
40
41   p_end = p_element;                            // p_end zeigt auf letztes Element
42 }
43
44 //----------------------------------------------------------------------
45 //  Queue : remove
46 //  Erstes Element entfernen und zurückgeben.
47 //----------------------------------------------------------------------
48 template <class T1>
49 T1 Queue<T1> :: remove ( void )
50 {
51   if( empty() ) {
52     std::cout << "\n*** remove(): attempt to remove from an empty queue ***\n\n";
53     exit(1);
54   }
55
56   T1            first_element;
57   Qelement<T1> *p_element;
58
59   first_element = p_front->value;     // Inhalt des zu entfernenden Datenelementes
60   p_element     = p_front;            // Zeiger auf das zu entfernende Qelement
61   p_front       = p_front->p_next;    // p_front auf das nächste Qelement setzen
62   delete  p_element;                  // Qelement entfernen
63
64   return  first_element;              // Datenelement zurückgeben
65 }
66
67 #endif   // ----- #ifndef QUEUE_CONTAINER_INC  -----
```

Die Definitionen der beiden Template-Klassen stehen in einer Header-Datei. Liste 6.5 zeigt die Verwendung. In diesem Beispiel wird eine Liste für **int**-Objekte eingerichtet. Die Handhabung für Nicht-Basisdatentypen sieht genauso aus.

Liste 6.5: Template-Klasse `Queue` (`6.3-1.qtest.cc`)

```
1  #include  <iostream>
2  #include  "queue-container.hh"
3
4  using namespace std;
5
6  int main  (void)
7  {
8    Queue<int>  liste;                          // Liste für int-Objekte
9
10   cout  <<  "\n\tListe mit einigen Elementen belegen:\n\t";
11   for( int i=0; i<10; i++ ) {
```

```
12      cout  << " " << i;
13      liste.add( i );
14      }
15
16    cout  <<  "\n\tFolgende Elemente entfernen:\n\t";
17    for( int i=0; i<5; i++ )
18      cout  << " " << liste.remove( );
19
20    cout  <<  "\n\tFolgende Elemente hinzufügen\n\t";
21    for( int i=25; i<35; i++ ) {
22      cout  << " " << i;
23      liste.add( i );
24      }
25
26    cout  <<  "\n\tAlle Elemente entfernen:\n\t";
27    while( !liste.empty() )                      // Iteration mittels Methode
28      cout  << " " << liste.remove( );
29
30    if( liste.empty() )
31      cout  <<  "\n\tListe liste ist leer";
32    cout  << "\n\n";
33
34    return  EXIT_SUCCESS;
35  }
```

Das Testprogramm in Liste 6.5 erzeugt folgende Ausgabe:

```
Liste mit einigen Elementen belegen:
 0 1 2 3 4 5 6 7 8 9
Folgende Elemente entfernen:
 0 1 2 3 4
Folgende Elemente hinzufügen
 25 26 27 28 29 30 31 32 33 34
Alle Elemente entfernen:
 5 6 7 8 9 25 26 27 28 29 30 31 32 33 34
Liste liste ist leer
```

6.3.2. Iteratoren

Ein **Iterator** ist eine Einrichtung zum geordneten Durchlaufen einer Datenstruktur. Eine typische Iterator-Konstruktion ist eine Schleife zum linearen Durchlaufen eines Feldes. Für Datenstrukturen, die üblicherweise keine indizierten Elemente haben (zum Beispiel Bäume, Mengen, Hashs), muß das Konzept der Schleife verallgemeinert werden. Diese Verallgemeinerung besteht in der Verwendung von Zeigern auf Objekte, für die zusätzlich einige Methoden definiert werden (Setzen auf das erste, letzte, nächste Objekt und so weiter).

> Ein Iterator ist eine Verallgemeinerung eines Zeigers. Er wird in einer eigenen Klasse gekapselt.

Bei einer Liste beginnt ein Durchlauf beim ersten Listenelement. Wenn Zeiger verwendet werden, dann kann ein Zeiger um jeweils ein Element weitergesetzt werden, bis das Listenende erreicht ist. Je nach Datenstruktur und Organisationsaufand können Hilfsfunktionen bereitgestellt werden, die einen Zeiger um mehrere Elemente weitersetzen, zurücksetzen oder auf

das n-te Element setzen. Zur Systematisierung und Kapselung dieser Aufgabe verwendet man eigene Klassen, sogenannte Iterator-Klassen.

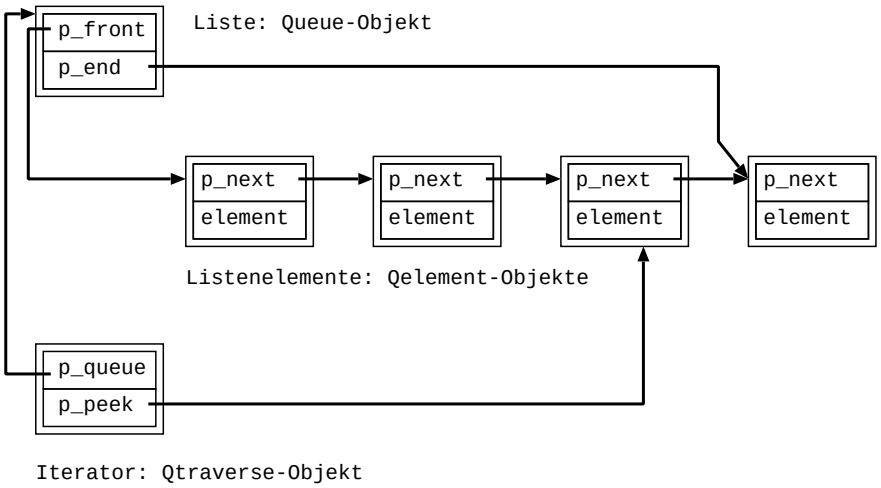


Abbildung 6.2.: Einfach verkettete Liste mit Iterator

Liste 6.6 zeigt eine solche Iterator-Klasse, die für die Zusammenarbeit mit der Klasse `Queue` entworfen wurde. Sie enthält als Datenelemente einen Zeiger auf eine Liste (`Queue<T1> *p_queue`) und einen Zeiger auf ein Listenelement (`Qelement<T1> *p_peek`). Abbildung 6.2 zeigt die Erweiterung des Beispiel aus Abbildung 6.1.

Liste 6.6: Iterator-Klasse für Queue-Objekte (`qtravers.hh`)

```
1  #ifndef  QTRAVERS_INC
2  #define  QTRAVERS_INC
3
4  #include  "queue-container.hh"
5
6  using namespace std;
7
8  // ============================================================================
9  //        Class:  Qtraverse
10 //  Description:  Iterator-Klasse für Queue-Objekte (Listen)
11 // ============================================================================
12 template <class T1> class Qtraverse
13 {
14   public:
15
16     Qtraverse ( Queue<T1> &queue ) {    // Konstruktor
17       p_queue = &queue; peek_reset();
18     }
19
20     void  peek_reset(void) {            // Iterator zurücksetzen
21       p_peek = 0;
22     }
23
24     bool  peek  ( T1 &cur_element );    // nächstes Element einer Liste
25
26   private:
27     Queue<T1>     *p_queue;             // Zeiger auf eine Queue
```

```
28      Qelement<T1>  *p_peek;              // Zeiger auf ein Qelement
29
30  };
31
32  //--------------------------------------------------------------------
33  //  Qtraverse:: peek
34  //  Das nächste Element einer Liste zurückgeben.
35  //--------------------------------------------------------------------
36    template <class T1>
37  bool  Qtraverse<T1>::peek ( T1 &cur_element )
38  {
39    if( p_peek == 0 )
40      p_peek  = p_queue->p_front;         // Zeiger auf das erste Element setzen
41    else
42      p_peek  = p_peek->p_next;          // Zeiger auf das nächste Element setzen
43
44    if( p_peek == 0 )                    // Ende der Liste ?
45      return false;                      // Fehler
46    cur_element = p_peek->value;         // Rückgabewert
47    return  true;                        // Erfolg
48  }   // -----  end of method Qtraverse::peek  -----
49
50  #endif   // ----- #ifndef QTRAVERS_INC  -----
```

Der Zeiger auf die Elemente einer Liste soll nicht direkt zugänglich, sondern in einem Iterator-Objekt gekapselt sein. Der Konstruktor übernimmt hierzu eine Referenz auf eine Liste und stellt den internen Elementzeiger auf Null (kein Verweis auf ein Objekt vorhanden). Die wichtigste Methode ist `peek`. Sie stellt den internen Zeiger auf das erste Listenelement, wenn er zuvor Null war, oder auf das nächste Listenelement der Liste, wenn der Zeiger zuvor bereits auf ein Listenelement gezeigt hat. Wurde ein nächstes Element gefunden, dann wird eine Referenz über den Aufrufparameter zurückgegeben und der **return**-Wert ist **true**. Wurde kein Element gefunden (Liste leer oder Listenende erreicht), dann wird **false** zurückgegeben. Mit Hilfe dieser Rückgabewerte können dann sehr einfach Kontrollstrukturen gesteuert werden.

Das Programm in Liste 6.7 zeigt die Verwendang. Zu der Liste `liste` wird der Iterator `listeIter` angelegt. Damit kann dessen Methode `peek` auf diese Liste angewendet werden. Nachdem die Liste gefüllt wurde, kann sie in der **while**-Schleife sehr einfach vollständig durchlaufen werden. Auch ein schrittweiser Durchlauf ist natürlich möglich. Der Iterator liefert das jeweils nächste Element zurück und zwar solange, bis das Listenende erreicht ist. Mit `listeIter.reset()` kann der Iterator bei Bedarf zurückgestellt werden.

Liste 6.7: Verwendung eines Iterators (`6.3-2.qtest.cc`)

```
1   #include  "queue-container.hh"
2   #include  "qtravers.hh"
3
4   using namespace std;
5
6   int main  (void)
7   {
8     int             elem;                       // Hilfsgröße
9     Queue<int>      liste;                      // Liste mit int-Elementen
10    Qtraverse<int>  listeIter(liste);           // Iterator für liste
11
12    //--------------------------------------------------------------------
```

```
13    //  Liste aufbauen
14    //-------------------------------------------------------------------------
15    cout  <<  "\n\n\tListe aufbauen :\n\t";
16    for( int i=0; i<10; i++ ) {
17      cout  << " " << i;
18      liste.add( i );
19      }
20
21    //-------------------------------------------------------------------------
22    //  Liste mit Hilfe eines Iterators vollständig durchlaufen
23    //-------------------------------------------------------------------------
24    cout  <<  "\n\n\tListe vollständig durchlaufen ...\n\t";
25    while( listeIter.peek( elem ) )              // Iterator steuert die Schleife
26      cout  << " " << elem;
27    cout  << "\n\n";
28
29    return  EXIT_SUCCESS;
30  }
```

Für den praktischen Einsatz sind noch einige Dinge zu bedenken. So könnte zum Beispiel die Liste während der Iteration verändert oder gelöscht werden. Zur Lösung könnten die Listen-Objekte Zähler enthalten, die beim Hinzufügen von Elementen erhöht, beim Entfernen heruntergezählt werden. Der Iterator speichert einen Vergleichswert und kann auf Ungleichheit der Zähler zum Beispiel mit `peek_reset` reagieren.

6.4. Die Standard Library

Zum *C++*-Standard gehört eine umfangreiche Bibliothek, die sich in mehrere Bereiche gliedern läßt (Tabelle A.3) und die etwa 50 Module umfaßt. Die Standard-*C*-Bibliothek ist ein Bestandteil der *C++*-Bibliothek.
Einzelne Teile sind objektorientiert aufgebaut. Dazu gehören insbesondere die Ein-/Ausgabemodule, die benutzerdefinierte Erweiterungen ermöglichen. Daneben ist der Hauptbestandteil die sogenannte Standard Template Library (STL), die in Abschnitt 6.4.1 kurz vorgestellt wird.

Zunächst ein Beispiel, das die Template-Klasse `valarray` verwendet. Das Modul `valarray` (Header `<valarray>`) stellt eine Template-Klasse für eindimensionale Felder mit numerischem Datentyp zur Verfügung. Die Klasse enthält bereits eine ganze Fülle von Methoden, die der Benutzer verwenden kann, sobald ein `valarray`-Objekt vereinbart ist:

```
valarray<double>  v1(1000);           // 1000 Elemente
valarray<int>     v2(-.33, 2000)      // 2000 Elemente, alle mit Wert -.33
valarray<double>  v3 = v1;            // 1000 Elemente
```

Liste 6.8 zeigt ein vollständiges Programm, in dem Vektoren mit 5 Elementen vom Typ **`double`** verwendet werden. Um die Vereinbarungen abzukürzen wird ein **`typedef`** verwendet (Zeile 7).

Im Hauptprogramm werden einige Methoden verwendet, die zur Ausstattung der Klasse `valarray` gehören. Sie geben einen ersten Eindruck von der Mächtigkeit dieser Klasse. Auch die Operatoren der Vektorarithmetik stehen bereits zur Verfügung, so daß eigene Operatorüberladungen nicht erforderlich sind.

Alle Elemente des Vektors `v1` werden mit dem Wert 10.5 initialisiert. Die benutzerdefinierte Funktion `setrand` wird von der Methode `apply` aufgerufen. `apply` wendet die als Parameter angegebene Funktion auf den Wert jedes Elementes an. Die Funktion `setrand` benutzt den Elementwert als Bereichsangabe und ersetzt ihn durch einen Zufallswert, der im Bereich zwischen Null und dem Elementwert liegt. `setrand` ist offensichtlich eine benutzerdefinierte Funktion. Ihre Signatur muß natürlich der entsprechen, die die Methode `apply` für ihren Parameter fordert.
Der Vektor `v2` entsteht aus dem Vektor `v1`, indem eine Kopie angelegt wird, in der alle Elemente zyklisch um einen Platz nach links verschoben sind (Methode `cshift`).
Der Vektor `v3` entsteht durch arithmetische Verknüpfung der ersten beiden Vektoren.

Die Zeilen 48 bis 50 zeigen die Verwendung der Methoden `sum`, `min` und `max` zur Bestimmung der Summe aller Vektorelemente, sowie des kleinsten und größten Elementwertes. Diese Funktionen sind als generische Funktionen vordefiniert und deshalb ohne weiteres Zutun auf `valarray`-Objekte anwendbar.
Der Ausdruck `(v1*v2).sum()` in Zeile 51 liefert zunächst einen Vektor als Zwischenergebnis, dessen Elemente das Produkt der jeweiligen Elemente der beiden Operanden der Multiplikation sind. Diese Elemente werden durch die Methode `sum` addiert. Der Ausdruck liefert also das Skalarprodukt der beiden Vektoren. Das Programm erzeugt die folgende Ausgabe:

```
v1 =     1.00    4.00    9.00    8.50    8.00
v2 =     4.00    9.00    8.50    8.00    1.00
v3 =    12.50   32.50   43.75   41.25   22.50
  Summe der Elemente von v3 =   152.50
Minimum der Elemente von v3 =    12.50
Maximum der Elemente von v3 =    43.75
        Skalarprodukt v1*v2 =   192.50
```

Liste 6.8: Verwendung des Bibliotheksmoduls valarray (6.4-1.valarray-stl.cc)

```
1  using namespace std;
2
3  #include <iostream>
4  #include <iomanip>
5  #include <valarray>                              // Standard Template Library
6
7  typedef   valarray<double>  darray;
8
9  //-------------------------------------------------------------------------
10 //  Zufallszahl zwischen low und high (einschließlich)
11 //-------------------------------------------------------------------------
12 int setrand( int low, int high )
13 {
14   return (low+rand()%(high-low+1));
15 }
16
17 //-------------------------------------------------------------------------
18 //  Vektor von Typ darray ausgeben
19 //-------------------------------------------------------------------------
20 void  print_darray ( darray v, int size, string text )
21 {
22   cout  << endl << text << " = ";
23   for( int i=0; i<size; i++ )
24     cout << setw(6) << v[i];
25 }
26
27 //-------------------------------------------------------------------------
28 //  Hauptprogramm
29 //-------------------------------------------------------------------------
30 int main( )
31 {
32   int    size     = 5;                           // Anzahl der Vektorkomponenten
33   darray v1(size);                               // 1. Vektor
34   darray v2(size);                               // 2. Vektor
35   darray v3(size);                               // 3. Vektor
36
37   for(int i=0;i<size;i++) {
38     v1[i] = setrand( 1, 10 );                    // 1. Vektor besetzen
39     v2[i] = setrand( 1, 10 );                    // 2. Vektor besetzen
40   }
41   print_darray( v1, size, "v1         " );      // 1. Vektor ausgeben
42   print_darray( v2, size, "v2         " );      // 2. Vektor ausgeben
43
44   v3  = 2.5*(v1+v2);                             // Skalar*Vektorsumme
45   print_darray( v3, size, "2.5*(v1+v2)" );      // Ergebnis ausgeben
46
47   cout  << "\n";
48   cout  << "\n  Summe der Elemente von v3 = " << setw(10) << v3.sum();
49   cout  << "\nMinimum der Elemente von v3 = " << setw(10) << v3.min();
50   cout  << "\nMaximum der Elemente von v3 = " << setw(10) << v3.max();
51   cout  << "\n        Skalarprodukt v1*v2 = " << setw(10) << (v1*v2).sum();
52
53   return 0;
54 }
```

6.4.1. Die Standard Template Library

Ein Teil der *C++*-Standardbibliothek ist die sogenannte Standard Template Library, kurz STL genannt. Die STL geht auf Arbeiten zurück, die in den frühen 90er Jahren bei Hewlett-Packard durchgeführt wurden, und aus denen eine Bibliothek mit generischen Containern, Iteratoren und Algorithmen entstanden ist. Alle Bestandteile wurden als Templates implementiert.
Die STL bietet umfangreiche Unterstützung für immer wieder verwendete Datenstrukturen, wie etwa Listen, Stapel, Felder, Mengen. Dazu stehen passende Iteratoren und Algorithmen zur Verfügung. Die Bibliothek ist inzwischen Teil des *C++*-Standards.

Die bereits eingeführten Konzepte Container und Iterator werden um generische Algorithmen ergänzt. Die Grundidee besteht darin, die generisch implementierten Algorithmen über Iteratoren auf die Container zugreifen zu lassen. Der Anwender sorgt durch die richtige Verbindung dieser Bestandteile dafür, daß das Ziel erreicht wird. Das kleine Beispiel in Liste 6.9 zeigt die Anwendung dieser Vorgehensweise.

Liste 6.9: Vergleich zweier Listen (`6.4-2.stl-liste-equal.cc`)

```
1  #include  <algorithm>                          // wegen equal()
2  #include  <iostream>
3  #include  <list>                               // doppelt verkettete Listen
4
5  using namespace std;
6
7  int main ( int argc, char *argv[] )
8  {
9    bool      vergleich;
10   int       listeLaenge   = 10;                // Listenlänge
11   list<int> liste1;                            // Originalliste
12   list<int> liste2;                            // Vergleichsliste
13
14   for ( int i = 0; i < listeLaenge; i += 1 )   // Liste mit Zufallswerten füllen
15     liste1.push_back( random()%20 );           // Werte: 0 ... 19 (einschl.)
16
17   liste2  = liste1;                            // 1. Liste kopieren
18
19   vergleich = equal( liste1.begin(),           // erstes Element, Originalliste
20                      liste1.end(),             // letztes Element, Originalliste
21                      liste2.begin()            // erstes Element, Vergleichsliste
22                      );
23   if ( vergleich )
24     cout  << "Die beiden Listen sind gleich.\n\n";
25   else
26     cout  << "Die beiden Listen sind NICHT gleich.\n\n";
27
28   return EXIT_SUCCESS;
29 }   // ----------  end of function main  ----------
```

Nachdem `liste1` erzeugt, gefüllt und auf `liste2` kopiert ist, wird die Funktion `equal` (Header `<algoritm>`) dazu verwendet, die Gleichheit der Listen zu überprüfen. Die Funktion `equal` besitzt folgenden Prototypen:

```
template <class InputIterator1, class InputIterator2>
bool equal( InputIterator1 first1, InputIterator1 last1,
            InputIterator2 first2);
```

Die drei Parameter sind Iteratoren. Der erste zeigt auf das erste Objekt der Originalliste. Hier soll der Vergleich beginnen. Der zweite Iterator zeigt hinter das letzte Objekt der Originalliste, das noch in den Vergleich eingehen soll. Der dritte Iterator zeigt auf das erste Element der anderen Liste, das in den Vergleich eingehen soll. Das Ergebnis ist ein Wahrheitswert.

Die Gleichheit der beiden Listen wird festgestellt, indem überprüft wird, ob die sich entsprechenden Listenelemente gleich sind. Da in diesem Beispiel die Listen **int**-Objekte enthalten, wird automatisch der arithmetische Vergleich verwendet. Bei Listen, die nicht aus Standarddatentypen bestehen, muß eine zweite Form von `equal` verwendet werden. Diese erhält über einen vierten Parameter einen Zeiger auf eine Vergleichsfunktion.

An den gezeigten Beispielen ist erkennbar, daß viele grundlegende Datenstrukturen und die dazugehörigen Algorithmen aus der Standardbibliothek bezogen werden können. Der Anwender braucht sich somit nicht immer wieder mit deren Implementierung zu beschäftigen. Bei der Implementierung der Standardbibliothek war die Laufzeiteffizienz ein wesentlicher Gesichtspunkt. Zu einer eingehenderen Betrachtung muß auf die Literatur verwiesen werden ([Jos12], [SGCS99], [SDT06]).

6.4.2. Erweiterung einer STL-Klasse

Liste 6.10: Klasse `Stack2<T>`: Erweiterung der STL-Klasse `stack<T>` (`stack2.hh`)

```
1  #ifndef  STACK2_INC
2  #define  STACK2_INC
3
4  using namespace std;
5
6  #include  <stack>
7
8  // ================================================================================
9  //        Class:  Stack2
10 //  Description:  Erweiterung der STL-Klasse stack<T>
11 // ================================================================================
12 template < class T >
13 class Stack2 : public stack <T>
14 {
15   public:
16   // ====================  LIFECYCLE     ======================================
17   Stack2 () : stack<T>() {};             // Konstruktor
18
19   // ====================  MUTATORS      ======================================
20   T     pop ( );                         // pop mit Rückgabewert
21   void  dup ( );                         // oberstes Stapelelement verdoppeln
22   void  swap( );                         // oberste Stapelelemente vertauschen
23
24 }; // -----  end of template class Stack2  -----
25
26 #include  "stack2.ii"                    // Implementierung
27
28 #endif   /* ----- #ifndef STACK2_INC  ----- */
```

STL-Klassen können ebenfalls als Grundlage für eigene Erweiterungen dienen. Liste 6.10 zeigt eine Erweiterungen der Klasse `stack<T>` um drei Methoden: die Methode `pop` besitzt einen

Rückgabewert, die Methode `dup` verdoppelt das oberste Stack-Element und die Methode `swap` vertauscht die beiden obersten Stack-Elemente.

Die neue Klasse `Stack2<T>` ist wieder eine Template-Klasse und deswegen sind Definition und Implementierung vollständig in der zugehörigen header-Datei enthalten. Die Implementierung der Methoden kann von der Klassendefinition getrennt werden, wenn man sie in eine eigene Datei (zum Beispiel `stack2.ii`, Liste 6.11) verlegt, die vom Präprozessor in die eigentliche header-Datei mittels `#include` aufgenommen wird.

Liste 6.11: Klasse `Stack2<T>`, Implementierung (`stack2.ii`)

```
1  //------------------------------------------------------------------------------
2  //       Class:  Stack2
3  //      Method:  pop
4  // Description:  pop mit Rückgabewert
5  //------------------------------------------------------------------------------
6    template < class T >
7  T Stack2<T>::pop (  )
8  {
9    T result  = stack<T>::top();
10   stack<T>::pop();
11   return result;
12 }   // -----  end of method Stack2<T>::pop  -----
13
14 //------------------------------------------------------------------------------
15 //       Class:  Stack2
16 //      Method:  dup
17 // Description:  oberstes Stapelelement verdoppeln
18 //------------------------------------------------------------------------------
19   template < class T >
20 void Stack2<T>::dup (  )
21 {
22   stack<T>::push( stack<T>::top() );
23 }   // -----  end of method Stack2<T>::dup  -----
24
25 //------------------------------------------------------------------------------
26 //       Class:  Stack2
27 //      Method:  swap
28 // Description:  oberste Stapelelemente vertauschen
29 //------------------------------------------------------------------------------
30   template < class T >
31 void Stack2<T>::swap (  )
32 {
33   T top1  = pop();                              // Stack2<T>::pop
34   T top2  = pop();                              // Stack2<T>::pop
35   stack<T>::push( top1 );
36   stack<T>::push( top2 );
37 }   // -----  end of method Stack2<T>::swap  -----
```

A. Tabellen

C++-Schlüsselwörter

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				

Tabelle A.1.: ***C++***-Schlüsselwörter

C++-Operatoren

Operator	Bedeutung	Bindung
`::`	Bereichsoperator (scope resolution)	links
`.`	Zugriff auf ein Strukturelement	links
`->`	Zeigerzugriff auf ein Strukturelement	links
`[]`	Indexklammern eines Feldelementes	links
`()`	Argumentklammern (Funktionsaufruf)	links
`++ --`	Inkrement, Dekrement (Postfix)	links
`sizeof` Ausdruck	Speichergröße eines Objektes	rechts
`sizeof(`Typ`)`	Speichergröße eines Datentyps	rechts
`++ --`	Inkrement, Dekrement (Präfix)	rechts
`!`	logische Verneinung	rechts
`~`	bitweises Komplement	rechts
`+ -`	Vorzeichen	rechts
`&`	Adreßoperator	rechts
`*`	Inhaltsoperator	rechts
`new` Typ	dynamische Speicherbelegung	
`delete` Zeiger	Speicherfreigabe	
`delete []` Zeiger	Speicherfreigabe	
`(`Zieltyp`)`	explizite Typumwandlung	rechts
`* / %`	Multiplikation, Division, Divisionsrest	links
`+ -`	Addition, Subtraktion	links
`<< >>`	bitweise nach links, rechts schieben	links
`< <= > >=`	arithmetische Vergleiche	links
`== !=`	arithmetische Vergleiche	links
`&`	bitweise UND	links
`^`	bitweise exklusives ODER	links
`|`	bitweise ODER	links
`&&`	logisches UND	links
`||`	logisches ODER	links
`=`	einfache Zuweisung	rechts
`+= -= *= /= %=`	Zuweisungsoperatoren	rechts
`&= ^= |=`		rechts
`<<= >>=`		rechts
`throw` A	Ausnahmeobjekt A auswerfen	
`,`	Kommaoperator	links

Tabelle A.2.: Vorrang und Bindung der *C++*-Operatoren (Auswahl)

C++-Standardbibliothek

Die ***C++***-Standardbibliothek ist sehr umfangreich und leistungsfähig. Die ***C***-Bibliothek ist ein Bestandteil der ***C++***-Bibliothek. Für eine Beschreibungen der Module und deren Handhabung muß auf geeignete Lehrwerke (z.B. [IH04]) und die Originaldokumentation verwiesen werden.

Tabelle A.3.: ***C++***-Standardbibliothek — Übersicht

Bereich	Header	Kurzbeschreibung
Container	`<bitset>`	boolesche Mengen
	`<deque>`	Schlange mit 2 Enden
	`<list>`	doppelt verkett. Listen
	`<map>`	Schlüssel-Wert-Paare
	`<queue>`	Schlange
	`<set>`	Menge
	`<stack>`	Stapel
	`<vector>`	eindimensionales Feld
Allgemeine Dienste	`<functional>`	Funktionsobjekte
	`<memory>`	Speicherbeschaffung
	`<utility>`	Vergleichsoperatoren
	`<ctime>`	***C***-Bibliothek
Iteratoren	`<iterator>`	Iteratoren
Algorithmen	`<algorithm>`	Algorithmen
	`<cstdlib>`	***C***-Bibliothek
Fehlerbehandlung	`<exception>`	Exception-Klassen
	`<stdexcept>`	Standard-Exceptions
	`<cassert>` `<cerrno>`	***C***-Bibliothek
Zeichenketten	`<string>`	dyn. Zeichenketten
	`<cctype>` `<cwchar>`	***C***-Bibliothek
	`<cstdlib>` `<cwctype>`	***C***-Bibliothek
	`<cstring>`	***C***-Bibliothek
Eingabe/Ausgabe	`<fstream>`	Datei-E/A
	`<iomanip>`	E/A-Manipulatoren
	`<ios>`	E/A-Basisklassen
	`<iosfwd>`	E/A
	`<iostream>`	Standard-E/A
	`<istream>`	Eingabe
	`<ostream>`	Ausgabe
	`<sstream>`	String-E/A
	`<streambuf>`	gepufferte E/A
	`<cstdio>` `<cstdlib>`	***C***-Bibliothek
	`<cwchar>`	***C***-Bibliothek
Internationalisierung	`<locale>`	Sprachanpassungen
	`<clocale>`	***C***-Bibliothek
Sprachunterstützung	`<exception>`	Exceptions-Behandlung

weiter auf der nächsten Seite

Bereich	Header	Kurzbeschreibung
	<limits>	numerische Grenzwerte
	<new>	Speicherbeschaffung
	<typeinfo>	Typerkennung
	<cfloat> <cstdarg>	*C*-Bibliothek
	<climits> <cstddef>	*C*-Bibliothek
	<csetjmp> <cstdlib>	*C*-Bibliothek
	<csignal> <ctime>	*C*-Bibliothek
Numerik	<complex>	komplexe Zahlen
	<numeric>	num. Operationen
	<valarray>	Zahlenvektoren
	<cmath> <cstdlib>	*C*-Bibliothek

Literaturverzeichnis

[IH04] Isernhagen ; Helmke: *Softwaretechnik in C und C++*. 4. Aufl. Carl Hanser Verlag, 2004. – Sehr umfangreich. Geht deutlich über die Bedürfnisse der ersten beiden Semester hinaus. Sehr gut geeignet als anspruchsvolle Einführung und als Nachschlagewerk, auch in weiterführenden Veranstaltungen.

[KM03] Koenig, Andrew ; Moo, Barbara E.: *Intensivkurs C++. Schneller Einstieg über die Standardbibliothek*. Addison-Wesley Longman Verlag, 2003. – ISBN 3827370299

[MBG04] Misfeldt ; Bumgardner ; Gray: *The Elements of C++-Style*. Cambridge University Press, 2004. – 182 S. – Programmierrichtlinien für Fortgeschrittene; Hinweise zu Formatierung und Dokumentation, Erläuterung allgemeiner Programmierprinzipien, viele Hinweisen zum richtigen Gebrauch von C++-Sprachelementen.

[Jos12] Josuttis, Nicolai M.: *The C++ Standard Template Library (2nd Edition)*. Pearson Education Inc., 2012. – ISBN 978-0-321-62321-8

[Mey05] Meyers, Scott: *Effective C++*. 3. Aufl. Pearson Education, Inc., 2005

[SDT06] Stephens, Ryan K. ; Diggins, Christopher ; Turkanis, Jonathan: *C++ Kochbuch*. O'Reilly, 2006. – ISBN 3897214474

[SGCS99] Silicon Graphics Computer Systems, Inc: *Standard Template Library Programmer's Guide*. http://www.sgi.com/, 1999

[Str00] Stroustrup, Bjarne: *The C++ Programming Language (Special 3rd Edition)*. Addison-Wesley Professional, 2000. – ISBN 0201700735

m

Abbildungsverzeichnis

Tabellenverzeichnis

Programmlisten