Einführung in ISO-C++

Holger Jakobs

10. Februar 2003

Inhaltsverzeichnis

Kapitel 1

Vorstellung von C++

C++ ist ein Versuch, C zu verbessern, ohne dessen Effizienz, Portabilität und Flexibilität zu opfern.

1.1 Eigenschaften von C++

1.1.1 Allgemeine Vorteile gegenüber C

- *Datenabstraktion:* Klassen in C++ erlauben die Definition von Datenstrukturen, deren interne Details verborgen bleiben. Dies schützt die Inhalte der Strukturen nicht nur vor unsachgemäßem Umgang mit ihnen, sondern ermöglicht es auch, die interne Darstellung zu verändern, ohne dass andere Teile des Programms davon beeinflusst werden.
- *Erweiterbarkeit:* C++ erlaubt die Erweiterung der Sprache selbst durch die Definition von neuen Datentypen (in Form von Klassen), die von den fest eingebauten kaum unterscheidbar sind, weil sie so gut integriert sind. Gutes Beispiel dafür sind neue numerische Datentypen wie `complex` oder `bcd`, auf die man die normalen numerischen Operatoren anwenden kann, oder auch `string`, denen man einfach eine Zeichenkette beliebiger Länge zuweisen kann, ohne dass sie überlaufen.
- *Programmieren im großen Stil:* Die Unterstützung beim Schreiben großer Programme ist deutlich besser als bei C. Insbesondere lässt sich ein Programm als eine Sammlung von Klassen leichter beherrschen und warten.
- *Wiederverwendbarkeit von Code:* Es ist in C++ leichter als in C, wiederverwendbare Komponenten zu schreiben. Klassen kann man vom Hauptprogramm gut trennen und in Klassenbibliotheken ablegen. Dass das mit der Wiederverwendbarkeit und Universalität gut klappt, sieht man am Erfolg kommerzieller Klassenbibliotheken (z. B. RogueWave) und auch an der Standardbibliothek von C++ (eine Weiterentwicklung der Standard Template Library).
- *Objektorientierte Programmierung:* C++ ist eine echte objektorientierte Sprache, die alles hierzu notwendige kennt: Klassen, Methoden, Polymorphismus, Vererbung.
- *Sicherheit:* C++ ist viel strenger als C. Beispielsweise **müssen** Funktionen vor ihrer Verwendung deklariert werden. Diese erhöhte Sicherheit erlaubt dem Compiler, viele Fehler zu erkennen, die in C unentdeckt blieben.

1.1.2 Nachteile von C++

C++ hat aber auch Nachteile, wie jede Programmiersprache:

- *Komplexität:* Die Sprache C mit ihrer Standardbibliothek ist schon nicht gerade übersichtlich. C++ enthält all dieses und noch sehr viel mehr. Außerdem können die ganzen Eigenschaften von C++ auch noch auf vielfältige Weise miteinander kombiniert werden.

- *Geringere Übersetzungsgeschwindigkeit:* Da der Compiler viel aufwendiger ist, dauern die Verarbeitung des Quellcodes und die Erzeugung des Objectcodes auch entsprechend länger. Komplexere Compiler sind übrigens auch fehlerträchtiger.
- *Fallen:* Jede Sprachen hat ihre Tücken. Einige der Tücken von C wurden vermieden, beispielsweise die Zeigerübergabe als „call by reference“, das es jetzt „echt“ gibt. Andere blieben erhalten, beispielsweise die ganze Zeigerarithmetik und auch die Übergabe von Zeigern als Parameter, die man zwar nicht mehr unbedingt benutzen muss, aber immer noch kann. Zusätzliche Fallen gibt es natürlich auch.

1.1.3 Die wichtigsten Veränderungen und Erweiterungen gegenüber C

- Kommentar `//...` zusätzlich zu `/* ... */`
- Deklarationsreihenfolge: Variablen können an beliebiger Stelle im Code deklariert werden – am besten dort, wo sie gleichzeitig mit einem Wert initialisiert werden können.
- Arrays, Strukturen, Unions usw. können in C++ mit beliebigen Werten initialisiert werden, auch mit Funktionswerten, Ausdrücken usw.

```
int n = 2;
int powers [] = {1, n, n*n, n*n*n, n*n*n*n};
```

- echte Konstanten:

```
const int n = 10;
int a[n]; // nur in C++
```

  In C++ sollte man Präprozessormakros nicht benutzen! Zitat aus dem Buch von Stroustrup, dem C++-Erfinder: „Die wichtigste Regel bei Makros ist: Benutze keine, es sei denn, es ist unabdingbar. Fast jedes Makro demonstriert eine Schwäche in der Programmiersprache, im Programm oder beim Programmierer.“

- Typdefinitionen: In C muss man `typedef` verwenden, wenn man einen neuen Typen definieren will. Ohne `typedef` definierte Strukturen sind nur Tags.

```
struct Complex {float real, imag;};
struct Complex x; /* "struct" in C notwendig */

typedef struct {float real, imag;} Complex;
/* mit typedef wird in C ein Typ definiert */
Complex x;
```

  In C++ sind wie zuerst definierte Strukturen automatisch Typen. `typedef` ist vollkommen überflüssig.

- Das for-Statement hat eine andere Syntax. In C:

```
for (Ausdruck1; Ausdruck2; Ausdruck3) Anweisung;
```

  In C++:

```
for (Anweisung1; Ausdruck2; Ausdruck3) Anweisung;
```

Daher kann man in C++ eine Variable in der `for`-Schleife deklarieren:

```
for (int i=0; i<n; i++) Anweisung2;
```

Der Gültigkeitsbereich dieser Variablen `i` hängt vom Compilerstandard ab:

AT&T 2.1	bis zum Ende des umschließenden Blocks
ISO C++	bis zum Ende der `for`-Schleife

Daher ist folgendes nur mit neueren, ISO-konformen Compilern möglich:

```
for (int i=0; i<n; i++) x[i]++;
for (int i=0; i<n; i++) y[i]++;
```

Die aktuellen C++-Compiler von Sun und HP (Stand Anfang 1999) meckern hier die doppelte Deklaration von `i` an, der GNU C++-Compiler hält sich schon an den neuen Standard und akzeptiert obigen Code.

- Typkonversionen in C nur mit cast-Ausdrücken, in C++ auch mit Funktionsschreibweise.

```
x = (float) i; // in C und in C++
x = float(i);  // nur in C++
```

- Ein-Ausgabe kann man in C++ wahlweise mit `<stdio.h>` oder mit `<iostream>` durchführen. Bitte aber nicht beides im selben Programm bei derselben Datei mischen. Es ist aber in Ordnung, für die Konsole (`cin` und `cout`) `<iostream>` zu verwenden und für Plattendateien `<stdio.h>`.

```
cout << "Werte fuer i und j eingeben: ";
cin >> i; cin >> j;
cout << "Summe ist " << i+j << endl;
```

Der Operator `<<` heißt Insertions- oder Ausgabeoperator, weil er in einen Stream (Datenstrom, hier `cout`) etwas einfügt, der Operator `>>` heißt Extraktions- oder Eingabeoperator, weil er aus einem Stream (Datenstrom, hier `cin`) etwas herausholt/extrahiert. Manchmal wird `<<` auch mit „sende nach“ und `>>` mit „empfange von“ erläutert.

`>>` liest immer nur ein Wort (bis zu einem Whitespace), auch wenn man in eine Zeichenkette einliest, d. h. es verhält sich wie `scanf ("%s", ...)`. Möchte man eine ganze Zeile einlesen, so muss man die Funktion `cin.getline(char *, int maxlen)` verwenden.

- Möchte man C-Funktionen in C++ verwenden, so muss man sie als solche deklarieren:

```
extern "C" void f (int);
extern "C" {
  void f (int);
  int  g (float);
}
extern "C" {
  #include "cfunkts.h"
}
```

Die Standard-Headerfiles sind alle schon mit entsprechenden Erweiterungen für die C++-Compiler ausgerüstet.

1.2 Inkompatibilitäten zwischen C und C++

- Es gibt einige neue Schlüsselwörter, die es in C nicht gab. Bezeichner dürfen mit diesen nicht identisch sein:
  `asm catch class delete friend inline new operator private protected public template try this throw virtual`
  im AT&T 2.1 Standard, und zusätzlich
  `and explicit namespace or_eq typename and_eq export using compl false not bitand const_cast not_eq true wchar_t bitor reinterpret_cast xor bool dynamic_cast mutable or static_cast typeid xor_eq`
  im ISO-C++.

- Funktionsdeklarationen ohne Parameter haben verschiedene Bedeutungen. In C ist die Anzahl Parameter nicht angegeben, in C++ ist sie 0. Klar ist nur `int f (void)`. Die Deklaration von Funktionen im „klassischen“ C-Stil (K&R-Stil) ist in C++ verboten.

- Die Größe von Zeichenkettenliteralen ist unterschiedlich. `sizeof ('a')` hat in C die Größe `sizeof (int)`, in C++ aber die Größe `sizeof (char)`.

- In C dürfen Variablen mehrfach (identisch) deklariert werden, in C++ ist das illegal.

- In C kann eine Zeichenkette bei der Initialisierung randvoll geschrieben werden, ohne Platz für das ASCII-NUL-Zeichen `\0` zu lassen.

```
char Name [4] = "Hugo"; /* geht nur in C, nicht in C++ */
```

- In C können Werte von Aufzählungen immer zwischen `int` und einem Aufzähltyp beliebig zugewiesen werden, weil sie automatisch konvertiert werden. In C++ muss man hier ausdrücklich casten.

```
enum Boolean {false, true};
Boolean b;
int i;
i = false; /* in C und C++ möglich */
b = 0;     /* nur in C möglich */

Boolean istkleinbuchstabe (char c) {
  return 'a' <= c && c <= 'z';
} /* geht so nur in C */

Boolean istkleinbuchstabe (char c) {
  return Boolean ('a' <= c && c <= 'z');
} /* geht so auch in C++ */
```

Viel besser ist es natürlich, `Boolean` als Klasse zu definieren, was im ISO-C++-Standard unter dem Name `bool` bereits geschehen ist.

- Gültigkeitsbereich

```
int T;
void f(void) {
  struct T {...};
  int i = sizeof (T);
}
```

In C bekommt `i` den Wert der Größe von `int` zugewiesen, in C++ die Größe der Struktur `T`, weil diese in C++ als Typ sichtbar ist – und nicht nur ein Tag. Allerdings kann man in C++ auch auf den Namen `T` aus dem äußeren Gültigkeitsbereich zugreifen, indem man den Scope-Operator `::` verwendet:

```
int T;
void f(void) {
  struct T {...};
  int i = sizeof (::T);
}
```

- `void *`-Zeiger: In C kann man jeden Zeiger auf einen `void *`-Zeiger zuweisen und umgekehrt. In C++ kann man zwar einem `void *`-Zeiger jeden beliebigen Zeiger zuweisen, aber nicht umgekehrt. Das Ergebnis von `malloc()` muss also immer gecastet werden:

```
int *intZeiger;
void *voidZeiger;
voidZeiger = intZeiger; /* geht in C und C++ */
intZeiger = voidZeiger; /* geht nur in C */

intZeiger = malloc (3*sizeof(int)); /* nur in C möglich */
intZeiger = (int *) malloc (3*sizeof(int)); /* C und C++ */
```

Aber natürlich ist es noch besser, völlig auf `malloc` zu verzichten und statt dessen `new` zu benutzen.

Kapitel 2

Standard-Ein- und -Ausgabe in C++

Man kann in C++ ganz normal mit den Funktionen `scanf()` und `printf()` aus der C-Bibliothek `stdio.h` arbeiten, aber das ist nicht C++-typisch und nutzt die Vorteile von C++ nicht aus. Daher sollen hier die in C++ üblichen Ein- und Ausgabemechanismen erläutert werden – zunächst für die Konsole.

Hierzu ein kleines Beispielprogramm:

```
#include <iostream> // Includen der passenden Headerdatei

int min (int, int); // Prototyp einer Minimum-Funktion

int main () {
  int x, y, z;
  cout << "x: "; cin >> x;
  cout << "y: "; cin >> y;
  cout << "die kleinere Zahl war: " << min (x, y) << endl;
  return 0;
}
```

Jetzt müssen wir die Funktion `min()` noch implementieren, damit der Linker keinen Fehler meldet:

```
int min (int a, int b) {
  if (a < b) return a;
  return b;
}
```

Alle in C++ enthaltenen Datentypen können mit `cin >>` eingelesen und mit `cout <<` ausgegeben werden. Man muss sich dabei überhaupt nicht um Formatierung kümmern. Natürlich kann man Ausgaben formatieren, wenn man die Anzahl Stellen oder Anzahl Nachkommastellen festlegen möchte, aber das soll zunächst nicht Thema sein. Wir beschränken uns auf die Standardformatierung.

Mehrere Werte hintereinander einlesen oder auslesen kann man durch die mehrfache Verwendung des `>>` bzw. `<<` Operators:

```
cin >> a >> b >> c;
cout << "Wert von a[" << i << "] ist " << a [i] << endl;
```

Das hier gezeigte `endl` schreibt ein Zeilenende und ist dem umständlichen `"\n"` des Preprocessors vorzuziehen, weil es gleichzeitig auch alle Daten im Puffer ausgibt – entsprechend `fflush()` aus der C-Standardbibliothek. Um lediglich die Daten im Puffer auszugeben (ohne Zeilenvorschub), schreibt man `cout.flush()` oder `cout << flush`

Dieses Verwenden des Operators `<<` mehrfach hintereinander setzt übrigens voraus, dass nach der Abarbeitung der ersten Operation für die nächste als linker Operand auch wieder ein Datenstrom zur Verfügung steht, d. h. dass das hier eingerahmte in `cout << x` `<< y;` auch einen Datenstrom darstellen

muss. Der Rückgabewert der Operation ist tatsächlich der Datenstrom (eine Referenz auf ihn), genau wie der Operator „+“ die Summe der Operanden zurückgibt (wenn auch keine Referenz).

Hier soll zunächst nicht auf Details wie Formatierung von Ausgaben eingegangen werden. Falls jetzt schon Interesse daran besteht, sei auf das Kapitel 10.2 auf Seite 71 verwiesen.

Zum Einlesen von Zeichenketten, die auch Leerzeichen enthalten können, d. h. ganzer Zeilen, dient die Methode `getline()`, deren Verwendung in Kapitel 7.4.4 auf Seite 45 erläutert wird.

Kapitel 3

Funktionen

Die Funktion `min()` ist eine ganz normale Funktion wie in C. Solche Funktionen heißen in C++ dann „freie“ Funktionen, weil sie zu keiner Klasse gehören, also keine Methoden sind.

3.1 Vergleich mit Funktionsmakro

Die Funktion zur Ermittlung des Minimums ist natürlich nur für passende Parameter verwendbar, was vordergründig als Nachteil gegenüber einem Preprocessor-Makro erscheint. Tatsächlich ist diese Einschränkung eher ein Vorteil, weil die Parameter geprüft werden. Einem Makro könnte man nämlich auch Parameter vom Typ `char*` übergeben, wobei dann aber die Adressen der Zeichenketten im Hauptspeicher verglichen würden, nicht aber die Zeichenketten selbst. Bei einer Funktion ist also die Sicherheit viel größer.

Führt man den Overhead beim Aufruf von Funktionen gegenüber der Makro-Expansion als Nachteil an, so kann man sagen, dass für den Fall, dass die Funktion wirklich sehr klein ist, diese wahlweise als `inline` deklariert werden kann, so dass der Code der Funktion jedesmal eingefügt wird, statt den Code nur einmal abzulegen und jedesmal einen echten Funktionsaufruf durchzuführen. Das Schlüsselwort `inline` wird noch vor den Rückgabetypen der Funktion geschrieben. In separatem Quelltext implementierte inline-Funktionen können nur innerhalb der Klasse verwendet werden.

3.2 Overloading

Neben verschiedenen Typen könnte man sich auch eine unterschiedliche Anzahl von Parametern bei einer Minimum-Funktion vorstellen. Das ist in C++ machbar, ohne verschiedene Funktionsnamen zu verwenden. Die Funktion heißt dann „overloaded“, auf Deutsch wird gelegentlich auch von „überladenen“ Funktionen gesprochen. Eine Funktion mit 3 Parametern kann man aus der mit 2 Parametern direkt ableiten.

```
int min (int a, int b, int c) {
  return min (a, min (b, c));
}
```

Auch eine Variante mit 4 Parametern ist wiederum ableitbar, entweder aus der mit 2 Parametern oder unter Verwendung der mit 3 Parametern, beispielsweise wie hier gezeigt.

```
int min (int a, int b, int c, int d) {
  return min (a, min (b, c, d));
}
```

Um auch Fließkomma-Parameter zuzulassen, muss man eine weitere Funktion schreiben. Natürlich könnte man sie mit Parametern vom typ `float` schreiben, aber `double` ist praktischer, weil dies mehr Werte umfaßt. Evtl. kann man sogar direkt mit `long double` arbeiten:

```
double min (double a, double b) {
  if (a < b) return a;
  return b;
}
```

Tatsächlich kann man mit einer Funktion auch das Minimum von zwei Zeichenketten ermitteln. Dabei kann es sich um die kürzere von zwei Zeichenketten handeln, aber typischer ist es, diejenige zu liefern, die in der Sortierreihenfolge weiter vorn steht. Der Vergleich kann natürlich selbst programmiert werden, aber die Funktion `strcmp()` bietet das bereits, so dass es einfacher und sicherer ist, sich dieser Standardbibliotheksfunktion zu bedienen. Das sähe dann wie folgt aus.

```
#include <cstring> // <string.h> C-Stringfunktionen

const char *min (const char *a, const char *b) {
  if (strcmp (a, b) < 0 return a;
  return b;
}
```

3.3 Matching-Regeln für Parameter

Nun stellt sich die Frage, wie der Compiler bei gleichnamigen Funktionen herausfindet, welche jeweils die richtige ist. Es wird einfach geprüft, ob die Parameter den passenden Typ haben. Technisch funktioniert das dadurch, dass an den eigentlichen Funktionsnamen die Liste der Parameter codiert angefügt wird (sogenanntes Name Mangling).

Sofern es eine Funktion gibt, deren Parameter perfekt passen, ist dieser Vorgang leicht nachvollziehbar. Leider sehen Parameter manchmal genau passend aus, sind es aber nicht. Von welchem Typ ist beispielsweise die Konstante `3.5`? `float`? Von wegen! Sie hat den Typ `double`!

Hier die Regeln für das Argument Matching:

1. Paßt genau. Wenn aktuelle und formale Parameter perfekt übereinstimmen, ist die Wahl leicht. Dabei werden manche Unterschiede ignoriert, z. B. sind die Typen `T&` und `T` identisch. Ein aktueller Parameter vom Typ `T*` paßt auf `const T*`, aber im Zweifel wird ein formaler Parameter vom typ `T*` vorgezogen, gleiches gilt für `T&` und `const T&`.

2. Paßt nach Promotion. (siehe auch Anmerkung auf Seite 14) Der Compiler versucht, aktuelle und formale Parameter durch integrale Promotion zur Deckung zu bringen, beispielsweise `float` zu `double`.

3. Paßt nach Standardkonversion. Der Compiler versucht, aktuelle und formale Parameter durch Anwendung von Standardkonversionen zur Deckung zu bringen. Alle numerischen oder Zeichentypen können ineinander konvertiert werden, außerdem können alle Zeiger zu `void*` konvertiert werden.

4. Paßt nach benutzerdefinierter Konversion (siehe Kapitel 5.1.1 auf Seite 32 und 6.4 auf Seite 40). Es werden die Konversionen angewandt, die in benutzerdefinierten Klassen definiert weden.

5. Paßt nach Ellipse. Ein Parameter in einer Ellipse `...`, d. h. einer offenen Parameterliste, gilt als noch schlechter passend. Hier wird der Parametertyp schließlich gar nicht geprüft. Ellipsen sind eher typisch für C als für C++.

Der Rückgabetyp wird übrigens **nicht** in den Namen hineincodiert. Daher darf es folgendes nicht geben:

```
void f(int);
int f(int);    // illegal, weil gleiche Parameterliste!
```

Genausowenig wird zwischen einem Argument als Value-Parameter und einem Argument gleichen Typs als Reference-Parameter (siehe Kapitel 3.5 auf Seite 17) unterschieden. Auch das ist daher nicht möglich:

Anmerkungen zu Promotions

Promotions gehen grundsätzlich von einem Typ A zu einem Typ B, wenn B den Wertebereich von A vollständig enthält. Das ist beispielsweise der Fall bei `short int` → `int` oder bei `char` → `int`. Es wird aber bei den Ganzzahlen immer nur Richtung `int` promotet, d. h. `char` wird nicht nach `short int` promotet. Auch wenn in `long int` (und sofern vorhanden auch in `long long int`) der Typ `int` vollständig enthalten ist, findet keine Promotion in Richtung auf diese Typen statt. Wohl aber gibt es eine Promotion von `float` nach `double`, nicht aber nach `long double`.
Die ganzzahligen Promotionen sind (nach Stroustrup):

1. Die Typen `char`, `signed char`, `short int` und `unsigned short int` werden in `int` promotet, wenn dies alle Werte des Quelltyps darstellen kann. Ansonsten wird in `unsigned int` promotet.
2. Der Typ `wchar_t` und Aufzählungstypen werden in den kleinsten der Typen `int`, `long` oder `unsigned long` umgewandelt, der alle Werte des Quelltyps darstellen kann.
3. Ein Bitfeld wird in ein `int` promotet, wenn alle Werte in `int` dargestellt werden können. Ansonsten wird, falls der Wertebereich ausreicht, in `unsigned int` promotet.
4. Der Typ `bool` wird nach `int` promotet, wobei `false` zu `0` und `true` zu `1` wird.

Es wird also von kleineren ganzzahligen Typen Richtung `int` und von kleineren gebrochenzahligen Typen Richtung `double` promotet. Der Datentyp `long long int` ist bislang nicht berücksichtigt, wird aber von einigen Compilern bereits angeboten.
Fazit: Man sollte nicht auf die Unterscheidung zwischen Promotions und Standardkonversionen bauen, da die Regeln recht unübersichtlich sind.

```
void f(int);
void f(int&);    // illegal, weil Unterschied nur Referenz!
```

Wohl aber ist es möglich, zwischen konstanten und nicht-konstanten Parametern zu unterscheiden. Folgendes geht also:

```
void f(int&);
void f(const int&);   // legal, weil andere Funktion!
```

Deklariert man eine Funktion in einem Block, so wird eine gleichnamige Funktion außerhalb des Blocks nicht overloaded, sondern überdeckt:

```
void f(int);

void g(void) {
  void f(double);   // verdeckt f(int)
  ...
}
```

Achtung! Das ist *kein* Overloading, sondern ein Verdecken einer Funktion, das in C genauso funktioniert. Es wird also innerhalb von `g()` in jedem Fall die Funktion `void f(double)` verwendet, sofern eine Promotion oder Standardkonversion genügt, um die aktuellen und formalen Parameter zur Deckung zu bringen. Andernfalls würde eine Fehlermeldung ausgegeben.

Beispiele zu Argument-Matching-Regeln

Fall 1

Folgende Prototypen sind gegeben:

```
void f (int, char);
void f (float, int);
```

Dann ist dieser Aufruf illegal

```
f (1, 2); // illegal, da keine Funktion am besten paßt
```

Am besten für den ersten Parameter paßt `f (int, char)`, aber für den zweiten Parameter paßt `f (float, int)` besser. Keine Funktion ist am besten für beide Parameter.

Dagegen ist der folgende Aufruf in Ordnung:

```
f (1.0, 2); // Aufruf von f (float, int)
```

Beide Funktionen sind gleich gut für den ersten Parameter, das vom typ `double` ist (**nicht `float`!**). Es ist eine Standardkonversion nötig von `double` nach `float`. Der zweite Parameter paßt auf `f (float, int)` exakt, so dass der Compiler eine Entscheidung treffen kann.

Auch dieser Aufruf ist legal:

```
f (1, 2.0); // Aufruf von f (int, char)
```

Der erste Parameter paßt am besten auf `f (int, char)`. Beide Funktionen brauchen eine Standardkonversion für den zweiten Parameter. Der Compiler wählt also die Funktion `f (int, char)`, da sie auf den ersten Parameter besser paßt und auf das zweite zumindest genauso gut.

Hier noch einmal ein illegaler Aufruf:

```
f (1.0, 2.0); // illegal, da keine Funktion am besten paßt
```

Beide Funktionen passen gleich gut auf den ersten Parameter, in beiden Fällen ist wieder eine Standardkonversion nötig. Dasselbe gilt für den zweiten Parameter, also kann der Compiler keine Entscheidung treffen.

Fall 2

Die Prototypen der overloadeten Funktionen sehen so aus:

```
void f (const int *);
void f (int *);
```

Hier die Funktionsaufrufe:

```
int *p;
const int *q;

f (p); // Aufruf von f (int *);
f (q); // Aufruf von f (const int *);
```

Obwohl Zeiger eigentlich genauso gut passen wie `const` Zeiger, wählt der Compiler doch lieber die Funktion mit dem Parameter ohne `const`, wenn der aktuelle Parameter nicht `const` ist. Bei einem aktuellen `const` Parameter kann die Funktion ohne den formalen `const` Parameter nicht gewählt werden.

Fall 3

Die Prototypen der overloaded Funktionen sehen so aus:

```
void f (signed char);
void f (unsigned char);
```

Hier der Funktionsaufruf:

```
char c;

f (c); // illegal, passt gleich gut
```

Der C++ Compiler unterscheidet *alle drei* char-Typen. Alle Umsetzungen zwischen beliebigen zwei von diesen verlangt eine Standardkonversion.

Ob der Typ `char` inhaltlich gleichbedeutend mit `signed char` oder `unsigned char` ist, ist implementationsabhängig und weder im C- noch im C++-Standard festgelegt. Manche Compiler erlauben durch Verwendung von Schaltern beim Aufruf eine Festlegung, ob der Zeichentyp ein Vorzeichen haben soll oder nicht. Schauen Sie auf den `man`-Pages der Compiler einmal nach. Ggf. steht die Information auf der Seite zum C-Compiler, wenn diese für beide Compiler gilt, weil es sich um einen integrierten C/C++-Compiler handelt.

Es ist in jedem Fall anzuraten, den Datentyp genau mit `signed` oder `unsigned` festzulegen und sich nicht auf implementationsabhängige Features zu verlassen. Übrigens gibt es inzwischen auch den Datentyp `wchar_t`, der die 16-bit-Zeichen des Unicode umfaßt.

Explizite Konversion

Möchten Sie sichergehen, dass die Parameter genau passen, dann ist es zu empfehlen, in Zweifelsfällen die Konversion selbst vorzunehmen.

```
void f (int, char);
void f (float, int);

int i;

f (float (i), i); // ruft f (float, int)
```

Vorsicht bei numerischen Literalen! Der Datentyp von nicht-ganzzahligen Literalen ist `double`, nicht etwa `float`. Durch Zusatz von `F` kann man aber auch `float`-Literale erzeugen.

```
f (1.0F, 2.0); // ruft f (float, int)
```

3.4 Default-Parameter

Nicht immer benötigt man viele Versionen einer Funktion, die man mit Overloading herstellt. Oft kann man Probleme auch durch default-Parameter lösen. Default-Parameter sind Argumente, die sich immer am Ende der Parameterliste befinden müssen und die nicht angegeben zu werden brauchen. Unsere `min()`-Funktionen sind dafür ein Beispiel. Um eine Minimum-Funktion für zwei, drei oder vier `int`-Parameter zu schreiben, genügt eine einzige mit zwei default-Parametern:

```
// Prototyp, nach aussen bekanntgemacht
int min (int, int, int c=INT_MAX, int d=INT_MAX);

// Definition
int minint (int a, int b) { // interne Hilfsfunktion
  if (a < b) return a;
  return b;
}

int min (int a, int b, int c, int d) {
  return minint (minint (a, b), minint (c, d));
}
```

Wenn man die Funktion in verschiedenen Compilationseinheiten deklariert, können die Anzahl und die Werte der default-Parameter durchaus variieren. Innerhalb derselben Compilationseinheit können weitere Deklarationen derselben Funktion weitere Parameter mit default-Werten versehen. Sie gelten immer ab der Stelle, an der sie stehen.

```
// ***************
// Datei dings.cpp
// ***************
void f (int i=0, float x=1.0); // gilt in dieser Datei

f (); // ok, i bekommt 0, x bekommt 1.0

// **************
// Datei bums.cpp
// **************
void f (int i=1, float x=0.0); // gilt in dieser Datei

f (); // ok, i bekommt 1, x bekommt 0.0

// ********************
// Datei dingsbums.cpp
// ********************
void f (int i, float x=1.0); // ein default-Parameter

f (4); // ok, i bekommt 4, x bekommt 1.0
f ();  // verboten, da kein default-Wert für i

void f (int i=0, float x); // weiterer default-Parameter
f ();  // ok, i bekommt 0, x bekommt 1.0
```

Die default-Werte für die Parameter werden also vom Compiler beim Erzeugen des Funktions-*Aufrufs* erzeugt und sind *nicht* im Object Code der Funktion selbst festgelegt.

3.5 Referenzparameter

Referenzparameter geben dem Programmierer einen „neuen“ Mechanismus der Parameterübergabe an die Hand. Tatsächlich ist er gar nicht so neu, denn FORTRAN kannte ausschließlich diesen Mechanismus, und Pascal bot ihn auch schon wahlweise an. Die Übergabe von Referenzparametern wird auch „call by reference“ genannt, im Gegensatz zum ausschließlichen „call by value“ von C.

Als „workaround“ für diesen Mangel der Sprache C hat man dort immer fleißig Zeiger übergeben, was aber sehr fehlerträchtig ist. Wer hat nicht schon einmal beim `scanf()` den Adreßoperator bei einfachen Parametern vergessen, so dass dann der `int`-Wert als Zeiger interpretiert wurde, was in den meisten Fällen zum Programmabsturz geführt hat?

Eine C-Funktion zum Tausch von zwei `int`-Werten und ihr Aufruf können so aussehen:

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

int x, y;

swap (&x, &y);   // Aufruf der Funktion mit &
```

In C++ ist das alles einfacher hinzubekommen, weil man insbesondere beim Aufruf der Funktion nicht mehr an den Adreßoperator denken muss. Aber auch in der Funktion ist das ganze Gehampele mit den Zeigern vorbei.

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

int x, y;

swap (x, y);   // Aufruf der Funktion ohne &
```

Die formalen Parameter `a` und `b` werden innerhalb der Funktion zu *Aliasen* der übergebenen aktuellen Parameter.

Als Nebeneffekt haben Referenzen auch den Vorteil, dass der Inhalt der Daten – es könnte sich ja um große Strukturen handeln – nicht kopiert wird.

Hinweis: An Referenzen können nur genau passende Parameter übergeben werden. Werte kann man konvertieren, aber Aliase müssen denselben Typ haben; ansonsten wird eine temporäre Variable angelegt, die referenziert wird. Daraus folgt, dass die übergebene Variable nicht geändert werden kann. Der Compiler gibt in einem solchen Fall auf jeden Fall eine Warnung aus.

```
float r, s;

swap (r, s); // Warnung: falscher Typ
```

Ebenso kann man Referenzen natürlich nur L-Values übergeben. Zur Erinnerung: L-Values sind all das, was links von einem Gleichheitszeichen einer Zuweisung stehen kann.

```
swap (x, y+2); // Warnung: kein L-Value
```

3.5.1 Konstante Referenzen

Referenzen können den Zusatz `const` tragen. Das bedeutet ähnlich wie bei Zeigern, dass ihr Inhalt durch die Funktion nicht verändert werden darf. So ein Unsinn: Erst übergibt man eine Referenz, weil man damit verändern darf, und dann verbietet man es wieder?

Die Übergabe per Referenz hat eben noch einen anderen, oben bereits erwähnten Vorteil gegenüber der Übergabe als Wert: Der Inhalt des Parameters wird nicht kopiert. Möchte man diesen Vorteil ausnutzen, aber trotzdem die Möglichkeit haben, einen nicht-L-Value zu übergeben und auch sicher zu sein, dass die Funktion an dem Inhalt nicht dreht, dann verwendet man eine konstante Referenz.

```
int add (const int &i, const int &j) {
   return i + j;
}

float x = 2.0;

cout << add (x, 1);
```

Tatsächlich hat `x` hier den falschen Typ (`float` anstelle von `int`. Wenn die Funktion `add()` aufgerufen wird, wird der Wert in einen temporäre Variable vom Typ `int` konvertiert. Der formale Parameter `i` ist dann ein Alias für diese temporäre Variable.

Beim zweiten Parameter ist es ähnlich. Obwohl der Typ stimmt, handelt es sich um keinen L-Value. Ein Funktionsaufruf, der die Erzeugung von temporären Variablen notwendig macht, funktioniert nur dann problemlos, d. h. ohne Warnung, wenn die Referenz konstant ist.

Wenn man eine Referenz nicht zum Verändern der Werte benutzt, sondern der Effizienz wegen, sollte man sie `const` deklarieren. Das sichert diese Eigenschaft auch dem Verwender der Funktion zu.

3.5.2 Referenzen als Funktionsrückgabewert

Referenzen sind L-Values, d. h. sie können verändert werden. Eine Besonderheit von C++ ist nun, dass Referenzen auch als Funktionswert zurückgegeben werden können. Auf diese Weise kann ein Funktionsaufruf auch auf der linken Seite einer Zuweisung stehen oder als Referenzparameter an eine andere Funktion übergeben werden.

Mit unserer Minimum-Funktion können wir da ein schönes Beispiel konstruieren. Aufgabe sei, der kleineren der beiden Variablen `x` und `y` eine 0 zuzuweisen.

```
int &min (int &a, int &b) {
    if (a < b) return a;
    return b;
}

int x, y;

min (x, y) = 0;
```

Hier ein weiteres Beispiel. In die kleinere der beiden Variablen soll ein Wert von der Tastatur eingelesen werden. Die `min()`-Funktion und die Variablen sind dieselben wie oben.

```
cin >> min (x, y);
```

Falls man einen Zeiger auf eine Variable übergeben bekommen hat, so kann man durchaus eine Referenz auf sie wieder zurückgeben. Das ist dann von Bedeutung, wenn man eine Referenz auf ein bestimmtes Array-Element zurückgeben möchte.

```
double &f (double arr[]) {
  int i;
  ...
  return arr [i]; // Rückgabe einer Referenz
}
```

Dadurch, dass `double &` als Rückgabetyp im Funktionskopf festgelegt wurde, wird bei `return arr [i]` automatisch eine Referenz auf das Array-Element zurückgegeben. Dass das Array nicht als Referenz übergeben wurde, stört nicht, denn die Übergabe als Zeiger (bzw. mit leeren eckigen Klammern, was ja gleichwertig ist) genügt. Tatsächlich kann man die Übergabe von Arrays auch nicht zusätzlich mit Referenzen konstruieren.

3.6 Auswirkungen von Referenzen auf Funktionsoverloading

Die Auswirkungen von Referenzen auf das vorher erläuterte in bezug auf Funktionen sind gering. Tatsächlich es es für das Argument Matching ohne Belang, ob ein Parameter als Referenz vereinbart wurde oder nicht. Beide sind hier völlig gleichwertig.

```
void f (int);
void f (int&);  // illegal, weil nicht unterscheidbar
```

Daher können sich zwei overloadete Funktionen auch nicht nur dadurch unterscheiden, dass eine ein Referenzparameter hat, die andere aber eines vom selben Typ ohne Referenz. Die Signaturen der Funktionen (Namen inklusive des Name Mangling für die Parameterliste) sind nämlich identisch und damit nicht unterscheidbar. Wohl aber wird zwischen Referenzen und konstanten Referenzen unterschieden:

```
void f (int&);
void f (const int&);  // legal, weil unterscheidbar
```

Die erste Variante wird für alle L-Value-Parameter aufgerufen, die zweite für alle Nicht-L-Values.

Kapitel 4

Klassen

Klassen sind die hauptsächliche Neuerung von C++ gegenüber C. Das kann man schon daran erkennen, dass die Sprache zunächst „C with classes“ genannt wurde. Klassen erlauben eine höhere Abstraktion und einen „natürlicheren“ Programmierstil. Beispielsweise kann man Objekte der Klasse `string` einfach einander zuweisen, sie mit $<$ vergleichen oder mit $+$ verketten – vorausgesetzt, die String-Klasse ist entsprechend implementiert, wie das in der C++ Standardbibliothek der Fall ist (Headerfile `<string>`).

Auch andere Operationen können für Objekte aus Klassen – sprich Variablen benutzerdefinierter neuer Typen – entsprechend denen für eingebaute Typen definiert werden. So werden die Multiplikation von komplexen Zahlen, BCD-Zahlen, Vektoren und Matrizen eine ganz einfache Sache. Man braucht dafür keine speziellen Funktionen mehr, sondern re-definiert die bereits vorhandenen Operatoren. Wenn es gut gemacht ist, kann man sogar gemischte Ausdrücke erstellen. z. B.

```
complex x = complex (1.1, 4); // komplexe Zahl mit Realteil 1.1
                              // und Imaginärteil 4.0
double  y = 7.5;
complex z;

z = x + y;   // Addition von komplexer und reeller Zahl
```

Hier werden eine komplexe Zahl und eine reelle Zahl miteinander verknüpft. Das Ergebnis ist eine komplexe Zahl. Es ist doch schön, wenn die Benutzung einer Klasse keine großen Kenntnisse darüber voraussetzt, wie man eine solche Klasse erzeugt. Das erleichtert den Einstieg in die Programmierung. Denken Sie nur einmal an die Probleme beim Zuweisen von Strings, die nun wirklich mit einem Gleichheitszeichen geschehen darf (natürlich ändert sich jetzt nicht die Behandlung von C-Strings, sondern man muss schon eine richtige String-Klasse aus der C++ Standardbibliothek benutzen).

Welche Datentypen sollte man nun erfinden? Es bieten sich folgende an:

- BCD-Zahlen (binär codierte Dezimalzahlen). Sie haben eine höhere Genauigkeit als `float` oder `double`, weil keine Umwandlung der Darstellung erfolgt. Bekanntlich ist ja bereits `0.1` als `double`-Wert nicht exakt darstellbar.
- komplexe Zahlen
- Datums- und Zeitwerte
- große Integer mit beliebiger Stellenanzahl (mehr als 32 bzw. 64 bit)
- rationale Zahlen als Brüche mit ganzzahligem Zähler und Nenner

Natürlich kann man nicht nur neue Datentypen erfinden, sondern auch bekannte mit neuer Funktionalität ausstatten:

- Arrays, die sich in ihrer Größe dynamisch ändern

- Arrays, die merken, wenn auf ein Element jenseits ihrer Grenzen zugegriffen wird
- Strings, deren Länge sich an die Bedürfnisse automatisch anpaßt und die keine vordefinierte Maximallänge haben
- Strings, deren Länge man bestimmen kann, ohne erst den ganzen Speicherbereich des Strings nach einem NUL-Zeichen durchsuchen zu müssen.

Außerdem kann man noch völlig neue Datenstrukturen erfinden, die man öfters braucht, aber in C++ selbst nicht direkt enthalten sind:

- verkettete Listen – einfach, doppelt, zirkulär
- Bäume
- Mengen
- Stacks
- Matrizen

Die C++ Standardbibliothek enthält bereits viele dieser Ideen. Schließlich war Ziel, dass man in C++ von Anfang an sehr effizient programmieren kann. Wir werden den immensen Umfang der C++ Standardbibliothek [1] zunächst ignorieren und vieles selbst entwickeln, denn der Lerneffekt ist hier das wesentliche.

Über diese allgemeinen, wiederverwendbaren Klassen hinaus werden viele Klassen entwickelt, die mit einer konkreten Aufgabenstellung im Rahmen der Anwendungsentwicklung zusammenhängen, beispielsweise Personal-, Artikel-, Bestellungs-, Rechnungs-Klasse. In Anwendungen gehört zu einer Klasse oft einer Tabelle in einer relationalen Datenbank. Die Objekte der Klasse werden dann in Tupeln der Tabelle gespeichert.

4.1 Entwurf einer Struktur für Brüche

Brüche bestehen aus einem Zähler und einem Nenner, beide sind ganzzahlig, der Nenner ist sogar positiv. In C würde man für einen Bruch eine Struktur definieren:

```
typedef struct {
  int zaehler;
  int nenner;
} bruch;
```

Das würden wir in eine Headerdatei packen, die sinnvollerweise `bruch.h` heißt. Zusätzlich würden wir noch ein paar Funktionen für den Umgang mit Brüchen hineinschreiben.

```
/* Datei bruch.h */

#ifndef __BRUCH_H
#define __BRUCH_H

typedef struct {
  int zaehler;
  int nenner;
} bruch;

bruch erzeuge (int z, int n);
```

[1] Die Standardbibliothek hieß früher einmal Standard Template Library und war von Hewlett-Packard, ist aber inzwischen erweitert, umbenannt und standardisiert worden

```
bruch add (bruch, bruch);
bruch sub (bruch, bruch);
bruch mul (bruch, bruch);
bruch div (bruch, bruch);
void print (bruch);

#endif
```

Programme, die den neuen Typ verwenden wollen, müssen jetzt `bruch.h` includen. Natürlich brauchen wir auch die Definitionen der einzelnen Funktionen, damit der Linker nicht meckert. Mit der Headerdatei haben wir ja nur den Compiler beruhigt und ihm versprochen, die Definitionen zum Link-Zeitpunkt zur Verfügung zu stellen. Sinnvollerweise nennen wir die Quelltextdatei mit den Definitionen `bruch.c`.

```
/* Datei bruch.c */

#include <stdio.h>
#include "bruch.h"
/* Man includet die eigene Headerdatei, damit
   die Uebereinstimmung der Deklarationen geprueft
   werden kann und die Typen und Konstanten
   vorhanden sind. */

int ggt (int m, int n) {
  int r;

  while (n != 0) {
    r = m % n;
    m = n;
    n = r;
  }

  return m;
} /* Funktion ggt */


bruch kuerze (bruch b) {
  bruch b1;
  int g;

  g = ggt (b.zaehler, b.nenner);
  b1.zaehler = b.zaehler / g;
  b1.nenner  = b.nenner  / g;
  if (b1.nenner < 0) {
    b1.zaehler *= -1;
    b1.nenner  *= -1;
  }
  return b1;
} /* Funktion kuerze */


bruch erzeuge (int z, int n) {
  bruch b;

  b.zaehler = z;
```

```
  b.nenner  = n;

  return kuerze (b);
} /* Funktion erzeuge */


bruch add (bruch b1, bruch b2) {
  bruch b;

  b.zaehler = b1.zaehler * b2.nenner +
              b2.zaehler * b1.nenner;
  b.nenner  = b1.nenner * b2.nenner;

  return kuerze (b);
} /* Funktion add */


/* entsprechende Funktionen fuer sub, mul, div */


void print (bruch b) {
  if (b.nenner == 1) {
    printf ("%d", b.zaehler);
  } else {
    printf ("%d/%d", b.zaehler, b.nenner);
  }
} /* Funktion print */
```

Beachten Sie, dass hier alle Brüche immer nur in gekürzter Form erscheinen. Ungekürzte Brüche kommen weder nach der Erzeugung noch nach irgendeiner Rechenoperation vor. Im folgenden werden wir dieses Beispiel nach und nach von C in C++ umschreiben.

4.2 Umbau zur Klasse

Von nun an wollen wir die Struktur zur Klasse umbauen. Das geschieht in vielen kleinen Schritten.

4.2.1 C++-gemäße Ausgabe mit `iostream`

Zunächst ändern wir die Ausgabefunktion `print()` dahingehend, dass sie `cout` verwendet. Natürlich müssen wir dazu auch `iostream` includen statt `stdio.h`.

```
void print (bruch b) {
  if (b.nenner == 1) {
    cout << b.zaehler;
  } else {
    cout << b.zaehler << "/" << b.nenner;
  }
} /* Funktion print */
```

4.2.2 Deklaration als Klasse statt als Struktur

Wir ersetzen in der Strukturbeschreibung das Wort `struct` durch `class`. Dadurch haben wir dann eine Klasse definiert. Allerdings kann man auf die Elemente in der Struktur nicht mehr zugreifen, weil sie jetzt als `privat` gelten. Daher müssen wir das Gegenteil `public` ausdrücklich dazuschreiben.

```
class bruch {
  public:
    int zaehler;
    int nenner;
};
```

Elemente vor dem `public:` sind automatisch privat. Übrigens sind – aus C++-Sicht – Strukturen nichts anderes als ein Sonderfall von Klassen, nämlich solche, bei denen alles `public` ist und die keine Methoden enthalten. Technisch können Strukturen in C++ sogar Methoden enthalten, aber dann sollte man wirklich lieber eine Klasse schreiben.

Datenelemente einer Klasse heißen (Daten-)Member (Mitglieder). Funktionselemente einer Klasse heißen (Funktions-)Member oder auch Methoden. Variablen vom Datentyp der Klasse heißen allgemein Objekte oder auch „Instances“ (Beispiele, Fälle) der Klasse. Das deutsche Wort Instanz trifft die Bedeutung nicht, wird aber gelegentlich auch verwendet.

4.2.3 Hinzufügen von Methoden

Das Besondere an Klassen aber sind die Methoden, d. h. die Funktionen zur Bearbeitung der Objekte gehören direkt zu Klasse hinzu. Daher werden sie auch in der Klasse definiert und nicht mehr außerhalb der Struktur wie in C. Diese Methoden sind im allgemeinen auch die einzige Möglichkeit, auf die Member von Objekten einzuwirken, d. h. das direkte Zugreifen (Lesen oder gar Verändern) auf die Datenelemente von Objekten soll nicht erlaubt werden. Daher werden die Member wieder in den `private` Bereich verlegt.

```
class bruch {
  public:
    void erzeuge (int zaehler, int nenner);
    void print();
  private:
    int zaehler;
    int nenner;
};
```

Die Methoden sind `public`, damit sie auch von außen benutzt werden können – im Gegensatz zu den Membern. Allerdings müssen nicht alle Methoden öffentlich zugänglich sein. Wenn sie nur innerhalb der Klasse benutzt werden, können sie `private` deklariert werden. Das bietet sich hier bei der Methode `kuerze()` an.

```
class bruch {
  public:
    void erzeuge (int zaehler, int nenner);
    void print();
  private:
    void kuerze();
    int zaehler;
    int nenner;
};
```

```
class bruch {
    int zaehler;
    int nenner;
  public:
    void erzeuge (int zaehl, int nenn);
    void print();
  private:
    void kuerze();
};
```

4.2.4 Definition von Methoden

Die Methoden wurden bislang nur deklariert, aber nicht mit Leben gefüllt. Das geschieht in der Definition. Diese kann direkt innerhalb der Klassendefinition geschehen, die sich ja für gewöhnlich in einer Headerdatei befindet. Das ist üblich bei kurzen Methoden, die dann als inline-Funktionen ohne Stack Frame umgesetzt werden. Die erzeugten Maschinenanweisungen werden bei jedem Aufruf dann erneut in das Programm eingefügt.

Gängiger ist es aber, sie in eine separate Quelltextdatei zu schreiben, um sie auch separat compilieren zu können.

Diese Quelltextdatei includet die Headerdatei und beschreibt den Inhalt der Funktion so:

```
void bruch::erzeuge (int zaehl, int nenn) {
  zaehler = zaehl;
  nenner = nenn;
  kuerze();
}
```

Der Name der Klasse und der sogenannte Scope-Operator `::` stehen vor dem Namen der Methode. Das ist notwendig, weil der Compiler sonst annähme, dass es sich um eine freie Funktion handelt. Eine weitere Besonderheit ist, dass die Member der Klasse benutzt werden können, ohne dass sie hier irgendwo deklariert worden sind. Generell gilt, dass Methoden Zugriff auf alle Member ihrer Klasse haben – egal, ob sie `public` oder `private` sind. Das liegt daran, dass sie im Zusammenhang mit einem konkreten Objekt aufgerufen werden.

4.2.5 Erzeugen von Objekten

Um Objekte einer Klasse zu erzeugen, verwendet man den Namen der Klasse als Typbezeichner. Wahlweise kann man auch das Wort `class` davorschreiben, aber das ist weniger üblich.

```
bruch b1;
class bruch b2;
```

4.2.6 Aufrufen von Methoden

Genauso wie bei Strukturen auf die Elemente mit dem `.`-Selektor (Punkt-Selektor) zugegriffen wird, geschieht es bei Klassen, wenn auf Methoden zugegriffen wird. `b1` wird hier *impliziter Parameter* der Methode `erzeuge()` genannt.

```
bruch b1;
b1.erzeuge(1,2); // weist 1/2 dem Bruch zu
```

Hier wird der Bruch mit dem Wert $\frac{1}{2}$ initialisiert. Weil immer ein impliziter Parameter – das Objekt, dessen Methode aufgerufen wird – benötigt wird, kann man die Funktion `erzeuge` auch nicht von außen ohne ein konkretes Objekt aufrufen. Innerhalb einer Member-Funktion kann man aber eine zweite Methode aufrufen, weil der implizite Parameter dann automatisch dasselbe ist. Diese Eigenschaft wird beim Aufruf von `kuerze()` innerhalb von `erzeuge()` verwendet (siehe Kapitel 4.2.4).

4.2.7 Operationen mit ganzen Objekten

Man kann mit Objekten in C++ einiges machen, ohne dafür irgend etwas definiert haben zu müssen. Zum Beispiel kann man die Werte von Objekten kopieren oder neue Objekte initialisieren:

```
bruch b1, b2;
b1.erzeuge(1,2);
b2 = b1;         // kopiert alle Datenmember
bruch b3 = b1;   // initialisiert mit anderem Objekt
```

Auch kann man Objekte als Parameter an Funktionen übergeben. Das illustrieren wir am Beispiel der Additionsmethode.

4.2.8 Methode mit Objekt als Parameter

Die Additionsmethode verwendet ein (anderes) Objekt als Werteparameter.

```
bruch bruch::add (bruch b2) {
  bruch b;

  b.zaehler =    zaehler * b2.nenner +
              b2.zaehler *    nenner;
  b.nenner  = nenner * b2.nenner;
  b.kuerze();
  return b;
} // Methode bruch::add
```

Wo ist der zweite Summand? Im impliziten Parameter; daher sieht die Verwendung so aus:

```
bruch b1, b2, b3;
// Eingabe von Werten fuer b1 und b2 ...
b3 = b1.add(b2);
```

Bei der Verarbeitung von `b2` als Werteparameter und auch bei der Rückgabe des Funktionswertes werden die Objekte komplett kopiert. Das kann bei großen Objekten Performance-Nachteile mit sich bringen, weshalb oft (konstante) Referenzen (siehe Kapitel 3.5.1 auf Seite 18) verwendet werden.

4.2.9 Die fertige Klasse `bruch`

Jetzt können wir die Dinge alle zusammenstellen und ergänzen, so dass wir eine richtige Klasse erhalten.

```
// Headerdatei bruch.h

class bruch {
public:
  void erzeuge (int zaehl, int nenn);
  bruch add (bruch);
  bruch sub (bruch);
  bruch mul (bruch);
  bruch div (bruch);
  void print();
private:
  void kuerze();
  int zaehler;
  int nenner;
};
```

Die Definition der Methoden steht in der Quelltextdatei, genauso wie die `ggt()`-Funktion

```
// Quelltextdatei bruch.cpp (oder bruch.C oder bruch.cc)

#include <iostream>
#include "bruch.h"

static int ggt (int m, int n) {
```

```
// static, damit sie lokal zu diesem Modul ist
  int r;

  while (n != 0) {
     r = m % n;
     m = n;
     n = r;
  }
  return m;
} // freie Funktion ggt()


void bruch::kuerze() {
  int g;

  g = ggt(zaehler, nenner);
  zaehler = zaehler / g;
  nenner  = nenner  / g;
  if (nenner < 0) {
    nenner  *= -1;
    zaehler *= -1;
  }
} // bruch::kuerze()


void bruch::erzeuge (int zaehl, int nenn) {
  zaehler = zaehl;
  nenner  = nenn;
  kuerze();
} // bruch::erzeuge()


bruch bruch::add (bruch b2) {
  bruch b;

  b.zaehler =    zaehler * b2.nenner +
              b2.zaehler *    nenner;
  b.nenner  = nenner * b2.nenner;
  b.kuerze();
  return b;
} // Methode bruch::add()

// andere Rechenmethoden entsprechend

void bruch::print() {
  if (nenner == 1) {
    cout << zaehler;
  } else {
    cout << zaehler << "/" << nenner;
  }
} // bruch::print()

// Ende der Datei bruch.cpp
```

4.3 Weitere Details über Klassen

4.3.1 `const` Methoden

Solange nichts anderes angegeben ist, kann eine Methode auf ihr Objekt (impliziter Parameter) beliebig zugreifen. Wenn wir also eine Methode aufrufen, müssen wir davon ausgehen, dass der Inhalt des Objekts modifiziert werden kann. Daher gibt es eine Möglichkeit, wie schon im Headerfile dem Anwender zugesichert werden kann, dass eine Methode ihr Objekt nicht verändern wird. Das ist im Beispiel bei der `print()`-Methode sinnvoll.

```
class bruch {
public:
  void print() const;
  ...
};

void bruch::print() const {
  if (nenner == 1) {
    cout << zaehler;
  } else {
    cout << zaehler << "/" << nenner;
  }
} // bruch::print()
```

Ein Versuch, eine nicht-`const` Methode auf ein `const` Objekt anzuwenden, liefert einen Fehler. Die Originalversion von `print()` könnte man also nicht auf einen formalen Parameter anwenden, der als konstante Referenz (siehe Kapitel 3.5.1 auf Seite 18) vereinbart wurde. Bei der mit `const` versehenen `print()`-Methode gibt es dagegen keine Probleme.

```
const bruch b1 = b2;

b1.erzeuge (2, 3);  // illegal
b1.print();         // ok, da print() const-Methode
```

Glücklicherweise kann man dieselbe Methode sowohl `const` als auch nicht-`const` vereinbaren, so dass dann – in Abhängigkeit davon, ob das Objekt `const` ist oder nicht – die passende Methode verwendet wird. Auch dies ist eine Form des Overloading.

```
void dings();
void dings() const;
```

4.3.2 Der `this`-Zeiger

Eine Methode kann jederzeit auf einen besonderen Zeiger namens `this` zurückgreifen, der immer auf das aktuelle Objekt (den impliziten Parameter) zeigt. Dies braucht man immer dann, wenn das aktuelle Objekt als Funktionswert zurückgeliefert werden soll. Das wäre beispielsweise der Fall, wenn wir eine Methode `max()` einführen, die einen größeren von zwei Brüchen zurückliefert. Die Methode hier liefert den *Wert* des größeren Bruchs zurück, keine Referenz auf ihn!

```
class bruch {
public:
  bruch max (const bruch &) const;
  ...
};
```

```
bruch bruch::max (const bruch &b) const {
  if (zaehler * b.nenner > nenner * b.zaehler) {
    return *this;
  } else {
    return b;
  }
}
```

Diese Methode würde so angewendet, wobei der größere der beiden Werte in `b3` kopiert würde.

```
bruch b1, b2, b3;
...
b3 = b1.max (b2);
```

4.3.3 Gültigkeitsbereich (Scope)

In der Programmiersprache C gibt es zwei Arten von Gültigkeitsbereichen: *dateiweit* (file scope) und *blockweit* (block scope). In C++ gibt es noch einen dritten Gültigkeitsbereich: *klassenweit* (class scope) – in neueren Versionen von C++ sogar noch einen vierten, nämlich namensraumweit (namespace scope, siehe Kapitel 13 auf Seite 99).

Die in einer Klasse deklarierten Namen haben Gültigkeit nur innerhalb der Klasse. Insbesondere gibt es keinen Konflikt – und kein Overloading – zwischen freien Funktionen und Methoden. Der Scope-Operator `::` kann verwendet werden, um auf Bezeichner zuzugreifen, die sonst nur innerhalb einer Klasse existieren.

Nehmen wir als Beispiel die Klasse `ios`, die Bestandteil der C++ Ein-/Ausgabebibliothek ist. In der Klassendefinition gibt es die Aufzählung `open_mode`, die die verschiedenen Datei-Modi darstellt:

```
class ios {
public:
  enum open_mode {
    in        = 0x01,
    out       = 0x02,
    ate       = 0x04,
    app       = 0x08,
    trunc     = 0x10,
    nocreate  = 0x20,
    noreplace = 0x40,
    binary    = 0x80
  };
  ...
};
```

Zugriff auf die Konstanten benötigt man, wenn man eine Datei öffnet. Da diese Namen aber nur in der Klasse existieren, beschränkt sich ihr Gültigkeitsbereich auf die Klasse. Daher sind sie außerhalb nicht sichtbar. Um diese Namen zu verwenden, schreibt man den Operator `::` mit dem Namen der Klasse als linken und den Namen der Konstanten als rechten Operanden hin, also z. B. `ios::in`.

Hierbei sehen wir auch, dass eine Klasse nicht nur Datenmember und Methoden enthalten kann, sondern auch Aufzählungen. Es sind sogar Klassen innerhalb von Klassen möglich. Ein wichtiger Zweck davon ist das Verstecken der Namen, so dass Namenskonflikte vermieden werden. In neueren Implementationen von C++ gibt es dazu aber zusätzlich die Namespaces (Namensbereiche oder -räume), siehe Kapitel 13 auf Seite 99.

Der Scope-Operator `::` wird auch dazu verwendet, auf globale Namen zuzugreifen, während man sich gerade in einer Klasse befindet, in der es einen gleichnamigen lokalen Bezeichner gibt, der den globalen Namen verdeckt. Wenn wir in der Brüche-Klasse eine `abs()`-Methode einführen möchten, brauchen wir nur den Zähler zu prüfen, weil der Nenner durch die regelmäßige Anwendung von `kuerze()` automatisch positiv ist. Wir stützen uns auf die in `<math.h>` vorhandene Funktion `abs()`.

```
#include <math.h> // C-Mathematikfunktionen

bruch bruch::abs() const {
  bruch b;
  b.zaehler = abs(zaehler);   // FEHLER!
  b.nenner  = nenner;
  return b;
}
```

Hier riefe sich die Methode selbst auf, allerdings mit falscher Parameteranzahl. Tatsächlich möchten wir die gleichnamige (freie) Funktion aufrufen, wozu wir den Scope-Operator brauchen:

```
#include <math.h>

bruch bruch::abs() const {
  bruch b;
  b.zaehler = ::abs(zaehler);   // ok
  b.nenner  = nenner;
  return b;
}
```

Jetzt beziehen wir uns nicht auf eine Klasse, sondern auf den globalen (unbenannten) Namensraum, weshalb `::` hier keinen linken Operanden hat. Der Operator wird unär (einstellig) benutzt – ähnlich wie es ja auch einen unären Minusoperator gibt, das Vorzeichen.

4.3.4 Statische Member

Es kann Datenmember einer Klasse geben, die nicht zu einem individuellen Objekt gehören, sondern zur Klasse an sich, d. h. nur ein einziges Mal existieren – unabhängig davon, ob und wieviele Objekte der Klasse es gibt. Diese Member heißen statische Member.

Hier ein Beispiel für Konten. Jedes Objekt stellt ein Konto dar. Das statische Datenmember enthält stets den Gesamtsaldo über alle Konten.

```
class Konto {
public:
  void create (int anfangsSaldo = 0) {
    saldo = anfangsSaldo; gesamtSaldo += anfangsSaldo;
  }

  void einzahlung (int betrag) {
    saldo += betrag; gesamtSaldo += betrag;
  }

  void abhebung (int betrag) {
    saldo -= betrag; gesamtSaldo -= betrag;
  }

private:
  static int gesamtSaldo;  // ueber alle Konten
  int saldo;  // fuer einzelnes Konto
}; // class Konto
```

Jedes Konto hat seine eigene Variable `saldo`, aber alle Objekte gemeinsam bedienen sich der Variable `gesamtSaldo`. Jedes Objekt hat die Größe eines `int`-Wertes.

Obwohl `gesamtSaldo` deklariert ist, hat der Compiler noch keinen Speicherplatz hierfür allokiert. Tatsächlich müssen alle statischen Datenmember außerhalb der Klassendefinition in der `.cpp`-Datei initialisiert werden. Das kann so aussehen:

```
int Konto::gesamtSaldo = 100;
```

Natürlich muss wieder der Scope-Operator verwendet werden, um klarzumachen, zu welcher Klasse die Variable gehört. Es ist auch bemerkenswert, dass dieses private Datenmember außerhalb der Klassendefinition erzeugt und initialisiert werden darf. Weitere Zugriffe darauf sind aber nicht möglich.

Dieses Beispiel zeigt wieder einmal deutlich, dass es unabdingbar ist, die Datenmember vor Veränderung von außen zu schützen und eine Beeinflussung ihrer Werte aussschließlich über Methoden zuzulassen. Hier wäre es sonst möglich, den `saldo` eines Objekts oder den `gesamtSaldo` zu ändern, so dass anschließend die Summe aller Salden nicht mehr den `gesamtSaldo` ergibt. Durch den Schutz ist sichergestellt, dass keine widersprüchlichen Werte auftreten können.

Neben statischen Datenmembern gibt es auch statische Methoden, d. h. statische Funktions-Member. Diese werden beispielsweise verwendet, um auf den Wert eines statischen Datenmembers zuzugreifen. Innerhalb der `Konto`-Klasse könnte man beispielsweise schreiben:

```
class Konto {
public:
  ...
  static int gesamt() {
    return gesamtSaldo;
  }
}; // class Konto
```

Durch die Deklaration mit `static` kann diese Funktion auch ohne aktuelles Objekt aufgerufen werden. Sie darf aber auch nur auf statische Datenmember zugreifen, also nicht auf das Member `saldo`.

```
cout << "Gesamtsaldo: " << Konto::gesamt() << endl;
```

Wahlweise kann man die Methode auch mit einem beliebigen Objekt der Klasse aufrufen: `konto1.gesamt()`.

Kapitel 5

Konstruktoren und Destruktoren

Wenn wir neue Objekte anlegen, so werden diese automatisch mit Nullen initialisiert. Allein schon bei unserer Brüche-Klasse ist das nicht angebracht, weil der Bruch den Wert $\frac{0}{0}$ bekäme, der bekanntlich nicht definiert ist. Bei komplexeren Klassen kann auch eine weitere Initialisierung notwendig sein als nur die Belegung mit Werten – beispielsweise das Allokieren von dynamischem Speicherplatz oder das Holen von Werten aus einem nichtflüchtigen Speicher.

5.1 Konstruktoren

Zur korrekten Initialisierung von Objekten dienen Konstruktoren. Pro Klasse kann es davon durchaus mehrere geben, genauso wie es auch overloadete Funktionen desselben Namens geben kann. Sie müssen sich entsprechend in ihrer Parameterliste unterscheiden. Ein Konstruktor ohne Parameter oder mit Parametern, die alle mit einem default-Wert versehen sind, heißt default-Konstruktor. Er wird immer bei der Erzeugung eines neuen Objekts der Klasse benutzt, auch wenn man ihn nicht explizit aufruft.

Konstruktoren tragen denselben Namen wie die Klasse selbst, haben aber keinen Rückgabetyp – auch nicht `void`.

```
class bruch {
public:
  ...
  bruch (int zaehl, int nenn) {
    zaehler = zaehl; nenner = nenn;
    kuerze();
  }
  bruch (int zaehl=0) {
    zaehler = zaehl; nenner = 1;
  }
}; // class bruch
```

5.1.1 Default-Konstruktoren

Hier sind zwei Konstruktoren dargestellt: ein default-Konstruktor, der den Bruch korrekt mit dem Wert 0 initialisiert, wenn man keinen Parameter angibt. Gleichzeitig ist er auch ein Konstruktor, der mit einem `int`-Wert aufgerufen werden kann – daher ist er auch ein **Typkonversions-Konstruktor**. Der zweite Konstruktor muss mit zwei `int`-Werten aufgerufen werden.

Tatsächlich ist es so, dass ein Klasse keine Konstruktoren zu haben braucht. Gibt es allerdings einen Konstruktor, kann man keine Objekte der Klasse mehr erzeugen, ohne einen Konstruktor zu verwenden. Wenn es nur einen Konstruktor mit einem Parameter gibt, kann man keine Objekte mehr ohne Angabe dieses Parameters anlegen. Um Objekte ohne Parameter anlegen zu können, braucht man dann zusätzlich noch einen default-Konstruktor, d. h. einen, den man (auch) ohne Parameter aufrufen kann.

```
bruch b1;           // ruft default-Konstruktor auf, Wert 0/1
bruch b2 = bruch(); // ruft default-Konstruktor auf, Wert 0/1
bruch b3 (2);       // ruft Konstruktor auf, Wert 2/1
bruch b4 (1, 2);    // ruft Konstruktor auf, Wert 1/2

bruch b5();      // deklariert Funktion mit Rückgabetyp bruch
```

5.1.2 Element-Initialisierungslisten

Ein Konstruktor kann auch eine Element-Initialisierungsliste vor dem Funktionsrumpf verwenden. Das sieht dann so aus:

```
class bruch {
public:
  ...
  bruch (int zaehl=0, int nenn=1):
         zaehler(zaehl), nenner(nenn) {
    kuerze();
  }
}; // class bruch
```

Dieser Konstruktor kann wahlweise ohne, mit einem oder mit zwei Parametern aufgerufen werden, ist also gleichzeitig ein default- und ein Typkonversions-Konstruktor für die Konversion von `int` nach `bruch`. Element-Initialisierungslisten werden insbesondere bei abgeleiteten Klassen verwendet, siehe Kapitel 9.3.1 auf Seite 57.

```
bruch b;

b = bruch (2); // Funktions-Schreibweise
b = (bruch) 2; // Cast-Schreibweise

bruch b1 = 3;  // impliziter Aufruf des Konversions-Konstruktors

b2 = 5;        // impliziter Aufruf des Konversions-Konstruktors

b2 = 4.5;      // 4.5 wird in int konvertiert, dann Aufruf
               // des Konversions-Konstruktors int -> bruch
```

Ein Wort der Warnung: Der Compiler kann immer Konstruktoren mit (genau) einem Parameter verwenden, um Werte zu konvertieren – unabhängig davon, ob der Programmierer sie dafür vorgesehen hat oder nicht.

5.2 Dynamischer Speicher

Dynamischer Speicher (auch Freispeicher genannt) ist ein Speicherbereich, von dem man zur Programmlaufzeit Speicherblöcke anfordern kann. In C finden die Anforderung und die Rückgabe von Speicher mit den Bibliotheksfunktionen `malloc()` (ggf. auch `calloc()`, `realloc()` und `alloca()`) und `free()` statt. In C++ dagegen ist die Verwaltung von dynamischem Speicher Bestandteil der Sprache selbst (nicht der Bibliothek). Es gibt hierzu die Operatoren `new` und `delete`.

Der Parameter für den Operator `new` ist eine Typangabe, der Rückgabewert ist der angeforderte Zeiger bzw. `0`, falls kein Speicher allokiert werden konnte. Bei neueren Implementationen wird statt dessen eine Ausnahme erzeugt (siehe Kapitel 12 auf Seite 91), es sei denn, man übergibt den Parameter `nothrow`, z. B. `x = new (nothrow) char [100]);`.

Anmerkung: In C++ wird `NULL` für den Nullzeiger nicht mehr verwendet, statt dessen schreibt man die Zahl `0`. Möchte man ein Makro `NULL` verwenden, so definiere man es als `const int NULL = 0;` – daher ist das Makro in `stdio.h`, das ja von C und C++ verwendet wird, auch so definiert:

```
#ifdef __cplusplus
#define NULL 0
#else
#define NULL (void*)0
#endif
```

```
int *intZeiger1, *intZeiger2,

intZeiger1 = new int; // belegt Speicher für 1 int

if ((intZeiger2 = new int) == 0) {
   // Fehler, Speicher nicht allokierbar
}
```

Mit `new` kann man aber nicht nur Speicher für Variablen der Grundtypen, sondern für Objekte aller Art holen. Um auf Daten- oder Methodenmember von dynamischen Objekten zuzugreifen, verwendet man `->` anstelle des Punktes – genauso wie in C.

```
bruch *b = new bruch;

b->print();
```

Durch Angabe einer Anzahl in eckigen Klammern kann man auch Platz für mehrere gleichartige Objekte holen, d. h. für ein Array.

```
const int anz = 10;
bruch *bruchArray;

bruchArray = new bruch [anz]; // holt Speicher fuer 10 Brueche
```

Im Gegensatz zu statischem Speicher ist dynamischer Speicher wie automatischer Speicher **nicht** initialisiert. Allerdings kann man eine Initialisierung bei `new` angeben.

```
int *intZeiger;
intZeiger = new int (0);
```

Leider läßt sich die Initialisierung nicht mit einer Anzahl kombinieren, d. h. man kann nicht automatisch alle Array-Elemente mit demselben Wert initialisieren lassen. Wohl aber kann man für jedes Array-Element einen Wert angeben. Da die Anzahl der Initialisierungselemente die Anzahl der Array-Elemente bestimmt, können die eckigen Klammern sogar leer bleiben – genau wie bei C-Zeichenketten.

```
int *intZeiger;

intZeiger = new int [] = {3,4,5};
```

Werden Klassenobjekte von `new` angelegt, werden aber sowieso die entsprechenden Konstruktoren verwendet, die eine ordnungsgemäße Initialisierung vornehmen sollten.

Der Operator `delete` gibt vorher mit `new` angeforderten Speicher wieder frei. Beim Freigeben von Arrays muss man ein Paar leere eckige Klammern mit angeben. Die Anwendung von `delete` auf Zeiger, der nicht von `new` stammt, ist verboten. Zusätzliche oder fehlende eckige Klammernpaare sind ebenfalls illegal. Wohl aber kann man `delete` auf einen Zeiger anwenden, der 0 enthält (`NULL`-Zeiger) – es passiert dann nichts.

```
delete intZeiger;

delete [] bruchArray;
```

5.3 Destruktoren

Nicht nur für den Fall der Konstruktion eines Objektes, sondern auch für den Fall der Zerstörung eines Objektes kann man eine Funktion vorsehen: den Destruktor. Der Destruktor wird automatisch aufgerufen, wenn ein Objekt aufhört zu existieren.

Wann genau ist das der Fall? Das hängt davon ab, ob das Objekt statisch ist (globale Variablen und lokale Variablen mit dem Attribut `static`), ob es automatisch ist (lokale Variable in einer Funktion) oder ob es dynamisch ist (vom Operator `new` erzeugt). Statische Objekte existieren während der gesamten Laufzeit des Prozesses, so dass ihr Destruktor erst mit dem Verlassen des Programms aufgerufen wird (Vorsicht: Bei einigen Compilern werden die Destruktoren für statische Objekte gar nicht aufgerufen, wenn man das Programm mit `exit()` verlässt statt mit `return` in der `main()`-Funktion. Verlässt man es mit `_exit()`, werden garantiert keine Destruktoren aufgerufen, keine Dateipuffer geschrieben usw.)

Automatische Objekte hören auf zu existieren, wenn die Funktion, zu der sie lokal sind, verlassen wird (genauer: wenn der Block, zu dem sie lokal sind, verlassen wird).

Dynamische Objekte hören auf zu existieren, wenn sie mit `delete` deallokiert werden.

Der Name eines Destruktors ist derselbe wie der der Klasse, aber mit einer Tilde „~“ davor. Destruktoren haben wie Konstruktoren keinen Rückgabetyp (auch nicht `void`). Es kann pro Klasse immer nur einen Destruktor geben, er hat keine Parameter (und kann daher auch nicht overloadet werden).

5.3.1 Einsatzmöglichkeiten

Wozu dienen Destruktoren nun? Zum einen kann man sie verwenden, um zu beobachten, wann Variablen „out of scope“ gehen, d. h. ihren Gültigkeitsbereich verlassen. Zum anderen haben sie aber auch einen „echten“ Einsatzzweck, nämlich die Freigabe von Speicher, der von einem Konstruktor belegt wurde. Ohne dies könnte es sehr leicht zu einem „Speicherleck“ (memory leak) kommen, bei dem im Laufe der Zeit immer wieder Speicher von Konstruktoren belegt wird, ohne dass dieser wieder freigegeben wird. Allerdings wird der gesamte Speicher spätestens bei Prozessende wieder freigegeben.

Hauptanwendung für Destruktoren sind also die Klassen, die Zeiger auf dynamisch allokierten Speicher enthalten. Eine `String`-Klasse ist dafür das einfachste Beispiel (auch wenn es eine ähnliche Klasse bereits in der C++ Standardbibliothek gibt):

```
class String {
  public:
    String (char *s);               // Konversions-Konstruktor
    String () {text = 0; len = 0; } // default-Konstruktor
    ~String() {delete [] text; }    // Destruktor
    // ... weitere Methoden ...
  private:
    char *text;
    int len;
};

String::String (char *s) {
  len = strlen (s);
  text = new char [len+1];
  strcpy (text, s);
}
```

Der Konstruktor `String (char *)` holt dynamisch Speicher für die C-Zeichenkette, während der Destruktor den Speicher automatisch wieder freigibt, wenn das Objekt aufhört zu existieren.

5.4 Kopierkonstruktor

Wenn ein Objekt durch ein anderes Objekt initialisiert wird, werden die einzelnen Datenmember einfach kopiert. Bei den Brüchen war das so in Ordnung: `bruch b2 = b1;` kopiert Zähler und Nenner von einem Bruch in den anderen, der neu erzeugt wird.

Für manche Klassen dagegen ist dieses Verhalten nicht akzeptabel, beispielsweise für die Klasse `String`. Würden dort die beiden Member `text` und `len` einfach nur kopiert, würden beide `text`-Zeiger auf denselben Speicherplatz zeigen. Die Anwendung des Destruktors auf eines der beiden Objekte würde den gemeinsam benutzten String deallokieren und den Zeiger `text` des anderen Objekts ungültig machen.

Wir brauchen hier einen besonderen Kopierkonstruktor, der dafür sorgt, dass der Text, auf den `text` zeigt, tatsächlich kopiert wird – und nicht der Zeiger. Ein solcher Kopierkonstruktor hat die Form `X::X(const X&)`, wenn die Klasse `X` heißt, und wird automatisch verwendet, wenn ein Objekt mittels eines anderen initialisiert wird.

Der Kopierkonstruktor für die `String`-Klasse sähe so aus:

```
String::String (const String& s) {
  len = s.len;             // Länge kopieren
  text = new char [len+1]; // Platz für neuen String holen
  strcpy (text, s.text);   // Text kopieren
}
```

Kapitel 6

Overloaden von Operatoren

Bislang haben wir nur Funktionen overloadet. Tatsächlich kennen aber die meisten Programmiersprachen overloadete Operatoren, nur kann man sie als Programmierer nicht selbst erzeugen oder für neue Datentypen erweitern. Das geht aber in C++! Beispielsweise erlauben fast alle Programmiersprachen, denselben Operator für die Addition von Ganzzahlen und Fließkommazahlen zu verwenden, obwohl die Operation, die dann ausgeführt werden muss, ganz verschieden ist. Der Kontext der Operanden bestimmt also die Funktionsweise des Operators (Ganzzahl- oder Fließkomma-Operation).

Die Funktionsweise der meisten Operanden kann auf die Verwendung mit Objekten von Klassen erweitert werden. Dabei können allerdings die Priorität und die Assoziativität nicht beeinflusst werden. Operanden ermöglichen eine intuitive Schreibweise im Programm, beispielsweise die „Addition“ von Zeichenketten, um sie zu verketten.

6.1 Operator-Funktionen

Mit unseren Brüchen könnten wir leichter arbeiten, wenn wir nicht immer die Methoden `add()`, `sub()`, `mul()` und `div()` aufrufen müssten, sondern direkt mit den gewohnten arithmetischen Operatoren arbeiten könnten. Genau das definieren wir jetzt.

```
class bruch {
  private:
    ...
  public:
    bruch operator* (bruch b2);
    ...
};

bruch bruch::operator* (bruch b2) {
  int z = zaehler * b2.zaehler;
  int n = nenner  * b2.nenner;
  return bruch (z, n);  // Verwendung eines Konstruktors!!
}
```

Wenn eine Operatorfunktion als Klassenmember definiert wird, ist einer der Operanden stets implizit vorhanden. Daher hat die `*`-Operatorfunktion nur einen Parameter, obwohl der Operator ja nicht unär, sondern binär ist.

Eine Operatorfunktion braucht allerdings kein Klassenmember zu sein; es genügt, wenn mindestens einer ihrer Parameter ein Klassenobjekt ist. Würde die `bruch`-Klasse die Methoden `gibZaehler()` und `gibNenner()` zur Verfügung stellen, so könnte man die Multiplikation auch außerhalb der Klasse definieren:

```
bruch operator* (bruch b1, bruch b2) {
  int z = b1.gibZaehler() * b2.gibZaehler();
  int n = b1.gibNenner()  * b2.gibNenner();
  return bruch (z, n);  // Verwendung eines Konstruktors!!
}
```

Die Verwendung des Operators bleibt völlig gleich, egal ob er als Klassenmember definiert wurde oder nicht:

```
bruch b1 (1, 2), b2 (5, 7), b3;

b3 = b1 * b2;
```

Ist der Operator ein Klassenmember, dann ist die obige Multiplikationsanweisung identisch mit `b3 = b1.operator* (b2);`. Im anderen Fall ist sie identisch mit `b3 = operator* (b1, b2);`, d. h. die Operatorfunktion wird als freie Funktion aufgerufen.

6.2 Friends (Freunde)

Neben der Deklaration von Membern (Daten- und Funktions-Member) können in einer Klasse auch befreundete Funktionen (friend functions) angegeben werden. Obwohl eine befreundete Funktion kein Member der Klasse ist, hat sie doch Zugriff auch auf die privaten Member der Klasse.

Operatorfunktionen werden oft als Freunde konstruiert. Beispielsweise könnten wir die arithmetischen Member-Funktionen `add`, `sub`, `mul` und `div` durch befreundete Operatorfunktionen ersetzen.

```
class bruch {
  friend bruch operator+ (bruch, bruch);
  friend bruch operator- (bruch, bruch);
  friend bruch operator* (bruch, bruch);
  friend bruch operator/ (bruch, bruch);
  ...
};
```

Da befreundete Funktionen keine Member sind, gibt es keinen Unterschied ob man sie im `public`- oder im `private`-Teil deklariert. Per Konvention schreibt man sie ganz an den Anfang einer Klasse.

Die Definition erscheint oft außerhalb der Klasse. Das Schlüsselwort `friend` lässt man dort weg. Den Scope-Operator braucht man dort auch nicht, weil es sich ja nicht um Member handelt.

```
bruch operator* (bruch b1, bruch b2) {
  int z = b1.zaehler * b2.zaehler;
  int n = b1.nenner  * b2.nenner;
  return bruch (z, n);
}
```

Der Unterschied zur obigen freien Funktion, die nicht mit der Klasse befreundet war, ist der hier erlaubte direkte Zugriff auf die privaten Datenmember. Eine Veränderung der übergebenen Brüche wird hier durch die Übergabe als Wert verhindert. Man hätte auch konstante Referenzen verwenden können.

Wir zeigen die verschiedenen Möglichkeiten des Operator-Overloadings an Hand der vier arithmetischen Operatoren. Schreiben Sie die zugehörige `.cpp`-Datei mit den Funktions-Definitionen. Verwenden Sie in den Operatorfunktionen die bisherigen Methoden. Lassen Sie dabei die Subtraktion auf der Addition aufbauen und die Division auf der Multiplikation.

```
class bruch {
  // Addition als "friend"
  friend bruch operator+ (bruch, bruch);
```

```
  public:
    ... // Konstruktoren und weitere
        // bekannte Funktionen
    int gibZaehler();
    int gibNenner();

    // Subtraktion und Division als Methoden
    bruch operator- (bruch);
    bruch operator/ (bruch);
  private:
    int zaehler;
    int nenner;
};

// Multiplikation als freie Funktion
bruch operator* (bruch, bruch);
```

6.3 Welche Operatoren können overloadet werden?

Alle C++-Operatoren können overloadet werden außer `::` (Gültigkeitsbereich, Scope), `.` (Elementauswahl, Selektor), `sizeof` (Größe), `.*` (Elementauswahl durch Funktionszeiger) und `?:` (bedingter Ausdruck – igitt!).

Falls man `=`, `[]`,`()` oder `->` overloaden möchte, müssen diese Klassenmember und nicht befreundete Operatorfunktionen sein. Die Anzahl Operanden, die Priorität und die Assoziativität der Operatoren bleiben gleich.

Die Operatoren zum Inkrementieren `++` und Dekrementieren `--` kann man allerdings getrennt für ihre Prefix- und ihre Postfix-Notation overloaden.

```
operator++();     // overloadet die Prefix-Variante
operator++(int);  // overloadet die Postfix-Variante
```

Bei der Postfix-Variante wird dem (eigentlich überflüssigen) Operanden, den man beim Aufruf ja gar nicht angibt, immer eine `0` übergeben – aber verwenden sollte man ihn nicht.

6.4 Overloaden des Zuweisungsoperators

Als Default für neue Klassen wird die Kopie eines Objekts memberweise durchgeführt – wie bei der Initialisierung eines Objektes aus einem anderen auch. Sollte dies nicht angemessen sein, so kann man den Zuweisungsoperator neu definieren oder die Zuweisung zwischen Objekten völlig verbieten.

Das Verbieten der Zuweisung ist besonders einfach, dazu braucht man sie lediglich im `private`-Teil der Klasse zu deklarieren.

Eine `String`-Klasse (siehe Kapitel 5.3.1 auf Seite 35) ist ein Beispiel dafür, dass man die Zuweisung neu definieren muss, wie wir oben bereits gesehen haben. So sollte man die Zuweisung neu definieren:

```
String& String::operator= (const String &s) {
  delete [] text; // Platz für alten Text freigeben
  len = s.len; // Länge kopieren
  text = new char [len+1]; // neuen Platz holen
  strcpy (text, s.text); // String kopieren
  return *this;
}
```

Für die Bedeutung von `this` siehe Kapitel 4.3.1 auf Seite 28. Der Funktionsrückgabewert ist eine Referenz auf das aktuelle Objekt, damit die Verkettung mehrerer Zuweisungen standardgemäß funktioniert. Beispielsweise funktioniert `a = b = 7` dergestalt, dass `b` den Wert `7` zugewiesen bekommt und anschließend der neue Wert der Variablen `a` zugewiesen wird. Der Operator `=` ist rechts-assoziativ, d. h. die rechte Zuweisung wird vor der linken ausgeführt. Dieses Verhalten, das aus C für die Standard-Datentypen bereits lange bekannt ist, sollte auch für die neuen Objekt-Datentypen genau so implementiert werden, um Überraschungen zu vermeiden. Tatsächlich ist der Prototyp `X& X::operator= (const X&)` genau so für eine Klasse `X` sogar vorgeschrieben.

6.5 Benutzerdefinierte Konversionen

Eine Klasse kann eine oder mehrere Methoden enthalten, die zur Konversion von Klassenobjekten in andere Datentypen dienen. Beispielsweise könnte die Klasse `bruch` Funktionen zur Konversion in `int` und `double` enthalten.

```
class bruch {
  public:
    operator int() { return zaehler / nenner; }
    operator double() {
      return double (zaehler) / nenner;
    }
  ...
};
```

Diese benutzerdefinierten Konversionen werden, wann immer notwendig, automatisch aufgerufen.

```
bruch b;
int i;
double d;

i = b;   // identisch mit i = int (b);
d = b;   // identisch mit d = double (b),
```

Zur Wiederholung: Konstruktoren dienen dazu, Konversionen auszuführen, deren Ergebnis ein Klassenobjekt ist, während die hier erläuterten benutzerdefinierten Konversionen dazu dienen, Konversionen mit Klassenobjekten durchzuführen. Das lässt sich an einer kleinen Klasse für logische Werte schön zeigen. Der Konstruktor macht ein boolesches Objekt aus einem `int`-Wert, während die Konversion von einem booleschen Objekt zu einem `int` mit Hilfe einer benutzerdefinierten Konversion erfolgt.

```
class boolean {
  public:
    boolean (int i) { value = i != 0; }
    operator int()  { return value; }
  private:
    int value;
};
```

Ein Wort der Warnung: Der Compiler kann immer benutzerdefinierte Konversionen verwenden, um Werte zu konvertieren. Durch die in C++ enthaltenen impliziten Konversionen ist es nicht immer leicht, alle möglichen Kontexte zu überblicken, in denen sie verwendet werden.

Kapitel 7

Grundlagen der C++ I/O-Bibliothek

Obwohl man in C++-Programmen durchaus die C-I/O-Bibliothek benutzen kann, ist doch sehr dazu zu raten, auf die C++-Bibliothek umzustellen. Nur sie ist objektorientiert, wesentlich sicherer in der Anwendung und vor allem leicht für die selbst definierten Klassen erweiterbar. Ein Verständnis der C++-I/O-Bibliothek hilft auch sehr beim Verständnis der Sprache C++ an sich.

In diesem Kapitel werden allerdings nur die Grundlagen der doch recht komplexen Bibliothek erläutert. Immerhin werden die dort angebotenen Klassen und die Methoden dargestellt und es wird die Möglichkeit gegeben, selbst <<- und >>-Operatoren für eigene Klassen zu definieren. Natürlich wird auch die Dateiarbeit besprochen.

7.1 Headerfiles der I/O-Bibliothek

Die I/O-Bibliothek hat nicht nur eine, sondern mehrere Headerdateien:

- `<fstream>`
  Liefert Datei-Ein- und Ausgabe (file i/o). Werden spezifische Dateioperationen gefordert (nicht nur generische Stream-Operationen), dann verwendet man diese Headerdatei.
- `<iomanip>`
  Stellt I/O-Manipulatoren zur Verfügung. Beeinflussen das Verhalten von Datenströmen, z. B. Formatierung und Zahlendarstellung (dezimal, hex).
- `<iostream>`
  Liefert Datenstrom-Ein- und Ausgabe (stream i/o), d. h. Ein- und Ausgabe auf alle Arten von Datenströmen.
- `<strstream.h>` (alt) bzw. `<sstream>` (neu)
  Liefert Ein- und Ausgabe in Strings (incore formatting, entsprechend `sprintf()` und `sscanf()` in C.

Die völlig veraltete I/O-Bibliothek mit der Headerdatei `<stream.h>` wird hier nicht weiter erläutert.

7.2 Datenströme

Die `iostream` Bibliothek stellt drei Klassen zur Verfügung, die den drei Arten von Datenströmen entsprechen:

- `istream`: Eingabeströme
- `ostream`: Ausgabeströme

- `iostream`: Ein- und Ausgabeströme

Ein-/Ausgabe-Operationen können auf Objekten dieser drei Klassen ausgeführt werden. Alle Operationen, die für eine der beiden Klassen `istream` oder `ostream` erlaubt sind, können auch auf Objekten der Klasse `iostream` ausgeführt werden. Beispielsweise ist `<<` (Insertion) für `ostream`-Objekte definiert, aber `>>` (Extraktion) für `istream`-Objekte. Also sind beide erlaubt für `iostream`-Objekte.

Es gibt vier vordefinierte Datenströme:

- `cin`: Standardeingabe, `istream`
- `cout`: Standardausgabe, `ostream`
- `cerr`: Standardfehler (ungepuffert), `ostream`
- `clog`: Standardfehler (gepuffert), `ostream`

Sie entsprechen den `FILE*` Dateizeigern `stdin`, `stdout` und `stderr` in C. Eine gepufferte Fehlerausgabe gibt es in C nicht.

Obwohl `cout` gepuffert ist, ist es an `cin` „gebunden“ (tied), d. h. wenn eine Eingabe von `cin` gelesen werden soll, wird vorher der Puffer von `cout` entleert.

7.3 Ausgaben

7.3.1 Der Operator <<

Mit Hilfe des Operators `<<`, der für alle Standard-Datentypen bereits vorhanden ist, kann man Ausgaben auf `ostream`-Objekte machen – natürlich auch auf `iostream`-Objekte. Darüber hinaus kann man auch Methoden der `ostream`-Klasse aufrufen.

Der Operator `<<` wird für alle neuen Klassen, bei denen eine Ausgabe Sinn hat, overloadet. Tatsächlich ist die eigentliche Bedeutung dieses von C übernommenen Operators das bitweise Verschieben nach links, und die des Operators `>>` das bitweise Verschieben nach rechts. Sie sind also schon durch die Standard-I/O-Bibliothek overloadet für die Ein-/Ausgabe.

Hier eine Liste der `<<`-Operatoren für die Klasse `ostream`:

```
// Zeichen
ostream& operator<<(signed char);
ostream& operator<<(unsigned char);
ostream& operator<<(char);

// Ganzzahlen
ostream& operator<<(short);
ostream& operator<<(unsigned short);
ostream& operator<<(int);
ostream& operator<<(unsigned int);
ostream& operator<<(long);
ostream& operator<<(unsigned long);
// ggf. weitere für long long (64-Bit-Zahlen)

// Fließkommazahlen
ostream& operator<<(float);
ostream& operator<<(double);
ostream& operator<<(long double);

// C-Zeichenketten
ostream& operator<<(const char *);
```

```
ostream& operator<<(const signed char *);
ostream& operator<<(const unsigned char *);

// Zeiger
ostream& operator<<(void *);
```

Die <<-Operatoren geben eine Referenz auf ihren linken Operanden zurück, so dass mehrfache Insertionen direkt hintereinander möglich sind. Beispielsweise bedeutet `cout << i << j;` eigentlich `(cout << i) << j;`. Das Ergebnis von `cout << i` ist `cout` im Zustand nach der Insertion. Dieser Datenstrom ist dann der linke Operand für die Insertion von `j`.

Wie alle Operatoren behalten auch die Insertion und die Extraktion die Priorität und die Assoziativität bei, egal, ob sie overloadet wurden oder nicht. Daher muss man ggf. Klammern verwenden.

7.3.2 Ausgabe von Zeigern

Wie oben gezeigt gibt es auch eine overloadete Version des Insertionsoperators für nicht typisierte Zeiger (`void *`). Zeiger – mit Ausnahme von Zeigern auf Zeichen – werden als hexadezimale Adressen ausgegeben. Zeiger auf Zeichen werden dagegen als C-Zeichenketten aufgefasst. Möchte man die Adresse einer C-Zeichenkette ausgeben, muss man sie vorher nach `void *` casten:

```
char zk[] = "Guten Tag!";
cout << zk << endl; // gibt Zeichenkette aus
cout << (void *) zk << endl; // gibt Hex-Adresse aus
```

7.3.3 Ausgabemethoden (`put()`, `write()`)

Speziell zum Zweck binärer Ausgaben gibt es einige Ausgabemethoden: `put()` und `write()`:

```
ostream& put (char); // Ausgabe eines Zeichens
ostream& write (const char *, int);
// Ausgabe einer C-Zeichenkette oder einer beliebigen Bytefolge
```

(Tatsächlich gibt es von `write()` mehrere overloadete Versionen wegen Zeichenketten aus `signed char` und `unsigned char`. Das ignorieren wir hier einmal, weil sie immer analog gebildet werden.)

Bei der Methode `write()` muss man neben dem Zeiger auf die Zeichenkette immer auch die Anzahl der auszugebenden Zeichen angeben. Daher kann sie nicht nur Null-terminierte Zeichenketten, sondern auch binäre Daten ausgeben, d. h. Strings, die Nullen enthalten. Hier bietet `write` mehr als der Insertionsoperator <<. Daher wird diese Methode immer dann verwendet, wenn man Daten binär ausgeben möchte.

Die hier genannten Methoden sind das Gegenstück zu `get()` (siehe Abschnitt 7.4.3 auf Seite 45) und `read()` (siehe Abschnitt 7.4.4 auf Seite 45).

7.4 Eingaben

Eingaben werden – analog zu den Ausgaben – mit einer overloadeten Version des >>-Operators durchgeführt oder durch Aufruf einer Methode der Klasse `istream`.

7.4.1 Der Operator >>

Diese Extraktionsoperatoren sind in der Klasse `istream` vorhanden:

```
// Zeichen
istream& operator>>(signed char&);
istream& operator>>(unsigned char&);
```

```
istream& operator>>(char&);

// Ganzzahlen
istream& operator>>(short&);
istream& operator>>(unsigned short&);
istream& operator>>(int&);
istream& operator>>(unsigned int&);
istream& operator>>(signed long&);
istream& operator>>(unsigned long&);
// ggf. weitere für long long (64-Bit-Zahlen)

// Fließkommazahlen
istream& operator>>(float&);
istream& operator>>(double&);
istream& operator>>(long double&);

// C-Zeichenketten
istream& operator>>(char *);
istream& operator>>(signed char *);
istream& operator>>(unsigned char *);
```

Entsprechend den Insertionsoperatoren geben auch die Extraktionsoperatoren eine Referenz auf ihren linken Operanden zurück, so dass man sie auf die gleiche Art und Weise verketten kann.

Alle Extraktionsoperatoren überspringen Whitespaces vor dem Lesen. Beim Lesen von Zeichenketten werden Whitespaces übersprungen, dann Zeichen gelesen bis zum nächsten Whitespace. Grundsätzlich muss man am Extraktionsoperator für C-Zeichenketten kritisieren, dass die Länge der Zeichenkette nicht geprüft wird (nicht geprüft werden kann!), so dass beim Lesen jederzeit Speicher hinter dem physikalischen Ende der Zeichenkette überschrieben werden kann. Das kann man beispielsweise so verhindern:

```
char zk [80];
cin >> setw (sizeof (zk)) >> zk;
```

`setw` ist ein Manipulator, der in `iomanip` defininiert ist. Er begrenzt die maximale Anzahl von Zeichen, die in der folgenden Operation (hier: Extraktion der Zeichenkette) übertragen werden. Einfacher ist natürlich die Verwendung „richtiger“ Strings der C++ Standardbibliothek aus `<string>` (siehe Kapitel 8 auf Seite 51), die nicht überlaufen können.

7.4.2 Prüfen auf Dateiende

Datenströme können wie boolesche Werte geprüft werden. Ihr boolescher Wert ist TRUE, wenn der Status ok ist, und FALSE, wenn irgendein Fehler aufgetreten ist, beispielsweise wenn das Dateiende erreicht ist und versucht wurde, darüber hinaus zu lesen.

Man könnte also programmieren:

```
char c;

while (cin >> c) { // Einlesen mit Prüfen auf Fehler/Dateiende
  ... // Verarbeitung von c
}
```

Eine andere Variante unter Einsatz des booleschen Negationsoperators `!` sähe so aus:

```
char c;

for (;;) {
```

```
  cin >> c;           // Einlesen
  if (!cin) break;    // Wenn Fehler/Dateiende, Schleife verlassen
  ... // Verarbeitung von c
}
```

Da aber auf diese Weise das Dateiende von einem anderen Fehler nicht unterscheidbar ist, kann man auf End-Of-File noch auf eine zusätzliche Art prüfen, nämlich mit der Methode `eof()`. Also wäre es gut, hinter die obigen Schleifen noch zu schreiben:

```
if (!cin.eof()) {
  cerr << "Ausgabe wg. Fehler beendet - nicht wg. EOF" << endl;
}
```

7.4.3 Die Methode `get()`

Da die Extraktionsoperatoren immer Whitespaces überspringen, bevor sie etwas einlesen, braucht man eine weitere Möglichkeit, Zeichen und Zeichenketten einzulesen, ohne dass irgendetwas weggelassen wird. Zum Einlesen von Zeichen dient die Methode `get()`, die ein Zeichen liest.

Sie liefert `EOF` zurück, wenn das Dateiende erreicht ist, so dass man üblicherweise schreibt:

```
char c;
while ((c = cin.get()) != EOF) {
  ... // Verarbeitung von c
}
```

Es gibt noch eine zweite, overloadete Variante der Methode, die einen Parameter hat – eine Referenz auf die Variable, in die gelesen werden soll. Der Rückgabewert ist dann der Datenstrom, aus dem gelesen wird. Dessen boolescher Wert kann dann wieder geprüft werden:

```
char c;
while (cin.get(c)) { // Einlesen mit Prüfen auf Fehler/Dateiende
  ... // Verarbeitung von c
}
```

Das Gegenstück zu `get()` ist `put()`, siehe Abschnitt 7.3.3 auf Seite 43.

7.4.4 Die Methode `getline()`

Die Methode `getline()` liest eine ganze Eingabezeile, wie das von den Funktionen `gets()` und `fgets()` aus C (und `readln` aus Pascal) bekannt ist. Das ist der Prototyp:

```
istream& getline (char *, int, char = '\n');
```

`getline()` liest Zeichen in die C-Zeichenkette (den ersten Parameter). Der zweite Parameter begrenzt die Anzahl einzulesender Zeichen, während das Treffen auf den dritten Parameter die Eingabe stoppt (ebenso wie das Erreichen des Dateiendes). Dieser Aufruf von `getline()` liest bis maximal `len-1` Zeichen eingelesen wurden oder bis ein Zeilenendezeichen oder das Dateiende erreicht wurden: `cin.getline (zk, len);`

Dieser Aufruf dagegen stoppt statt am Zeilenende schon beim nächsten Leerzeichen: `cin.getline (zk, len, ' ');`

Die Methode schreibt immer ein Null-Terminierungszeichen an das Ende der C-Zeichenkette. Das Zeichen, das das Beenden des Einlesevorgangs verursacht hat, wird zwar aus dem Eingabestrom entfernt, aber nicht abgespeichert.

Nach einem Einlesevorgang mittels `getline()` kann man mit der Methode `gcount()` abfragen, wieviele Zeichen verarbeitet wurden:

```
cin.getline (zk, len);
cout << cin.gcount() << " Zeichen gelesen" << endl;
```

Hinweis: Bezüglich Einlesen von C++-Strings aus der Klasse `<string>` siehe Kapitel 8.2.6 auf Seite 54

7.4.5 Weitere Eingabe-Methoden (`read()`, `ignore()`, `peek()`, `putback()`)

Die Methode `read()` liest binär eine Folge von Bytes, die typischerweise von der Methode `write()` (siehe Kapitel 7.3.3 auf Seite 43) geschrieben wurden. Genauso wie bei `getline()` kann man anschließend mit `gcount()` (siehe Kapitel 7.4.4 auf der vorherigen Seite) fragen, wieviele Zeichen tatsächlich gelesen wurden.

```
istream& read (char *, int);
// Eingabe einer C-Zeichenkette oder einer beliebigen Bytefolge
```

Die Methode `peek()` lünkert in den Eingabestrom, welches Zeichen als nächstes zum Lesen bereitsteht, ohne es wirklich zu lesen. Prototyp: `int peek();`. Ähnlich wie bei der C-Funktion `getc()` wird ein `int` geliefert, obwohl es sich eigentlich um ein Zeichen handelt.

Die Methode `putback()` legt das zuletzt (üblicherweise mit `get()` gelesene) Zeichen wieder in den Datenstrom zurück. Das muss nicht mehrfach hintereinander funktionieren, sondern ist nur für ein einziges Zeichen garantiert. Prototyp: `istream& putback (char);`

Die Methode `ignore()` überliest Zeichen im Eingabestrom, d. h. setzt den Lesezeiger nach vorn, ohne Daten zu empfangen. Prototyp: `istream& ignore (int = 1, int = EOF);` Der erste Parameter gibt an, wieviele Zeichen zu überlesen sind, der zweite gibt an, hinter welchem Zeichen vorzeitig anzuhalten ist. Anwendungsbeispiele:

```
cin.ignore();        // 1 Zeichen überlesen
cin.ignore(10);      // 10 Zeichen überlesen
cin.ignore(10, ';'); // 10 Zeichen überlesen, aber falls
// unterwegs ein Semikolon vorkommt, dieses lesen und dann anhalten
```

7.5 Overloaden der Operatoren >> und <<

Die Anwender der Programmiersprache C++ können neue, overloadete Versionen der Extraktions- und Insertionsoperatoren für die von ihnen eingeführten Klassen definieren. Das sollte man für alle Klassen tun, deren Objekte eingelesen und ausgegeben werden können. Auf diese Weise werden sie auf dieselbe Weise ein- und ausgebbar wie die bereits in der Sprache selbst enthaltenen Datentypen.

Um einen Insertionsoperator für die Klasse `X` zu definieren, braucht man folgenden Prototypen:

```
friend ostream& operator<< (ostream&, const X&);
```

Ein Extraktionsoperator für dieselbe Klasse sieht so aus:

```
friend istream& operator>> (istream&, X&);
```

Als Beispiel werden wir die beiden für unsere Brüche-Klasse (siehe Kapitel 4.2.2 auf Seite 23) benötigten Operatoren definieren.

```
class bruch {
  friend ostream& operator<< (ostream&, const bruch&);
  friend istream& operator>> (istream&, bruch&);
}

ostream& operator<< (ostream& os, const bruch& b) {
  if (b.nenner == 1) {
    return os << b.zaehler;
  } else {
    return os << b.zaehler << "/" << b.nenner;
  }
```

```
} // operator <<

istream& operator>> (istream& is, bruch& b) {
  return is >> b.zaehler >> b.nenner;
} // operator >>
```

Nun können Brüche wie alle anderen Datentypen eingelesen und ausgegeben werden. Bei der Ausgabe wird nur eine Ganzzahl ausgegeben, wenn der Nenner 1 ist, sonst werden der Zähler, ein Schrägstrich und der Nenner ausgegeben. Bei der Eingabe hingegen werden immer zwei Zahlen ohne einen Schrägstrich dazwischen erwartet.

In dieser Version sind die beiden Operationen nicht kompatibel, d. h. vom Insertionsoperator ausgegebene Brüche können vom Extraktionsoperator nicht eingelesen werden. Es ist eine gute Übung, diese beiden kompatibel zu machen.

7.6 Datei-Ein- und -Ausgabe

Für Datei-Ein- und -Ausgabe braucht man die Headerdatei `<fstream>`, in der die folgenden Klassen definiert sind:

- `ifstream` – Eingabedateien
- `ofstream` – Ausgabedateien
- `fstream` – Ein-/Ausgabedateien

Das Schreiben in und das Lesen aus Dateien wird prinzipiell genauso gehandhabt wie das Arbeiten mit den Standard-Datenströmen für die Konsole. Es kann jede Operation, die auf `cout` gemacht wird, auch auf ein beliebiges Objekt der Klassen `ofstream` oder `fstream` erfolgen. Das gilt selbstverständlich auch für selbst definierte, overloadete Operatoren, d. h. wir können unsere Brüche auch direkt in Dateien ausgeben und aus diesen einlesen!

7.6.1 Öffnen und Schließen von Dateien

Dateien können mit den Konstruktoren der Klassen geöffnet werden, oder aber man verwendet die Methode `open()`:

```
ofstream ausgabeDatei ("ausgabe.txt"); // deklarieren und öffnen
ifstream eingabeDatei ("eingabe.txt"); // deklarieren und öffnen

ofstream ausgabe2; // nur deklarieren
ifstream eingabe2; // nur deklarieren

ausgabe2.open ("ausgabe2.txt"); // öffnen
eingabe2.open ("eingabe2.txt"); // öffnen
```

Um zu prüfen, ob das Öffnen erfolgreich war, prüft man den booleschen Wert des Datenstrom-Objekts:

```
if (!ausgabeDatei) {
  cerr << "Konnte Datei nicht öffnen" << endl;
  exit (EXIT_FAILURE);
}
```

Um eine Datei zu schließen, ruft man die Methode `close()` auf. Eine Datei wird automatisch geschlossen, wenn das Objekt seinen Gültigkeitsbereich verlässt, d. h. der Destruktor sorgt für das ordnungsgemäße Schließen der Datei.

```
ausgabeDatei.close();
```

7.6.2 Datei-Modi

Die Konstruktoren von Datei-Strömen können einen weiteren Parameter haben, der den Modus der Datei angibt. Er wird aus einzelnen Bits zusammengesetzt, die folgende Werte haben können:

- `ios::in` – Datei soll für Eingabe geöffnet werden, default bei `ifstream`.
- `ios::out` – Datei soll für Ausgabe geöffnet werden, default bei `ofstream`. Wenn sie bereits besteht, wird ihr Inhalt gelöscht, sofern nicht gleichzeitig `ios::ate` oder `ios:app` angegeben wird.
- `ios::ate` – Öffnen mit Positionierung am Dateiende, sinnvoll zu kombinieren mit `ios::out`
- `ios::app` – Öffnen zum Anhängen, d. h. vor jedem Schreibvorgang wird automatisch ans Dateiende positioniert.
- `ios::trunc` – Inhalt beim Öffnen löschen, passiert beim Öffnen zum Schreiben sowieso
- `ios::nocreate` – Öffnen soll schiefgehen, wenn die Datei nicht bereits existiert.
- `ios::noreplace` – Öffnen soll schiefgehen, wenn die Datei bereits existiert; es sei denn, dass gleichzeitig `ios::ate` oder `ios:app` angegeben wird.
- `ios::binary` – Öffnen im Binärmodus statt im Textmodus, sinnvoll zu kombinieren mit `ios::in` und/oder `ios::out`

Einer oder mehrere diese Modi können beim Öffnen mittels des bitweisen ODER-Operators verknüpft angegeben werden:

```
ofstream aus ("aus.txt", ios::out | ios::noreplace);
```

Wenn ein Modus für eine Datei der Klasse `ofstream` angegeben wird, muss `ios::out` ebenfalls angegeben werden. Die Angabe `ios::out` ist zwar default für den Parameter, aber sie ist nur dann entbehrlich, wenn der Parameter gänzlich weggelassen wird.

Entsprechend muss man den Modus `ios::in` bei Dateien der Klasse `ifstream` mit angeben, sofern man einen Modus mitgibt. Bei Dateien der Klasse `fstream` dagegen muss sowieso mindestens einer der Modi angegeben werden, weil sie keinen Default-Modus haben.

7.6.3 Beispielprogramm: Kopieren einer Textdatei

Das folgende kleine Programm kopiert eine Textdatei zeichenweise in die andere. Die Dateinamen werden als Parameter beim Programmaufruf angegeben:

```
#include <fstream>
#include <stdlib.h>

int main (int argc, char *argv[]) {
  if (argc != 3) {
    cerr << "usage: " << argv [0] << " fromfile tofile" << endl;
    exit (EXIT_FAILURE);
  }

  ifstream source (argv[1]);
  if (!source) {
    cerr << "can't open " << argv [1] << endl;
    exit (EXIT_FAILURE);
  }
```

```
  ofstream dest (argv [2]);
  if (!dest) {
    cerr << "can't create " << argv [2] << endl;
    exit (EXIT_FAILURE);
  }

  char ch;
  while (dest && source.get(ch)) dest.put (ch);

  return (EXIT_SUCCESS);
} // main()
```

7.6.4 Wahlfreier Zugriff

Unter wahlfreiem Zugriff versteht man das Zugreifen (und Lesen bzw. Schreiben) auf beliebige Teile einer Datei. Hier wird also der Dateiinhalt nicht unbedingt sequentiell verarbeitet, sondern man springt auf eine Position in der Datei und schreibt oder liest dort. Wenn eine Datei eine feste Struktur hat, d. h. aus gleich langen Sätzen besteht, kann man leicht ausrechnen, an welcher Byteposition der Datei ein bestimmter Satz beginnt und diesen lesen bzw. (über-)schreiben.

Hierzu gibt es die Methoden `seekg()`, `seekp()`, `tellg()`, `tellp()`. Positionen innerhalb einer Datei werden angegeben durch Werte des Datentyps `streampos` (stream position, intern oft ein `long int`), Abstände zu einer anderen Position durch Werte des Datentyps `streamoff` (stream offset, intern ebenfalls oft ein `long int`).

Die Methode `seekg()` ist auf Eingabe- und Ein-/Ausgabe-Datenströme anwendbar und positioniert den Schreib-/Lesezeiger des Datenstroms auf eine bestimmte Position – entweder absolut in der Variante mit einem Parameter oder in Bezug auf eine andere Position, die als zweiter Parameter angegeben wird und einen der Werte `ios::beg` (relativ zum Dateibegin, also wie die Variante mit einem Parameter), `ios::cur` (relativ zur aktuellen Position – current = aktuell) oder `ios::end` (relativ zum Dateiende) enthalten muss.

```
istream& seekg (streampos);
istream& seekg (streamoff, seek_dir);
```

Die Methode `tellg()` dient zur Abfrage der aktuellen Position: `streampos tellg()`.

Die entsprechenden Methoden, deren Name nicht auf `g`, sondern auf `p` enden, sind auf Ausgabe- und Ein-/Ausgabeströme anwendbar und haben gleiche Funktion.

7.7 Formatierung in Zeichenketten

7.7.1 `strstream` für C++ im AT&T-Standard

Die Headerdatei `strstream.h` enthält die notwendigen Methoden zur formatierten Ausgabe in Zeichenketten, d. h. das, was man in C mit `sprintf()` macht. Man kann damit binäre Daten formatiert in Zeichenketten schreiben oder auch aus ihnen lesen.

Es gibt die Klassen `istrstream` für das Lesen aus Zeichenketten und `ostrstream` für das Schreiben in Zeichenketten. Um die Ein-/Ausgabe durchzuführen, erzeugt man ein entsprechendes Objekt und wendet den Insertions- bzw. den Extraktionsoperator darauf an.

```
char zk[] = "12 34 56";
int i;

istrstream (zk) >> i;                // i wird 12
ostrstream (zk, sizeof (zk)) << 78;  // zk wird "78 34 56",
   // die ersten beiden Zeichen "12" wurden überschrieben
```

Der `ostrstream`-Konstruktor verlangt die Angabe einer Größe, damit nicht über das Ende der Zeichenkette hinaus geschrieben wird. Ein Versuch, die Länge der Zeichenkette zu überschreiten, führt zu einem Fehlerstatus des Streams.

Wenn Daten in den Strom hineingeschrieben werden, wird kein Begrenzungszeichen ASCII-Null hinzugefügt. Wenn man eines anhängen möchte, muss man das angeben:

```
ostrstream (zk, sizeof (zk)) << 78 << ends   // zk wird "78"
```

7.7.2 `stringstream` für C++ im ANSI/ISO-Standard

Die Formatierung in Zeichenketten hat sich mit der Standardisierung von C++ in ANSI/ISO wesentlich geändert. Auch haben sich die Namen der Headerdateien und der Klassen geändert.

Voraussetzung für die „neuen“ String-Streams ist das Vorhandensein von C++-Strings, d. h. der Klasse `string`, definiert in der Headerdatei `<string>` (Achtung, nicht verwechseln mit der C-Headerdatei `<string.h>`, statt derer man sowieso `<cstring>` verwenden sollte! Siehe Kapitel 8 auf der nächsten Seite).

Die hier beschriebenen String-Streams benötigen keine C-Zeichenkette mehr als Grundlage und können auch nicht mehr überlaufen (es sei denn, es kann kein Hauptspeicher mehr allokiert werden). Ein Beispiel für eine Ausgabe-String-Stream:

```
#include <sstream>
#include <string>

ostringstream ost;  // Anlegen des String-Streams

// Hineinschreiben wie in einen Datei-Strom
ost << "Differenz beträgt " << x - y << ".";

string s; // Anlegen eines Strings
s = ost.str(); // Zuweisen des Stream-Inhalts in den String
```

Einen Input-String-Stream kann man beispielsweise dazu verwenden, die Wörter eines Strings untereinander auszugeben:

```
#include <sstream>
#include <string>

string meinText ("enthält mehrere Wörter nebeneinander");

istringstream ist (meinText);
// Anlegen des String-Streams, initalisiert mit dem String meinText

string w; // für je ein Wort

while (ist >> w) cout << w << endl;
// wortweise herauslesen und als Zeilen ausgeben
```

Kapitel 8

Die Standardklasse `string`

In C gibt es gar keinen Datentyp für Zeichenketten. Stattdessen werden Arrays aus `char` dafür verwendet. Es gibt eine ganze Reihe von Funktionen zur Stringbehandlung (in `<string.h>`, in C++ jetzt <cstring> genannt) und für die Ein- und Ausgabe, so dass man fast den Eindruck gewinnt, es gäbe in C Zeichenketten. Sobald ein Programm aber einen kleinen Fehler enthält oder auch nur die Standardfunktion `gets()` verwendet, ist es dem baldigen Absturz geweiht, weil keinerlei Kontrollmechanismen vorhanden sind, die das Überschreiben von Speicherplatz jenseits der Grenzen des Arrays verhindern. Ebenso muss man auch immer daran denken, allokierten Speicherplatz manuell wieder freizugeben.

Hier bot sich also ein großes Betätigungsfeld für die Designer der C++-Standardbibliothek. Sie erfanden die Standardklasse `string`, für die die Headerdatei `<string>` includet werden muss (im Gegensatz zur Headerdatei `<string.h>`, die die C-Stringfunktionen beschreibt). Diese Klasse gibt es erst in neueren Versionen von C++, insbesondere in denen, die dem ANSI/ISO-Standard entsprechen, nicht aber in älteren AT&T-Versionen von C++.

Die Strings in C++ sind einfach zu handhaben und robust. Es gibt viele handliche Operatoren:

`<<`	`>>`	Ein- und Ausgabe
+	+=	Verkettung/Anhängen
`==`	`!=`	Prüfung auf Gleichheit/Ungleichheit
`<=`	`<`	alphabetischer Vergleich auf kleiner bzw. kleiner/gleich
`>=`	`>`	alphabetischer Vergleich auf größer bzw. größer/gleich

Das Schöne an diesen Operatoren ist zudem, dass die Operanden nicht nur Strings, sondern auch einzelne Zeichen sein dürfen.

Die Strings existieren nicht nur basierend auf dem Datentyp `char`, sondern auch noch einmal basierend auf dem Datentyp `wchar_t`, den es ja schon in C gab, wenn er auch wenig verwendet wurde. Dieser Datentyp dient der Speicherung eines 16-bit-Zeichen, d. h. eines Zeichens im Unicode-Zeichensatz, der fast alle Zeichen aller Sprachen darstellen kann (u. a. neben allen europäischen Zeichen auch Arabisch, Japanisch, Chinesisch, Koreanisch usw.). Übrigens arbeiten heute bereits viele Systeme ausschließlich intern mit Unicode, so z. B. Tk ab Version 8.2 (auf allen Plattformen) und Windows NT ab Version 4.0. Strings basierend auf `wchar_t` gehören zur Klasse `wstring`, während die basierend auf `char` zur Klasse `string` gehören.

Da es sich nur um verschiedene Instantiierungen derselben Basisklasse (`basic_string`) handelt, sind alle Member identisch, weshalb hier im Folgenden immer nur die Klasse `string` erwähnt wird.

8.1 Erzeugen von String-Objekten

Beim Anlegen von String-Objekten braucht man keine maximale Länge anzugeben, weil Strings automatisch mitwachsen, wenn sich ihr Inhalt ändert. Das kostet zwar einiges an Verwaltungsaufwand, macht die Handhabung aber speicherplatzoptimal und sicher. Beim wahlweise anzuwendenden Initialisieren können

C-Zeichenketten und andere Strings verwendet werden, wobei sogar das Ausschneiden von Teilzeichenketten geht, ohne dass man dazu eine extra Methode bräuchte.

```
#include <string>
using namespace std; // siehe Abschnitt über Namensbereiche

string zk1;  // default-Konstruktor, leere Zeichenkette
string zk2 ("Beispiel"); // Konstruktor aus char *
string zk3 (zk2); // Kopierkonstruktor

char c_zk[] = "Dies ist eine C-Zeichenkette";
string zk4 (c_zk, 10); // Initialisieren mit dem Beginn
       // einer C-Zeichenkette, also "Dies ist e"

zk4 = c_zk;
string zk5 (zk4, 5, 3); // Initialisieren mit einem
       // Teil eines anderen Strings, d. h. ab dem
       // Zeichen mit dem Index 5 werden 3 Zeichen
       // kopiert, also "ist"

string zk6 (zk4, 5); // Initialisieren mit einem Teil
       // eines anderen Strings, d. h. ab dem Zeichen
       // mit dem Index 5 werden alle Zeichen kopiert,
       // also "ist eine C-Zeichenkette"

string zk7 (75, '-'); // mit 75 Strichen initialisieren
```

Die Verwendung von `using namespace std;` ist nicht immer notwendig, der aktuelle GNU-Compiler verlangt es nicht. Näheres zu Namensbereichen siehe Kapitel 13 auf Seite 99.

8.2 Arbeiten mit String-Objekten

8.2.1 Zuweisung von String-Objekten

Strings dürfen auch problemlos zugewiesen werden, so dass das lästige Aufrufen von `strcpy` wegfällt. Um nachträglich, also nicht beim Erzeugen, eine Teilzeichenkette zuzuweisen, gibt es die Methode `assign()`:

```
string zk1 ("Dies ist eine lange Zeichenkette");
string zk2;
zk2.assign (zk1, 14, 4); // zugewiesen wird "lang"
```

8.2.2 Längenbestimmung von String-Objekten

Um die Länge eines Strings zu bestimmen, ist keine aufwendige Funktion wie `strlen()` mehr nötig, die die ganze Zeichenkette durchiterieren muss, bis sie auf das Stringende-Zeichen stößt, sondern es gibt eine Methode `length()`, die die in einem privaten Datenmember gespeicherte Länge zurückliefert. Diese ist vom Datentyp `string::size_type`. Man kann sie beispielsweise beim Aufruf von `write()` (siehe Kapitel 7.3.3 auf Seite 43) verwenden:

```
ofstream aus ("ausgabe");
string zk ("Beispielstring");
aus.write (zk.c_str(), zk.length());
```

Hier wird direkt noch eine weitere Methode vorgestellt, nämlich `c_str()`, um aus einem String eine C-Zeichenkette zu machen. Es gibt nämlich keine automatische Konversion von einem String zu einer C-Zeichenkette, auch wenn es in der anderen Richtung über den oben gezeigten Konstruktor durchaus funktioniert.

Den gelieferten `char *` Zeiger darf man allerdings nicht speichern, weil sich die Adresse, an dem die Zeichenkette intern gespeichert ist, bei Änderungen des Stringinhaltes jederzeit ändern kann. Also bitte nur zum sofortigen Gebrauch verwenden und notfalls mit dem guten alten `strcpy()` kopieren.

8.2.3 Zugriff auf einzelne Zeichen von String-Objekten

Um auf einzelne Zeichen zugreifen zu können, kann man dieselbe Schreibweise wie bei den klassischen C-Zeichenketten verwenden, weil der `[]`-Operator für diesen Zweck overloadet wurde. Tatsächlich wurde er sogar zweimal overloadet, nämlich einmal für das Lesen eines Zeichens im String und einmal für das Schreiben eines solchen, wozu er eine Referenz auf das Zeichen zurückliefern muss:

```
char& operator[] (string::size_type position);
char  operator[] (string::size_type position) const;
```

Da die zweite Variante den String garantiert unverändert lässt, ist die Operatormethode `const`, so dass man sie auf `const` Objekte anwenden kann.

Im wesentlichen gleichbedeutend (auch zwei Mal implementiert) ist die Methode `at()`, die im Unterschied zum obigen Operator eine Bereichsprüfung enthält, also ggf. eine Ausnahme erzeugt (siehe Kapitel `Ausnahmen`).

8.2.4 Einfügen und Löschen von Zeichen in String-Objekten

Die Methode `insert()` erlaubt das Einfügen von Zeichen und Zeichenketten an beliebiger Stelle im String. Ein Überlaufen kann nicht passieren, weil der String nötigenfalls vergrößert wird. Um ans Ende anzuhängen, verwendet man einfacher den Operator `+=`.

Löschen von Zeichen geht mit der Methode `erase()`, die die Zeichen herauslöscht, aber die physikalische Größe des String-Objekts unverändert lässt.

```
string zk1 ("Hallo");
zk1 += " duda?";
zk1.insert (5, ", was machst denn");
zk1.insert (25, " ");
cout << zk1 << endl;
// gibt aus: "Hallo, was machst denn du da?"

zk1.erase (15, 10);
zk1.insert (15, "en denn Sie");
cout << zk1 << endl;
// gibt aus: "Hallo, was machen denn Sie da?"

zk1.erase (5); // löscht ab Zeichen 5
cout << zk1 << endl;
// gibt aus: "Hallo"

zk1.erase(); // löscht gesamte Zeichenkette
cout << zk1 << endl;
// gibt nur einen Zeilenvorschub aus
```

8.2.5 Suchen und Ersetzen in String-Objekten

Zum Suchen gibt es die Methoden `find()` zum Suchen vom Beginn der Zeichenkette und `rfind()`, die von rechts sucht. Sie geben die Position der gesuchten Teilzeichenkette zurück oder aber den Wert `string::npos`, falls sie nicht fündig werden.

Zum Ersetzen gibt es die Methode `replace()`. Hier werden die Startposition, ab der ersetzt werden soll, die Anzahl Zeichen, die ersetzt werden soll, und die Zeichenkette, die stattdessen eingesetzt werden soll, angegeben. Das bedeutet, dass die Methode `replace` keine Suchfunktion beinhaltet.

```
string zk1 = "Diese Zeichenkette ist das Original - noch!";
zk1.replace (19, 3, "war");
zk1.insert (38, "eben ");

string zk2 = "Es werden alle 'e' durch 'i' ersetzt.";
string::size_type pos=0;
while ((pos = zk2.find ("e", pos)) != string::npos) {
  zk2.replace (pos, 1, "i");
}
```

8.2.6 Einlesen von String-Objekten

Strings können mit dem üblichen `<<`-Operator eingelesen werden, allerdings werden dann zunächst Whitespaces überprungen und es wird auch nur bis zum darauf folgenden Whitespace gelesen. Möchte man mehr als nur ein Wort – üblicherweise eine Zeile – einlesen, so verwendet man die Funktion (nicht die Methode!) `getline()`.

```
istream& getline (istream&, string&, char='\n');
```

Es wird eine Zeichenfolge vom Eingabestrom in den String eingelesen, bis das im dritten Parameter angegebene Zeichen angetroffen wird (default: Newline). Rückgabe ist eine Referenz auf den verwendeten Eingabestrom, so dass man zum Einlesen eines Strings und einer Zahl schreiben kann:

```
// <string> muss includet sein!
string s;
int i;
getline (cin, s) >> i;
```

Hinweis: Die Anzahl der von `getline()` gelesenen Zeichen kann man nicht mittels `gcount()` abfragen.

8.2.7 Platzverwaltung von String-Objekten

Damit ein String nicht ständig reallokiert werden muss, weil er zwischendurch öfters mal wächst, kann man für ihn direkt eine bestimmte Größe reservieren. Das geht mit der Methode `reserve (string::size_type)`.

Kapitel 9

Vererbung

Um die Wiederverwendbarkeit von Code zu verbessern, gibt es in C++ das Konzept der Vererbung. Dabei wird eine neue Klasse erzeugt, die Eigenschaften einer bereits bestehenden Klasse erbt. Meist handelt es sich bei der neuen Klasse um eine Klasse, die mehr Eigenschaften hat als die alte Klasse, sozusagen eine Art Spezialisierung. Die neue Klasse heißt abgeleitete Klasse, die alte Klasse heißt Basisklasse. Von einer Basisklasse kann es durchaus mehrere abgeleitete Klassen geben.

Den Begriff der Klassifizierung gibt es schon viel länger als es Computer gibt. Biologen beispielsweise klassifizieren Tiere. „Tiere“ wäre z. B. eine prima Basisklasse. Von dieser recht allgemeinen Basisklasse kann man spezialisiertere Klassen ableiten, z. B. „Säugetiere“, „Insekten“, „Amphibien“, „Vögel“. Die Spezialisierung kann auch weitergehen, d. h. man kann die Säugetiere weiter unterteilen in einzelne Tierarten wie „Hunde“, „Katzen“ usw.

Ein anderes Beispiel wären geometrische Formen. „Kreis“, „Dreieck“ und „Quadrat“ könnten von der Klasse „Form“ als allgemeinerer Klasse abgeleitet sein. Auf diese Weise müssten allgemeine Attribute wie `Farbe` bereits bei der Klasse „Form“ definiert sein, und spezielle Attribute wie `Mittelpunkt` und `Radius` bei der Klasse „Kreis“ definiert werden.

So etwas nennt man dann eine Klassenhierarchie. Im Folgenden sollen nun die Mechanismen der Klassenableitung beschrieben werden. Es wird gezeigt, wie man Klassenbibliotheken erstellen kann. Die Unterstützung der Klassenhierarchien, gemeinsam mit der Bereitstellung „virtueller Methoden“, macht C++ zu einer ojektorientierten Sprache.

Bezüglich des Entwurfs von Klassenhierarchien und der Theorie dahinter sei auf Literatur zum Thema Software Engineering verwiesen.

9.1 So hilft die Vererbung beim Programmieren

Falls Sie beim Schreiben eines C++-Programms feststellen, dass Sie eine Klasse benötigen, gibt es drei Möglichkeiten, wie Sie vorgehen können:

- In der Klassenbibliothek gibt es eine passende Klasse. Verwenden Sie diese – fertig!
- In der Klassenbibliothek gibt es eine ähnliche, allgemeinere Klasse. Leiten Sie eine neue Klasse von dieser ab – fertig!
- Es gibt gar nichts Passendes in der Bibliothek. In diesem Fall müssen Sie eine ganz neue Klasse schaffen – wie das in den bisherigen Beispielen immer der Fall war.

9.2 Unsere Beispiel-Klassenhierarchie „Fahrzeug“

Es gibt eine Basisklasse `Fahrzeug`, von der zwei Klassen abgeleitet sind: `Auto` und `Fahrrad`. Die Gemeinsamkeiten, `Hersteller` und `Baujahr`, sind in der Basisklasse abgelegt. Die Angaben, die ausschließlich den spezialisierten Objekte zugeordnet werden können, stehen in den abgeleiteten Klassen.

Es handelt sich um eine *echte Vererbung*, denn schließlich haben sowohl Fahrräder als auch Autos alle Eigenschaften eines Fahrzeugs, denn beide sind letztendlich Fahrzeuge.

```
class Fahrzeug {
  public:
    Fahrzeug();
    Fahrzeug (char *Herst, int Bj=0);
    Fahrzeug (int Bj);
    ~Fahrzeug();
    void Ausgabe();
  private:
    char *Hersteller;
    int Baujahr;
};

class Auto: public Fahrzeug {
  public:
    Auto();
    Auto (int Hubraum, int Leistung);
    Auto (char *Hersteller);
    ~Auto();
  private:
    int ccmHubraum;
    int kWLeistung;
};

class Fahrrad: public Fahrzeug {
  public:
    Fahrrad() {cmRahmenhoehe = 0; AnzahlGaenge = 0;}
    Fahrrad (int Rahmenhoehe, int AnzGaenge);
    Fahrrad (char *H, int Bj, int Rh, int G) ;
    ~Fahrrad() {};
  private:
    int cmRahmenhoehe;
    int AnzahlGaenge;
};
```

9.3 Einfachvererbung

Bei der Einfachvererbung erbt eine neue Klasse Eigenschaften von einer bereits bestehenden. Das muss bei der Deklaration der Klasse angedeutet werden:

```
class AbgeleiteteKlasse: public Basisklasse {
 ...
};
```

Die `AbgeleiteteKlasse` *erbt* alle Member der `BasisKlasse` – mit Ausnahme der Konstruktoren und Destruktoren. Also: Alle Member der `BasisKlasse` sind auch Member von `AbgeleiteteKlasse`. Sinnvollerweise werden in der neuen Klasse weitere Member hinzudefiniert.

Obwohl alle Member geerbt werden, können Methoden aus der neuen Klasse nicht unbedingt auf sie zugreifen. Lediglich auf `public` Member der Basisklasse besteht uneingeschränkter Zugriff. Auf `private` Member der Basisklasse haben die Methoden aus der abgeleiteten Klasse keinen Zugriff, weil sie nur den Methoden und Freunden der Basisklasse zur Verfügung stehen.

Tatsächlich gibt es noch eine dritte Einstellung für die Freigabe: `protected` – so deklarierte Member der Basisklasse stehen auch abgeleiteten Klassen offen.

```
class Basisklasse {
  public:
     ...     // verfügbar in AbgeleiteteKlasse und sonstwo
  protected:
     ...     // verfügbar in AbgeleiteteKlasse
  private:
     ...     // nicht verfügbar in AbgeleiteteKlasse
};
```

9.3.1 Eigenschaften abgeleiteter Klassen

Objekte abgeleiteter Klassen können Variablen der Basisklasse zugewiesen werden, aber nicht umgekehrt. In der Zuweisung von `A` auf `B` werden die gemeinsamen Datenmember kopiert, die übrigen gehen verloren.

```
Basisklasse         B;
AbgeleiteteKlasse   A;

B = A;   // legal!
A = B;   // illegal!
```

Eine Variable vom Typ `Basisklasse*` oder `Basisklasse&` kann auf ein Objekt einer abgleiteten Klasse zeigen. Dabei kann die abgeleitete Klasse auch indirekt von der Basisklasse abgeleitet sein (d. h. kann auch ein Enkel oder Urenkel sein). Es bleibt das gesamte Objekt verfügbar, so dass man insbesondere Funktionen bzw. Methoden schreiben kann, die eine Referenz (oder einen Zeiger) auf ein Objekt der Basisklasse als Argument übernehmen, aber auch mit Objekten aus abgeleiteten Klassen funktionieren. Bei Werteparametern ginge bei der Übergabe alles verloren, was nicht in der Basisklasse vorkommt.

```
AbgeleiteteKlasse   A;
Basisklasse         *zeiger = &A;
Basisklasse         &ref    = A;
```

Ein formaler Parameter vom Typ `Basisklasse*` passt auch auf einen aktuellen Parameter, wenn dieser von der `Basisklasse` abgeleitet ist. Entsprechendes gilt auch für Referenzen vom Typ `Basisklasse&`.

Methoden der Basisklasse können in den abgeleiteten Klassen weiterbenutzt werden. Allerdings können sie auch ohne weiteres neu definiert werden, was man *Overriding* nennt – im Gegensatz zum Overloading, bei dem sich die Parameterliste unterscheidet. Technisch beruht beides auf einer Unterscheidung anhand der Signatur, die aus der Parameterliste bestimmt wird. Bei Overriding unterscheidet sich aber nicht die sichtbare Parameterliste, sondern nur der implizite Parameter in Gestalt des Objekts, dessen Methode aufgerufen wird.

Initialisierung von Objekten abgeleiteter Klassen Beim Anlegen eines Objekts einer abgeleiteten Klasse wird zunächst der Konstruktor der Basisklasse aufgerufen, um dessen Datenmember zu initialisieren. Erst danach wird der Konstruktor der abgeleiteten Klasse aufgerufen, so dass er sich um seine eigenen (zusätzlichen) Datenmember kümmern kann. Falls es in der Basisklasse nicht nur einen einzigen, parameterlosen Konstruktor gibt, kann man in einer Element-Initialisierungsliste den Konstruktor der Basisklasse auswählen (siehe auch Kapitel 5.1.2 auf Seite 33).

Analog werden beim Zerstören eines Objekts die Destruktoren der am weitesten abgeleiteten Klasse als erstes aufgerufen, dann die der nächsten Basisklasse und immer so weiter bis zum Urahn in der Klassenhierarchie.

Konstruktoren für die Klasse `Auto`

```
Auto::Auto() : Fahrzeug ("Auto") {
  ccmHubraum  = 1500;
  kWLeistung  = 50;
}
```

Dies ist zwar der default-Konstruktor für die Klasse `Auto`, aber es wird nicht der default-Konstruktor der Klasse `Fahrzeug` verwendet. Da `Hersteller` ein privates Element der Klasse `Fahrzeug` ist, kann man es nur im Rahmen einer Elementinitialisierungsliste über den Konstruktor der Basisklasse initialisieren. Es wird hier, da kein Hersteller bekannt ist, die Zeichenkette „Auto“ eingetragen.

Ein anderer Konstruktor für `Auto` kann so aussehen:

```
Auto::Auto (char *Hersteller) : Fahrzeug (Hersteller) {
  ccmHubraum  = 1500;
  kWLeistung  = 50;
}
```

Hier wird ein Argument des Konstruktors von `Auto` direkt an die Element-Initialisierungsliste für die Basisklasse `Fahrzeug` weitergegeben. Jetzt könnte man noch eine ganze Reihe von Konstruktoren für `Auto` angeben, je nachdem, welche Angaben man mitgeben möchte. Hier ist der Einsatz von default-Werten gefragt!

Konstruktoren für die Klasse `Fahrrad`

```
Fahrrad::Fahrrad (char *H, int Bj, int Rh, int G) :
        Fahrzeug (H, Bj) {
  cmRahmenhoehe = Rh;
  AnzahlGaenge  = G;
}
```

Auch hier verwenden wir wieder einen Konstruktor der Basisklasse `Fahrzeuge` in der Element-Initialisierungsliste.

9.3.2 Ein-/Ausgabeoperatoren für abgeleitete Klassen

Für die abgeleitete Klasse sollen meist die Ein-/Ausgabeoperatoren overloadet werden, die für die Basisklasse bereits bestehen (siehe Abschnitt 7.4.5 auf Seite 46). Hierzu werden sie ebenfalls als `friend` deklariert. Da man auf die Member der Basisklasse nicht zugreifen kann, muss innerhalb der Operatoren der abgeleiteten Klasse jeweils der Operator der Basisklasse verwendet werden. Hierzu wird auf das auszugebende Objekt `f` eine Referenz der Basisklasse `fz` erzeugt.

```
class Fahrrad: public Fahrzeug {
friend ostream& operator<< (ostream &, const Fahrrad &);
friend istream& operator>> (istream &, Fahrrad &);
...
};

ostream& operator<< (ostream &os, const Fahrrad &f) {
  const Fahrzeug &fz = f; // Referenz der Basisklasse
  os << fz  << " "
     << f.cmRahmenhoehe << " "
     << f.AnzahlGaenge;
  return os;
}

istream& operator>> (istream &is, Fahrrad &f) {
  Fahrzeug &fz = f; // Referenz der Basisklasse
  is >> fz >> f.cmRahmenhoehe >> f.AnzahlGaenge;
}
```

Für den Ausgabeoperator täte es zwar auch ein normaler Cast, aber dabei wird der Kopierkonstruktor der Basisklasse bemüht, denn es wird eine Kopie erzeugt. Das ist erstens nicht laufzeitoptimal und zweitens für die Eingabe nicht verwendbar, weil die Eingabe auf eine Kopie nicht die gewünschte Wirkung hat.

9.3.3 Private Ableitung

Bei der Ableitung kann die Basisklasse auch `private` deklariert werden:

```
class privatAbgeleiteteKlasse: private Basisklasse {
 ...
};
```

Die Wirkung ist, dass Objekte der privat abgeleiteten Klasse nicht Variablen der Basisklasse zugewiesen werden können. Gleichzeitig werden alle aus der Basisklasse geerbten Member automatisch zu privaten Membern der abgeleiteten Klasse – was für Datenmember und auch für Methoden gilt.

```
Basisklasse               B;
privatAbgeleiteteKlasse   A;

B = A;   // illegal!
A = B;   // illegal!
```

Nun, wann setzt man die private Vererbung ein? Es gibt zwei grundsätzlich verschiedene Formen der Klassenableitung:

- *echte Vererbung:* Die abgeleitete Klasse ist eine höher spezialisierte Variante der Basisklasse. Es sollte die `public` Ableitung benutzt werden, da die abgeleitete Klasse das gesamte Verhalten der Basisklasse zeigen soll. Die Beziehung zwischen den Klassen wird oft als **ist**-Beziehung bezeichnet, denn in unserer Beispielklassenhierarchie **ist** ein Auto eine Art Fahrzeug.
- *Schichtung:* Die Basisklasse hilft lediglich bei der Realisierung der abgeleiteten Klasse. Es sollte die `private` Ableitung benutzt werden, da die abgeleitete Klasse nicht alle Eigenschaften der Basisklasse haben soll. Tatsächlich mag die abgeleitete Klasse wenig mit der Basisklasse gemeinsam haben, was das nach außen gezeigte Verhalten angeht. Die Beziehung zwischen den Klassen wird oft als **hat**-Beziehung bezeichnet, weil die Objekte quasi ein Objekt der Basisklasse enthalten, ohne eines zu sein. Dieser Fall ist sehr selten und wird im folgenden nicht weiter besprochen. Im allgemeinen kann man dies wesentlich einfacher durch Einbauen eines Objektes der anderen Klasse als Datenmember lösen.

9.3.4 Virtuelle Methoden

C++ bietet einen besonderen Mechanismus an – virtuelle Methoden –, um Methoden in der Basisklasse und in abgeleiteten Klassen anders zu definieren und diese auch beim Zugriff über Zeiger und Referenzen auf Objekte richtig zuordnen zu können. Bei statischen Objekten ist die Zuordnung ja ohnehin kein Problem (siehe Kapitel 9.3.1 auf Seite 57), weil ihr Typ zum Compilationszeitpunkt unveränderlich feststeht. Zeiger und Referenzen dagegen können mal auf ein Objekt der Basisklasse und mal auf ein Objekt einer daraus abgeleiteten Klasse zeigen. Die Entscheidung, welche Methode aufgrund des Objekts, auf das verwiesen wird, angwendet werden muss, kann erst zur Laufzeit fallen.

Deklaration virtueller Methoden Es ist notwendig, die Methode bereits in der Basisklasse als virtuell zu deklarieren. In der abgeleiteten Klasse kann die Methode als virtuell deklariert werden oder auch nicht, ohne dass dies eine Auswirkung hätte. Selbstverständlich müssen alle Versionen der Methode denselben Rückgabetyp und dieselbe Parameterliste haben, ansonsten hätten wir es ja auch mit Overloading zu tun (siehe Kapitel 3.1 auf Seite 12).

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

class AbgeleiteteKlasse: public Basisklasse {
  public:
    void f(); // virtual void f(); ginge auch!
}
```

Hinweis: Virtuelle Methode müssen immer in der Klasse, in der sie deklariert werden, auch definiert werden. Auch dann, wenn man sie (noch) gar nicht aufruft.

Aufruf von virtuellen Methoden

Virtuelle Funktionen müssen über Zeiger oder Referenzen aufgerufen werden. Ein Zeiger oder eine Referenz vom Typ `Basisklasse` kann auf ein Objekt dieser Klasse oder eines aus `AbgeleiteteKlasse` verweisen. Wenn der Zeiger oder die Referenz verwendet wird, um eine virtuelle Funktion aufzurufen, wird *automatisch* die richtige Version der Methode verwendet, obwohl der Zeiger bzw. die Referenz an sich zur `Basisklasse` gehört.

```
Basisklasse b, *p;
AbgeleiteteKlasse a;

p = &b;
p->f();      // ruft Basisklasse::f()

p = &a;
p->f();      // ruft AbgeleiteteKlasse::f()
```

Dynamische Bindung

Die dynamische Bindung wird auch „späte Bindung“ genannt, weil sie zur Laufzeit und nicht zur Compilationszeit stattfindet wie die „frühe Bindung“. Die Verwendung von virtuellen Funktionen bedarf der dynamischen Bindung, weil der Compiler zur Compilationszeit noch keine Entscheidung treffen kann.

```
if (...) p = &b; else p = &a;

p->f();   // ruft in Abhängigkeit von der Bedingung
          // Basisklasse::f() oder AbgeleiteteKlasse::f()
```

Die dynamische Bindung erlaubt dem Programmierer, Datenstrukturen zu erzeugen (z. B. Vektoren, Listen), die Objekte aus verschiedenen Klassen einer Klassenhierarchie enthalten und dann Methoden auf diese Objekte anzuwenden. Je nachdem zu welcher Klasse das Objekt gehört, kann die Methode entsprechend ausgewählt werden, d. h. das Objekt kann auf den Methodenaufruf passend reagieren.

Darüber hinaus bietet die dynamische Bindung auch das Hinzufügen von neuen Klassen und die Entfernung alter Klassen, ohne dass man den Code für die Verarbeitung von Datenstrukturen (z. B. Vektoren, Listen) dazu anpassen müsste. Tatsächlich ist es sogar möglich, neue Klassen zu einer Klassenhierarchie hinzuzufügen, ohne dass man ihren Quellcode haben müsste – abgesehen von den Headerfiles.

Rein virtuelle Methoden

Eine virtuelle Methode kann man in der Basisklasse undefiniert lassen (also nur deklarieren), indem man `= 0` dazuschreibt.

```
class Basisklasse {
  public:
    virtual void f() = 0;
  ...
};
```

Diese Methode ist jetzt „rein virtuell“. Die abgeleiteten Klassen *müssen* jetzt eine Definition für die Methode enthalten – bislang *durften* sie.

Abstrakte Basisklassen

Eine Klasse, die mindestens eine rein virtuelle Methode enthält, ist eine *abstrakte* Basisklasse. Sie kann nicht dazu verwendet werden, Objekte zu erzeugen, sondern sie dient lediglich als Ausgangspunkt für die Ableitung neue Klassen.

Als Beispiel nehmen wir mal an, wir richten eine Klasse `Konto` ein, von der die Klassen `Sparkonto` und `Girokonto` abgeleitet werden. Da alle Konten entweder Spar- oder Girokonten sein müssen, aber nicht nicht einfach nur ein Konto sein können, bietet es sich ein, `Konto` als abstrakte Basisklasse einzurichten.

9.3.5 Dynamische Member und Klassenableitung

Wenn eine Klasse dynamische Member hat, sind immer folgende drei Dinge erforderlich:

1. Destruktor, der den dynamischen Teil deallokiert
2. Kopierkonstruktor, der den dynamischen Teil richtig kopiert
3. Zuweisungsoperator, der den dynamischen Teil richtig kopiert

Falls eine von einer solchen Klasse abgeleitete Klasse ebenfalls dynamische Member hat, braucht sie ebenfalls diese drei Dinge. Hierbei muss sichergestellt sein, dass die geerbten dynamischen Member aus der Basisklasse und auch die neuen dynamischen Member aus der abgeleiteten Klasse richtig behandelt werden. Das soll hier an einem Beispiel genau dargestellt werden.

Die Basisklasse

```
class Fahrzeug {
 public:
  Fahrzeug();  // default-Konstruktor
  Fahrzeug (char *Herst, int Bj=0);
  Fahrzeug (int Bj);
  Fahrzeug (const Fahrzeug&);  // Kopier-Konstruktor
  ~Fahrzeug();  // Destruktor
  Fahrzeug &operator= (const Fahrzeug&);
  virtual void Ausgabe() const;
 private:
  char *Hersteller;
  int Baujahr;
};

Fahrzeug::Fahrzeug() {
    Hersteller=0; // Null-Pointer!!
```

```
    Baujahr=0;
} // Fahrzeug()

Fahrzeug::Fahrzeug (char *Herst, int Bj) {
    if (Herst == 0) {
      Hersteller = 0;
    } else {
      Hersteller = new char [strlen (Herst) + 1];
      strcpy (Hersteller, Herst);
    }
    Baujahr = Bj;
} // Fahrzeug()

Fahrzeug::Fahrzeug (int Bj) {
    Hersteller=0; // Null-Pointer!!
    Baujahr = Bj;
} // Fahrzeug()

Fahrzeug::Fahrzeug (const Fahrzeug &f) {
    if (f.Hersteller == 0) {
        Hersteller = 0;
    } else {
        Hersteller = new char [strlen (f.Hersteller) + 1];
        strcpy (Hersteller, f.Hersteller);
    }
    Baujahr = f.Baujahr;
} // Fahrzeug()

Fahrzeug::~Fahrzeug() { delete [] Hersteller; }

Fahrzeug &Fahrzeug::operator= (const Fahrzeug &f) {
    delete [] Hersteller;

    if (f.Hersteller == 0) {
        Hersteller = 0;
    } else {
        Hersteller = new char [strlen (f.Hersteller) + 1];
        strcpy (Hersteller, f.Hersteller);
    }
    Baujahr = f.Baujahr;
    return *this;
} // operator=

void Fahrzeug::Ausgabe() const {
    if (Hersteller != 0) cout << Hersteller << " ";
    cout << Baujahr;
} // Ausgabe()
```

Abgeleitete Klasse

```
class Auto: public Fahrzeug {
 public:
  Auto();
  Auto (char *Herst, int Bj, char * Modell, int Hubraum, int Leistung);
```

```
  Auto (char *Modell, int Hubraum, int Leistung);
  Auto (char *Hersteller, char *Modell);
  Auto (const Auto &);
  ~Auto();
  void Ausgabe() const;
  Auto &operator= (const Auto&);
 private:
  char *Modell;
  int ccmHubraum;
  int kWLeistung;
};
```

Default-Konstruktoren Die Klasse `Auto` basiert auf der Klasse `Fahrzeug`, weshalb sie sie `public` erbt. Beim Default-Konstruktor ergibt sich keine Besonderheit, weil der Default-Konstruktor der Basisklasse automatisch von den Konstruktoren der abgeleiteten Klasse aufgerufen werden, sofern man nichts anderes angibt. Falls die Basisklasse keinen Default-Konstruktor hat, meldet der Compiler einen Fehler.

```
Auto::Auto() {
    Modell = 0; // NULL-Zeiger
    ccmHubraum = 0;
    kWLeistung = 0;
} // Auto()

Auto::Auto (char *_Modell, int Hubraum, int Leistung) {
  if (_Modell == 0) {
    Modell = 0;
  } else {
    Modell = new char [strlen (_Modell) + 1];
    strcpy (Modell, _Modell);
  }
  ccmHubraum = Hubraum;
  kWLeistung = Leistung;
} // Auto()
```

Andere Konstruktoren Soll in einem Konstruktor der abgeleiteten Klasse ein anderer Konstruktor als der Default-Konstruktor der Basisklasse verwendet werden, weil auch Member aus der Basisklasse mit Parametern des Konstruktors initialisiert werden sollen, so ist der gewünschte Konstruktor hinter der Parameterliste durch einen Doppelpunkt getrennt anzugeben. Die Liste der hier aufgeführten Konstruktoren nennt man Element-Initialisierungliste.

```
Auto::Auto (char *Herst, int Bj, char *M, int Hubraum, int Leistung) :
Fahrzeug (Herst, Bj) {
  if (M == 0) {
    Modell = 0;
  } else {
    Modell = new char [strlen (M) + 1];
    strcpy (Modell, M);
  }
  ccmHubraum = Hubraum;
  kWLeistung = Leistung;
} // Auto()

Auto::Auto (char *Herst, char *_Modell) : Fahrzeug (Herst) {
  if (_Modell == 0) {
```

```
    Modell = 0;
  } else {
    Modell = new char [strlen (_Modell) + 1];
    strcpy (Modell, _Modell);
  }
  ccmHubraum = 0;
  kWLeistung = 0;
} // Auto()
```

Kopier-Konstruktor Beim Kopierkonstruktor wird in der Element-Initialisierungliste der Kopierkonstruktor der Basisklasse aufgerufen, dem das zu kopierende Objekt übergeben wird. Hier besteht ja eine problemlose Kompatibilität zwischen dem formalen Parameter vom Typ `Fahrzeug&` (Basisklassenreferenz) und dem aktuellen Parameter `Auto` (Objekt der abgeleiteten Klasse). Über den Kopier-Konstruktor der Klasse `Fahrzeug` werden die zur Basisklasse gehörenden Member initialisiert, über den Code im Rumpf des Konstruktors hier die Member der abgeleiteten Klasse.

```
Auto::Auto (const Auto &a) : Fahrzeug (a) {
  if (a.Modell == 0) {
    Modell = 0;
  } else {
    Modell = new char [strlen (a.Modell) + 1];
    strcpy (Modell, a.Modell);
  }
  ccmHubraum = a.ccmHubraum;
  kWLeistung = a.kWLeistung;
}
```

Zuweisungsoperator

Genau das gleiche muss auch bei der Erstellung des Zuweisungsoperators der abgeleiteten Klasse passieren. Man benötigen den Zuweisungsoperator aus der Basisklasse, um die Member der Basisklasse richtig zuzuweisen. Er ist hier ausnahmsweise mal nicht in der Infix-Schreibweise, sondern als Methode aufzurufen, wobei auch der Scope-Operator angewendet wird.

Das Objekt, dessen Methode verwendet wird, ist das aktuelle Objekt, also `*this`; die Methode heißt `Fahrzeug::operator=`, der Parameter ist eigentlich ein Fahrzeug, aber das Auto, das gerade zugewiesen werden soll, tut es der Kompatibilität wegen auch. Daher ist der Aufruf des Basisklassen-Zuweisungsoperators `this->Fahrzeug::operator=(a);` oder auch kurz `Fahrzeug::operator=(a);`.

Diesem kann dann die Zuweisung der Member aus der abgeleiteten Klasse folgen.

```
// Zuweisungsoperator einer abgeleiteten Klasse - muss
// intern den Zuweisungsoperator der Basisklasse verwenden!
Auto & Auto::operator= (const Auto &a) {
  Fahrzeug::operator=(a);
  delete [] Modell;
  Modell = new char [strlen (a.Modell) + 1];
  strcpy (Modell, a.Modell);
  ccmHubraum = a.ccmHubraum;
  kWLeistung = a.kWLeistung;
  return *this;
}
```

Destruktor und andere Methoden Beim Destruktor gibt es nichts zu berücksichtigen, weil die Destruktoren automatisch der Reihe nach aufgerufen werden. Es genügt also, sich um die Member der abgeleiteten Klasse zu kümmern.

```
Auto::~Auto() {
  delete [] Modell;
} // ~Auto()
```

Auch bei anderen Methoden gibt es nichts zu berücksichtigen. Methoden der Basisklasse kann man jederzeit aufrufen, was bei der Ausgabemethode auch sinnvoll erscheint.

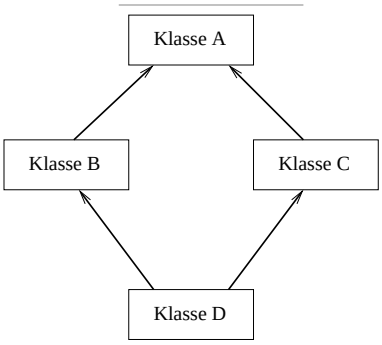
```
void Auto::Ausgabe() const {
  Fahrzeug::Ausgabe();
  if (Modell != 0) cout << " " << Modell;
  cout << " " << ccmHubraum << " " << kWLeistung;
} // Ausgabe()
```

9.4 Mehrfachvererbung

Eine Klasse kann nicht nur von einer Basisklasse erben, sondern auch von mehreren Basisklassen. Es werden dann alle Member – Daten- und Methodenmember – von allen Basisklassen geerbt. Einige Klassen können auch `public` geerbt werden, während andere `private` geerbt werden. Innerhalb der C++ Ein-/Ausgabebibliothek wird von Mehrfachvererbung reichlich Gebrauch gemacht, während die Sprache Java gar keine Mehrfachvererbung ermöglicht.

9.4.1 Virtuelle Basisklassen

```
class klasseA { public: int a; };
class klasseB: public klasseA { public: int b; };
class klasseC: public klasseA { public: int c; };
class klasseD: public klasseB, public klasseC { int d; };
```

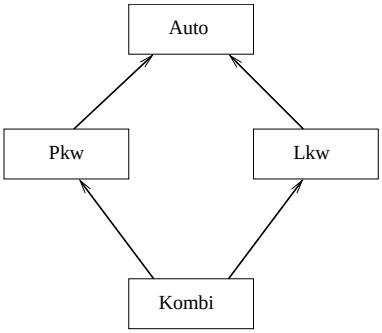


Erbt eine Klasse ein und dieselbe Basisklasse mehrfach, so haben ihre Objekte die Elemente dieser Basisklasse auch mehrfach enthalten, so dass man immer angeben muss, welche der dann gleichnamigen Datenmember nun gemeint sind.
Member der Klasse A werden von B und C gleichermaßen geerbt. Die Klasse D erbt die Elemente aus A auf zwei Wegen: über B und über C. Daher ist `a` auch zwei Mal vorhanden, als `klasseB::a` und als `klasseC::a`.
Hinweis: Die Pfeile in der Abbildung sind zu lesen als „erbt von“, nicht als Richtung der Vererbung.
Um das mehrfache Auftauchen gleicher Member zu verhindern, muss man die Basisklasse virtuell erben.

```
class klasseA { public: int a; };
class klasseB: virtual public klasseA { public: int b; };
class klasseC: virtual public klasseA { public: int c; };
class klasseD: public klasseB, public klasseC { int d; };
```

Jetzt gibt es nur ein einziges Datenelement `a` in der Klasse D, so dass auch keine Qualifikation mit dem Scope-Operator mehr nötig ist.

9.5 Mehrfachvererbung und virtuelle Basisklassen



Bei Verwendung von virtuellen Basisklassen gibt es bezüglich der Konstruktoren Besonderheiten zu berücksichtigen. Stellt man sich die hier dargestellte Klassenhierarchie vor, bei der in der Klasse `Auto` ein dynamischer Anteil ist, stellt man sofort fest, dass von den Konstruktoren der Klassen `Pkw` und `Lkw` nicht in beiden Fällen ein Konstruktor der Auto-Klasse aufgerufen werden darf, der für diesen dynamischen Anteil Speicher allokiert. Die Folge wäre, dass der dynamische Anteil zweimal allokiert würde.
Auch bei anderen Membern kann es Schwierigkeiten geben, wenn sie mehrfach initialisiert werden. In Konstruktoren kann beliebiger Code ablaufen. Ein mehrfaches Ausführen solchen Codes muss verhindert werden, weil es leicht zu Fehlern führt.
Als Folge davon wird von den Konstruktoren der virtuellen Basisklassen `Pkw` und `Lkw` kein Konstruktor der Auto-Klasse aufgerufen, so dass nur der Default-Konstruktor durch die Kombi-Klasse (einmal) aufgerufen wird. Auch wenn in den Konstruktoren der Pkw- und der Lkw-Klasse ein Konstruktor der Auto-Klasse in der Element-Initialisierungsliste steht, wird er nicht berücksichtigt, weil die Klasse `Auto` virtuell geerbt wurde.

Konstruktoren bei virtuellen Basisklassen

Um die Konstruktion eines Kombi-Objekts, das sowohl die Member `AnzahlTueren` aus der Klasse `Pkw` als auch das Member `Nutzlast` aus der Klasse `Lkw` besitzt, so zu realisieren, dass die Member, die aus der Klasse `Auto` geerbt wurden, genau einmal initialisiert werden, führt man in der Element-Initialisierungliste Konstruktoren aller drei Basisklassen auf.

Die Verwendung eines Auto-Konstruktors in der Element-Initialisierungliste eines Pkw- oder Lkw-Konstruktors funktioniert nur dann wie gewünscht, wenn der Pkw- oder Lkw-Konstruktor direkt aufgerufen wird, d. h. wenn wirklich ein Pkw- oder ein Lkw-Objekt konstruiert wird. Wird aber ein Kombi-Objekt konstruiert, so wird – um einen doppelten Aufruf des Auto-Konstruktors zu vermeiden – aus der Auto-Klasse nur der default-Konstruktor verwendet.

```
Kombi::Kombi (char *Herst, int Bj, int t, double n, int s) :
  Pkw (Herst, Bj, t), Lkw (Herst, Bj, n) {
  AnzSitzplaetze = s;
}
```

Hier kann also der Fehler auftreten, dass der Auto-Anteil gar nicht mit Herstellername und Baujahr initialisiert wird, weil die Aufrufe von `Pkw (Herst, Bj, t)` `Lkw (Herst, Bj, n)` keine Wirkung bezüglich des Auto-Anteils haben. Es sieht eher so aus, also würde versehentlich mehrfach initialisiert, tatsächlich geschieht es aber überhaupt nicht, weil der default-Konstruktor aus der Auto-Klasse implizit benutzt wird. Tatsächlich muss der Konstruktor aus `Kombi` den Konstruktor aus `Auto` selbst direkt aufrufen. Als Folge unterbleibt dann der Aufruf des default-Konstruktors aus der Auto-Klasse.

```
Kombi::Kombi (char *Herst, int Bj, int t, double n, int s) :
  Auto (Herst, Bj), Pkw (Herst, Bj, t), Lkw (Herst, Bj, n) {
  AnzSitzplaetze = s;
}
```

Das kann einerseits geschehen wir hier gezeigt, oder aber auch vereinfacht, wenn die entsprechenden Konstruktoren zur Verfügung stehen.

```
Kombi::Kombi (char *Herst, int Bj, int t, double n, int s) :
  Auto (Herst, Bj), Pkw (t), Lkw (n) {
  AnzSitzplaetze = s;
}
```

Dies ist die von der Wirkung her identische, aber vom Quellcode her offensichtlichere Variante, weil hier ganz klar ist, dass vom Auto-Konstruktor der Auto-Anteil, vom Pkw-Konstruktor nur der Pkw-Anteil und vom Lkw-Konstruktor nur der Lkw-Anteil initialisiert wird. Es wird – im Gegensatz zur obigen Version – auch nicht der Eindruck erweckt, der Auto-Anteil würde dreifach initialisiert.

Komplettes Beispielprogramm

Hier ein komplettes Beispielprogramm mit diesen Klassen. Der Kürze wegen sind die Zeilen für die Kontrollausgaben hier weggelassen, aber in der Originaldatei enthalten.

```
#include <cstring>   // <string.h> C-Stringfunktionen
#include <iostream>

class Auto {
public:
  Auto();
  Auto (char *Herst, int Bj);
  Auto (const Auto &);
  ~Auto();
  virtual void Ausgabe() const;
  Auto &operator= (const Auto&);
private:
  char * Hersteller;
  int Baujahr;
};

class Pkw : virtual public Auto {
public:
  Pkw();
  Pkw (int t);
  Pkw (char *Herst, int Bj, int t);
  void  Ausgabe();
private:
  int AnzTueren;
};


class Lkw : virtual public Auto {
public:
  Lkw();
  Lkw (double n);
  Lkw (char *Herst, int Bj, double n);
  void Ausgabe();
private:
  double Nutzlast;
};

class Kombi: public Pkw, public Lkw {
public:
  Kombi();
  Kombi (int s);
  Kombi (char *Herst, int Bj, int t, double n, int s);
  void Ausgabe();
private:
  int AnzSitzplaetze;
};

/********************** AUTO **********************/

Auto::Auto() {
    Hersteller=0; // Null-Pointer!!
    Baujahr=0;
} // Auto()

Auto::Auto (char *Herst, int Bj) {
    if (Herst == 0) {
      Hersteller = 0;
    } else {
      Hersteller = new char [strlen (Herst) + 1];
      strcpy (Hersteller, Herst);
    }
    Baujahr = Bj;
} // Auto()

Auto::Auto (const Auto &a) {
    if (a.Hersteller == 0) {
```

```
      Hersteller = 0;
    } else {
      Hersteller = new char [strlen (a.Hersteller) + 1];
      strcpy (Hersteller, a.Hersteller);
    }
    Baujahr = a.Baujahr;

} // Auto()

Auto::~Auto() {
    delete [] Hersteller;
} // ~Auto()

Auto &Auto::operator= (const Auto &a) {
    delete [] Hersteller;
    if (a.Hersteller == 0) {
        Hersteller = 0;
    } else {
        Hersteller = new char [strlen (a.Hersteller) + 1];
        strcpy (Hersteller, a.Hersteller);
    }
    Baujahr = a.Baujahr;
    return *this;
} // operator=

void Auto::Ausgabe() const {
    if (Hersteller != 0) cout << Hersteller << " ";
    cout << Baujahr << " ";
} // Ausgabe()

/************************** PKW ***********************/

Pkw::Pkw() {
  AnzahlTueren = 0;
} // Pkw()

Pkw::Pkw (int t) {
  AnzTueren = t;
} // Pkw()

Pkw::Pkw (char *Herst, int Bj, int t) : Auto (Herst, Bj) {
  AnzTueren = t;
} // Pkw()

void Pkw::Ausgabe() {
  cout << AnzTueren << "Türer ";
}

/************************** LKW ***********************/

Lkw::Lkw() {
  Nutzlast = 0.0;
} // Pkw()

Lkw::Lkw (double n) {
  Nutzlast = n;
} // Lkw()

Lkw::Lkw (char *Herst, int Bj, double n) : Auto (Herst, Bj) {
  Nutzlast = n;
} // Lkw()
void Lkw::Ausgabe() {
  cout << "Nutzlast " << Nutzlast << " t  ";
}

/************************** Kombi ***********************/

Kombi::Kombi() {
} // Kombi()

Kombi::Kombi (int s) {
  AnzSitzplaetze = s;
} // Kombi()

Kombi::Kombi (char *Herst, int Bj, int t, double n, int s) :
  Auto (Herst, Bj), Pkw (t), Lkw (n) {
  AnzSitzplaetze = s;
}

void Kombi::Ausgabe() {
```

```
  Auto::Ausgabe();
  Pkw::Ausgabe();
  Lkw::Ausgabe();
  cout << AnzSitzplaetze << " Plaetze ";
}

/************************** MAIN ***********************/

int main() {
  Kombi k ("Peugeot", 1976, 5, 1.3, 5);
  k.Ausgabe(); cout << endl;
}
```

Diese Datei steht auch unter `~bibjah/FORALL/CPP/raute.cpp` zur Verfügung.

9.6 Vererbung in der Ein-/Ausgabe-Bibliothek

Die C++ Ein-/Ausgabe-Bibliothek mach ausgiebig Gebrauch von Mehrfachvererbung und virtuellen Basisklassen. Das Studium dieser Bibliothek und ihres Aufbaus ist ein ausgezeichneter Weg, die Anwendung dieser Besonderheiten von C++ in der Praxis zu lernen.

Eingabe- und Ausgabeströme (Streams) haben sehr viel gemeinsam. Um Mehrfachprogrammierung zu vermeiden, haben die Designer der Ein-/Ausgabe-Bibliothek eine Klasse `ios` definiert, die die Gemeinsamkeiten der beiden Klassen `istream` und `ostream` enthält. `ios` enthält:

- Verwaltung des Status' des Stroms
- Speicherung von Formatierungsinformation
- Bereitstellung einer Schnittstelle zu den tieferliegenden Schichten der Ein-/Ausgabe-Bibliothek

`istream` und `ostream` sind beide von `ios` abgeleitet:

```
class istream: virtual public ios {
  ...
};

class ostream: virtual public ios {
  ...
};
```

Die Klasse `iostream` ist dann wieder von den beiden Klassen `istream` und `ostream` abgeleitet:

```
class iostream: public istream, public ostream {
  ...
};
```

Da die Klasse `ios` für `istream` und `ostream` eine virtuelle Basisklasse ist, sind alle Elemente aus ihr in `iostream` nur ein Mal enthalten. Weitere Details hierzu siehe Kapitel 10 auf Seite 71.

9.6.1 Die Klassen `ifstream`, `ofstream` und `fstream`

Auch diese Klassen sind über Mehrfachvererbung definiert:

```
class fstreambase: public virtual ios {
  ...
};

class ifstream: public fstreambase, public istream {
  ...
```

```
};
class ofstream: public fstreambase, public ostream {
  ...
};

class fstream: public fstreambase, public iostream {
  ...
};
```

Die Klasse `fstreambase` stellt dateispezifische Operationen zur Verfügung, die man bei anderen Datenströmen nicht benötigt, insbesondere das Öffnen und Schließen.

Die Dateistrom-Klassen haben Zugriff auf die „protected“ Member von `ios`, einschließlich `state`, die verändert wird, wenn ein Fehler auftritt. Beispielsweise wird ein „Fehler-Bit“ in `state` gesetzt, wenn eine Datei nicht im gewünschten Modus geöffnet werden kann. In Kapitel 10 auf der nächsten Seite wird gezeigt, wie man dieses Bit prüft.

9.7 Erweiterung der Stream-Klassen

Die `iostream` Bibliothek kann leicht erweitert werden, um Operationen auch auf anderen Strukturen als Dateien auszuführen. Ein Beispiel dafür ist die `strstream` Bibliothek, die Formatierung in Zeichenketten ermöglicht (entspricht `sprintf()`).

Diese Klasse ist auf ganz ähnliche Weise wie die Dateistrom-Klassen abgeleitet:

```
class strstreambase: public virtual ios {
  ...
};

class istrstream: public strstreambase, public istream {
  ...
};
class ostrstream: public strstreambase, public ostream {
  ...
};

class strstream: public strstreambase, public iostream {
  ...
};
```

Kapitel 10

Fortgeschrittene Ein- und Ausgabe

Bislang haben wir nur recht einfache Ein- und Ausgabeoperationen vorgenommen und sogar die Formatierung bis hierher verschoben, obwohl das in C damals recht einfach war mit `printf()` und den Format-Angaben. C++ bietet hier aber deutlich mehr, was andererseits dazu führt, dass das Ganze auch unübersichtlicher ist.

Ein Datenstrom hat verschiedene Werte, die seinen (aktuellen) Zustand beschreiben. Sie lassen sich einteilen in Status-Flags, Formatierungs-Flags und Öffnungsmodus. Die Klasse `ios` stellt die notwendigen Manipulatoren und Methoden zur Verfügung.

10.1 Status-Flags

Der Status eines Streams kann verschiedene Werte haben. Diese kann man nach Operationen abfragen oder auch verändern. Das Statuswort setzt sich aus folgenden Bits zusammen:

- `ios::goodbit` – Das ist eigentlich kein eigenes Bit, sondern wenn der Wert des Statuswortes gleich diesem ist, dann ist kein Bit gesetzt.
- `ios::eofbit` – Dieses Bit ist gesetzt, falls bei der letzten Leseoperation das Dateiende erreicht wurde, d. h. keine weiteren Daten mehr gelesen werden können.
- `ios::failbit` – Dieses Bit ist gesetzt, falls bei der letzten Leseoperation ein Fehler aufgetreten ist. Das kann auch schon der falsche Datentyp sein, d. h. beim Lesen von Zahlen standen Buchstaben im Datenstrom. Es ist damit keine Aussage über den möglichen Erfolg weiterer Einlesevorgänge verbunden.
- `ios::badbit` – Dieses Bit ist bei fatalen Fehlern gesetzt, d. h. wenn auch weitere Operationen auf diesem Datenstrom nicht erfolgreich sein können.

Entsprechende Methoden zum Abfragen des Statuswertes stehen zur Verfügung, siehe Tabelle 10.1 auf der nächsten Seite.

Die Methode `clear()` benötigt man immer nach Formatfehlern bei der Eingabe, um das `failbit` wieder zu löschen. Ansonsten bleibt es nämlich gesetzt und wird nachfolgende Operationen irritieren.

10.2 Format-Flags

Bislang haben wir immer mit den Standardformatierungen gearbeitet, aber auch bei C++ kann man sowohl das Ausgabeformat wie auch das Eingabeformat vielfältig beeinflussen.

Die Format-Flags werden – ähnlich wie das Statuswort eines Datenstroms – in einer Variablen vom Typ `ios::fmtflags` gespeichert. Meist handelt es sich um einen `long`-Wert, aber mit den Details der Implementation wollen wir uns nicht aufhalten, sondern die Schnittstelle beschreiben.

Mit der Methode `flags()` kann man die gesamte Flags-Variable lesen oder schreiben. Dadurch kann man die Einstellungen als Ganzes sichern und später wieder restaurieren. Um einzelne Flags zu setzen

Tabelle 10.1: Methoden zur Abfrage des Stream-Status

`bool good() const;` Liefert `true`, wenn der Datenstrom in gutem Zustand ist, d. h. kein Bit im Statuswort ist gesetzt.
`bool eof() const;` Liefert `true`, wenn im Statuswort das `eofbit` gesetzt ist.
`bool fail() const;` Liefert `true`, wenn im Statuswort das `failbit` oder das `badbit` gesetzt ist.
`bool bad() const;` Liefert `true`, wenn im Statuswort das `badbit` gesetzt ist.
`operator void*() const;` Dies ist der berühmte Konvertierungsoperator, der es ermöglicht, einen Datenstrom dort einzusetzen, wo ein logischer Ausdruck gefordert ist. Er liefert einen Nullzeiger, wenn `failbit` oder `badbit` gesetzt ist.
`operator !() const;` Der Negationsoperator liefert genau das logische Gegenteil vom `operator void*()`, d. h. eine wahren Wert, wenn mit dem Datenstrom etwas nicht stimmt.
`iostate rdstate() const;` Mit dieser Methode kann man das komplette Statuswort auf einmal abfragen.
`void clear (iostate state = goodbit);` Mittels dieser Methode kann man im Statuswort alle Bits löschen (default-Parameter) oder aber auch ein neues Statuswort setzen.
`void setstate (iostate state);` Mittels dieser Methode kann man im Statuswort ein weiteres Bit setzen. Tatsächlich ist es äquivalent zu `clear (rdstate() | state)`

oder zu löschen, verwendet man die Methoden `setf()` und `unsetf()`. Darüber hinaus gibt es noch Manipulatoren (aus `<iomanip>`), die man einfach in den Datenstrom einfügt und damit das Verhalten des Stroms beeinflusst.

Siehe auch Tabelle 10.2 auf der nächsten Seite.

Die Bits der Format-Flags kann man in Gruppen einteilen, in sogenannte Bitfelder. Das sind mehrere Bits, die gemeinsam genau eine Eigenschaft festlegen. Da gibt es das `ios::basefield`, das das Zahlensystem bei ganzzahligen Operationen festlegt. Es setzt sich aus den Bits `ios::oct`, `ios::dec` und `ios::hex` zusammen. Natürlich darf von diesen drei Bits immer nur ein einziges gesetzt sein, damit kein undefinierter Zustand entsteht. Ein Aufzähltyp wäre hier sicherlich die klarere Lösung gewesen.

Damit man nicht versehentlich unbestimmte Zustände erzeugt, kann man mit der Methode `setf()` genau ein Bit eines Bitfeldes setzen, d. h. evtl. vorher gesetzte Bits werden garantiert gelöscht. Um auf hexadezimale Ausgabe umzustellen, schreibt man `cout.setf (ios::hex, ios::basefield);` Von nun an werden alle ganzzahligen Werte als Hexadezimal-Zahlen ausgegeben.

Wenn man nicht weiß, welches Zahlensystem eingestellt ist, aber jetzt mal eben eine Hex-Zahl zwischendurch ausgeben will, ohne die Einstellung dauerhaft zu veränder, geht man so vor:

```
ios::fmtflags alteFlags = cout.flags();
cout.setf (ios::hex, ios::basefield);
cout << hexzahl << endl;
cout.flags (alteFlags);
```

Die alten Flags werden gespeichert, mit `setf()` werden die gewünschten eingestellt und bei der Ausgabe auch benutzt. Anschließend wird der alte Zustand wieder hergestellt.

10.2.1 Formatierung von Ganzzahlen

Üblicherweise werden ganze Zahlen einfach ziffernweise ausgegeben. Manchmal ist es aber nützlich zu wissen, in welchem Zahlensystem eine Zahl ausgegeben wurde. Dezimale Zahlen tragen nie ein Präfix,

Tabelle 10.2: Methoden für Format Flags

Methode	Beschreibung
`fmtflags flags();`	Liefert den aktuellen Zustand der Format-Flags.
`fmtflags flags (fmtflags f);`	Liefert den aktuellen Zustand der Format-Flags und stellt anschließend den im Parameter übergebenen neuen Wert ein.
`fmtflags setf (fmtflags f);`	Setzt zusätzlich alle Bits aus f in den Format-Flags. Vorsicht, kann leicht zu ungültigen Zuständen führen. Liefert alten Zustand der Format-Flags.
`fmtflags setf (fmtflags f, fmtflags bitfeld);`	Löscht alle Bits, die im bitfeld gesetzt sind, d. h. alle Bits eines Bitfelds, und setzt anschließend die Flags aus `f` im Bitfeld. Kann bei richtiger Handhabung nicht zu ungültigen Zuständen führen. Liefert alten Zustand der Format-Flags.
`fmtflags unsetf (fmtflags f);`	Löscht alle Bits, die in `f` gesetzt sind, in den Format-Flags. Vorsicht, kann leicht zu ungültigen Zuständen führen. Liefert alten Zustand der Format-Flags.

aber auf Wunsch tragen oktale Zahlen eine führende `0` und hexadezimale Zahlen ein `0x` davor. Das stellt man durch Setzen des Format-Flags `ios::showbase` ein.

Um auch positive Werte mit einem Vorzeichen zu versehen, setzt man das Flag `ios::showpos` – sonst wird nur das negative Vorzeichen angezeigt.

Um Hexziffern (und das Hex-Präfix) mit Groß- statt Kleinbuchstaben auszugeben, setzt man das Flag `ios::uppercase`.

Insgesamt ist die Handhabung dieser Format-Flags mit Manipulatoren leichter als mit den bislang benutzten Methoden. Tatsächlich verbirgt sich hinter den Manipulatoren auch nichts anderes, aber sie werden einfach in den Datenstrom mit eingebaut und nicht separat aufgerufen. Hierzu ist der Operator `<<` für Zeiger auf Funktionen overloadet.

Folgende Manipulatoren gibt es für die Formatierung von Ganzzahlen:

- `oct`, `hex`, `dec` – Es wird das entsprechende Flag im Bitfeld `ios::basefield` gesetzt. Es kann kein ungültiger Zustand eingestellt werden.
- `showbase`, `noshowbase` – Setzt bzw. löscht die Ausgabe von Präfixen fürs Zahlensystem
- `showpos`, `noshowpos` – Setzt bzw. löscht die Ausgabe von postiven Vorzeichen
- `uppercase`, `nouppercase` – Setzt bzw. löscht die Ausgabe von Großbuchstaben bei Hexzahlen.

Eine Ausgabe der Variablen i in allen drei Zahlenformaten kann also so erfolgen:

```
cout << dec << i << oct << i << hex << i << endl;
```

Beispielprogramm für die Formatierung von Ganzzahlen (ohne Verwendung von Manipulatoren):

```
#include <iostream>

int main () {
  int a=5, b=-345, c=38525;

  cout << "Ausgabe im Standardformat:" << endl;
  cout << a << " " << b << " " << c << endl;

  // speichere Standardzustand, zeige positive Vorzeichen (showpos)
  ios::fmtflags standard = cout.setf (ios::showpos);
```

```
    cout << "Ausgabe mit Vorzeichen:" << endl;
    cout << a << " " << b << " " << c << endl;
    // stelle Standardzustand wieder her
    cout.flags (standard);

    cout << "Ausgabe im Standardformat:" << endl;
    cout << a << " " << b << " " << c << endl;

    // lösche alle Bits im Bereich "basefield", setze ios::hex
    cout.setf (ios::hex, ios::basefield);
    cout << "Ausgabe im Hexadezimalformat:" << endl;
    cout << a << " " << b << " " << c << endl;

    // lösche alle Bits im Bereich "basefield", setze ios::hex
    cout.setf (ios::oct, ios::basefield);
    cout << "Ausgabe im Oktalformat:" << endl;
    cout << a << " " << b << " " << c << endl;

    return 0;
} // main
```

So sieht die Ausgabe des Programms aus:

```
Ausgabe im Standardformat:
5 -345 38525
Ausgabe mit Vorzeichen:
+5 -345 +38525
Ausgabe im Standardformat:
5 -345 38525
Ausgabe im Hexadezimalformat:
5 fffffea7 967d
Ausgabe im Oktalformat:
5 37777777247 113175
```

10.2.2 Formatierung von Fließkommazahlen

Fließkommazahlen werden auf zwei verschiedene Arten ausgegeben: entweder als Festkommawert `ios::fixed` oder in wissenschaftlicher Schreibweise, d. h. mit Exponent `ios::scientific`. Diese beiden Flags bilden zusammen das Bitfeld `ios::floatfield`, schließen sich also gegenseitig aus. Allerdings ist es möglich, dass keines der beiden Flags gesetzt ist – das ist sogar die default-Einstellung. Dann wird die Darstellung ohne Exponent verwendet, solange die Zahl nicht zu klein oder zu groß dafür wird.

Sind ganze Werte in einer Fließkommazahlen gespeichert, so wird auch kein Dezimalpunkt ausgegeben. Möchte man immer den Dezimalpunkt haben und damit auch immer dieselbe Anzahl Stellen nach demselben (siehe unten bei Genauigkeit), so setzt man das Flag `ios::showpoint`.

Auch hier gibt es die Möglichkeit, statt des direkten Setzens und Löschen der Flags Manipulatoren zu verwenden.

- `fixed`, `scientific` – Es wird das entsprechende Flag im Bitfeld `ios::floatfield` gesetzt. Es kann kein ungültiger Zustand eingestellt werden.
- `showpoint`, `noshowpoint` – Setzt bzw. löscht die ständige Ausgabe des Dezimalpunkts

Die Genauigkeit, d. h. die Anzahl Stellen hinter dem Komma (bzw. Dezimalpunkt) wird mit den `precision()`-Methoden eingestellt. Wie üblich kann man sie ohne Parameter aufrufen, wodurch der alte Wert geliefert wird. Übergibt man einen `int`-Parameter, so bestimmt dieser die Anzahl Nachkommastellen und es wird die alte Einstellung zurückgeliefert. Möchte man also mal eben eine Zahl `x` mit vier Stellen Genauigkeit ausgeben, ohne irgendwelche Einstellungen nachhaltig zu verändert, schreibt man:

```
int alteGenauigkeit = cout.precision (2);
cout << x << endl;
cout.precision (alteGenauigkeit);
```

Alternativ dazu kann man auch den Manipulator `setprecision (stellenanzahl)` verwenden, Beispiel `cout << setprecision (2) << a << endl;`

Ohne eines der Flags `ios::fixed` und `ios::scientific` gibt die Genauigkeit die Anzahl Stellen an, die verwendet wird. Ist `ios::showpoint` gesetzt, so wird diese Anzahl immer angezeigt, auch wenn gar kein Dezimalpunkt nötig wäre oder wenn die weiteren Stellen nur Nullen sind. Ohne `ios::showpoint` ist es die maximale Anzahl von zu benutzenden Stellen, es können aber – nach Notwendigkeit – auch weniger sein.

Ist die Anzahl der Vorkommastellen größer als mit der Genauigkeit darstellbar, wird auf Exponent-Schreibweise umgestellt. Die Mantisse bleibt erhalten, es kommen 4 Zeichen für den Exponenten hinzu: das `e`, das Vorzeichen und 2 Ziffern.

Gleiches geschieht, wenn die Anzahl Stellen nicht mehr darstellbar ist ohne Exponent und ohne die Gesamtbreite der Darstellung von Genauigkeit plus 4 Zeichen zu sprengen.

Bei erzwungender Festkommadarstellung mit `ios::fixed` gibt die Genauigkeit die Anzahl Nachkommastellen an. Ohne `ios::showpoint` ist es die maximale Anzahl Stellen, mit `ios::showpoint` wird immer genau diese Anzahl Nachkommastellen verwendet. Sehr kleine Zahlen sind dann nicht mehr von Null unterscheidbar, sehr große Zahlen führen zu immer größerer Vorkomma-Stellenanzahl bei immer noch voller Nachkommastellen-Anzeige, auch wenn diese Stellen keinerlei Signifikanz mehr haben.

Bei erzungender Exponent-Darstellung mit `ios::scientific` werden immer die angegebene Anzahl Nachkommastellen und eine Vorkommastelle angezeigt, dazu ein Exponent – auch wenn dieser 0 sein sollte. Hier ist die Anzahl Zeichen, die benötigt wird, garantiert für jeden Wert gleich.

Beispielprogramm für die Formatierung von Fließkommazahlen (ohne Verwendung von Manipulatoren):

```
#include <iostream>

int main () {
  double d = 344.3454, e = 12;

  cout << "Standardformat: " << d << " " << e << endl;

  ios::fmtflags standard = cout.setf (ios::showpoint);
  cout << "mit gesetztem ios::showpoint "
   << d << " " << e << endl;

  cout.setf (ios::fixed);
  cout << "mit zusätzlich gesetztem ios::fixed " << d << " " << e << endl;

  // lösche alle Bits im Floatfield, setzte ios::scientific
  cout.setf (ios::scientific, ios::floatfield);
  cout << "mit ios::showpoint + ios::scientific " << d << " " << e << endl;

  // setze Ausgabepräzision auf 8 Stellen
  int old_prec = cout.precision(8);
  cout << "zusätzlich 8 Stellen :"
       << d << " " << e << endl;

  // lösche alle Bits im Floatfield, setzte ios::fixed
  cout.setf (ios::fixed, ios::floatfield);
  cout << "dasselbe mit ios::fixed " << d << " " << e << endl;
```

```
    return 0;
} // main
```

So sieht die Ausgabe des Programms aus:

```
Standardformat: 344.345 12
mit gesetztem ios::showpoint 344.345 12.0000
mit zusätzlich gesetztem ios::fixed 344.345400 12.000000
mit ios::showpoint + ios::scientific 3.443454e+02 1.200000e+01
zusätzlich 8 Stellen :3.44345400e+02 1.20000000e+01
dasselbe mit ios::fixed 344.34540000 12.00000000
```

10.2.3 Sonstige Formatierungsmethoden

Gerade die Gesamtbreite der Ausgabe kann man aber auch für alle anderen Ausgaben genau vorbestimmen mit der Methode `width()`, der man eine natürliche Zahl übergibt. Diese Festlegung hat aber – im Gegensatz zu allen anderen Einstellungen – nur für die genau darauf folgende Ausgabe eine Wirkung und nicht etwa bis zu einer neuen, anderen Einstellung. Alternativ kann auch der Manipulator `setw (`*`breite`*`)` verwendet werden, Beispiel: `cout << setw (10) << a << endl;`

Die Feldbreite hat auch bei Eingaben eine Wirkung, wenn man Zeichenketten mit dem Operator `>>` einliest. Es werden maximal `n - 1` Stellen eingelesen, wenn die Breite auf `n` eingestellt ist.

Zusätzliche Stellen, die nicht benötigt werden, werden mit Leerzeichen aufgefüllt. Wahlweise kann man auch ein anderes Füllzeichen bestimmen mit der Methode `int fill (int ch) const;`. Wie erwartet wird das alte Füllzeichen als Wert zurückgeliefert. Auch hier gibt es eine overloadete Version, die nur das aktuelle Füllzeichen liefert, ohne etwas zu verändern: `int fill() const;`. Die Einstellung des Füllzeichens auf `*` kann für das Drucken von Schecks nützlich sein. Alternativ kann auch der Manipulator `setfill (`*`füllzeichen`*`)` verwendet werden, Beispiel `cout << setfill ('*') << betrag << endl;`

Üblicherweise wird in den Feldern rechtsbündig ausgegeben. Das kann man durch die Flags im Bitfeld `ios::adjustfield` beeinflussen.

- `ios::left`, Manipulator `left` – Durch Setzen dieses Flags oder Einfügen des Manipulators in den Datenstrom werden ab jetzt die Ausgaben linksbündig getätigt.
- `ios::right`, Manipulator `right` – Durch Setzen dieses Flags oder Einfügen des Manipulators in den Datenstrom werden ab jetzt die Ausgaben rechtsbündig getätigt.
- `ios::internal`, `internal` – Durch Setzen dieses Flags oder Einfügen des Manipulators in den Datenstrom werden ab jetzt die Ausgaben mit Vorzeichen linksbündig und der Zahl rechtsbündig getätigt. Die Füllzeichen erscheinen hier also zwischen Vorzeichen und dem Wert selbst.

Der aktuelle GNU C++ Compiler (Version 2.95) hat diese Manipulatoren noch nicht, so dass man die Flags mit `setf()` setzen muss.

10.2.4 Weitere Manipulatoren

Bereits früher verwendet haben wir die Manipulatoren `flush` zum Leeren des Ausgabepuffers und `endl` zum Senden eines Newline-Zeichens und anschließendem Leeren des Ausgabepuffers. Bei der Ausgabe in Stringstreams (siehe Kapitel 7.7.2 auf Seite 50) findet noch `ends` Verwendung, das ein Stringendezeichen in den Stringstream schreibt und den Ausgabepuffer leert.

Nur für Eingabeströme verwendbar ist der Manipulator `ws`, der dazu dient, Whitespaces zu überspringen. Anwendung: `cin >> ws ; cin.getline (a, sizeof (a));` Evtl. noch im Eingabepuffer verbliebene Newlines, Tabs oder Leerzeichen werden nicht in `a` gespeichert und führen auch nicht zur Beendigung der Eingabe.

Ob bei Eingaben mittels `>>` alle Whitespaces übersprungen werden, kann man durch das Flag `ios::skipws` festlegen. Dieses Flag ist normalerweise gesetzt.

Möchte man bei der Ausgabe von booleschen Werten nicht `0` und `1` ausgegeben haben, sondern `false` und `true`, so muss man das Flag `ios::boolalpha` setzen.

10.3 Eigene Manipulatoren

Die Operatoren `<<` und `>>` sind so gestrickt, dass sie auch Funktionen (genauer: Zeiger auf Funktionen) als Argumente akzeptieren. Falls die „ausgegebene“ bzw. „eingelesene“ Funktion einen der folgenden Prototypen hat, so wird nicht der Zeiger ausgegeben, sondern die Funktion aufgerufen:

```
ios & f (ios &);
istream & f (istream &);
ostream & f (ostream &);
```

Der Manipulator `hex` beispielsweise ist eine Funktion, die so aussieht:

```
inline ios& hex(ios& i) {
   i.setf(ios::hex, ios::dec|ios::hex|ios::oct);
   return i;
}
```

Natürlich hätte im rechten Parameter auch `ios::basefield` stehen können, was sogar kürzer gewesen wäre.

Es ist also völlig egal, welche der drei folgenden Anweisungen man benutzt, denn es passiert immer genau dasselbe:

```
cout << hex;
cout.operator<< (hex);
hex (cout);
```

Möchten wir selbst einen neuen Manipulator schreiben, muss dieser lediglich der Schnittstellenkonvention genügen. Ein schönes Beispiel dafür wäre der Manipulator `DM`, der dafür sorgt, dass die nachfolgend ausgegebene Fließkommazahl in einem ordentlichen Währungsformat erscheint. Da der Manipulator `fixed` beim aktuellen GNU C++ Compiler (Version 2.95) fehlt, definieren wir den gleich mit.

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

ios &fixed (ios &stream) {
  stream.setf (ios::fixed, ios::floatfield);
  return stream;
} // Manipulator fixed

ostream &DM (ostream &os) {
  os << "DM ";
  os.width (10);
  os << fixed << setprecision (2);
  return os;
} // Manipulator DM

int main () {
  double x = 234.445;

  cout << DM << x << endl;
} // main()
```

Die Erzeugung von Manipulatoren mit Parametern ist ungleich komplizierter. Daher ist eher anzuraten, normale Methoden zu benutzen.

10.4 Verbinden von Streams

Die Datenströme für die Standard-Ein- und -Ausgabe sind miteinander verbunden (tied), so dass der Puffer von `cout` immer geflusht wird, bevor etwas von `cin` eingelesen wird. So wird sichergestellt, dass Eingabeaufforderungen immer sichtbar sind, wenn eine Eingabe verlangt wird. Ein manuelles Aufrufen von `flush` (als Methode oder Manipulator) entfällt dadurch.

Tatsächlich kann man auch selbst einen Ausgabestrom an einen Eingabestrom anbinden oder auch abfragen, welcher Ausgabestrom an einen bestimmten Eingabestrom angebunden ist. Hierzu gibt es in der Klasse `ios` diese Methoden:

```
ostream* tie() const;        // liefert Adresse des angebundenen Streams
ostream* tie (const ostream *name); // bindet name an aktuellen Stream
```

Das Binden von Ausgabe- an Eingabestrom muss nicht unbedingt erforderlich sein, aber es garantiert das gewünschte Verhalten auf portable Art.

Kapitel 11

Templates

Neuere Versionen von C++ – insbesondere natürlich das ANSI/ISO C++ – haben einen Mechanismus zur Herstellung generischer Klassen. Genauso wie „strukturierte Programmierung“ ein bekanntes Paradigma ist, ist auch die „generische Programmierung“ eines. Hierbei wird nicht nur Wert auf Wiederverwendbarkeit gelegt, was schon länger angestrebt wird als es C++ gibt, sondern man bemüht sich von Anfang an darum, Code auf eine Weise zu schreiben, dass er für möglichst viele Anwendungsfälle unverändert benutzt werden kann. Während also die Wiederverwendbarkeit darauf zielt, Code mit wenigen Änderungen in anderen Zusammenhängen nutzbar zu machen, ist bei der generischen Programmierung keinerlei Aufwand zu treiben, sondern der Code unmittelbar zu gebrauchen. Hierzu muss der Compiler einigen Aufwand treiben und entsprechend neuen Maschinencode für den neuen Anwendungsfall erzeugen.

Nehmen wir als Beispiel mal unsere selbstgeschriebene String-Klasse. Dabei handelt es sich um eine intelligentere und praktischere Art der Verwaltung von Zeichenketten, die aus Zeichen vom Typ `char` bestehen. Aber ist das schon die ganze Wahrheit? Schließlich gibt es auch den Datentyp `wchar_t` für Unicode-Zeichen (`www.unicode.org`) und wer weiß, was zukünftig alles noch kommt. Grundsätzlich werden die Zeichenketten, die aus 16-Bit-Zeichen bestehen, auch nicht anders zu behandeln sein als die Zeichenketten aus 8-Bit-Zeichen. Sie werden nach wie vor eingelesen, aneinandergereiht, ausgegeben, nach Zeichen durchsucht, in Worte zerlegt oder es werden einzelne Zeichen abgefragt oder ersetzt.

Man sieht also, dass es eigentlich überflüssig ist, all diese Mechanismen erneut zu implementieren, nur weil sich der Basisdatentyp von einer Zeichensorte zu einer anderen ändert. Genau hier setzt der Template-Mechanismus ein. Das Wort Template bedeutet so viel wie Schablone oder Vorlage. Man erstellt also für eine bestimmte Konstruktion – hier eben String – eine Schablone, die jederzeit für verschiedene Datentypen (Klassen) angewendet werden kann, ohne dass man die Konstruktion selbst wiederholen müsste.

Hinweis: Tatsächlich ist auch die String-Klasse in der C++ Standardbibliothek als Template aufgebaut und mit zur Zeit zwei Zeichentypen realisiert. Ebenso übrigens auch die Datenströme (Streams), die als Ströme von Zeichen implementiert sind, wobei die Zeichen `char` oder auch `wchar_t` sein können.

11.1 Beispiel String-Template

Die hier als Beispiel dienende String-Klasse soll nur das Notwendigste enthalten, um zu zeigen, wie Templates überhaupt funktionieren. Die Klasse selbst wird mit einem Parameter versehen, der keinen Wert, sondern einen Typen enthält. Auch wenn das Wort `class` davorsteht, handelt es sich bei `C` um einen Typen, der keine Klasse sein muss.

Der Parameter ist – im Gegensatz zu den bisher verwendeteten – in spitzen Klammern angegeben.

```
template <class C> class String {
  private:
    int len;   // Länge
    C   *rep;  // Repräsentation des Strings als Zeiger auf
               // Template-Parameter-Typ C
```

```
  public:
    String() { rep = 0; len = 0; }
    String (const C *, int);
    String (const String &);
    ~String();
    String & operator= (const String &);
    void print ();
}; // END OF template class String
```

Die Klassendeklaration beginnt mit dem Schlüsselwort `template`, gefolgt von dem Template-Parameter in spitzen Klammern. Der Parameter ist hier eine Klasse bzw. ein Typ und wird `C` genannt. Der Name ist natürlich völlig beliebig, aber es werden oft einzelne Großbuchstaben dafür verwendet. Innerhalb der Klassendefinition wird dieser Bezeichner `C` für den Basisdatentyp des Strings stehen und kann ganz normal wie jeder andere Typ verwendet werden.

Bei der Realisierung einer konkreten Klasse mit Hilfe der Schablone schreibt man dann beispielsweise

```
String<char> cs;
```

Wir verwenden also den Klassennamen `String`, gefolgt von dem nun konkreten Basisdatentypen `char` wiederum in spitzen Klammern. Diese Kombination stellt jetzt die Beschreibung einer konkreten Klasse dar, so dass das `cs` dahinter jetzt einen Character-String deklariert. Der hier beschriebene Vorgang wird auch Klassen-Instantiierung genannt.

Im einfachsten Fall schreibt man die komplette Definition des Klassentemplates einschließlich der Methoden in die Headerdatei. Dann steht dem Compiler alles zur Verfügung, was der zur Erzeugung des Codes für die einzelnen Instantiierungen benötigt. Das kann für das String-Template so aussehen:

```
#include <iostream>

template <class C> class String {
private:
  int len;   // Länge
  C   *rep;  // Repräsentation des Strings als Zeiger auf
             // Template-Parameter-Typ C

public:
  String() {
    rep = 0;   // Zeiger initialisieren
    len = 0;   // Länge ist 0
  }

  // Konstruktor aus Array bzw. Zeiger auf Template-Parameter-
  // Typ C und Längenangabe
  String (const C* s, int l) {
    rep = new C [l];                // Platz allokieren
    len = l;                        // Länge zuweisen
    for (int i=0; i < l; i++) {
      rep [i] = s [i];              // Elemente kopieren
    } // for
  }

  // Kopierkonstruktor
  String (const String& s) {
    rep = new C [s.len];            // Platz allokieren
```

```
      len = s.len;                      // Länge zuweisen
      for (int i=0; i < s.len; i++) {
        rep [i] = s.rep [i];            // Elemente rüberkopieren
      }
    }

    // Destruktor
    ~String() {
      if (rep != 0) delete [] rep;   // ggf. Platz freigeben
    } // Destruktor

    // Zuweisungsoperator
    String & operator= (const String &s2) {
      if (rep != 0) delete [] rep;   // ggf. alten Platz freigeben
      rep = new C [s2.len];          // neuen Platz allokieren
      len = s2.len;                  // Länge zuweisen
      for (int i=0; i < len; i++) {
        rep [i] = s2.rep [i];        // Elemente rüberkopieren
      }
      return *this;
    } // operator=

    // Ausgabe
    void print() {
      for (int i=0; i < len; i++) {  // Elemente einzeln ausgeben
        cout << rep [i];
      }
    } // print()
  }; // END OF template class String
```

Mit den einzelnen Elementen eines Strings, deren Typ vom Parameter `C` bestimmt wird, werden einige Operationen durchgeführt. Beispielsweise werden sie sehr oft einander zugewiesen, in der Methode `print()` werden sie mittels des `<<`-Operators ausgegeben. Dieses muss mit dem verwendeten Typen möglich sein, sonst kann man keinen String aus ihm konstruieren. Der Compiler sorgt dafür, dass jeweils die zum Typen `C` passenden Operationen aufgerufen werden.

Wenn man in einem kleinen Beispiel-Hauptprogramm das Template instantiiert, kann man es gleich ausprobieren:

```
int main() {
  String<char> cs ("hallo", 5);
  cs.print();
  cout << endl;
} // main()
```

Es wird ein String aus gewöhnlichen `char` erzeugt, die verwendeten Operationen werden für diesen Typen gebaut und compiliert. Um zu zeigen, dass man nicht nur diesen Typen, sondern fast jeden Datentypen benutzen kann, um daraus Strings zu bauen, erweitern wir das Hauptprogramm:

```
int main() {

  // String aus char
  String<char> cs ("hallo", 5);
  cs.print();
  cout << endl;
```

```
    // String aus int
    String<int> is ((int[]){1, 2, 3}, 3);
    is.print();
    cout << endl;

    // String aus double
    String<double> ds ((double[]){1.0, 2.2, -3.4}, 3);
    ds.print();
    cout << endl;

    // String aus char-Arrays
    String<char*> cps ((char*[]){"hallo", "du", "da"}, 3);
    cps.print();
    cout << endl;
} // main()
```

Das letzte Beispiel zeigt besonders die Flexibilität des Konzepts. Wer hätte schon von Anfang an daran gedacht, einen String aus C-Zeichenketten zusammenzusetzen? Da aber der Typ `char*` die notwendigen Eigenschaften hat, funktioniert das durchaus. Allerdings muss man mit der nicht ganz korrekten Zuweisungsoperation von `char*` leben, die ja nicht die Zeichenfolgen, sondern nur die Zeiger kopiert. Ein Weg aus dieser Misere sind sogenannte Spezialisierungen, für die verwiesen wird auf das Buch von Stroustrup, Die C++ Programmiersprache, 3. Auflage, Kapitel 13.5.

11.2 Separate Compilation von Templates

Grundsätzlich möchten wir nicht, dass so viel Code wie im vorigen Abschnitt gezeigt in der Headerdatei steht, weil dieser ganze Code bei jedem einzelnen Compilationsvorgang wieder durch den Compiler geschleust werden muss, was Rechenzeit kostet. Viel praktischer ist es, nur die Klassendefinition und Methodendeklarationen in die Headerdatei zu schreiben und die Definitionen der Methoden in einer separaten `.cpp`-Datei abzulegen. Ausserdem wird die Headerdatei viel kürzer und übersichtlicher, wenn nur die Funktionsköpfe enthalten sind.

Da aber zum Zeitpunkt der Übersetzung noch gar nicht feststeht, für welche Datentypen das Template später einmal instantiiert werden soll, benötigt der Compiler aber die Definitionen bei jeder Instantiierung erneut. Tatsächlich ist der Template-Mechanismus nämlich nicht viel mehr als ein – zugegeben recht komplexer – Textersetzungsvorgang, dessen Ergebnis anschließend ganz normal compiliert wird.

Wir können ja mal versuchen, Deklarationen und Definitionen zu trennen. Die Headerdatei sähe dann so aus:

```
#ifndef _string_template_h
#define _string_template_h

#include <iostream>

template <class C> class String {
private:
  int len;
  C   *rep;

public:
  String() { rep = 0; len = 0; }

  String (const C*, int);
  String (const String&);
```

```
    ~String();
    String & operator= (const String &);
    void print ();
  }; // END OF template class String
#endif
```

Für den default-Konstruktor lohnt sich ein Ablegen in der separaten Datei nicht, daher ist er hier definiert. Ansonsten stehen hier nur die Deklarationen. Die Quelldatei für die Definitionen sieht so aus:

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


template<class C>
String<C>::String (const C* s, int l) {
  rep = new C [l];
  len = l;
  for (int i=0; i < l; i++) {
    rep [i] = s [i];
  } // for
}

template<class C>
String<C>::String (const String& s) {
  rep = new C [s.len];
  len = s.len;
  for (int i=0; i < s.len; i++) {
    rep [i] = s.rep [i];
  }
}

template<class C>
String<C>::~String() {
  cout << "Destruktor" << (void*) rep << endl;
  if (rep != 0) delete [] rep;
} // Destruktor


template<class C>
String<C> & String<C>::operator= (const String &s2) {
  if (rep != 0) delete [] rep;
  rep = 0;
  len = 0;
  rep = new C [s2.len];
  len = s2.len;
  for (int i=0; i < len; i++) {
    rep [i] = s2.rep [i];
  } // for
  return *this;
} // operator=


template<class C>
void String<C>::print() {
```

```
  for (int i=0; i < len; i++) {
    cout << rep [i];
  } // for
} // print()
```

Jede einzelne Methode, Konstruktoren und Destruktoren müssen mit dem Zusatz `template <class C>` und der Angabe der Klasse `String<C>` vor dem Scope-Operator versehen werden. Wird ein Objekt der Klasse als Funktionsrückgabetyp verwendet, muss auch hier `String<C>` stehen, während innerhalb der Parameterliste oder in der Funktion selbst `String` genügt.

Schreibt man jetzt wieder ein kleines Test-Hauptprogramm, so ergibt sich beim Linken ein Problem. Hier das Test-Hauptprogramm:

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

int main() {
  String<char> cs ("hallo", 5);
  cs.print();
  cout << endl;
}
```

Beim Linken sieht man folgende Meldungen (Verwendung des GNU C++ Compilers auf Linux):

```
~> g++ -c string_template.cpp
~> g++ template_main.cpp string_template.o
In function ‘main’:
undefined reference to ‘String<char>::String(char const *, int)’
undefined reference to ‘String<char>::print(void)’
undefined reference to ‘String<char>::~String(void)’
undefined reference to ‘String<char>::~String(void)’
collect2: ld returned 1 exit status
```

Diesen Meldungen kann man entnehmen, dass es keine Instantiierungen des Templates mit dem Parameter `char` gibt, der Linker sie aber braucht, um das Programm zu binden. Tatsächlich haben wir bislang nur das Template selbst erzeugt, aber keinen Maschinencode für irgendeine konkrete Instantiierung, denn dazu bräuchte der Compiler bei dem Compilerlauf, in dem ein `String<char>` steht, auch die Information über das Template. Dies haben wir aber soeben auf zwei Compilerläufe verteilt.

Für dieses Problem gibt es zwei Lösungen:

- Man kann den Template-Quellcode in jeden Compilerlauf mit einbeziehen. Das geht z. B. dadurch, dass man die `.cpp`-Datei am Ende der Headerdatei includet. Das wird auch bei der String-Klasse der C++ Standardbibliothek so gemacht. Der Nachteil dieser Lösung ist die höhere Aufwand beim Compilieren und dass ein und dieselbe Instantiierung bei einem größeren Programm für jedes Modul erneut geschieht und erst vom Linker die überflüssigen Exemplare entfernt werden.

- Die zweite Möglichkeit ist die, konkrete Instantiierungen in der `.cpp`-Datei vorzunehmen, ohne dazu Objekte anzulegen. Man fügt einfach ein `template class String<char>;` ans Ende der Datei an. Der Nachteil dieser Methode ist aber, dass nur die Instantiierungen vorhanden sind, die man in der Quelldatei erzeugt hat. Das relativiert den Sinn von Templates natürlich.

Man muss sich also überlegen, für welche Variante man sich entscheidet. Schreibt man eine Klasse für den eigenen Gebrauch im Rahmen eines größeren Projekts, so ist die zweite Lösung die naheliegendere, weil man jederzeit die evtl. zusätzlich benötigten Instantiierungen in der Quelldatei nachtragen kann.

Schreibt man dagegen eine Bibliothek, die in allen möglichen Zusammenhängen verwendet werden soll, ist die erste Möglichkeit die „wartungsfreie“ und damit bessere.

11.3 Template-Beispiel Warteschlange

Ein weiteres Beispiel soll die Anwendung von Templates illustrieren. Es handelt sich um eine Wartschlange für Objekte beliebigen Typs, wobei in einer Wartschlange natürlich alle Objekte vom selben Typ sein müssen. Eine Warteschlange arbeitet nach dem FIFO-Prinzip, d. h. das Objekt, das zuerst in die Schlange gestellt wurde, wird als erstes wieder herausgeholt. Es ist also quasi das Gegenteil von einem Stack, bei dem immer das zuletzt auf den Stapel gelegte Objekt als erstes wieder heruntergenommen wird.

So sieht die Headerdatei aus, die hier wieder alle Definitionen enthält, damit das Template mit beliebigen Datentypen instantiiert werden kann:

```
#ifndef _queue_h
#define _queue_h

#include <iostream>

template <class T> class Queue {
  private:
    T *vekPtr;
    int max;
    int tip, tail;
    bool empty;

  public:
    Queue (int n);          // erzeuge Queue mit n Elementen
    Queue (const Queue&);   // Kopierkonstruktor
    ~Queue();               // Destruktor

    const Queue<T> & operator= (const Queue &); // Zuweisung

    bool isEmpty() const { return empty; }
    bool enqueue (const T&); // stelle in Warteschlange
    T dequeue();             // entnehme aus Warteschlange
};


template <class T>
Queue<T>::Queue (int n) {
  vekPtr = new T[n];
  max = n;  tip = tail = 0; empty = true;
} // Konstruktor


template <class T>
Queue<T>::Queue (const Queue &q) {
  max    = q.max;
  tip    = q.tip;
  tail   = q.tail;
  empty  = q.empty;
  vekPtr = new T[max];
  for (int i=0; i < max; i++) {
    vekPtr [i] = q.vekPtr [i];
  } // for
} // Kopierkonstruktor
```

```
template <class T>
Queue<T>::~Queue() {
  delete [] vekPtr;
}


template <class T>
bool Queue<T>::enqueue (const T &a) {
  if (tip != tail || empty) {  // falls noch Platz frei
    vekPtr [tail] = a;
    tail = ++tail % max;
    empty = false;
    return true;
  }
  return false; // Es war kein Platz mehr!
} // enqueue()


template <class T>
T Queue<T>::dequeue() {
  if (empty) {
    cerr << "Queue leer!" << endl; exit (1);
    // kann später mit Ausnahmebehandlung verbessert werden
  }

  int i = tip;
  tip = ++tip % max;
  if (tip == tail) empty = true;
  return vekPtr [i];
} // dequeue()
#endif
```

Auch hierzu brauchen wir eine kleines Test-Hauptprogramm, das noch zwei Funktionen mit sich bringt, die eine Warteschlange mit Grunddatentyp `int` als Parameter haben.

```
#include <iostream>
#include <iomanip>
#include "queue.h"

Queue<int> intQueue1 (100); // max. 100 int-Werte in der Schlange

void einlesen (Queue<int> &q);
void auslesen (Queue<int> &q);

int main () {
  einlesen (intQueue1);
  Queue<int> intQueue2 (intQueue1); // Kopierkonstruktor
  cout << "\nInhalt der 1. Schlange:\n";
  auslesen (intQueue1);
  cout << "\nInhalt der Kopie der 1. Schlange:\n";
  auslesen (intQueue2);
  return 0;
} // main()
```

```
void einlesen (Queue<int> &q) {
  int data;
  cout << "\nBitte int-Werte != 0 eingeben\n";
  while (cin >> data && data != 0) {
    if (!q.enqueue (data)) {
      cerr << "Warteschlange voll!" << endl;
      break;
    } // if
  } // while
} // einlesen()


void auslesen (Queue<int> &q) {
  if (!q.isEmpty()) {
    cout << "Inhalt der Warteschlange:\n";
    while (!q.isEmpty()) cout << setw (8) << q.dequeue();
    cout << endl;
    return;
  } // if
  cout << "Warteschlange ist leer\n";
} // auslesen
```

11.4 Funktions-Templates

Die beiden Funktionen `einlesen()` und `auslesen()` sind ausdrücklich nur für `int`-Schlangen geeignet. Aber auch diese kann man sehr leicht mit einem Typen parametrisieren, so dass sie für beliebigen Warteschlangen verwendbar werden:

```
// Prototypen
template<class T> void einlesen (Queue<T> &q);
template<class T> void auslesen (Queue<T> &q);

// Definitionen
template <class T>
void einlesen (Queue<T> &q) {
  T data;
  cout << "\nBitte Werte != 0 eingeben\n";
  while (cin >> data && data != 0) {
    if (!q.enqueue (data)) {
      cerr << "Warteschlange voll!" << endl;
      break;
    } // if
  } // while
} // einlesen()

template <class T>
void auslesen (Queue<T> &q) {
  if (!q.isEmpty()) {
    cout << "Inhalt der Warteschlange:\n";
    while (!q.isEmpty()) cout << setw (8) << q.dequeue();
```

```
      cout << endl;
      return;
    } // if
    cout << "Warteschlange ist leer\n";
  } // auslesen
```

Durch die Parametrisierung der Funktionen können wir jetzt auch problemlos das ganze Programm auf `double` umschreiben oder einfach auch nur eine weitere Warteschlange mit dem Grunddatentyp `double` hinzufügen:

```
Queue<double> doubleQueue1 (100); // max. 100 double-Werte in der Schlange
einlesen (doubleQueue1);
Queue<double> doubleQueue2 (doubleQueue1); // Kopierkonstruktor
cout << "\nInhalt der 1. Schlange:\n";
auslesen (doubleQueue1);
cout << "\nInhalt der Kopie der 1. Schlange:\n";
auslesen (doubleQueue2);
```

Die Einschränkung bezüglich der verwendbaren Typen ergibt sich wieder aus den Operationen, die auf die Werte vom Grunddatentyp angewendet werden. Hier also müssen die Objekte, die in die Warteschlange eingereiht werden sollen, zuweisbar, einlesbar und ausgebbar mit den Standardoperatoren und vergleichbar mit einer ganzen Zahl (0) sein. Das kann man aber grundsätzlich sogar mit Artikeln machen, wie das Beispiel zeigt:

```
#include <iostream>
#include <iomanip>
#include "queue.h"

// Definition einer neuen Klasse
class Artikel {
  friend ostream &operator<< (ostream &os, const Artikel &a) {
    os << a.Nr << ", " << a.Beschreibung
       << ", " << a.Preis << endl;
    return os;
  } // operator <<

  friend istream &operator>> (istream &is, Artikel &a) {
    is >> a.Nr;
    if (a.Nr == 0) return is;
    is >> a.Beschreibung;
    is >> a.Preis;
    return is;
  } // operator >>

  private:
    int Nr;
    char Beschreibung [20];
    double Preis;
  public:
    Artikel() { Nr = 0; Beschreibung[0] = '\0'; Preis = 0.0; }

    // Für den Vergleich mit 0 muss ein Artikel nach int
    // konvertierbar sein:
    operator int () { return Nr; }
}; // class Artikel
```

```
// Und schon können wir eine Artikelschlange bilden:
Queue<Artikel> ArtikelQueue1 (100); // max. 100 Artikel in der Schlange

template<class T> void einlesen (Queue<T> &q);
template<class T> void auslesen (Queue<T> &q);

int main () {
  einlesen (ArtikelQueue1);
  Queue<Artikel> ArtikelQueue2 (ArtikelQueue1); // Kopierkonstruktor
  cout << "\nInhalt der 1. Schlange:\n";
  auslesen (ArtikelQueue1);
  cout << "\nInhalt der Kopie der 1. Schlange:\n";
  auslesen (ArtikelQueue2);
  return 0;
} // main()

// Definition der Funktionen einlesen() und auslesen()
// wie gehabt als Funktions-Templates.
```

11.5 Weitere Template-Parameter

Klassentemplates können nicht nur einen einzigen Parameter haben, sondern durchaus auch mehrere. Die Parameter müssen keine Typnamen sein, sondern können durchaus normale Werte sein, auch Zeiger und Referenzen. Wenn wir unsere Queue ohne dynamischen Speicher realisieren möchten, können wir statt des Zeigers ein Feld verwenden. Die Größe der Schlange wird dann als Template-Parameter übergeben und nicht beim Anlegen des einzelnen Schlangen-Objekts.

Die Template-Parameter müssen Konstanten, Referenzen oder Zeiger sein. Bei Referenzen muss das referenzierte Objekt global oder statisch sein, bei Zeigern muss auf etwas Globales gezeigt werden.

Da keine dynamischen Elemente mehr vorhanden sind, erübrigen sich Destruktor, Kopierkonstruktor und Zuweisungsoperator, so dass die Klassendefinition kürzer wird. Das sieht dann so aus:

```
#ifndef _queue2_h
#define _queue2_h

#include <iostream>

template <class T, int n> class Queue {
  private:
    T vek [n];              // Feld mit n Elementen vom Typ T
    int tip, tail;
    bool empty;

  public:
    Queue () {              // default-Konstruktor
      tip = tail = 0; empty = true;
    }

    bool isEmpty() const { return empty; }
    bool enqueue (const T&); // stelle in Warteschlange
    T dequeue();             // entnehme aus Warteschlange
};
```

```
  template <class T, int n>
  bool Queue<T,n>::enqueue (const T &a) {
    if (tip != tail || empty) {  // falls noch Platz frei
      vek [tail] = a;
      tail = ++tail % n;
      empty = false;
      return true;
    }
    return false; // Es war kein Platz mehr!
  } // enqueue()


  template <class T, int n>
  T Queue<T, n>::dequeue() {
    if (empty) {
      cerr << "Queue leer!" << endl; exit (1);
      // kann später mit Ausnahmebehandlung verbessert werden
    }

    int i = tip;
    tip = ++tip % n;
    if (tip == tail) empty = true;
    return vek [i];
  } // dequeue()
  #endif
```

Zum Anlegen eines Objekts muss man jetzt beide Parameter angeben, beispielsweise `Queue <double, 100> myQueue`. Der Nachteil dieser Variante ist, dass für jede Größe von Schlangen der gesamte Code der Klasse neu erzeugt werden muss, weil sie jeweils mit der Konstanten `n` komplett neu generiert wird. Bei vielen verschiedenen Schlangengrößen (bei gleichem Grunddatentyp) wird dadurch der Code aufgebläht.

11.6 Defaultwerte für Template-Parameter

Auch bei Templates können die Parameter vorbelegt sein. Hierzu braucht man bloß die Defaultwerte hinter einem Gleichheitszeichen anzugeben. Möchten wir als default `int`-Queues mit 100 Elementen erzeugen, so schreiben wir:

```
  template <class T = int, int n = 100> class Queue {
  ...
};
```

Nun können wir folgende Schlangen deklarieren:

```
  Queue <char, 100> char100Queue;
  // Schlange mit 100 char

  Queue <double> double100Queue;
  // Schlange mit 100 double

  Queue <> int100Queue;    // leere spitze Klammern!!
  // Schlange mit 100 int
```

Man darf aber keinesfalls die leeren spitzen Klammern der leeren Template-Parameterliste vergessen, da es die Klasse `Queue` als solches nicht gibt.

Kapitel 12

Ausnahmebehandlung

Ein wesentlicher Fortschritt gegenüber früheren Versionen von C++ ist die Einführung der Ausnahmebehandlung. In sehr vielen Situationen können während der Laufzeit eines Programms Fehler auftreten. Diese müssen üblicherweise immer an Ort und Stelle behandelt werden, wodurch der Quelltext mit ständigen Abfragen auf mögliche Fehler gespickt ist. Das dient nicht gerade der Übersichtlichkeit, weil der eigentliche Ablauf dadurch oft verschleiert wird.

Die Lösung für dieses Problem ist die sogenannte Ausnahmebehandlung (exception handling). Man kann einen längeren Programmblock ausführen lassen und dahinter die Fehlerbehandlung in einem weiteren Block zusammenfassen. Immer wenn während des Programmblocks ein Fehler auftritt – eine Ausnahme ausgelöst wird (exception is thrown) –, wird zur Fehlerbehandlung gesprungen. Ist die Art der Ausnahme dort bekannt, kann sie behandelt werden. Andernfalls wird – sofern vorhanden – zum nächsten umschließenden Fehlerbehandlungsblock gesprungen. Oft ist dieser in einer rufenden Funktion oder auch erst in der `main()`-Funktion, so dass der Funktionsaufrufstack bis dahin abgewickelt wird. Wird eine Ausnahme gar nicht abgefangen, führt sie letztendlich zu einem Programmabbruch.

Ein Rücksprung an die Stelle, wo die Ausnahme ausgelöst wurde, ist auch nach erfolgter Ausnahmebehandlung nicht möglich, aber ggf. kann eine Schleifenkonstruktion dazu führen, dass der Block, in dem der Fehler passiert ist, erneut durchlaufen wird.

Die Ausnahmebehandlung ist im übrigen auch für Konstruktoren sehr wichtig, weil diese keinen Rückgabewert haben und daher auch nicht ohne weiteres auf korrekte Ausführung geprüft werden können.

Beim Schreiben von Bibliotheken wird man sicherlich Situationen erkennen können, an denen Fehler auftreten – beispielsweise Formatfehler beim Einlesen von Daten (Buchstaben, obwohl Ziffern erwartet wurden o. ä.). Als Autor der Bibliothek stellt man den Fehler dann zwar fest, kann aber nicht wissen, wie im konkreten Anwendungsprogramm darauf reagiert werden soll (Wiederholung der Eingabe, Ausgabe einer Warnung, Programmabbruch, Ignorieren des Fehlers nach Annahme des numerischen Wertes 0 oder noch anders). So ein overloadeter `>>`-Operator kann aber auch keinen Wert zurückliefern, der über die Fehlersituation informieren könnte. Auch das ist ein typischer Fall für die Anwendung von Ausnahmen. Ausnahmen transportieren sogenannte „out-of-band“-Informationen, d. h. solche, die nicht zur eigentlichen Verarbeitung gehören. Man könnte sie auch als Meta-Informationen bezeichnen.

Ausnahmen kommen sicher seltener vor als Funktionsaufrufe. Allerdings muss nicht jede Ausnahme irgendetwas Katastrophales sein, es ist nur eine weitere Möglichkeit der Kommunikation zwischen Komponenten eines Programms. Sehr oft handelt es sich aber um Fehler.

12.1 Fehlersituationen

Typische Fehlersituationen, in denen Ausnahmen ausgelöst werden können, sind:

- Fehler beim Dateizugriff
- Fehler bei `new` (kein Speicher mehr) und `delete` (Speicher war nicht mit `new` allokiert worden oder wurde bereits freigegeben)
- Überlauf, Unterlauf, z. B. Division durch 0

- Dereferenzierung eines ungültigen Zeigers

Ob in einer solchen Situation tatsächlich eine Ausnahme erzeugt wird, hängt von der Programmierung ab. Man kann eine Funktion so schreiben, dass sie in bestimmten Situationen eine Ausnahme erzeugt, denn das Erzeugen einer Ausnahme geschieht durch ein ganz normales Schlüsselwort: `throw`

12.2 Erzeugen einer Ausnahme

Ausnahmen werden erzeugt durch das Schlüsselwort `throw`, daher spricht man auch vom „Werfen einer Ausnahme“. Hinter dem Schlüsselwort ist anzugeben, was den geworfen werden soll; dabei handelt es sich um ein Objekt, das sogenannte Ausnahmeobjekt. Es ist der Träger der Information über die Ausnahme. Das Objekt kann von beliebigem Typ sein; üblicherweise legt man sich entsprechende Klassen oder Klassenhierarchien von Ausnahmeobjekten an.

Die Fehlerklasse kann Daten enthalten, oder auch nur eine leere Klasse (ohne Komponenten) sein. Die Daten im Objekt können aber weitere Information über die Fehlerursache enthalten. Gelegentlich wird auch eine Zeichenkette „geworfen“.

Das zu werfende Objekt muss allerdings statisch oder global sein, damit es nicht durch die Stack-Abwicklung bis zum „Fangen“ der Ausnahme, d. h. der Ausnahmebehandlung, zerstört wird.

Eine weitere Bedingung ist, dass die Klasse des Ausnahmeobjekts einen Kopierkonstruktor hat, weil die Objekte beim Auslösen kopiert werden.

12.3 Abfangen einer Ausnahme

Ausnahmen werden durch sogenannte `try-catch`-Blöcke abgefangen. Die grundsätzliche Konstruktion sieht so aus:

```
try {
  // Anweisungen, die ein throw enthalten können
  static fehler f1 ("Beispiel"); // fehler ist eine (Fehler-)Klasse
  static char[] mist = "Mist";

  if (...) throw mist;

  if (...) throw f1;

}
catch (char *s) {
  cerr << "Es ist der Fehler " << s << " aufgetreten.\n"
}
catch (fehler &f) {
  cerr << "Fehler aufgetreten: " << f << endl;
  // setzt voraus, dass für die Klasse fehler der Ausgabeoperator
  // definiert ist
}
catch (...) {     // catch mit Ellipse - passt auf alles!
  cerr << "nicht erwartete Ausnahme aufgetreten - Pech!" << endl;
  throw; // Ausnahme weiterwerfen
} // Ende der try/catch-Konstruktion
```

Voraussetzung für obiges Code-Fragment ist die Existenz einer Klasse `fehler`, für die ein Konstruktor mit einem `char*`-Parameter und der Ausgabeoperator `<<` existieren.

Der Parameter für die Ausnahme kann ein Werteparameter, eine (evtl. konstante) Referenz oder ein (evtl. konstanter) Zeiger sein. Bei Referenzen und Zeigern hat man den Vorteil, dass keine Kopie nötig

ist und dass der Aufruf von virtuellen Funktionen bei Ausnahme-Klassenhierarchien funktioniert (siehe Kapitel 9.3.4 auf Seite 59).

Im `try`-Block steht der übliche Code der Verarbeitung. Dem `try`-Block folgen ein oder mehrere `catch`-Blöcke, die jeweils eine Parameterliste (mit genau einem Parameter) wie eine Funktion haben. Die Reihenfolge der `catch`-Blöcke ist von Bedeutung, denn der Typ der Ausnahme wird in der Reihenfolge der Blöcke verglichen. Sobald einer passt, wird der Block betreten. Daher muss der `catch (...)`-Block immer als letzter auftreten. Bei den anderen ist die Reihenfolge dann wichtig, wenn verschiedene Ausnahmetypen aus derselben Klassenhierarchie stammen. Wenn es die Ausnahmeklassen `ak1` und `ak2` gibt und `ak2` von der Klasse `ak1` abgeleitet ist, dann würde eine Referenz oder ein Zeiger auf `ak1` auch auf ein Ausnahmeobjekt der Klasse `ak2` passen, so dass evtl. nicht der Block benutzt wird, der beabsichtigt wurde.

Hier ein Beispiel für eine kleine Klassenhierarchie von Ausnahmeobjekten. Die Klasse `MatheFehler` ist Basisklasse für die drei spezialisierteren Klassen. Beim Vergleich der Typen passt ggf. das Ausnahmeobjekt auf die Klasse `Ueberlauf`, wenn nicht, passt jedes Objekt aus diesen Klassen aber auf die Basisklasse. Hat eine Ausnahmeklasse mehrere Basisklassen, passt sie auf Handler aller ihrer Basisklassen.

```
class MatheFehler { };
class Ueberlauf    : public MatheFehler { };
class Unterlauf    : public MatheFehler { };
class NullDivision : public MatheFehler { };

try {
  // ...
}
catch (Ueberlauf) {     // speziellerer Fall
  cerr << "Überlauf aufgetreten\n";
}
catch (MatheFehler) {   // allgemeinerer Fall
  cerr << "MatheFehler aufgetreten, der kein Überlauf ist.\n";
}
```

Sollte während der Verarbeitung des `try`-Blockes keine Ausnahme auftauchen, so werden die `catch`-Blöcke ignoriert; das Programm wird hinter ihnen fortgesetzt.

Wird aber eine Ausnahme ausgelöst – sei es durch die hier sichtbaren `throw`-Anweisungen oder aber durch `throw`-Anweisungen in irgendwelchen aufgerufenen Funktionen, die dort nicht abgefangen werden –, so wird der Typ des geworfenen Objekts mit den Parametern der `catch`-Blöcke verglichen. Ist der Typ passend, so wird der `catch`-Block betreten, so dass die Ausnahmesituation behandelt werden kann.

Das Programm wird nach der Fehlerbehandlung fortgesetzt hinter dem letzten `catch`-Block – es sei denn, innerhalb der Fehlerbehandlung wird eine neue Ausnahme ausgelöst (siehe Kapitel 12.3.5 auf der nächsten Seite) oder das Programm beendet.

12.3.1 Abfangen aller Ausnahmen

Mit einer Ellipse als Parameterliste `(...)` werden alle Ausnahmen im `try`-Block abgefangen. Allerdings hat man dann keine nähere Information über die Ausnahme zur Verfügung. Daher sollte man dies nur als letzten Ausweg in ein Programm einbauen. Durch ein `catch (exception &e)` kann man immerhin schon alle Ausnahmen abfangen, die zur Klassenhierarchie der Standard-Ausnahmen (siehe Kapitel 12.3.7 auf Seite 95) gehören. Wenn man die eigenen Ausnahmeklassen von `exception` ableitet, fallen alle schon darunter. Man sollte dann aber nicht vergessen, die Methode `what()` (siehe Kapitel 12.3.7 auf Seite 96) zu definieren.

12.3.2 Nicht abgefangene Ausnahmen

Wird eine Ausnahme gar nicht abgefangen, so wird die Funktion `terminate()` (aus dem Namensbereich `std`) aufgerufen, die das Programm durch Aufrufen von `abort()` beendet. Das gleiche geschieht auch,

wenn bei der Abwicklung des Stacks auf der Suche nach einem passenden `catch` durch einen Destruktor eine weitere Ausnahme ausgelöst wird.

Ob hierbei Destruktoren aufgerufen werden oder nicht, ist implementationsabhängig. Daher ist es oft günstig, im Hauptprogramm *jede* Ausnahme zu fangen, so dass der korrekte Aufruf aller Destruktoren sichergestellt ist.

12.3.3 Ausnahmen in Konstruktoren

Gerade für Konstruktoren ist der Mechanismus der Ausnahmen ideal zur Fehlerbehandlung, denn sie liefern keinen Wert zurück, der geprüft werden könnte. Man könnte lediglich im Objekt ein boolesches Datenelement vorsehen, dessen Wert man über eine Methode prüfen könnte. Aber hierdurch würde man sich ja darauf verlassen, dass im Anwendungscode Erzeugung und Fehlerbehandlung vermischt werden, was wir möglichst vermeiden sollten.

Daher ist es also besser, Ausnahmen zu erzeugen und zu werfen, falls ein Objekt nicht richtig initialisiert werden kann, z. B. weil nicht genügend Speicher da ist, eine Datei nicht geöffnet werden konnte oder ähnliches.

Falls eine Ausnahme durch einen Element-Initialisierer (siehe Kapitel 9.3.1 auf Seite 57) erzeugt wird, kann man diese innerhalb des Konstruktors abfangen, wenn man den Element-Initialisierer in den `try`-Block mit aufnimmt. Das sieht dann so aus:

```
Auto::Auto() try : Fahrzeug ("Auto") {
  ccmHubraum  = 1500;
  kWLeistung  = 50;
}
catch (...) {
  cerr << "Initialisierung nicht möglich\n" << endl;
  throw; // weiterwerfen der Ausnahme
}
```

Allerdings wird diese Konstruktion noch nicht von allen Compilern unterstützt. Der GNU C++ Compiler (Version 2.95) beachtet den `catch`-Zweig nicht und wirft die Ausnahme direkt an die Funktion, die den Konstruktor aufgerufen hat. Ein weiterer Fehler des GNU-Compilers ist, dass der Destruktor zu einem Konstruktor aufgerufen wird, der eine Ausnahme ausgelöst hat, obwohl dieser evtl. gar nicht den Speicher allokieren konnte, den der Destruktor dann wieder freigeben möchte. Das führt zu einer Ausnahme innerhalb der Ausnahmesituation und damit zu einem sofortigen Programmabbruch.

Retten kann man die Situation dadurch, dass man alle Zeiger auf dynamischen Speicher erst mit 0 initialisiert, bevor man versucht, Speicher zu allokieren. Geht das dann schief, bleibt der Zeiger ein Zeiger auf 0, so dass ein anschließendes `delete` des Zeigers kein Problem darstellt.

12.3.4 Ausnahmen in Destruktoren

Destruktoren werden bekanntlich immer dann aufgerufen, wenn ein Objekt seinen Gültigkeitsbereich verlässt. Dies kann in zwei Zusammenhängen geschehen: Erstens bei ganz normalem Verlassen des Gültigkeitsbereichs, zweitens beim Abwickeln des Stacks im Rahmen einer Ausnahmebehandlung.

Beim Programmieren kann man mit der Funktion `uncaught_exception()` feststellen, welcher der beiden Fälle gerade eingetreten ist, so dass der Destruktor ggf. eine weitere Ausnahme erzeugen kann oder nicht. Auch diese Funktion ist nicht in allen Compilern implementiert; der aktuelle GNU-Compiler (Version 2.95) kennt sie nicht.

Falls innerhalb einer Ausnahmesituation eine weitere Ausnahme auftritt – wie in einem Destruktor möglich –, wird das Programm sofort beendet.

12.3.5 Weitergabe von Ausnahmen

Oft kann eine Situation nicht völlig gehandhabt werden, so dass die Weitergabe der Ausnahme nach einer Behandlung auf lokaler Ebene notwendig ist. Hierzu kann man die Ausnahme (das Ausnahmeobjekt) an

umschließende `catch`-Blöcke weitergeben.

Hierzu ruft man einfach erneut `throw` mit dem Ausnahmeobjekt auf. Wird diese Ausnahme anschließend nicht erneut abgefangen, führt das wiederum zum Programmabbruch – wie immer bei ungefangenen Ausnahmen.

Aus einem default-Handler, d. h. einem `catch`-Block mit Ellipse, kann man die nicht näher bekannte Ausnahme mit einem einfachen `throw` weitergeben. Das funktioniert übrigens auch bei anderen Handlern.

12.3.6 Spezifikation von Ausnahmen

Einer Funktion sieht man im allgemeinen nicht an, ob sie evtl. eine Ausnahme wirft oder nicht – es sei denn, man durchsucht ihren Quellcode. Erstens ist dieser oft nicht verfügbar, zweitens ist das viel Arbeit und drittens wollten wir uns eigentlich um Implementationsdetails nicht kümmern.

Andererseits ist es schon wichtig zu wissen, ob man mit einer Ausnahme rechnen muss oder nicht. Daher kann – und sollte man – im Funktionskopf angeben, welche Ausnahmen unter Umständen erzeugt werden. Gibt es keine solche Angabe, so kann die Funktion alle Ausnahmen erzeugen. Wenn eine solche Angabe vorhanden ist und die Funktion eine andere Ausnahme erzeugt, führt das zu sofortigem Programmabbruch (über eine `bad_exception`).

```
void f1() throw (exc1, exc2) {
  // ...
}
void f2() throw() {
  // ...
}
void f3() {
  // ...
}
```

Die Funktion `f1()` erzeugt lediglich Ausnahmen vom Typ `exc1` und `exc2`, die Funktion `f2()` erzeugt garantiert keine Ausnahmen, während die Funktion `f3()` alle möglichen Ausnahmen erzeugen kann. Die Angaben der möglichen Ausnahmen müssen bei der Definition der Funktion wiederholt werden und exakt mit denen in der Deklaration übereinstimmen.

12.3.7 Standard-Ausnahmen

Der Compiler und die Standardbibliotheksroutinen erzeugen eine ganze Reihe von Ausnahmen, die genau spezifiziert sind. Hier eine kurze Beschreibung der wichtigsten davon.

Ausnahmen der Sprache:

Ausnahme	von	Header
`bad_alloc`	`new`	`<new>`
`bad_cast`	`dynamic_cast`	`<typeinfo>`
`bad_typeid`	`typeid`	`<typeinfo>`
`bad_exception`	`Ausnahme-Spezifikation`	`<exception>`

Nur die letzte ist in diesem Dokument beschrieben, siehe Kapitel 12.3.6.

Ausnahmen der Standardbibliothek:

Ausnahme	von	Header
`out_of_range`	`at()`	`<stdexcept>`
`invalid_argument`	`Bitset-Konstruktor`	`<stdexcept>`
`overflow_error`	`bitset<>::to_ulong`	`<stdexcept>`
`ios_base::failure`	`ios_base::clear()`	`<ios>`

Das Schöne an all diesen Ausnahmen ist, dass sie zu einer Hierarchie gehören und daher mit einem `catch (exception &e)` abfangbar sind. Alle diese Ausnahmen haben auch eine Methode `what()`, deren Ergebnis ein C-String (`char*`) ist, der einen über die Ursache der Ausnahme informiert. Mit folgender Konstruktion fängt man also alle Standard-Ausnahmen ab und zeigt die Fehlerursache an, bevor das Programm beendet wird.

```
int main () {
  try {
    // ... bisheriger Inhalt der Main-Funktion
  }
  catch (exception &e) {
    cerr << e.what() << endl;
    abort();
  }
  return 0;
} // main()
```

Beim aktuellen GNU C++ Compiler sind in `<stdexcept>` folgende Ausnahmeklassen definiert, d. h. dass lediglich `ios::failure` noch nicht implementiert ist:

Klasse	**abgeleitet von**
`logic_error`	`exception`
`domain_error`	`logic_error`
`invalid_argument`	`logic_error`
`length_error`	`logic_error`
`out_of_range`	`logic_error`
`runtime_error`	`exception`
`range_error`	`runtime_error`
`overflow_error`	`runtime_error`
`underflow_error`	`runtime_error`

Dies stellt, neben den direkt von `exception` abgeleiteten Klassen `bad_alloc`, `bad_exception`, `bad_typeid` und `bad_cast` die komplette Klassenhierarchie der Ausnahmen dar.

12.4 Beispielprogramm mit Ausnahmen

In diesem kleinen Beispielprogramm wird eine spezielle Klasse namens `fehler` angelegt, deren Objekte nur dazu dienen, als Ausnahme erzeugt zu werden. Sie enthält nur einen Konstruktor mit einem Parameter vom Typ `char*`, so dass man immer eine Zeichenkette – als Fehlerbeschreibung – mitgeben muss.

Damit man das Fehlerobjekt direkt ausgeben kann, wird der Ausgabeoperator overloadet.

In Abhängigkeit von der Eingabe wird eine Ausnahme vom Typ `char*`, `fehler` oder `int` ausgelöst. Da die `int`-Ausnahme nicht explizit gefangen wird, passt sie auf die Ellipse (`...`). Würde der Eingabeoperator eine Ausnahme auslösen, würde diese ebenfalls durch das letzte `catch` abgefangen.

```
// ~bibjah/FORALL/CPP/exc_bsp1.cpp
#include <iostream>
#include <cstring>    // <string.h> C-Stringfunktionen

class fehler {
  friend ostream & operator<<(ostreamq &os, fehler &f) {
    os << f.text;
    return os;
  }
public:
```

```
  fehler (char *s) {
    strncpy (text, s, sizeof (text));
    text [sizeof (text) - 1] = '\0';
  }
private:
  char text [100];
};

char zk[] = "char*-Ausnahme";

int main() {
  try {
    char c;
    cin >> c;
    cout << c << endl;
    switch (c) {
      case 'f': {
        cout << "Werfe fehler-Objekt" << endl;
        throw fehler ("Beispielfehlerobjekt");
      }
      case 'c': {
        cout << "werfe char* Ausnahme ..." << endl;
        throw zk;
      }
      case 'i': {
        cout << "werfe int Ausnahme ..." << endl;
        throw 42;
      }
    } // switch
    cout << "keine Ausnahme ausgelöst" << endl;
  }
  catch (fehler f) {
    cerr << "Fehler-Objekt gefangen\n"
         << f << endl;
  }
  catch (char *s) {
    cerr << "Fehler " << s << endl;
  }
  catch (...) {
    cerr << "unbekannte Ausnahme" << endl;
  }

  cout << "Programmende!\n";
  return 0;
} // main()
```

12.5 Erweiterung der `vector`-Klasse

In der Standardbibliothek (schon in der älteren STL) gibt es eine Klasse `vector` in der Headerdatei `<vector>` (bzw. `<vector.h>`). Dort sind Arrays beliebiger Größe definiert, d. h. man kann die Arraygröße zur Laufzeit angeben, was sonst nur über eigene Verwaltung mit dynamischem Speicher möglich ist. Hier aber erhält man einen typsicheren Vektor, der zur Laufzeit in der Größe veränderbar ist – nicht nur bei der Erzeugung, sondern jederzeit.

Leider prüft der Operator `[]` aber nicht zur Laufzeit, ob er auf ein gültiges Element des Vektors zugreift, so dass im Fehlerfall ein simpler „segmentation fault“ (Speicherzugriffsfehler) auftritt, den man nicht mit `catch` abfangen kann.

Also liegt es nahe, von der an sich schon ganz guten Klasse `vector` eine verbesserte Version abzuleiten. Natürlich muss es sich wieder um eine Template-Klasse handeln, damit die Array-Elemente von beliebigem Typ sein können.

```
#include <stdexcept>
#include <vector>

template <class T> class Vec: public vector<T> {
public:
  Vec(): vector<T>() {}
  Vec (int s) : vector<T> (s) {}

  T& operator[] (int i) throw (bad_alloc, out_of_range) {
    if (i < 0) {
      throw out_of_range ("out_of range: Vec[i], negative index");
    }
    if (i > size() - 1) {
      resize (i+1);
    }
    return vector<T>::operator[] (i);
  }

  const T& operator [] (int i) const throw (out_of_range) {
    if (i < 0) {
      throw out_of_range ("out_of range: Vec[i], negative index");
    }
    if (i >= size()) {
      throw out_of_range ("out_of_range: Vec [i], index too big");
    }
    return vector<T>::operator[] (i);
  }
}; // template class Vec
```

Beim Zugriff auf (noch) nicht existierende Elemente, was mit der Methode `size()` geprüft wird, wird der Vektor passend vergrößert, sofern er als Referenz zur Verfügung steht. Bei negativen Indexen und bei Vektoren, die nur als konstante Referenz zur Verfügung stehen, kann man aber nur eine Ausnahme auslösen. Es wird die Standard-Ausnahme `out_of_range` hierfür verwendet, deren Konstruktor ein beschreibender Text übergeben werden muss.

Wenn ein Vektor langsam, aber stetig wächst, ist ein manuelles `resize()` zu empfehlen, damit es nicht so oft ausgeführt werden muss. Man kann auch Vektoren immer ein bisschen größer als gerade nötig anlegen, indem man das `resize (i+1)` ersetzt durch ein `resize (i+11)` oder ähnlich.

Kapitel 13

Namensbereiche

Namensbereiche (name spaces), auch Namensräume genannt, dienen dazu, die Wahrscheinlichkeit von Namenskollisionen (name clashes) zu verringern, indem man neben den Gültigkeitsbereichen global, lokal und klassenweit noch zusätzlich den Gültigkeitsbereich Namensbereich einführt.

13.1 Deklaration eines Namensbereichs

Der Namensbereich `beispielspace` wird deklariert durch:

```
namespace beispielspace {
  double f (double);
}
```

Im Namensbereich ist jetzt eine Funktion `f()` deklariert, deren Namen nicht mit einer globalen Funktion `f()` kollidiert, denn sie heiß vollständig `beispielspace::f()`. So muss sie bei ihrer Definition aus genannt werden:

```
double beispielspace::f (double x);
   return sqrt (x);
}
```

13.2 Using-Deklaration und -Direktive

Aus Namensbereichen kann man Bezeichner durch Voranstellen des Scope-Operators verwenden, man importiert einzelne Namen aus dem Namensbereich mit `using beispielspace::f` oder aber man „importiert“ den ganzen Namensbereich mit `using namespace beispielspace;`.

Der erste Fall ist praktisch, wenn man eine Funktion aus einem Namensbereich innerhalb eines lokalen Gültigkeitsbereichs öfters aufrufen muss und sich die Schreibarbeit des qualifizierten Aufrufs mit Namen und Scope-Operator sparen möchte. Außerhalb von Funktionen ist die Verwendung unsicher, denn es könnte Namenskollisionen geben. Allerdings werden sie zur Compilationszeit festgestellt, sind also nicht problematisch.

Die zweite Version, das Importieren eines kompletten Namensbereichs, ist prinzipiell eine Übergangslösung, weil Namensbereiche noch nicht von allen Compilern unterstützt werden und weil viel Code für ältere Versionen von C++ geschrieben wurden, die auch noch keine Namensbereiche kannten. Daher braucht man das `using namespace std;` beim GNU C++ Compiler auch nicht zu schreiben, weil alter Code dies nicht enthält, aber ohne Änderung compilierbar bleiben soll.

Übrigens braucht man den Namen einer Funktion nicht zu qualifizieren, wenn sie innerhalb einer anderen Funktion aufgerufen wird, deren Parameter aus demselben Namensbereich stammen. Das kann sehr praktisch sein:

```
namespace  Zeit {
  class Datum { /* ... */ };
    string format (const Datum&);
    // ...
  };
}

void f (Zeit::Datum d) {
  string s = format (d);
  // ...
} // f()
```

Auch wenn im Kontext von `f()` keine Funktion `format()` vorhanden ist, so wird sie dennoch gefunden, denn im Namensbereich des Parameters gibt es eine passende Funktion. Dies ist insbesondere bei Templates sehr hilfreich, wo die Qualifizierung sehr schwierig würde.

13.3 Aliasnamen für Namensbereiche

Verwendet man einen Namensbereich oft, so ist es praktischer, wenn der Name kurz ist. Andererseits ist das Risiko von Namenskollisionen bei sehr kurzen Namen hoch. Beim Entwurf eines Namensbereichs sollte man also lange und beschreibende Namen verwenden. Der Anwender kann sich leicht einen kurzen Aliasnamen erzeugen, für dessen Kollisionsfreiheit er die Verantwortung in seinem Programm trägt.

```
namespace kurz = langer_und_eindeutiger_Name;
```

Schon kann man mit `kurz::myfunction()` leichter qualifizieren.

13.4 Unbenannte Namensbereiche

Oft möchte man Namenskollisionen verhindern, aber nicht ständig neue, garantiert eindeutige Namen erfinden. Namen von Namensbereichen können schließlich immer noch kollidieren. Daher wurden die unbenannten Namensbereiche erfunden:

```
namespace {
  void f () {
    // ...
  }
}
```

Die Funktion `f()` ist nun im Namensbereich „versteckt“, aber wie kann man sie überhaupt aufrufen, wo doch der Namensbereich keinen Namen trägt? Nun, tatsächlich ist bei einem unbenannten Namensbereich ein implizites `using namespace` enthalten. Dadurch, dass die Namen innerhalb des Namensbereichs aber nur in der aktuellen Übersetzungseinheit (also dieser Quelldatei) bekannt sind, sind Namenskollisionen mit anderen Quelldateien ausgeschlossen.