

Einführung in C++

B. Plannerer

8. Juli 2001

Vorwort

Dieses Skriptum . . .

. . . begleitet den gleichnamigen Einführungskurs und soll vornehmlich als Gedächtnisstütze und Beispielsammlung dienen. Es ist daher sicherlich nicht selbsterklärend und erst recht nicht vollständig. Geplant ist jedoch eine schrittweise Komplettierung und Aktualisierung dieser Unterlagen während der Gesamtlauzeit des Kurses.

Fehler

Da das Skriptum in seiner ersten Auflage sicherlich zahllose Fehler und Ungeheimheiten enthalten wird, bin ich für jede Anregung und jede Fehlermeldung dankbar. Kritik, Kommentare und Beschimpfungen bitte per Email an:

`Plannerer@ieee.org`

Urheberrecht

Das Skriptum ist urheberrechtlich geschützt. Kein Teil dieses Skriptums darf ohne schriftliche Genehmigung des Autors übersetzt oder in irgendeiner Form durch Fotokopie oder andere Verfahren reproduziert oder in eine für Maschinen, insbesondere für Datenverarbeitungsanlagen, verwendbare Sprache übertragen werden. Ebenso sind die Rechte der Wiedergabe durch Vortrag vorbehalten.

©1997, 2001 Bernd Plannerer, München

Kapitel 1

Einleitung

1.1 Zielsetzung

Dieser Kurs ist gedacht für C-Programmierer, die bereits erste eigene Schritte mit C++ unternommen haben und daher zwar mit der erweiterten Syntax und den neuen Sprachkonstrukten von C++ grob vertraut sind, die sich jedoch noch etwas unsicher im Umgang mit den zahlreichen neuen Features von C++ fühlen und daher eine Zusammenstellung von Sonderfällen, Tips und Tricks suchen. Es wird weiterhin versucht, kritische oder fehlerträchtige Sprachkonstrukte genauer zu beleuchten. Diese eingeschränkte Zielsetzung hat zur Folge, daß der Kurs keinesfalls eine vollständige Sprachbeschreibung beinhalten oder gar die Lektüre der einschlägigen Fachliteratur ersetzen kann. Weiterhin beinhaltet dieser Kurs keine Einführung in die verschiedenen objektorientierten Analyse- und Designmethoden, zu denen sich zahlreiche gute (und weniger gute) Bücher finden lassen, da eine sinnvolle Behandlung dieser Thematik den Rahmen eines kompakten Einführungskurses sprengen würde.

1.2 Literatur

Aus der großen Vielfalt der Bücher, die zu C++ bereits erschienen sind, wurden zur Erstellung dieses Kurses einige brauchbare Vertreter herangezogen, welche in Kapitel 8 aufgelistet sind. An dieser Stelle sei vorab nur ein Buch besonders erwähnt, welches neben seiner verständlichen, umfassenden Darstellung auch durch seinen akzeptablen Preis besticht: *“C++ Grundlagen und Programmierung”* von Martin Hitz, erschienen im Springer-Verlag Wien New York, ISBN 3-211-82415-4.

1.3 Source-Code

Diejenigen Beispiele, die vollständige und daher übersetzbare Programmtexte darstellen, sind als Source-Code beim Autor erhältlich. Alle Beispiele wurden mit dem GNU C++ - Compiler übersetzt.

1.4 Kursaufbau

Nach einer kurzen Beschreibung einiger “neuen Features” von C++, die an sich noch nichts mit objektorientierter Programmierung zu tun haben, werden in Kapitel 3 Hinweise zur Implementierung der Konstruktoren und Destruktoren gegeben. Danach werden in Kapitel 4 Klassenmethoden und insbesondere Operatoren behandelt. Es schließen sich Hinweise zur Ein-Ausgabe mittels Streams in Kapitel 5 an. Kapitel 6 gibt eine kurze Übersicht über die Verwendung von Template-Klassen und Funktionen, während Kapitel 7 sich mit dem Exception-Handling auseinandersetzt.

Kapitel 2

Von C zu C++

2.1 Kommentare

In C++ Sourcen sollte man stets das C++-Kommentarzeichen (`//`) verwenden. Es leitet einen Kommentar ein, der bis zum Zeilenende reicht. Vorteilhaft ist hierbei, daß sich diese Kommentare verschachteln lassen, während dies bei den C-Kommentaren, die mit `/*` eingeleitet und mit `*/` abgeschlossen werden, nicht möglich ist. Dies zeigt nachfolgendes Beispiel:

```
...  
  
int i ;  
int j ;  
  
/*  
i++ ; /* add 1 to i */  
j++ ;  
*/  
...
```

Der Versuch, das Inkrementieren beider Variablen durch Auskommentieren zu verhindern, scheitert syntaktisch daran, dass der Kommentar bereits mit dem ersten schließenden Kommentarzeichen beendet wird. Unter Verwendung des Kommentarzeichens `//` kann der Code korrekt auskommentiert werden:

```
...  
  
int i ;  
int j ;  
  
//i++ ; // add 1 to i
```

```
//j++ ;
```

```
...
```

Die gemischte Verwendung von C - und C++ - Kommentaren ist wohl nur dann gerechtfertigt, wenn größere Abschnitte im Code *temporär* auskommentiert werden sollen:

```
...
```

```
int i ;  
int j ;
```

```
/*
```

```
//i++ ; // add 1 to i  
//j++ ;
```

```
... // several lines of code
```

```
*/
```

```
...
```

Hier sollte jedoch genau geprüft werden, ob nicht eine Präprozessoranweisung zur bedingten Übersetzung (`#ifdef ... #endif`) sinnvoller eingesetzt wäre.

2.2 Präprozessoranweisungen

Zur Konstantendefinition wird in C eine Makrodefinition verwendet:

```
#define DAYS 365
```

Hierbei wird durch die Präprozessoranweisung

```
#define identifier token-string
```

im Source-Code ein Textersatz von `DAYS` durch den nachfolgenden Text bis zum Zeilenende durchgeführt. Nachteilig hierbei ist, dass somit erstens keine Typsicherheit gegeben ist und zweitens kein symbolischer Name `DAYS` erzeugt wird, der z.B. von einem Debugger ausgewertet werden könnte. Eine geeignete Konstantendefinitionen in C++ würde hingegen durch die Definition einer konstanten Variable (sic!) unter Verwendung des Schlüsselwortes `const` erfolgen:

```
const int days = 365 ;
```

Somit ist `days` eine konstante Variable vom Datentyp `int`, deren Wert auch vom Debugger angezeigt werden kann. Hinweis: Konstanten müssen bei ihrer Definition immer auch initialisiert werden! Wird der Typname bei der Definition weggelassen, so wird implizit der Datentyp `int` angenommen:

```
const days = 365 ; // days ist vom Typ int
```

Weniger häufig anzutreffen ist die zweite Form der Makrodefinition, die “parametrisierten (*function-like*)” Makros:

```
#define identifizier( identifizier,...,identifizier ) token-string
```

Hierbei werden die in den runden Klammern aufgelisteten formalen Parameter im nachfolgenden Text expandiert, wie etwa im folgenden Beispiel der Funktion `CUB()`:

```
#define CUB(a) (a)*(a)*(a)
```

Hinweis: Die öffnende runde Klammer muß unmittelbar (also ohne Leerzeichen) auf den Makronamen folgen, ansonsten wird der Rest der Zeile zum Textersatz wie im einfachen, nicht parametrisierten Makro verwendet. Die Verwendung von parametrisierten Makros hat den Hauptnachteil, dass ein formaler Parameter, der in der Makrodefinition mehrfach benutzt wird (wie `a` in unserem Beispiel), auch mehrfach durch sein aktuelles Argument ersetzt wird:

```
//mac1.cc

#include <iostream.h>

#define CUB(a) (a)*(a)*(a)
int main ()
{
  int i = 2 ;

  cout << "i: " << i << endl ;
  cout << "i hoch drei: " << CUB(i) << endl ;           // OK
  cout << "(i++) hoch drei: " << CUB(i++) << endl ;     // oops !
  cout << "neues i: " << i << endl ;
}
```

Die Ausgabe dieses Programmes liefert:

```
i: 2
i hoch drei: 8
(i++) hoch drei: 8
neues i: 5
```

Diese Art von Seiteneffekten können in C++ vermieden werden, wenn statt einer Makrodefinition eine entsprechende inline-Funktion definiert wird:

```
//mac2.cc
#include <iostream.h>

inline int cub (int i)
{
return(i*i*i);
}

int main ()
{
int i = 2 ;

cout << "i: " << i << endl ;
cout << "i hoch drei: " << cub(i) << endl ;           // OK
cout << "(i++) hoch drei: " << cub(i++) << endl ;
cout << "neues i: " << i << endl ;
}
```

Dieses Programm liefert nun die Ausgabe:

```
i: 2
i hoch drei: 8
(i++) hoch drei: 8
neues i: 3
```

Anzumerken bleibt noch, dass diese Variante von `cub()` nur noch für den Datentyp `int` definiert ist, während das Makro `CUB` z.B. auch für `double`-Werte funktioniert. Durch den in Kapitel 6.2 behandelten Template-Mechanismus kann diese Allgemeingültigkeit jedoch wieder erreicht werden.

2.3 Aufzählungstypen

Eine weitere Möglichkeit zur Definition von Konstanten ist die Verwendung des Aufzählungstyps (`enum`) wie etwa in folgendem Beispiel:

```
enum status {ok, notOk, invalid} ;
```

Hierbei wird implizit der ersten Konstanten der Wert 0 zugewiesen, die weiteren Konstanten erhalten Werte in aufsteigender Reihenfolge, sodass beispielsweise `invalid` den Wert 2 erhält. Es kann jedoch auch eine explizite Wertzuweisung erfolgen:

```
enum salary {peanuts = 10, acceptable = 100,
             good, super = 1000, bad = 1, stillBad} ;
```

Die explizite Wertezuweisung darf also Lücken besitzen und sie muss nicht in aufsteigender Reihenfolge erfolgen. So erhält `good` den Wert 101 und `stillBad` den Wert 2. Mit `salary` wurde ein eigener Datentyp erzeugt, der wie alle Aufzählungstypen implizit in einen `int` konvertiert werden kann:

```
int gehalt = acceptable ;
```

Die Konversion vom `int` zu `salary` ist jedoch nicht implizit:

```
salary myGehalt = peanuts ;    // OK, same types
int gehalt = super ;          // OK, implicit conversion
myGehalt = gehalt ;           // oops, no implicit conversion exists !
```

Neben den Aufzählungstypen, die einen Typnamen (wie z.B. `salary`) besitzen, gibt es auch *anonyme* Aufzählungstypen, die lediglich der Definition der Konstanten dienen:

```
enum {smallBuffSize = 100, largeBuffSize = 1000} ;
```

Somit können die Konstanten `smallBuffSize` und `largeBuffSize` im Programmtext zusammen mit der impliziten Typumwandlung benutzt werden:

```
int buffer[smallBuffSize] ;
```

Die anonymen Aufzählungstypen werden im Zusammenhang mit konstanten Klassenkomponenten noch näher betrachtet werden.

2.4 Sichtbarkeit und Speicherklassen

2.4.1 Local Scope

Wird ein Objekt innerhalb eines Blocks deklariert, so ist es bis zum Ende des Blocks bekannt. Sofern innerhalb eines inneren Blocks eine weitere Deklaration erfolgt, überlagert diese die Deklaration des äußeren Blocks:

```
// loc.cc

#include <iostream.h>

int main ()
{
int i = 4 ;

    { //block
int i = 2 ; // i is local
i++ ;
cout << "block: i: " << i << endl ;
    }
```

```

        }//block

cout << "main: i: " << i << endl ;
}

```

liefert die Ausgabe:

```

block: i: 3
main: i: 4

```

Dieses “Feature” von C++ sollte jedoch nur sparsam eingesetzt werden, da hierdurch einerseits die Sourcen schwer verständlich werden, besonders aber die Wartbarkeit sinkt, da beispielsweise die nachträgliche Entfernung der inneren Deklaration von *i* keine Fehlermeldung des Compilers erzeugt und (eventuell unbemerkt) das Programmverhalten verändert. Die Entfernung der inneren Deklaration liefert folgende Ausgabe:

```

block: i: 5
main: i: 5

```

2.4.2 Global Scope

Wird ein Objekt außerhalb jedes Blocks deklariert, so ist dieses bis zum Ende der Quelldatei bekannt (*global*). Wird diese Deklaration durch eine Deklaration eines inneren Blocks überlagert, so kann auf das globale Objekt durch Verwendung des “*scope resolution operators*” (*::*) zugegriffen werden:

```

// glob.cc

#include <iostream.h>

int i = 6 ; // global
int main ()
{
int i = 4 ; // local in main

    { //block
int i = 2 ; // i is local in block
::i++ ; // scope resolution
cout << "block: i: " << i << endl ;
} //block

cout << "main: i: " << i << endl ;
cout << "global: i: " << ::i << endl ;
}

```

liefert:

```
block: i: 2  
main: i: 4  
global: i: 7
```

Auch hier gilt wieder wie im obigen Abschnitt, dass derlei Konstruktionen nicht gerade die Wartbarkeit der Software fördern.

2.4.3 Die Speicherklassen `auto` und `static`

Speicherklasse `auto`

Wird ein Objekt innerhalb eines Blocks ohne weitere Angaben zur Speicherklasse deklariert, so ist seine Speicherklasse `auto`. Das Schlüsselwort `auto` ist daher redundant und wird in der Regel nicht benutzt. Automatische Variablen werden mit dem Betreten des Blocks, in dem sie definiert sind, automatisch erzeugt und ihre Lebensdauer reicht bis zum Verlassen des Blocks. Der Initialisierungswert einer automatischen Variable ist *zufällig*, sofern sie nicht bei ihrer Definition explizit initialisiert wird. Hinweis: mit der Speicherklassenbezeichnung "*register*" werden ebenfalls automatische Variablen deklariert.

Speicherklasse `static`

Soll die Lebensdauer eines Objektes über das Verlassen des Blocks, in dem es definiert ist, bis hinaus zum Programmende verlängert werden, so kann dies erreicht werden, indem man es der Speicherklasse `static` zuordnet:

```
static int i = 42 ;
```

Erfolgt bei der Definition eine Initialisierung des Objekts, so findet diese *vor* Betreten der Funktion `main()` bzw. vor dem ersten Betreten des Sichtbarkeitsbereiches statt. Statische Variablen ohne explizite Initialisierung werden stets mit dem Wert Null initialisiert. Eine besonders bekannte Anwendung statischer Variablen ist es, mitzuzählen, wie oft eine bestimmte Funktion aufgerufen wird:

```
//stat.cc  
  
#include <iostream.h>  
  
void count ()  
{  
    static int counter = 1 ;  
    cout << "called " << counter << " times." << endl ;  
    counter++ ;  
}  
  
int main ()
```

```

{
const int maxCalls = 5 ;

for (int i = 0 ; i < maxCalls ; i++)
    {
        count();
    }//for i
}

```

Das Programm liefert die Ausgabe:

```

called 1 times.
called 2 times.
called 3 times.
called 4 times.
called 5 times.

```

Die Lebensdauer von Objekten, die außerhalb jedes Blocks definiert wurden (also *global* sind), erlischt erst mit dem Programmende. Sie gehören also ebenfalls zur Speicherklasse **static**. Wird jedoch für diese Objekte zusätzlich das Schlüsselwort **static** explizit angegeben, so bewirkt dies, dass das Objekt nicht exportiert wird, also nicht von getrennt übersetzten Programmteilen aus erreicht werden kann. Wird in diesem Source-File versucht, danach noch denselben Objektnamen **extern** zu deklarieren (z.B. um ein gleichnamiges Objekt aus einem anderen Programmteil zu referenzieren), so gelingt dies nicht: Es wird dennoch das gleichnamige Objekt aus demselben File referenziert, wobei dieses auch nicht durch die "nachträgliche" **extern** - Deklaration exportiert wird. Würde man jedoch versuchen, ein bereits als **extern** deklariertes Objekt danach noch als **static** zu definieren, würde dies zu einer Fehlermeldung des Compilers führen.

Globale Konstanten

Einen weiteren Sonderfall stellen *globale Konstanten* dar: sie sind *implizit static*, werden also nicht exportiert. Möchte man von einem getrennt übersetzten Programmteil aus auf die globale Konstante zugreifen, so muss sie nicht nur bei der Deklaration, sondern auch bei der *Definition* als **extern** gekennzeichnet werden:

```

//file_a:
...
extern const int days = 365 ; // definition, export days
...

//file_b:
extern const int days;          //declaration, import days from file_a
...

```

Üblicher ist es jedoch, solche Konstanten in einem entsprechenden Header-File zu definieren, sodass in *beiden* Source-Files, die jeweils das Header-File inkludieren, die Konstanten*definition* (implizit `static`) erfolgt.

2.5 Pointer und Referenzen

2.5.1 Pointer

Die Benutzung von Pointern in C++ unterscheidet sich nicht grundlegend von der in C, jedoch gibt es einige Erweiterungen, die nachfolgend betrachtet werden. Pointer können in C++ als Pointer auf `const` Objekte definiert werden. Damit kann erreicht werden, dass das Objekt, auf das der Pointer zeigt, nicht über diesen Pointer verändert werden kann:

```
int i = 42 ;
const int *pi = &i ;
*pi = 0 ;           //not allowed
```

Will man verhindern, dass auch der Pointer selbst verändert werden kann, so wird der Pointer ebenfalls als `const` definiert:

```
const int * const pi = &i ;
```

Ein spezieller `const` Pointer ist der Pointer auf die (ungültige) Adresse 0, der in C durch die Konstante `NULL` gekennzeichnet wurde. In C++ existiert zwar `NULL` noch aus Kompatibilitätsgründen, jedoch darf vereinbarungsgemäß auch direkt der Wert 0 verwendet werden.

Für `void` Pointer gilt in C++: Jeder Pointer kann implizit auf einen `void *` umgewandelt werden, jedoch kann im Gegensatz zu C kein `void *` implizit in einen Pointer auf einen anderen Typ umgewandelt werden.

2.5.2 Referenzen

Referenzen können als alias-Namen für Objekte betrachtet werden. Nach ihrer Definition kann der Name der Referenz auf ein Objekt ebenso verwendet werden wie der Name des Objektes selbst:

```
//ref.cc

#include <iostream.h>

int main()
{
    int i = 0 ;
    int & ref = i ;
    i++ ;
    ref++ ;
}
```

```

        cout << "i: " << i << " ref: " << ref << endl ;
    }

```

Das Programm liefert die Ausgabe:

```
i: 2 ref: 2
```

Eine weitere Interpretationsmöglichkeit für Referenzen ist, sich eine Referenz als konstanten Pointer vorzustellen, der in der Verwendung stets automatisch dereferenziert wird. Aus dieser Interpretation erklärt sich auch, warum eine Referenz stets bei ihrer Definition auch initialisiert werden muss (siehe obiges Beispiel). Diese Initialisierung erfolgt jedoch manchmal auch implizit, so etwa wenn eine Referenz als Parameter einer Funktionsdeklaration auftritt oder auch wenn eine Funktion eine Referenz als Ergebnis zurückliefert. In beiden Fällen erfolgt dann die Initialisierung der Referenz automatisch. Eine besondere Bedeutung haben Referenzen in Verbindung mit dem Schlüsselwort `const`. Durch die Definition einer Referenz auf ein konstantes Objekt kann ein Objekt vor dem Überschreiben geschützt werden:

```

int i = 42 ;
const int &ref = i ;
...
ref = 5 ;           //error ! read-only

```

`ref` ist somit ein “read-only”-alias für `i`. Die wichtigste Bedeutung haben Referenzen jedoch im Zusammenhang mit Funktionsparametern und Rückgabewerten, wie in den Abschnitten 2.6.3 und 2.6.4 gezeigt wird.

2.6 Funktionen

2.6.1 Default-Argumente

In C++ kann man für Parameter default-Werte vereinbaren, die eingesetzt werden, falls das zugehörige Argument beim Funktionsaufruf fehlt. Hierbei müssen die Parameter mit default-Werten stets *am Ende* der Parameterliste stehen:

```

enum units {kmh, mph} ;
...
void printSpeed(int speed, units unit = kmh) ;

```

Unzulässig wäre aber die Deklaration von:

```
printSpeed(units unit = kmh, int speed) ;
```

Mögliche Aufrufe von `printSpeed()` sind:

```

printSpeed(50) ;           //prints speed with unit kmh
printSpeed(50, mph) ;     //prints speed with unit mph

```

2.6.2 Überladen von Funktionen

Eine Funktion wird in C++ nicht nur durch ihren Funktionsnamen, sondern auch zusätzlich durch ihre Parameterliste eindeutig indentifiziert. Daher kann ein Funktionsname auch mehrfach benutzt werden, sofern sich die Parameterlisten der Funktionen voneinander unterscheiden. Die Funktionen

```
print(int i) ;
```

und

```
print(double i) ;
```

sind also zwei voneinander verschiedene Funktionen (die jedoch hoffentlich dieselbe Semantik besitzen!).

Hinweise:

- Funktionen werden nicht anhand des mit `return` zurückgegebenen Datentyps unterschieden!
- Wenn sich zwei Funktionen nur dadurch unterscheiden, dass ein Argumenttyp `T` einmal direkt (per value) und einmal als Referenz `T&` übergeben werden, sind die Funktionen nicht unterscheidbar!
- Dies gilt auch, wenn in der einen Funktion ein Typ `T` und in der anderen Funktion `const T` übergeben wird, es sei denn, die Argumente werden durch Pointer (`T *` bzw. `const T *`) oder mittels Referenzen (`T&` bzw. `const T&`) übergeben!

Vorsicht ist geboten, wenn Funktionen nicht nur überladen, sondern auch noch mit default-Werten versehen werden: Die Funktionen

```
print(int) ;
```

und

```
print(int, int base = 10);
```

sind beim Aufruf

```
print(42) ;
```

nicht unterscheidbar.

Bei der Wahl zwischen der Verwendung von default-Argumenten und überladenen Funktionen sollte unterschieden werden, ob wirklich verschiedene Algorithmen zum Einsatz kommen, oder ob lediglich ein Parameter auf einen sinnvoll gewählten default-Wert gesetzt werden soll. Ist dies der Fall, dann sollte besser mit default-Argumenten gearbeitet werden.

2.6.3 Referenzen als Funktionsparameter

Neben den in C üblichen Möglichkeiten, ein Objekt an eine Funktion zu übergeben (entweder als value-Parameter oder durch Übergabe eines Pointers auf das Objekt) kann in C++ auch eine Referenz auf das Objekt übergeben werden:

```
//refpar.cc

#include <iostream.h>

void f (int & a)
{
    a++ ;
}

int main ()
{
    int i = 0 ;
    f(i) ;
    cout << "i: " << i << endl ;
}
```

Hierdurch wird die Variable `i` in `f()` verändert, was jedoch auf den ersten Blick nicht offensichtlich ist. Generell gilt, dass, wenn eine Funktion das Argument verändern soll, tunlichst ein Pointer auf das Objekt übergeben werden sollte. Dann ist sichergestellt, dass die Veränderung des Arguments auch im Funktionsaufruf sichtbar wird. Soll die Funktion das übergebene Objekt nicht verändern, so kann es im Prinzip als value-Parameter übergeben werden. Dann wird zur Laufzeit eine Kopie des Objektes erstellt. Bei größeren Objekten jedoch ist dies eventuell recht zeitaufwendig, sodass hier eine Referenz auf ein konstantes Objekt übergeben werden sollte:

```
//crefpar.cc

#include <iostream.h>

void f (const int & a)
{
    cout << "value: " << a << endl ;
}

int main ()
{
    int i = 0 ;
    f(i) ;
    cout << "i: " << i << endl ;
}
```

Da die Referenz ein readonly-alias des übergebenen Objektes darstellt, wird zur Laufzeit keine Kopie des Objektes erzeugt. Ein irrtümlicher Versuch, das Objekt in der Funktion zu verändern, würde bereits vom Compiler erkannt, da ja eine Referenz auf eine `const` Objekt vereinbart wurde.

2.6.4 Referenzen als Ergebniswerte

Eine Funktion kann auch eine Referenz auf ein Objekt als Ergebnis zurückgeben. Hierbei gilt es jedoch zu beachten, dass das Objekt, auf das sich die Referenz bezieht, auch nach dem Verlassen der Funktion noch existieren muss. Nachfolgendes Beispiel ist also falsch:

```
int & f ()
{
int i = 42 ;
return i ;    //absolutely wrong !
}
```

Es sei an dieser Stelle davor gewarnt, das Objekt, auf das sich die Referenz bezieht, mit `new` zu erzeugen und dann eine Referenz auf das erzeugte Objekt zurückzugeben. Da die Tatsache, dass das Objekt mit `new` erzeugt wurde, aus dem Aufruf der Funktion nicht hervorgeht, sind Speicherlöcher bei solchen Konstruktionen im Wortsinn “vorprogrammiert” !

Kapitel 3

Konstruktoren und Destruktoren

In diesem Kapitel werden zunächst die Eigenschaften von Konstruktoren und Destruktoren besprochen. Dabei werden auch einige Sonderfälle bei der Konstruktion und Destruktion von Objekten betrachtet. Danach wird noch auf die Problematik des Copy-Konstruktors und des Assignment-Operators eingegangen.

3.1 Konstruktor

Konstruktoren sind spezielle Methoden, die kein Ergebnis zurückliefern, sondern die der Initialisierung von Objekten dienen und die bei der Instantiierung eines Objekts aufgerufen werden. Diese findet je nach Lebensdauer und Speicherklasse des Objekts zu unterschiedlichen Zeitpunkten statt:

- globale, statische Variablen werden vor dem Betreten von `main` erzeugt, ihr Konstruktor wird also vor dem Betreten des Funktionskörpers von `main` aufgerufen.
- funktionslokale, statische Variablen werden beim ersten Betreten der Funktion initialisiert, danach nicht mehr.
- lokale, automatische Variablen werden beim Betreten ihres Sichtbarkeitsbereiches initialisiert.

Bei der Initialisierung eines Objekts durch den Konstruktor erfolgt zunächst die Initialisierung der Daten-Member anhand der Initialisierungsliste, danach erfolgt die Ausführung der Anweisungen, die im Funktionskörper des Konstruktors stehen.

Für Member, die nicht über die Initialisierungsliste explizit initialisiert werden, wird automatisch der Default-Konstruktor aufgerufen.

Auch integrale Datentypen, die keinen Konstruktor besitzen, wie etwa der Typ `int`, können über die Initialisierungsliste initialisiert werden.

Die Reihenfolge der Initialisierung der Member ist stets die Reihenfolge der *Deklaration der Member*, nicht die Reihenfolge in der Initialisierungsliste, da es zu einer Klasse ja mehrere Konstrukturfunktionen mit eventuell verschiedenen Initialisierungslisten geben kann. Eindeutig ist nur die Deklarationsreihenfolge der Klassen-Member.

Es sollte zur Initialisierung von Membern kein Assignment im Funktionskörper verwendet werden, sondern direkt die Initialisierungsliste benutzt werden, denn ansonsten wird, wie oben beschrieben, für das Member zunächst der Default-Konstruktor aufgerufen und danach erst wird dem Daten-Member durch Aufruf des Assignment-Operators der gewünschte Wert zugewiesen. Dies verdeutlicht folgendes Beispiel:

```
//  
//init.cc  
  
#include <iostream.h>  
  
class A  
{  
private:  
int mValue ; // data member  
  
public:  
// default constructor  
A () :  
mValue(0)  
{cout << "A::A() called" << endl ;} ;  
  
// conversion constructor  
A (int i) :  
mValue(i)  
{cout << "A::A(int) called" << endl ;} ;  
  
// assignment operator  
class A & operator= (const A & r)  
{  
cout << "A::operator=(const A &) called" << endl ;  
mValue = r.mValue ;  
return *this ;  
} ;  
  
// destructor  
~A()  
{cout << "A::~~A() called" << endl ;} ;
```

```

} ; // class A

class B
{
private:
A mA ; // data member is class A

public:

// conversion constructor
B(int i) : mA(i) {cout << "B::B(int) called" << endl ;} ;

// destructor
~B () {cout << "B::~B() called" << endl ;} ;
} ; //class B

class C
{
private:
A mA ; // data member is class A

public:

// conversion constructor
C(int i)
{
cout << "C::C(int) called" << endl ;
mA = i ; // this also implies type conversion from int to A
} ;

// destructor
~C () {cout << "C::~C() called" << endl ;} ;
} ; //class B

int main ()
{
cout << "instantiate class B" << endl ;
B b(5) ; // construct B from int
cout << "instantiate class C" << endl ;
C c(42) ; // construct C from int
}

//

```

Das Programm liefert die Ausgabe:

```

instantiate class B
A::A(int) called
B::B(int) called
instantiate class C
A::A() called
C::C(int) called
A::A(int) called
A::operator=(const A &) called
A::~~A() called
C::~~C() called
A::~~A() called
B::~~B() called
A::~~A() called

```

Wie man sieht, wird gegenüber der Anwendung der direkten Initialisierung des Members `mA` in Klasse `B` bei der Zuweisung in Klasse `C` zusätzlich der Default-Konstruktor von `A` sowie der Zuweisungsoperator aufgerufen. Man beachte hierbei die implizite Typumwandlung von `int` auf `A` in der Zuweisung mittels des Konversions-Konstruktors `A::A(int)`.

Konstante Member können nur über die Initialisierungsliste initialisiert werden, da ihnen ja nachträglich im Funktionskörper des Konstruktors kein Wert mehr zugewiesen werden darf!

Da Referenzen konstante Pointer sind, müssen sie ebenfalls in der Initialisierungsliste initialisiert werden!

3.1.1 Statische Daten-Member

Wird ein Member in der Klassendefinition als `static` deklariert, so wird nur *eine einzige* Instanz dieses Member erzeugt, auf die alle Instanzen der Klasse zugreifen können. Dieses statische Member wird hierbei *nur als deklariert und nicht als definiert* angesehen. Die *Definition und Initialisierung* statischer Daten-Member darf nicht innerhalb des Konstruktors erfolgen, da der Konstruktor ja während der Programmlaufzeit *mehrfach* aufgerufen werden könnte, eine Initialisierung aber nur *einmal* erfolgen darf. Die Definition und Initialisierung statischer Daten-Member muss daher außerhalb des Konstruktors im Programmtext erfolgen. Dies gilt natürlich auch für *konstante* statische Daten-Member. Sie dürfen nicht bei ihrer Deklaration in der Klassendefinition initialisiert werden (denn dann wäre es ja eine Definition und statische Daten-Member werden ja innerhalb der Klassendefinition nur deklariert und nicht definiert), sondern sie müssen im Programmtext außerhalb der Klassendefinition initialisiert werden.

Dies führt zu folgendem Problem: Soll der Wert der Konstante im folgenden Teil der Klassendefinition benutzt werden, so ist dies mangels Initialisierung nicht möglich:

```

class Buffer
{

```

```
private:
    static const int bufsize = 1000 ;
    char mBuffer[bufsize];
    ...

} ; //class Buffer
```

Die Initialisierung von `bufsize` ist hier nicht zulässig, und ohne Initialisierung kann der Compiler die Definition `char mBuffer[bufsize];` mangels Größenangabe für `bufsize` nicht akzeptieren. Hier kann Abhilfe geschaffen werden, indem mit einem anonymen Aufzählungstyp gearbeitet wird, der ja standardgemäß nach `int` konvertiert werden kann:

```
class Buffer
{
private:
    enum {bufsize = 1000} ;
    char mBuffer[bufsize];
    ...
} ; //class Buffer
```

Für den anonymen Aufzählungstyp ist die Initialisierung innerhalb der Klassendefinition zulässig.

3.2 Default-Konstruktor

Eine besondere Form des Konstruktors ist der *Default-Konstruktor*, der keinen Parameter besitzt. Nur wenn zu einer Klasse überhaupt kein Konstruktor angegeben wird, erzeugt der Compiler einen (wirkungslosen) Default-Konstruktor. Ansonsten ist der Default-Konstruktor explizit zu definieren.

Der Default-Konstruktor wird typischerweise bei der Initialisierung von Feldern aufgerufen:

```
//
//defcon.cc
#include <iostream.h>

class A
{
private:
int mValue ; // data member

public:
// default constructor
A () :
mValue(0)
{cout << "A::A() called" << endl ;} ;
```

```

// destructor
~A()
{cout << "A::~A() called" << endl ;} ;

} ; // class A

int main ()
{
A classArray[4] ;
}

//

```

Das Programm liefert die Ausgabe:

```

A::A() called
A::A() called
A::A() called
A::A() called
A::~A() called
A::~A() called
A::~A() called
A::~A() called

```

Arrays können also nur für Klassen definiert werden, die auch einen Default-Konstruktor besitzen. Es empfiehlt sich also dringend, einen solchen zu definieren. Dies erpart dann oft den unschönen Umweg, zur Realisierung von Arrays von Objekten zunächst Arrays mit Pointern auf die Objekte einzusetzen und die Objekte dann mit `new` erzeugen zu müssen.

3.3 Destruktor

Der Destruktor dient dem definierten Zerstoren von Objekten, etwa wenn die Lebensdauer einer lokalen Variable mit dem Verlassen des Blocks beendet ist oder beim expliziten Freigeben eines mit `new` erzeugten Objekts durch den Aufruf von `delete` oder wenn ein vom Compiler erzeugtes temporäres Objekt nicht mehr benötigt wird. Der Destruktor muss insbesondere den Speicher für dynamisch erzeugte Member wieder freigeben um Speicherlöcher zu vermeiden.

Da es immer nur *einen* Destruktor, aber durchaus mehrere Konstruktoren geben kann, ist es nötig, dass im Destruktor alle denkbaren Zustände des Objekts korrekt behandelt werden. Generell empfiehlt es sich daher, Pointer auf dynamisch zu erzeugende Daten-Member in der Initialisierungsliste des Konstruktors mit `0` vorzubelegen, denn dann kann im Destruktor gefahrlos ein `delete` auf diesen Pointer durchgeführt werden.

Generell sei empfohlen, die Destruktoren von Basisklassen stets virtuell zu deklarieren, damit unter allen Umständen auch für abgeleitete Klassen der richtige Destruktor aufgerufen wird.

Wichtig ist, dass ein Array von Objekten, das mit `new []` angelegt wurde, auch stets mit `delete []` wieder freigegeben wird, da bei einem einfachen Aufruf von `delete` nur für das erste Objekt im Array der Destruktor aufgerufen würde!

Dies zeigt folgendes Beispiel:

```
//
//dest.cc
#include <iostream.h>

class A
{
private:
int mValue ; // data member

public:
// default constructor
A () :
mValue(0)
{cout << "A::A() called" << endl ;} ;

// destructor
~A()
{cout << "A::~~A() called" << endl ;} ;

} ; // class A

int main ()
{
cout << "allocate array1" << endl ;
A * array1 = new A[3] ; //allocate array1
cout << "allocate array2" << endl ;
A * array2 = new A[4] ; //allocate array2
cout << "delete [] array1" << endl ;
delete [] array1 ; //correct
cout << "delete array2 (wrong)" << endl ;
delete array2 ; //wrong !!!
}

//
```

Das Programm liefert die Ausgabe:

```
allocate array1
A::A() called
```

```

A::A() called
A::A() called
allocate array2
A::A() called
A::A() called
A::A() called
A::A() called
delete [] array1
A::~A() called
A::~A() called
A::~A() called
delete array2 (wrong)
A::~A() called

```

Für das Array `array2` wird wegen des fehlerhaften Aufrufes von `delete` nur der Speicher für das erste Element wieder freigegeben.

Sofern eine Klasse statische Daten-Member enthält (z.B. ein Pointer auf ein File) dürfen diese nicht durch den Destruktor “deinitialisiert” werden (z.B. Schließen des Files), da ansonsten der Wert des statischen Member bereits vom ersten Aufruf des Destruktors an ungültig wäre. Dieser erste Aufruf des Destruktors könnte aber – unbemerkt vom Programmierer – beispielsweise vom Compiler selbst vorgenommen werden, etwa dann, wenn ein vom Compiler erzeugtes temporäres Objekt wieder gelöscht wird ! Analog zur Initialisierung statischer Member muss daher auch ihre Deinitialisierung im Programmtext erfolgen und darf nicht im Destruktor vorgenommen werden !

3.4 Copy-Konstruktor

Der Copy-Konstruktor einer Klasse T ist insofern eine besondere Variante der Konstrukturfunktionen, dass er als Parameter eine Referenz auf ein (in der Regel konstantes) Objekt der gleichen Klasse T erhält. Die Signatur des Copy-Konstruktors ist also:

```
T::T (const T&)
```

Es wird normalerweise eine Referenz auf ein *konstantes* Objekt übergeben, da der Konstruktor ja nur eine Kopie des Objektes initialisieren und das “Original” unverändert lassen soll. Das Original muss freilich immer als *Referenz* übergeben werden, da die Übergabe als value-Parameter den endlos rekursiven Aufruf des Copy-Konstruktors bedeuten würde! Wird für eine Klasse kein Copy-Konstruktor definiert, so erzeugt der Compiler eine default-Variante, die jedes Member der Kopie mit dem entsprechenden Wert des Originals initialisiert. Die Initialisierung der Kopie erfolgt wie immer zunächst über die Initialisierungsliste und danach durch das Ausführen der Anweisungen im Funktionskörper. Besondere Beachtung verdienen jedoch *Pointer* als Member, wenn diese auf Speicherbereiche zeigen, die dynamisch erzeugt wurden. In solchen Fällen darf

der Wert des Pointers im Original nicht einfach in den entsprechenden Pointer in der Kopie übertragen werden, vielmehr muss der “darunter liegende” Speicherbereich explizit durch Aufruf von `new` allociert werden. Diese Methode, eine Kopie eines Objektes unter korrekter Berücksichtigung der im Objekt enthaltenen Verweise auf weitere Speicherbereiche zu erstellen, wird als “*tiefe* Kopie” (engl.: “*deep copy*”) bezeichnet, im Gegensatz zur einfachen, Member für Member erfolgenden “*flachen*” Kopie (engl.: “*shallow copy*”), wie sie vom Compiler automatisch generiert wird. In Fällen also, in denen eine Klasse Pointer auf dynamisch erzeugten Speicher besitzt, ist also der vom Compiler erzeugte default Copy-Konstruktor nicht funktionsfähig ! Da bei einer flachen Kopie in diesem Fall Original und Kopie *denselben* Speicherplatz verwenden würden, würde eine eigentlich als lokal gedachte Veränderung des Speicherbereiches durch die Kopie auch für das Original sichtbar werden, und wenn bei der Zerstörung eines der beiden Objekte im Destruktor korrekterweise der allocierte Speicher wieder mit *delete* freigegeben wird, so würde damit natürlich ungewollt auch der Speicherbereich des anderen Objekts gelöscht werden!

Sobald also eine Klasse auch nur einen Pointer auf dynamisch erzeugten Speicher enthält, muss der Copy-Konstruktor selbst implementiert werden. Möchte man aus bestimmten Gründen keine eigene Implementierung des Copy-Konstruktors angeben, so sollte dieser zumindest als private Methode der Klasse deklariert (nicht aber definiert) werden, damit ein unbeabsichtigter Aufruf des (ansonsten automatisch erzeugten) Copy-Konstruktors vom Compiler (bzw. für abgeleitete Klassen erst vom Linker) erkannt und somit vermieden wird:

```
class NoCopy
{
    ...
    private:
        NoCopy(const NoCopy &) ; //cannot be called by users:
                                //declared but not defined
}

```

3.5 Assignment-Operator

Verwandt mit dem Copy-Konstruktor ist der Zuweisungs- (assignment-) Operator, für den ebenfalls vom Compiler eine default-Variante erzeugt wird, sofern keine explizite Definition des Zuweisungsoperators angegeben wird. Auch beim default-Zuweisungsoperators findet die Zuweisung Member für Member statt. Die Signatur des Zuweisungsoperators lautet:

```
T & T::operator= (const T&)
```

3.5.1 Pointer auf dynamisch erzeugten Speicher

Völlig analog zum Copy-Konstruktor muss auch beim Zuweisungsoperator darauf geachtet werden, dass eventuell vorhandene Pointer auf dynamisch erzeugten

Speicher nicht einfach mit dem Wert des entsprechenden Members des zuzuweisenden Objektes überschrieben werden dürfen. Vielmehr ist gegebenenfalls der bereits allocierte Speicher, auf den der Pointer zeigt, freizugeben, neuer Speicher in der benötigten Größe zu allocieren und danach diesem Speicherbereich die Werte des korrespondierenden Speicherbereiches der rechten Seite zuzuweisen. Dies ist in nachfolgendem Beispiel gezeigt:

```
//
//string.cc
#include <iostream.h>
#include <string.h>

class String
{
private:
size_t mLength ;
char * mBuffer ;
String(const String &) ;
public:
String () : mLength(0), mBuffer(0)
{cout << "String::String() called." << endl ;} ;
String (const char * sz) ;
String & operator= (const String & rs) ;
void status ()
{cout << "length: " << mLength << ", Buffer: " << mBuffer << endl ;} ;
~String()
{ cout << "String::~~String() called." << endl ; delete [] mBuffer ;} ;
} ; //class String

//implementation part

// constructor
String::String (const char *sz)
: mLength (0),
mBuffer(0)
{
cout << "String::String(const char *) called." << endl ;
if( (mLength=strlen(sz)) == 0)
{
return ; // empty string
}

// get memory
mBuffer = new char[mLength+1];
if (mBuffer==0)
{
```

```

cerr << "out of memory." << endl ;
exit(0);
} //if
//copy String
strcpy (mBuffer,sz) ;
mBuffer[mLength] = '\0' ;
} //String::String

//assignment operator
String & String::operator= (const String & rs)
{
cout << "String::operator= called." << endl ;
//check for self assignment
if(this == &rs)
{
cout << "String::operator= : self assignment detected." << endl ;
return (*this) ;
}

if(mLength != rs.mLength)
{
//delete old memory
delete [] mBuffer ;
//get new memory ;
mBuffer = new char [rs.mLength+1] ;
if (mBuffer == 0)
{
cerr << "out of memory." << endl ;
exit(0) ;
} //if
} //if

//copy values
strcpy(mBuffer, rs.mBuffer) ;
mLength = rs.mLength ;
mBuffer[mLength] = '\0' ;

return (*this) ;
} //operator=

int main ()
{
String s ("Hello World") ;
String test ;
String chain ;

```

```

test = s ;
test.status() ;
test = "Hello" ;
test.status() ;
test = test ;
chain = test = s ;
chain.status() ;
} //main
//

```

Das Programm liefert die Ausgabe:

```

String::String(const char *) called.
String::String() called.
String::String() called.
String::operator= called.
length: 11, Buffer: Hello World
String::String(const char *) called.
String::operator= called.
String::~~String() called.
length: 5, Buffer: Hello
String::operator= called.
String::operator= : self assignment detected.
String::operator= called.
String::operator= called.
length: 11, Buffer: Hello World
String::~~String() called.
String::~~String() called.
String::~~String() called.

```

3.5.2 Returnwert

Wie aus der Signatur des Zuweisungsoperators ersichtlich ist, gibt dieser eine Referenz auf ein Objekt gleichen Typs zurück. Dies ist zwar nicht zwingend erforderlich, jedoch nötig, um übliche Kettenzuweisungen, wie etwa `chain = test = s ;` im Beispielprogramm, zu ermöglichen. Da der Zuweisungsoperator rechtsassoziativ ist, ist dieser Ausdruck gleichwertig zu:

```
chain.operator=(test.operator=(s)) ;
```

Wichtig ist, dass stets der *Linkswert* (engl.: *lvalue*) der Zuweisung zurückgegeben werden sollte, also das Objekt, für das der Zuweisungsoperator aufgerufen wurde (`return (*this)` im Beispiel). Die rechte Seite (*rvalue*) kann nach obiger Signatur nicht als `T&` zurückgegeben werden, da sie als konstante Referenz (`const T&`) übergeben wird. Selbst wenn man dies (unschön) umginge, indem man die rechte Seite als `T&` übergeben würde, birgt die Rückgabe des *rvalue* eine Gefahr in sich: Da der *rvalue* auch ein vom Compiler erzeugtes, temporäres

Objekt sein könnte (etwa weil der Compiler wie im obigen Beispiel eine implizite Typumwandlung von `char*` nach `String` in der Zeile

```
test = "Hello" ;
```

vornehmen muss), ist es denkbar, dass eine Kettenzuweisung wie

```
chain = test = "oops!" ;
```

nicht korrekt funktioniert, weil der Compiler das temporäre Objekt, das für die Zuweisung

```
test = "oops!" ;
```

erzeugt wurde, bereits wieder zerstört hat, bevor der Zuweisungsoperator

```
chain.operator=
```

aufgerufen wird!

3.5.3 Selbstzuweisung

Es sollte bei der Implementierung des Zuweisungsoperators geprüft werden, ob nicht eine Selbst-Zuweisung erfolgt (z.B. `test = test ;`). In diesem Fall muss beispielsweise darauf geachtet werden, dass dynamisch allocierter Speicher der linken Seite nicht einfach mit `delete` freigegeben werden darf (was bei einer Fremd-Zuweisung korrekt wäre), da ja sonst auch der Speicher der rechten Seite zerstört würde! Im obigen Beispiel der `String`-Klasse wird daher der Fall `this == &rs` explizit abgeprüft.

3.5.4 Abgeleitete Klassen und Zuweisungsoperator

Sofern eine Klasse `B` von der Klasse `A` abgeleitet wurde, gilt es sicherzustellen, dass auch die von der Klasse `A` geerbten Member zugewiesen werden müssen, da dies natürlich *nicht automatisch* geschieht! Es ist hierbei nicht möglich, im Zuweisungsoperator der Klasse `B` auf private Member der Klasse `A` zuzugreifen, so dass eine direkte Zuweisung der einzelnen `A`-Member innerhalb `B.operator=` nicht durchführbar ist. Stattdessen muss hier explizit der Zuweisungsoperator der Klasse `A` aufgerufen werden, was man durch folgende Konstruktion erreicht:

```
B & B::operator= (const B & r)
{
    ...
    ((A&) *this) = r ;
    ...
}
```

Es wird also `*this` (die linke Seite) in eine Referenz auf `A` umgewandelt, und danach der Zuweisungsoperator `A::operator=` von `A` aufgerufen. Keinesfalls darf hier auf `A` statt auf `A&` umgewandelt werden, da dann eine Kopie von `A` erzeugt würde. Dieser Kopie würden dann die Werte der rechten Seite zugewiesen, und `*this` bliebe unverändert. Alternativ könnte man auch schreiben:

```
...  
A::operator=(r) ;  
...
```

Dies ist erlaubt, da sich Aufrufe von Member-Funktionen innerhalb anderer Member-Funktionen immer auf das aktuelle Objekt (`*this`) als linke Seite beziehen.

3.6 Zusammenfassung

- Bei Konstruktoren sollten Daten-Member über die Initialisierungsliste initialisiert werden und nicht mittels Zuweisung.
- Konstante Daten-Member und Referenzen müssen über die Initialisierungsliste initialisiert werden.
- Statische Member sollten weder im Konstruktor noch im Destruktor verändert werden.
- Integer Klassen-Konstanten können durch anonyme enums realisiert werden.
- Copy-Konstruktor und Assignment-Operator sind korrekt zu implementieren (deep copy), sobald eine Klasse dynamisch erzeugten Speicher verwendet.
- Der Assignment-Operator sollte stets eine Referenz auf den Linkswert (`*this`) zurückgeben.
- Beim Assignment-Operator auf Selbstzuweisung prüfen.
- Bei abgeleiteten Klassen muss im Assignment-Operator auch für die korrekte Zuweisung der Member der Basisklasse gesorgt werden.

Kapitel 4

Klassenmethoden

Neben den speziellen Klassenmethoden “Konstruktor” und “Destruktor” gilt es auch für allgemeine Klassenmethoden auf einige Besonderheiten zu achten, die nachfolgend dargestellt werden.

4.1 Konstante Klassenmethoden

Sofern eine Klassenmethode das Objekt, für das sie aufgerufen wird, nicht verändert, kann die Klassenmethode auch als `const` deklariert werden. Ist eine Klassenmethode einer Klasse `T` als `const` deklariert, so wechselt der Typ des Pointers `this` von `T * const` zu `const T * const`, und es kann innerhalb der Methode nur noch lesend auf Daten-Member zugegriffen werden. `const` Klassenmethoden (und nur diese) können auch für `const` Objekte aufgerufen werden, wie dies in folgendem Beispiel an der Erweiterung der Klasse `String` gezeigt wird:

```
class String
{
    ...
public:
    ...
    //for const Strings
    const char & operator[] (int i) const
    {
        cout << "String::operator[] const called." << endl ;
        return mBuffer[i] ;
    };

    //for non-const Strings
    char & operator[] (int i)
    {
        cout << "String::operator[] (non-const) called." << endl ;
    };
};
```

```

    return mBuffer[i] ;
} ;

...
} ;

```

Die folgenden Aufrufe sind damit erlaubt:

```

int main ()
{
    const String cstring("Const");
    String ncstring("nonConst");
    cout << "char: " << cstring[0] << endl;
    ncstring[0] = 'N' ;
    ncstring.status() ;
}

```

Die Ausgabe lautet:

```

String::String(const char *) called.
String::String(const char *) called.
String::operator[] const called.
char: C
String::operator[] (non-const) called.
length: 8, Buffer: NonConst
String::~String() called.
String::~String() called.

```

Generell gilt, dass alle Methoden, die den Objektzustand nicht verändern, auch als `const` deklariert werden sollten. Dies mag zwar als unnötiger Aufwand erscheinen, wenn man die entsprechenden Klassen nur selbst benutzt, es wird jedoch rasch zur Notwendigkeit, wenn auch andere Anwender die Klasse benutzen, *denn sie werden durchaus auch const Objekte instantiieren, ob dies vom Besitzer der Klasse nun so beabsichtigt war oder nicht*, etwa wenn ein Objekt an eine Funktion als `const &` übergeben wird.

Ein besonderer Fall tritt ein, wenn eine Klassenmethode nur Member verändert, die der internen Implementierung zuzuordnen sind, nicht aber den vom Anwender aus sichtbaren Objektzustand. In diesem Fall würde die Deklaration der Methode als `const` jede Veränderung eines Members (auch diejenigen, die nur implementierungsbedingt sind) verhindern, wenn nicht die Möglichkeit bestünde, die Konstanz des Pointers `this` wieder "wegzucastern". Dies sei im folgenden Beispiel gezeigt: Die `String`-Klasse sei um verschiedene Methoden erweitert worden, die auch die Länge des Strings verändern. Es wird zusätzlich eine Funktion `getLength()` definiert, die die aktuelle Länge des Strings zurückliefert. Die aktuelle Länge wird nur dann berechnet, wenn sie sich verändert hat, ansonsten wird der Wert aus einem Buffer geholt. Über ein Flag wird angezeigt, ob

der Wert im Buffer gültig ist oder ob eine Neuberechnung erforderlich ist. Da `getLength()` den Wert des Strings nicht verändert, soll sie als `const` deklariert werden (*konzeptionelle Konstanz*). Dies wird wie folgt erreicht:

```
int String::getLength() const
{
//make this non-const
String * const myThis = this ;

    if (!mLengthIsValid)
        {
            myThis->mLength = strlen(mBuffer) ;
            myThis->mLengthIsValid = 1 ;
        }
    return mLength ;
} //get_length
```

Durch die Umwandlung von `this` in einen `String * const` können nun die Daten-Member verändert werden, obwohl die Methode als `const` deklariert wurde.

4.2 Operatoren

Operatoren können entweder als member-functions (Komponentenoperator) oder als nonmember-functions deklariert werden. Für jeden Operator darf jedoch immer nur eine der beiden Varianten deklariert werden.

Bei der Implementierung von Operatoren ist zusätzlich sorgfältig darauf zu achten, dass der Return-Typ korrekt definiert wird, um nicht schwer zu findende Speicherlöcher zu produzieren.

Ebenfalls kritisch ist die vernünftige Festlegung der Semantik der Operatoren, die ja im Prinzip frei definierbar ist.

Diese Probleme werden im Folgenden genauer betrachtet. Als Beispiel wird in diesem Abschnitt die Klasse `Complex` dienen, da sich für sie anschauliche und sinnvolle Operatoren definieren lassen.

4.2.1 Unäre Komponentenoperatoren

Allgemein hat ein n-ärer Komponentenoperator n-1 Argumente, da der ganz links stehende Operand immer implizit durch `*this` vorgegeben wird. Da vom Compiler *für den Pointer `this` keine impliziten Typumwandlungen in andere Klassen vorgenommen werden*, findet also beim Komponentenoperator auch keine implizite Typumwandlung auf den ganz links stehenden Operanden statt.

Dies sei am Beispiel des unären Operators “-” gezeigt. Gegeben sei folgende Definition einer Klasse `X` und einer Klasse `Y`, sowie der Nicht-Komponenten-Operator `operator-`:

```

class Y
{
...
} ; // class Y

class X
{
...
X(const Y &) ;           // conversion from Y to X
...

} ; // class X

// unary nonmember-function operator -
X operator-(const X&) ;

```

Man betrachte folgendes Programmfragment:

```

X a ;
Y b ;
...
a = -b ;

```

Der Ausdruck `a = -b ;` ist zulässig, da er vom Compiler durch `a = -(X(b))` ersetzt wird. Wäre der Operator als Komponentenoperator deklariert worden, etwa als

```
X X::operator-() const ;
```

so könnte die rechte Seite der Zuweisung nicht vom Compiler übersetzt werden, da für die Klasse `Y` kein `operator -` definiert wurde, und ja keine impliziten Typumwandlung von `*this` vorgenommen werden. Es ist jedoch im Allgemeinen durchaus erwünscht, dass solche schwer nachvollziehbaren impliziten Typumwandlungen unterbleiben, denn sie erschweren die Lesbarkeit und damit auch die Wartbarkeit der Programme erheblich. Es kann daher empfohlen werden, *unäre Operatoren als Komponentenoperatoren* zu deklarieren.

4.2.2 Präfix- und Postfix-Form

Eine interessante Frage stellt sich bei Betrachtung der unären Komponentenoperatoren `operator++` und `operator--`: Worin unterscheidet sich die Signatur der Operatoren in der Präfix-Form (z.B. `++a`) von der der Postfix-Form (z.B. `a++`)? Um die Operatoren in der gewohnten Weise anhand ihrer Signatur unterscheiden zu können, wurde vereinbart, dass die Postfix-Form einen zusätzlichen `int` als Parameter erhält, der immer mit dem Wert `0` vorbelegt ist. Wird die Postfix-Form nicht explizit definiert, so wird vom Compiler stets der Präfix-Operator

verwendet. Präfix- und Postfix-Form seien am Beispiel der Klasse `Complex` verdeutlicht, bei der der Operator `++` den Realteil um 1.0 erhöht:

```
//  
  
//complex.cc  
#include<iostream.h>  
  
class Complex  
{  
private:  
double mRe ;  
double mIm ;  
  
public:  
  
// constructor  
Complex (double re, double im = 0.0 )  
: mRe(re), mIm(im)  
{;} ;  
  
// copy-constructor  
Complex (const Complex & c)  
: mRe(c.mRe), mIm(c.mIm)  
{  
cout << "Complex::Complex(const Complex&) called." << endl ;  
} ;  
  
// destructor  
~Complex ()  
{  
cout << "Complex::~~Complex() called." << endl ;  
}  
  
// assignment-operator  
Complex & operator= (const Complex & c)  
{  
cout << "Complex::operator= called." << endl ;  
mRe = c.mRe ;  
mIm = c.mIm ;  
return (*this) ;  
}  
  
// assignment-operator+=  
Complex & operator+= (const Complex & c)  
{  
cout << "Complex::operator+= called." << endl ;
```

```

mRe += c.mRe ;
mIm += c.mIm ;
return (*this);
}

// prefix-operator++
const Complex & operator++()
{
cout << "Complex::operator++ called (prefix)." << endl ;
mRe += 1.0 ;
return *this ;
} ;

//postfix-operator++
const Complex operator++(int)
{
cout << "Complex::operator++ called (postfix)." << endl ;
Complex copy(*this) ;
++(*this);
return copy ;
} ;

// output-operator
friend ostream & operator<< (ostream & os, const Complex & c)
{
return ( os << "( " << c.mRe << " , " << c.mIm << " ) " ) ;
} ;

// one version of binary operator +
// friend const Complex operator+ (const Complex & c1, const Complex & c2)
// {
// cout << "operator+ (const Complex&, const Complex&) called."
// << endl ;
// return Complex( (c1.mRe + c2.mRe) , (c1.mIm + c2.mIm) ) ;
// }

} ; // class Complex

// better version of operator +
const Complex operator+ (const Complex & c1, const Complex & c2)
{
cout << "operator+ (const Complex&, const Complex&) called." << endl ;
return (Complex(c1) += c2) ;
}

int main ()

```

```

{
Complex a(1.5) ;

cout << "a: " << a << endl ;
cout << "pre-increment: " << ++a << endl ;
cout << "a: " << a << endl ;
cout << "post-increment: " << a++ << endl ;
cout << "a: " << a << endl ;

a += 2.0 ;
cout << "a+2.0 : " << a << endl ;

Complex b = 3.14 + a ;
cout << "3.14+a: " << b << endl ;
}
//

```

Die zugehörige Ausgabe lautet:

```

a: ( 1.5 , 0 )
Complex::operator++ called (praefix).
pre-increment: ( 2.5 , 0 )
a: ( 2.5 , 0 )
Complex::operator++ called (postfix).
Complex::Complex(const Complex&) called.
Complex::operator++ called (praefix).
Complex::Complex(const Complex&) called.
Complex::~Complex() called.
post-increment: ( 2.5 , 0 )
Complex::~Complex() called.
a: ( 3.5 , 0 )
Complex::operator+= called.
Complex::~Complex() called.
a+2.0 : ( 5.5 , 0 )
operator+ (const Complex&, const Complex&) called.
Complex::Complex(const Complex&) called.
Complex::operator+= called.
Complex::Complex(const Complex&) called.
Complex::~Complex() called.
Complex::Complex(const Complex&) called.
Complex::~Complex() called.
Complex::~Complex() called.
3.14+a: ( 8.64 , 0 )
Complex::~Complex() called.
Complex::~Complex() called.

```

Wie man sieht, muss in der Postfix-Form eine Kopie von `*this` angelegt

werden, da als Return-Wert der Wert *vor* der Anwendung des Operators zurückgegeben werden muss. Der Return-Typ wurde als `const Complex` und nicht als `const Complex &` angegeben, da ja nie eine Referenz auf ein temporäres Objekt zurückgegeben werden darf. Bei der Präfix-Form kann jedoch gefahrlos `const Complex &` zurückgegeben werden, da (`*this`) natürlich auch nach dem Aufruf des Operators noch existiert.

Zu beachten ist, dass in der Postfix-Implementierung die Präfix-Form eingesetzt wird, um beide Varianten stets konsistent zu halten.

Für beide Operator-Varianten wurde der Return-Typ als `const` definiert, um den Gebrauch von Sprachkonstrukten wie etwa `a++++` zu verbieten. Da ein solcher Ausdruck gleichwertig mit `(a++)++` ist, würde hier zunächst das Objekt `a` einmal inkrementiert und danach das temporäre Objekt, welches vom ersten Operator zurückgegeben wird. Dies entspricht nicht der gewünschten Semantik des Operators. Der Ausdruck `i++++` ist auch verboten, wenn `i` vom integralen Typ `int` ist.

Generell kann empfohlen werden, sich bei der Definition von Operatoren an der Semantik der entsprechenden Operatoren des Datentyps `int` zu orientieren.

4.2.3 Zuweisungsoperatoren

Für die binären Zuweisungsoperatoren (`+=`, `-=`, `*=`, `/=`, `%=` etc.) gilt, dass sie immer das Objekt verändern, welches auf der linken Seite der Zuweisung steht. Eine implizite Typenumwandlung ist somit nie erforderlich.

Daher sollten die Zuweisungsoperatoren als Komponentenoperatoren deklariert werden.

Da die Zuweisungsoperatoren stets ein bereits beim Aufruf des Operators existierendes Objekt verändern, können sie gefahrlos als Returnwert eine Referenz auf das Objekt (`*this`) zurückgeben, wie dies bereits für den Assignment-Operator `operator=` in Abschnitt 3.5 beschrieben wurde.

4.2.4 Restliche binäre Operatoren

Für die restlichen binären Operatoren (`+`, `-`, `*`, `/`, `%` etc.) erwartet man ein bezüglich der impliziten Typumwandlung symmetrisches Verhalten, so dass etwa für die Objekte `a` und `b`, die beide vom Typ `Complex` seien, die Anweisungen `b = a + 3.14` ; sowie `b = 3.14 + a` ; beide gültig sein sollten. Würde man nun den `operator+` als Komponentenoperator von `Complex` deklarieren, so wäre der zweite Ausdruck wegen der bereits besprochenen fehlenden impliziten Typumwandlung des linken Operanden nicht übersetzbar.

Wird der `operator+` aber wie im Beispiel als nonmember-function deklariert, so kann der Compiler die implizite Umwandlung von `double` in den Typ `Complex` vornehmen und somit den Ausdruck der rechten Seite auswerten. Muss bei der Implementierung der Operatoren auf private Member der Operanden zugegriffen werden, so muss der Operator wie im Beispiel als `friend` deklariert werden. Zu beachten ist, dass der `operator+` zwar innerhalb der Klassendefinition von `Complex` definiert wird, dass er jedoch *kein* Member der Klasse ist.

Für die binären Operatoren empfiehlt sich also die Implementierung als non-member-functions.

Bei der Betrachtung des binären `operator+` fällt auf, dass dieser wie im Falle des `operator++` ebenfalls ein Objekt vom Typ `const Complex` zurückliefert, um Sprachkonstrukte wie etwa `(a+b)++` zu verbieten. Wichtig ist auch zu erkennen, dass die binären Operatoren keines der beiden Funktionsargumente verändern sollen (diese werden ja auch als `const &` übergeben. Es soll aber ein korrekt initialisiertes Objekt als Ergebniswert zurückgegeben werden. *Somit ist es bei binären Operatoren unvermeidlich, ein temporäres Objekt anzulegen und eine Kopie (keine Referenz !) dieses Objektes zurückzugeben*, wie dies auch schon im Falle der Postfix-Form des `operator++` erforderlich war.

Der Runtime-Overhead zur Erzeugung des temporären Objektes sowie der Kopie des Return-Wertes kann jedoch vom Compiler minimiert werden, wenn wie im Beispiel der Konstruktoraufruf in die `return`-Anweisung eingebettet wird. Die Eliminierung dieses Overheads wird als *“return value optimization”* bezeichnet.

Unschön an der vorgestellten Implementierung des binären `operator+` ist jedoch, dass nun zwei getrennte Implementierungen fuer den Fall des Zuweisungsoperators `operator+=` und des binären `operator+` existieren, die getrennt gepflegt werden müssen. Um dieses Problem zu umgehen, ist folgende Implementierung günstiger:

```
const Complex operator+ (const Complex & c1, const Complex & c2)
{
    cout << "operator+ (const Complex&, const Complex&) called."
         << endl ;
    return (Complex(c1) +=c2) ;
}
```

Bei dieser Implementierung wird der Komponentenoperator `+=` eingesetzt, so dass die Implementierungen von `operator+=` und `operator+` zwangsweise konsistent sind. Darüber hinaus muss nun der `operator+` nicht mehr unschön als `friend` deklariert werden, da der Zuweisungsoperator als Komponentenoperator realisiert ist und somit Zugriff auf die privaten Daten-Member hat. Erkauft wird dieser Vorteil jedoch damit, dass je nach Qualität der Compilers diese Variante weniger gut optimiert werden kann.

4.3 Zusammenfassung

In der nachfolgenden Übersicht sind nochmals die Implementierungsempfehlungen für die Operatoren zusammengefasst:

Operator	Beispiel	Membership	Returnwert
unär	+, -	member	by value
unär, postfix	<code>operator++ (int)</code>	member	const, by value
binär, Zuweisung	+=, -=	member	by reference
binär allgem.	+, -	nonmember	const, by value

Kapitel 5

Ein-/Ausgabe mit Streams

Die Ein- bzw. Ausgabe kann in C++ komfortabel über sog. *Streams* erfolgen. Im Rahmen dieses Kurses wird hierbei vornehmlich auf die Streams zur formatierten Ein-/Ausgabe eingegangen, die von der Basisklasse `ios` abgeleitet sind. Die von `streambuf` abgeleiteten Klassen dienen der unformatierten Handhabung von Streams und werden hier nicht näher besprochen. Eine umfassende Darstellung der Streams findet sich in [Hitz].

5.1 ios

Von der Basisklasse `ios` abgeleitet sind die Klassen `istream` und `ostream`, die der formatierten Ein- bzw. Ausgabe dienen. Je nachdem, ob die davon abgeleiteten Klassen intern ein `strstreambuf`-Objekt (Buffer im Hauptspeicher) oder ein `filebuf`-Objekt (File-Buffer) verwalten, werden sie als `istream` oder `ifstream` bzw. als `ostream` oder `ofstream` bezeichnet. Drei Streams sind immer verfügbar und in `iostream.h` deklariert:

```
extern istream cin ;
extern ostream cout, cerr ;
```

Diese Streams müssen nicht explizit instantiiert werden.

5.2 Ausgabe auf Streams

Zur Ausgabe auf Streams wird der `operator<<()` benutzt, welcher bereits im Beispiel `complex` eingesetzt wurde. Für die eingebauten Datentypen wie `int`, `float`, `char*` etc. ist der Operator bereits als Komponentenoperator implementiert:

```
ostream & ostream::operator<<(T) {...};
```

Für selbst definierte Datentypen kann der `operator<<()` natürlich nicht als Komponente von `ostream` deklariert werden, da der Benutzer ja in der Regel nicht der Eigentümer der Klasse `ostream` ist. Hier muss der globale Operator verwendet werden:

```
ostream & operator<<(ostream & , const T &) ;
```

Die Output-Operatoren geben vereinbarungsgemäß eine Referenz auf das erste Argument (also `*this` für den Komponentenoperator bzw. `ostream` für den globalen Operator) zurück. Dies ermöglicht die Verkettung von Ausgaben, wie etwa in folgender Anweisung:

```
int a=1 ;
int b=2 ;
cout << a << b ;
```

Dies führt zum Aufruf

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

Wie man sieht, ist der Ausgabeoperator linksassoziativ.

5.2.1 Formatierung

Feldbreite

Mit der Methode `width()` kann die Feldbreite der Ausgabe beeinflusst werden. Durch Aufruf von `width(n)` wird die minimale Anzahl der auszugebenden Zeichen für die nachfolgende (und nur die nachfolgende) Ausgabeoperation eines numerischen Wertes oder eines Strings auf `n` gesetzt. Die alte Feldbreite wird dabei als Return-Wert zurückgegeben. Der Default-Wert für die Feldbreite ist 0.

So liefert etwa

```
cout.width(5) ;
cout << '(' << 123 << ')' << endl ;
```

die Ausgabe:

```
( 123)
```

Wird `width()` ohne Parameter aufgerufen, so wird einfach die aktuelle Feldbreite zurückgegeben.

Füllzeichen

Will man das Füllzeichen, das zum Auffüllen des Feldes verwendet wird ändern, so ist dies durch Aufruf von `fill()` möglich. Das alte Füllzeichen wird als Return-Wert zurückgegeben.

So liefert

```

cout.width(5) ;
cout.fill('#') ;
cout << '(' << 123 << ')' << endl ;

```

die Ausgabe:

```
(##123)
```

Der Aufruf von `fill()` ohne Parameter liefert das aktuelle Füllzeichen.

Formatflags

Zur weiteren Beeinflussung der Formatierung sind in `ios` Flags definiert, die mittels der Methoden `setf()` und `flags()` manipuliert werden können. Nachfolgend sei zur Übersicht über die möglichen Optionen ein Auszug aus der Klassendefinition von `ios` gegeben:

```

...
enum { skipws=_IO_SKIPWS,           // skip whitespace on input
       left=_IO_LEFT,              // padding after value
       right=_IO_RIGHT,            // padding before value
       internal=_IO_INTERNAL,      // padding between sign and value
       dec=_IO_DEC,                // base is decimal
       oct=_IO_OCT,                // base is octal
       hex=_IO_HEX,                // base is hexadecimal
       showbase=_IO_SHOWBASE,     // show integer base
       showpoint=_IO_SHOWPOINT,   // print trailing zeros
       uppercase=_IO_UPPERCASE,   // print base / exp in uppercase
       showpos=_IO_SHOWPOS,       // explicit '+' for pos. value
       scientific=_IO_SCIENTIFIC,  // scientific
       fixed=_IO_FIXED,            // fixpoint
       ...
};
...

```

Die angegebenen numerischen Werte sind implementierungsabhängig und hier dem GNU C++ - Compiler entnommen. Die Methode `fmtflags ios::flags(fmtflags)` liefert die alten Optionen zurück und erwartet die Bitmask der neuen Optionen als Argument. Etwas komfortabler ist die Methode `fmtflags ios::setf(fmtflags value, fmtflags mask)`, die nur diejenigen Bits setzt bzw. löscht, die in der Bitmaske `mask` auf Eins gesetzt sind. Folgende Bitmasken sind vordefiniert:

```

enum { // Masks.
       basefield=dec+oct+hex,
       floatfield = scientific+fixed,
       adjustfield = left+right+internal
};

```

Damit kann beispielsweise auf hexadezimale Darstellung umgeschaltet werden:

```
cout.setf(ios::hex, ios::basefield);
```

Wird `flags()` ohne Argument aufgerufen, so werden die aktuellen Optionen zurückgegeben.

Ausrichtung

mit den Flags `ios::left` und `ios::right` kann zwischen links- und rechtsbündiger Ausgabe innerhalb einer mit `width()` definierten Feldbreite gewählt werden. Durch Setzen von `ios::internal` werden die Füllzeichen zwischen dem Vorzeichen und dem Zahlenwert plaziert.

Gleitkommazahlen

Durch die Methode `precision()` wird die Zahl der Nachkommastellen der Gleitkommazahl eingestellt, falls als Format `ios::scientific` gewählt wurde. Im Falle der Fixpunktdarstellung (`ios::fixed`) wird hierdurch die Gesamtzahl der ausgegebenen Stellen bezeichnet. Die Angabe der Stellen begrenzt stets nur die Zahl der auszugebenden Nachkommastellen, so dass gegebenenfalls gerundet wird. Übersteigt bereits der ganzzahlige Wert die angegebene Stellenzahl, so wird dieser dennoch korrekt dargestellt, jedoch dann ohne Angabe von Nachkommastellen. Sind zur Zahlendarstellung weniger Nachkommastellen erforderlich als angegeben, so werden standardgemäß keine nachfolgenden Nullen ausgegeben. Dies kann jedoch durch Setzen von `ios::showpoint` erzwungen werden.

5.2.2 Manipulatoren

Bei den bisherigen Formatierungsangaben war jeweils der direkte Aufruf der Methode zur Formatierung erforderlich, bevor der Datentyp ausgegeben werden konnte. Dadurch zerfällt eine einfache Ausgabeanweisung in mehrere Anweisungen, die abwechselnd die Format-Optionen einstellen und die zugehörigen Daten ausgeben. Eine komfortablere Möglichkeit zur Beeinflussung der Formatierung bieten die sog. *Manipulatoren*, welche unmittelbar in die Ausgabeanweisung eingesetzt werden können. Als Beispiel sei hier der Manipulator `hex` genannt, der das Setzen von `ios::hex` bewirkt:

```
cout << "hex: " << hex << 123 << " dec: " << dec << 123 << endl ;
```

liefert etwa die Ausgabe:

```
hex: 7b dec: 123
```

Hierbei sei auch auf den Manipulator `endl` hingewiesen, der den Wagenrücklauf auf den Stream ausgibt und die Methode `flush()` des Buffers aufruft.

Ein solcher Manipulator könnte realisiert werden, indem eine spezielle Mittelklasse definiert wird, für welche man den `operator<<()` so überlädt, dass er die gewünschte Methode des Stream-Objektes aufruft. Hierbei könnten dann auch Parameter mit übergeben werden, wie dies im Folgenden gezeigt ist:

```

// manipulator with integer argument
class myPrecision
{
    int mi ;

    public :

    myPrecision (int i): mi(i)
    {cout << "myPrecision::myPrecision(" << i << ") called"<< endl ; } ;

    friend ostream & operator <<(ostream & o,const myPrecision & mp)
    {o.precision(mp.mi); return o ;}

} ;

```

Der Aufruf

```
cout << myPrecision(2) << 1.234567 << endl << endl;
```

liefert dann das gewünschte Ergebnis:

```
myPrecision::myPrecision(2) called
1.23e+00
```

Ein Objekt der Klasse `myPrecision` wird also mit dem Wert 2 instantiiert und danach der globale, als `friend` zu `myPrecision` deklarierte Ausgabeoperator für `myPrecision` aufgerufen. In dieser Methode wird dann die Manipulation an dem Stream vorgenommen.

Um nicht jedesmal eine eigene Klasse definieren zu müssen, wird in `iostream.h` für Manipulatoren *ohne* Parameter der Ausgabeoperator `operator<<()` so überladen, dass er einen Pointer auf eine Funktion als Argument erwartet, welche ihrerseits eine Referenz auf einen `ostream` als Parameter erwartet und eine Referenz auf einen `ostream` zurückgibt:

```

typedef ostream& (*__omanip)(ostream&);
...
class ostream : virtual public ios
{
    ...
    ostream& operator<<(__omanip func) { return (*func)(*this); }
    ...
} ;

```

In diesem Beispiel wurde der besseren Lesbarkeit halber der Funktionspointer mittels `typedef` mit dem Namen `__omanip` versehen.

Diese spezielle Variante eines Operators, welcher nur eine Methode, die als Argument übergeben wird, aufruft, wird auch als *Applikator* bezeichnet.

Ausgehend von diesem bereits deklarierten Applikator kann man leicht Manipulatoren ohne Parameter realisieren, denn man muss nur die zugehörige Funktion definieren:

```

ostream & myFormat (ostream & o)
{
    o.setf(ios::showpoint,ios::showpoint);
    o.setf(ios::scientific,ios::floatfield);
    o.precision(8);
    return (o) ;
}

```

Diese Funktion kann nun vom Applikator aufgerufen werden:

```

cout << 12.34567 << endl
     << "my Format: " << myFormat << 12.34567 <<  endl << endl;

```

Dies liefert die Ausgabe:

```

12.3457
my Format: 1.23456700e+01

```

Um Manipulatoren *mit* Parametern allgemein zu realisieren, ist ein aufwendigeres Verfahren nötig, welches nun kurz vorgestellt werden soll:

Hierzu wird zunächst angenommen, dass ein Parameter vom Typ `int` übergeben werden soll. Nun wird eine spezielle Mittlerklasse definiert:

```

class Manip_int
{
    int mi ;
    ostream & (* mpfunc)(ostream &,int) ;
    public :

    // constructor
    Manip_int (ostream & (*pfunc)(ostream & o, int), int i)
        : mpfunc(pfunc), mi(i)
    {cout << "Manip_int::Manip_int() called." << endl ; }

    // output operator
    friend ostream & operator << (ostream & o, const Manip_int & m)
    {
        cout << "operator<<(): applying function to ostream..." << endl ;
        return (m.mpfunc(o, m.mi)) ; // apply function mpfunc to ostream o
    }

} ; // class Manip_int

```

Im Konstruktor dieser Klasse wird ein Funktionspointer und der `int`-Parameter gespeichert. Für diese Klasse wird wieder der Ausgabeoperator als Applikator definiert, welcher die gespeicherte Funktion `mpfunc` auf den Stream `ostream` anwendet und dabei der Funktion den gespeicherten Parameter `mi` übergibt.

Der Operator wird als `friend` zur Klasse `Manip_int` deklariert, da er auf die privaten Member zugreifen muss.

Nun wird eine Funktion definiert, welche beim Aufruf mit einem `int`-Parameter ein Objekt von Typ `Manip_int` erzeugt, welches mit dem gewünschten Funktionspointer und dem `int`-Parameter initialisiert wird:

```
// define a function creating the manip_int object
Manip_int myWidth (int i )
{
    // initialize the Manip_int object with function and argument
    cout << "myWidth: creating Manip_int..." << endl ;
    return Manip_int(&setWidth, i) ;
}
```

Nun fehlt noch die Definition der eigentlichen Stream-Manipulation, also derjenigen Funktion, die als ersten Parameter eine `ostream &` erwartet und als zweiten Parameter den `int`-Wert. Diese Funktion muss als Returnwert eine `ostream &` zurückgeben:

```
// define a function setting the fieldwidth and the fillpattern
ostream & setWidth (ostream & o, int value)
{
    cout << "setWidth: manipulating ostream..." << endl ;
    o.width(value);
    return o ;
}
```

Nun kann eine Ausgabeanweisung wie folgt aussehen:

```
cout << myWidth(8) << 123 << endl << endl;
```

Was folgende Ausgabe liefert:

```
myWidth: creating Manip_int...
Manip_int::Manip_int() called.
operator<<(): applying function to ostream...
setWidth: manipulating ostream...
    123
```

Wie man sieht, erzeugt der Aufruf von `myWidth()` zunächst ein Objekt der Klasse `Manip_int`, welches als Initialisierungswerte den Funktionspointer `setWidth` und den `int`-Wert `8` erhält. Dieses Objekt wird nun durch seinen Ausgabeoperator "ausgegeben", d.h., der Applikator wendet nun die Funktion `setWidth` auf den Stream an.

Um nun nicht für jeden Datentyp `T`, der als Parameter in einem Manipulator verwendet wird, eine eigene Klasse `Manip_T` definieren zu müssen, wird üblicherweise in `iomnip.h` eine allgemeine *Template-Klasse* für den Manipulator mit einem Argument definiert. Beim GNU C++ Compiler sieht diese Definition wie folgt aus:

```

template <class TP> class omanip {
    ostream& (*_f)(ostream&, TP);
    TP _a;
public:
    omanip(ostream& (*f)(ostream&, TP), TP a) : _f(f), _a(a) {}
    //
    friend
        ostream& operator<<(ostream& o, const omanip<TP>& m);
};

```

Wie man sieht, ist hier der `DatenTyp`, den der Manipulator als Argument für die Funktion `_f` speichert, als Template `TP` realisiert. Die Implementierung des Ausgabeoperators lautet:

```

template <class TP>
inline ostream& operator<<(ostream& o, const omanip<TP>& m)
{ return (*m._f)(o, m._a); }

```

Dies entspricht genau der Implementierung aus `Manip_int`. Um nun einen Manipulator für einen *konkreten Datentyp* zu implementieren, müssen noch zwei Funktionen realisiert werden: Die Funktion zur eigentlichen Stream-Manipulation (im Beispiel die Funktion `setWidth(ostream&, int)`) und diejenige Funktion, die das Manipulator-Objekt erzeugt (im Beispiel `myWidth(int)`). Die Funktion zur Stream-Manipulation muss in jedem Fall individuell realisiert werden, denn in ihr wird ja die neue Eigenschaft des Streams nach ihrer Anwendung definiert. Die zweite Funktion aber, die der Erzeugung des passenden Manipulator-Objektes dient, kann über Macro-Aufrufe generiert werden, was entsprechend Arbeit spart. Die entsprechende Macrodefinition lautet beim GNU C++ - Compiler:

```

//
// Macro to define an iomanip function, with one argument
// The underlying function is ‘__iomnip_<name>’
//
#define __DEFINE_IOMANIP_FN1(type,param,function) \
    extern ios& __iomnip_##function (ios&, param); \
    inline type<param> function (param n) \
        { return type<param> (__iomnip_##function, n); }

```

Wie man sieht, kann in dem Macro sowohl der `DatenTyp` des Parameters über den Macro-Parameter `param` wie auch die zugehörige Manipulations-Funktion über den Macro-Parameter `function` angegeben werden. Zusätzlich kann mit dem ersten Argument auch noch der Typ des Manipulators festgelegt werden. Hier wird festgelegt, ob es sich um einen Manipulator handelt, der die Streams `ios`, `ostream` oder `istream` manipuliert. Der Aufruf des Macros zur Erzeugung von `setprecision()` lautet beispielweise:

```

__DEFINE_IOMANIP_FN1( smanip, int, setprecision)

```

Die zugehörige Manipulations-Funktion muss dann unter dem Namen `__iomanip_setprecision` definiert werden. Nachfolgend sei eine kurze Übersicht über die wichtigsten Manipulatoren gegeben:

Manipulator	Funktion
<code>ios& oct(ios&)</code>	oktales Zahlensystem
<code>ios& dec(ios&)</code>	dezimales Zahlensystem
<code>ios& hex(ios&)</code>	hexadezimales Zahlensystem
<code>ostream& endl(ostream&)</code>	Wagenrücklauf und <code>flush()</code>
<code>ostream& ends(ostream&)</code>	String Terminieren und <code>flush()</code>
<code>ostream& flush(ostream&)</code>	Flush Stream
<code>SMANIP<int> setbase(int)</code>	Zahlenbasis setzen
<code>SMANIP<int> setfill(int)</code>	Füllzeichen setzen
<code>SMANIP<int> setprecision(int)</code>	Gleitkommagenauigkeit setzen
<code>SMANIP<int> setw(int)</code>	Feldbreite setzen
<code>SMANIP<long> resetiosflags(long)</code>	Optionsflags rücksetzen
<code>SMANIP<long> setiosflags(long)</code>	Optionsflags setzen

5.3 Eingabe mittels Streams

Die Eingabe mittels Streams wird analog zur Ausgabe über den Eingabeoperator `operator>>()` durchgeführt. Für eingebaute Datentypen ist dieser wieder als Komponentenoperator realisiert:

```
istream & istream::operator>>(T) {...};
```

Selbst definierte Datentypen müssen wieder auf den globalen Operator ausweichen:

```
istream & operator>>(istream &, T&) {...} ;
```

Auch die Eingabeoperatoren geben vereinbarungsgemäß eine Referenz auf ihr erstes Argument zurück, um Verkettungen zu erlauben:

```
cin >> a >> b ;
```

Der Eingabeoperator ist ebenfalls linksassoziativ.

5.3.1 Whitespaces

Whitespaces (dazu gehört auch `'\n'`) werden per default überlesen. Dies kann jedoch durch explizites Löschen des Flags `ios::skipws` verhindert werden, so dass auch Trennzeichen explizit eingelesen werden müssen. Hierzu dient dann der Manipulator `ws`.

Bei Strings gilt die Ausnahme, dass ein String immer durch Whitespaces terminiert wird, dass also ein String, der mit Whitespaces beginnt, unabhängig vom Status von `ios::skipws` garnicht erst eingelesen wird, es sei denn die führenden Whitespaces werden zuvor explizit durch den Manipulator `ws` entfernt.

5.3.2 Felddbreite

Um einen Überlauf des Speichers beim Einlesen von Strings (`char *`) zu vermeiden, kann mit dem Manipulator `setw(int n)` die maximale Zeichenanzahl angegeben werden. Es werden dann nur `n-1` Zeichen eingelesen und der String anschließend nullterminiert. Ein Wert von 0 für die Felddbreite ist der Default-Wert und bedeutet “keine Begrenzung”.

5.3.3 Unformatierte Eingabe

Sollen unabhängig vom Status von `ios::skipws` keine Whitespaces überlesen werden, etwa zum Einlesen ganzer Zeilen aus einem File, so kann mit den Methoden zur unformatierten Eingabe gearbeitet werden. Bei diesen Methoden wird aus dem Stream direkt in einen angegebenen Buffer (`char *buff`) eingelesen. Nachfolgend seien die wichtigsten Methoden genannt:

- `int istream::get()` liest ein Zeichen vom Stream und gibt es als Returnwert zurück. Im Fehlerfall wird EOF zurückgegeben. Da EOF üblicherweise den Wert `-1` hat, ist der Returnwert vom Datentyp `int`, damit eine Unterscheidung vom Charakter mit dem Wert (`char`)(`-1`) möglich ist.
- `istream & istream::get(char& c)` liest ein Zeichen in die Variable `c` ein.
- `istream & istream::get(char *buff, int n, char delim='\n')` liest solange Zeichen in den Buffer `buff`, bis (in dieser Reihenfolge) entweder `n-1` Zeichen eingelesen wurden, oder ein Lesefehler auftritt oder das nächste einzulesende Zeichen gleich `delim` ist. *Der Delimiter verbleibt also im Stream.* Der eingelesene String wird anschließend nullterminiert.
- `istream & istream::getline(char *buff, int n, char delim='\n')` liest im Gegensatz zu `get()` den Delimiter mit ein. Wird der Delimiter nicht gefunden, weil schon `n-1` Zeichen eingelesen wurden, so wird `ios::failbit` gesetzt.
- `istream & istream::ignore(int n=1, int delim=EOF)` überliest die `n` folgenden Zeichen, oder bricht ab, falls `delim` gelesen wurde. *Der Delimiter wird also mit überlesen.* Setzt man `n=MAXINT`, so wird die Anzahl der bereits eingelesenen Zeichen nicht abgeprüft.
- `int istream::peek()` liefert das nächste einzulesende Zeichen, ohne es aus dem Stream zu entfernen.
- `istream & istream::putback(char c)` stellt den Character `c` in den Stream “zurück”, so dass er als nächstes Zeichen ausgelesen wird.
- `istream & istream::unget()` stellt das zuletzt eingelesene Zeichen in den Stream zurück, so dass es als nächstes Zeichen eingelesen wird.

5.4 Binäre Ein-/Ausgabe mit Streams

Die bisher besprochenen Methoden zur Ein- und Ausgabe bezogen sich alle auf die Übertragung von ASCII-Zeichenketten. Nicht immer möchte man aber alle Daten in ASCII-Zeichen konvertieren, bevor sie ein- oder ausgegeben werden, etwa wenn Gleitkommazahlen in ein File geschrieben werden sollen, da dies entweder zu sehr langen Zeichenketten oder zu einem Genauigkeitsverlust führt. Nachfolgend werden daher die Methoden zur binären Ein-/Ausgabe und zur Positionierung innerhalb von Streams vorgestellt.

5.4.1 Methoden zur binären Ein-/Ausgabe

- `ostream & ostream::put(char c)` gibt den einzelnen Character `c` aus.
- `istream & istream::get(char & c)` liest, wie bereits besprochen, das nächste Zeichen in die Variable `c` ein.
- `ostream & ostream::write(const char *buff, int n)` schreibt `n` Zeichen aus dem Buffer `buff` auf den Stream. Da es sich hier um eine Methode zur binären Ausgabe handelt, dürfen auch Nullzeichen im Buffer vorkommen. Es sei darauf hingewiesen, dass auf jeden Fall versucht wird, `n` Zeichen aus dem Buffer `buff` zu lesen. Der übergebene Buffer muss also mindestens die Länge `n` besitzen!
- `istream & istream::read(char * buff, int n)` liest `n` Zeichen aus dem Stream und trägt sie in den Buffer `buffer` ein. Auch hier werden Nullzeichen mit eingelesen und auch hier muss der Buffer mindestens `n` Zeichen lang sein.

5.4.2 Methoden zur Positionierung

Um innerhalb eines Streams die Position des nächsten Schreib- oder Lesevorganges einstellen zu können, sind entsprechende Positioniermethoden vorgesehen. Diese werden hier den Klassen `istream` bzw. `ostream` zugeordnet, obwohl sie in manchen Implementierungen eventuell nur als Methoden von `streambuf` verfügbar sind. Die Methoden zur Positionierung der Schreib- und Lesemarke werden durch das Suffix “p” bzw. “g” voneinander unterschieden, da sich bei gleichzeitigem Schreiben und Lesen (wie etwa bei Verwendung der Klasse `iostream`, die von `istream` und `ostream` abgeleitet ist) die Positionen von Schreib- und Lesemarke verschieden sein können.

- `streampos ostream::tellp()` liefert die aktuelle Position im Stream zurück.
- `ostream & ostream::seekp(streampos pos)` positioniert an die absolute Position `pos`.
- `ostream & ostream::seekp(streamoff off, seek_dir dir)` positioniert um den relativen Offset `off`, ausgehend vom Bezugspunkt `dir`, wobei `dir`

gewählt werden kann aus: `ios::beg` (Anfang des Streams), `ios::cur` (aktuelle Position) und `ios::end` (Ende des Streams).

- `streampos istream::tellg()` gibt die aktuelle Position im Stream zurück.
- `istream & istream::seekg(streampos pos)` positioniert an die absolute Position `pos`.
- `istream & istream::seekg(streamoff off, seek_dir dir)` positioniert relativ wie bei `ostream::seekp()` beschrieben.

Ebenfalls nützlich:

- `int istream::gcount()` liefert die Anzahl der bei der letzten unformatierten Eingabe tatsächlich gelesenen Zeichen. War die letzte Eingabe eine formatierte Eingabe, so ist das Ergebnis undefiniert.
- `int ostrstream::pcount()` liefert die Zahl der zuletzt ausgegebenen Zeichen. Vorsicht: `pcount()` ist eine Methode von `ostrstream`, aber `gcount()` eine Methode von `istream` !

5.5 Stream Zustände

Bei allen bisher beschriebenen Methoden können Fehler auftreten. Um eine Fehlerabfrage zu erlauben, wurde in der Klasse `ios` der `enum io_state` definiert, der folgende Zustände besitzt:

```
enum io_state
{
    goodbit = _IOS_GOOD,
    eofbit = _IOS_EOF,
    failbit = _IOS_FAIL,
    badbit = _IOS_BAD
};
```

Der Status des Streams kann mit der Methode `io_state ios::rdstate()` abgefragt werden. Um die Abfrage der Bitmaske zu erleichtern, können die einzelnen Werte über spezielle Methoden abgefragt werden:

```
int ios::good() const ; // next operation might be successful
int ios::eof() const ; // end of file reached
int ios::fail() const ; // failbit or badbit is set, next operation will fail
int ios::bad() const ; // badbit is set, stream corrupted
```

Dabei gilt, dass die Zustände `good` oder `eof` darauf hinweisen, dass die letzte Operation erfolgreich war. Ist der Zustand `fail` oder `bad`, so ist die letzte Operation gescheitert. Wurde dabei versucht in eine Variable einzulesen, so bleibt deren Wert bei Scheitern der Operation normalerweise unverändert. Der Unterschied zwischen den Zuständen `fail` und `bad` ist nicht genau definiert; `fail`

bedeutet, dass der Stream selbst noch intakt ist, während im Zustand `bad` auch der Stream zerstört wurde. Hinweis: Laut [Hitz] ist die Abfrage des Zustandes `eof` unzuverlässig.

Einen bequemen Test, ob die letzte Operation erfolgreich war, ermöglicht der Operator `int ios::operator!()`, der `ios::fail()` entspricht. Die inverse Logik besitzt der Operator `ios::operator void*()`, der einen von 0 verschiedenen Wert nur annimmt, wenn der Stream im Zustand `good` ist. Damit sind Ausdrücke wie

```
...
if (!cin)
{
...
}
```

bzw. auch

```
while(cin)
{
...
}
```

möglich.

5.6 File Ein-/Ausgabe

5.6.1 ofstream und ifstream

Nachdem die bisher vorgestellten Methoden sich vorwiegend auf die Klassen `istream` und `ostream` bezogen, sollen nun die Methoden der hiervon abgeleiteten File-Streams `ofstream` und `ifstream` vorgestellt werden, die zur Ein-/Ausgabe in bzw. aus Files benötigt werden.

5.6.2 open und close

Zum Öffnen und Schließen von Files gibt es die Folgenden Möglichkeiten:

- `void ofstream::open(const char * filename, open_mode mode)` öffnet das File mit Namen `filename` im durch `mode` angegebenen Modus. Für den Modus sind definiert:

```
enum open_mode
{
    in = _IO_INPUT,           //open for reading
    out = _IO_OUTPUT,        //open for writing
    ate = _IO_ATEND,         //open and seek to eof
    app = _IO_APPEND,        //append always at eof
}
```

```

trunc = _IO_TRUNC,          //truncate file
nocreate = _IO_NOCREATE,    //fail if file does not exist
noreplace = _IO_NOREPLACE, //fail if file already exists
bin = _IOS_BIN, // Deprecated - ANSI uses ios::binary.
binary = _IOS_BIN          //do not replace CR by CR/LF (e.g., DOS)
};

```

- `void ofstream::close()` leert die Buffer und schließt das File. `close()` wird auch vom Destruktor automatisch aufgerufen.
- `void ifstream::open(const char * filename, open_mode mode)` öffnet das File mit Namen `filename`, analog wie bei `ofstream`.
- `void ifstream::close()` - siehe `ofstream::close()`.

Eine bequeme Methode, einen File-Stream zu instantiiieren und gleichzeitig dabei das File zu öffnen bieten folgende Konstruktor-Varianten:

- `ofstream::ofstream(const char * filename, open_mode mode)`
- `ifstream::ifstream(const char * filename, open_mode mode)`

Schlägt das Öffnen eines Files fehl, so kann dies am Zustand des Streams erkannt werden. Unter Verwendung des bereits besprochenen Operators `ios::operator!()` kann also beispielsweise mit `if(!file)` geprüft werden, ob ein Fehler beim Öffnen auftrat. Ob das zu einem Stream gehöriges File geöffnet ist, kann mit `is_open()` abgefragt werden.

5.6.3 Gleichzeitige Ein- und Ausgabe

Soll auf einen Stream gleichzeitig gelesen und geschrieben werden, so kann dies durch die gemeinsame Nutzung des `streambuf` von `istream` und `ostream` erreicht werden. Hierzu gibt es folgende spezielle Konstruktoren:

- `ostream::ostream(streambuf *)`
- `istream::istream(streambuf *)`

Somit kann nun beispielsweise die gleichzeitige Ein-/Ausgabe aus einem bzw. in ein File erreicht werden, indem man zunächst den `ofstream` (im Schreib- Lese-modus) konstruiert und danach einen `istream` instantiiert, der auf den Buffer des `ofstream` arbeitet. (Hinweis: ein passender `ifstream` kann nicht erzeugt werden, da dieser keinen Konstruktor besitzt, der einen Stream als Parameter akzeptiert.) Auf den `streambuf` des `ofstream` kann mit der Methode `streambuf * ios::rdbuf()` zugegriffen werden. Es ergibt sich also folgende Anweisungs-folge:

```

//open file
ofstream outfile("t.tmp",ios::out|ios::in) ;
if(!outfile)

```

```

    {
    cerr << "open failed for file 't.tmp'" << endl ;
    exit (-1) ;
    } // if
// construct istream from outfile's buffer
istream infile(outfile.rdbuf());

```

5.6.4 Beispiel

Nachfolgendes Beispiel zeigt die File- Ein-/Ausgabe einer Klasse. Hierbei wird auch oben beschriebenes Verfahren zur gleichzeitigen Ein- und Ausgabe verwendet.

```

//
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>

class persRecord
{
private:
enum {
fieldNameLen=15,
firstNameLen=256,
familyNameLen=256,
addressLen=512,
phoneLen=20};
char firstName[firstNameLen];
char familyName[familyNameLen];
char address[addressLen];
char phone[phoneLen];
public:

persRecord () {} ;

friend ostream & operator << (ostream & o, const persRecord &r)
{
o << "firstName[" << r.firstName << ']' << endl
<< "familyName[" << r.familyName << ']' << endl
<< "address[" << r.address << ']' << endl
<< "phone[" << r.phone << ']' << endl ;
return o ;
}

friend istream & operator >> (istream & i , persRecord & r)
{

```

```

const char fieldDelimiter = ']' ;
const char fieldNameDel = '[' ;
char dummy ;

//read firstName
i.ignore(r.fieldNameLen, fieldNameDel) ;
i.get(r.firstName,r.firstNameLen,fieldDelimiter) ;

//read familyName
i.ignore(r.fieldNameLen, fieldNameDel) ;
i.get(r.familyName,r.familyNameLen,fieldDelimiter) ;

//read address
i.ignore(r.fieldNameLen, fieldNameDel) ;
i.get(r.address,r.addressLen,fieldDelimiter) ;

//read phone
i.ignore(r.fieldNameLen, fieldNameDel) ;
i.get(r.phone,r.phoneLen,fieldDelimiter) ;

//remove last fieldDelimiter from stream
i.ignore() ;

return i ;
}

void interactiveInput ()
{
char c ;

//get firstname including whitespaces
cout << "enter firstName: " << endl ;
cin.get(firstName,firstNameLen,'\n') ;

//get familyName
cout << endl << "enter familyName: " << endl ;
//remove '\n' and other whitespaces
cin >> ws ;
cin.get(familyName,familyNameLen,'\n') ;

// get multiple lines for address
cout << endl << "enter address, terminate with '#': " << endl ;
//remove '\n' and other whitespaces
cin >> ws ;
cin.get(address,addressLen,'#') ;

```

```

//remove '#'
cin.get(c) ;

//get phone: use setw() to avoid overflow
cout << endl << "enter phone: " << endl ;
cin >> setw(phoneLen) >> phone;
}

} ; //class persRecord

main ()
{
persRecord pers ;

pers.interactiveInput() ;
cout << endl << pers << endl ;

//open file
cout << "opening file 't.tmp'..." << endl ;
ofstream outfile("t.tmp",ios::out|ios::in|ios::trunc) ;
if(!outfile)
{
cerr << "open failed for file 't.tmp'" << endl ;
exit (-1) ;
} // if

// construct istream from outfile's buffer
istream infile(outfile.rdbuf());

//write pers to file
cout << "writing persRecord..." << endl ;
outfile << pers ;

//construct empty persRecord
persRecord readPers ;

//read file
cout << "reading file..." << endl ;
//rewind
infile.seekg(0) ;
infile >> readPers ;

//display result
cout << "read:" << endl << readPers << endl ;

```

```

//read last character from file ('\n')
cout << "reading last char..." << endl ;
char c='E' ; //dummy-char

infile.get(c);

//should NOT fail
if (infile.fail())
{
cerr << "error reading char" << endl ;
}
else
{
cerr << "read was ok" << endl ;
}

if (infile.eof())
{
cerr << "eofbit is TRUE" << endl ;
}
else
{
cerr << "eofbit is FALSE" << endl ;
}

cout << "trying to read another one..." << endl ;
infile.get(c);

//should have failed
if (infile.fail())
{
cerr << "error reading char" << endl ;
}
if (infile.eof())
{
cerr << "eofbit is TRUE" << endl ;
}
else
{
cerr << "eofbit is FALSE" << endl ;
}
// last character read should be '\n'
cout << "read: " << c << endl ;
cout << "done." << endl ;
//close will be called by destructor
}

```

```
//
```

Das Programm liefert beispielsweise folgenden Dialog:

```
enter firstName:
myname

enter familyName:
family1 family2

enter address, terminate with '#':
mystreet mynumber
mylocality
mystate#

enter phone:
12345678

firstName[myname]
familyName[family1 family2]
address[mystreet mynumber
mylocality
mystate]
phone[12345678]

opening file 't.tmp'...
writing persRecord...
reading file...
read:
firstName[myname]
familyName[family1 family2]
address[mystreet mynumber
mylocality
mystate]
phone[12345678]

reading last char...
read was ok
eofbit is FALSE
trying to read another one...
error reading char
eofbit is TRUE
read:

done.
```

Wie man sieht, können für die Adresse mehrere Zeilen eingelesen werden, da als Delimiter '#' gewählt wurde. Nach dem Wiedereinlesen aus dem File wird noch der vom Output-Operator geschriebene Zeilenumbruch eingelesen. Ein danach folgender Leseversuch scheitert, da das Ende des Files erreicht wird. Das `eofbit` im Stream wird erst nach dem Leseversuch gesetzt, also wenn EOF bereits gelesen wurde. Die Abfrage *vor* dem letzten Leseversuch gibt daher keinen Hinweis darauf, dass das Fileende erreicht ist.

5.7 Buffer Ein-/Ausgabe

Neben der Ein-/Ausgabe in Files ist es oft nützlich, (vor allem) formatierte Ausgaben in einen Buffer vornehmen zu können. Hierzu werden die Klassen `istream` und `ostream` benutzt, die im Gegensatz zu `ifstream` bzw. `ofstream` keinen `filebuf`- sondern einen `stringstream`-Buffer verwenden (`filebuf` und `stringstream` sind beide von `streambuf` abgeleitet). Für `istream` und `ostream` kann durch unterschiedliche Konstruktoraufrufe zwischen automatischer Buffer-Verwaltung und der Verwendung eines beim Konstruktoraufruf als Parameter übergebenen Buffers gewählt werden:

- `ostream::ostream(char * buff, int n, int mode=ios::out)` verwendet `buff` zur Ausgabe. Die maximale Länge des Buffers ist `n`. Falls in `mode ios::ate` oder `ios::app` gesetzt sind, wird die Ausgabe an der Position des ersten Nullzeichens im Buffer fortgesetzt, die neue Ausgabe also an den bereits im Buffer befindlichen String angehängt.
- `ostream::ostream()` verwendet einen sich automatisch an die erforderliche Größe anpassenden Buffer.
- `istream::istream(const char * buff, int n)` liest die Eingabe aus `buff`. Der String in `buff` ist genau `n` Zeichen lang, also nicht nullterminiert.
- `istream::istream(const char * buff)` liest die Eingabe aus `buff`. Der String muss nullterminiert sein.

Bei beiden Klassen kann mit der Methode `const char * ostream::str()` bzw. `const char * istream::str()` lesend auf den Buffer zugegriffen werden. Bei beiden Klassen können natürlich alle in `ostream` bzw. `istream` definierten Methoden, etwa zur Positionierung, verwendet werden. Ein Beispiel zur Ein- und Ausgabe mittels `istream` und `ostream` ist nachfolgend aufgelistet:

```
//  
  
#include<iostream.h>  
#include<stringstream.h>  
  
int main ()  
{
```

```

const int buffsize=5 ;
char buff1[buffsize]={'1',' ','2',' ','3'} ;
char buff2[buffsize]={'4',' ','5','\0','\0'} ;
int a,b,c,d=-1 ;

istream in1(buff1,buffsize) ;

//read three ints from in1
in1 >> a >> b >> c ;
if (!in1)
{
cerr << "error in istream reading a...c" << endl ;
}

in1 >> d ;
// should fail : no more characters in buffer
if (!in1)
{
cerr << "error in istream reading d" << endl ;
}

//d should be unchanged
cout << "read: " << a << ' '
<< b << ' ' << c << ' '
<< d << endl ;

//create new istream with zero-terminated buffer
istream in2(buff2) ;

in2 >> a >> b ;
if (!in2)
{
cerr << "error in istream reading a...b" << endl ;
}

//should fail : String is zero-terminated,
//no more characters to read
c = -1 ;
in2 >> c ;
if (!in2)
{
cerr << "error in istream reading c" << endl ;
}

//c should be unchanged

```

```

cout << "read: " << a << ' '
<< b << ' ' << c << endl ;

//create ostream
ostream out1(buff1,buffsize) ;

int i = 1234;
// write i to out1 and add string terminator '\0'
out1 << i << ends ;
cout << "write: " << out1.str() << endl ;

//use same buffer for append by using out2
//output will be written beginning from the string terminator
ostream out2(buff1,buffsize,ios::app) ;
out2 << 'A' ;
cout << "write: " << out2.str() << endl;

//should NOT fail :-)
if(!out2)
{
cerr << "error in out2" << endl ;
}

} //main

//

```

Die Programmausgabe lautet:

```

error in istrstream reading d
read: 1 2 3 -1
error in istrstream reading c
read: 4 5 -1
write: 1234
write: 1234A

```

Kapitel 6

Templates

Durch die Verwendung von Template-Klassen und Template-Funktionen bietet C++ die Möglichkeit, eine gewünschte Funktionalität unabhängig von einem konkreten Datentyp zu formulieren. Ein typisches Beispiel für dieses Stilmittel ist die Realisierung von Container-Klassen, wie etwa Listen oder Bäume. Dabei ist die Funktionalität der Verwaltung der Listen- bzw. Baumelemente unabhängig vom Datentyp des Elements, denn die Verwaltung der Listenelemente ist für Listen von Elementen des Typs `char`, `long`, `int` etc. natürlich immer identisch, lediglich die Bezeichnung des Datentyps im Source-Code ist unterschiedlich. Durch die Verwendung eines Platzhalters (*Templates*) anstelle des konkreten Datentyps kann nun auf elegante Weise für obige Fälle eine generelle Formulierung gefunden werden.

6.1 Template-Klassen

Die Syntax der Definition einer Template-Klasse sei an nachfolgendem Beispiel vorgestellt:

```
template <class T>
class Array
{
    private:
        T * mData ; //array of T
        long mLength ;
        ...
    public:
        Array() : mData(0), mLength(0) {;} ;
        Array(long size) ;
        const T& get_element(long index) const ;
        ...
} ;
```

Durch `template <class T>` wird dem Compiler mitgeteilt, dass die nachfolgende Klassendefinition den Bezeichner `T` als Platzhalter für einen beliebigen Datentyp verwendet. Innerhalb der Klassendefinition kann dann `T` wie ein bekannter Datentyp verwendet werden. Bei der Implementierung von Klassenmethoden im Implementierungsfile müssen das `template`-Schlüsselwort sowie die Template-Argumente wiederholt werden, wie das nachfolgende Beispiel zeigt:

```
template <class T>
Array<T>::Array(long size) : mLength(size)
{
    ...
    mData = new T [size] ;
    ...
} ;

template <class T>
const T& Array<T>::get_element(long index)
{
    ...
} ;
```

Hierbei ist zu beachten, dass das Template-Argument `T` bei der Konstruktor-Definition nur *einmal* auftaucht, und nicht, wie man vielleicht erwarten könnte, zweimal, also

```
Array<T>::Array ()
```

und nicht

```
Array<T>::Array<T> ()
```

Bei der Instantiierung einer Template-Klasse muss nun als Typ-Argument der gewünschte Datentyp angegeben werden:

```
Array<double> my_array(100) ;
```

Wie man sieht, können Template-Klassen auch mit integralen Datentypen als Typ-Argument instantiiert werden, obwohl das `class` Schlüsselwort in `template <class T>` den Verdacht nahelegt, dies sei nur mit Datenklassen möglich.

Template-Klassen können auch ihrerseits wieder Template-Klassen als Typ-Argument verwenden:

```
Array<Array<double> > my_matrix ;
```

Hier ist darauf zu achten, dass zwischen der schließenden spitzen Klammer des inneren Typ-Arguments und der schließenden Klammer des äußeren Typ-Arguments ein Space vorhanden sein muss, da der Compiler ansonsten das Token "`>>`" mit `operator>>` verwechselt !

Es ist auch möglich, neben dem/den Typargument(en) auch konstante Ausdrücke als Template-Argumente zu übergeben. Im Beispiel des Array-Templates

könnte etwa die Größe des Arrays als konstanter Ausdruck mit übergeben werden:

```
template <class T, long size>
class Array
{
    private:
        T mData[size] ;
        ...
    public:
        T get_element(long index) ;
} ;

template <class T, long size>
Array<T,size>::get_element(long index) {...} ;
```

Bei der Instantiierung

```
Array<double,100> my_array ;
```

wird dann vom Compiler der Code für ein Array mit der Größe `long size = 100` erzeugt. Zu beachten ist, dass natürlich die gleiche Funktionalität durch eine entsprechende Größenangabe im Konstruktor erreicht werden kann, wie dies im vorherigen Beispiel gezeigt wurde.

Für die Instantiierung von Template-Klassen gilt: Zwei Template-Klassen sind dann vom gleichen Typ, wenn sowohl die Typ-Argumente vom gleichen Typ sind, als auch die konstanten Ausdrücke den gleichen Wert besitzen.

Generell sollte beachtet werden, dass mit jeder Instantiierung einer Template-Klasse für einen weiteren Datentyp vom Compiler auch der zugehörige Code aller Klassenmethoden für diesen Datentyp generiert wird. Sofern es sich hierbei nur um inline-Methoden handelt, ist dies weitgehend unkritisch, während bei komplexeren Methoden hingegen sehr viel Code erzeugt werden könnte. Um dies zu verhindern, kann man die Funktionalität der Template-Klasse und ihre Typabhängigkeit programmtechnisch trennen. Für eine Container-Klasse beispielsweise wird man innerhalb der komplexen Verwaltungsmethoden nicht mit Pointern auf den Platzhalter `T` arbeiten, so dass diese Methoden für jeden Datentyp vom Compiler dupliziert werden müssen, sondern man definiert zunächst eine Basisklasse für den Container, welche alle Verwaltungsmethoden beinhaltet. Diese Verwaltungsmethoden aber arbeiten stets mit `void`-Pointern (`void *`), wie das im nachfolgenden Beispiel für eine sehr einfache Liste angedeutet ist:

```
class ListBase
{
    ...
    public:
        ListBase() ;
    ...
}
```

```

        void insert(void *) ;
        void * get_next() ;
        ...
};

```

Die Verwendung des `void *` ist unbedenklich, da nur Daten des gleichen Typs in der Liste gehalten werden und die wünschenswerte Typsicherheit wie folgt sichergestellt wird: Von dieser Basisklasse wird nun eine Template-Klasse abgeleitet, welche in ihren Klassenmethoden die nötigen Casts auf den Datentyp `T` vornimmt. Diese Funktionen können, da sie sehr klein sind, alle als `inline` deklariert werden. Dadurch sind sie hinsichtlich der Code-Duplizierung vernachlässigbar:

```

template <class T>
class List : public ListBase
{
    ...
    public:
    List : ListBase() {...} ;
    ...
    void insert (const T * elem) {ListBase::insert((void*)elem);}
    T * get_next () { return ( (T*) ListBase::get_next()) ; }
    ...
};

```

Auf diese Weise verhindert man zwar die unnötige Duplizierung von Code, jedoch wird die Implementierung der Container-Klassen deutlich aufwendiger. Ob dieses Verfahren auf einen konkreten Anwendungsfall anzuwenden ist, sollte daher jeweils sorgfältig überlegt werden.

6.2 Template-Funktionen

Analog zu den Template-Klassen können auch Template-Funktionen definiert werden, deren Semantik generell formuliert ist und die dann je nach Typargument für einen konkreten Datentyp vom Compiler instantiiert werden. Als einfaches Beispiel sei hier die `swap()`-Funktion gezeigt:

```

template <class T>
void swap (T & a , T & b)
{
    T t(a) ;
    a = b ;
    b = t ;
}

```

Anzumerken ist, dass jedes Typargument der Template-Funktion auch in der Argument-Liste der Funktion auftauchen muss. Die Instantiierung der Template-Funktion erfolgt durch ihren Aufruf mit einem konkreten Datentyp:

```
int a = 3 ;
int b = 4 ;
swap(a,b) ;
```

Implizite Typenumwandlungen werden bei Template-Funktionen *nicht* vorgenommen: Der Aufruf

```
long a = 3 ;
int b = 4 ;
swap(a,b) ;
```

beispielsweise kann vom Compiler nicht aufgelöst werden, da die Funktion `swap()` in ihrer Argument-Liste zweimal den gleichen Typ `T` erwartet.

6.3 Template-Spezialisierungen

Sowohl für Template-Klassen als auch für Template-Funktionen können Spezialisierungen der allgemeinen Schablone für bestimmte Datentypen angegeben werden. Dies ermöglicht z.B. die Laufzeit- oder Speicherplatz-Optimierung für bestimmte Datentypen. Die Syntax der Spezialisierung lautet für Template-Klassen wie folgt:

```
class Array<char>
{
... //special implementation for char's
};
```

Es wird also statt des Schlüsselwortes `template <class T>` und des Platzhalters `T` direkt derjenige Datentyp angegeben, für den die spezielle Implementierung erfolgen soll.

Bei Template-Funktionen erfolgt die Spezialisierung, indem einfach eine Funktion definiert wird, die auf das Template passt:

```
void swap(Complex & a, Complex & b) {...} ;
```

Zu Beachten ist aber, dass *jede* Funktion, die auf die Template-Funktion passt, als Spezialisierung der Template-Funktion behandelt wird, auch wenn sie in einer anderen Übersetzungseinheit (z.B. einer Library) definiert wurde! Spezialisierungen werden also erst zur Link-Zeit erkannt. Bei der Definition von Template-Funktionen sollte man also gebräuchliche Namen tunlichst vermeiden!

Bei der Spezialisierung von Template-Klassen gilt ebenfalls, dass jede Spezialisierung (in irgendeiner Übersetzungseinheit) Vorrang vor der allgemeinen Implementierung hat. Eine Ausnahme bilden jedoch inline-Methoden. Diese darf der Compiler zur Compile-Zeit expandieren, auch wenn später noch eine Spezialisierung entdeckt werden sollte. Abhilfe gegen diese ungewollte Expansion kann geschaffen werden, indem sichergestellt wird, dass die *Deklaration* der spezialisierten Klasse im Quelltext *vor* der ersten Benutzung irgendeiner Klassenmethode dieser Klasse steht. Hierdurch wird erreicht, dass dem Compiler bereits vor der Expansion der inline-Funktion die spezialisierte Variante bekannt ist.

6.4 Zusammenfassung

- Templates dienen als verallgemeinerte Formulierung einer Funktionalität unabhängig vom konkreten Datentyp.
- Neben den Typ-Parametern ist auch die Angabe konstanter Ausdrücke in der Template-Argument-Liste möglich.
- Template-Instanziierungen erzeugen für jeden neuen Datentyp zusätzlichen Code.
- Durch die Trennung in eine (nicht Template-) Basisklasse und in eine davon abgeleitete Template-Klasse kann dies verhindert werden.
- Für jeden Datentyp sind Spezialisierungen möglich.
- Spezialisierungen von Template-Klassen und -Funktionen werden erst zur Link-Zeit erkannt. Sie haben immer Vorrang vor der allgemeinen Implementierung.
- Bei der Namesgebung von Template-Funktionen sollten daher gebräuchliche Namen vermieden werden.

Kapitel 7

Exceptions

In C++ bieten Exceptions die Möglichkeit, Ausnahmesituationen zuverlässig zu erkennen und zu behandeln. Im Gegensatz zu der Abfrage von Rückgabewerten, die vom Aufrufer einer Methode ignoriert werden können, kann das “Werfen” einer Exception vom Programm nicht “übersehen” werden. Das Werfen einer Exception geschieht innerhalb eines `try`-Blocks durch den Aufruf von `throw(T)`, wobei `T` ein beliebiges Objekt sein darf, welches z.B. Informationen über die aufgetretene Ausnahmesituation speichert. Sobald eine Exception-Klasse mit `throw()` geworfen wird, wird der nächste Exception-Handler (`catch`-Block) gesucht, der diese Exception-Klasse (oder eine ihrer Basisklassen) behandeln kann. Falls sich im Anschluss an den `try`-Block ein passender Exception-Handler findet, werden alle Anweisungen innerhalb dieses `catch`-Blocks ausgeführt und an das Ende aller dem `try`-Block folgenden `catch`-Blöcke verzweigt, wo die nächste Anweisung normal abgearbeitet wird. Findet sich kein passender Exception-Handler, so wird im call-Stack nach oben gegangen, d.h., es wird ein passender Exception-Handler innerhalb der nächsthöheren Aufrufebeine gesucht usw. Findet sich dabei selbst auf der Ebene von `main()` kein passender Exception-Handler, so wird die Funktion `terminate()` aufgerufen, die im Normalfall das Programm beendet.

Nachfolgendes Beispiel zeigt eine Funktion `except()`, die abhängig vom Wert des übergebenen Parameters drei unterschiedliche Exceptions wirft. Während die erste und zweite davon von den nachfolgenden `catch`-Blöcken gefangen wird, führt die dritte Exception zum Aufruf von `terminate()`: //

```
//except.cc
#include <iostream.h>

enum exceptionType{int_ex, long_ex, invalid_ex} ;

void except(exceptionType e)
{
    switch (e)
```

```

    {
    case int_ex :
        throw int(1) ;
        break ;
    case long_ex:
        throw long(2) ;
        break ;
    case invalid_ex:
    default:
        throw char('?') ;
    }//switch
}

main ()
{
    cout << "exception test:" << endl ;
    try
    {
        cout << "throwing integer..." << endl ;
        except(int_ex) ;
    }
    catch (long& l)
    {
        // this handler should not be entered
        cout << "got long exception: " << l << endl ;
    }
    catch (int& i)
    {
        cout << "got integer exception: " << i << endl ;
    }

    try
    {
        cout << "throwing long..." << endl ;
        except(long_ex) ;
    }
    catch (long& l)
    {
        cout << "got long exception: " << l << endl ;
    }
    catch (int& i)
    {
        cout << "got integer exception: " << i << endl ;
    }
}

```

```

try
{
    cout << "throwing unexpected exception type..." << endl ;
    except(invalid_ex) ;
}
catch (long& l)
{
    cout << "got long exception:" << l << endl ;
}
catch (int& i)
{
    cout << "got integer exception:" << i << endl ;
}

} //main

//

```

Das Programm liefert die Ausgabe:

```

exception test:
throwing integer...
got integer exception: 1
throwing long...
got long exception: 2
throwing unexpected exception type...
IOT trap/Abort

```

7.1 catch(...)

Um jede beliebige Exception fangen zu können, wird der Ausdruck `catch(...)` verwendet. Nun wird von unserem Beispiel auch die dritte Exception gefangen:

```

//
//exall.cc
#include <iostream.h>

enum exceptionType{int_ex, long_ex, invalid_ex} ;

void except(exceptionType e)
{
    switch (e)
    {

```

```

        case int_ex :
            throw int(1) ;
            break ;
        case long_ex:
            throw long(2) ;
            break ;
        case invalid_ex:
        default:
            throw char('?') ;
    }//switch
}

main ()
{
    cout << "exception test:" << endl ;
    try
    {
        cout << "throwing integer..." << endl ;
        except(int_ex) ;
    }
    catch (long& l)
    {
        // this handler should not be entered
        cout << "got long exception: " << l << endl ;
    }
    catch (int& i)
    {
        cout << "got integer exception: " << i << endl ;
    }

    try
    {
        cout << "throwing long..." << endl ;
        except(long_ex) ;
    }
    catch (long& l)
    {
        cout << "got long exception: " << l << endl ;
    }
    catch (int& i)
    {
        cout << "got integer exception: " << i << endl ;
    }
}

```

```

try
{
    cout << "throwing unexpected exception type..." << endl ;
    except(invalid_ex) ;
}

catch (long& l)
{
    cout << "got long exception:" << l << endl ;
}
catch (int& i)
{
    cout << "got integer exception:" << i << endl ;
}
catch (...)
{
    cout << "got any exception" << endl ;
}
cout << "end." << endl ;

} //main

```

```
//
```

Die Programmausgabe lautet:

```

exception test:
throwing integer...
got integer exception: 1
throwing long...
got long exception: 2
throwing unexpected exception type...
got any exception
end.

```

7.2 Rethrow

Falls eine Ausnahmesituation innerhalb eines `catch`-Blocks nicht behandelt werden kann, ist es möglich, die gleiche Exception mit `throw` ; (also ohne Argument) noch einmal zu werfen, so dass die Exception vom nächsthöheren Exception-Handler gefangen und behandelt werden kann. Im nachfolgenden Beispiel wird der Wert der Integer-Exception benutzt, um unterschiedliche Fehlerursachen darzustellen. Während die Werte `cause_a` und `cause_b` im nachfolgenden Catch-Block behandelt werden können, wird jede Exception mit einem anderen, nicht behandelbaren Wert erneut geworfen ("rethrow") : //

```
//rethrow.cc
```

```

#include <iostream.h>

enum exceptionCause{cause_a, cause_b, cause_c} ;

void except(exceptionCause c)
{
    switch (c)
    {
        case cause_a :
            throw (int(cause_a)) ;
            break ;
        case cause_b:
            throw (int(cause_b)) ;
            break ;
        case cause_c:
        default:
            throw (int(cause_c)) ;
    }//switch
};//except

void func(exceptionCause c)
{
    try
    {
        except(c) ;
    }
    catch (int c)
    {
        // identify cause
        switch (c)
        {
            case cause_a:
            case cause_b:
                //cause_a or cause_b can be handled here
                cout << "can handle cause a or b" << endl ;
                break ;
            default:
                //all exceptions specifying a different cause will be rethrown
                cout << "rethrowing exception that can not be handled" << endl ;
                throw ;
        }//switch
    }
};//func

main ()

```

```

{
    cout << "exception test:" << endl ;

    try
    {
        cout << "throwing cause a..." << endl ;
        func(cause_a) ;
    }
    catch (...)
    {
        // this handler should not be entered
        cout << "got an exception " << endl ;
    }

    try
    {
        cout << "throwing cause b..." << endl ;
        func(cause_b) ;
    }
    catch (...)
    {
        // this handler should not be entered
        cout << "got an exception" << endl ;
    }

    try
    {
        cout << "throwing cause c..." << endl ;
        func(cause_c) ;
    }
    catch (...)
    {
        // this handler will be entered
        cout << "got an exception that could not be handled" << endl ;
    }
}

```

```

} //main

```

```

//

```

Das Programm liefert die Ausgabe:

```

exception test:
throwing cause a...

```

```
can handle cause a or b
throwing cause b...
can handle cause a or b
throwing cause c...
rethrowing exception that can not be handled
got an exception that could not be handled
```

7.3 Klassenhierarchien und Exceptions

Falls eine Klasse B von der Klasse A abgeleitet wurde, dann werden von `catch(A& a)` auch alle Exceptions der Klasse B gefangen. Hierdurch kann eine Hierarchische Gliederung der Fehlerbedingungen erreicht werden, wie dies im nachfolgenden Beispiel gezeigt wird: //

```
//exhier.cc
#include <iostream.h>

enum exceptionType {mathError, underflow, overflow} ;

class MathError
{
public:
    MathError() {} ;
    virtual int get_info() const {return mathError ;} ;
};

class Underflow : public MathError
{
public:
    Underflow() {} ;
    //some special methods
    virtual int get_info() const {return underflow ;} ;
};

class Overflow : public MathError
{
public:
    Overflow() {} ;
    virtual int get_info() const {return overflow ;} ;
};

void except(exceptionType e)
{
    switch (e)
    {
```

```

    case mathError:
        throw MathError() ;
        break ;
    case underflow:
        throw Underflow() ;
        break ;
    case overflow:
        throw Overflow() ;
        break ;
    default:
        throw MathError() ;
} //switch
}

main ()
{
    cout << "exception test:" << endl ;

    try
    {
        cout << "throwing Underflow exception type..." << endl ;
        except(underflow) ;
    }
    catch (const Underflow & u)
    {
        cout << "got underflow exception: " << u.get_info() << endl ;
    }
    catch (const MathError & m)
    {
        //this handler should not be entered
        cout << "got MathErr exception:" << m.get_info() << endl ;
    }

    try
    {
        cout << "throwing Overflow exception type..." << endl ;
        except(overflow) ;
    }
    catch (const Underflow & u)
    {
        cout << "got underflow exception: " << u.get_info() << endl ;
    }
    catch (const MathError& m)
    {
        //this handler will handle all other MathError-Exceptions

```

```

        cout << "got MathError exception:" << m.get_info() << endl ;
    }

} //main

//

```

Da die `catch`-Blöcke nacheinander getestet werden, werden alle Ausnahmen vom Typ `Underflow` getrennt behandelt, während eine `Overflow`-Exception in der Standard-Behandlung für `MathError`-Exceptions gefangen und behandelt wird. Zu beachten ist, dass die Objekte *by reference* gefangen werden. Dies verhindert, dass z.B. bei der Übergabe an den zweiten `catch`-Block ein "slicing" erfolgt, so dass innerhalb des Blocks nur noch die Teile der Basisklasse sichtbar wären und somit auch fälschlicherweise die virtuelle Funktion der Basisklasse aufgerufen würde.

7.4 Einsatz von Exceptions

Allgemein gilt die Grundregel: Ausnahmebehandlungen sollen ausschliesslich für Ausnahmesituationen reserviert werden. Exceptions dienen nicht dazu, Returnwerte zu ersetzen! Der Vorteil der Exceptions, den "Gut-Fall" und den Fehlerfall programmtechnisch übersichtlich zu trennen, verkehrt sich hier ins Gegenteil: Werden Exceptions als Rückgabewerte mißbraucht, ist der Programmfluss des "Gut-Falles" nicht mehr leicht zu durchschauen, da der normale Programmfluss nun durch die Exception unterbrochen wird. Da weiterhin bei der Ausnahmebehandlung der Stack aufgerollt wird, wobei u.U. zahlreiche Blöcke spontan verlassen werden, erhöht sich die Gefahr von Speicherlöchern erheblich, falls innerhalb der Blöcke Speicher allociert wurde. Darüber hinaus sollte ebenfalls bedacht werden, dass die Abarbeitung einer Ausnahmebehandlung deutlich langsamer erfolgt als das Auswerten eines `return`-Statements.

7.5 Throw by value / catch by reference

Eine wichtige Eigenschaft des `throw`-Statements ist, dass die Exception-Objekte immer *by value* geworfen werden, dass also immer zuerst eine Kopie des Objekts erstellt wird, bevor es geworfen wird. Dies stellt sicher, dass nicht automatisch erzeugte Objekte per Referenz an den Exception-Handler weitergegeben werden, denn der Sichtbarkeitsbereich des Objekts wird durch das `throw`-Statement ja verlassen, so dass eine Referenz auf eine automatische Variable ungültig wäre (vgl. hierzu Kapitel 2.6.4). Eine Ausnahme hiervon ist der "rethrow" : hier wird das bereits gefangene Objekt ohne eine Kopie zu erstellen erneut geworfen.

Wie bereits erwähnt, ist es ratsam, das Ausnahme-Objekt *by reference* zu fangen. Zum Einen wird das "slicing" des Objektes umgangen, zum Anderen

wird eine weitere, unnötige Kopie (neben der bereits beim Werfen des Objektes erzeugten Kopie) vermieden.

7.6 Typumwandlung bei catch

Eine implizite Typumwandlung (z.B. von `int` zu `double`) findet beim Fangen von Exceptions nicht statt. Wie schon erwähnt, wird jedoch von einem `catch`-Block, der eine Klasse `A` fängt, auch jede von `A` abgeleitete Klasse gefangen. Als weitere implizite Typumwandlung werden von einem `void *` auch alle anderen Pointer gefangen. Wichtig hierbei ist, dass die Exceptions in der Reihenfolge der `catch`-Blöcke gefangen werden. Ein spezialisierter `catch`-Block muss also immer *vor* einem generalisiertem `catch`-Block stehen, sonst werden alle Exceptions vom generalisierten Exception-Handler gefangen!

7.7 Exception-Spezifikation und unexpected()

Für jede Funktion kann in C++ angegeben werden, ob und wenn ja, welche Exceptions sie wirft. Dies erfolgt durch die Exception-Spezifikation, die nach der Parameterliste der Funktion angegeben wird:

```
void func(int i) throw (int, double) {...};
```

Falls die Funktion `func()` eine Exception wirft, die nicht in der Exception-Spezifikation aufgeführt ist, wird automatisch der `unexpected`-Handler aufgerufen, der normalerweise die Funktion `terminate()` aufruft. Dies geschieht unabhängig davon, ob übergeordnete `catch`-Blöcke die Exception gefangen hätten oder nicht! Der Compiler seinerseits warnt natürlich davor, wenn innerhalb der Funktion eine Exception explizit geworfen wird, die in der Spezifikation nicht aufgeführt wurde. Problematisch ist jedoch, dass natürlich auch von Funktionen, die von `func()` aufgerufen werden, Exceptions geworfen werden können. Und diese Funktionen besitzen unter Umständen gar keine Exception-Spezifikation! In diesem Fall kann es dann unerwartet zur Terminierung des Programms kommen, wenn von diesen Funktionen eine Exception geworfen wird, die nicht in der Spezifikation des Aufrufers enthalten ist. Um anzuzeigen, dass eine Funktion keine Exceptions wirft, kann eine leere Exception-Spezifikation angegeben werden:

```
void func(int i) throw() {...};
```

Hier dürfen nun innerhalb von `func` nur noch Funktionen aufgerufen werden, die ihrerseits eine leere Exception-Spezifikation besitzen. Funktionen, die keine Exception-Spezifikation besitzen, sind dann nicht mehr zugelassen.

Eine Möglichkeit, die Terminierung des Programms bei Auftreten einer nichtspezifizierten Exception (beispielsweise `bad_alloc`, welche von `new` geworfen wird) zu verhindern ist es, mittels der Methode `set_unexpected` den `unexpected`-Handler selbst so zu definieren, dass er die Exception in eine bekannte default-Exception umwandelt:

```

class MyDefaultException
{
    ...
};

void convertToDefault {throw MyDefaultException();};

set_unexpected(convertToDefault) ;

```

Sofern nun jede Exception-Spezifikation die Klasse `MyDefaultException` beinhaltet, treten keine unerwarteten Terminierungen durch mangelhafte Exception-Spezifikationen mehr auf.

Definiert man `convertToDefault` wie folgt

```

void convertToDefault(throw ;) ;

```

und setzt den `unexpected`-Handler wieder wie oben beschrieben, so wird anstelle des `rethrows` innerhalb des `unexpected`-Handlers vom Compiler stets die Exception `bad_exception` geworfen. Diese kann dann analog zu `MyDefaultException` in jede Exception-Spezifikation aufgenommen werden, was wieder den gewünschten Effekt erreicht.

7.8 Laufzeitverhalten von Exceptions

Exception-Handling ist relativ langsam, sowohl im “Gut-Fall” aufgrund des von den `try`- und `catch`-Blöcken erzeugten Overheads als auch im Fehlerfall durch die Ausführung des `catch`-Blocks. Da jedoch Exceptions nur in Ausnahmefällen auftreten, ist der zweite Fall normalerweise unkritisch. Ob der Runtime-Overhead im “Gut-Fall” erträglich ist, kann nur durch Benchmarks für den jeweiligen Compiler ermittelt werden.

7.9 Exceptions und Speicherlöcher

Besondere Aufmerksamkeit verdient bei der Verwendung von Exceptions die Gefahr von Speicherlöchern. Da das Werfen von Exceptions den normalen Programmfluss unterbricht, werden eventuell mit `new` erzeugte Speicherbereiche nicht mehr freigegeben! Dies gilt insbesondere für Exceptions innerhalb von Konstruktoren. Da ein Konstruktor keinen Returnwert zurückgeben kann, ist es naheliegend, das Auftreten eines Fehlers durch das Werfen einer Exception anzuzeigen. Dabei ist zu beachten, dass bereits allocierter Speicher zuvor wieder freigegeben werden muss! Insbesondere gilt, dass für das Objekt, das gerade konstruiert wurde, als die Exception auftrat, *kein* Destruktor aufgerufen wird, da der Destruktor immer nur für vollständig initialisierte Objekte aufgerufen wird. Das Freigeben bereits allocierter Speicherbereiche muss also vom Programmierer selbst vorgenommen werden. Hierzu bietet es sich an, im Konstruktor in einem `catch(...)`-Block alle evtl. auftretenden Exceptions zu fangen, die

Aufräumarbeiten vorzunehmen und danach die gewünschte Exception zu werfen. Für non-pointer-Member-Objekte, die bereits über die Initialisierungsliste vollständig initialisiert wurden, wird natürlich der jeweilige Destruktor korrekt aufgerufen, wenn im Konstruktor der umschliessenden Klasse eine Exception auftritt.

Ein weiteres Problem ergibt sich, wenn während eines Destruktor-Aufrufes eine Exception geworfen wird: Vereinbarungsgemäss wird `terminate()` aufgerufen, wenn während der Abarbeitung einer Exception eine weitere Exception auftritt. Nun wird aber während der Ausnahmebehandlung der Stack aufgerollt, wodurch auch mehrere Destrukturen aufgerufen werden können. Sollte einer dieser Destrukturen nun seinerseits eine Exception werfen, so wird natürlich `terminate()` aufgerufen und das Programm beendet. Einzige Abhilfe ist, innerhalb des Destruktors jede auftretende Exception abzufangen. Innerhalb dieses `catch(...)`-Blocks sollte jedoch keine Aktion erfolgen, um zu verhindern, dass hierdurch nicht wiederum Exceptions hervorgerufen werden. Es ergibt sich folgendes Konstrukt:

```
try
{
... // destruktion-code
} // try

catch(...) {} ; // no action, catch all exceptions
```

Diese einzeilige Anweisung verhindert, dass durch evtl. auftretende Exceptions die `terminate()`-Funktion aufgerufen wird. Auch im Normalfall, also falls der Destruktor nicht im Rahmen einer Ausnahmebehandlung aufgerufen wird, ist es unschön, ihn mit einer Exception zu verlassen, da ja u.U. noch nicht alle allocierten Ressourcen wieder freigegeben wurden und somit Speicherlöcher entstehen könnten.

7.10 Zusammenfassung

- Exceptions sorgen für eine übersichtliche Trennung von “Gut-Fall” und Ausnahmefall.
- Exceptions können nicht übersehen werden.
- Exceptions können durch Bildung von Klassenhierarchien in generelle und spezielle Probleme gegliedert werden.
- Exceptions sind Ausnahmefälle und sollten nicht als Rückgabewerte mißbraucht werden, da sie den normalen Programmfluss unterbrechen.
- Die Verwendung von Exception-Spezifikationen erfordert eine besonders umsichtige Programmierung.
- Exceptions können Speicherlöcher erzeugen.

- Exceptions werden by value geworfen und sollten by reference gefangen werden.

Kapitel 8

Literatur

Nachfolgend sei eine kurze Aufstellung empfehlenswerter Literatur gegeben:

- Martin Hitz, C++ Grundlagen und Programmierung, Springer Verlag Wien New York, ISBN 0-387-82415-4
- Scott Meyers, Effective C++, Addison-Wesley, ISBN 0-201-56364-9
- Scott Meyers, More Effective C++, Addison-Wesley, ISBN 0-201-63371-X
- Robert B. Murray, C++ Strategies and Tactics, Addison-Wesley, ISBN 0-201-56382-7