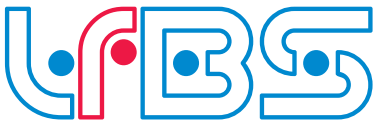
Praktikum Informatik 2 - Skript



Inhaltsverzeichnis

Literaturverzeichnis

Im folgenden Abschnitt finden Sie einen kleinen Auszug aus der zur Verfügung stehenden Literatur zu C++ und objektorientierter Programmierung.

Es sei hier ausdrücklich darauf hingewiesen, dass das Skript nur als Kurzreferenz dient und zur umfassenden Vorbereitung weitere Literatur zwingend erforderlich ist.

1. Stroustrup, Bjarne
   Die C++ Programmiersprache, 4. Auflage
   Addison Wesley 2000
   ISBN: 382731660X

2. Prinz, Peter
   C++- Lernen und professionell anwenden
   mitp 2008
   ISBN: 3826617649

3. Prinz, Peter
   C++- Das Übungsbuch.
   mitp 2007
   ISBN: 3826617657

4. Schildt, Herbert
   C++- Die professionelle Referenz
   mitp 2004
   ISBN: 3826613678

5. Josuttis, Nicolai
   Objektorientiertes Programmieren in C++
   Addison Wesley 2001
   ISBN: 3827317711

6. Breymann, Ulrich
   C++
   Hanser 2005
   ISBN: 3446402535

7. Kuhlins, Stefan
Die C++-Standardbibliothek. Einführung und Nachschlagewerk
Springer 2005
ISBN: 3540256938

8. Schneeweiß, Ralf
Moderne C++ Programmierung
Springer 2007
ISBN: 3540222812

Weitere Literaturhinweise finden Sie auf der Homepage (`http://www.lfbs.rwth-aachen.de/pi2`) unter Literatur und im Lehr- und Lernbereich (L^2P) der „Einführungsveranstaltung zum Praktikum“.

Dort finden Sie außerdem eine Auswahl von Links zu Online-Tutorien und Seiten mit grundlegenden und weiterführenden Informationen zu objektorientierter Programmierung unter C++.

1 Die Programmiersprache C++: Grundlagen

1.1 Lexikalische Konventionen

1.1.1 Token

Die Eingabe wird vom Compiler in sogenannte Token zerlegt, in C++ sind dies:

- Bezeichner (Identifier)
- Schlüsselworte (Keywords)
- Literale (Literals)
- Operatoren (Operators)
- Trennzeichen (Separators)

Leerzeichen, Tabulatoren, Zeilenenden und Kommentare werden ignoriert, sie trennen aber Token voneinander.

1.1.2 Kommentare

Ein Kommentar beginnt mit den Zeichen `/*` und endet mit `*/`.

```
/*
Diese Schreibweise fuer einen Kommentar ermoeglicht es den Text
ueber mehrere Zeilen zu verteilen.
*/
```

Solche Kommentare können nicht verschachtelt werden. Darüberhinaus lässt sich der Rest einer Zeile durch // als Kommentar kennzeichnen.

```
int i;     // Zähler
```

1.1.3 Bezeichner

Ein Bezeichner (engl: *identifier*) ist eine beliebig lange Folge von Buchstaben und Ziffern, wobei das erste Zeichen allerdings ein Buchstabe sein muss. Zu den Buchstaben wird auch das Underscore-Zeichen _ gerechnet. Bei Bezeichnern wird zwischen Groß- und Kleinschreibung unterschieden.

1.1.4 Schlüsselworte

In C++ gibt es folgende Schlüsselworte, die reserviert sind und nicht anderweitig benutzt werden dürfen:

```
asm     continue  float   new        signed    try
auto    default   for     operator   sizeof    typedef
break   delete    friend  private    static    union
case    do        goto    protected  struct    unsigned
catch   double    if      public     switch    virtual
char    else      inline  register   template  void
class   enum      int     return     this      volatile
const   extern    long    short      throw     while
```

Bei Schlüsselworten wird wie bei allen C++-Bezeichnern Groß- und Kleinschreibung unterschieden. Bei ASCII-Texten werden folgende Zeichen zur Interpunktion verwendet:

```
! % ^ & * ( ) - + = { } | ~ [ ] \ ; ' : " < > ? , . /
```

Außerdem folgende Zeichenkombinationen als Operatoren:

```
-> ++ -- .* ->* << >>  <=  >= == != &&
|| *= /= %= +=  -= <<= >>= %= ^= |= ::
```

1.1.5 Literale

Integer-Konstanten

Bei Integer (ganzzahligen)-Konstanten gibt es folgende Erscheinungsformen:

- Eine Zahl, die mit einer Ziffer ungleich 0 beginnt, ist eine Dezimalkonstante, z. B. `1234` oder `3990`.
- Eine Zahl, die mit `0` beginnt, ist eine Oktalkonstante, z. B. `015` oder `0377`.
- Eine Zahl, die mit `0x` beginnt, ist eine Hexadezimalkonstante, z. B. `0xffff` oder `0x5da7`.
- Der Typ einer Konstanten (siehe auch weiter hinten) richtet sich nach ihrem Wert und den entsprechenden Wertebereichen der Datentypen in der Implementation. Durch Angabe von `L` oder `U` hinter der Konstanten kann das aber auch explizit angegeben werden, z. B. `1999U (unsigned)` oder `0xabcdL (long)`.

Character-Konstanten

- Ein Zeichen, eingeschlossen in einfache Hochkommata ('x') ist eine Zeichenkonstante (Character-Konstante), nämlich der numerische Wert des Zeichens in der internen Codierung des Rechners (meistens `ASCII`). Beispiel: '0' hat den numerischen Wert 48 bei ASCII-Codierung.
- Folgende Sonderzeichen werden in Character-Konstanten verstanden:

`\0`	0-Byte (Stringende)
`\n`	Zeilenvorschub (Newline)
`\t`	Horizontaler Tabulator (Tab)
`\v`	Vertikaler Tabulator (Tab)
`\b`	Rückwärts löschen (Backspace)
`\r`	Zeilenende (Carriage Return)
`\f`	Seitenvorschub (Form Feed)
`\a`	Lärm (Bell)
`\?`	Fragezeichen (normalerweise einfach?)
`\'`	Hochkomma (Single Quote)
`\"`	Anführungszeichen (Double Quote)
`\`*nnn*	Oktaler Zeichencode
`\x`*hhh*	Hexadezimaler Zeichencode

Fließkomma-Konstanten

- Eine Zahl mit Vorkommastellen, dem Punkt ., Nachkommastellen und optional der Exponentenangabe mit e und Exponent ist eine Fließkommakonstante. Beispiel: `12.35` oder `1.602e-19` ($1.602 * 10^{-19}$).
- Der Typ einer solchen Konstanten ist double, durch Angabe von `F (float)` oder `L (long double)` kann der entsprechende Typ explizit angegeben werden, z. B. `0.25F`.

Bei Gleitkommazahlen bzw. Ausdrücken muss entweder die Exponentenschreibweise verwendet werden oder es muss ein Punkt enthalten sein, sonst kann es bei Ausdrücken zu unerwünschtem Verhalten kommen. So ergibt z. B. der Ausdruck 1/2 als Ergebnis `0`, da dies eine Integer-Division ist. Soll aber tatsächlich eine Fließkommadivision durchgeführt werden, so müssen die Operanden den entsprechenden Typ haben, im Beispiel führt $\frac{1.0}{2.0}$ zum gewünschten Ergebnis `0.5`.

C-String-Konstanten

- Eine Sequenz von Zeichen, eingeschlossen in Anführungszeichen, ist eine C-String-Konstante, z. B. `"abc"`.
- Der Typ einer C-String-Konstanten ist ein Array von `char`. Eine C-String-Konstante sollte nicht verändert werden.

Bei C-String-Konstanten können die gleichen Sonderzeichen mit \ eingegeben werden wie bei Character-Konstanten, z. B. `"Test.\n"`. Ein C-String wird vom Compiler durch ein abschließendes 0-Byte gekennzeichnet, `"abc"` wird im Speicher also als `'a' 'b' 'c' '\0'` abgelegt.

1.2 Grundlegende Konzepte

1.2.1 Fundamentale Datentypen

Character

`char`	„Normaler“ `char`-Typ
`signed char`	Vorzeichenbehaftet
`unsigned char`	Nicht vorzeichenbehaftet

Ob der Default-`char`-Typ vorzeichenbehaftet ist oder nicht, legt der Standard nicht fest. In sehr vielen Implementationen ist der Default aber `signed`.

Integer

`short`	Vorzeichenbehaftet
`int`	Vorzeichenbehaftet
`long`	Vorzeichenbehaftet

Alle diese Typen gibt es auch in einer `unsigned` Variante, z. B. `unsigned long`.

Fließkomma

`float`	Einfach genaue Fließkommazahl
`double`	Doppelt genaue Fließkommazahl
`long double`	Extra genaue Fließkommazahl

Der von den Datentypen belegte Speicherplatz und deren Wertebereich ist stark implementationsabhängig, für 32-Bit-Architekturen (z. B. SUN SPARC-Stations und PCs ab 386er) gelten im Regelfall die in Tabelle 1.1 angegebenen Werte.

Umwandlungen zwischen den Datentypen passieren normalerweise automatisch. Darüberhinaus existieren Cast-Operatoren, die eine explizite Typumwandlung erzwingen:

```
double x;
x = double(5);     // Passiert aber auch schon automatisch
x = (double) 5/6; // Erzwingt Fliesskommadivision
```

Typ	Größe (in Byte)	Wertebereich
Char/Integer mit Vorzeichen		
`char`	1	-128...127
`short`	2	-32768...32767
`int`	4	-2147483648...2147483647
`long`	4	-2147483648...2147483647
Char/Integer ohne Vorzeichen		
`unsigned char`	1	0...255
`unsigned short`	2	0...65535
`unsigned int`	4	0...4294967295
`unsigned long`	4	0...4294967295
Fließkomma		
`float`	4	$\approx 1.4 * 10^{-45}...3.4 * 10^{+38}$
`double`	8	$\approx 4.94 * 10^{-324}...1.8 * 10^{+308}$
Boolean		
`bool`	1	$true, false$

Tabelle 1.1: Typen in C++

Void

Der Datentyp `void` ist syntaktisch ein fundamentaler Datentyp. Er kann aber nur in Zusammenhang mit zusammengesetzten Datentypen verwendet werden, da es kein Objekt vom Typ `void` gibt. `void` wird benutzt um anzugeben, dass eine Funktion keinen Rückgabewert hat oder als Basistyp für einen Pointer auf ein Objekt mit unbekanntem Typ.

```
void f()     // Funktion liefert keinen Wert
void* p;     // Pointer auf Objekt mit unbekanntem Typ
```

Boolean

Logische Werte - sogenannte Wahrheitswerte - werden mit dem Typ bool dargestellt.

`bool`	true oder false

1.2.2 Zusammengesetzte Datentypen

C++ bietet folgende zusammengesetzte Datentypen, die aus den elementaren Typen gebildet werden können:

- Arrays (Felder) von Objekten
- Funktionen
- Pointer auf Objekte oder Funktionen

- Referenzen auf Objekte oder Funktionen
- Konstanten (siehe `const`)
- Klassen
- Strukturen
- Unions (Varianten)

Pointer (Zeiger)

Ein Pointer ist ein Datentyp, der die Adresse eines Objekts beinhaltet. Deklariert wird ein Pointer folgendermaßen:

```
TYPE* name;
```

`TYPE*` ist der Datentyp „Pointer auf `TYPE`“. `name` enthält die Adresse eines Objektes vom Typ `TYPE`. Es gibt zwei prinzipielle Operationen mit Pointern:

- *Dereferencing* (`*`)
  Zugriff auf das Objekt, auf das der Pointer zeigt.
- *Address of* (`&`)
  Referenzierung eines Objekts über einen Pointer.

Beispiel:

```
char c1 = 'A';
char* p = &c1;  // p enthält jetzt die Adresse von c1
char c2 = *p; // c2 enthält jetzt ebenfalls 'A'
```

Null-Pointer

Um zu kennzeichnen, dass ein Pointer eine ungültige Adresse hat, gibt es den speziellen Wert Null-Pointer. Der Null-Pointer wird in C++ durch das Symbol 0 angesprochen, 0 kann jedem Pointer zugewiesen werden.

```
char* p;
if (error)
  p = 0;    // Null-Pointer
else
  p = some_function();
```

Die Präprozessor-Konstante `NULL` wie aus C bekannt sollte aufgrund der engeren Typprüfung von C++ für diesen Zweck nicht benutzt werden.

Arrays

Arrays sind ein- oder mehrdimensionale Felder eines Datentyps, z. B. ein Array aus Integern. Felder werden über den Index-Operator `[]` indiziert.

```
double x;
double a[10]; // Ein Array mit 10 doubles, indiziert von 0..9
int k[5]; // Ein Array mit 5 ints,     indiziert von 0..4
a[0] = 0.0;   // Zuweisung an das Element
a[9] = 0.9;
k[3] = 125;
a[3] = k[3];
char matrix[10][20];  // char-Matrix aus 10 x 20 Elementen
matrix[0][0] = 'a'; // Zuweisung
matrix[0][10] = 'b';
matrix[9][0] = 'c';
```

Zu beachten: die Indizes eines mit n dimensionierten Arrays laufen immer von `0...n-1`. Die Indizierung wird nicht überprüft, so dass das Programm gegebenenfalls abstürzt. Je nach Entwicklungsumgebung kann eine Index-Überprüfung vom Debugger durchgeführt werden.

Pointer und Arrays

Der Zugriff auf ein Array kann auch über Pointer erfolgen, tatsächlich ist nämlich `a[i]` nichts anderes als `*(a+i)`.

```
int* p;        // Pointer auf int
int a[10];     // Array aus 10 ints
int i;
p = &(a[3]);   // p zeigt jetzt auf das Element a[3]
i = *p;        // *p ist der Inhalt von a[3]
p++;           // p zeigt jetzt auf a[4]
*p = i;        // a[4] erhält jetzt über p den Wert von i
```

Dies kann z. B. ausgenutzt werden, um dynamisch Arrays anzulegen:

```
int n = 20;
int i;
char* s;
s = new char [n];   // Speicherbereich mit n char allokieren
for(i = 0; i < n; i++)
  s[i] = '';        // Speicherbereich initialisieren
delete [] s;        // Speicherbereich freigeben
```

Referenzen

Referenzen sind vor allem für Funktionsargumente und -rückgabewerte sowie überladene Operatoren sinnvoll. Die Syntax für Referenzen ist ähnlich der von Pointern:

```
TYPE& name;
```

Zu beachten ist allerdings, dass Referenzen immer initialisiert werden müssen:

```
int a;
int& b;      // Ungültig, da keine Initialisierung!
int& c = a; // Gültig.
```

Eine Referenz ist ein alternativer Name für ein Objekt. Über die Referenz c im Beispiel kann hier das Objekt a angesprochen werden.

Strukturen

Strukturen sind zusammengesetzte Datentypen aus verschiedenartigen Elementen. Der Gebrauch von Strukturen soll an folgenden Beispielen erläutert werden:

```
// Beispiel für eine Strukturdeklaration:
struct Geburtsdatum
{
  char* name;
  int   tag, monat, jahr;
};

// Definition eines Objektes:
Geburtsdatum mm;
mm.name  = "Max Mueller";
mm.tag   = 29;
mm.monat = 2;
mm.jahr  = 1999;
```

Strukturen sind in C++ tatsächlich Klassen mit dem einzigen Unterschied, dass in Strukturen alle Mitglieder standardmäßig public sind (siehe auch Beschreibung Klassen in 2.1).

1.2.3 Konstanten und Aufzählungen

Benutzerdefinierte Konstanten drücken aus, dass sich Werte nicht direkt ändern. Symbolische Konstanten führen zu besser wartbarem Code als direkt im Code eingesetzte Literale. Das Schlüsselwort `const` kann der Deklaration eines Objekts zugefügt werden, wobei dadurch der Typ des Elements verändert wird.

Da eine Konstante nicht verändert werden darf, muss sie bei ihrer Definition initialisiert werden. Folgendes Beispiel verdeutlicht die Verwendung von benutzerdefinierten Konstanten:

Beispiel:

```
const double pi = 3.14159265;
const int array[] = { 0, 1, 2 ,3 };
const int i;     // Fehler, keine Initialisierung
pi = 25.9;       // Fehler, da Zuweisung
```

Bei Zeigern sind zwei Objekte beteiligt, der Zeiger und das Element auf den der Zeiger zeigt. Um den Zeiger konstant zu machen muss anstatt des *-Deklarator-Operators `*const` benutzt werden. Ein `const` vor dem Basistyp oder vor dem * bezieht sich immer auf den Basistyp.

Beispiel:

```
char* const c1;    // Konstanter Zeiger auf char
char const* c2;    // Zeiger auf konstanten char
const char* c3;    // Zeiger auf konstanten char
```

Wenn möglich sollten Konstanten auch bei der Parameterübergabe in Funktionen benutzt werden, damit man sicherstellen kann, dass der übergebene Parameter in der Funktion nicht verändert werden kann.

```
void print(const int num)
{
  num += 1; // Fehler: num ist konstant
  cout << num << endl;
}
```

Aufzählungstypen

Mit `enum` wird ein neuer Datentyp (Aufzählungstyp) definiert. Dann können Variablen dieses Typs anlegt werden. Damit kann

- einer Variablen Elemente des Aufzählungstyp zugewiesen werden, über dessen Namen.
- sicherstellt werden, dass eine Variable nur bestimmte Werte annehmen darf

Beispiele:

```
enum Farbe {rot, gelb, blau}; // =enum Farbe {rot=0, gelb=1, blau=2}
enum CentMuenzen {C01=1, C02, C05=5, C10=10, C20=20, C50=50};
```

Wenn man keine Werte angibt, beginnt die Aufzählung bei 0 und wird jeweils um 1 erhöht. Angegebene Werte müssen `int`-Konstanten sein. Fehlende Werte erhalten dann den nächsten freien Werte (z. B. `C02` erhält den Wert 2).

Variablen werden wie folgt angelegt:

```
Farbe eineFarbe=rot;
CentMuenzen eineMuenze;
```

Mit diesen Variablen kann man alle Berechnungen und Abfragen durchführen, die mit `int`-Variablen möglich sind.

Präprozessorkonstanten

Schließlich können auch in C-Manier Konstanten mit dem Präprozessor definiert werden:

```
#define PI 3.14159265
#define false 0
#define true  1
```

Die Verwendung von `const` und `enum` ist jedoch vorzuziehen.

1.2.4 Deklaration und Definition

Eine Deklaration macht einen Namen dem Programm bekannt. Eine Deklaration ist gleichzeitig auch eine Definition, es sei denn,

- sie deklariert eine Funktion ohne den Funktionsrumpf
- sie enthält die `extern`-Spezifikation und keine Initialisierung oder einen Funktionsrumpf
- sie ist eine Deklaration eines `static` Mitglieds einer Klasse
- sie ist eine Deklaration eines Klassennamens
- sie ist eine `typedef`-Deklaration

Beispiele für Definitionen:

```
int a;
int f(int x) { return x + a; }
struct S { int a; int b;};
enum { up, down };
```

Beispiele für reine Deklarationen:

```
extern int a;
int f(int);
struct S;
typedef int* pInteger;
```

1.2.5 Gültigkeitsbereich

In C++ gibt es vier Arten von Gültigkeitsbereichen:

- Lokal
  Ein Name, der lokal in einem Block deklariert wird, kann nur dort und nicht außerhalb benutzt werden. Formale Parameter einer Funktion werden so behandelt, als wären sie im äußeren Block der Funktion deklariert.
- Funktion
  Labels können überall innerhalb der Funktion benutzt werden, in der sie deklariert wurden. Den Funktionsgültigkeitsbereich gibt es nur für Labels.
- File
  Ein Name, deklariert außerhalb aller Blöcke und Klassen, hat den Gültigkeitsbereich des Files und kann überall nach seiner Deklaration benutzt werden. Diese Namen werden *global* genannt.
- Klasse
  Der Name eines Klassenmitglieds ist lokal in dieser Klasse und kann nur von einer Memberfunktion der Klasse, über den Punkt-Operator, den Pfeil-Operator, den Bereichsauflösungs-Operator (auch Scope-Operator genannt) oder von einer abgeleiteten Klasse benutzt werden.

Durch Deklaration eines gleichen Namens in einem eingeschlossenen Block oder einer Klasse wird der „äußere“ Name „versteckt“. Über den Scope-Operator kann man allerdings darauf auch zugreifen.

```
int x;          // Global

void some_function()
{
  int x;        // Versteckt globale Variable
  x = 1;        // Zuweisung an lokale Variable
  ::x = 1;      // Zuweisung an globale Variable über Scope-Operator
}
```

1.2.6 Speicherklassen

Es gibt drei fundamentale Speicherklassen in C++:

- Automatischer Speicher
  Hier werden lokale Variablen und Funktionsargumente abgelegt. Sie werden automatisch bei der Definition angelegt und am Ende des Gültigkeitsbereichs wieder gelöscht.
- Statischer Speicher
  Im statischen Speicher werden globale Variablen, Variablen von Namensbereichen, statische Klassenelemente und statische Variablen in Funktionen abgelegt. Derartige Objekte existieren und behalten ihren Wert während der gesamten Ausführung des Programms.

- Freispeicher
  Diese Speicherklasse wird benutzt, wenn Speicherplatz explizit mit `new` angefordert wird. Diese Art Speicher wird auch *dynamischer Speicher* oder *Heap* genannt und muss vom Programm selbst wieder explizit mit `delete` freigegeben werden.

1.2.7 Dynamische Speicherverwaltung mit `new/delete`

Gewöhnlich haben Objekte eine Lebensdauer, die durch ihren Gültigkeitsbereich bestimmt wird. Manchmal ist es jedoch sinnvoll, Objekte zu erzeugen, deren Lebensdauer vom aktuellen Gültigkeitsbereich unabhängig sind bzw. deren Art und Anzahl erst zur Ausführung des Programms bekannt ist. Dafür gibt es die Operatoren new und delete. Objekte, die durch new angelegt wurden, befinden sich im Freispeicher. Bei dynamischer Programmierung ist der Programmierer selbst für die Speicherverwaltung verantwortlich, d. h. Objekte, die mit new explizit angelegt werden müssen auch wieder explizit mit delete zerstört werden.

In C++ gibt es im Allgemeinen keine automatische Speicherbereinigung (garbage collection). Vergisst also der Programmierer, nicht mehr benötigte Objekte zu zerstören, dann wird dieser Speicher erst zum Programmende vom Betriebssystem wieder freigegeben. Diese Art der Speicherverschwendung wird auch als Speicherleck bezeichnet.

Dynamische Speicherverwaltung führt aber noch zu anderen Problemen. Man denke nur an den Fall, dass viele Referenzen auf das gleiche Objekt zeigen. Das referenzierte Objekt darf erst dann zerstört werden, wenn es keine Referenzen mehr auf dieses Objekt gibt. Wird auf ein versehentlich zerstörtes Objekt zugegriffen, dann gibt es meist die gefürchtete Speicherschutzverletzung. Es gibt jedoch Hilfsklassen, mit denen die Referenzen auf ein Objekt verwaltet werden können.

Trotz der Gefahren, die dynamisches Programmieren mit sich bringt, kann man darauf nicht verzichten. Die Notwendigkeit (speziell in C++) und Eleganz (zumindest für einen C++ Programmierer) wird hoffentlich beim Bearbeiten der Aufgaben einsichtig.

Beispiel für das Erzeugen/Zerstören eines Objekts:

```
int* p = new int;
delete p;
```

Beispiel für das Erzeugen/Zerstören eines Feldes von Objekten:

```
char* p = new char [10];
delete [] p;
```

Der Unterschied zwischen dem Anlegen/Zerstören von Objekten und Feldern ist unbedingt zu beachten. Wird ein `delete` auf ein Feld angewendet oder ein `delete[]` auf ein Objekt, bedeutet dies undefiniertes Verhalten, was meist zu einer Speicherverletzung oder, noch schlimmer, einer anderen Programmsemantik führt.

Der Aufruf von `delete` auf einen NULL-Zeiger ist jedoch unproblematisch.

1.3 Ausdrücke / Operatoren

C++ beinhaltet eine Vielzahl von Operatoren, die in Ausdrücken Werte verändern können. Tabelle 1.2 ist komplett aus dem Buch von Bjarne Stroustrup (siehe Literaturhinweise) übernommen und enthält eine Zusammenfassung aller Operatoren in C++. Für jeden Operator ist ein gebräuchlicher Name und ein Beispiel seiner Benutzung angegeben. Die hier vorgestellten Bedeutungen treffen für eingebaute Typen zu. Zusätzlich kann man Bedeutungen für Operatoren definieren, die auf benutzerdefinierte Typen angewendet werden (siehe Operatorüberladung).

`Klassenname` ist der Name einer Klasse, ein `Element` ein Elementname, ein `Objekt` ein Ausdruck, der ein Klassenobjekt ergibt, ein `Zeiger` ein Ausdruck, der einen Zeiger ergibt, ein `Ausdruck` ein Ausdruck und ein `Lvalue` ein Ausdruck, der ein nichtkonstantes Objekt bezeichnet.

Bereichsauflösung	`Klassenname :: Element`
Bereichsauflösung	`Namensbereichs-Name :: Element`
global	`:: Name`
global	`:: qualifizierter-Name`
Elementselektion	`Objekt.Element`
Elementselektion	`Zeiger->Element`
Indizierung	`Zeiger[Ausdruck]`
Funktionsaufruf	`Ausdruck(Ausdrucksliste)`
Werterzeugung	`Typ(Ausdrucksliste)`
Postinkrement	`Lvalue++`
Postdekrement	`Lvalue--`
Typidentifikation	`typeid(Typ)`
Laufzeit-Typinformation	`typeid(Ausdruck)`
zur Laufzeit geprüfte Konvertierung	`dynamic_cast<Typ>(Ausdruck)`
zur Übersetzungszeit geprüfte Konv.	`static_cast<Typ>(Ausdruck)`
ungeprüfte Konvertierung	`reinterpret_cast<Typ>(Ausdruck)`
const-Konvertierung	`const_cast<Typ>(Ausdruck)`
Objektgröße	`sizeof Objekt`
Typgröße	`sizeof(Typ)`
Präinkrement	`++Lvalue`
Prädekrement	`--Lvalue`
Komplement	`~Ausdruck`
Nicht	`!Ausdruck`
einstelliges Minus	`-Ausdruck`
einstelliges Plus	`+Ausdruck`
Adresse	`&Lvalue`
Dereferenzierung	`*Ausdruck`
Erzeugung (Belegung)	`new Typ`

continued on next page

continued form previous page	
Erzeugung (Belegung und Init.)	`new Typ(Ausdrucksliste)`
Erzeugung (Plazierung)	`new(Ausdrucksliste) Typ`
Erzeugung(Plazierung und Init.)	`new(Ausdrucksl.) Typ(Ausdrucksl.)`
Zerstörung (Freigabe)	`delete Zeiger`
Feldzerstörung	`delete [] Zeiger`
Cast (Typkonvertierung)	`(Typ) Ausdruck`
Elementselektion	`Objekt.*Zeiger-auf-Element`
Elementselektion	`Objekt->*Zeiger-auf-Element`
Multiplikation	`Ausdruck * Ausdruck`
Division	`Ausdruck / Ausdruck`
Modulo (Rest)	`Ausdruck % Ausdruck`
Addition	`Ausdruck + Ausdruck`
Subtraktion	`Ausdruck - Ausdruck`
Linksschieben	`Ausdruck « Ausdruck`
Rechtsschieben	`Ausdruck » Ausdruck`
Kleiner als	`Ausdruck < Ausdruck`
Kleiner gleich	`Ausdruck <= Ausdruck`
Größer als	`Ausdruck > Ausdruck`
Größer gleich	`Ausdruck >= Ausdruck`
Gleich	`Ausdruck == Ausdruck`
Ungleich	`Ausdruck != Ausdruck`
Bitweises Und	`Ausdruck & Ausdruck`
Bitweises Exklusiv-Oder	`Ausdruck ^ Ausdruck`
Bitweises Oder	`Ausdruck | Ausdruck`
Logisches Und	`Ausdruck && Ausdruck`
Logisches Oder	`Ausdruck || Ausdruck`
Bedingte Zuweisung	`Ausdruck ? Ausdruck : Ausdruck`
Einfache Zuweisung	`Lvalue = Ausdruck`
Multiplikation und Zuweisung	`Lvalue *= Ausdruck`
Division und Zuweisung	`Lvalue /= Ausdruck`
Modulo und Zuweisung	`Lvalue %= Ausdruck`
Addition und Zuweisung	`Lvalue += Ausdruck`
Subtraktion und Zuweisung	`Lvalue -= Ausdruck`
Linksschieben und Zuweisung	`Lvalue «= Ausdruck`
Rechtsschieben und Zuweisung	`Lvalue »= Ausdruck`
Und und Zuweisung	`Lvalue &= Ausdruck`
Oder und Zuweisung	`Lvalue |= Ausdruck`
Exklusiv-Oder und Zuweisung	`Lvalue ^= Ausdruck`
Ausnahme werfen	`throw Ausdruck`
Komma (Sequenzoperator)	`Ausdruck, Ausdruck`

Tabelle 1.2: Zusammenfassung der Operatoren

Hinweise

- Jeder Kasten enthält Operatoren gleicher Priorität. Die Operatoren in den oberen Kästen haben höhere Priorität als die in den unteren. Beispielsweise bedeutet `a+b*c` das gleiche wie `a+(b*c)`, da `*` eine höhere Priorität hat als `+`.
- Einstellige Operatoren und Zuweisungsoperatoren sind rechtsbindend, alle anderen linksbindend. Beispielsweise bedeutet `a=b=c` das gleiche wie `a=(b=c)` und `*p++` bedeutet `*(p++)`.
- Die Ergebnistypen von arithmetischen Operatoren werden nach einer Menge von Regeln bestimmt, die als die „üblichen arithmetischen Konvertierungen“ bekannt sind. Das generelle Ziel ist es, ein Resultat des „größten“ Operandentyps zu erzeugen. Beispielsweise ergibt `double+int` einen `double`-Wert oder `int*long` ergibt einen `long`-Wert.
- Die Reihenfolge der Auswertung von Teilausdrücken innerhalb eines Ausdrucks ist undefiniert. Speziell kann man nicht erwarten, dass ein Ausdruck von links nach rechts ausgewertet wird. Beispielsweise ist es undefiniert ob bei dem Code

  ```
  int x = f(2) + g(3);
  ```

  zuerst `f()` oder zuerst `g()` aufgerufen wird. Der Grund dafür ist, dass manche Compiler dadurch besser optimieren können.

1.4 Anweisungen

Es folgt eine Zusammenfassung aller Anweisungen, die es in C++ gibt. Sie ist aus dem Buch von Bjarne Stroustrup (siehe Literaturhinweise) übernommen.

```
Anweisung:
  Deklaration
  { Anweisungsliste_opt }
  try { Anweisungsliste_opt } Handler-Liste
  Ausdruck_opt;

  if(Bedingung) Anweisung
  if(Bedingung) Anweisung else Anweisung
  switch(Bedingung) Anweisung

  while(Bedingung) Anweisung
  do Anweisung while(Bedingung);
  for(for-Init-Anweisung; Bedingung_opt;Ausdruck_opt) Anweisung

  case Konstanter-Ausdruck: Anweisung
  default: Anweisung

  break;
  continue;
  return Ausdruck_opt

  goto Bezeichner;
  Bezeichner: Anweisung
```

```
Anweisungsliste:
  Anweisung Anweisungsliste_opt

Bedingung:
  Ausdruck
  Typangabe Deklarator=Ausdruck

Handler-Liste:
  catch(Ausnahme-Deklaration) { Anweisungsliste_opt }
  Handler-Liste Handler-Liste_opt
```

Man beachte, dass eine Deklaration eine Anweisung ist und dass Zuweisungen und Funktionsaufrufe Ausdrücke sind. Die Anweisung zur Handhabung von Ausnahmen, `try`-Blöcke, sind im entsprechenden Kapitel zur Ausnahmebehandlung beschrieben. Alle anderen Anweisungen werden in den folgenden Kapiteln beschrieben.

1.4.1 Deklarationen als Anweisungen

Eine Deklaration ist eine Anweisung. Der Grund, Deklarationen überall da zu erlauben, wo auch eine Anweisung erlaubt ist, liegt darin, es dem Programmierer zu ermöglichen, Fehler durch nicht initialisierte Variablen zu vermeiden und höhere Lokalität im Code zu erzielen. Konstanten werden dadurch erst möglich und schließlich ist es oft effizienter, die Definition zu verzögern, bis ein passender Initialisierer verfügbar ist (vor allem bei benutzerdefinierten Typen).

1.4.2 `if-else`-Anweisung

Bedingte Anweisung:

`if (` *expression* `)` *statement*

bzw. mit `else`-Teil:

`if (` *expression* `)` *statement*$_1$ `else` *statement*$_2$

Bei dieser Anweisung wird zunächst die Bedingung (*expression*) bewertet. Ist das Ergebnis von Null verschieden, so wird *statement*$_1$ ausgeführt, ansonsten wird in der `if-else`-Variante *statement*$_2$ ausgeführt.

Beispiel:

```
int a = 1;
int b = 2;
int z = 0;
if (a < b) {     // Bedingung in runden Klammern
  z = b;         // Geschweifte Klammern bei nur einer
}                // Anweisung eigentlich nicht notwendig
else
  z = a;         // else-Teil kann auch entfallen

if (a < b)
  z = 1;       // a < b  => z = 1
```

```
else if (a == b)  // Alternative Bedingung
  z = 0;          // a = b  => z = 0
else
  z = -1;         // a > b  => z = -1
```

1.4.3 `switch`-Anweisung

Bedingte Anweisung mit Ausführung abhängig vom Wert eines Ausdrucks.

```
switch ( expression )
{
      case  const-expr1:      statements
      case  const-expr2:      statements
      default:                statements
}
```

Für *expression* sind nur int- oder char-Ausdrücke zulässig. Normalerweise wird eine `switch`-Anweisung aus einem Block und mehreren Labels und Anweisungen bestehen. Wichtig ist, dass es sich um einen konstanten Ausdruck (*const-expr*) hinter dem Schlüsselwort `case` handelt (keine Verwendung von Variablen usw.).

Beispiel:

```
char c;

switch (c)
{
  // x wird durch * ersetzt
  case 'x':
  c = '*';
  break;
  // Vokale werden durch ? ersetzt
  case 'a':
  case 'e':
  case 'i':
  case 'o':
  case 'u':
    c = '?';
    break;
  default:  // Alle anderen werden durch Leerzeichen ersetzt
    c = ' ';
    break;
}
```

Beachten Sie, dass es sich bei `case` um eine Einsprungmarke handelt, von der aus der Programmablauf fortgesetzt wird. Insbesondere werden auch die Anweisungen der nachfolgenden `case`-Labels ausgeführt, solange bis die Ausführung der `switch`-Struktur durch ein `break`-Statement abgebrochen wird (sog. *fall-through*). Im Regelfall wird daher jeder Block hinter einem `case`-Label durch ein `break`-Statement abgeschlossen.

1.4.4 `while`-Anweisung

Bei dieser Schleifenstruktur wird zunächst der Ausdruck bewertet. Ist dieser wahr (ungleich Null), so wird die abhängige Anweisung ausgeführt. Nach der Ausführung wird der Ausdruck erneut bewertet. Dieser Vorgang wird so lange durchgeführt, bis die Bewertung des Ausdrucks falsch (gleich Null) ist. Dann wird mit der Anweisung, die hinter *statement* steht fortgefahren.

`while (` *expression* `)` *statement*

1.4.5 `for`-Anweisung

Die `for`-Schleife ist wie die `while`-Schleife kopfgesteuert, d. h. die Bedingung (hier: $expression_2$) wird vor dem ersten Ausführen der bedingten Anweisung geprüft. Zusätzlich enthält die `for`-Schleife einen Initialisierungsausdruck ($expression_1$), der vor der Schleife einmalig vor Beginn ausgeführt wird. Weiterhin einen Iterations-Ausdruck ($expression_3$), der zusätzlich am Ende jeder Ausführung der bedingten Anweisung durchgeführt wird.

`for (` $expression_1$`;` $expression_2$`;` $expression_3$ `)` *statement*

Eine `for`-Schleife entspricht also im Prinzip einer `while`-Schleife:

```
// Diese for-Schleife ist identisch ...
for (a; b; c)
  d;
// ... mit einer solchen while-Schleife
a;
while (b)
{
  d;
  c;
}
```

Beispiel für die Anwendung der for-Schleife:

```
double x[100];
int i;
for (i = 0; i < 100; i++)
  x[i] = 0.0; // Nullsetzen eines Feldes

int k = 5;
int fak = 1;
for (i = k; i > 1 ; i--)
    fak *= i;   // fak = Fakultät von k
```

1.4.6 `do-while`-Anweisung

Im Gegensatz zu den bisher genannten Schleifenkonstruktionen ist die `do-while`-Schleife fußgesteuert. Die bedingte Anweisung wird mindestens einmal beim Start der Schleife ausgeführt. Nach dieser Ausführung wird der Ausdruck *expression* ausgewertet. Ist das Ergebnis ungleich Null, so wird die bedingte Anweisung erneut durchgeführt.

`do` *statement* `while (` *expression* `);`

Man beachte das Semikolon am Ende der Konstruktion.

1.4.7 Sprunganweisungen

Wie bereits erwähnt, existieren in C++ mehrere Sprunganweisungen (engl: *jump-statements*), die vor allem bei den Schleifen und der `switch`-Anweisung zum Einsatz kommen. Durch den Einsatz der Sprunganweisungen wechselt der Programmablauf unbedingt.

- `break;`

  Dieser Sprung darf nur innerhalb einer Schleifenstruktur oder einer `switch`-Aweisung benutzt werden. In beiden Fällen wird die Struktur verlassen und mit der folgenden Anweisung fortgefahren. Sind mehrere solcher Strukturen ineinander verschachtelt, so bezieht sich die Sprunganweisung nur auf die kleinste umgebende Struktur.

- `continue;`

  `continue` darf nur in Schleifenstrukturen verwendet werden. In allen `while`-Schleifen wird die bedingte Anweisung verlassen und mit dem Ausdruck, der über Durchführung der Schleife entscheidet, fortgefahren. Wird die Sprunganweisung in einer `for`-Schleife verwendet, so wird ein Sprung zum Iterations-Ausdruck durchgeführt.

- `return` *opt-expression* `;`

  Die return-Anweisung wird benutzt, um die Ausführung einer Funktion zu beenden und zum Aufrufer zurückzukehren. Der optionale Ausdruck *opt-expression* wird dem Aufrufer zurückgeliefert. Erreicht der Programmablauf das Ende einer Funktion, so ist dies äquivalent zu einer `return`-Anweisung.

- `goto Bezeichner;`

  ```
  ...

  Bezeichner: Anweisung
  ```

  Sprung zu einer Anweisung, die durch eine Marke gekennzeichnet ist. Die Verwendung dieser unkontrollierten Sprünge sollte grundsätzlich vermieden werden. Wer also ein `goto` benutzt, sollte darüber mehrmals nachdenken. Im Praktikum soll dieser Sprungbefehl nicht verwendet werden.

1.4.8 `extern`-Anweisung

Die `extern`-Anweisung wird verwendet um eine bereits global definierte Variable in einer anderen Datei bekannt zu machen. Dies ist dann nur eine weitere Deklaration (wobei der Typ exakt übereinstimmen muss), und nicht eine neue Definition dieser Variablen.

```
Datei1.cpp:
  int x;
  double y = 10.0;

Datei2.cpp:
  double z;
  extern double y;

  z = ++y;     // z hat den Wert 11.
```

1.5 Funktionen

Eine Funktion ist eine Zusammenfassung von Anweisungen zu einer Einheit. Sie erhält Objekte eines bestimmten Typs als Argumente und liefert wiederum als Ergebnis ein Objekt eines bestimmten Typs.

Eine Funktionsdefinition in C++ spezifiziert den Namen der Funktion, den Typ des zurückgelieferten Objektes und die Typen und Namen der Argumente. Das Zurückliefern eines Wertes aus einer Funktion geschieht mit Hilfe der `return`-Anweisung. Die Funktion `main()` hat eine spezielle Bedeutung, sie wird beim Start des Programms aufgerufen.

```
long fak(int k)    // Returnwert long
{                  // Ein Argument int k
  ...
  return erg;      // Ergebnis
}

int main()         // Hauptprogramm
{
  long x;
  x = fak(5);      // Aufruf der Funktion
  return 0;        // Ende des Programms
}
```

Funktionsparameter werden in C++ normalerweise als Wert übergeben (*call by value*). Soll eine Referenz übergeben werden (*call by reference*), so muss dazu der Übergabe-Parameter als Pointer oder Referenz deklariert werden. Ausnahme bilden lediglich Arrays, die immer per *call by reference* übergeben werden.

Bei *call by value* wird eine Kopie des Werts an die Funktion übergeben. Dies hat zur Folge, dass Änderungen an den Parameter-Variablen nach dem Rücksprung aus der Funktion wirkungslos sind.

```
void test(int x, int y)
{
  x = 10;
  y = 20;      // x ist hier 10, y 20
}

int main()     // Hauptprogramm
{
  int wert1 = 15;
  int wert2 = 25;
  test(wert1, wert2); // Aufruf der Funktion
  // wert1, wert2 sind jetzt immer noch 15, 25
  return 0;
}
```

Möchte man Variablen per *call by reference* übergeben, so ist dies über Pointer möglich, aber das ist relativ umständlich:

```
void some_function(int* p) {
  *p = 99;
}

int main()
{
  int a;
  some_function(&a);
  // a hat jetzt den in some_function()
  // zugewiesenen Wert
}
```

In C++ kann dies einfacher über Referenzen realisiert werden. Das Beispiel mit der Funktion ließe sich mit Referenzen folgendermaßen umschreiben:

```
void some_function(int& i) {
  i = 99;
}

int main()
{
  int a;
  some_function(a);
  // a hat jetzt den in some_function()
  // zugewiesenen Wert
}
```

In C++ ist es außerdem möglich, mehrere Funktionen mit gleichlautendem Funktionsnamen zu versehen. Dieses „Überladen“ von Funktionsnamen unterscheidet der Compiler beim Aufruf und der Definition der Funktionen durch ihre Parametertypen.

```
double pos(double x) {       // Funktion für "double"-Argumente
  return (x < 0) ? -x : x;   // liefert Absolutbetrag
}

int pos(int x) {           // Funktion für "int"-Argumente
  return (x < 0) ? 0 : x;    // liefert für negative Werte 0
}

int main()
{
  pos(-1);     // pos(int)-Funktion wird aufgerufen(liefert 0)
  pos(-1.0);   // pos(double)-Funktion wird aufgerufen
               // (liefert 1.0)
}
```

Bei der Deklaration von Funktionen kann eine weitere Flexibilität auch durch *Default-Parameter* erreicht werden. Dabei wird einem Funktionsparameter standardmäßig ein Wert zugeordnet. Nur wenn die Funktion einen abweichenden Wert erhalten soll, muss dieser explizit angegeben werden.

```
void test(int x, int y = 1) { ... } // Standardwert y=1

int main()
{
  test(5);     // x=5; y=1
  test(7,4);   // x=7; y=4
}
```

Beachten Sie ggf. auftretende Mehrdeutigkeiten beim Überladen von Funktionen mit Default-Parametern. Default-Parameter können nur für hinten stehende Parameter benutzt werden.

1.6 Präprozessorkonzept

Vorteilhaft für die Programmentwicklung wirkt sich das Präprozessorkonzept der Sprache C/C++ aus. Zeilen, die im Sourcecode mit # eingeleitet werden, heißen Kontrollzeilen. Sie dienen der Kommunikation mit dem Präprozessor. Eine Kontrollzeile hat Auswirkungen auf die nach ihr stehenden Zeilen bis zum Ende der Datei. Sie werden üblicherweise am Anfang eines Sourcecodes eingefügt.

Eine Kontrollzeile der Form `#include <Dateiname>` bewirkt, dass der Präprozessor die Kontrollzeile durch eine Kopie des Inhalts der spezifizierten Datei ersetzt. Üblicherweise werden damit Headerdateien eingebunden. Gesucht wird in den voreingestellten Include-Verzeichnissen. Eine Kontrollzeile der Form `#include ''Dateiname''` führt zu einer ersten Suche im aktuellen Verzeichnis.

Mit der Direktive `#define` können symbolische Konstanten und funktionsähnliche Makros definiert werden. Es gilt folgende Syntax:

```
#define <Makroname> <Ersetzungtext>
```

Der Präprozessor ersetzt vor der Compilierung jedes Auftreten von `<Makroname>` im Sourcecode durch den `<Ersetzungstext>`. Einem Funktionsmakro können (ebenso wie einer Funktion) Parameter übergeben werden. Der Vorteil von Funktionsmakros besteht darin, dass keine Annahmen über den Typ der Parameter gemacht werden. Man kann also, im Gegensatz zu Funktionen, ein und dieselbe Definition für alle Typen verwenden.

Beispiel:

```
#include <iostream>
#include <iomanip>
/* Definition einer symbolischen Konstanten */
#define MfG "Mit freundlichen Gruessen"

/* Definition eines Funktionsmakros */
#define max(a,b) ((a > b) ? a : b)

void main()
{
  cout.setf(ios::fixed);
  cout.precision(2);

  /* Verwendung symbolischer Konstanten */
  cout << MfG << endl;

  /* Verwendung von Funktionsmakros */
  int xi = 10;
  int yi = 11;
  cout << "Maximum von " << xi << " und " << yi << " : "
       << max(xi, yi)) << endl;

  double xd = 10.0;
  double yd = 11.0;
  cout << "Maximum von " << xd << " und " << yd << " : "
       << max(xd, yd)) << endl;
}
```

Mit der Direktive `#undef` werden Makrodefinitionen wieder aufgehoben. Mit den bedingten Direktiven `#ifdef` und `#ifndef` lässt sich prüfen, ob ein Makro gegenwärtig definiert ist oder nicht. Hilfreich ist dies z. B. zur Verhinderung der Mehrfacheinbindung von Headerdateien.

Die Direktiven

```
#ifndef __MYFILE_H
  #include <myfile.h>
#endif
```

verhindern z. B. die Mehrfacheinbindung der Headerdatei myfile.h, da in dieser Headerdatei das Makro

`__MYFILE_H`

definiert wird:

```
#ifndef __MYFILE_H
  #define __MYFILE_H
  /* Deklarationen der Headerdatei */
#endif
```

Zusätzlich unterstützen die Direktiven wie `#if`, `#elif`, `#else` und `#endif` die bedingte Compilierung des Codes. Überprüft werden dabei Konstantenausdrücke (0 entspricht `FALSE` und alles andere entspricht `TRUE`).

2 Abstraktionsmechanismen in C++

Ein Problem bei der Entwicklung großer Programme besteht darin, dass die inneren Abhängigkeiten der Variablen im Programm nicht mehr überschaubar sind. In C versucht man dieses Problem durch Strukturieren der Programme in Module, die nur über ihre eindeutig definierten Schnittstellen kommunizieren, in den Griff zu bekommen.

C++ erweitert diese Möglichkeiten von C durch die Konzepte der objektorientierten Programmierung (OOP). Grundelement der objektorientierten Programmierung ist dabei die Zusammenfassung ggf. komplex strukturierter Daten und ihrer zugehörigen Methoden (Elementfunktionen) zu Klassen. Dabei kann dann gewöhnlich nur über Methoden auf die internen Datenstrukturen zugegriffen werden.

2.1 Klassen

Im Unterschied zu C können in C++ Datenstrukturen zusätzlich Elementfunktionen (sog. *Methoden*) zugeordnet werden. Diese dienen der Manipulation der in der Datenstruktur enthaltenen Variablen (sog. *data member*). Solche Datenstrukturen mit den zugehörigen Methoden nennt man Klassen. Statt des aus C bekannten Schlüsselworts `struct` für Datenstrukturen werden Klassen mit dem neuen Schlüsselwort `class` eingeleitet.

Die Verwendung von `struct` in C++ ist zwar abwärtskompatibel zu C, sollte aber in C++ nicht weiter benutzt werden. Die Erweiterung von C++ besteht darin, zusätzlich zu den Datenelementen noch Methoden zu definieren und den Zugriff genauer zu kontrollieren. Für die Zugriffskontrolle gibt es die Zugriffsspezifizierer `public` und `private`.

Beispiel:

```
class beispiel
{
  public:
    int i;
  private:
    int j;
  public:
    void reset();
};

void beispiel::reset()
{
  j = 0;
```

```
}

int main()
{
  beispiel Objekt;
  Objekt.j=1;    // Fehler, da j private
  Objekt.reset(); // Zugriff auf j nur über Funktion möglich
}
```

Während auf öffentliche (`public`) Datenelemente von jeder Stelle des Programms aus zugegriffen werden darf (im Beispiel i), dürfen auf die privaten Datenelemente nur die Klassenmethoden zugreifen (im Beispiel j). Zugriffsspezifizierer können in beliebiger Reihenfolge und auch mehrfach innerhalb der Klasse verwendet werden und sind bis zur Angabe des nächsten Zugriffsspezifizierers gültig.

Eine Vereinbarung einer Klasse mit dem Schlüsselwort `class` unterscheidet sich von einer mit `struct` nur darin, dass bei `class` dessen Elemente bei Fehlen eines Zugriffsspezifizierers `private` sind, bei `struct` aber `public`. Damit sind folgende Codesequenzen äquivalent:

```
struct <Klassenname>
{
  private:
  /* ... */
};
```

und:

```
class <Klassenname>
{
  /* ... */
};
```

auf der einen, sowie:

```
class <Klassenname>
{
  public:
  /* ... */
};
```

und:

```
struct <Klassenname>
{
  /* ... */
};
```

auf der anderen Seite.

In einer Klassendefinition sollten i. A. die Variablen `private` und die Methoden `public` definiert sein.

Die Erzeugung einer Variablen eines Klassentyps wird als *Instanziierung eines Objekts einer Klasse* bezeichnet. Ein Objekt (eine Instanz) ist also nichts anderes als eine Variable eines Klassentyps. Die Instanziierung von Objekten einer Klasse erfolgt analog zur Deklaration von Variablen der Standarddatentypen wie `int` oder `float` (vgl. Beispielprogramm). Zugriff auf die Mitglieder (Data Member und Methoden) eines Objektes erhält man über den *Punktoperator* (engl. *member dot operator*).

Konstruktoren und Destruktoren

Methoden, die automatisch bei der Instanziierung eines neuen Objekts zu einer Klasse aufgerufen werden, heißen *Konstruktoren*. Konstruktoren sorgen dafür, dass bei der Instanziierung eines Objekts alle Datenelemente initialisiert werden. Im Sourcecode erkennt man die Konstruktoren daran, dass sie denselben Namen wie die Klasse tragen.

Methoden, die automatisch beim Löschen von Objekten einer Klasse aufgerufen werden, heißen *Destruktoren*. Sie geben den vom Objekt belegten Speicher wieder frei. Sie tragen ebenfalls den gleichen Namen wie die Klasse, sind jedoch zusätzlich durch das Tildezeichen ~ gekennzeichnet.

Konstruktoren und Destruktoren haben keinen Rückgabetyp. Ein Destruktor hat auch keine Parameter. Einen Konstruktor ohne Parameter nennt man *Standardkonstruktor*. Er sollte zu jeder Klasse definiert sein.

```
// Klassenbeispiel_mit_Konstr_und_Destr.c
#include <iostream>

class Feld
{
  private:
    int p_iAnz;
    int* p_ptFeld;
  public:
    Feld();          // Standardkonstruktor
    Feld(int iVor); // Konstruktor
    ~Feld();         // Destruktor
};

Feld::Feld()
{
  p_iAnz = 0;
  p_ptFeld = 0;
  std::cout << "Leeres Feld" << std::endl;
}

Feld::Feld(int iAnz)
{
  int i;
  p_iAnz = iAnz;
  p_ptFeld = new int [iAnz];
  for (i = 0; i < iAnz; i++)
    p_ptFeld[i] = 0;
  std::cout << "Feld mit " << p_iAnz <<" Elem." << std::endl;
}
```

```
Feld::~Feld()
{
  delete [] p_ptFeld;
  p_iAnz = 0;
  std::cout << "Aufruf des Destruktors" << std::endl;
}

// Hauptprogramm
int main()
{
  std::cout << "Anfang" << std::endl;
  Feld Objekt1(3);
  std::cout << "Ende" << std::endl;
}
```

Anhand der Ausgabe ist zu erkennen, wann Konstruktor und Destruktor aufgerufen werden:

```
Anfang
Feld mit 3 Elem.
Ende
Aufruf des Destruktors
```

Im Konstruktor wird mit `new` Speicherplatz für die Feldelemente `Feld` reserviert und initialisiert. Der Destruktor gibt den reservierten Speicherplatz wieder frei. Sein Aufruf erfolgt automatisch, wenn der Gültigkeitsbereich des Objektes verlassen wird. Eine Initialisierung von Werten einer Klasse kann anstatt mit Zuweisung im Konstruktorrumpf auch mit einer sogenannten Konstruktorinitialisierungsliste geschehen. Ein Konstruktor der Form:

```
Klasse::Klasse(Typ1 a, Typ2 b)
{
  private_a = a;
  private_b = b;
  /* ... */
};
```

wird mittels Initialisierungsliste zu:

```
Klasse::Klasse(Type1 a, Typ2 b) : private_a(a), private_b(b)
{
  /* ... */
};
```

Die zweite Art der Initialisierung ermöglicht im Gegensatz zur ersten auch die Initialisierung von Konstanten. Datenelemente, denen dynamisch (also zur Laufzeit des Programms) Speicher zugewiesen wird, können allerdings nur im Rumpf des Konstruktors initialisiert werden. Als Richtlinie schlägt unter anderem Scott Meyers (siehe Literaturhinweise) vor, wann immer möglich die Konstruktorinitialisierungsliste zu benutzen. Bei mehreren Konstruktoren wird anhand von Anzahl und Typ der Parameter unterschieden, welcher Konstruktor aufgerufen wird (s. Kapitel 2.4).

Neben dem *Standardkonstruktor* (ohne Parameter) gibt es noch einen anderen ausgezeichneten Typ eines Konstruktors: den *Copykonstruktor*. Dieser hat als Parameter eine (konstante) Referenz auf seine Klasse.

Beispiel:

```
class Klasse
{
  Klasse();                 // Standardkonstruktor
  Klasse(const Klasse&);    // Copykonstruktor
};
```

Beim Copykonstruktor gilt aus historischen Gründen, dass er implizit existiert, wenn er nicht definiert wurde. Er kopiert dann Byte für Byte. Der Copykonstruktor wird automatisch immer dann aufgerufen, wenn eine Variable an eine Funktion/Methode nicht als Referenz übergeben wird. Dies kann unangenehme Folgen haben, wenn innerhalb der Klasse dynamische Datenstrukturen existieren. Stellen Sie sich hierzu einen Zeiger auf einen String vor. Wenn das Objekt byteweise kopiert wird, dann wird der Zeiger kopiert und nicht der Inhalt. Es existieren dann zwei Zeiger auf den gleichen String. Das Löschen eines der Objekte führt dann unter Umstände dazu, dass der String gelöscht wird und das verbleibende Objekt weiter auf den nicht mehr existierenden String verweist (*wilder Zeiger*). Dies ist gefährlich und sollte vermieden werden. Daher sollte man den Copykonstruktor immer dann selbst definieren, wenn die Klasse dynamische Elemente hat, oder den Aufruf eines Copykonstruktors ganz verbieten (d. h. Fehler zur Compilierzeit, wenn nicht mit Referenzen gearbeitet wird). Dazu kann man ihn in der Klasse als `private` deklarieren:

```
class Klasse
{
  private:
    Klasse(const Klasse&);
};
```

Der Parameter des Copykonstruktors muss übrigens eine Referenz sein, da ansonsten bei seinem Aufruf wieder kopiert würde, was zu einer unendlichen Rekursion führen würde.

Als Beispiel soll in einer einfachen Stringklasse der Copykonstruktor definiert werden:

```
class HString
{
  private:
    char* p_sString;
    int iSize;
  public:
    HString();
    HString(const HString&);
    /*...*/
};

HString::HString()
{
  iSize = 0;
  p_sString = 0;
```

```
}

HString::HString(const HString& aHString)
{
  iSize = aHString.iSize;
  p_sString = new char [iSize + 1];
  strcpy(p_sString, aHString.p_sString);
}
```

Konstante Elementfunktionen

Methoden können als konstant deklariert werden. Dazu muss hinter der Deklaration das Schlüsselwort `const` hinzugefügt werden.

Beispiel:

```
class Klasse
{
  public:
    int GetElement() const;
  private:
    int Element;
};

int Klasse::GetElement() const
{
  ++Element;     // Fehler, da Element verändert wird
  return Element;
}
```

Konstante Methoden dürfen den Zustand des Objektes nicht verändern, weswegen es im Beispiel an gekennzeichneter Stelle zu einem Fehler beim Compilieren kommt. Dadurch wird es ermöglicht, eine semantische Beschränkung zu spezifieren, die zur Compilierzeit geprüft wird. Der Aufwand konstante Methoden zu deklarieren ist meist sehr viel kleiner als Fehler zu finden, die durch ungewolltes Verändern eines Zustandes entstehen. Daher sollte man wann immer möglich Methoden als konstant deklarieren.

Statische Klassenelemente

Data Member einer Klasse können mit Hilfe des Schlüsselworts `static` als statisch deklariert werden. Von statischen Klassenmitgliedern wird nur *eine* Kopie erzeugt, unabhängig davon, wieviele Objekte der Klasse instanziiert werden. Eine statische Variable ist also der Klasse selbst zugeordnet und nicht den Objekten der Klasse. Auch Funktionen können statisch deklariert werden, wenn sie Zugriff auf Elemente einer Klasse benötigen, jedoch nicht für ein bestimmtes Objekt aufgerufen werden.

Da statische Data Member existieren, ohne dass ein Objekt der Klasse existieren muss, erfolgt ihre Initialisierung außerhalb der Klasse und außerhalb jeder Funktion (auch außerhalb von main). Zur besseren Übersicht sollte der Code dazu aber im Sourcefile der jeweiligen Klasse stehen. Bei der Initialisierung muss der Datentyp der Variablen wiederholt werden und es muss ein Wert zugewiesen werden.

Das nächste Programmbeispiel zeigt den Unterschied von statischen und nicht-statischen Variablen sowie die Benutzung einer statischen Funktion:

```
#include <iostream>

class beispiel_static
{
  public:
    beispiel_static()
    {
      iAnzahl++;
      iNummer = iAnzahl;
    }

    static void vPrintAnzahl()
    {
      std::cout << "Anzahl erzeugter Objekte: " << iAnzahl;
    }

    void vPrintNummer()
    {
      std::cout << "Nr. des erzeugten Objekts: " << iNummer;
    }

    void vPrint()
    {
      vPrintAnzahl(); std::cout << " ... ";
      vPrintNummer(); std::cout << endl;
    }
  private:
    static int iAnzahl;
    int iNummer;
};

// Initialisierung der statischen Variablen
int beispiel_static::iAnzahl = 0;

int main()
{
  beispiel_static::vPrintAnzahl(); cout << endl;
  beispiel_static a;
  a.vPrint();
  beispiel_static b;
  a.vPrint();
  b.vPrint();
}
```

Die Ausgabe dieses Programmes lautet:

```
Anzahl erzeugter Objekte: 0
Anzahl erzeugter Objekte: 1 ... Nr. des erzeugten Objekts: 1
```

```
Anzahl erzeugter Objekte: 2 ... Nr. des erzeugten Objekts: 1
Anzahl erzeugter Objekte: 2 ... Nr. des erzeugten Objekts: 2
```

Der `this`-Zeiger

Werden mehrere Objekte einer Klasse erzeugt, so sind nur die Data Member in jedem Objekt vorhanden. Die Methoden hingegen werden für jede Klasse nur einmal erzeugt, da ihr Code für jedes Objekt gleich ist. Eine Mehrfacherzeugung wäre unnötige Speicherverschwendung. Bei Aufruf einer Methode muss der Compiler daher wissen, für welches Objekt der Klasse die Methode ausgeführt werden soll. Dies wird erreicht, indem der Methode als erstes Argument immer die Adresse des jeweiligen Objektes übergeben wird. Diese Adresse wird als `this`-Zeiger bezeichnet. Die Übergabe geschieht automatisch und unsichtbar für den Programmierer. Beispielsweise wird die Implementation der Methode

```
void myClass::printName()
{
  std::cout << "Der Name ist: " << name << std::endl;
}
```

intern in

```
void printName(myClass* this)
{
  std::cout << "Der Name ist: "<< this->name << std::endl;
}
```

konvertiert. Ebenso wird der Aufruf der Funktion von

```
classObject.printName();
```

in das Konstrukt

```
printName(&classObject);
```

konvertiert. Anwendungsbeispiele zur Verwendung des `this`-Zeigers finden Sie in Kapitel 2.4.

2.2 Vererbung / Inheritance

Die Vererbung (engl.: *Inheritance*) ist ein Mechanismus, der es erlaubt, Datenstrukturen und Methoden einer Klasse (Basisklasse) bei der Bildung neuer Klassen (Unterklasse, abgeleitete Klasse) wiederzuverwenden. In der Unterklasse kann die Datenstruktur erweitert werden und/oder die Methoden erweitert oder modifiziert werden. Weiterhin stellt die Vererbung ein Hilfsmittel zur Strukturierung von Anwendungsproblemen dar, die im allgemeinen viele ähnliche Elemente beinhalten.

```
class <KlassenName>:<Zugriffsspezifizierer> <BasisklassenName>
```

Elementzugriff in der Basisklasse	Zugriffsspezifizierer der Klasse	Elementzugriff in der Ableitungsklasse
`private`	`public`	*nicht möglich*
`protected`	`public`	`protected`
`public`	`public`	`public`
`private`	`protected`	*nicht möglich*
`protected`	`protected`	`protected`
`public`	`protected`	`protected`
`private`	`private`	*nicht möglich*
`protected`	`private`	`private`
`public`	`private`	`private`

Tabelle 2.1: Zugriffsmöglichkeiten auf Basisklassenelemente

Mit der obigen Syntax wird eine Klasse vereinbart, die alle Datenelemente und Methoden einer bereits vorhandenen Klasse erbt. Der Zugriffsspezifizierer gibt an, ob Datenelemente und Methoden der Basisklasse innerhalb der abgeleiteten Klasse zugänglich sind. Hat die Basisklasse alle Datenelemente z. B. `public` deklariert, und der Zugriffsspezifizierer ist `private`, so gelten alle Datenelemente der Basisklasse für ein Objekt der abgeleiteten Klasse als `private` deklariert.

Mit der Vererbung wird neben `public` und `private` die Einführung eines weiteren Schlüsselworts zur Kennzeichnung von Zugriffsrechten auf Datenelemente und Methoden notwendig. Das Schlüsselwort heißt `protected`. Genau wie auf `private`-Mitglieder einer Klasse kann auch auf `protected`-Mitglieder nicht von außerhalb der Klasse zugegriffen werden, jedoch werden `protected`-Mitglieder an eine abgeleitete Klasse vererbt, `private`-Mitglieder nicht.

Tabelle 2.1 gibt einen Überblick über die Zugriffsmöglichkeiten auf Elemente von Basisklassen. In der Regel verwendet man für die Klasse den Zugriffsspezifizierer `public`, da dann der Zugriff auf Mitglieder der abgeleiteten Klasse in gleicher Weise möglich ist wie der Zugriff auf Mitglieder der Basisklasse. Beachte: Der Defaultwert ist `private`.

Abbildung 2.1 zeigt ein Beispiel für die Verwendung einer Klassenhierarchie anhand von Zeichnungselementen, wie man sie bei grafischen Benutzeroberflächen findet.

```
// Klassenvereinbarungen
class TShape
{
  protected:
    int Color;
    int X, Y;
  public:
    TShape(int XPos, int YPos, int AColor); // Konstruktor
    void SetColor(int AColor);
    void Drag(int X, int Y);
};

class TBox: public TShape
{
  protected:
```

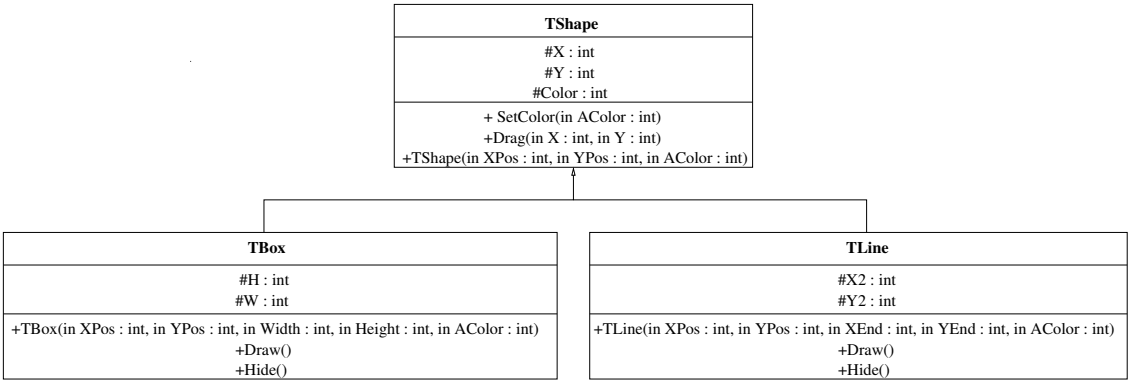


Abbildung 2.1: Darstellung einer Klassenhierarchie

```
    int W, H;
  public:
    TBox(int XPos, int YPos, int Width, int Height, int AColor);
    void Draw();
    void Hide();
};

class TLine: public TShape
{
  protected:
    int X2, Y2;
  public:
    TLine(int XPos, int YPos, int XEnd, int YEnd, int AColor);
    void Draw();
    void Hide();
};

// Konstruktoren
TShape::TShape(int XPos, int YPos, int AColor))
{
  Color = AColor;
  X = XPos;
  Y = YPos;
}

TBox::TBox(int XPos, int YPos, int Width, int Height, int AColor)
 : TShape(XPos, YPos, AColor) // Aufruf des Vorfahrenkonstruktors!
{
  W = Width;
  H = Height;
}

TLine::TLine(int XPos, int YPos, int XEnd, int YEnd, int AColor)
  : TShape(XPos, YPos, AColor)
{
  X2 = XEnd;
  Y2 = YEnd;
}

// übrige Methoden
```

```
void TShape::SetColor(int AColor)
{
  Color = AColor;
}

void TShape::Drag(int XPos, int YPos)
{
  X = XPos;
  Y = YPos;
}

void TBox::Draw()
{
  // Rechteck zeichnen
}

void TBox::Hide()
{
  int AColor = Color;
  SetColor(0);  // Schwarz zeichnen
  Draw();
  SetColor(AColor);
}

void TLine::Draw()
{
  // Linie zeichnen
}

void TLine::Hide()
{
  int AColor = Color;
  SetColor(0);  // Schwarz zeichnen
  Draw();
  SetColor(AColor);
}

// Hauptprogramm
void main()
{
  TBox Box(10,10,20,20,1);
  TLine Line(10,10,30,30,1);

  // Objekte zeichnen
  Box.Draw();
  Line.Draw();

  // Objekte verschieben
  Box.Hide();
  Box.Drag(0,0);
  Box.Draw();
  Line.Hide();
  Line.Drag(0,0);
  Line.Draw();
}
```

Ein Objekt der abgeleiteten Klasse ist immer auch ein Objekt der Basisklasse. Die Umkehrung gilt nicht. Durch Casting kann man ggf. die Zugehörigkeit zu einer bestimmten Unterklasse erzwingen. Dies ist zum Beispiel erforderlich, wenn man das Ergebnis einer allgemeinen Methode in einer Unterklasse weiterverwenden will:

```
TShape* ptShape;
TLine* ptLine;
//..
ptShape = ptLine  // o.k.
ptLine = static_cast<TLine*>(ptShape)  // Casting erforderlich
```

Bei einem Konstruktor mit Initialisierungsliste kann (sollte) auch ein Konstruktor der Basisklasse (*Vorfahrenkonstruktor*) angegeben werden. Fehlt die Angabe des Konstruktors, wird der Standardkonstruktor der Basisklasse benutzt.

```
TLine::TLine(int XPos, int YPos, int XEnd, int YEnd, int AColor)
             : TShape (XPos, YPos, AColor) {}
```

Die Initialisierung erfolgt in folgender Reihenfolge:

1. Initialisierung der Basisklasse
2. Initialisierungsliste
3. Ausführung des Codes im Konstruktor

2.2.1 Polymorphie

Besteht für verschiedene Klassen eine gemeinsame Funktionalität (z. B. `Draw`), deren Realisierung für die Klassen aber unterschiedlich ist, so kann man in den verschiedenen Klassen Methoden mit gleicher Schnittstelle (Name, Typ, Parameter) einrichten, die dann die entsprechende angepasste Rolle übernehmen. Man spricht dann von Polymorphie (= Vielgestaltigkeit). Möchte man Polymorphie in einer Basisklasse und einer abgeleiteten Klasse nutzen, muss die Funktion in der Basisklasse als virtuell (Schlüsselwort `virtual`) deklariert werden.

Welche Funktion bei polymorpher Deklaration benutzt wird, wird erst zur Laufzeit entsprechend dem Typ des Objektes, für das die Methode aufgerufen wird, ermittelt. Man spricht hier auch von dynamischer bzw. später Bindung (engl.: *late binding*). Hierbei wird beginnend mit der höchsten Basisklasse geprüft, ob eine virtuell deklarierte Funktion in der abgeleiteten Klasse überschrieben wird. Ist dies der Fall, wird diese benutzt, ggf. bei virtueller Deklaration wieder mit Prüfung in der Unterklasse. Möchte man gezielt die Methode einer bestimmten Klasse aufrufen, so kann dies explizit durch Angabe des Klassennamens (Vorfahren `Klasse::Methode(...);`) geschehen.

Betrachtet man in diesem Zusammenhang das Beispiel in Kapitel 2.2 genauer, wird man feststellen, dass die Methoden `Hide` bei `TBox` und bei `TLine` identisch sind, bis auf den Aufruf einer unterschiedlichen `Draw`-Methode.

Wenn man in die Basisklasse (`TShape`) eine virtuelle Methode `Draw` einfügt, wird erst zur Laufzeit entschieden, welche Methode (`TBox::Draw()` oder `TLine::Draw()`) tatsächlich

aufgerufen wird. Aus diesem Grund brauchen `TBox` und `TLine` auch keine eigene `Hide`-Methode mehr, sondern diese kann bereits in `TShape` deklariert werden. Ebenfalls kann `Drag` nun selbständig das Objekt auf dem Bildschirm löschen und neuzeichnen. Damit ergibt sich folgendes geändertes Codefragment für die Klassen:

```
class TShape
{ ...
  virtual void Draw() {}; // inline Definition
        // (leerer Anweisungsblock)
  void Hide();
  ...
};
...
void TShape::Hide()
{
  int AColor = Color;
  SetColor(0);       // zum Löschen schwarz zeichnen
  Draw();
  SetColor(AColor);
}

void TShape::Drag(int XPos, int YPos)
{
  Hide();
  X = XPos;
  Y = YPos;
  Draw();
}
...
// TBox::Hide und TLine::Hide entfallen

// Hauptprogramm
void main()
{
  TBox Box(10,10,20,20,1);
  TLine Line(10,10,30,30,1);

  // Objekte zeichnen
  Box.Draw();
  Line.Draw();

  // Objekte verschieben
  Box.Drag(0,0);
  Line.Drag(0,0);
}
```

In diesem Fall wird die Deklaration der Methode `Draw` als virtuell an die abgeleiteten Klassen weitervererbt. Es ist sinnvoll (aber nicht notwendig), das Schlüsselwort virtual auch in den abgeleiteten Klassen für geerbte virtuelle Funktionen zu benutzen. Dadurch kann vermieden werden, dass der Programmierer bei der Programmentwicklung ständig die gesamte Vererbungshierarchie dahingehend überprüfen muss, ob eine Funktion virtuell ist oder nicht.

Angemerkt sei noch, dass Destruktoren virtuell erklärt werden sollten. Ein virtueller Destruktor sorgt dafür, dass Nachfahren korrekt gelöscht werden können, auch wenn man nur einen Zeiger hat, der vom Typ des Vorfahren ist. Ein Nachfahrendestruktor ruft zudem au-

tomatisch den Vorfahrendestruktor auf, um sicherzustellen, dass der durch den Vorfahren belegte Speicherplatz wieder freigegeben wird.

Beispiel:

```
class BasisKlasse
{
  public:
    void* Inhalt;
    BasisKlasse() { Inhalt = NULL; };   // inline Def. Konstruktor
    virtual ~BasisKlasse() {};         // Destruktor
};

class Nachfahre : public BasisKlasse
{
  public:
    Nachfahre(char* Data);  // Konstruktor
    virtual ~Nachfahre(); // Destruktor
};

Nachfahre::Nachfahre(char* Data) : BasisKlasse()
{
  Inhalt = (void*) new char [strlen(Data)];
  strcpy(Inhalt, Data);
}

Nachfahre::~Nachfahre()
{
  delete Inhalt;
}

void main()
{
  BasisKlasse* AObject = new Nachfahre("Test"); // Zuweisung OK!
  ...
  delete AObject;  //hier wird ~Nachfahre() aufgerufen!
}
```

Konstruktoren dürfen jedoch nicht virtuell erklärt werden. Wäre ein Konstruktor eine virtuelle Methode, so würde seine Aufrufadresse erst während der Laufzeit ermittelt. Diese müsste dann aber in der Liste stehen, die der Konstruktor bei der Erzeugung eines Objekts anlegt. Die Adresse des Konstruktors wäre also erst dann verfügbar, wenn er bereits aufgerufen wurde („Henne und Ei“-Problem).

2.2.2 Abstrakte Klassen

Zum Aufbau von Klassenstrukturen ist es oft sinnvoll, gemeinsame Eigenschaften mehrerer Klassen in einer Oberklasse zusammenzufassen, obwohl eigentlich keine Objekte dieser Oberklasse existieren. So könnte z. B. eine Basisklasse Fahrzeuge bereits alle notwendigen Funktionen zur Beschreibung von PKW und LKW enthalten. Eine solche Klasse nennt man „Abstrakte Klasse“. Es können keine Objekte einer abstrakten Klasse erzeugt werden. In C++ realisiert man eine abstrakte Klasse durch Definition mindestens einer „rein virtuellen“ Funktion. Eine rein-virtuelle Methode wird dadurch deklariert, dass sie den Wert 0 zugewiesen bekommt. Es ergibt sich folgende Syntax für eine solche Methode:

```
virtual <Typ> <Funktionsname> (<Parameterliste>) = 0;
```

Die rein-virtuellen Funktionen müssen in allen abgeleiteten Klassen durch nicht „rein-virtuelle“ Funktionen überschrieben werden.

2.2.3 Mehrfachvererbung

In C++ ist es möglich, eine Klasse von mehreren Basisklassen abzuleiten. Sie erbt dann die Datenelemente und Methoden aller Oberklassen. Mehrfachvererbung sollte jedoch nicht ersatzweise zur Darstellung einer has-a-Beziehung verwendet werden: Ein Personenzug kann z. B. nicht als Unterklasse von Zug (enthält Lokomotive) und Personenwaggons dargestellt werden, sondern ist Unterklasse von Zug und enthält Personenwaggons. (Fast) alle Probleme lassen sich ohne Mehrfachvererbung darstellen.

Eine Mehrfachvererbung wird einfach durch Angabe aller Basisklassen hinter dem Ableitungsoperator implementiert. Häufig entstehen Vererbungsunklarheiten (oder es werden Konstruktoren mehrfach aufgerufen), wenn die Basisklassen selber von einer gemeinsamen Oberklasse erben. Dann müssen gezielt einzelne Basisklassen mit dem Schlüsselwort `virtual` virtuell erklärt werden.

Im folgenden Beispiel vererbt die Grundklasse an zwei abgeleitete Klassen und diese wiederum an eine Klasse mit dem Namen `ZweifachabgeleiteteKlasse`:

```
class Grundklasse
{
  ...
};

class AbgeleiteteKlasse1: virtual public Grundklasse {...};
class AbgeleiteteKlasse2: virtual public Grundklasse {...};
class ZweifachabgeleiteteKlasse:
  public AbgeleiteteKlasse1, public AbgeleiteteKlasse2
  {...};
```

Ein Mehrfachaufruf des Konstruktors der Grundklasse bei Instanziierung eines Objekts der Klasse ZweifachabgeleiteteKlasse wird hier vermieden.

2.3 Freundschaften

Ein grundlegendes Konzept von C++ ist die Datenkapselung, d. h. dass nur die Methoden einer Klasse Zugriff auf die privaten Mitglieder besitzen. Hierdurch kann das Objekt selbst kontrollieren, welche Werte in seine Data Member geschrieben werden und Fehleingaben abfangen. In Ausnahmefällen kann es jedoch sinnvoll sein, dass auch Funktionen, die nicht Mitglied der Klasse sind, Zugriff auf deren private Mitglieder erhalten. Dies lässt sich durch `friend`-Funktionen erreichen. Eine als `friend` deklarierte Funktion ist nicht Mitglied der Klasse, hat aber Zugriff auf die privaten Mitglieder dieser Klasse.

Es sei betont, dass die Klasse selbst die „Freundschaft“ anbieten muss, d. h. sie muss der betreffenden Funktion den Zugriff auf ihre Mitglieder erlauben. Da durch `friend`-Funktionen das Konzept der Datenkapselung unerwünschterweise umgangen wird, sollte man diese Funktionen so sparsam wie möglich einsetzen. Viele Programmierer sind der Ansicht, dass ihre Verwendung überhaupt nicht nötig ist. Das Beispiel zeigt die Verwendung von Freundschaften:

```
#include <string>
#include <iostream>

class person
{
  private:
    int alter;
    std::string name;
  public:
    person(int i, std::string wort);  // Konstruktor
    ~person();  // Destruktor
    void anzeigen();
    // hier wird der Funktion "geburtstag" die
    // Freundschaft angeboten
    friend void geburtstag(person);
};

person::person(int i, std::string wort)
  : alter(i), name(wort) { }

person::~person() { }

void person::anzeigen(void)
{
  std::cout << name << " ist "
    << alter << " Jahre alt." << std::endl;
}

void geburstag(person a)
{
  // Zugriff auf privates Mitglied von a
  a.alter += 1;
}

// Hauptprogramm
int main()
{
  person hans(42, "Hans");
  hans.anzeigen();
  geburtstag(hans);
  hans.anzeigen();
}
```

Die Ausgabe des obigen Programmes lautet:

```
Hans ist 42 Jahre alt.
Hans ist 43 Jahre alt.
```

Es sei noch erwähnt, dass eine Klasse nicht nur Funktionen, sondern auch ganzen Klassen die Freundschaft anbieten kann. Alle Mitgliedsfunktionen der befreundeten Klasse erhalten unbeschränkten Zugriff auf alle Mitglieder der eigenen Klasse.

Man sollte die Verwendung von Freundschaften soweit wie möglich vermeiden und z. B. durch entsprechende Zugriffsfunktionen ersetzen, da bei jeder Änderung der Daten sonst ggf. auch in allen befreundeten Klassen der Zugriff geändert werden muss.

2.4 Überladen

Normalerweise erlauben es Programmiersprachen, lediglich eine einzige Definition für ein und denselben Funktionsnamen anzugeben, damit der Compiler eindeutig entscheiden kann, welcher Code verwendet werden soll. In C++ gilt diese Einschränkung nicht. Es können mehrere Funktionen gleichen Namens definiert werden. Diese Technik wird als *Überladen* (engl. *overloading*) bezeichnet. Überladen ist dabei nicht nur für Funktionen sondern auch für Operatoren möglich.

2.4.1 Funktionen / Methoden

Für überladene Funktionen stellt sich die Frage, anhand welchen Kriteriums der Compiler entscheidet, welcher Code auszuführen ist. Beim Aufruf einer Funktion müssen die übergebenen Argumente, die bei der Funktionsdefinition angegebenen Datentypen besitzen. Der Compiler trifft die Entscheidung über den auszuführenden Code anhand der übergebenen Argumente. Das bedeutet, dass sich überladene Funktionen in der Anzahl und/oder dem Typ der übergebenen Argumente unterscheiden müssen. Bitte beachten Sie, dass ein veränderter Rückgabetyp einer Funktion kein Kriterium darstellt. Der Compiler meldet einen Fehler, wenn sich zwei Funktionen lediglich im Typ ihres Rückgabewertes unterscheiden. Das folgende Beispiel zeigt eine überladene Funktion, die das Maximum zweier `int`- oder `char`-Werte liefert:

```
int max(int a, int b) { return (a > b) ? a : b; }
char max(char a, char b) { return (a > b) ? a : b; }
```

Der Versuch, die Funktion ein weiteres Mal zu überladen, so dass der größere zweier `int`-Werte gedruckt wird, führt zu einem Fehler:

```
void max(int a, int b)
{
  if (a > b)
    cout << a << " ist groesser als " << b << endl;
  else if (b > a)
    cout << b << " ist groesser als " << a << endl;
  else
    cout << a << " ist gleich " << b << endl;
}
```

Diese Funktion unterscheidet sich von `int max(int a, int b)` nur im Typ des Rückgabewerts (`void` statt `int`). Daher kann der Compiler beim Funktionsaufruf nicht entscheiden, welcher Code ausgeführt werden soll.

Ähnlich der außerhalb einer Klasse definierten Funktionen können auch klasseneigene Methoden überladen werden. Den verschiedenen Versionen einer überladenen Methode können verschiedene Zugriffsrechte gegeben werden. Eine sehr häufig überladene Methode ist der Konstruktor. Folgendes Beispiel hat drei Konstruktoren, einen Standardkonstruktor, einen parametrisierten Konstruktor und einen privaten Copykonstruktor.

```
#include <string>

class Student
{
  private:
    std::string p_Name;
    int p_semester;
    Student(const Student& s);       // Copykonstruktor
  public:
    Student() : p_Name(""), p_semester(0) {}  // Standardkonstruktor
    Student(const std::string& n, int s)
    : p_Name(n), p_semester(s) {}   // param. Konstruktor
};

int main()
{
  Student student;                // Standardkonstruktor
  Student arthur("Arthur", 42); // param. Konstruktor
  Student s3(s1);                 // Fehler, Copykonstruktor private
}
```

Man beachte die Unterschiede zwischen

1. Vererbung (Methodenname aus Basisklasse)
2. Polymorphie (Gleicher Methodenname und gleiche Parameter in verschiedenen Klassen)
3. Überladung (Gleicher Methodenname und verschiedene Parameter in einer Klasse)

2.4.2 Operatoren

In C++ sind Operatoren prinzipiell als Funktionen definiert und können somit wie Funktionen benutzt und auch überladen werden. Der Name einer Operatorfunktion besteht aus dem festen Bestandteil `operator` und seinem Zeichen. Der Funktionsname für den Operator + lautet also `operator+` und die Anweisung `int a = 4 + 3;` wird vom Compiler für den Programmierer unsichtbar in `int a = operator+(4, 3);` übersetzt.

Da es unsinnig wäre, das Verhalten der Operatoren auf in C++ eingebauten Typen (z. B. `int`) zu verändern, muss bei der Operatorüberladung mindestens ein Operand ein Objekt sein. Operatoren auf komplexeren Datenstrukturen können individuell und problemangepasst definiert werden. So stellt z. B. die Implementation eines Plus-Operators zur Stringverkettung, wie er in anderen Programmiersprachen standardmäßig implementiert ist, in C++ kein Problem dar. Folgende Operatoren können in C++ überladen werden:

```
+       -       *       /       %       ^       &
|       ~       !       =       <       >       +=
-=      *=      /=      %=      ^=      &=      |=
<<      >>      >>=     <<=     ==      !=      <=
>=      &&      ||      ++      --      ->*     ,
->      []      ()      new     new[]   delete  delete[]
```

In C++ werden Klassenmethoden wie normale Funktionen behandelt, die implizit als ersten Parameter den `this`-Zeiger der zugehörigen Instanz bekommen. Gleiches gilt selbstverständlich auch für Operatoren, die somit innerhalb (als Klassenmethode) oder auch außerhalb (als Funktion) einer Klasse definiert werden können. Ein zweistelliger Operator kann durch eine Klassenmethode mit einem Parameter oder durch eine globale Funktion mit zwei Parametern definiert werden. Ein einstelliger Operator kann durch eine Klassenmethode ohne Parameter oder durch eine globale Funktion mit einem Parameter definiert werden.

Was ist besser, Operatordefinition innerhalb oder außerhalb der Klasse? Leider lässt sich diese Frage nicht allgemeingültig beantworten. Als Regel schlägt Bjarne Stroustrup (der Entwickler von C++) vor: Eine Funktion sollte Klassenmethode sein, soweit es keinen guten Grund dagegen gibt. Manchmal ist eine Definition innerhalb der Klasse unmöglich, wenn z. B. der erste Parameter ein nicht-Klassenobjekt sein soll, da ja bei Methoden der `this`-Zeiger immer implizit der erste Parameter ist. Beispiel hierfür sind der Ein- und Ausgabeoperator (`<<, >>`), bei denen der erste Operand ein istream bzw. ostream ist, oder auch bei arithmetischen Operatoren auf gemischten Typen, also z. B. `operator+(int, X)`.

Als Beispiel folgt die Deklaration der am häufigsten überladenen Operatoren für eine Typklasse X:

```
class X
{
  public:
    const X operator++(int);     // Postfix-Inkrement
    X& operator++();             // Praefix-Inkrement
      // analog fuer --
    X& operator+=(const X&);     // X+=X
      // analog fuer alle op=
    X& operator+(const X&);      // arithm. +
      // analog fuer arith., logische und bit Operatoren
    bool operator==(const X&);   // Vergleich
      // analog fuer !=,<,>,<=,>=
    X& operator[](int);          // Subskript
    void operator()();           // Funktionsaufruf
    operator int() const;        // Konvertieroperator
};

istream& operator>>(istream&, X&);
ostream& operator<<(ostream&, const X&);
```

Anmerkungen:

- Wie bei den meisten Methoden sollten konstante Referenzen übergeben werden, damit Objekte nicht beim Funktionsaufruf kopiert werden müssen, was unter Umständen sehr lange dauern kann. Man kann sich die Übergabe konstanter Referenzen als ein schnelles call-by-value vorstellen. Schnell, weil nur eine Referenz übergeben wird und call-by-value weil auf den Wert nur lesend zugegriffen werden darf.

- Die Zuweisungsoperatoren (`=, +=, -=, ...`) sollten immer eine nichtkonstante Referenz auf this zurückgeben, da dies bei den C++ Standardtypen (`int, double, ...`) auch so üblich ist. Dadurch sind Zuweisungsketten wie `a=(b=c);` oder auch `(a=b)=c;` möglich. Über den Sinn solcher Zuweisungsketten lässt sich streiten, aber warum sollte man von bestehenden Konventionen abweichen.

- Damit der Operator zum Index-Zugriff (`operator[]`) auch auf der linken Seite einer Zuweisung vorkommen kann (z. B. `a[i] = 5;`), sollte er eine Referenz zurückgeben. Weiterhin kann der Zugriffsoperator anstatt int auch einen anderen Parameter bekommen, z. B. ist bei der STL-map der Parameter ein `const string &`.

- Um über Zeiger auf Elemente eines Objektes zugreifen zu können wurde der Operator `->` definiert, d. h. der Zugriff erfolgt dermaßen: `Zeiger->Element`. Dies ist die verkürzte Schreibweise für `(*Zeiger).Element`.

- Syntaktisch wird die Definition von Präfix bzw. Postfix durch den übergebenen int unterschieden, er hat ansonsten keine weitere Bedeutung und wird auch meistens deswegen nicht benannt. Um den semantischen Unterschied zwischen den Präfix und Postfix Varianten der Inkrement bzw. Dekrement Funktion zu verstehen, merkt man sich am besten folgende Regel. Präfix bedeutet „erhöhen und holen“, während Postfix „holen und erhöhen“ bedeutet. Die Postfix-Variante kann keine Referenz zurückgeben, denn hier muss ein temporäres Objekt zurückgegeben werden, da ja der alte Wert zurückgeliefert werden muss, während der aktuelle Wert inkrementiert wird. Für Integer würden die beiden Inkrementversionen folgendermaßen aussehen:

```
MyInt& MyInt::operator++()           // Praefix-Inkrement
{
  *this += 1;
  return *this;
}

const MyInt MyInt::operator++(int)   // Postfix-Inkrement
{
  MyInt oldValue = *this;
  ++(*this);
  return oldValue;
}
```

  Die Postfix Variante sollte einen konstanten Wert zurückliefern, da dies zum einen Standard in C++ ist und weiterhin Anweisungen wie `i++++;` zurückgewiesen werden, da man ansonsten ein nichtintuitives Verhalten hat (Das zweite ++ erhöht nur eine temporäre Variable). Allerdings macht die Anweisung `++++i;` durchaus Sinn und ist durch die Referenzrückgabe auch möglich.

- Alle Operatoren, bei denen als Operand (das, was links vom Operator steht) ein Element der Klasse definiert ist, können (sollen) innerhalb der Klasse definiert werden,

damit im Operator auf die privaten Variablen der Klasse zugegriffen werden kann. Ist der Operand kein Element der Klasse (wie bei den Ein-/Ausgabe-Operatoren, wo es der entsprechende Stream ist), so muss der Operator außerhalb der Klasse definiert werden. Dies führt dazu, dass auf die privaten Variablen nicht zugegriffen werden kann. Dort ist es meist sinnvoll, eine entsprechende Memberfunktion zu implementieren und diese dann aufzurufen.

Der Ausgabeoperator `operator<<()` sollte eine Referenz auf `ostream` zurückliefern, damit Anweisungen wie `cout << a << b;` möglich sind. Dies ist einsichtig, wenn man sich diese Anweisung in einer Notation mit Operatorfunktion betrachtet:

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

Gleiches gilt für den Eingabeoperator `operator>>()`.

Für eine Klasse `Mitarbeiter` mit Name und Vorname könnte man sich folgenden Ausgabeoperator vorstellen:

```
class Mitarbeiter {
  public:
    /* ... */
    string sGetName() { return Name; }
    string sGetVorname() { return Vorname; }
  private:
    string Name;
    string Vorname;
};

ostream& operator <<(ostream& out, Mitarbeiter& x)
{
  out << x.sGetName() << ", " << x.sGetVorname();
  return out;
}
```

- Mit dem Konvertierungsoperator wird eine implizite Konvertierung von einem Benutzerdefinierten Typ zu einem schon definierten Typ, der auch fundemental sein kann, definiert. Im Beispiel wird eine implizite Konvertierung von `X` nach `int` definiert. Man Beachte, dass der Typ, zu dem konvertiert wird, Teil des Namens ist und nicht als Rückgabetyp der Konvertierungsfunktion anzugegeben ist. Ein Konvertierungsoperator gleicht damit eher einem Konstruktor. Dieser Operator sollte allerdings mit Vorsicht eingesetzt werden, da schnell Mehrdeutigkeit entstehen können, die der Compiler nicht auflösen kann.

2.5 Typumwandlung

In Ausdrücken und Zuweisungen können fundamentale Datentypen (`char, int, float, double, ...`) beliebig gemischt werden. Um die gewünschte Aktion durchführen zu können, müssen die verschiedenen Werte auf einen gemeinsamen Nenner gebracht werden. Dazu

werden die Werte automatisch (wenn möglich) so umgewandelt, dass keine Informationen verloren gehen. Es besteht aber für den Programmierer durch Angabe des Typs in Klammern auch die Möglichkeit, eine Typumwandlung zu erzwingen. Dabei sollten mögliche Datenverluste beachtet werden.

```
int i; float f; double d;

d = i + f;              // i->float, (i+f)->double
i = (int) f + (int) d;  // f->int, d->int, evtl. Datenverlust!
          // Nachkommastellen von f und d werden abgeschnitten
```

In C++ kann derselbe Effekt auch für benutzerdefinierte Typen (Klassen) erreicht werden. Hierbei unterscheidet man zwischen der impliziten (automatischen) und der expliziten (erzwungenen) Umwandlung.

2.5.1 Implizite (automatische) Typumwandlung

Eine automatische Typumwandlung kann auf zwei Arten ermöglicht werden: Durch Definition eines bestimmten Konstruktors oder eines speziellen Umwandlungsoperators.

Konstruktor zur Typumwandlung

Eine automatische Typumwandlung wird durch jeden Konstruktor, der mit genau einem Argument aufgerufen werden kann, definiert. Dazu zählen auch Konstruktoren mit mehr als einem Parameter, die aber jeweils Default-Argumente besitzen.

```
class Bruch {
  public:
    Bruch(int x = 0, int y = 1){ p_x = x; p_y = y; }
    Bruch operator*(const Bruch& b){
      Bruch h(b);
      h.p_x *= p_x;
      h.p_y *= p_y;
      return h;
    }
  private:
    int p_x, p_y;   // Zähler und Nenner
}

int main() {
  Bruch x(5, 2); Bruch y;
  y = x * 15;       // 15->Bruch(15)
  return 0;
}
```

Zur Umwandlung von `int` nach `Bruch` wird der Konstruktor benutzt, der eine `int`-Zahl als Parameter akzeptiert. Die Umwandlung ist so aber nur in diese Richtung möglich. Der

Ausdruck `15 * x` ist nicht möglich, da für einen fundamentalen Datentyp keine Umwandlungsfunktionen definiert werden können. In diesem Fall müsste man eine Umwandlung selbst durchführen, z. B. in der Form `Bruch(15) * x`.

Manchmal können bei dieser Form der Typumwandlung Probleme auftreten. Man definiert einen entsprechenden Konstruktor, möchte aber keine automatische Typumwandlung festlegen. Dies kann man verhindern, indem man dem Konstruktor das Schlüsselwort `explicit` voranstellt.

Typumwandlung durch Operator

Alternativ zum Konstruktor erfolgt eine automatische Typumwandlung auch durch die Definition eines entsprechenden Umwandlungsoperators. Dieser wird definiert durch das Schlüsselwort `operator`, gefolgt von dem Typ, in den man umwandeln möchte. Dabei muss es sich nicht um einen fundamentalen Datentyp handeln.

```
class Bruch {
  public:
    explicit Bruch(int x = 0, int y = 1){ p_x = x; p_y = y; }
    operator double() const {
      return (double) (p_x / p_y);
    }
  private:
    int p_x, p_y;    // Zähler und Nenner

int main() {
  Bruch x(5, 2);
  double y;

  y = x * 15;        // x->double
  y = 10 * x;        // hier auch möglich x->double
  return 0;
}
```

Hier wird `x` mittels des Operators von `Bruch` nach `double` umgewandelt. Die Anwendung von `explicit` beim Konstruktor ist notwendig, da die Umwandlung sonst zweideutig ist.

2.5.2 Explizite (erzwungene) Typumwandlung

In der ursprünglichen Spezifikation von C++ waren für die explizite Typumwandlung zwei Alternativen vorgesehen:

Funktionale Notation

Diese ermöglicht eine Typumwandlung durch Aufruf eines entsprechenden Konstruktors. Dabei kann auch eine Liste von Argumenten übergeben werden.

```
Bruch y = x * Bruch(25, 11);
Bruch y = x * Bruch(15);
```

Cast-Notation

Hierbei wird der Datentyp, in den umgewandelt werden soll, in Klammern dem umzuwandelnden Objekt vorangestellt. Die Umwandlung erfolgt entsprechend der automatischen Typumwandlung durch den entsprechenden Konstruktor mit einem Parameter oder durch einen passenden Umwandlungsoperator.

```
Bruch y = x * (Bruch) 10;
double z = (double) Bruch(10, 7);
```

Diese Schreibweise stellt eine Kompatibilität zu C dar und ist im Gegensatz zur funktionalen Notation auch in der Lage, zusammengesetzte Typennamen (Zeiger) zu verwenden.

```
(char*) p_sName
```

Da Typumwandlungen verschiedenen Zwecken dienen können, wurden in der neuesten C++-Spezifikation vier Operatoren zur genauer definierten Typumwandlung eingeführt:

`static_cast`, `dynamic_cast`, `const_cast`, `reinterpret_cast`.

Die gängigsten sind die ersten beiden Operatoren und werden daher hier kurz erläutert.

- `static_cast<typ> (parameter)`

  Der static_cast-Operator konvertiert den `parameter` in einen Typ `typ`. Dies geschieht zwischen verwandten Typen, wie etwa verschiedenen Zeigertypen, zwischen einer Aufzählung und einem integralen Typ oder einem Gleitkommatyp und einem integralen Typ. Man kann auch Zeiger innerhalb einer Klassenhierarchie bewegen, d. h. die Umwandlung von einer Basisklasse in eine abgeleitete Klasse (Downcast) oder umgekehrt (Upcast). Im Gegensatz zum dynamic_cast findet keine Überprüfung zur Laufzeit statt, womit der Programmierer für die Richtigkeit der Umwandlung verantwortlich ist.

  ```
  class B {/*...*/};
  class D : public B {/*...*/};

  void f(B* pb, D* pd) {
    D* pd2 = static_cast<D*> (pb);
    // unsicher, pb zeigt evtl. nicht auf D sondern auf B

    B* pb2 = static_cast<B*> (pd);  //sicher
  }
  ```

- `dynamic_cast<typ> (parameter);`

  Der dynamic_cast-Operator konvertiert den parameter in einen Typ `typ`. `typ` muss ein Zeiger oder eine Referenz auf eine zuvor definierte Klasse sein oder ein Zeiger auf void. Als `parameter` wird entsprechend ein Zeiger oder ein Objekt einer Klasse übergeben. Der Zweck des dynamic_cast ist die Behandlung von Fällen, in denen die Korrektheit der Umwandlung nicht durch den Compiler ermittelt werden kann. Der Operator schaut dabei zur Laufzeit auf den Typ des zu wandelnden Objekts (`parameter`) und entscheidet, ob die Umwandlung sinnvoll ist. Falls nicht, wird eine 0 zurückgegeben.

```
class B {/*...*/};
class D : public B {/*...*/};

void f() {
  B* pb = new D; // unklar, aber ok
  B* pb2 = new B;

  D* pd = dynamic_cast<D*> (pb);
  // ok, pb zeigt auf D

  D* pd2 = dynamic_cast<D*> (pb2);
  // pd2 == 0, da pb2 auf B zeigt
}
```

Einen Sonderfall stellen Variablen dar, die von einem `enum`-Datentyp abgeleitet wurden. Leider erfolgt dann beim Casten keine Typprüfung. `static_cast` ist nicht möglich, da `enum` eigentlich ein Relikt aus C ist.

Man kann die Typdefinition auch auf eine Klasse beziehen. Dann muss man diese Definition `static` machen, um auf die Werte über Klassenname:: zugreifen zu können.

Beispiel:

```
class Geld {
  public:
    static enum CentMuenzen {C01=1, C02, C05=5, C10=10, C20=20, C50=50};
  private:
    CentMuenzen p_Einwurf;
}
```

Der Zugriff ist dann mit `Geld::CentMuenzen` oder `Geld::C01` möglich.

Zur ausführlichen Beschreibung der Operatoren ziehen Sie bitte weitere Literatur zurate.

2.6 Templates

Oftmals unterscheiden sich Klassen oder Funktionen zur Lösung verschiedenster Probleme nur in den Datentypen, auf denen sie arbeiten, nicht jedoch in ihrer Funktionalität. Zum Beispiel wird eine Klasse, die eine Warteschlange implementiert, immer Funktionen zum

Anhängen und Entfernen von Objekten bereitstellen müssen. Lediglich die Datentypen der Objekte können sich von Fall zu Fall unterscheiden. Genauso ist der Ablauf einer Sortierroutine für unterschiedlichste Datentypen immer gleich und beruht nur auf dem Vergleich und dem Vertauschen von zwei Elementen.

In C++ stellen *Templates* ein Hilfsmittel zur Generierung des entsprechenden Codes dar. Ein Template ist ähnlich einer Schablone. Erst durch das Ausfüllen der Schablone wird es zu eigenständigen Code.

Ein Template ist eine Klasse oder Funktion mit (noch) nicht definierten Datentypen. Die eigentlichen Datentypen werden erst bei der Instanziierung eines Objekts einer Templateklasse oder beim Aufruf einer Templatefunktion als Parameter übergeben. Die Vereinbarung erfolgt mit dem Schlüsselwort `template`. Erst bei der Übersetzung wird für jeden benutzten Datentyp der entsprechende Code generiert. Das gesamte Template muss sich in einer Datei befinden und in den Quellcode der benutzenden Funktion eingebunden werden. Bei Templates steht also sowohl Deklaration als auch Methodencodierung in der *.h-Datei.

Syntaktisch beginnt ein Template immer mit einer Zeile der Form:

```
template <class T>
```

Die folgenden Regeln sind bei der Definition der `template-Zeile` zu beachten:

- `<class T>` steht als Platzhalter für einen benutzten Datentyp bzw. eine Klasse. Überall dort, wo im Template ein entsprechender Typ verwendet werden soll, benutzt man diesen Platzhalter. Der Buchstabe T ist lediglich ein (häufig benutztes) Beispiel, es kann auch jeder andere Bezeichner verwendet werden.
- Die `<class T>`-Vereinbarung ist eine Liste. Es können auch mehrere Datentypen benutzt werden. Alle folgenden Vereinbarungen sind erlaubt:

  ```
  <class T>
  <class A, class B>
  <class T1, class T2>
  ```

- Auf den `template`-Kopf folgt die normale Definition der Klasse (Datenstrukturen, Methoden) oder der Funktion (Rückgabetyp, Name, Parameterliste). Hängt der Rückgabetyp und/oder der Typ eines Parameters von der Funktion ab, die aus dem Template generiert wird, so verwendet man `T` als Datentyp. Dabei kann `T` auch für lokale Variablen verwendet werden.

2.6.1 Funktionentemplates

Funktionentemplates werden immer dann sinnvoll eingesetzt, wenn Algorithmen unabhängig vom Datentyp auf bestimmte Eigenschaften (Methoden) dieser Datentypen zurückgeführt werden können. Am einfachsten erlernt man die Verwendung von Funktionentemplates anhand eines Beispiels:

Die Berechnung von Zinsen aus Zinssatz und Kapital ist immer dann möglich, wenn zwischen den verwendeten Datentypen der `*-operator` definiert ist. Für Standardtypen ist die Operation natürlich auch ohne Template möglich, aber für Kapital ist ja auch eine

spezielle Klasse (z. B. als BCD-Code oder über verschiedene Währungen) vorstellbar. Das Template sieht dann so aus:

```
// Funktionstemplate zur Verzinsung:
template <class T>
float zinsen(T kapital, float zinssatz)
{
  return (kapital * zinssatz / 100);
}
```

Der erste Parameter der Funktion kann ein beliebiger Datentyp sein (angezeigt durch das `T`) vorausgesetzt die *-Operation zwischen diesem Datentyp und `float` ist definiert und liefert als Ergebnis einen `float`-Wert. Aufgrund dieser Funktionsschablone sind z. B. folgende Aufrufe möglich:

```
int geld1 = 1000;
float geld2 = 1000.0;
float prozent = 0.03;
float ertrag;
ertrag = zinsen(geld1, prozent);  // 1. Parameter ist int
ertrag = zinsen(geld2, prozent);  // 1. Parameter ist float
```

Sinnvoller ist es aber, wenn diese Funktion auch einen Wert vom selben Typ wie Kapital zurückliefern würde. Das ginge mit folgender Templatedefinition:

```
template <class T>
T zinsen(T kapital, float zinssatz)
{
  T hilf;
  double q = zinssatz / 100.0;
  hilf = kapital * q;
  return hilf;
}
```

Hier muss der `*-operator` zwischen dem verwendeten Datentyp und `double` definiert sein und ein Objekt dieses Typs zurückliefern. Die Hilfsvariable `q` ist erforderlich, da sonst auch der `/-operator` für `T` definiert sein müsste. Die Hilfsvariable `hilf` ist nicht notwendig, sondern soll nur die Definition von Hilfsvariablen des Schablonentyps darstellen.

Die Funktionsschablone kann also für alle Variablentypen genutzt werden, für welche die innerhalb der Funktion verwendeten Operatoren definiert sind. Der Compiler erzeugt zur Übersetzungszeit den benötigten Code für alle Funktionsaufrufe. In diesem Fall ist die Funktion also tatsächlich mehrmals vorhanden.

2.6.2 Klassentemplates

Klassentemplates werden immer dann eingesetzt, wenn Datenstrukturen unabhängig vom Typ der gespeicherten Elemente dargestellt werden können und immer die gleichen Eigenschaften und Methoden anbieten. Alle abstrakten Datentypen (Listen, Bäume, Stack etc.)

sind typische Beispiele. Für diese gibt es daher auch die bereits vorgefertigten Templates in der STL (s. Kapitel 3.4).

Den Aufbau eines Klassentemplates soll das folgende Beispiel eines Stacks verdeutlichen:

```
// Klassentemplate
template <class T> // Def.zeile eines Templates für Datentyp T
class Stack
{
  private:
    T theElem;        // Speicher für das Datenelement
    Stack* ptNext;    // Zeiger auf den Nachfolger
  public:
    Stack() { ptNext = 0 } // K'tor: unterstes Element des Stacks=0
    Stack(T aElem, Stack *ptS = 0) : theElem(aElem), ptNext(ptS) { }

    Stack* push(T aElem); // Fügt Element oben hinzu
    Stack* pop();         // Löscht oberstes Element
    bool empty() { return (ptNext == 0); }  // Stack leer ?
    T top() { return theElem; }
};

template <class T>        // Templatefunktion
Stack<T>* Stack<T>::push(T aElem)
{
  Stack<T>* p = new Stack<T>(aElem, this);
  return p;
}

template <class T>
Stack<T>* Stack<T>::pop()
{
  if (empty()) return this;
  Stack<T>* p = ptNext;
  delete this;
  return p;
}
```

Der Name einer Klasse, die aus solch einem Template generiert wird, ist der Name des Klassentemplates (hier: `Stack`), gefolgt von den konkreten Datentypen in spitzen Klammern. Folgender Code erzeugt z. B. zwei Stacks, einen zur Verwaltung von Integer-Werten und einen zur Verwaltung von Gleitkommawerten:

```
Stack<int> stInt;
Stack<float> stFloat;
```

Hinweis: Man kann die Implementierung der Methoden direkt innerhalb der Klassendefinition durchführen (wie bei `empty` oder `top`) oder wie sonst getrennt als Templatefunktion. Bei der Verwendung des Scope-Operators:: muss dem Compiler mitgeteilt werden, dass es sich lediglich um ein Template und nicht um eine fertige Klasse handelt. Dies geschieht durch das angehängte Platzhalteelement in spitzen Klammern (`<T>`) an den Templatedatentyp (wie z. B. oben in `Stack<T>`).

Template-Parameter dürfen von beliebigem Typ sein, vorausgesetzt die Argumente können vom Compiler ausgewertet werden. Eine Definition für eine Templateklasse eines Vektors könnte also z. B. mit der Vektorgröße als Parameter so aussehen:

```
template <class TElem, int TSize>
class Vector
{
  private:
    TElem array[TSize];
    /*...*/
};
```

Dann wäre folgende Definition erlaubt:

```
Vector<double, 100> ds // Vektor mit 100 double-Werten
```

die Definition

```
int n = 100;
Vector<double, n> ds
```

aber nicht, da `n` nicht konstant ist und die Größe von `n` beim Compilieren nicht feststeht.

2.7 Exception Handling (Ausnahmebehandlung)

Unter Exceptions (Ausnahmen) versteht man besondere Programmsituationen, die zur Laufzeit eines Programms auftreten können, insbesondere die sogenannten Laufzeitfehler. Beispiele sind falsche Eingaben, mathematische Fehler (z. B. Divisionen durch Null), Speicherfehler oder Bereichsüberschreitungen bei Vektoren, aber auch andere Situationen, die eine normale Fortsetzung des Programms verbieten. Diese Situationen können bisher nur synchron, das heißt direkt an der Stelle des Auftretens abgehandelt werden. Beim Entwurf von Klassen oder Bibliotheken ist aber oft an der Stelle, an der die Ausnahmesituation entdeckt wird, nicht klar, wie sie geeignet zu behandeln ist. Dann muss durch Setzen eines Fehlercodes und Rücksprung über mehrere Ebenen eine Fehlerbehandlung vorgesehen werden. C++ stellt mit dem «Exception Handling» ein elegantes Konzept zur asynchronen Fehler- (Ausnahme-) Behandlung bereit: Die Bearbeitung der Ausnahmesituation wird vom Ort des Entdeckens direkt an den Ort des Aufrufs weitergereicht.

Eine Exception ist ein Objekt eines beliebigen Typs, das am Entdeckungsort geworfen (`throw`) und am Behandlungsort durch den zugehörigen Exception Handler gefangen (`catch`) wird. Das Objekt enthält alle für die Behandlung wichtigen Informationen. Meist werden für diesen Zweck eigens definierte Klassen verwendet. Der zu überwachende Code steht in einem `try`- Block, der ggf. von mehreren Exception Handlern überwacht wird. `throw` und `catch` stehen normalerweise nicht innerhalb derselben Funktion (dann könnte man auch `if-then-else` zur synchronen Fehlerbehandlung benutzen). Zur Verdeutlichung diene zunächst folgendes Beispiel:

Die Stackklasse aus vorigem Kapitel soll bei Aufruf von `top()` auf leerem Stack eine Fehlermeldung liefern.

```
template <class T>
T Stack<T>::top()
{
  if (empty()) {    // Werfen der Exception als char*
    throw "Error: Stack empty.\n";
  }
  return theElem;
}
```

Hier wird als einfachste Form direkt ein C-String (`char*`) geworfen, der die Meldung enthält. Um diese Fehlersituation zu berücksichtigen, muss der Aufruf der Funktion `top()` in einem try- Block stehen, der einen Exception-Handler für `char*` enthält.

```
try {
  // Dieser Block wird von den Exception Handlern überwacht.
  /*...*/
  x = aStack.top();
  /*...*/
}
catch (const char* s) { cout << s; }  // Handler für Typ char*
```

Beim Wurf der Exception wird mit dem Code des entsprechenden Handlers fortgesetzt, hier wird also der Text auf `cout` ausgegeben. Der Code hinter dem `throw`-Statement wird nicht mehr ausgeführt, ebenso wird der entsprechende `try`-Block ohne Bearbeitung des nach dem Aufruf stehenden Codes verlassen. Das geworfene Fehlerelement (hier der String) wird an die Variable des `catch`-Blocks (hier s) übergeben und kann so referenziert werden.

Bei größeren Projekten oder Klassen, die von anderen benutzt werden, sollte man eigene Exception-Klassen (oder sogar eine Klassenhierarchie) einführen, um eine Kollision mit anderen Programmteilen zu vermeiden. Die entsprechenden Exception-Klassen müssen dann Elementvariablen zur Aufnahme aller Daten haben, die die Ausnahme beschreiben. Diese werden beim `throw` durch Aufruf entsprechender Konstruktoren gefüllt. Außerdem empfiehlt sich die Definition einer (polymorphen) Bearbeitungsfunktion (z. B. `behandleException()`), die dann im `catch`-Block aufgerufen werden kann.

Ein Teil einer mathematischen Fehlerhierarchie könnte dann so aussehen:

```
class MathError
{
    // Variable zum Speichern der Fehlerbeschreibung
  protected:
    char p_szErrString[50];
    // Konstruktor mit Fehlerbeschreibung
  public:
    MathError(const char* s)
    {
      strcpy(p_szErrString, s);
      return;
    }
    // Fehlerbehandlung: Ausgabe der Beschreibung
    virtual void behandleFehler()
    {
      cout << p_szErrString;
    }
};
```

```
class NegWurzel: public MathError
{
  // Variable zum Speichern des (negativen) Arguments
  protected:
    double p_dArg;
  public:
    // Konstruktor mit Fehlerbeschreibung und Argument
    NegWurzel(const char* s, double a) : MathError(s), p_dArg(a) { }

    // Fehlerbehandlung: Ausgabe der Beschreibung mit Argument
    virtual void behandleFehler()
    {
      MathError::behandleFehler();
      cout << " Argument:" << p_dArg;
    }
};
```

Ein Exception Handler dazu könnte dann die Fehler über eine Referenz zu MathError abfangen und behandeln:

```
try {
    /*...*/
}
catch (MathError& e) {
  // Handler für MathError und alle Ableitungen davon
  e.behandleFehler();
}
```

Ebenso wären separate catch- Blöcke möglich (z. B. wenn keine polymorphe Behandlung des Fehlers existiert):

```
try {
    /*...*/
}
catch (NegWurzel& e) {
  // behandle NegWurzel
}
catch (MathError& e) {
  // behandle alle anderen MathError
}
```

Eine Möglichkeit für eine Ausnahme könnte im Programm folgendermaßen geworfen werden:

```
  throw NegWurzel("Negative Wurzel", -9.0);
```

Zusammenfassend kann das Exception Handling in C++ so dargestellt werden:

An der Stelle des Codes, wo die Ausnahmesituation auftritt (meistens in einer Bibliothek oder allgemein benutzten Klasse) wird ein Objekt einer Klasse geworfen, welches das Ausnahmeereignis repräsentiert. Die Syntax dazu lautet:

`throw` <*Ausnahmeobjekt*>;

Um auf dieses Ausnahmeobjekt reagieren zu können, muss der Aufruf der (Bibliotheks-)Methode innerhalb eines `try`-Blockes stehen, der einen zum Ausnahmeobjekt passenden Exception Handler (`catch`-Block) enthält. Die Syntax eines `try-catch`-Blocks lautet:

```
try {
  // Dieser Teil des Codes wird überwacht
  // hier Aufruf der Bibliotheksfunktion
}
catch (<Ausnahmeklasse1>& aExc) {
  // hier werden Ausnahmen der Ausnahmeklasse1
  // und aller Unterklassen bearbeitet
}
[ catch (<Ausnahmeklasse2>& aExc) {
  // hier werden Ausnahmen der Ausnahmeklasse2
  // und aller Unterklassen bearbeitet
} ]
/*...*/
[ catch (/*...*/) {
  // hier werden alle übrigen Ausnahmen behandelt
} ]
```

Hinweise:

1. Es können beliebig viele `catch`-Blöcke zu einem `try`-Block angegeben werden. Sie werden in der angegebenen Reihenfolge berücksichtigt. Man beachte, dass auch Unterklassen gefangen werden. Daher müssen Spezialfälle zuerst abgefangen werden.

2. Das konkrete Ausnahmeobjekt kann über `aExc` im catch-Block referenziert werden.

3. `try-catch`-Blöcke können geschachtelt werden und ihrerseits wieder `throw` Anweisungen enthalten. Die Auflösung der `try-catch`-Blöcke erfolgt von innen nach aussen.

4. Ein Ausnahmeobjekt kann innerhalb eines `catch`-Blocks zur weiteren Verarbeitung an höhere `try`-Verschachtelungen weitergeworfen werden. Dies erfolgt durch `throw;` (ohne Objekt).

5. Der `(...)-catch`-Block muss, wenn gewünscht, als letzter angegeben werden. Man beachte, dass dieser Block alle Exceptions (auch vom System ausgelöste) fängt und kein Referenzobjekt zur Verfügung stellt. Dieser Exception-Handler sollte nur sehr vorsichtig benutzt werden. Es empfiehlt sich hier meist, die Ausnahme nach Ausführung der spezifischen Behandlung an höhere (System-) Exception-Handler weiterzuwerfen.

6. Wird eine Exception ohne Überwachung durch einen entsprechenden Exception Handler geworfen, erfolgt eine Systemfehlermeldung mit Programmabbruch.

7. Beim Werfen der Ausnahme wird direkt auf den entsprechenden `catch`-Block verzweigt. Die restlichen Anweisungen der werfenden Funktion und des `try`-Blocks werden nicht mehr ausgeführt.

Das Werfen einer Ausnahme beeinflusst die Schnittstelle einer Funktion, da die aufrufende Funktion einen `try`-Block und entsprechende Exception Handler bereitstellen muss. Daher ist es sinnvoll, die Ausnahmen, die geworfen werden können als Teil der Funktionsdeklaration zu spezifizieren. Die Syntax für eine erweiterte Funktionsdeklaration lautet dann:

<*Rückgabetyp*> <*Funktionsname*> (<*Parameterliste*>) `throw` (<*Ausnahmeliste*>);

also z. B.

```
void fkt(int i) throw (exc1, exc2);
```

Dies garantiert dem Aufrufer, dass die Funktion `fkt` nur Ausnahmen aus den Klassen (Typen) `exc1` und `exc2` und davon abgeleiteten Klassen wirft. Der Versuch, eine andere Ausnahme zu werfen, führt dann zu einem Übersetzungsfehler. Die erweiterte Funktionsdeklaration muss sowohl bei der Definition als auch beim Prototyp benutzt werden. Eine Funktionsdeklaration ohne `throw` erlaubt das Werfen beliebiger Ausnahmen. Eine Funktion, die keine Ausnahmen wirft, kann dies durch den Zusatz `throw()` explizit in die Schnittstelle aufnehmen.

2.8 Namensbereiche

Ein Namensbereich (engl. *namespace*) ist ein Mechanismus zum Gruppieren logisch zusammengehöriger Deklarationen. Diese Technik wird vorwiegend bei größeren Projekten eingesetzt, bei denen durch Namensbereiche repräsentierte Module oft Hunderte von Funktionen, Klassen und Templates beinhalten. In diesem Praktikum werden zwar keine eigenen Namensbereiche definiert, da jedoch die C++ Standardbibliothek diese Technik einsetzt (alles, was zu ihr gehört, liegt im Namensbereich std), werden sie in diesem Kapitel kurz beschrieben.

Ein Namensbereich ist ein Gültigkeitsbereich mit Namen, und somit einer Klasse sehr ähnlich. Im Prinzip ist eine Klasse ein Namensbereich mit einigen zusätzlichen Sprachmitteln. Folglich treffen die üblichen Regeln für Gültigkeitsbereiche auch für Namensbereiche zu.

Beispiel:

```
#include <iostream>
#include <string>

namespace name1
{
  double f();
  int g(int);
}

namespace name2
{
  void h(std::string);
  int zahl;
}
```

```
double name1::f() { /*...*/ }
int name1::g(int i) { /*...*/ }

void name2::h(std::string s)
{
  std::cout << s << name1::f() << " "
            << name1::g(zahl)<< std::endl;
}
```

Anmerkungen:

- Der Qualifizierer `name2` bei der Implementierung von `h` ist notwendig, um festzuhalten, dass die Funktion `h` zu `name2` gehört und nicht eine globalen Funktion ist.
- Die Variable `zahl` braucht nicht qualifiziert zu werden, da sie zum selben Namensbereich (`name2`) gehört wie die Funktion `h` selbst.
- Die Funktionen `f` und `g` gehören zum Namensbereich `name1` und müssen somit qualifiziert werden.
- `string`, `cout` und `endl` gehören zur C++ Standardbibliothek und somit zum Namensbereich `std`.

Der Sinn von Namensbereichen ist an dieser Stelle vielleicht nicht ersichtlich, aber wenn man sich ein großes Projekt mit vielen Hunderten von Klassen und Funktionen vorstellt, wird einem schnell klar, dass durch Namensbereiche Namenskollisionen vermieden werden, die gerade bei großen Projekten nur schwer zu finden sind.

Wird ein Name häufig außerhalb seines Namensbereichs benutzt, wird es schnell lästig und auch unübersichtlich ihn immer wieder zu qualifizieren. Dies kann mit Hilfe der `using`-Deklaration vermieden werden, die ein lokales Synonym erzeugt.Es können auch alle Namen eines Namensbereichs mit der `using`-Direktive verfügbar gemacht werden. Man betrachte sich für die beiden Varianten von `using` (Direktive bzw. Deklaration) folgendes Beispiel:

```
#include <iostream>
#include <string>

using namespace std;  // using Direktive

namespace name1
{
  double f();
  int g(int);
}

namespace name2
{
  void h(std::string);
  int zahl;
```

```
  using name1::f;      // using Deklaration
}

double name1::f() { /*...*/ }
int name1::g(int i) { /*...*/ }

void name2::h(string s)
{
  using name1::g;      // using Deklaration
  cout << s << f() << " " << g(zahl)<< endl;
}
```

Anmerkungen:

- Aufgrund der `using`-Direktive können alle Namen der C++ Standardbibliothek in diesem Beispiel ohne Qualifizierer benutzt werden. Da diese Form der Veröffentlichung von Namen den Mechanismus der Namensbereiche zunichte macht, sollte eine globale `using`-Direktive eher sparsam eingesetzt werden. Vor allem sollte sie wann immer möglich in Headerdateien vermieden werden, weil automatisch alle Dateien, die diese Datei einbinden, sie auch beinhalten.
- Es ist meistens eine gute Idee, eine `using`-Deklaration so lokal wie möglich zu halten. Speziell sollte man sich bei `using`-Deklarationen innerhalb einer Namensbereichsdefinition überlegen, ob alle Funktionen einen Bezug zu dem Synonym haben.
- Da im Praktikum nur mit dem Namensraum `std` der Standardbibliothek gearbeitet wird, kann dort zur Vereinfachung von dieser Regel abgewichen werden und die Anweisung

  ```
  using namespace std;
  ```

  in allen .cpp- und .h-Dateien nach den Includes der Standardbibliothek eingesetzt werden.

3 Die C++ Standardbibliothek

In C++ können sämtliche Standard-C-Funktionen benutzt werden. Dazu muss jeweils das entsprechende Headerfile vorher mit

```
#include <file.h>
```

eingebunden werden, z.B. `#include <math.h>` für die Mathematikfunktionen. Die Headerfiles enthalten jeweils die Deklarationen.

Wo es möglich ist, sollten aber in C++ die entsprechenden alternativen Klassenbibliotheken benutzt werden. Diese können durch Einbinden des entsprechenden Headerfiles (ohne .h) bereitgestellt werden:

```
#include <file>
```

Für Ein- und Ausgabe sollen **nicht** die `stdio`-Funktionen wie `putchar(), printf()` etc. benutzt werden, sondern die C++ Klassen-Bibliothek `iostream`.

3.1 Ein- und Ausgabe

Für Ein- und Ausgabe bietet C++ die sogenannten Streams. Dies sind Klassen bzw. vorgefertigte Templates, die diese Funktionen beinhalten. Ein großer Vorteil der Streams ist, dass die Ein- und Ausgabe über die überladenen Operatoren `>>` und `<<` realisiert wird, womit eine übersichtliche Programmierung und eine einfache Erweiterung um benutzerdefinierte Datentypen ermöglicht wird.

Im einfachsten Fall werden Sie die Ausgabe auf den Bildschirm und die Eingabe von der Tastatur benötigen. Dazu stellt C++ die Klassen `istream` und `ostream` sowie die Objekte

`cin`	Standardeingabe (Tastatur)
`cout`	Standardausgabe (Bildschirm)
`cerr`	Fehlerausgabe (Bildschirm)

zur Verfügung .

Um die Streams zu benutzen, müssen Sie

```
#include <iostream>
```

einbinden. (Beachte: Ohne Extension .h)

3.1.1 Ausgabe

Eine Ausgabe erfolgt durch die Anweisung:

```
cout << object;
```

Mehrere Ausgaben können auch direkt miteinander verbunden werden:

```
cout << object << object << ... << object;
```

Der Operator << ist bereits für alle eingebauten Datentypen definiert, die Ausgabe erfolgt immer entsprechend des jeweiligen Objekts.

Beispiel:

```
double x = 25.391;
int i = 10;
char s[] = "Test";

cout << "Dies ist die Zahl x: " << x << endl;
cout << "Dies ist ein " << s << " mit i = " << i << endl;
```

`endl` steht für neue Zeile, stattdessen kann auch \n in einem String verwendet werden. Die Ausgabe des Beispiels ist:

```
Dies ist die Zahl x: 25.391
Dies ist ein Test mit i = 10
```

3.1.2 Formatierte Ausgabe

Die Stream-Klassen bieten in der Streams-2.0-Implementation folgende Methoden zur Manipulation der Ausgabe:

`width()`	Lesen/Setzen der Feldbreite für Ausgabe
`fill()`	Lesen/Setzen des Füllzeichens für Ausgabe
`flags()`	Lesen/Setzen der Flags für Ausgabe
`setf()`	Setzen von einzelnen Flags
`unsetf()`	Löschen von einzelnen Flags
`precision()`	Nachkommastellen bei Fließkommaausgabe

Das Setzen der Feldbreite hat nur Einfluss auf die nächste Ausgabeoperation!!!

Ohne Angabe eines Parameters liefern `width()`, `fill()`, `flags()` und `precision()` jeweils den aktuellen Wert.

Für die Ausgabeformatierung stehen folgende Flags zu Verfügung:

`ios::skipws`	führende Spaces beim Einlesen ignorieren (default)
`ios::noskipws`	führende Spaces beim Einlesen nicht ignorieren
`ios::left`	Linksbündige Ausgabe
`ios::right`	Rechtsbündige Ausgabe
`ios::internal`	Vorzeichen linksbündig, Wert rechtsbündig
`ios::dec`	Dezimale Ausgabe
`ios::oct`	Oktale Ausgabe
`ios::hex`	Hexadezimale Ausgabe
`ios::showbase`	Ausgabe der Zahlenbasis
`ios::showpoint`	Ausgabe aller Nachkommazahlen, auch wenn 0
`ios::showpos`	Ausgabe von + vor positiven Werten
`ios::uppercase`	„E“ und „X“ statt „e“ und „x“
`ios::scientific`	Darstellung: `d.ddddedd`
`ios::fixed`	Darstellung: `dddd.dd`
`ios::unitbuf`	Ausgabe nach jeder Operation
`ios::stdio`	Ausgabe nach jedem Zeichen

Als Masken für `setf()` stehen außerdem zur Verfügung:

`ios::basefield`	Alle Zahlenbasis-Flags
`ios::adjustfield`	Alle Ausrichtungs-Flags
`ios::floatfield`	Alle Fließkomma-Flags

Beispiele für die Anwendung der Methoden:

```
double x = 19.275;
int i = 125;
cout.width(6);
cout.precision(3);
cout.setf(ios::fixed, ios::floatfield);
cout << x << endl;
cout.width(10);
cout.precision(5);
cout.setf(ios::fixed, ios::floatfield);
cout << x << endl;
cout.width(10);
cout << i << endl;
cout.width(6);
cout << i << endl;
```

Ausgabe:

```
19.275
  19.27500
       125
   125
```

Um das ganze etwas zu vereinfachen, existieren die sogenannten IO-Manipulatoren, die das Setzen der Ausgabeparameter direkt als Ausgabeelemente über << ermöglichen. Um sie zu verwenden, muss

```
#include <iomanip>
```

eingebunden werden. Die zur Verfügung stehenden IO-Manipulatoren sind:

`setbase()`	Setzen der Zahlenbasis
`setfill()`	Setzen des Füllzeichens für die Ausgabe
`setprecision()`	Nachkommastellen bei Fließkommaausgabe
`setw()`	Setzen der Feldbreite für die Ausgabe
`setiosflags()`	Setzen der `ios`-Flags
`resetiosflags()`	Rücksetzen der `ios`-Flags

Die Parameter entsprechen denen der o.g. Methoden. Am Beispiel ist nochmals die obige Ausgabe realisiert:

```
double x = 19.275;
int i = 125;
cout << setw(6) << setprecision(3)
     << setiosflags(ios::fixed) << x << endl;
cout << setw(10) << setprecision(5)
     << setiosflags(ios::fixed) << x << endl;
cout << setw(10) << i << endl;
cout << setw(6) << i << endl;
```

Alle Parameter sind unabhängig voneinander. Das bedeutet zum Beispiel, dass man für einen Wechsel von links- auf rechtsbündige Ausgabe `left` zurücksetzen **und** `right` setzen muss. Ansonsten ist das Resultat nicht vorhersehbar.

3.1.3 Eingabe

Möchte man umgekehrt den Wert einer Variablen von der Tastatur einlesen, so existiert dafür der >>-Operator. Dieser funktioniert analog zur Ausgabe:

```
cin >> variable;
```

Die Eingabe muss mit der Taste Enter abgeschlossen werden. Ausgewertet wird die Eingabe für jede Variable soweit, wie die gelesenen Zeichen zum Typ der Variable passen: z.B. endet die Eingabe für `int` bei allen nicht Ziffern, bei `char[]` beim Leerzeichen. Alle Leerzeichen innerhalb der Eingabe werden als Trennzeichen behandelt, soweit nicht das `skipws`-Flag gesetzt ist.

Ein als `enum` definierter Datentyp kann nicht einfach mit dem >>-Operator eingelesen werden. Stattdessen müssen Sie eine Varibale (z.B. ein `int`) einlesen und mir einer if-Afrage oder switch den entsprechenden `enum`-Wert zuweisen.

Variablen, die von einem `enum`-Datentyp abgeleitet wurden, kann man nicht direkt einlesen (mit >>-Operator) oder ausgeben. Zum Einlesen muss man eine entsprechende `int`-Varibale einlesen. Für die Zuweisung muss man dann casten oder eine if-then-else-Abfrage benutzen.

Beispiel:

```
CentMuenzen eG=C50;
int iG;
cout << "Gib Cent-Muenze:" << endl;
cin >> iG;
eG = (CentMuenzen) iG;
cout << "Cent-Muenze: " << (int) eG << endl;
```

3.2 File I/O

C++ unterstützt zwei Arten von Files: sequentielle Files und Files mit wahlfreiem Zugriff. In diesem Praktikum werden nur erstere behandelt.

Es gibt lediglich drei mögliche Aktionen, die man auf einem sequentiellen File ausführen kann: Daten aus dem File einlesen, Daten in ein neues File schreiben und Daten an ein existierendes File anhängen. Ein sequentielles File ähnelt also einem Tonband. Genausowenig wie bei einem Tonband die aufgenommenen Lieder umgeordnet werden können, ohne das ganze Band zu löschen, können auch bei einem sequentiellen File die Daten nicht rearrangiert werden.

3.2.1 Schreiben von Daten in ein File

Die Ausgabe von Daten in ein File erfolgt ganz analog zur Ausgabe von Daten auf dem Bildschirm. Betrachten Sie zunächst folgendes Beispielprogramm:

```
#include <fstream>
#include <iostream>

void main(void)
{
  // Testvariablen
```

```
    char name[] = "Arthur";
    int semester = 42;
    double schnitt = 4.2;

    // Öffnen des Files für die Ausgabe
    ofstream outfile("Ausgabedatei.dat");

    // Schreibe die Daten in das File
    outfile << "Name: " << name << endl;
    outfile << "Semester: " << semester << endl;
    outfile << "Notenschnitt: " << schnitt << endl;
}
```

Nach der Initialisierung einiger Testvariablen erzeugt das Programm ein Objekt mit Namen `outfile`, welches den Datentyp `ofstream` (für *output file stream*) hat. `ofstream` ist eine Unterklasse von `ostream`, der allgemeinen Klasse, für die der `<<`-Operator definiert wird. `cout` ist ein Objekt von `ostream`. Dieser Typ ist im Headerfile `fstream` deklariert, welches daher zu Beginn eingebunden werden muss. Als Parameter erhält der Konstruktor eines `ofstream`-Objektes den Namen des zu erzeugenden Files (hier: *Ausgabedatei.dat*). Das Objekt `outfile` repräsentiert nun das File, in das die Ausgabe erfolgen soll. Selbstverständlich können Sie auch einen beliebigen anderen Namen für das Objekt verwenden. Ganz analog zur Ausgabe auf dem Bildschirm können die Daten nun mit Hilfe des Ausgabeoperators `<<` in das File geschrieben werden.

Beachten Sie, dass das File weder explizit geöffnet noch geschlossen werden muss. Der Konstruktor öffnet das File beim Erzeugen eines `ofstream`-Objektes automatisch und der Destruktor schließt das File wieder, wenn der Gültigkeitsbereich des Objektes verlassen wird.

Das Programm erzeugt bei seiner Ausführung die Datei *Ausgabedatei.dat* mit folgendem Inhalt:

```
Name: Arthur
Semester: 42
Notenschnitt: 4.2
```

3.2.2 Lesen von Daten aus einem File

Genauso einfach wie das Schreiben in ein File ist mit C++ auch das Lesen aus einem File. Das folgende Programm liest die Daten aus der Datei *Ausgabedatei.dat* wieder ein und gibt sie auf dem Bildschirm aus:

```
#include <fstream>
#include <iostream>

void main(void)
{
  // Testvariablen
```

```
    char name[7];
    int semester;
    double schnitt;
    char infoString[3][255];
    // Öffnen des Files zum Lesen
    ifstream infile("Ausgabedatei.dat");
    // Lese die Daten aus dem File
    infile >> infoString[0] >> name;
    infile >> infoString[1] >> semester;
    infile >> infoString[2] >> schnitt;
    // Schreiben auf den Bildschirm
    cout << infoString[0] << name << endl;
    cout << infoString[1] << semester << endl;
    cout << infoString[2] << schnitt << endl;
}
```

Nach der Definition der Variablen, welche die einzulesenden Daten aufnehmen sollen, wird ein Objekt vom Typ `ifstream` (für *input file stream*) erzeugt, welches automatisch das gewünschte File zum Lesen öffnet. `ifstream` ist eine Unterklasse von `istream`, der allgemeinen Klasse, für die der >>-Operator definiert wird. `cin` ist Objekt von `istream`. Der überladene Eingabeoperator >> wird dann verwendet um die Daten aus dem File in die Variablen einzulesen. Zu beachten ist hierbei, dass Leerzeichen, Tabulatoren und Zeilenumbrüche als Trennzeichen dienen um die einzelnen Daten im File voneinander abzugrenzen. Ein in der Datei gespeicherter String darf also keine Leerzeichen enthalten, da er beim Lesen sonst als zwei (oder mehr) Strings interpretiert wird. Auch ist darauf zu achten, dass die Datentypen der Variablen mit den Datentypen der gelesenen Werte übereinstimmen, da es ansonsten zu einem Lesefehler kommt. Das Programm gibt folgendes auf dem Bildschirm aus:

```
Name:Arthur
Semester:42
Notenschnitt:4.2
```

Ist die Menge der einzulesenden Daten nicht von vornherein bekannt, so muss es eine Möglichkeit geben zu erkennen, wann das Dateiende erreicht ist. Dies kann mittels der Funktion `eof()` getestet werden. Folgendes Programmfragment liest solange `int`-Werte aus einer Datei ein, bis das Dateiende erreicht ist:

```
// Variable zum Einlesen
int zahl;
// Öffnen der Datei
ifstream fin("Testdatei.dat");
// Lies, bis Dateiende erreicht
do {
fin >> zahl;
cout << "Gelesene Zahl : " << zahl << endl;
} while (!fin.eof());
```

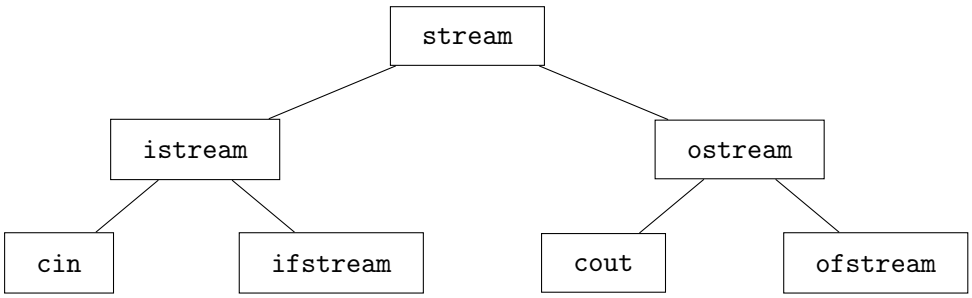


Abbildung 3.1: Klassenhierarchie stream

Beachten Sie, dass das Dateiende nicht beim Lesen der letzten Zeile, sondern erst beim darauffolgenden Versuch eine Zeile zu lesen, erkannt wird.

Zur Verdeutlichung der Zusammenhänge sei in Abbildung 3.1 nochmal die Klassenstruktur der IO dargestellt.

3.2.3 Abfangen von Fehlern

Beim Lesen von Files und Schreiben in Files können Fehler auftreten, die abgefangen werden müssen. Beispielsweise könnte man versuchen, aus einem nicht existierenden File zu lesen, das Floppy-Laufwerk könnte nicht verfügbar oder die Diskette physikalisch defekt sein. Ein `ifstream`- oder `ofstream`-Objekt besitzt vier member functions die zum Abfangen solcher Fehler verwendet werden können: `good()`, `eof()`, `fail()` und `bad()`.

Wir konzentrieren uns hier auf die Verwendung von `good()`. Diese Memberfunktion liefert `true` zurück, wenn kein Fehler auftrat. Sobald es bei einer Ein-/Ausgabefunktion (Öffnen, Lesen, Schreiben) zu einem Fehler kommt, liefert die Funktion `false` zurück.

Die folgende Funktion `liesFile` liest 100 int-Werte aus einer Datei ein. Konnte die Datei mit dem als Parameter übergebenen Namen nicht geöffnet werden oder kam es während des Lesens zu einem Fehler, so liefert die Funktion `false` zurück, ansonsten `true`.

```
bool liesFile(const char* dateiname, int zahl[]) {
  // Öffne die Datei zum Lesen
  // (weitere Flags siehe Datei-Flags)
  ifstream fin(dateiname);
  // Prüfe, ob Öffnen erfolgreich war
  if (!fin.good())
    return false;
  // Lies die Werte ein
  for (int i = 0; i < 100; i++)
    fin >> zahl[i];
  // Liefere Fehlercode zurück
  // (true, falls alles OK war, sonst false)
  return fin.good();
}
```

3.2.4 Datei-Flags

Für den genauen Modus der Dateibearbeitung existieren einige Flags, die in der Klasse `ios` wie folgt definiert werden:

`ios::in`	Lesen (Default bei `ifstream`)
`ios::out`	Schreiben (Default bei `ofstream`)
`ios::app`	Anhängen
`ios::ate`	ans Ende Positionieren (''`at end`'')
`ios::trunc`	alten Dateiinhalt löschen
`ios::nocreate`	Datei muss existieren
`ios::noreplace`	Datei darf nicht existieren

Die Flags können mit dem `|`-Operator verknüpft werden. Übergeben werden sie dann dem Konstruktor als optionales zweites Argument. Die folgende Anweisung öffnet z.B. eine Datei, die bereits existieren muss, zum Schreiben und sorgt dafür, dass der geschriebene Text an das Ende der Datei angehängt wird:

```
fstream datei("xyz.out", ios::out | ios::app | ios::nocreate);
```

3.2.5 Fehlerzustände, Stream-Status

Für den prinzipiellen Zustand eines Streams werden in der Klasse ios Flags definiert:

`ios::goodbit`	alles in Ordnung, kein (anderes) Bit gesetzt
`ios::eofbit`	End-Of-File
`ios::failbit`	Fehler: letzter Vorgang nicht korrekt abgeschlossen
`ios::badbit`	fataler Fehler: Zustand nicht definiert

Bei `ios::goodbit` von einem Flag zu reden, ist etwas verwirrend, da es typischerweise den Wert 0 besitzt und damit automatisch anzeigt, ob ein anderes Flag gesetzt ist.

Der Unterschied zwischen `ios::failbit` und `ios::badbit` ist klein, aber durchaus von Interesse:

- `ios::failbit` wird i.A. gesetzt, wenn ein Vorgang nicht korrekt durchgeführt werden konnte, der Stream aber prinzipiell in Ordnung ist. Typisch dafür sind Formatfehler beim Einlesen. Wenn z.B. ein Integer eingelesen werden soll, das nächste Zeichen aber ein Buchstabe ist, wird dieses Flag gesetzt.
- `ios::badbit` wird i.A. gesetzt, wenn der Stream prinzipiell nicht mehr in Ordnung ist oder Zeichen verlorengegangen sind. Dieses Flag wird z.B. beim Positionieren vor einen Dateianfang gesetzt.

3.3 Strings

Mit einem String der Standardbibliothek von C++ wird die Bearbeitung von Zeichenketten gegenüber der „klassischen“ Form über `char`-Felder wesentlich erleichtert. Voraussetzung für die Benutzung ist die Einbindung der Bibliothek `<string>` und die Benutzung des Standardnamensraums durch

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

Es können dann Elemente der neuen Klasse `string` mit verschiedenen Konstruktoren erstellt werden:

1. `string myString`

   erstellt einen leeren String.

2. `string myString(10, 'A')`

   erstellt einen String mit 10 Zeichen und dem Inhalt `''AAAAAAAAAA''`. Üblicher als eine Initialisierung mit `'A'` ist natürlich eine mit Blank.

3. `string myString("Dies ist ein String")`

   erstellt einen String mit 19 Zeichen und dem Inhalt *"Dies ist ein String"*.

Die Ein-/Ausgabe der Strings erfolgt über die überlagerten `<<` bzw. `>>`-Operatoren. Beim Eingabeoperator wird dabei ggf. der String auf die benötigte Größe verlängert.

Beispiel:

```
#pragma warning(disable:4786)
#include <string>
#include <iostream>

using namespace std;

void main()
{
  string s1(10, 'A');
  string s2("Dies ist ein String.");
  string s3;

  cout << s1 << endl;
  cout << s2 << endl;
  cout << "Geben Sie einen String ein:" << endl;
  cin >> s3;
  cout << s3 << endl;
}
```

Die Ausgabe dieses Programms ist:

```
AAAAAAAAAA
Dies ist ein String.
Geben Sie einen String ein:
(Eingabe:abcd) abcd
```

Hinweis: Benutzen Sie bei Einsatz der Bibliothek `<string>` sowie im weiteren aller STL-Klassen `#pragma warning(disable:4786)`. Damit verhindern Sie die Ausgabe von Warnungen bei zulangen Variablennamen. In diesen Bibliotheken werden zur Vermeidung von Überschneidungen lange Variablennamen benutzt.

Auch die Vergleichsoperatoren, sowie der `+` - und der `+=` -Operator sind in intuitiver Weise für Strings überlagert. Die Vergleichsoperatoren liefern das Ergebnis des zeichenweisen Vergleichs anhand der zugrundeliegenden Codierung (`ASCII`), die `+` -Operatoren dienen zur Aneinanderreihung (Konkatenation) zweier Strings.

Beispiel:

```
#pragma warning(disable:4786)
#include <string>
#include <iostream>

using namespace std;

void main()
{
  string s1("ABC");
  string s2;
  string s3;

  s2 = "AB";
  s3 = s1 + s2;
  cout << s3 << endl << "Maximum: ";
  if (s1 > s2)
    cout << s1;
  else
    cout << s2;
}
```

Die Ausgabe dieses Programms ist:

```
ABCAB
Maximum: ABC
```

Der Zugriff auf die einzelnen Zeichen eines Strings erfolgt mit dem `[ ]`-Operator, wobei auch hier mit dem Index 0 auf das erste Zeichen zugegriffen wird. Im Gegensatz zu den C-Strings wird der String aber nicht durch `\0` beendet, sondern das Ende wird durch die aktuelle Länge des Strings bestimmt. Die Länge kann durch die Methode `length()` bestimmt werden.

Beispiel:

```
#pragma warning(disable:4786)
#include <string>
#include <iostream>

using namespace std;

void main()
{
  string s1("ABC");
  int i;

  for (i = 0; i < s1.length(); i++)
    cout << s1[i];
}
```

Neben diesem direkten Zugriff auf die einzelnen Elemente kann auch sequentiell auf die Zeichen eines Strings zugegriffen werden. Dies geschieht über die Inkrement-/ Dekrementoperatoren `++` und `--` auf einen Iterator auf die Elemente von String. Einen entsprechenden Iterator deklariert man durch `string::iterator`. Spezielle Methoden (`begin()` und `end()`) liefern jeweils Iteratoren auf den Anfang und hinter das Ende des Strings. Weitere Informationen zu Iteratoren finden Sie im Kapitel 3.4.2. Das folgende Beispiel erzeugt die Ausgabe:

```
A B C D
```

Beispiel:

```
  #include <string>
  #include <iostream>

  using namespace std;

  void main()
  {
    string s1("ABCD");
    string::iterator it;

    for (it = s1.begin(); it < s1.end(); it++)
      cout << *it << ' ';
  }
```

Für die komfortable Bearbeitung von Stringobjekten stehen diverse Methoden zur Verfügung. Die in Tabelle 3.1 aufgeführte Auswahl ist nicht vollständig und für einige Methoden gibt es neben den dargestellten auch noch weitere Aufrufvarianten mit anderen Parametertypen (siehe dazu die Literatur oder Onlinehilfe).

Methode	Funktionalität
`lenght(), size()`	Liefert die aktuelle Länge des Strings
`empty()`	Liefert `true`, falls leerer String
`insert(pos, str)`	Fügt `str` vor `pos` im String ein
`erase(pos, anzahl)`	Löscht im String ab `pos` `anzahl` Zeichen
`replace(pos, anzahl, str)`	Ersetzt ab `pos` bis zu `anzahl` Zeichen durch die Zeichen von `str`
`find(str, pos)`	Liefert die Position des **ersten** Auftretens von `str` im String ab `pos` (-1 falls nicht vorhanden)
`rfind(str, pos)`	Liefert die Position des **letzten** Auftretens von `str` im String bei oder vor `pos` beginnend (-1 falls nicht vorhanden)
`find_first_of(str, pos)`	Liefert die Position des **ersten** Auftretens eines Zeichens von `str` im String ab `pos` (-1 falls nicht vorhanden)
`find_last_of(str, pos)`	Liefert die Position des **letzten** Auftretens eines Zeichens von `str` im String bei oder vor `pos` (-1 falls nicht vorhanden)
`find_first_not_of(str, pos)` `find_last_not_of(str, pos)`	Liefert analog die Position des Zeichens im String, das nicht in `str` vorkommt (-1 falls alle Zeichen im String in `str` vorkommen)
`substr(pos, laenge)`	Liefert einen Substring ab `pos` im String mit der angegebenen `laenge`

`compare(str)`
`compare(pos, anz, str)`
`compare(pos, anz, str, pos1, anz1)`
	Vergleicht den String zeichenweise mit `str` und liefert als Ergebnis:
	-1 falls `str` $<$ aktuellem String
	0 falls `str` $=$ aktuellem String
	1 falls `str` $>$ aktuellem String
	Durch die übrigen Parameter kann der Vergleich auf einen Teilstring des aktuellen Strings (`pos, anz`) bzw. des Vergleichstrings (`pos1, anz1`) eingeschränkt werden.

Tabelle 3.1: Bearbeitungsfunktionen für Strings

Es ist zu beachten, dass Stringobjekte und C-Strings nicht verwechselt oder gemischt werden dürfen, auch wenn für die meisten obigen Stringmethoden auch entsprechende überlagerte Methoden für C-Strings als Parameter existieren. Für Funktionen, die C-Strings erwarten (z.B. Funktionen der Windowsbibliothek) existieren aber noch keine entsprechenden Aufrufe für Stringobjekte. Allerdings können sie in beide Richtungen umgewandelt werden:

- C-Strings in Stringobjekte durch Aufruf des entsprechenden Konstruktors

- Stringobjekte in C-Strings durch die Methode `c_str()`

3.3.1 String-Streams

Ein Stream kann einem String zugeordnet werden. Damit kann man unter Verwendung der Formatierungsmöglichkeiten aus einem String lesen (wie z.B. von der Standardeingabe `cin`) oder in einen String schreiben (wie auf die Standardausgabe `cout`). Solche Streams werden String-Streams (Klasse `stringstream`) genannt. Sie werden in `<sstream>` definiert.

Wie bei den Stream-Klassen für Dateien gibt es analog Stream-Klassen für Strings:

- `istringstream` - für Strings, aus denen gelsen wird (input string stream)
- `ostringstream` - für Strings, in die geschrieben wird (output string stream)
- `stringstream` - für Strings, die zum Lesen und Schreiben verwendet werden

Die Klassen erben von `iostream` und übernehmen daher auch den Funktionsumfang der Oberklasse, wie z.B. die Ein-/Ausgabeoperatoren. Weiterhin stellt `stringstream` folgende Funktionen zur Verfügung:

- `str()` - liefert den Ausgabestrom als String zurück
- `rdbuf()` - liefert die Adresse des internen String-Buffers

Beispiel:

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

int main() {
  string namen[3];

  for (int i = 0; i < 3; i++) {
    stringstream s;          // Beachte: stringstream lokal
    // Ausgabe "Objekt" + Nummer auf Stringstream, z.B. "Objekt1"
    s << "Objekt" << i + 1;
    namen[i] = s.str();
  }

  for (int i = 0; i < 3; i++)      // Kontrollausgabe
    cout << namen[i] << endl;
}
```

Die Ausgabe dieses Programms ist:

```
Objekt1
Objekt2
Objekt3
```

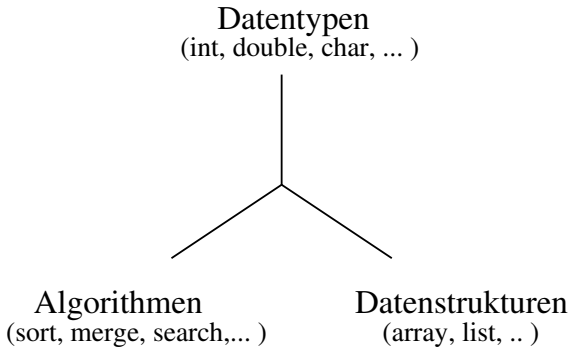


Abbildung 3.2: Softwarekomponenten

3.4 STL (Standard Template Library)

Die „Standard Template Library“ (STL) ist eine C++-Bibliothek, die ursprünglich im HP-Entwicklungslabor in Palo Alto entwickelt wurde, mittlerweile aber von fast allen C++-Entwicklungsumgebungen angeboten wird und 1994 in den erweiterten C++-Standard aufgenommen wurde. Sie ermöglicht dem Programmierer, einfachen, effizienten und wiederverwendbaren Code auf der Basis einer intensiven Nutzung von Templates (bzw. daraus generierten Klassen) zu erstellen. Für die wichtigsten Datenstrukturen (Vektoren, Verzeichnisse, Listen, Bäume, ...) und allgemeinen Algorithmen (Suchen, Sortieren, ...) werden entsprechende Implementierungen angeboten. Im Folgenden sollen einige STL-Elemente vorgestellt werden. Diese Beschreibung ist jedoch nicht vollständig und es sei hier auf die Literatur- bzw. WWW-Linkliste hingewiesen.

Die Idee, die hinter STL steckt, soll an folgender Überlegung deutlich werden. Jede Software setzt sich aus verschiedenen Komponenten des in Abbildung 3.2 dargestellten dreidimensionalen Raumes zusammen. Damit müssen auch entsprechend viele verschiedene Codevarianten entworfen werden: z.B. je ein Array für `int`, für `double`, für `char` usw., ebenfalls je eine Liste für alle Datentypen, dann ein Sortieralgorithmus für `int`-Arrays, für `int`-Listen, für `double`-Arrays usw. Dies lässt sich vereinfachen, in dem man zunächst Datenstrukturen entwickelt, die beliebige Datentypen enthalten können und dann auf diesen Datenstrukturen Algorithmen, die wieder mit beliebigen Datenstrukturen arbeiten, die bestimmte grundlegende Funktionen verstehen. Diese komponentenbasierte Softwarebibliothek ist die Idee von STL.

STL kennt folgende Komponentenarten:

Container	Klasse zur Beschreibung einer Datenstruktur mit Methoden zum Aufnehmen, Löschen und Verwalten von Objekten (z.B. doppeltverkettete Listen)
Iteratoren	Abstrahierter Zeigerzugriff auf Behälter, um allgemeine Algorithmen entwerfen zu können (z.B. <- oder ++ -Operator)
Algorithmen	Funktionsvorschrift, die auf verschiedenen Datenstrukturen arbeiten kann (z.B. sortieren)
Funktionsobjekte	Funktionen, die in Objekte gekapselt werden

3.4.1 Container

Container sind Klassen zur Aufnahme von Objekten beliebigen Typs mit Methoden zum Einfügen, Löschen und Verwalten dieser Objekte. Verschiedene Container unterscheiden sich in der Art und Weise, wie die Objekte zueinander arrangiert sind, ob die Elemente sortiert sind und wie der Zugriff auf die einzelnen Elemente erfolgt. Entsprechend unterscheidet die STL *Sequence Container* und *Associative Container*.

Sequence Container ist die Organisation einer endlichen Menge von Objekten (alle desselben Typs) in einer strikt linearen Ordnung. Der Zugriff auf die Elemente ist nur von einem Element zum nächsten bzw. vorigen, bzw. über die genaue Position eines Elementes möglich. STL stellt dabei die Typen Feld (`vector`), verkettete Liste (`list`), und Schlangen (`deque`) zur Verfügung.

Zu den gerade beschriebenen Containern gibt es noch sogenannte *Adapter*. Diese Container-Adapter stellen zu einem Basis-Container eine eingeschränkte Schnittstelle zur Verfügung. Dabei handelt es sich um den `stack` und die `queue` bzw. `priority_queue`.

Associative Container bieten im Gegensatz dazu (zusätzlich) die Möglichkeit des schnellen Zugriffs auf einzelne Elemente über einen Schlüssel. STL kennt dabei die Typen `set` und `map`, deren Elemente einen eindeutigen Schlüssel besitzen sowie `multiset` und `multimap`, bei denen auch mehrere Elemente denselben Schlüssel besitzen dürfen. Set (Multiset) und Map (Multimap) unterscheiden sich dadurch, dass bei „Set“ der Schlüssel Bestandteil der Daten ist, während bei „Map“ Daten und Schlüssel unabhängig voneinander sind.

In Tabelle 3.2 finden Sie nochmal eine Zusammenfassung aller Container der STL. Im Praktikum werden wir nur die Containertypen `vector, list` und `map` verwenden, auf die wir nun näher eingehen wollen.

vector

Um in C++ (ohne STL) mehrere gleichartige Objekte zusammenzufassen, definieren wir ein entsprechendes Feld, z.B. für 5 `int`-Zahlen:

```
int feld[max_size];
```

Wenn wir dies codieren, müssen wir die maximale Anzahl der Elemente kennen (spätestens beim dynamischen Allokieren des Speichers). Mit Hilfe der STL können wir statt eines (festdimensionierten) Feldes einen entsprechenden Vektor definieren. Definiert ist der Vector in `<vector>` und wird dort wie folgt deklariert:

```
template<class T> class vector {};
```

Wie oben schon beschrieben handelt es sich hierbei um einen sequenziellen Container, dessen Größe sich nach Bedarf anpassen lässt:

```
vector<int> feld;
```

oder um eine Anfangsgröße des Vektors festzulegen:

```
vector<int> feld(5);
```

Container	Beschreibung
Sequence Container	
`vector`	lineare, benachbarte Speicherung; schnelles Einfügen nur am Ende
`deque`	lineare, nicht benachbarte Speicherung; schnelles Einfügen an Außenstellen
`list`	beidseitig verkettete Liste; schnelles Einfügen an beliebigen Stellen
Associative Container	
`multiset`	Datenmenge, schneller assoziativer Zugriff, Duplikate zulässig
`set`	wie `multiset`, jedoch keine Duplikate zulässig
`multimap`	Menge von Schlüssel/Wert-Paaren, schneller Zugriff über Schlüssel, Duplikate für Schlüssel zulässig
`map`	wie `multimap`, jedoch keine Duplikate für Schlüssel zulässig
Adapter	
`stack`	strikte FILO-Datenstruktur (Kellerspeicher)
`queue`	strikte FIFO-Datenstruktur (Warteschlange)
`priority_queue`	queue, die Elemente mit Priorität speichert und somit die Reihenfolge des Zugriffs bestimmt

Tabelle 3.2: Containerklassen in der Übersicht

Methoden	Funktionalität
`size_type size() const`	Liefert die aktuelle Anzahl der Elemente
`bool empty() const`	Liefert `true`, falls Vektor leer
`void push_back(const T& val)`	Fügt `val` am Ende des Vektors ein
`void pop_back()`	Löscht das letzte Element im Vektor
`void swap(vector<T>& vec)`	Tauscht den Inhalt mit dem von `vec`

Tabelle 3.3: Einige Funktionen von vector

Auch ein Initialwert kann bei der Definition direkt mitgegeben werden, z.B. 3 `float`-Elemente mit dem Wert `-1`:

```
vector<float> feld(3, -1.0);
```

Auf die einzelnen Elemente des Vektors kann mit dem `[]`-Operator zugegriffen werden, beginnend für das erste Element mit 0 (Beispiel: `feld[4]`). Der Zugriff auf nicht definierte Elemente führt auch hier (wie bei C++-Feldern) zu unvorhersehbaren Ergebnissen. Wo nötig, sollte also der Index zuvor auf Gültigkeit getestet werden bzw. eine eigene (sichere) Vektorklasse implementiert werden.

Tabelle 3.3 enthält einige grundlegende Funktionen des Vectors. Für eine vollständige Übersicht sei auch hier wieder auf die empfohlene Literatur verwiesen.

Beispiel:

```
#include <vector>
#include <iostream>
```

```
using namespace std;

void main() {
  vector<int> v1(3);
  int i;

  for (i = 0; i < 3; i++)
      v1[i] = i;
  for (i =0; i < v1.size(); i++)
    cout << v1[i] << ' ' ;
  cout << endl;

  v1.pop_back();
  for (i = 0; i < v1.size(); i++)
    cout << v1[i] << ' ' ;
  cout << endl;

  v1.push_back(5);
  v1.push_back(6);
  for (i = 0; i < v1.size(); i++)
    cout << v1[i] << ' ' ;
  cout << endl;
  return;
}
```

Die zugehörige Ausgabe ist dann:

```
0 1 2
0 1
0 1 5 6
```

Es ist möglich, einen Vektor direkt einem anderen zuzuweisen (=-Operator) und zwei Vektoren auf Gleichheit zu testen (==-Operator). Für Gleichheit müssen die beiden Vektoren gleich groß sein und alle Elemente übereinstimmen.

list

Eine Liste ist ein sequentieller Container, der sich optimal für das Einfügen und Löschen an beliebigen Stellen seiner Struktur eignet. Er ist in `<list>` definiert und weist folgende Deklaration auf:

```
template<class T> class list {};
```

Betrachten Sie Abbildung 3.3. Anders als bei einem Vector bietet eine Liste keinen Zugriff auf Indexbasis. Die Funktion begin() liefert einen Iterator auf das 1. Element der Liste. Dieser Iterator kann mit ++ und -- vor- und rückwärts bewegt werden. Das Ende der Liste kann durch die Funktion `end()` abgefragt werden. Die Beschreibung für Iteratoren finden Sie in Kapitel 3.4.2.

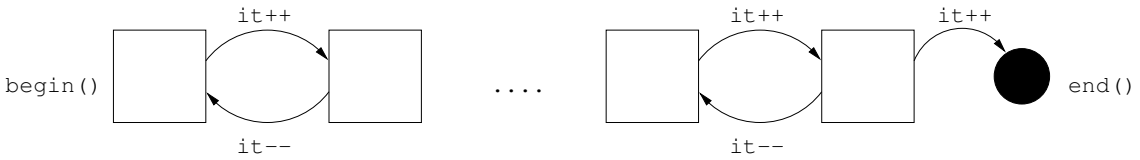


Abbildung 3.3: Durchlaufen einer Liste

Methoden	Funktionalität
`size_type size() const`	Liefert die aktuelle Anzahl der Elemente
`bool empty() const`	Liefert true, falls Liste leer
`iterator begin()`	Liefert Iterator, der auf das erste Element zeigt
`iterator end()`	Liefert Iterator, der hinter das letzte Element zeigt
`void push_back(const T& val)`	Fügt val am Ende der Liste ein
`void push_front(const T& val)`	Fügt val am Anfang der Liste ein
`void pop_back()`	Löscht das letzte Element
`iterator insert(iterator pos, const T& val)`	Fügt val an Position pos ein und gibt einen Iterator auf das neue Element zurück.
`void erase(iterator pos)`	Löscht das Element an Position pos.

Tabelle 3.4: Einige Funktionen von list

Die Definition einer Liste erfolgt z.B. für eine Liste mit `string`-Elementen durch:

```
list<string> meineListe;
```

Eine Liste besitzt dieselben Memberfunktionen zum Erweitern, Löschen und Abfragen der Größe wie ein Vektor. Um Elemente in die Liste einzufügen oder zu löschen, stehen zusätzlich die Funktionen `insert()` und `erase()` zur Verfügung (s. Tabelle 3.4). Diese Funktionen existieren zwar auch für Vektoren, führen dort aber zu einer Reallokation des gesamten Vektors und machen alle Iteratoren, die sich auf Elemente hinter der Einfüge-/Löschposition beziehen, ungültig. Sie sind also dort nur in Sonderfällen sinnvoll einsetzbar.

Zum umgekehrten Durchlaufen der Liste kann man einen reverse-iterator mit den zugehörigen Funktionen `rbegin()` (=letztes Element) und `rend()` (**vor** dem 1. Element) benutzen. Entsprechend gibt es auch die umgekehrten push-/pop-Funktionen: `push_front(val)` und `pop_front()`. Mit der Funktion `reverse()` kann die Reihenfolge der Elemente in der Liste umgekehrt werden.

Es ist möglich, eine Liste oder einen Vektor direkt einer anderen Liste zuzuweisen (=-Operator) und zwei Listen auf Gleichheit zu testen (==-Operator). Für Gleichheit müssen die beiden Listen gleich groß sein und alle (in der richtigen Reihenfolge) Elemente übereinstimmen.

Beispiel:

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

```
using namespace std;
typedef list<int> ListInt;

void main()
{
  ListInt v;
  ListInt::iterator itL;
  ListInt::reverse_iterator itR;

  v.push_back(7);            // v: 7
  v.insert(v.begin(), 3);    // v: 3 7
  v.insert(v.end(), 8);      // v: 3 7 8
  itL = v.begin();
  itL++;
  v.erase(itL);              // v: 3 8

  for (int i = 0; i < 3; i++)
    v.insert(v.begin(), i); // v: 2 1 0 3 8

  //Rückwärtsausgabe
  for (itR = v.rbegin(); itR != v.rend(); itR++)
    cout << setw(2) << *itR;

  return;
}
```

Ausgabe dieses Programms ist: `8 3 0 1 2`

Statt ein einzelnes Element in die Liste einzufügen, kann man auch alle Elemente des Intervalls [first, last) einfügen. Der notwendige Funktionsaufruf ist dann:

```
void insert(iterator pos, const T* first, const T* last)
```

Der durch `first` und `last` bestimmte Teil einer Liste (oder eines Vektors bzw. Feldes) wird an der Position `pos` eingefügt.

Beispiel:

```
#include <iostream>
#include <list>
#include <string>

using namespace std;
typedef list<string> ListString;

void main()
{
  ListString aSList1, aSList2;
  ListString::iterator itL;

  aSList1.push_back("gelb");
  aSList1.push_back("blau");
  aSList1.push_back("rot");

  aSList2.push_front("weiss");
  aSList2.push_front("lila");
  aSList2.push_front("schwarz");
```

```
  aSList1.insert(aSList1.begin(), aSList2.begin(), aSList2.end());

  for (itL = aSList1.begin(); itL != aSList1.end(); itL++)
    cout << *itL << ' ';

  return;
}
```

Ausgabe dieses Programms: `schwarz lila weiss gelb blau rot`

map

Maps sind assoziative Container zur Speicherung von Schlüssel/Werte-Paare. Dies ermöglicht die sortierte Speicherung der Daten und den schnellen Zugriff auf die Daten über den Schlüssel. Maps sind in `<map>` definiert und enthalten dort folgende Deklaration:

```
template<class Key, class Value, class Compare>
class map {};
```

Im Folgenden werden nur einige elementare Eigenschaften der Maps beschrieben. Für weitergehende Information sei wieder auf die Literatur verwiesen. Im einfachsten Fall erfolgt die Deklaration einer Map über:

```
map([Key], [Value])
```

also z.B.

```
map<string, int> MapStringInt;
```

zur Definition einer Map, die int-Zahlen enthält, auf die über einen String zugegriffen wird, oder

```
map<long int, Student> MapStd;
```

zur Definition einer Map, die Studentendaten enthält, auf die über eine `long int`-Zahl (z.B. Matrikelnummer) zugegriffen wird.

Über den Subskript-Operator `operator[]()` erfolgt dann der Zugriff auf die einzelnen Datenpaare. Die Funktionen `begin, end, rbegin, rend, empty` und `size` sind analog zu den übrigen Containern auch für Maps definiert. In Tabelle 3.5 finden Sie noch einige hilfreiche Funktionen für Maps.

Maps unterstützen Random-Access-Iteratoren und den =-Operator. Die Dereferenzierung eines Iterators liefert jeweils ein `pair`-Objekt aus Schlüssel und Wert. Mit `first` kann dann auf den Schlüssel, mit `second` auf den Wert zugegriffen werden. Man beachte, dass dies keine Memberfunktionen sind und sie daher ohne `()` geschrieben werden.

Beispiel:

Methoden	Funktionalität
`size_type count(const Key& key) const`	Liefert die Anzahl der Paare, deren Schlüssel mit `key` übereinstimmt
`iterator find(const Key& key) const`	Liefert Iterator auf ein Paar, falls key mit Schlüssel übereinstimmt. Sonst Iterator auf `end()`
`Value& operator[](const Key& key)`	Verknüpft `key` mit einem Standardwert, falls noch kein Wert für `key` vorhanden und gibt Zeiger auf diesen neuen Wert zurück. Ist Wert für `key` vorhanden, wird dieser Zeiger geliefert

Tabelle 3.5: Einige Funktionen von map

```
#include <iostream>
#include <string>
#include <map>
#include "Student.h"

using namespace std;
typedef map<long int, Student> MapStd;

void main()
{
  MapStd StudentMap;
  MapStd::iterator itM;

  StudentMap[124252] = Student("Meier","ET");
  StudentMap[242521] = Student("Mueller","ET");
  StudentMap[254115] = Student("Schulze","ET");

  for (itM = StudentMap.begin(); itM != StudentMap.end(); itM++)
  {
    cout <<  itM->first << ' ';
    cout << (itM->second).getName() << endl;
  }
  return;
}
```

mit folgender Definition für Student:

```
#include <string>

using namespace std;

class Student
{
  public:
    string getName();
    Student();
    Student(char*, char*);
    virtual ~Student();
  private:
    string p_sFach;
```

```
    string p_sName;
};

Student::Student() {}
Student::Student(char* aName, char* aFach) :
    p_sName(aName), p_sFach(aFach) {}
Student::~Student() {}
string Student::getName() { return p_sName; }
```

Die Ausgabe ist:

```
124252 Meier
242521 Mueller
254115 Schulze
```

Der Vorteil der Nutzung einer Map statt eines Vektors bei einem `int`-Schlüssel ist, dass man hier nur die wirklich benutzten Elemente anlegt und nicht soviele wie der maximale Schlüsselwert.

Eine sehr häufige Anwendung der Maps ist die Zuordnung zwischen externen Textschlüsseln auf interne Werte, wie im folgenden Beispiel die Zuordnung der Wochentagsnamen zu dem entsprechenden Wochentag. Man beachte: Die Reihenfolge der Elemente in der sequentiellen Auflistung entspricht der Sortierfolge (<-Operator) des Schlüssels. Falls ein Schlüssel beim Zugriff nicht gefunden wird, wird 0 zurückgegeben.

Beispiel:

```
#include <iostream>
#include <string>
#include <map>

using namespace std;
typedef map<string, int> MapStrInt;

void main()
{
  MapStrInt Wochentage;
  MapStrInt::iterator itM;
  string sTag;
  Wochentage["Montag"] = 1;
  Wochentage["Dienstag"] = 2;
  Wochentage["Mittwoch"] = 3;
  Wochentage["Donnerstag"] = 4;
  Wochentage["Freitag"] = 5;
  Wochentage["Samstag"] = 6;
  Wochentage["Sonntag"] = 7;

  for (itM = Wochentage.begin(); itM != Wochentage.end(); itM++)
    cout <<  itM->first  << ' ' <<  itM->second << endl;

  do {
    cout << endl << "Gib Wochentag: ";
    cin >> sTag;
    cout << endl << sTag;
    cout << " ist der " << Wochentage[sTag];
    cout << ". Wochentag.";
```

```
  }
  while(sTag != "");

  return;
}
```

Die Ausgabe dieses Programms lautet dann:

```
Dienstag 2
Donnerstag 4
Freitag 5
Mittwoch 3
Montag 1
Samstag 6
Sonntag 7

Gib Wochentag: (Eingabe: Montag)

Montag ist der 1. Wochentag.
Gib Wochentag: (Eingabe: Dienstag)

Dienstag ist der 2. Wochentag.
Gib Wochentag: (Eingabe: Freitag)

Freitag ist der 5. Wochentag.
Gib Wochentag: (Eingabe: Ende)

Ende ist der 0. Wochentag.
```

Die Funktionen `erase()` und `insert()` funktionieren analog zu den Listen. Um einen Iterator auf ein bestimmtes Element zu erhalten, existiert die Memberfunktion

```
find([schlüssel])
```

die einen entsprechenden Iterator liefert.

```
Beispiel:
#include <iostream>
#include <string>
#include <map>

using namespace std;
typedef map<string, int> MapStrInt;

void main()
{
  MapStrInt Wochentage;
  MapStrInt::iterator itM;
  string sTag;

  Wochentage["Montag"] = 1;
  Wochentage["Dienstag"] = 2;
  Wochentage["Mittwoch"] = 3;
```

```
  Wochentage["Donnerstag"] = 4;
  Wochentage["Freitag"] = 5;
  Wochentage["Samstag"] = 6;
  Wochentage["Sonntag"] = 7;

  for (itM = Wochentage.begin(); itM != Wochentage.end(); itM++)
    cout <<  itM->first  << ' ' <<  itM->second << endl;

  itM = Wochentage.find("Donnerstag");
  Wochentage.erase(itM);
  itM = Wochentage.find("Montag");
  Wochentage.erase(itM,Wochentage.end());

  cout << endl;
  for (itM = Wochentage.begin(); itM != Wochentage.end(); itM++)
    cout <<  itM->first  << ' ' <<  itM->second << endl;

  return;
}
```

Ausgabe:

```
Dienstag 2
Donnerstag 4
Freitag 5
Mittwoch 3
Montag 1
Samstag 6
Sonntag 7

Dienstag 2
Freitag 5
Mittwoch 3
```

Vollständigkeitshalber ist in Abbildung 3.4 noch eine Übersicht aller Containerschnittstellen dargestellt.

3.4.2 Iteratoren

Iteratoren sind eine Verallgemeinerung der C++-Pointer, die es gestatten, die Elemente eines Containers (oder C-Arrays, C++-iostreams, ...) zu durchlaufen. Ein Iterator hat zu jedem Zeitpunkt eine bestimmte Position innerhalb eines Containers, die er so lange beibehält, bis er durch einen neuen Befehlsaufruf wieder versetzt wird.

Die STL definiert fünf Kategorien von Iteratoren, die hierarchisch angeordnet sind:

Input	kann in einem Schritt ein Element in Vorwärtsrichtung lesen
Output	kann in einem Schritt ein Element in Vorwärtsrichtung schreiben
Forward	Kombination von Input- und Output-Iterator
Bidirectional	wie Forward, kann sich aber auch rückwärts bewegen
Random-Access	wie Bidirectional, kann jedoch auch springen

Die grundlegenden Operationen eines Iterators sind:

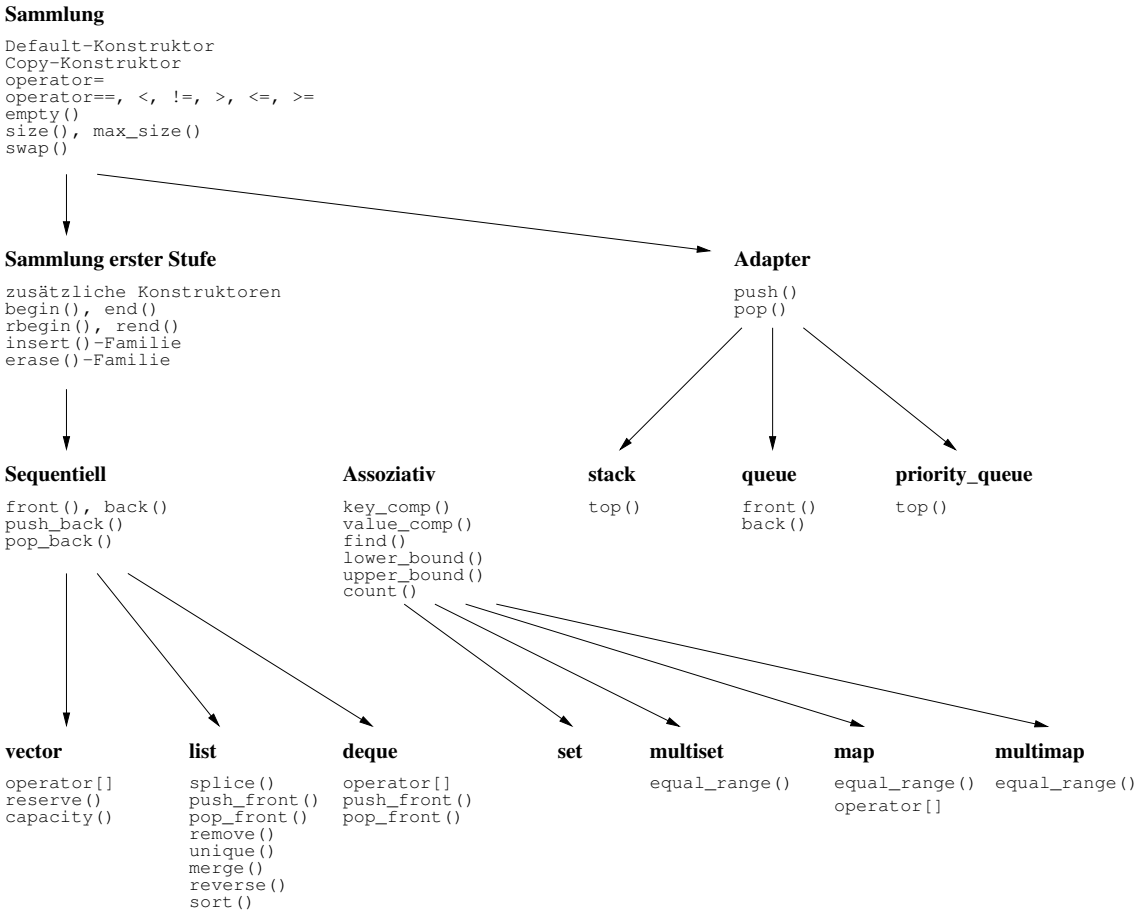


Abbildung 3.4: Hierarchie der Containerschnittstellen

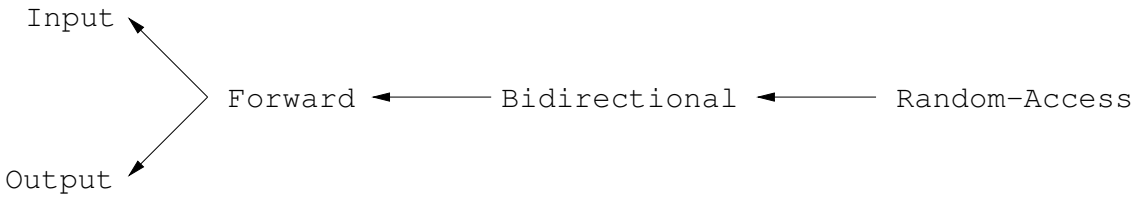


Abbildung 3.5: Hierarchie der Iteratoren

- die Dereferenzierung (Operatoren `*` und `->`), die ein Element des richtigen Typs liefert (Typ der Elemente des Containers)
- die pre- und post-Inkrement-Operatoren (`++` bzw. `--`)
- der Test auf Gleichheit/Ungleichheit (`==` bzw. `!=`)
- nur die zusätzlichen Operatoren zu definieren, die für die jeweilige Containerklasse effizient ausführbar sind (z.B. kein Random-Access-Zugriff für Listen).

Bei Random-Access-Iteratoren sind zusätzlich

- der Index-Operator `[]`,

Methoden	Funktionalität
`iterator begin()`	Liefert Iterator auf das 1. Element
`iterator end()`	Liefert past-the-end. Dereferenzierung dieses Iteratorwertes ist nicht möglich.

Tabelle 3.6: Einige Funktionen, die Iteratoren liefern

- die Addition/Subtraktion `+, +=, -, -=` und
- die Vergleichsoperatoren (`<, >, <=, >=`) definiert.

Im Rahmen dieses Skriptes können nur elementare Eigenschaften zur Nutzung der Iteratoren beschrieben werden. Zur Vertiefung, insbesondere für die genaue Unterscheidung verschiedener Iteratortypen (const-Iterator, Reverse-Iterator) und zur Konstruktion eigener Iteratoren, sei auf die Literatur verwiesen.

Iteratoren erhält man als Ergebnis einer entsprechenden Funktion (wie `begin()` oder `end()`) oder explizit über eine Deklaration wie:

```
vector<int>::iterator itV1;
```

Möchten Sie mittels eines Iterators innerhalb einer Konstant-Elementfunktion auf einen Container zugreifen, muss auch der Iterator als konstant deklariert werden, etwa:

```
list<int>::const_iterator citL1;
```

Iteratoren, die hinter das letzte Element eines Containers zeigen, heißen past-the-end. Die wichtigsten Funktionen von Containern, die einen Iterator liefern, finden Sie in Tabelle 3.6.

Beispiel:

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

using namespace std;

void main()
{
  vector<int> v(3, 1);
  v.push_back(7);        // v: 1 1 1 7
  int sum = 0;
  vector<int>::iterator itV = v.begin();
  cout << "Summe(";
  while (itV != v.end())
  {
    cout << setw(2) << *itV;
    sum += *itV;
    itV++;
```

```
  }
  cout << " ) = " << sum << endl;
  return;
}
```

Die Ausgabe ist dann: `Summe( 1 1 1 7 ) = 10`

3.4.3 Algorithmen

Um einen Container der STL effektiv nutzen zu können, sollte dieser Standardoperationen wie das Abfragen der Größe, Kopieren, Iterieren, Sortieren und Suchen nach Elementen unterstützen. Dazu bietet die STL etwa 60 Algorithmen, welche die fundamentalen Anforderungen eines Anwenders erfüllen. Sie werden alle in `<algorithm>` deklariert.

Jeder Algorithmus wird durch eine Template-Funktion ausgedrückt. Der Zugriff auf Elemente einer Datenstruktur erfolgt über Iteratoren, sodass diese Algorithmen auf unterschiedliche Datenstrukturen (Container, C-Arrays, Sammlungen anderer Hersteller) angewendet werden können.

Nachfolgend werden einige Algorithmen vorgestellt. Die Anwendung dieser Algorithmen ist für die Lösung der Praktikumsaufgaben jedoch nicht vorgeschrieben. Dieses und das folgende Kapitel dienen nur zur vollständigen Beschreibung der STL.

```
template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last,
                  Function f)
```

Algorithmus `for_each` führt `f` mit jedem Element des Intervalls `[first, last)` aus und gibt den Eingabeparameter `f` zurück.

```
template<class InputIterator, class Predicate, class Size>
void count_if(InputIterator first, InputIterator last,
              Predicate pred, Size& n)
```

`count_if` zählt die Elemente des Intervalls `[first, last)`, für die `pred` true ergibt und addiert die Anzahl zu `n`. Beachten Sie, dass `n` nicht automatisch mit Null vorbelegt wird.

```
template<class OutputIterator, class Size, class T>
void fill_n(OutputIterator first, Size n, const T& value)
```

`fill_n` weist, beginnend an der Position `first`, `n` Elementen den Wert `value` zu. Der Iterator steht dann an der Stelle `first + n`.

```
template<class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last)
```

`sort` sortiert alle Element des Intervalls `[first, last)` in aufsteigender Reihenfolge. Zum Vergleich der Elemente wird der <-Operator verwendet. Es gibt eine zweite Version des Algorithmus bei der die gewünschte Vergleichsfunktion zusätzlich angegeben werden kann.

```
template<class T>
```

```
const T& max(const T& a, const T& b)
```

`max` gibt einen Zeiger auf den größeren der beiden Werte `a` und `b` zurück bzw. `a`, wenn die Werte gleich sind. Verglichen wird wieder mit dem <-Operator. Auch hier gibt es eine Version mit wählbarem Vergleichsoperator. Analog zu `max` gibt es auch einen `min`-Algorithmus.

3.4.4 Funktionen und Funktionsobjekte

Bei den Algorithmen fällt auf, dass einige als Parameter ein Prädikat bzw. eine Funktion verlangen. Es besteht somit die Möglichkeit, einem Algorithmus selbst geschriebenen Code zu übergeben bzw. die in der STL definierten Funktionsobjekte zu verwenden.

Soll eine Funktion, die für viele Elemente aufgerufen wird, zwischen den Aufrufen Daten aufheben und am Ende ein Resultat liefern, ist dafür eine Memberfunktion einer Klasse einer alleinstehenden Funktion vorzuziehen, da diese die Möglichkeit bietet, Daten zu verwalten. Dies soll folgendes Beispiel verdeutlichen.

```
template<class T>
class Summe
{
  public:
    Summe(T i = 0) : res(i) {}
    void operator()(T x) { res += x; }  // Aufruf-Operator
    T ergebnis() { return res; }
  private:
    T res;
};

void f(list<double>& ld)
{
  Summe<double> s;
  // s() fuer jedes Element von ld aufrufen
  for_each(ld.begin(), ld.end(), s);
}
```

Man sieht, dass `s` über den Aufruf-Operator `operator()` wie eine Funktion behandelt wird. Objekte mit einem solchen Operator nennt man funktionsartiges Objekt, Funktor oder einfach Funktionsobjekt.

Um das Erstellen von Funktionsobjekten zu erleichtern, werden in der STL zwei Basisklassen `unary_function` (1 Parameter) und `binary_function` (2 Parameter) definiert, von denen eigene Funktionsobjekte abgeleitet werden können.

Funktionsobjekte der STL werden in folgende Gruppen unterteilt:

- Prädikate
- arithmetische Funktionsobjekte
- Adapteroperator()

Prädikate sind solche Funktionsobjekte, die einen bool zurückliefert. Sie werden oft zum Auslösen von Aktionen oder zum Ordnen von Elementen verwendet. Beispielsweise könnte man hier `equal_to, less` oder `logical_not` aufführen.

Arithmetische Funktionsobjekte stellen die standardisierten arithmetischen Funktionen zur Verfügung, wie `plus, multiplies` oder `modulus`.

Adapter nennen sich solche Funktionsobjekte, denen noch eine Hilfsfunktion zugeordnet ist, mit der das Erstellen von Instanzen vereinfacht wird. Zum Negieren eines Prädikates gibt es z.B. das Funktionsobjekt `unary_negate` mit der Hilfsfunktion `not1()`.

Alle Funktionsobjekte sind in `<functional>` deklariert und meist als Template definiert. Eine ausführliche Auflistung aller Funktionsobjekte finden Sie wiederum in der vorgeschlagenen Literatur.

Abbildungsverzeichnis

Tabellenverzeichnis

Index

RWTH Aachen
Lehrstuhl für Betriebssysteme

N.N.
Kopernikusstr. 16
52056 Aachen
Germany

Tel.: +(49)-241-8027634
Fax: +(49)-241-80627634

WWW: http://lfbs.rwth-aachen.de

RWTH Aachen
Lehrstuhl für Allgemeine Elektrotechnik
und Datenverarbeitungssysteme

Univ.-Prof. Dr.-Ing. Tobias G. Noll
Schinkelstr. 2
52062 Aachen
Germany

+(49)-241-8097600
+(49)-241-8092282

http://eecs.rwth-aachen.de

E-Mail: PI2Betreuung@cip1.eecs.rwth-aachen.de