C++, Eine Einführung

Matthias Müller[1] und Stefan Schwarzer[2]

März 2004

Manuskript Version 12e

[1]Höchstleistungsrechenzentrum Stuttgart, Allmandring 30, 70550 Stuttgart, mueller@hlrs.de
[2]Eugenstr. 22, 72072 Tübingen,sts@ica1.uni-stuttgart.de

Vorwörter

Dieses Skript ist eine kurze, leider recht unvollständige Zusammenfassung des Kurses *Naturwissenschaftliche Programmierung in C++* der im Oktober 1998 am Institut für Computeranwendungen im Rahmen des Studium Generale stattgefunden hat. Der Kurs richtet sich an Studenten mit Programmiererfahrung, die sich in die Ideen der objektorientierten Programmierung am Beispiel C++ und die Anfangsgründe der Programmentwicklung in Arbeitsgruppen einarbeiten wollen. Es umfasst den Inhalt der Vorlesungen und der Vorträge einiger Kursteilnehmer, denen hier noch einmal ausdrücklich für ihr Engagement gedankt sei: *Erik Bietzek, Ulrich Eberhardinger, Olav Finkenwirth, Ansgar Hennecke, Reinmar Mück, Martin Sochor, Oliver Strauß und Martin Treiber.*

Matthias Müller
Stefan Schwarzer Stuttgart, den 8. Oktober 1998

Zur Neuauflage des Kurses im September 1999 erscheint auch eine neue Auflage des Skripts. Vieles wurde unverändert übernommen oder geringfügig ergänzt. Die Grafiken wurden überarbeitet und am Anfang findet sich eine kurze Einführung in die Gründe für objektorientierte Programmierung. Wir danken den Kursteilnehmern für ihre Ausdauer und ihre Ideen bei den Projekten und die Vorbereitung ihrer Vorträge, deren Inhalt teilweise in dieses Skript eingeflossen ist bzw. noch einfließen wird: *Gabor Durovska, Martin Galle, Radomir Pletikapa, Danny Steinkopf, und Bin Yang*

Matthias Müller
Stefan Schwarzer Stuttgart, den 19. September 1999

Aufgrund der regelmäßigen Nachfragen entschlossen wir uns, eine kleine Auflage dieses Skripts zu drucken. Für das Korrekturlesen bedanken wir uns bei *Katrin Bidmon.*

Matthias Müller
Stefan Schwarzer Stuttgart, den 23. Juni 2000

Mit der neuen Rolle des Kurses als Teiles des *Batchelor*-Studienganges *Computational Physics* an der Universität Stuttgart wurde das Skript leicht überarbeitet, konzeptionelle Fehler korrigiert und die Programmbeispiele zu einem großen Teil überprüft. S.S. bedankt sich bei seinem Arbeitgeber für das Verständnis und die Zeit, Lehre an der Universität Stuttgart durchführen zu können.

Matthias Müller
Stefan Schwarzer Tübingen, den 16. Februar 2003

Auch im Jahr 2004 fand der Kurs im Rahmen *Batchelor*-Studienganges *Computational Physics* an der Universität Stuttgart statt, im Vorfeld wurde das Skript deutlich erweitert. Wir bedanken uns bei Dr. Ralf Ebner vom Leibniz Rechenzentrum (LRZ). Die Verwendung des Skripts am LRZ hat uns ermutigt das Skript weiterzuentwicklen. Viele Fehler wurden aufgrund seiner Hinweise beseitigt. S.S. bedankt sich bei seinem Arbeitgeber für das Verständnis und die Zeit, Lehre an der Universität Stuttgart durchführen zu können.

Matthias Müller
Stefan Schwarzer Stuttgart und Tübingen, März 2004

Inhaltsverzeichnis

1 Allgemeines **7**
1.1 Gründe für objektorientierte Programmierung . . . 7
1.2 Eigenschaften objektorientierter Programmiersprachen . . . 8
1.3 Eigenschaften von C++ im Speziellen . . . 11

2 Grundlegende Sprachkonstrukte **13**
2.1 Ein kurzes Programmbeispiel . . . 13
2.2 Kommentare . . . 14
2.3 Datentypen, Variablen, Konstanten . . . 14
2.3.1 Grundlegende Datentypen und Deklaration . . . 14
2.3.2 Initialisierung . . . 16
2.3.3 Variablenattribute . . . 17
2.3.4 Zeiger, Referenzen und Felder . . . 19
2.3.5 `void` . . . 24
2.3.6 `typedef` . . . 24
2.4 Kontrollstrukturen . . . 25
2.4.1 if-Anweisung . . . 25
2.4.2 Ternärer ?:-Operator . . . 26
2.4.3 (do) while-Schleifen . . . 26
2.4.4 for-Schleifen . . . 26
2.4.5 break und continue . . . 28
2.4.6 goto-Anweisung, Sprungmarken . . . 28
2.4.7 switch-Anweisung . . . 29
2.5 Funktionen und Operatoren . . . 30
2.5.1 Deklaration . . . 30
2.5.2 Definition . . . 30
2.5.3 Argumentübergabe . . . 31
2.5.4 inline-Funktionen . . . 31
2.5.5 Defaultargumente . . . 32
2.5.6 Überladen von Funktionen . . . 33
2.5.7 Das main-(Unter)Programm . . . 34
2.5.8 Funktionen mit einer variablen Anzahl von Argumenten . . . 34
2.5.9 Operatoren . . . 35
2.6 Bezugsrahmen von Bezeichnern . . . 37
2.6.1 Namensräume . . . 37
2.6.2 Dateibezugsrahmen . . . 39
2.6.3 Lokale Bezugsrahmen . . . 41
2.6.4 Klassenbezugsrahmen . . . 41
2.7 Ein- und Ausgabe . . . 41
2.7.1 Allgemeines . . . 41
2.7.2 Wie funktioniert die Ein- oder Ausgabe in C++ . . . 42

2.7.3 Ein- und Ausgabe selbstdefinierter Typen . . . 43
2.7.4 Wichtige Elementfunktionen der iostreams . . . 43
2.7.5 Formatierung . . . 43
2.7.6 Spezielle Ein- und Ausgabefunktionen der `i/ostream` Klassen . . . 46
2.7.7 Schreiben und Lesen von Dateien . . . 46
2.8 Dynamische Speicherverwaltung . . . 48
2.8.1 Überladen des `operator new` . . . 49
2.9 Programmorganisation: Implementierung und Deklaration . . . 50
2.9.1 Deklarationsdateien . . . 50
2.9.2 Implementierungsdateien . . . 50
2.10 Übungen . . . 52
2.10.1 Das Hello-World-Programm . . . 52
2.10.2 Ein- und Ausgabe . . . 52

3 Klassen **53**
3.1 Deklaration von Klassen . . . 53
3.1.1 Datenelemente . . . 53
3.1.2 Elementfunktionen . . . 55
3.1.3 this - Zeiger . . . 56
3.1.4 Geburt und Tod von Objekten . . . 57
3.1.5 Defaultkonstruktor . . . 59
3.1.6 Kopieren und Zuweisung von Objekten . . . 60
3.1.7 Datenkapselung und Zugriffsprivilegien: `private, public` . . . 61
3.1.8 `friend`-Deklaration . . . 64
3.1.9 Temporäre Objekte . . . 65
3.1.10 Überladen von Operatoren . . . 66
3.1.11 Typumwandlung . . . 72
3.1.12 Hintergründe zum Überladen von Operatoren . . . 74
3.1.13 Fragen . . . 74
3.2 Vererbung . . . 75
3.2.1 Syntax . . . 76
3.2.2 Zugriffsprivilegien . . . 77
3.2.3 Abstrakte Basisklassen . . . 77
3.2.4 Eigenschaften spezieller Elementfunktionen . . . 80
3.2.5 Virtuelle Funktionen . . . 80
3.2.6 protected . . . 87
3.3 Mehrfachvererbung . . . 87
3.3.1 `virtual`-Vererbung . . . 88
3.4 Übungen . . . 90
3.4.1 Operatoren und Elementfunktionen . . . 90
3.4.2 Vererbung . . . 90
3.4.3 Klassenschnittstellen, spezielle Funktionen . . . 90
3.4.4 Virtuelle Funktionen (Fortgeschrittene) . . . 91

4 Templates **93**
4.1 Grundidee . . . 93
4.2 Template-Funktionen . . . 93
4.3 Template-Klassen . . . 95
4.3.1 Herkömmliche Programmierung . . . 95
4.3.2 Typunabhängige Programmierung . . . 97
4.3.3 Ein Stack mit dynamischer Speicherverwaltung . . . 99
4.4 Template Spezialisierung . . . 101
4.5 Template `friends` . . . 104

4.6 Traits . . . 105
4.7 Überladen von Template-Funktionen . . . 106
4.8 Template-Elementfunktionen . . . 109
4.9 Expression Templates . . . 109
4.9.1 Template Rekursion . . . 109
4.9.2 Metaprogramm-Kochrezept . . . 113
4.10 Template-Basisklassen . . . 115
4.11 Übungen . . . 118
4.11.1 Stack/-Stapelspeicher . . . 118
4.11.2 Templatefunktionen . . . 118
4.11.3 Matrix-/Vektorklasse . . . 119

5 Die C++ Standardbibliothek 121
5.1 Standard Template Library . . . 121
5.1.1 Zugriff auf Elemente beliebiger Container . . . 123
5.1.2 Eigenschaften der STL-Container-Klassen . . . 125
5.1.3 Container-Adapter-Klassen . . . 129
5.1.4 Algorithmen . . . 130
5.1.5 Iterator-Adaptoren . . . 137
5.1.6 Nachtrag zur Motivation . . . 139
5.2 Benutzung der Klasse `std::string` . . . 139
5.2.1 Benutzungbeispiel: Manipulation von Dateinamen . . . 140
5.3 `valarray` . . . 143
5.4 Anwendungsbeispiele für STL Klassen . . . 144
5.4.1 Zweidimensionale Felder . . . 144
5.5 Übungen . . . 148
5.5.1 Die Klasse `vector` . . . 148
5.5.2 Der `map`-Container . . . 148
5.5.3 Algorithmen . . . 148
5.5.4 Iterator-Gültigkeit . . . 148

6 Exceptions 151
6.1 Exceptions zur Fehlerbehandlung . . . 151
6.1.1 Exceptions in Ein- und Ausgabe . . . 154
6.1.2 Exception des `operator new()` . . . 155
6.2 Exceptions in Funktionsdeklarationen . . . 155
6.3 Vermeiden von Resource Leaks mit Auto Pointern . . . 156
6.3.1 Kopieren von `auto_ptr` . . . 157
6.4 Übungen . . . 159
6.4.1 Verwendung von Exceptions . . . 159
6.4.2 Exception des operator new . . . 159
6.4.3 Exception oder spezieller Fehlerwert? . . . 159
6.4.4 Exceptions bei der Ein- und Ausgabe . . . 159
6.4.5 Ein intelligenter Zeiger . . . 159

7 Programmentwicklung 161
7.1 Versionsverwaltung mit CVS . . . 161
7.1.1 Einrichtung eines CVS Zentralverzeichnisses . . . 162
7.1.2 Benutzung . . . 162
7.1.3 Beschreibung der Kommandos . . . 162
7.2 Coding Standards . . . 167
7.2.1 Dateinamen . . . 167
7.2.2 Variablennamen . . . 168

7.2.3 Variablendeklaration . . . 168
7.2.4 Performance, const, Zeiger oder Referenz . . . 168
7.2.5 #include Direktiven . . . 169
7.2.6 Präprozessoranweisungen . . . 169
7.2.7 Kommentare . . . 170
7.2.8 Formatierung . . . 170
7.2.9 Editorunterstützung . . . 171
7.3 Objektorientiertes Design . . . 171
7.3.1 Beziehung zwischen Objekten . . . 171
7.3.2 Klassenschnittstellen . . . 172
7.3.3 Allgemein oder speziell? . . . 172

8 Systemspezifische Fragen und Bibliotheken 173
8.1 Vorüberlegungen . . . 173
8.2 Threads . . . 174
8.2.1 Grundprinzip . . . 174
8.2.2 Synchronisierung . . . 177

9 C++ und die weite Welt 181
9.1 Entwicklungsziele, Eigenschaften und Geschichte von C++ . . . 181
9.2 Java — zum Vergleich mit C++ . . . 182

A Beispiele 191
A.1 Einführendes Beispiel . . . 191
A.2 Programmierstile . . . 191
A.2.1 Prozedurale/Modulare Programmierung . . . 192
A.2.2 Objektorientiertes C++-Programm zur Erzeugung von Zufallszahlen 193
A.2.3 Objektorientiertes C-Programm zur Erzeugung von Zufallszahlen . . 194
A.2.4 Vererbung . . . 195
A.2.5 Typunabhängige (Generische) Programmierung: Feldgrenzentests . . 195
A.3 Eine Datumsklasse . . . 196
A.3.1 Schnittstellendefinition in `Date.h` . . . 196
A.3.2 Implementierung `Date.cc` . . . 197
A.3.3 Hauptprogramm `main.cc` . . . 198

Index 201

Kapitel 1

Allgemeines

1.1 Gründe für objektorientierte Programmierung

Die objektorientierte Programmierung entstand in den Labors von XEROX in dem Wunsch, die Bedienung von Computern zu vereinfachen. Dort kondensierten sich die Ideen der Objektorientierung in der Sprache SMALLTALK. Schnell zeigte sich, dass das neue Programmierparadigma gegenüber herkömmlichen Ansätzen große Vorzüge aufwies, die sich im Wesentlichen aus einer sehr weitgehenden Unterstützung der Lokalisierung von Datenzugriffen ergeben.

Was damit gemeint ist, sieht man in den Abbildungen 1.1 und 1.2, in denen die typische Struktur eines prozedural geschriebenen Programmes mit Daten und Unterprogrammen und die eines modular aufgebauten Programmes mit Modulen skizziert sind. Im ersten Fall ist der algorithmische Teil durch Aufteilung in einzelne Unterprogramme zwar recht übersichtlich, aber diese Unterprogramme greifen auf einen (globalen) Datenpool zu, der im Prinzip durch jedes der Unterprogramme modifiziert werden kann. In derartig geschriebenen Programmen existiert keine eindeutige Zuordnung von Unterprogrammen und Daten. Die Verwendung lokaler Variablen in Unterprogrammen ist in der Regel auch in älteren Programmiersprachen vorgesehen, wenn auch nicht immer (wie in frühen Versionen von BASIC) realisiert worden.

Die Einführung von Modulen (Programmteilen, die aus Daten und Unterprogrammen bestehen) erzwingt scheinbar eine solche Zuordnung, denn jedes Modul kann globale (äußere) oder Daten aus einem anderen Modul nur dann ändern, wenn es dies explizit ankündigt. Ansonsten wird das Problem der globalen Daten allerdings nur um eine Ebene verlagert, denn innerhalb jedes Moduls gilt das im vorigen Abschnitt Gesagte weiter.

Warum diese Formen der Programmorganisation problematisch ist, wird klar, wenn man sich überlegt, wie man bei Programmänderungen vorgehen muss. Über die gemeinsamen Datenbereiche besteht nämlich eine Wechselwirkung zwischen zwei Unterprogrammen, die bewirken kann, dass bei Änderung eines Unterprogramms ein anderes seine Arbeit nicht mehr korrekt verrichtet. Ein einfaches Beispiel ist das Auslassen der Initialisierung einer Variable in einem Unterprogramm und deren spätere Benutzung in einem anderen. Zumindest im Prinzip muss also für Änderungen in einem Programmteil der Programmtext des gesamten Programms auf Abhängigkeiten geprüft werden. Es ist daher nicht schwierig vorauszusagen, dass es irgendwann unmöglich wird, ein so aufgebautes Programm zu pflegen. Der Kern des Problems ist, dass *a priori* keine Aussagen darüber gemacht werden können, von welcher Stelle des Programmes aus Änderungen an den Daten erfolgen können.

In der objektorientierten Programmierung hat man daher versucht, die Daten und den Programmcode (noch) enger zu verbinden. Beide werden in Klassen als Datenelemente und Methoden vereint. Die resultierende Programmtopologie sieht man in Abbildung 1.3. Im

Idealfall erfolgt der Zugriff auf die Daten nur über die als Methoden spezifizierten Unterprogrammschnittstellen. Eine Änderung der Implementierung der Methode oder der verwendeten Datenelemente hat damit nur lokale Auswirkungen, die leicht aufzufinden sind. Um diese Kapselung der Daten zu ermöglichen bzw. zu erzwingen, besitzen alle objektorientierten Sprachen Konstruktionen, mit denen der Zugriff auf Klassendatenelemente geregelt und auf Wunsch des Klassenprogrammierers verhindert werden kann.

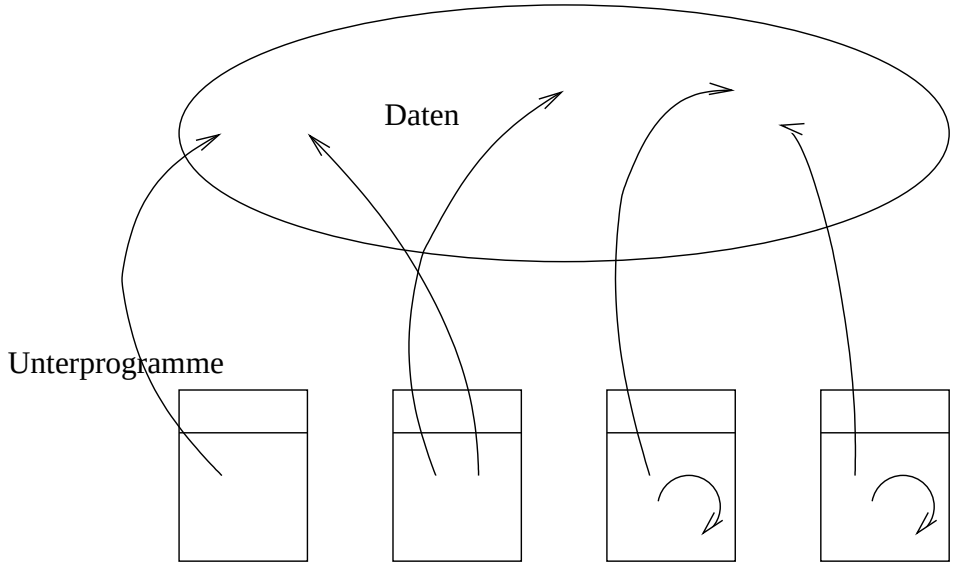


Abbildung 1.1: Topologie einer klassischen prozeduralen Programmiersprache der ersten Generation. Alle Unterprogramme haben Zugriff auf die globalen Daten und auf lokale Variablen.

Darüber hinaus existieren in objektorientierten Sprachen Ansätze, um eine konsequente, stufenweise Weiterentwicklung des vorhandenen Programmcodes zu unterstützen, ohne dabei die Lauffähigkeit bereits bestehender Programme zu beeinträchtigen. Dazu kann man zu bestehenden Klassen Datenelemente und Methoden hinzufügen oder alte Methoden überschreiben. Tatsächlich kann dieser neue Code dann sogar von geeignet geschriebenen "alten" Programmen genutzt werden, ohne dass dazu eine Neuübersetzung der "alten" Programmteile nötig wäre. Die zu Grunde liegenden Ideen belegt man häufig mit den Schlagwörtern Vererbung (Ergänzung bestehender Klassen) und Polymorphismus (Möglichkeit, neue Klassen in altem Code zu verwenden), die uns später wieder begegnen werden.

1.2 Eigenschaften objektorientierter Programmiersprachen

Im Folgenden stellen wir für einige der wichtigsten objektorientierten Sprachen zusammen, welche Programmierkonzepte von Sprachkonstruktionen unterstützt werden. Wir richten uns dabei nach Grady Booch [3], der unter anderem die in Tabelle 1.1 aufgeführten Kriterien verwendet, um eine Sprache bezüglich ihrer Objektorientierung zu klassifizieren. [1]

[1] Das Fehlen bestimmter Eigenschaften gibt häufig zu "Sprach"kriegen Anlass, in denen Programmierer "ihre" Sprache als "besser" herausstellen wollen. Hierzu sei nur erwähnt, dass sich die Konzepte der Objektorientierung unter Inkaufnahme syntaktischer Hässlichkeiten in *jeder* hinreichend vollständigen Computersprache (BASIC, C, etc.) verwirklichen lassen.

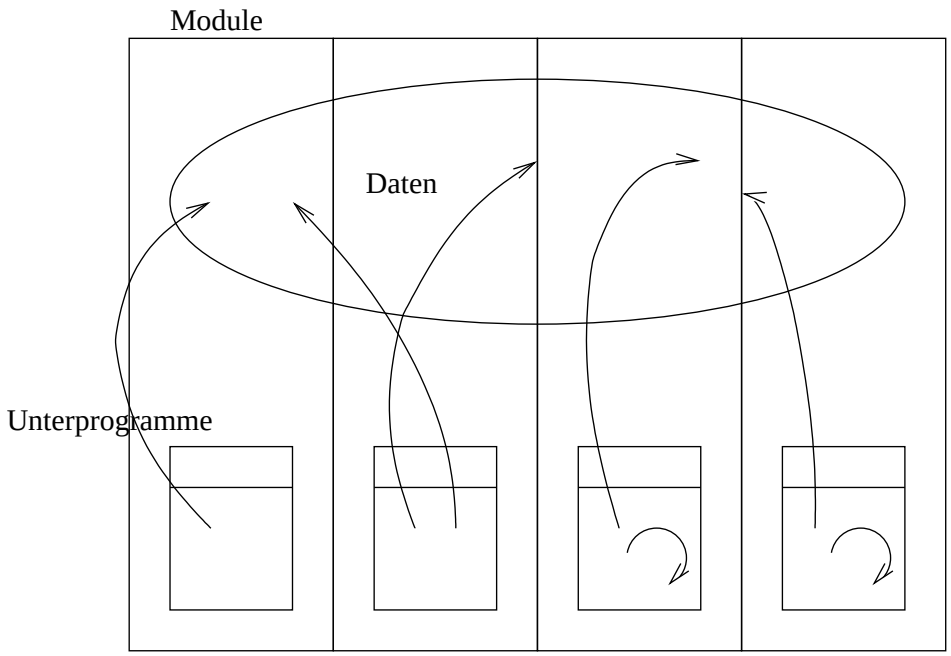


Abbildung 1.2: Topologie einer klassischen Programmiersprache mit Modulen. Globale Zugriffe auf Daten sind zwar weniger erforderlich, aber noch immer wichtiger Bestandteil der Programmstruktur.

Gebiet	Eigenschaft	Smalltalk	Eiffel	Ada	C++	Java
Abstraktion	Instanz-Variablen	ja	ja	ja	ja	ja
	Instanz-Methoden	ja	ja	ja	ja	ja
	Klassen-Variablen	ja	nein	nein	ja	ja
	Klassen-Methoden	ja	nein	nein	ja	ja
Kapselung	von Variablen	priv.	priv.	pub./priv.	alle	alle
	von Methoden	pub.	pub./priv.	pub./priv.	alle	alle
Modularität		nein	ja	ja	ja	ja
Hierarchie	Vererbung	einfach	mehrfach	Ada9x	mehrfach	einfach
	Generische Prog.	nein	ja	ja	ja	nein
	Metaklassen	ja	nein	nein	nein	?
Typen	typsicher	nein	ja	ja	ja	ja
	Polymorphismus	einfach	ja	Ada9x	einfach	ja
Parallelität	Multitasking	indirekt	nein	ja	indirekt	ja
Persistenz	Persistente Objekte	nein	nein	nein	nein	ja

Tabelle 1.1: Sprachelemente verschiedener objektorientierter Programmiersprachen.

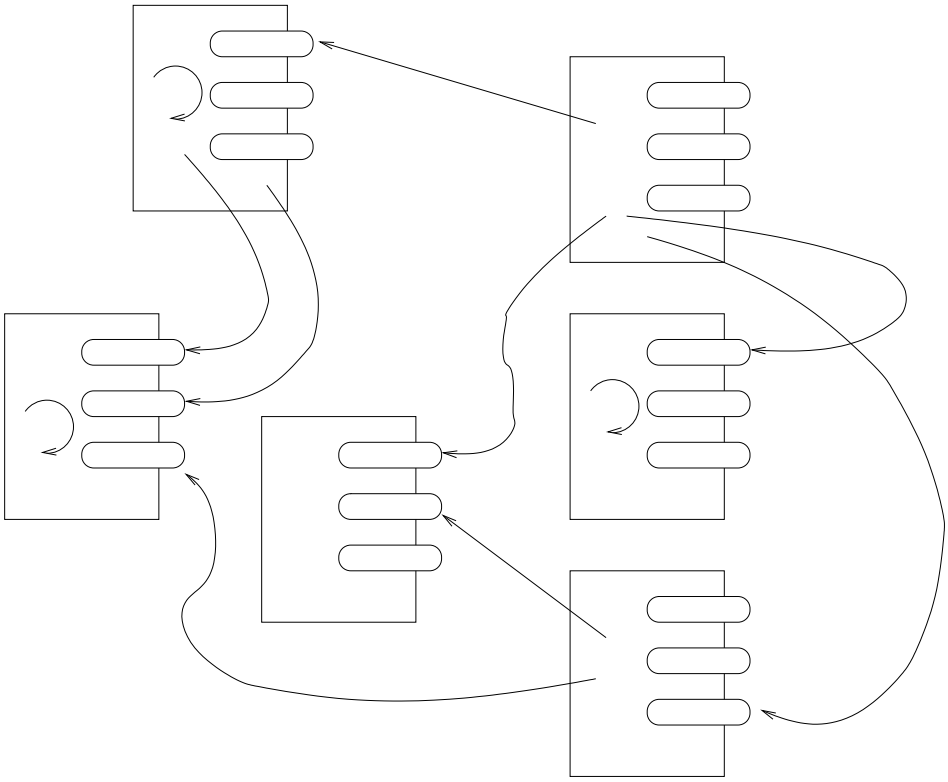


Abbildung 1.3: Topologie einer objektorientierten Programmiersprache. Die Daten sind mit den Programmcode vereint. Der Zugriff erfolgt im Idealfall nur über festgelegte Schnittstellen.

1.3 Eigenschaften von C++ im Speziellen

Wir werden im Folgenden auf C++ eingehen, das viele dieser Eigenschaften aufweist. C++ zeichnet aus, dass es eine Weiterentwicklung von ANSI C ist und dass "alter" C-Code mit geringem Aufwand in C++-Programme integriert werden kann. Somit ist eine große Softwarebasis vorhanden, auf der sich eigene Programme aufbauen lassen.

C++ ist eine sehr komplexe Sprache. Schon der nackte Text des Sprachstandards einschließlich der Schnittstellen der Standardbibliothek umfasst 750 Textseiten. Dafür eröffnet C++ allerdings auch viele Freiheiten und unterstützt als so genannte *multi paradigm* Sprache sowohl prozedurale und modulare als auch objektorientierte und generische (typunabhängige) Programmierung, so dass zusammen mit den Eigenschaften von C für (fast) jedes Problem dass nötige Werkzeug bereitgestellt wird:

1. prozedurale Programmierung
   - Blockstrukturierung (beliebig "lokale" Variablen)
   - Funktionen mit Prototypen wie in ANSI C
2. modulare Programmierung
   - Namensräume (`namespace`) zur Einschränkung des Sichtbarkeitsbereichs von Namen.
   - Überladen von Funktionen nach Argumenttypen
   - "`static`" wie in C.
3. objektorientierte Programmierung
   - Klassen
   - einfache und mehrfache Vererbung
   - Polymorphismus
4. generische Programmierung
   - Typen und Konstanten als Parameter (`template`) für Klassen und Funktionen
   - Standard-Template Bibliothek als Bestandteil des Sprachstandards

Die oben aufgelisteten, den einzelnen "Paradigmen" zu Grunde liegenden Konzepte werden wir in den folgenden Kapiteln dieses Skriptes genauer kennen lernen.

Kapitel 2

Grundlegende Sprachkonstrukte

Nach einem kurzen Programmbeispiel werden in diesem Kapitel die grundlegenden Sprachkonstrukte vorgestellt, die für die Programmierung erforderlich sind. Da C++ weitgehend abwärtskompatibel zu C ist und diese Sprache nur erweitert, sind die meisten C-Programmierern bereits bekannt. Aber schon in diesem Kapitel finden sich neue Sprachelemente wie das Überladen von Funktionen und Operatoren, eine andere Form der dynamischen Speicherverwaltung und eine komplett neu gestaltete Form der Ein- und Ausgabe.

2.1 Ein kurzes Programmbeispiel

Natürlich darf als erstes kleines Programmbeispiel in C++ ein Hello-World-Programm nicht fehlen:

```
// helloworld.cc:

#include <iostream>

int main(){
   std::cout << "Hello World!\n";
   return 0;
}
```

Hier ist zum Vergleich das entsprechende Programm in C abgedruckt:

```
// helloworld.c:

#include <stdio.h>

int main(){
   printf("Hello World!\n");
   return 0;
}
```

In beiden Beispielen erfolgt die Einbindung benutzter Funktionen durch die Angabe einer Deklarationsdatei, die mit einer `#include` Anweisung durch den C-Präprozessor gefunden und eingebunden wird. Man beachte, dass die Deklarationsdateien der C++-Standardbibliothek keine `.h`-Endungen aufweisen. Der wesentliche sichtbare Unterschied ist die in C++ überarbeite Behandlung der Ein- und Ausgabe. In C++ wird statt eines Funktionsaufrufes der `operator<<` genutzt, und die Ausgabe auf das Terminal wird durch Verwendung eines

Objektes mit Namen `std::cout` bewirkt. Wir werden Ein- und Ausgabe ausführlich in Abschnitt 2.7 behandeln.

Die Erzeugung von Programmcode für die obigen Beispiele ist systemabhängig und abhängig von den Programmierwerkzeugen auf der jeweiligen Plattform. Auf Rechnern unter UNIX-ähnlichen Betriebssystemen mit der GNU-Programmierumgebung wird man nach Erstellung des Programmcodes mit einem Texteditor das Programm durch

```
g++ helloworld.cc -o helloworld
```

übersetzen und mit

```
./helloworld
```

ausführen können. Die `-o` Option des Compilers legt einen Namen für das ausführbare Programm fest. Lässt man die Option fort, so wird der Name `a.out` verwendet. Wir verweisen auf die verfügbare Dokumentation des Compilers für weitere Optionen. Auf Grund der Kompatibilität von C++ mit C lässt sich auch das `helloworld.c`-Programm mit einem C++-Compiler übersetzen.

2.2 Kommentare

Kommentare werden durch `//` eingeleitet und erstrecken sich bis zum Ende der Zeile. Zeilenumbrüche werden ansonsten wie Leerzeichen behandelt und haben keine syntaktische Relevanz. Die C-Kommentarsyntax `/* Kommentar */` gilt auch weiterhin in C++. `/* */` Kommentare lassen sich nicht schachteln und sind daher zum "Auskommentieren" größerer Programmbereiche ungeeignet. Falls nämlich im auskommentierten Programmbereich bereits ein solcher C-Kommentar vorhanden gewesen sein sollte, so wird die erste schließende `*/`-Kombination den gesamten Kommentarbereich beenden und so dazu führen, dass ein Teil des Programmes wieder eingeblendet wird. Die wahrscheinlichste Konsequenz ist ein Syntax-Fehler, wenn der Compiler die zweite schließende Kombination `*/` findet. Allerdings erlaubt die `/* */`-Syntax, einen Kommentar an beliebiger Stelle beenden zu können, was in manchem Kontext hilfreich ist (wie etwa im Beispiel im nächsten Absatz).

Große Programmbereiche werden am einfachsten mit Präprozessordirektiven ausgeblendet, kleinere mit der `//`-Kommentarsyntax. Auch sollte man C++-Kommentare in Zeilen vermeiden, die Präprozessordirektiven enthalten, da der Präprozessor aus C-Zeiten möglicherweise nur C-Kommentarsyntax korrekt behandelt.

```
#include <iostream>  /* I/O Klassen aus der Standardbibliothek */

#if 0                /* das ganze Hauptprogramm wird auskommentiert */
int main(){          // main ist Schluesselwort und Programmstartpunkt
   for ( int i=0 /* Schleifenvariable */; i< 10 ; i++ ) { // for-Schleife
       ; // eine Erklaerung fuer das Nichtstun
   } // Ende der for-Schleife
}
#endif               /* Ende des Auskommentierens... */
```

2.3 Datentypen, Variablen, Konstanten

2.3.1 Grundlegende Datentypen und Deklaration

Die grundlegenden, eingebauten Datentypen von C++ sind `bool, char, int, float, double`. Daneben gibt es die Typen `wchar_t`, der in C++ durch die Bibliothek bereitgestellt wird und 2-byte-Zeichen repräsentiert sowie `void`, der das Fehlen des Rückgabewertes einer

Funktion kennzeichnet oder zur Deklaration einen generisches Zeigers verwendet werden kann, der groß genug ist, um auf beliebige Objekte oder Funktionen verweisen zu können.

Der Typ `bool`, der in C++ neu eingeführt wurde, findet wie logische Typen in anderen Sprachen Anwendung für die Speicherung zweiwertiger logischer Variablen, wie der Resultate von Vergleichsoperationen (2.5.9) oder der Argumente von Kontrollstrukturen. Er kann die Werte `true` oder `false` besitzen. Weiterhin ist er frei in den `int` Datentyp umwandelbar, wobei die Konstante `true` einer 1 und `false` einer 0 entspricht. Diese Umwandlung stellt die Aufwärtskompatibilität zu alten Programmen her, die `int`'s verwendeten, um Wahrheitswerte zu speichern. Dort galt, dass `0` "false" bedeutete und alle anderen Werte "true." Einige einfache Deklarationen sind:

```
bool    test;
test  = 0 < 5;  // Vergleich ist 'true', daher test nach dieser
                //   Zuweisung 'true'
int   i(test);  // i wird mit ()-Initialisierung auf 1 gesetzt
const bool mytrue(true); // Initialisierung mit einer bool-Konstante
```

Der Typ `int` ist der Datentyp, der auf der jeweiligen Maschine vom Compilerschreiber als der für die Verarbeitung integraler Daten "geeignetster" angesehen wird. Über seine Größe ist im Standard nichts festgelegt. Mit Hilfe des `sizeof` Operators kann man seine Größe in Einheiten der Größe eines `char` erfragen, der gleichzeitig der kleinste Datentyp ist. Ein `char` wurde vom Compilerschreiber als der für Einzel(schrift)zeichen geeignete Typ gewählt. Verschiedene Rechner unterstützen noch weitere ganzzahlige Datentypen, z.B. neben 32-bit `int` auch 16- oder 64-bit große Variablen. Diese lassen sich ggf. mittels `short` und `long` oder gar `long long` Deklarationen ansprechen. Nach diesen Schlüsselworten darf `int` fehlen.

`float` und `double` sind Gleitkommatypen einfacher und doppelter Genauigkeit, die zu numerischen Zwecken geeignet sind. `long` kann zur Modifikation von `double` verwendet werden, um vierfache Genauigkeit zu erhalten (falls diese unterstützt sein sollte). Die exakte Gleitkommarepräsentation ist abhängig von der Hardware-Unterstützung.

Der Sprachstandard garantiert für die Größe der Datentypen nur die folgenden Eigenschaften:

`sizeof(bool)` = `sizeof(char)`
≤ `sizeof(short)` ≤ `sizeof(int)` ≤ `sizeof(long)` ≤ `sizeof(long long)`,
sowie für die Gleitkommatypen
`sizeof(float)` ≤ `sizeof(double)` ≤ `sizeof(long double)`
Der Gebrauch der oben erwähnten Typen `long long` und `long double` ist zur Zeit noch nicht standardisiert, werden aber von vielen Systemen und Compilern bereits bereitgestellt.

Die folgenden Beispiele deklarieren verschiedene Datentypen.

```
char  c = '\n';
int  i_var;        // integer variable with name i_var
long int il_var = 1L;   // L denotes long constant
long unsigned int ilu_var = 1LU;  // L denotes long constant,
                                  // U denotes unsigned
long long ill_var; // long long integer variable
short is_var;      // short integer variable
double d_var = 34.2;
```

Variablendeklarationen sind in C++ nicht auf den Anfang eines Blockes beschränkt (wie in C), sondern können überall dort erfolgen, wo die Variable erstmals an einer Stelle auftaucht, wo auf sie zugewiesen werden kann. Die Nutzung dieser Option erhöht die Lesbarkeit des Programmes, da die Verwendung eines Variablennamens in der Regel auf wenige Programmzeilen beschränkt ist und nicht mehr künstlich von der Deklaration und ggf. Initialisierung getrennt werden muss.

Besonders praktisch ist die Deklaration innerhalb einer `for(;;)` Schleife, die den Existenzbereich der Schleifenvariablen auf den Schleifenkörper beschränkt.

```
double  a;
std::cin >> a;       // Einlesen vom Terminal
int   i_trunc = a;   // Nachkommateil entfernen

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

2.3.2 Initialisierung

Die Initialisierung/Konstruktion einer Variablen kann in der Form *type variable = expression* oder äquivalent mittels runder Klammern durch *type variable(expression)* erfolgen. Das Gleichheitszeichen bedeutet an dieser Stelle Initialisierung und *nicht* Zuweisung, da die Variable gerade erst ins Leben gerufen wird.[1]

Wird keine Initialisierung explizit angefordert (wie in der ersten Zeile des unten stehenden Beispiels), so wird die Variable defaultinitialisiert: Im Falle automatischer Variablen eingebauter Typen (s.u.) ist der Inhalt unbestimmt, im Falle automatischer Klassenvariablen (nicht reine C-`structs`) wird deren Defaultkonstruktor aufgerufen, Feldelemente werden grundsätzlich defaultinitialisiert, statische Variablen null-initialisiert. Defaultinitialisierung benannter Variablen im regulären Programmkontext erfolgt bei Weglassen aller `()` am Variablennamen und bei temporären Variablen in Ausdrücken oder in Funktionsargumenten durch Anhängen eines leeren Klammerpaares an den Klassennamen.

```
int a;     // default- (d.h. nicht) initialiserter int
int b(3);  // int, mit 3 initialisiert
int c = 3; //  aequivalent
int d();   // VORSICHT: Funktionsdeklaration einer Funktion
           //  ohne Argumente, die int zurueckliefert
f(int());  // Aufruf einer Funktion mit einem default
           //  initialisierten, temporaeren int
```

Zusammenfassung:
(i) Deklarationen können an fast beliebiger Stelle im Programm stehen
(ii) `sizeof(type)` kann eingesetzt werden, wenn die Größe eines Typs im Programm ermittelt werden muss. Ein solches Wissen ist häufig für die portable Programmierung in verschiedenen Programmierumgebungen nützlich.
(iii) Initialisierung erfolgt in `()` oder `=` Form; es findet niemals Zuweisung, sondern immer Initialisierung statt
(iv) "keine" Initialisierung ist im Regelfall Default-Initialisierung und liefert bei eingebauten Typen unbestimmte Variableninhalte.
(v) Warnung:

```
int   a();
```

deklariert keine default-initialisierte Variable vom Typ `int`, sondern eine Funktion, die `int` zurückgibt und keine Argumente erwartet.

[1] Die Klammersyntax wird direkt in den Aufruf eines entsprechenden Konstruktors übersetzt (siehe 3), bei Verwendung des Gleichheitszeichens wird der Compiler u. U. zwei Konstruktoren benutzen: einen, um ggf. den Typ der rechten Seiten in den Typ der linken zu wandeln und einen weiteren, um eine Kopie der Variablen anzulegen. Der Compiler darf die Benutzung des Kopierkonstruktors eliminieren, wenn dadurch keine sichtbaren Konsequenzen für die Programmausführung entstehen.

2.3.3 Variablenattribute

Neben den Attributen `short`, `long` und `long long` können ganzzahlige Variablen noch durch `signed` und `unsigned` modifiziert werden, um dann entweder ganze ($\in Z$) oder natürliche Zahlen ($\in N_o$) zu repräsentieren. Wird ein solches Attribut verwendet, so kann ein eventuelles `int` der Typdeklaration entfallen.

Lebensdauer der Variablen, Lage der Variablen im Systemspeicher

`auto` Falls keine weitere Angabe erfolgt, sind Variablen `auto`matic. Dies bedeutet, dass sie angelegt werden, sobald das Programm über die Zeile läuft, in der sie deklariert sind. Sobald das Programm den Block verlässt, in dem sie deklariert sind, wird der zugehörige Speicher wieder freigegeben. Der Block ist im Allgemeinen das innerste die Variable umschließende Paar geschweifter Klammern, oder wie im Fall der in den ()-Bereichen von Anweisungen wie `for(;;)` deklarierten Variablen, das Ende dieser Anweisung. In diesem Fall wird der Speicherplatz freigegeben, sobald diese Anweisung vollständig abgearbeitet ist. Eine Variable dieser Art wird vom Compiler entweder im Stack-Bereich des Programmes oder aber auch in einem Prozessorregister verwaltet, falls das Programm nicht explizit oder implizit eine Adresse dieser Variable verwendet.

`static` Das Schlüsselwort `static` bewirkt, dass eine Variable während der gesamten Programmlebensdauer existiert. Diese Variablen werden vom Compiler vor Programmbeginn meist auf dem Heap-Speicher angelegt und auf garantierte Anfangswerte initialisiert. Eine `static` Variable innerhalb einer Funktion wird beim ersten Aufruf der Funktion nullinitialisiert und behält zwischen Funktionsaufrufen ihren jeweiligen Wert bei. Eine automatische Variable hingegen wird jedes Mal neu angelegt und ggf. initialisiert. Die folgende Funktion benutzt die Eigenschaften lokaler statischer Variablen, um zu zählen, wie häufig sie aufgerufen wurde:

```
int f(){
  static int count;
  return ++count;
}
```

Das Schlüsselwort `static` hat leider noch ein paar weitere, scheinbar unverwandte Bedeutungen. Für Variablen, die global außerhalb aller Funktionen deklariert sind, bewirkt es, dass deren Name nur in der Übersetzungseinheit sichtbar ist, in der die Deklaration erfolgte. Ansonsten sind solche Variablen global für das ganze Programm sichtbar, bzw. in dem Namensraum der Deklaration (vgl. 2.6).

Für Variablen, die in Klassenkörpern deklariert sind, bewirkt `static`, dass eine einzige Instanz der Variablen über alle Instanzen von Objekten dieser Klasse existiert. Dies ermöglicht eine Art globale Kommunikation zwischen verschiedenen Objekten der gleichen Klasse, die vor der Außenwelt verborgen bleibt.

`register` Das Schlüsselwort `register` bewirkt, dass der Compiler versucht, die Variablen soweit möglich in Prozessorregistern zu halten. Da dies von neuen Compilern ohnehin als Optimierungsstrategie angewandt wird, wird dieses Schlüsselwort in der Regel ignoriert. Schlimmstenfalls kann es schaden; man sollte es also nicht mehr verwenden. Von Variablen, die explizit als `register` erklärt wurden, kann keine Adresse erhalten werden (vgl. Abschnitt 2.3.4).

const, volatile, mutable

Variablen, die `const` erklärt werden, dürfen nicht mehr verändert werden. Dies kann dazu dienen, in einen Namensraum Konstanten einzuführen. Der Compiler kann diese bereits

zur Compilezeit weg optimieren. Diese Technik ersetzt auch die unter C weit verbreitete Gewohnheit, solche Ersetzungen durch den Präprozessor machen zu lassen. Dessen Makros kollidieren allerdings manchmal mit Template-Parametern, die meist ebenso wie Makros vollständig großgeschrieben werden, was dann zu merkwürdigen Effekten führt. Die konsequente Benutzung von `const` erlaubt dem Compiler weitere Optimierungen, so dass man Variablen und Funktionsargumente sorgfältig daraufhin überprüfen sollte, ob es möglich ist, sie als `const` zu erklären.

```
namespace xyz {
const int   Size = 100;
const float Pi = 3.1415926;
}
```

`mutable` kann bei Klassenvariablen dazu benutzt werden, dem Compiler die Veränderung einer Variablen zu erlauben, auch wenn das Objekt als solches in einem Kontext verwendet wird, in dem es eigentlich `const` ist. `mutable` kann dazu dienen, auszudrücken, dass eine Größe konzeptionell und nicht bitweise konstant ist.[2]

`volatile` dient dazu, den Compiler darauf hinzuweisen, dass eine Variable ihren Wert ändern kann, auch ohne dass eine Zuweisung oder ein anderweitiger Zugriff auf die Variable durch das Programm erfolgt ist. Dieses Verhalten ist typisch für hardware-Register, die im Adressraum des Programmes liegen. Der Compiler muss in diesem Fall von zu aggressiver Optimierung abgehalten werden.

Zusammenfassung:
(i) `static` dient zur Kennzeichnung langlebiger lokaler Variablen, `auto` und `register` sind Schlüsselworte, werden aber nur noch sehr selten genutzt. Bei globalen Variablen schränkt `static` den Sichtbarkeitsbereich des Namens auf die jeweilige Übersetzungseinheit ein.
(ii) `const` hilft dem Compiler bei der Optimierung und dem Benutzer zur Garantie eines nur lesenden Zugriffs auf eine Variable. Es sollte daher konsequent wo immer möglich eingesetzt werden. `const` deklarierte Variablen ersetzen die in C üblichen Präprozessormakros zur Festlegung von Konstanten.

Speicher	cv-qualifier	Größe	Vorzeichen	Typ

`static (s)`
`register (r)`
`auto (a)`
	`const (c)`
	`volatile (v)`
	`mutable (m)`
		`short (s)`
		`long (l)`
		`(long long) (ll)`
			`signed (s)`
			`unsigned (u)`	

				`void`
`[s, a, r]`	`[c, m] [v]`			`bool`
`[s, a, r]`	`[c, m] [v]`			`wchar_t`
`[s, a, r]`	`[c, m] [v]`			`float`
`[s, a, r]`	`[c, m] [v]`		`[s, u]`	`char`
`[s, a, r]`	`[c, m] [v]`	`[s, l, ll]`	`[s, u]`	`int`
`[s, a, r]`	`[c, m] [v]`	`[l, ll]`		`double`

Tabelle 2.1: Übersicht über erlaubte Kombinationen von Eigenschaftspezifizierungen von sprachbereitgestellten Typen. Aus jeder eckigen Klammer kann eine Option gewählt werden. Speicherort wird vor cv-Qualifikation angegeben. Größe und Vorzeichen können ggf. getauscht werden. Der Typ `int` kann entfallen, wenn Vorzeichen oder Größe festgelegt werden.

[2] Denken wir uns ein Zeichenkettenobjekt, dass mittels Referenzen auf eine Repräsentation verweist. Solche Objekte müssen Referenzzähler verwalten, die beim Kopieren (konzeptionell eine Leseoperation, bei der das "abstrakte" Objekt nicht verändert wird) modifiziert werden müssen — typische Kandidaten für eine `mutable` Qualifikation.

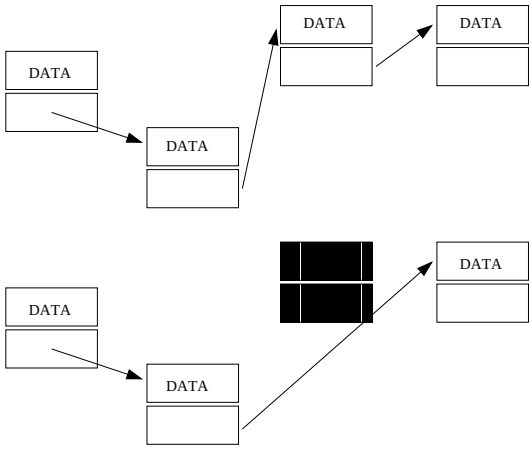


Abbildung 2.1: Benutzung von Zeigern am Beispiel einer verketteten Liste. Deren Elemente bestehen aus den eigentlichen Daten und einem Zeiger auf das jeweils folgende Element. Oben sieht man die Liste vor dem Entfernen eines Elementes, unten danach.

2.3.4 Zeiger, Referenzen und Felder

Zeiger

Vom Blickpunkt eines Programmes ist jede erreichbare Speicherzelle im virtuellen Speicher durch eine eindeutige Zahl gekennzeichnet (Teile des virtuellen Adressbereiches können auf Harddisk gehalten sein, andere können im RAM liegen, wieder andere, gegenwärtig unbenutzte, müssen vom Betriebssystem gar nicht physikalisch repräsentiert sein). Auf regulären PC's mit "modernen" Betriebssystemen lassen sich diese Adressen im Zahlbereich normaler `int` Variablen repräsentieren, auf anderen Architekturen können sie darüber hinaus gehen. C/C++ bietet Möglichkeiten diese "Namen" der Speicherstellen für Programmzwecke nutzbar zu machen. Wenn z.B. gefordert ist, dass Daten in eine geordnete Liste schnell eingefügt und später wieder gelöscht werden müssen, so ist diese Funktionalität nicht durch die Anordnung der Zahlen in einem linearen Feld zu erreichen. Sowohl das Einfügen als auch das Entfernen einer Zahl aus diesem Feld erfordert das Umkopieren aller "oberhalb" des modifizierten Elementes liegenden Daten, im Mittel also etwa halb so viele Operationen wie das Feld Elemente aufweist. Dieser Aufwand ist nur bei kleinen Feldern akzeptabel. Eine Teillösung dieses Problems liegt in einer verketteten Liste, in der jedes Element noch die Information erhält, "wo" das jeweils nächste im Speicher liegt (d.h. welches der Name der Speicherzelle ist, die den Wert des Elementes speichert). Die tatsächliche Position des Elementes spielt keine Rolle, so dass immer nur die Speicherzellennamen manipuliert werden müssen. Um zum Beispiel ein Element x zu entfernen, muss nur der Namenseintrag des Vorgängers von x in der Liste so geändert werden, dass er nicht mehr auf x sondern auf dessen Nachfolger zeigt. Kopieren der Elementinhalte ist nicht mehr nötig.

Um den Umgang mit solchen Speicherzellennamen unabhängig vom jeweiligen Rechner zu gestalten, hält C/C++ Zeiger bzw. Pointer-Typen bereit. Ein Zeiger kann auf einen beliebigen Datentyp verweisen, der auch wieder ein Zeiger sein kann. Zur Deklaration benutzt man das Symbol `*`. Um die Adresse einer Variablen zu erhalten, verwendet man den `&` Operator.

```
int  i = 5;        // Initialisierung
const int c_i(i);  // Initialisierung mit einer Variablen
int* p_i = &i;     // p_i ist ein Zeiger auf einen int,
                   //   hier die Variable i
```

```
const int *cp_i = &i; // i kann nicht durch Zugriff ueber
                      // cp_i geaendert werden
//  p_i  = &c_i;  FEHLER, durch p_i koennte sonst c_i geaendert werden
int **pp_i = (&p_i);  // Zeiger auf einen Zeiger auf einen int
```

Konstante Zeiger werden durch das Setzen von `const` *rechts* vom Zeigersymbol `*` deklariert. Sie müssen bereits bei der Deklaration initialisiert werden. Jede weitere Zuweisung auf den Zeiger wird vom Compiler als Fehler erkannt.

```
const char newline = '\n';
const char null = '\0';
// const Zeiger auf const char:
const char * const p_nl = &newline;
// p_nl = '0';   FEHLER
```

Um den Wert einer Speicherstelle zu ändern, auf die ein Zeiger verweist, oder diese in einer Zuweisung oder Rechnung zu benutzen, benutzen wir den unären `*` Operator.

```
*p_i = 27;              // auf i wird 27 zugewiesen
*p_i = 2 * (*p_i) + 5;  // aequivalent zu i = 2 * i + 5;
```

Das Symbol `*` hat damit zwei Bedeutungen. Steht es in einer Variablendeklaration, d.h. bei einem Typ, so kennzeichnet es einen Zeiger. Vor einer Variablen (von Zeigertyp) bewirkt der Operator `*` den Zugriff auf den Inhalt der Speicherstelle, auf die der Zeiger verweist.

Die Motivation, die hinter den Regeln für Zuweisung, Initialisierung oder Konversion von const (oder volatile) deklarierten Zeigern oder Mehrfachzeigern steht, ist dass die Sprache garantieren soll, dass die als const deklarierten Größen nicht durch Manipulation des betroffenen Zeigers geändert werden können.

```
double d(5);
const double cd(d);       // konstante Kopie von d
double *pd(&d);           // Zeiger auf d
const double *cpd(&d);    // d kann durch cpd nicht geaendert werden
double **ppa(&pd);        // ok
const double **cppd(&pd); // FEHLER!
```

Wäre die letzte Initialisierung legal, könnte versehentlich ein konstantes Objekt verändert werden:

```
const double pi(3.1415926);
*cppd = &pi;     // legal, pd zeigt jetzt auf pi, nicht mehr auf d
*pd   = 2.71828; // aendert ueber pd den Wert von pi!
```

Analog zu `*` hat das Symbol `&` zwei Bedeutungen. Bei einer Variablendeklaration, d.h bei einem Typ kennzeichnet er eine Referenz. Vor einer Variablen ist es der "address of" Operator, der die Adresse der Variablen liefert.

Mit Zeigern kann Arithmetik getrieben werden; im obigen Beispiel lässt z.B. `p_i += 5` den Zeiger um 5 Speicherstellen für `int` weiterwandern. Zur Durchführung dieser Operation benötigt der Compiler die Information, wie groß der Datentyp ist, auf den verwiesen wird. Mit dieser Information kann der Compiler dann die richtige Schrittweite berechnen, um zum fünften Element nach `p_i` zu gelangen. Ebenfalls kann die Differenz von Zeigern gebildet werden, wobei das Ergebnis ein `int` ist. Ein definiertes Ergebnis hat diese Operation jedoch nur, wenn die beiden Zeiger in ein und dasselbe Datenfeld verweisen, s.u..

Wenn der Wert einer Speicherstelle benötigt wird, die man durch Fortschalten des Zeigers um eine gewisse Zahl von Speicherstellen erhält, so gibt es dafür eine kurze Notation mittels des "Feldzugriffsoperators" `[]`:

```
  *(p_i + 25 ) = 15;   // value of 25th past element pointed
                       // to by p_i is 15
  p_i[25]      = 15;   // equivalent
```

Natürlich sind auch negative "Indizes" oder Offsets zulässig. Sollten diese Operationen in Speicherbereiche hineinführen, die nicht explizit vom Programm angefordert wurden, so ist das Resultat jedoch undefiniertes Verhalten. Auf UNIX Systemen erhält man dabei häufig "segmentation violation", aber auch das Überschreiben von anderweitig im Programm definierten Variablen kann vorkommen.

Der Sprachstandard garantiert, dass 0 niemals als Adresse eines wirklichen Datenelementes vorkommt. Dieser Wert lässt sich also verwenden, um irgendwie geartete Fehler anzuzeigen, die bei einer Operation mit Zeigern aufgetreten sind.

Felder

Felder werden in C/C++ mittels `[]` deklariert. Gleichzeitig dient dieser Operator auch zum Zugriff auf Feldelemente. Felder beginnen in C/C++ immer mit dem Index 0, der letzte gültige Index hat daher den Wert der um eins verringerten Größe des Feldes. Mehrdimensionale Felder werden durch mehrfache `[]`-Paare deklariert.

```
#include <string>

int main(){
  // Feld von 20 int's:
  int         iarr[20];
  // Feld von 20 Zeichenketten:
  std::string str[20];
  // Feld von 3 strings, initialisiert mit Werten
  std::string str2[] = { "Matthias", "Stefan", "Oliver" };
  // 'zweidimensionales' Feld von 20x30 Elementen vom Typ int
  int         ia[20][30];
  for ( int i=0; i < 20; i++ )
    for ( int j=0; j < 30; j++ )
      ia[i][j] = i * j;
}
```

Zwischen Zeigern und Feldern besteht ein enger Zusammenhang (Zeiger-Feld-Dualität), der sich bereits oben in der Verwendung von `[]` zur "indirekten" Dereferenzierung von Zeigern andeutete. Ein Feld a[..] wird intern durch den Compiler sofort in einen Zeiger auf das erste Element verwandelt. Der Feldzugriff wird dann vollständig nach den Regeln der Zeigerarithmetik abgewickelt. Das bedeutet, dass bei Zugriffen zunächst der in den eckigen Klammern stehende Wert auf diesen Zeiger addiert und dann der `*` Operator angewandt wird. C/C++-Felder beginnen immer bei 0, so dass bei dieser Operation kein Offset berücksichtigt werden muss.

```
 int a[20];
 int *p1 = a;   // a ist Name des Feldes und kann als Adresse des
                //   ersten Elementes benutzt werden
                //   int *p1 = &(a[0])  ist auch moeglich
 p1[5] = 4;     // Zugriff mittels Zeigerarithmetik, a[5] ist jetzt 4
  a[5] = 4;     // aequivalent

 int b[20][30]; // b ist ein Feld von 20 Elementen, die ihrerseits
                //   Felder von 30 Elementen vom Typ int sind
```

```
// int *p2 = b;   FEHLER: falscher Typ
int  *p2=b[0]; // ok, p zeigt auf den ersten int im ersten Feld
               //   von 30 ints
p2[29] = 7;    // setzt  a[0][29] auf 7
int  *p3[30];  // deklariert ein Feld von 30 Zeigern auf int
int (*p4)[30]; // deklariert einen Zeiger auf ein Feld von 30 int
      p4 = b;  // ok, p4 kann jetzt anstelle von b benutzt werden
p4[12][15]  = 26; // ok
 b[12][15]  = 26; // aequivalent
*(p2 + 12*30 + 15) = 26; // aequivalent
```

Um ein Feld an eine Funktion (vgl. 2.5) zu übergeben, muss man einen entsprechenden Zeiger übergeben. In einer Dimension ist das einfach ein Zeiger auf das erste Datenelement, in zwei Dimensionen muss das ein Zeiger auf ein Feld der passenden (festen) Anzahl von Elementen sein. Nur die erste Dimension in der Deklaration kann dabei entfallen, alle anderen müssen zur Compilezeit feststehen, damit der Compiler den für die Zeigerarithmetik nötigen Code erzeugen kann. Es gibt keine wirkliche Notwendigkeit für diese Einschränkung, die C-Felder praktisch unbrauchbar für den Gebrauch in realem numerischen Code macht.[3] Man benutzt sie daher häufig für kurze Vektoren oder kleine Matrizen, bei denen die Größe z.B. durch die Anzahl der räumlichen Dimensionen des Problems gegeben ist und sich während eines Programmlaufs nicht ändert.

```
void f( int a[][30] ){ /*... */  }
void g( int (*p)[30] ){ /* .. */ }

int  a[20][30];
int a2[10][30];
int  b[15][31];

int main(){
  f(a); g(a); f(a2); g(a2); // ok
  // f(b); g(b);               FEHLER, zweite Dimension nicht == 30
}
```

Wir werden später (5.4.1) sehen, wie man dynamische, mehrdimensionale Felder einfach z.B. mit Hilfe des `vector`-Typs der Standardbibliothek erzeugen kann.

Wir wollen hier kurz auf die dynamische Speicherverwaltung vorgreifen und eine weitere Technik zur dynamischen Erzeugung mehrdimensionaler Felder schildern. Der `operator new` *type*`[`*N*`]` gibt einen Zeiger auf das erste Element eines linearen Speicherbereiches von `N` Variablen des Typs *type* zurück. Um ein zweidimensionales Feld von, sagen wir, Dimension N×M zu erzeugen, kann man zunächst einen Speicherbereich von N×M Elementen anfordern. Dann kann man ein Hilfsfeld von Zeigern einrichten, die auf die Positionen der Variablen `a[i][0]` verweisen, $\mathtt{i} = 0..N-1$. Ein weiterer Zeiger auf die erste Position dieses Hilfsfeldes kann syntaktisch wie ein zweidimensionales Feld behandelt werden. Dieser kann dann auch unproblematisch an Unterprogramme weitergereicht werden und erlaubt damit die Übergabe dynamisch angelegter Felder.

```
#include <iostream>

// Funktion, die ein 2D Feld erwartet
void f(int **a, int rows, int columns){ /* ... */ }
```

[3] Aktuelle Versionen des GNU-Compilers lassen solche dynamischen Felder zu. Es gibt jedoch in C++ Klassen wie `std::vector` oder `std::valarray` die die Funktionalität von Feldern variabler Größe bereitstellen.

```
int main(){
  // ...
  int  nx, ny;
  std::cin >> nx >> ny; // Felddimensionen einlesen
  // Zeiger auf das erste Element des Hilfsfeldes
  int **array;
  // Hilfsfeld fuer nx Zeiger auf int
  array = new int *[nx];
  // Speicher anfordern fuer nx * ny Feldelemente
  array[0] = new int[nx*ny];
  // array[0] ist jetzt ein Zeiger auf die nullte 'Spalte' von
  //   ny Elementen und zahlenmaessig gleich dem Zeiger auf den
  //   Anfang des Gesamtfeldes

  // restliche Spaltenanfaenge berechnen und aufs
  //   Hilfsfeld zuweisen
  for(int i=1; i < ny; ++i )
    array[i] = array[0] + i * ny;

  if ( nx > 23 && ny > 10 )
    array[23][10] = 17;  // ok wenn 23 < nx und 10 < ny

  // Aufruf von f mit array
  f( array, nx, ny );
}
```

Diese Technik lässt sich für Felder mit beliebig vielen Indizes anwenden und führt dann zu entsprechend hohem Grad der verwendeten Zeiger. Wir gehen in Abschnitt 5.4.1 auf alternative Möglichkeiten ein.

Referenzen

Eine Referenzvariable vereinbart einen zweiten Namen für eine bereits bestehende Variable gleichen Typs. Daher ist klar, dass eine Referenzvariable bei der Deklaration initialisiert werden muss. Danach ändern Zuweisungen nur den Inhalt der referenzierten Variable, aber nicht die Referenzvariable selbst (das wäre gleichbedeutend mit der Etablierung eines neuen Namensaliases). Viele Compiler implementieren den zugrunde liegenden Mechanismus mit Hilfe von Zeigern und wir wollen zwei Programmsegmente, die die gleiche Aufgabe mit Zeigern und mit Referenzen erledigen, gegenüberstellen. Es kann jedoch nicht genug betont werden, dass Referenzen keine Zeiger sind und die jeweilige Implementierung vollständig dem Compilerbauer überlassen ist.

```
int  i;                          //    int i;
int &i_2 = i;                    //    int * const i_2 = &i;
i_2 = 5;   // i ist jetzt 5      //    *i_2 = 5;
i = i_2 * i + 5; // i == 30      //    i = (*i_2) * i + 5;
std::cout << i_2;                //    std::cout << *i_2;
```

Man sieht, dass sich eine ähnliche Funktionalität auch mit konstanten Zeigern erreichen läßt, dass aber die Benutzung des "value of" Operators * entfällt. Referenzen sind daher einfacher zu verwenden als konstante Zeiger und daher auch weniger fehlerträchtig.

Es ist entscheidend, dass Referenztypen auch von Funktionen zurückgeliefert werden können und damit Funktionen sinnvoll auf der linken Seite von Zuweisungen stehen können.

```
int &access( int a[], int i ){ return a[i]; }

int main(){
  int a[20];
  access(a,10) = 5;  // ok, a[10] is 5
}
```

Der Sinn dieses Beispiels wird klarer, wenn man sich vorstellt, dass in `access()` Bereichsgrenzentests oder Ähnliches stattfinden können.

Konstante Referenzen können als Funktionsargumente dienen, wenn die Funktion nur lesend auf die Variable zugreift. In C++ kann die Übergabe als `const &` häufig das aufwendige Kopieren einer Klassenvariable verhindern und laufzeiteffizienteren Code erzeugen.

Die Übergabe als nicht konstante Referenz wie die Übergabe von Information per Zeiger verhindert häufig Optimierungen (Stichwort: Aliasing von Variablen) und sollte nur dort verwendet werden, wo sie wirklich nötig ist. Auch wird der Compiler nicht ohne Warnung erlauben, temporär erzeugte Variablen, wie die Werte von Ausdrücken oder Zwischenergebnisse einer Rechnung, als nicht konstante Referenzen an Funktionen zu übergeben.

2.3.5 `void`

Das Schlüsselwort `void` kann manchmal anstelle einer Typangabe stehen. Es dient dazu, Funktionen zu kennzeichnen, die keinen Wert an die aufrufenden Umgebung zurückgeben. In C dient `void` als "Argumenttyp" einer Funktion auch dazu, das Fehlen von Argumenten anzudeuten. In C++ wird dazu einfach kein Funktionsargument geschrieben. Ein Zeiger auf `void` ist eine generische Zeigervariable. Es ist garantiert, das jeder Zeiger auf einen beliebigen Datentyp in einen Zeiger auf `void` umgewandelt werden kann, ohne dass dabei ein irreversibler Informationsverlust auftritt. In C wurden daher `void *` häufig dazu eingesetzt, Information an Routinen zu übergeben, die mit beliebigen Zeigern arbeiten konnten (z.B. Sortierroutinen wie `quicksort`, Rückgabewert der Speicherallozierungsroutine `malloc`). In C++ sollten in solchen Fällen besser `template`-Funktionen zum Einsatz kommen, die die Typsicherheit des Programmes erhalten.

2.3.6 `typedef`

`typedef`'s erlauben, neue Namen für bestehende Typen einzuführen, die sich exakt wie der ursprüngliche Typ verhalten. Es wird dabei kein neuer Typ eingeführt.

```
typedef int     Int32;
typedef struct{ int lower, upper; } Interval;
```

Sie können unter anderem dazu dienen, die etwas kryptische Deklarationssyntax von C++ freundlicher zu gestalten. Das folgende Beispiel zeigt die Definition eines Feldes von Zeigern auf Funktionen.

```
// Deklaration ohne typedef
double (*a[20])(double);

// DblFctInt ist eine Funktion mit Rueckgabetyp double
//  und Argumenttyp int
typedef double DblFctInt(int);

// hier ein Zeiger auf eine solche Funktion
typedef DblFctInt *Ptr_DblFctInt;
```

```
// und ein Feld zwanzig solcher Zeiger
// Deklaration aequivalent zu erster [double (*...)()]
Ptr_DblFctInt b[20];
```

2.4 Kontrollstrukturen

Für die Umsetzung von Algorithmen in einer bestimmten Programmiersprache ist es essentiell, dass diese Sprache Kontrollstrukturen aufweist, die in der Lage sind, bestimmte Anweisungsblöcke nur unter bestimmten Bedingungen oder mehrfach abzuarbeiten. Die in C (und C++) meist verwendeten Kontrollstrukturen sind das `if`-Statement zur bedingten Ausführung und die `for`-Schleife zur Wiederholung von geblockten Anweisungen. Daneben existieren noch die und abweisenden `while`- und nichtabweisenden `do ... while` Schleifen, das `switch() case:`-Statement, das aufgrund des Inhalts einer Variablen einen auszuführenden Block anspringt und das verpönte `goto`, das sich manchmal sinnvoll dazu verwenden lässt, aus tief verschachtelten Schleifen herauszuspringen.

2.4.1 if-Anweisung

Eine `if`-Anweisung kann, muss aber keinen `else`-Teil enthalten, der durchlaufen wird, wenn der Bedingungsteil zu logisch falsch auswertet. Sowohl im `if` als auch im `else`-Zweig können entweder einzelne Anweisungszeilen (abgeschlossen durch Semikolon) oder in Klammerpaare `{}` gesetzte Anweisungsblöcke auftreten. Ein `else`-Block bezieht sich immer auf die gerade vorstehende `if`-Anweisung. Im folgenden Beispiel ist die zweite `if`-Anweisung ein einzelnes Statement im `else`-Teil der ersten.

```
#include <iostream>
#include <cmath>

int main(){

  double sol1, sol2;
  double p= 2., q= 3.; // or other values

  double rad = 0.25*p*p - q;
  if ( rad > 0. ) { // begin of if block
    sol1 = 0.5*p + std::sqrt(rad);
    sol2 = 0.5*p - std::sqrt(rad);
  } // end if block
  else if ( rad < 0. )
    std::cout << "graph does not cut the x-axis\n";
  else   // refers to last if(rad < 0.)
    sol1 = sol2 = 0.5 * p;

  if ( sol1 > 0. )
    std::cout << "graph cuts positive x-axis\n";

  // else branch missing

  // ...
}
```

2.4.2 Ternärer ?:-Operator

Als Einschub soll hier der ternäre Operator `?:` behandelt werden. Dieser kann dazu dienen, innerhalb von Ausdrücken Teilausdrücke bedingt auszuwerten.

```
inline int abs(int number) {
  // berechne Absolutbetrag
  return number > 0 ? number : -number;
}
```

Ist der vor dem Fragezeichen stehende Ausdruck wahr, so wird der Teilausdruck vor dem Doppelpunkt, sonst derjenige nach dem Doppelpunkt ausgewertet. Der resultierende Typ der beiden alternativen Ausdrücke muss gleich sein. Aufgrund der geringen Priorität dieses Operators und zur Verdeutlichung der Zusammengehörigkeit der Teilausdrücke ist es in vielen Fällen sinnvoll, großzügig Klammern zu setzen.

```
  int i1   = 3 > 4 ? 0 : 1;          //  i1   is 1
  int i1_3 = 3 * ( 3 > 4 ? 0 : 1 );  //  i1_3 is 3
  int i2_3 = 3 *   3 > 4 ? 0 : 1;    //  i2_3 is 0
```

2.4.3 (do) while-Schleifen

Schleifen, die mit `while` gebildet werden, werden so oft durchlaufen, bis der Bedingungsteil der Anweisung `false` wird. Falls die Bedingung bereits falsch ist, wenn das Programm auf die `while`-Schleife trifft, so wird der Schleifenkörper nicht durchlaufen.

```
  #include <iostream>

  int main(){
    // ...
    char c=0;
    while( c != 'y' && c != 'n' )
      std::cin >> c;  // kann auch ein Block in { } sein
    // ...
  }
```

Das obige Beispiel erfordert die Initialisierung der Zeichenvariablen mit einem Wert, so dass die Schleife auf jeden Fall durchlaufen wird. In solchen Fällen ist häufig die Verwendung einer nicht abweisenden Schleife mit `do ... while(...)` intuitiver:

```
  char c;
  do
    std::cin >> c;    // kann auch ein Block in { } sein
  while( c != 'y' && c != 'n' );
```

2.4.4 for-Schleifen

Häufig enthalten Schleifen neben der Schleifenbedingung noch einen Initialisierungsteil und möglicherweise Kontrollanweisungen, die den Wahrheitsgehalt der Schleifenbedingung beeinflussen und jedes mal während eines Durchlaufes ausgeführt werden. Solch eine Schleife könnte, programmiert mit `while`, etwa so aussehen:

```
  int a[20];
  // ..
  int i=0;            // Schleifeninitialisierung
```

```
while ( i < 20 ) {  // Schleifenbedingung, abweisend falls falsch
  a[i] = i;
  i++;              // Schleifenkontrolle, durchgefuehrt nach jedem
                    //   Durchlauf
}
```

Zur Abkürzung lässt sich in diesem Falle `for(;;)` verwenden. Die folgende Konstruktion mit `for` ist der obigen (bis auf den Sichtbarkeitsbereich der Variablen `i` vollständig äquivalent.

```
int a[20];
// ..
for( int i=0; i < 20; i++ )
  a[i] = i;           // benutze Elemente 0 .. 19
```

Sie besitzt mehrere Vorteile. Erstens ist die Schleifenkontrolle in den drei durch Semikolon getrennten Teilen der `for`-Anweisung gut sichtbar zusammengefasst und nicht über den Schleifenkörper verstreut. Zweitens ist der Verwendbarkeitsbereich der Variablen `i` auf die Schleifenkontrollstruktur und den Schleifenkörper beschränkt. Drittens kann bei "einfachen" Schleifen der Compiler höhere Optimierungen vornehmen, was vor allem auf vektorisierende Compiler zutrifft.

Als weiteres Beispiel für die Benutzung von `for` sei die Bearbeitung einer verketteten Liste angeführt:

```
struct Element {
  int      value;
  Element *next;
};

void f(){
  Element elem;
          elem.next = 0;
  Element *root = &elem;
  //...
  for ( Element *next=root; next != 0; next = next->next )
    next->value = 42;  // benutze next->value;
}
```

Der Kommaoperator kann ggf. dazu dienen, komplexere, zusammengesetzte Bedingungen in der Schleife unterzubringen. Das folgende Beispiel liefert einen Zeiger auf das erste Element > 0 in einer Untersequenz begrenzter Länge in einem Feld.

```
int a[20];
int *p, i;
int segment_length=4;
// ...

for( i=0, p=a; i < segment_length && *p < 0 ; p++, i++ )
  ;  // leerer Schleifenkoerper
```

Da die Übersicht stark leidet, sollte man dieses Mittel sparsam einsetzen und ggf. wieder auf `while` Schleifen ausweichen. Gleichermaßen dürfen Initialisierungs-, Test-, und Kontrollteil des `for` leer sein. Der leere Testteil entspricht dabei einer immer wahren Bedingung.

2.4.5 break und continue

Aus einer einzigen nicht verschachtelten Ebene einer mittels `for` oder `while` gebildeten Schleife kann man durch die Benutzung von `break` herausspringen. Die Schleifenbearbeitung wird abgebrochen und direkt nach der Schleife wieder aufgenommen, ohne dass noch weitere Kontrollanweisungen oder Schleifentests durchgeführt werden. Das `break` kann also z.B. dazu dienen, Schleifen frühzeitig, z.B. bei Fehlerbedingungen, abzubrechen oder "Endlosschleifen" zu verlassen.

```
#include <errno.h>

void read_from_stdin(){ /* setzt errno */ }
void write_to_stdout(){ /* dto.        */ }

int main(){
  while( true ){
    read_from_stdin();
    // error ist globale Zustandsvariable, gesetzt nach Aufrufen
    // der C-Bibliothek, ggf. Kopiervorgang abbrechen
    if ( errno > 0 ) break;
    write_to_stdout();
  }
}
```

Die `continue` Anweisung bewirkt, dass die Bearbeitung des Schleifenkörpers an dieser Stelle abgebrochen und dann bei der Kontrollanweisung bzw. dem Schleifentest wieder aufgenommen wird.

```
#include <cmath>

int main(){
  // ...
  double a[20];
  // ...
  double sum_sqrt=0.;
  for( int i=0; i < 20; i++ ){
    if ( a[i] < 0. ) continue;   // naechstes Element benutzen
    sum_sqrt += sqrt( a[i] );
  }
  // ...
}
```

Es sei nochmals betont, dass `continue` und `break` sich nur auf die innerste Schleifenebene beziehen und nicht dazu benutzt werden können, tiefer verschachtelte Schleifen zu verlassen.

2.4.6 goto-Anweisung, Sprungmarken

Zu dem Zweck tiefer verschachtelte Schleifen zu verlassen, kann die oft verpönte `goto` Anweisung benutzt werden.

```
void f(){
  int a[20][30];

  for( int i=0; i<20; i++ )
```

```
    for( int j=0; j <30; j++ )
      if ( a[i][j] == 42. )  // Antwort gefunden?
        goto end_of_loop;    // ja, simuliere ein break ueber
                             //   mehrere Verschachtelungsebenen
end_of_loop: ; // Marke: wir springen hierher von innerhalb der Schleife
}
```

Die Marke `end_of_loop` kennzeichnet das Sprungziel des `goto`. Der Name des Sprungzieles kann innerhalb des Funktionskörpers frei gewählt werden und wird durch einen Doppelpunkt abgeschlossen. Das Problematische an `goto`-Anweisungen ist, dass man den zugehörigen Sprungmarken nicht ansehen kann, *von wo* aus das `goto` erfolgt, mit dem sie angesprungen werden. Außerdem neigt die Verwendung eines `goto` dazu, weitere nach sich zu ziehen. Im obigen Beispiel müsste vermutlich nach der Schleife eine Behandlung des "nicht-gefunden" Falles anschließen und man müsste dann das Ansprungziel des `goto` mit einem weiteren `goto` umgehen. Besser setzt man oben eine Variable `found` auf einen bestimmten Wert und benutzt diesen dann in weiteren Verzweigungsanweisungen. Mit `goto` dürfen lokale Blöcke Klammerebenen verlassen, aber nicht betreten werden. Man darf beispielsweise nicht vom `if`-Teil einer Entscheidung in den `else`-Teil springen, oder von außen in einen Schleifenkörper hinein. Ebenso darf der Kontext einer Funktion nicht verlassen werden.

Durch Ersetzen des `goto` durch die Anweisungen nach dem Sprungziel können `goto`'s immer vermieden werden. Wie wir oben gesehen habe, sind sie allerdings in Einzelfällen bei der Programmstrukturierung hilfreich. Wie so häufig entscheidet die Dosis über Gift- oder Heilwirkung...

2.4.7 switch-Anweisung

Ein gezähmtes `goto` ist die switch-Anweisung, die je nach Wert einer ganzzahligen Variablen verschiedene Sprungziele einer direkt anschließend mit `case`-Anweisungen gebildeten Liste auswählt und die Verarbeitung dann an dieser Stelle fortsetzt. Im Prinzip können `switch`-Anweisungen immer durch `if...else if`-Konstrukte ersetzt werden sind aber häufig übersichtlicher. In objektorientierten Sprachen und C++ insbesondere lassen sich anstelle `switch`-artiger Konstrukte häufig polymorphe Objekte verwenden (vgl. Abschnitt 3.2.5).

```
#include <iostream>

void decide(char c){
  switch( c ){
  case 'a': std::cout << "hello world!";
            break;
  case 'b': std::cout << "HELLO ";
          // absichtliches Auslassen von break, um fuer 'b'
          // HELLO WORLD! auszugeben
  case 'c': std::cout << "WORLD!";
          break;
  default:  std::cerr << "unknown input";
          break;
  }
}
```

Das obige Programmsegment gibt Kleinbuchstaben aus, wenn der Buchstabe `a` übergeben wurde, im Falle von `c` das Wort `WORLD!` und im Falle von `b` die Zeichenfolge `HELLO WORLD!`.

Dieses merkwürdig erscheinende Verhalten entsteht dadurch, dass nach Ansprung der passenden `case`-Marke alle folgenden Anweisungen bis zu einem eventuellen `break` durchlaufen werden. Da dies häufig nicht das beabsichtigte Verhalten ist, lohnt es sich immer einen entsprechenden Kommentar zu hinterlassen. Die `default`-Marke ist optional und wird angesprungen, falls sich sonst keine passende in der `case`-Liste findet.

2.5 Funktionen und Operatoren

2.5.1 Deklaration

Außer in wenigen Ausnahmefällen erzwingt C++ die Spezifikation von Argument und Rückgabetypen von Funktionen.

```
double ipow(double  base, int i);
```

deklariert eine Funktion, deren Returnwert in einem Ausdruck an die Stelle einer `double` Variablen treten kann und die zwei Argumente erwartet, vom Typ `double` und vom Typ `int`.

Im Unterschied zu C ist die Angabe des Rückgabetyps zwingend vorgeschrieben. In C führte das Weglassen des Rückgabetyps zur Annahme, dass es sich um eine Funktion handelt, die `int` zurückliefert. Im neuen C99-Standard ist dies allerdings ebenfalls verboten. Für Funktionen, die keinen Rückgabewert liefern, gibt es das Schlüsselwort `void` (siehe Kapitel 2.3.5).

```
void abort();
```

Während die Angabe des/der Argumenttyps/en zwingend ist, ist die eines Namens für die Argumentvariable/n dem Benutzer freigestellt. Es ist jedoch guter Stil, hier einen Namen anzugeben, um die Bedeutung des Argumentes klar zustellen. Eine Funktion muss deklariert sein, bevor sie im Programm verwendet werden kann. Deklarationen können im Prinzip beliebig häufig wiederholt werden, sie sind ein Versprechen an den Compiler, diese Funktion später oder in einem anderen Programmteil zu deklarieren. Reguläre Funktionen sind innerhalb eines Namensraumes immer globale Objekte. Es gibt keine Möglichkeit, Funktionen innerhalb lokaler Bezugsrahmen zu definieren, z.B., innerhalb eines geschweiften Klammerpaares, oder eines `begin`/`end`-Blockes wie in PASCAL. Man beachte, dass Deklarationen durch ein Semikolon abgeschlossen werden.

2.5.2 Definition

Die Definition einer Funktion ist die Festlegung des Programmtextes, der bei Aufruf der Funktion ausgeführt werden soll. Geht die Definition einer Funktion ihrer Benutzung voraus, so kann sie eine Deklaration ersetzen. Die Spezifikation des Programmsegmentes für die Funktion erfolgt innerhalb eines geschweiften Klammerpaares, das *nicht* durch ein Semikolon abgeschlossen wird.

```
double ipow(double base, int exponent)
{
   double result = 1.;
   int    neg_exp = 0;
   if (exponent < 0 ) {
      neg_exp = 1;
      exponent = -exponent;
   }
   for (int i=0; i < exponent; i++ )
```

```
        result *= base;
    if ( neg_exp )
        result = 1. / result;
    return result;
}
```

Das Schlüsselwort `return` bricht das Unterprogramm an der spezifizierten Stelle ab, legt den Wert des dahinter angegebenen Ausdrucks auf dem Stack ab und kehrt ins Hauptprogramm zurück. Im obigen Fall besteht dieser Ausdruck nur aus der Variable `result`. Runde Klammern werden an dieser Stelle nicht benötigt. Ein Unterprogramm kann mehrere `return`-Anweisungen enthalten. Dies kann ein nützliches Mittel sein, um z.B. bei Angabe unpassender Argumenttypen oder Problemen bei der Berechnung die Bearbeitung frühzeitig abzubrechen, kann aber auch zu Unübersichtlichkeit bei längeren Funktionen führen. Nach Möglichkeit sollte man daher versuchen, mit einer `return`-Anweisung am Ende der Funktion auszukommen (*single point of exit*).

2.5.3 Argumentübergabe

Die Übergabe von Argumenten an eine Funktion erfolgt "by value", d.h., dass in C und C++ vor dem Aufruf der Funktion Kopien der Argumente gemacht werden, die üblicherweise auf dem Stack abgelegt werden. Dann erfolgt der Aufruf der Funktion, die dann nur auf den Kopien dieser Variablen auf dem Stack operiert. Es können so niemals die Werte der Variablen im aufrufenden Programm verändert werden. Möchte man doch diesen Effekt erreichen, z.B. in Funktionen, die Werte aus einer Datei oder Benutzereingaben einlesen, so benutzt man Zeiger- oder Referenztypen als Argumente (2.3.4).

Die Reihenfolge der Auswertung der Argumentausdrücke ist nicht festgelegt. Programme, die davon abhängen, zeigen undefiniertes Verhalten:

```
int i;
double a[20];
// ...
// undefiniert: kann auf zwei Weisen interpretiert werden
//   f(i,a[i])  oder  f(i,a[i+1]):
f(i++, a[i]);
```

2.5.4 inline-Funktionen

Folgende Operationen finden bei einem Funktionsaufruf und beim Rückkehren aus einer Funktion statt:

1. Sichern des lokalen Programmkontextes (z.B. Variablen, die in CPU-Registern gehalten wurden, Rücksprungadresse) auf den Stack,

2. Kopieren der Funktionsargumente (Zeiger oder Objekte/Variablen) aus dem lokalen Kontext auf den Stack, um Überschreiben aus der Funktion heraus zu verhindern,

3. Anspringen der Einsprungadresse der Funktion,

4. Ausführen der Anweisungen in der Funktion und Berechnung des Rückgabewertes,

5. Ablegen/Kopieren des Rückgabewertes auf den Stack,

6. Rücksprung in das aufrufende Programm, Wiederherstellen des lokalen Kontextes.

Die Schritte 1, 5 und 6 stellen dabei Aufwand dar, der eliminiert werden kann, wenn man den Funktionsaufruf umgehen könnte. Für kurze Funktionen, die in einem Programm sehr häufig aufgerufen werden, können diese mit einem Funktionsaufruf verbundenen Abläufe einen signifikanten Anteil der Gesamtausführungszeit der Funktion darstellen. Diesen Zusatzaufwand umgeht man durch Inlining der Funktion; gewissermaßen wird dabei der Funktionskörper an der entsprechenden Stelle im ablaufenden Programm eingefügt. In C hat man an dieser Stelle häufig Makros verwendet, die aber den Nachteil unintuitiver Nebeneffekte haben.

```
#define SQR(a) ((a)*(a))
int b = 2, c;
    c = SQR(b++);  // c = ((a++) * (a++)); (undefiniertes Resultat)
                   // meist ist c hier 2*3 = 6! und b ist 4.
```

Der Schritt 2 bleibt in der Regel nötig, weil natürlich auch weiterhin keine Variablen unkontrolliert verändert werden dürfen.

Wenn man in C++ in der Deklarationsdatei (2.9) eine Funktion *definiert* und mit dem Schlüsselwort `inline` kennzeichnet, so versucht der Compiler, den Funktionsaufruf durch Inlining wegzuoptimieren.

```
inline int sqr(int a){ return a*a; }
  int  b = 2, c;
  c   = sqr(b++);  // b++ wird einmal ausgewertet
                   // daher ist c jetzt 4, b ist 3
```

Da der Nachteil des Inlining vergrößerte Programme sind, empfiehlt es sich, mit diesem Mittel sparsam umzugehen. Lange Funktionen sind in Implementierungsdateien besser aufgehoben als in Deklarationsdateien. Klassen-Elementfunktionen, die im Klassenkörper definiert werden, behandelt der Compiler automatisch als `inline`.

Zusammenfassung:
(i) Inlining ist gut für kurze Funktionen, die effizient gemacht werden müssen;
(ii) `inline`-Funktionen müssen in der Deklarationsdatei definiert werden;
(iii) Elementfunktionen, die im Klassenkörper definiert sind, sind per Default `inline`.

2.5.5 Defaultargumente

Häufig stellt sich die Frage, wie man eine Funktion "erweitern" kann, ohne die Funktionalität eines bestehenden Programmes zu verändern, das diese Funktion verwendet, oder wie man eine Funktion, die zur vollständigen Charakterisierung mehrere Argumente benötigt, einfach bedienbar gestalten kann. Diese Aufgaben können mit Defaultargumenten gelöst werden. Zum Beispiel kann eine Funktion zum Löschen eines Feldes so aussehen:

```
void clear(int size, int *array){
  for (int i=0; i < size; i++ )
     array[i] = 0;
}
```

Diese lässt sich später leicht erweitern zu einer Funktion, die das Feld mit beliebigen Werten füllt:

```
void clear(int size, int *array, int value=0){
  for (int i=0; i < size; i++ )
     array[i] = value;
}
```

Das Defaultargument muss einen konstanten und eindeutigen Wert besitzen, der zur Compilezeit feststeht. Das bedeutet, dass er in jeder Übersetzungseinheit nur einmal spezifiziert werden darf. Wenn sichergestellt ist (z.B. mit include guards in der Deklarationsdatei), dass der Compiler nur eine Deklaration der Funktion sieht, dann ist der beste Platz, um den Defaultwert festzulegen, in der Funktionsdeklaration in der Deklarationsdatei. Es ist allerdings zulässig, diesen Wert auch bei der Definition der Funktion festzulegen, wenn die Funktion nur in dieser Übersetzungseinheit verwendet wird.

2.5.6 Überladen von Funktionen

Im Gegensatz zu C sind in C++ die Argumenttypen zusätzlich zum Funktionsnamen relevant bei der Auswahl der aufzurufenden Funktion. Damit ist es möglich, eine `print`-Funktion für `int` zu schreiben, eine für `double` und eine weitere für `struct Date {...}`. Die Mehrfachverwendung des Funktionsnamens erlaubt, diesen von Typinformation freizuhalten und sich auf die Funktion zu beschränken. Als C-Beispiel können hier `sprintf` und `fprintf` dienen, in denen das `f` und `s` den Typ des ersten Argumentes angeben. Diese Hässlichkeit, die darüber hinaus noch zu Fehlern führen kann, wenn beispielsweise der `FILE *` einmal durch einen `char *` ersetzt würde, wird in C++ vermieden; der Compiler wählt automatisch die zu den Argumenten passende Funktion aus.

Man kann sich vorstellen, dass der Compiler dafür sorgt, dass gewissermaßen programmintern komplexere Namen für Funktionen verwendet werden (name mangling), die neben dem Funktionsnamen auch noch die Argumenttypen enthalten.

```
void
swap(int &a, int &b){   // internal name e.g. swap_int_int
  int tmp = a;
  a = b; b = tmp;
}

void swap(double &a, double &b){     // swap_double_double
/*... */ }

void f(){
  double a, b;
  int    i, j;
  swap(a, b);  // double version
  swap(i, j);  // int    version
}
```

Um zu verhindern, dass diese internen Namen auch für Funktionen erzeugt werden, die aus C-Bibliotheken geladen werden, müssen diese für den Compiler sichtbar als `extern "C"` deklariert werden.

```
extern "C" double sqrt(double);
```

Dieser Mechanismus erlaubt dem Compiler sowohl das Erzeugen der richtigen Namen als auch die Berücksichtigung anderer Eigenarten der Sprache, an deren Prozeduren der Linker das Programm anbinden soll. Zum Beispiel könnte bei einem Funktionsaufruf die Reihenfolge, in der Parameter auf dem Stack abgelegt werden, eine andere als die in C++ sein.

Durch Überladen lässt sich ein ähnlicher Effekt erzielen wie mit Defaultargumenten, nämlich, dass nachträglich weitere Versionen der "gleichen" Funktion mit erweiterten Argumentlisten bereitgestellt werden können. Nachteil des Überladens zu diesem Zweck ist, dass dabei der Programmcode dupliziert wird, was zu größeren Programmen führt und größere Disziplin bei der Unterhaltung des verdoppelten Codes erfordert (Bugs müssen in beiden Versionen entfernt werden).

2.5.7 Das main-(Unter)Programm

Jedes Programm muss genau eine Funktion mit dem Namen `main` enthalten. Dort beginnt die Ausführung des Programmes, nachdem die Initialisierung globaler statischer Variablen abgeschlossen wurde. Die Funktion `main` darf nicht `static` oder `inline` deklariert werden und darf nicht rekursiv verwendet werden. Der Standard fordert, dass `main` immer einen `int` an die aufrufende Umgebung zurückgeben soll und dass mindestens die beiden Prototypen

```
int main() { /* ... */ }
```

und

```
int main(int argc, char *argv[]) { /* ... */ }
```

zulässig sind. Bei der zweiten Form gibt `argc` die Anzahl der Kommandozeilenparameter und an `argv[]` enthält ihre Werte als Zeichenketten; `argv[0]` ist der Programmname.

Wenn `main` keine `return`-Anweisung enthält, so erzeugt der Compiler stillschweigend am Ende von `main` ein `return 0`. Unter UNIX benutzt man üblicherweise die Konvention, dass die `0` bei fehlerfreier Ausführung des Programmes und andere Werte sonst an die aufrufende Umgebung zurückgegeben werden.

2.5.8 Funktionen mit einer variablen Anzahl von Argumenten

Als Erbstück aus der C-Sammlung können Funktionen mit einer variablen Anzahl von Argumenten (wie z.B. die `printf()` Funktion in C) definiert werden. Die Syntax erfordert in diesem Falle die Spezifikation des Typs des ersten Argumentes, die weiteren werden durch drei Punkte angedeutet, wie etwa im Prototyp der Terminalausgabefunktion in C:

```
int printf(char *format, ...);
```

Bei der Implementierung solcher Funktionen verwendet man Makros aus `<stdarg.h>`, die das Entlanghangeln an der Argumentliste erlauben [10]. Hier sei nur ein einfaches Beispiel gezeigt, in dem das erste Argument die Anzahl der folgenden`double`-Argumente angibt:

```
#include <stdarg.h>

// summiere die durch ... angedeuteten Argumente
int sum(int n_args, ...);

int main(int argc, char *argv[] ){
  return sum(3, sum(3,1,2,3), sum(2,1,2), sum(1,1)); // = 10
}

int sum(int n_args, ...)
{
  va_list  args;         // Typ in <stdarg.h> definiert
  va_start(args,n_args); // mit erstem Argument initialisieren
  int sum(0);
  for ( int i=0; i < n_args; ++i ){
    // wir summieren jetzt ueber n_args Argumente,
    //  von denen wir hoffen, dass sie alle vom Typ int sind.
    //  Das Makro va_arg macht aus args und dem Typ des naechsten
    //  Arguments dessen Wert
    sum += va_arg(args,int);
  }
```

```
    va_end(args);          // Aufraeumen
    return sum;
  }
```

Die Verwendung der Ellipsis `...` umgeht den Argumenttyptest für die dadurch angedeuteten Variablen, so dass hier zur Laufzeit Probleme auftreten können. Vermutlich haben viele Leser, die C gelernt haben, das eine oder andere Mal Programmabstürze verursacht, weil bei der Eingabe nicht der der Formatanweisung entsprechende Typ an die `getf()`-Funktion übergeben wurde. Daher wird für C++ empfohlen, die Benutzung und Deklaration solcher Funktionen zu vermeiden. Statt dessen gibt es Mechanismen wie Überladen von Funktionen oder Defaultargumente, die eine sichere Alternative darstellen.

2.5.9 Operatoren

Operatoren liefern einen wesentlichen Anteil zur Lesbarkeit eines Programmes. Während wir ohne Weiteres einen Ausdruck der Form

```
3 * z + 5
```

lesen und interpretieren können, bereitet z.B.

```
plus(mult(3,z),5)
```

wesentlich größere Schwierigkeiten. Wir unterscheiden binäre Operatoren mit zwei Argumenten, wie die Multiplikations- und Additionsoperatoren, und unäre mit einem, wie den Adressoperator `&` oder das unäre `-`.

Die arithmetischen Operatoren sind `+, -, *, /` und `%`. Der Operator `%` (modulo) bildet den Rest der ganzzahligen Division des linken mit dem rechten Operanden. Wenn beide positiv sind, ist das Ergebnis positiv und echt kleiner als der rechte Operand. Für einen oder zwei negative Operanden ist das Vorzeichen des Ergebnisses implementierungsabhängig. Die Kombination mit einem Gleichheitszeichen (Zuweisungsoperator) erspart die Wiederholung des linken Operanden:

```
int i=3, j=6;      // Initialisierung
i    = 5;          // Zuweisung
i   += 3 * 4 + j;  // i = i + (3 * 4 + j), i wird 23
i   /= 5;          // integer Division, i ist jetzt 4
```

Die Zuweisungsanweisungen in C haben den jeweils auf die linke Seite zugewiesenen Wert, wenn sie als Teilausdrücke auftauchen.

Für Operationen zwischen zwei integralen Datentypen stehen bitweise logische und Schiebeoperatoren zur Verfügung, für die C++ zur Unterstützung beschränkter Zeichensätze auch Schlüsselwörter reserviert (siehe Ersatzsequenz in Tabelle 2.2). Auch diese Operatoren lassen sich alle mit dem Zuweisungsoperator verbinden.

Die logischen Operatoren sind `&&` (oder auch `and`) und `||` (`or`) und die Negation `!`. Die Operatoren `&&` und `||` haben die besondere Eigenschaft, dass die Auswertung eines Ausdruckes abgebrochen wird, sobald dessen Wahrheitswert feststeht. Der Ausdruck wird von links nach rechts ausgewertet (Assoziativität). Die letzte Eigenschaft teilen sie mit dem Komma-Operator `,` — letzterer wird manchmal benutzt, um komplexe `for`-Schleifen zu konstruieren. Sein Wert ist der Wert des rechtsstehenden Ausdruckes.

```
double    a[20];
unsigned int[5];

// safe, even if some ind[i] >= 20, since the last
// expression will not  be evaluated in that case
```

Symbol	Ersatzsequenz	Bedeutung
`|`	`bitor`	bitweises oder
`&`	`bitand`	bitweises und
`^`	`xor`	bitweises ausschließendes oder
`<<`		Linksschieben, n-fach
`>>`		Rechtsschieben, n-fach
`~`	`compl`	Komplement, bitweises not, unär
`|=`	`or_eq`	bitweises oder, Zuweisung nach links
`&=`	`and_eq`	bitweises und, Zuweisung
`^=`	`xor_eq`	bitweises ausschließendes oder, Zuweisung
`<<=`		Linksschieben, n-fach mit Zuweisung
`>>=`		Rechtsschieben, n-fach mit Zuweisung
`~=`	`compl_eq`	Komplement, bitweises not, unär, Zuweisung

Tabelle 2.2: Bitmanipulationsoperatoren

```
for (int i=0; i < 5 && ind[i] < 20 && a[ind[i]] >= 0 )
   sqrt(a[ind[5]]);

// sequence operator used to combine two expressions
int i, j;
for ( i=0, j=2; i < 18; ++i, ++j )
   a[i] = a[j];

// bizarre but legal use of ,
i = 5*i, 3;    // i is 3, 5*i computed, but discarded
```

Die (arithmetischen) Vergleichsoperatoren sind `==`, `!=`, `<`, `<=`, `>` und `>=`. Der Wert eines logischen Ausdruck und das Resultat einer Vergleichsoperation in C++ sind vom Typ `bool`, d.h. entweder `true` oder `false`. Dieser Typ ist frei in `int` konvertierbar und liefert dann eine 1 (true) oder 0 (false). In der anderen Richtung wird jeder Wert ungleich 0 als wahr (`true`) und der Wert 0 als falsch (`false`) interpretiert.

Der einzige ternäre Operator ist `?:`, der zur Auswahl von Unterausdrücken (gleicher, bzw. zumindest miteinander kompatibler Typen wie `int` und `char`) dient. Falls der Ausdruck links von `?` wahr ist, wird der erste, sonst der zweite Ausdruck verwendet. Im folgenden Beispiel muss aufgrund der geringen Priorität von `?:` eine Klammer gesetzt werden, um eine korrekte Auswertung der Ausgabeanweisung zu erreichen (siehe Präzedenztabelle unten).

```
int i;
// std::cout << ( i > 0 ? i : "error" );  FEHLER:  ? int : char*
std::cout << ( (i > 0) ? i : 0 ); // druckt nie eine negative Zahl
```

Der Operator `()` bewirkt Funktionsaufruf, `[]` dient als Indizierungsoperator für Zeiger und Felder. Der Operator `.` dient zur Auswahl eines Elementes aus einer Struktur und `->` dem gleichen Zweck, wenn ein Zeiger auf eine Struktur gegeben ist.

```
struct Person {
  std::string name;
  int age;
};

void f(){
```

```
    Person    p;
    Person *p_p = &p;
    p.age = 25;
    p_p->age = 25;      // aequivalent
  }
```

Die unären Operatoren `*` und `&` dienen zum Finden des Wertes einer Zeigervariablen, bzw. zum Finden der Adresse einer (beliebigen) Variablen.

Die Operatoren `++` und `--` gibt es in Postfix- und Prefix-Form, und sie bewirken, dass der Operand um eins inkrementiert bzw. dekrementiert wird. In der Postfix-Form ist der Wert der Operation der des Operanden *vor* der In- bzw. Dekrementierung, in der Prefix-Form der Wert nach der Operation.

```
int i=5, j;
j = i++;        // j ist jetzt 5 (i vor Inkrementierung )
                // i wurde inkrementiert auf 6
double a[20];
std::cout << a[++j];
std::cout << a[j++];   // druckt a[6] zweimal, j ist jetzt 7

// j++ = i;     // FEHLER: kann nicht auf eine temporaere Zahl zuweisen
// i = (j++)++; // FEHLER: kann ++ nicht auf eine temp.   Zahl zuweisen
i = ++++j;      // ok (!), j inkrementiert zweimal vor dem Gebrauch
                // i ist jetzt 9
```

Bei der Verwendung der Postfix-Operatoren werden in Ausdrücken temporäre Objekte mit dem unveränderten Wert erzeugt, deren Erzeugung der Compiler nicht immer wegoptimieren kann. Es daher empfehlenswert, wenn möglich die Prefixformen der Operatoren zu benutzen.

Operatorausdrücke werden nach den in C üblichen Regeln für Rang und ggf. bei Mehrfachvorkommen mit der folgenden Assoziativität behandelt:

Bezeichne `@` einen eingebauten Operator. C++ erlaubt es, diesen als Funktion mit Namen `operator@(...)` anzusprechen. Der Ausdruck

```
3 * z + 5
```

könnte also auch geschrieben werden als

```
operator+( operator*(3,z), 5).
```

Mittels dieser Syntax lassen sich Operatoren genau wie Funktionen überladen. Dies kann dazu dienen, benutzerdefinierte Typen wie z.B. Matrizen oder komplexe Zahlen mit intuitivem Benutzerinterface zu versehen (siehe 3.1.10).

2.6 Bezugsrahmen von Bezeichnern

Der Begriff Scope bezeichnet den Bezugs-, bzw. Sichtbarkeitsrahmen von Variablen, Funktionen, Klassennamen, Sprungzielen usw. Die Sprache C++ unterscheidet fünf verschiedene Bezugsrahmen: (i) Namensraum, (ii) Datei, (iii) lokale Blöcke, (iv) Funktion, (v) Klasse. Diese sollen im Folgenden einzeln diskutiert werden.

2.6.1 Namensräume

Namensräume sind dazu gedacht, Symbole wie Funktionsnamen oder Variablen in größeren Einheiten zusammenzufassen Durch ihre Benutzung wird vermieden, dass es bei gleichzeitiger Verwendung mehrerer großer Programmpakete zu Problemen wegen doppelt vergebener

Operator	Auswertung von
`::`	links nach rechts
`() [] -> .  X++ X--  typeid XXX_cast<TYPE>()`	links nach rechts
`! ~ ++X --X + - * & (TYPE) sizeof new delete`	rechts nach links
`.*  ->*`	links nach rechts
`*  /  %`	links nach rechts
`+  -`	links nach rechts
`>>  <<`	links nach rechts
`< <= > >=`	links nach rechts
`==  !=`	links nach rechts
`&`	links nach rechts
`^`	links nach rechts
`|`	links nach rechts
`&&`	links nach rechts
`||`	links nach rechts
`= *= /= %= += -= <<= >>= &= |=  ^=`	rechts nach links
`?:`	rechts nach links
`throw`	–
`,`	links nach rechts

Tabelle 2.3: Bindungsstärke (Priorität) und Assoziativität von Operatoren

Namen kommt. Bei der Benutzung von Teilen dieser Pakete spezifiziert man den Namen mit vorangestelltem `namespace` Namen oder importiert diese Teile mit `using` Direktiven.

```
// library.h:

namespace library {
  class string {        // neue string Implementierung
  //...
  };
}

// main.cc:
#include <iostream>
using std::cout;        // cout ist von hier an aequivalent zu
                        //  std::cout
#include <library.h>
using library::string;  // string entspricht von hier an
                        //   library::string

#include <project.h>
using namespace project;// VORSICHT: importiert alle Namen aus
                        //   Namensraum project

// ...
```

Durch `using` Direktiven kann man einzelne Elemente oder auch einen ganzen Namensraum "importieren", d.h. auf seine Elemente zugreifen ohne explizit anzugeben, in welchem Namensraum es sich befindet. Das Importieren ganzer Namensräume sollte nur in Ausnahmefällen zur Anwendung kommen, da für den Programmierer nicht mehr ersichtlich ist, welche Symbole benutzt wurden. Manchmal ist diese Option unumgänglich, wenn man

Programme übersetzen will, die mit Compiler ohne Namensraumunterstützung entwickelt wurden.

```
#include <iostream>
using namespace std;  // importiert alle Namen aus std

// ...
cout << ...           // cout kann hier ohne std:: Zusatz
                      //    benutzt werden
```

Bei der Benutzung von Namensräumen sollte man folgende Erfahrungen berücksichtigen:

1. Das, auch nur teilweise, Importieren aus Namensräumen mittels `using` in Header-Dateien sollte ganz vermieden werden. Wenn diese Header in einem anderen Projekt wieder verwendet werden sollen, so kann es dort sonst zu Namenskonflikten mit Namen aus anderen Namensräumen kommen,

2. Das Importieren ganzer Namensräume oder von Teilen von Namensräumen in *Implementierungs*dateien ist weniger kritisch, denn der Programmierer der Implementierungsdatei hat die volle Kontrolle darüber, welche Symbole durch Headerdateien bekannt gemacht werden,

3. Man kann relativ lange Namen für Namensräume verwenden und diese den Benutzer mit so genannten aliases zu verwendbarer Länge abkürzen lassen:

   ```
   namespace p3t = ICA1_Stuttgart_U_physics_toolbox;
   ICA1_Stuttgart_U_physics_toolbox::initialize();  // ok
   p3t::initialize();                               // aequivalent
   ```

   Man beachte dabei, dass innerhalb der Namensräume die Variablen ohne Qualifikation zur Verfügung stehen, der lange Name also keine Behinderung darstellt. Eine Kollision eines solchen langen Namens mit einem Namen in einem anderen Paket ist extrem unwahrscheinlich.

4. genügend viele, logisch zusammengehörige Komponenten (Klassen, Funktionen) sollten in einen Namensraum verpackt werden.

Programmelemente außerhalb aller `namespace`-Blöcke befinden sich im *globalen* Namensraum. Dort befinden sich daher auch die Deklarationen und Definitionen von Symbolen der C-Bibliothek.

2.6.2 Dateibezugsrahmen

Ein Projekt besteht im Normalfall aus mehreren Deklarationsdateien (oder "Header"-Dateien) die per Konvention meist die Endung `.h` besitzen und aus Implementierungsdateien, die meist durch `.cc` oder `.cpp` gekennzeichnet sind. Jede Implementierungsdatei schließt eine beliebige Anzahl von Deklarationsdateien ein.

Für Variablen, die außerhalb eines Klassen-, Funktionskörpers und Namensraumes in einer Datei deklariert sind, wird bei Programmbeginn Speicherplatz bereitgestellt. Ihre Namen müssen im gesamten Programm eindeutig sein, um Konflikte beim Zusammenfügen der Programmteile zu verhindern. Bei ungewollten Namenskonflikten besteht die Möglichkeit, die entsprechende Funktion oder Variable `static` zu deklarieren. Sie ist dann nur innerhalb der jeweiligen Datei sichtbar.

Ansonsten ist es möglich, mit `extern` explizit einen "gewollten" Namenskonflikt zu kennzeichnen. Nur an einer Stelle im gesamten Programm darf bei der Deklaration der Variablen dann die `extern`-Bezeichnung fehlen. Dort wird dann tatsächlich Speicherplatz für

die Variable reserviert. Mit anderen Worten, um eine Variable für ein ganzes Projekt global zu erklären, deklariert man sie außerhalb aller Funktionen in einer Implementierungsdatei. Ihre Existenz wird in anderen Übersetzungseinheiten dann durch eine wiederholte `extern`-Deklaration, am besten in einer Deklarationsdatei, bekannt gemacht.

```
//  header.h
extern int Error_id;   // globale Fehlervariable bekanntmachen
void  f();             // kann Error_id setzen


//  header.cc
#include "header.h"
int Error_id;          // hier wird Platz wirklich reserviert

void f(){
  // Error_id = ...
}

//  main.cc:
#include "header.h"   // macht Error_Id bekannt

int main(){
  f();
  if ( Error_id == 0 ) //  alles klar
     /* no error */;
  else
     /* process error */;
}
```

Im Falle, dass der verwendete Compiler Namensräume unterstützt, sollte die Einschränkung auf Dateibezugsrahmen jedoch möglichst nicht mittels des Schlüsselwortes `static` erfolgen, sondern die Variable in einen Namensraum zusammen mit den Funktionen eingebracht werden, die auf sie zugreifen. Notfalls kann sie mittels scope-Operator oder using-Anweisungen dann auch in andere Namensräume importiert werden. Um sicherzugehen, dass die Variable *niemals* außerhalb der Übersetzungseinheit (Implementierungsdatei plus eingeschlossene Deklarationsdateien) sichtbar werden kann, kann der unbenannte Namensraum verwendet werden.

```
// a.cc:
namespace vanilla {
  int yoghurt;
}

namespace {
  int feigenblatt;
}

// b.cc:
namespace food {
  vanilla::yoghurt = 7; // ok
  // ::feigenblatt = 7; // FEHLER
  // feigenblatt = 7;   // FEHLER
}
```

2.6.3 Lokale Bezugsrahmen

Hier gilt, dass lokale Namen globale verdecken. Jeder Block `{}` führt einen lokalen Bezugsrahmen ein, in dem lokale Variablen definiert werden können. Will man explizit auf globale Namen zugreifen, so verwendet man den Scope-Operator `::`.

```
static int i;  // i global in der Uebersetzungseinheit

int f(int i){  // Argument i verdeckt globales i
  // ...
  {
    int i;     // lokales i verdeckt Argument i
    int i_new = ::i; // greift auf globales i zu
               // es gibt keine Moeglichkeit, das Argument i anzusprechen
  }
  // ...
}
```

Lokale Bezugsrahmen können im Zusammenhang mit `goto`-Anweisungen dem Compiler u.U. gewisse Optimierungen erlauben:

```
goto label;
{
  // hier eingefuehrte Variablen sind lokal
  // dieser Block hilft evtl. dem Compiler
}
label: // ...
```

Nützlicher für den Programmierer ist die folgende Anwendung, die dem Compiler korrekte Resourcenallokation und –freigabe wie im Falle von Speicherverwaltung oder Benutzung einer Datei aufbürdet.

```
double variable;
{
  std::ifstream  in("data"); // Datei "data" oeffnen
  in >> variable;            // Daten lesen
}                            // Blockende schliesst Datei aufgrund des
                             //   Destruktor-Aufrufes von 'in' implizit
```

2.6.4 Klassenbezugsrahmen

Funktionen und Variablen im Klassenbezugsrahmen werden durch `.name`, `->name`, die für nicht-statische Elemente erforderlich sind oder auch durch `Klassenname::name` für statische Variablen (oder statische Elementfunktionen) angesprochen. Statische Variablen existieren unabhängig von der Instanzbildung nur einmal für die ganze Klasse, d.h. alle Objekte/Instanzen einer Klasse teilen sich diese Variable.

Da Klassen neben Element- und statischen Funktionen und Variablen auch weitere Klassen als "innere" Klassen enthalten können, können sie manchmal funktional an die Stelle kleiner Namensräume treten.

2.7 Ein- und Ausgabe

2.7.1 Allgemeines

Die Ein- und Ausgabe von C++ wurde gegenüber C vollständig überarbeitet. Dies war unter anderem deshalb notwendig, weil in C aufgrund des Fehlens von Funktionsüberla-

dung die Ein- und Ausgabefunktionen variable Argumentenzahlen haben und daher eine Typüberprüfung bei der Programmübersetzung nicht stattfindet. Insbesondere bei der Eingabe musste dabei peinlich genau auf die Kohärenz von Formatanweisung und Argumenttyp (Zeiger) geachtet werden, was häufig zu Fehlern Anlass gab. Programmierern, die Funktionen wie `printf()` aus C gewohnt sind, werden die C++ Ein-/Ausgabe anfänglich recht umständlich finden. Mit zunehmender Übung und in Hinblick auf die erreichte Typsicherheit relativiert sich dieser Aufwand jedoch.

Die beiden wichtigsten Klassen, mit denen man bei der Ein-/Ausgabe auf das Terminal zu tun hat, sind `istream` und `ostream`. Beide sind im header `iostream` deklariert und sind die Basisklassen der spezialisierteren `(i/o)fstream` und `(i/o)stringstream` Klassen, die formatierte Ausgabe in Dateien und `string`s leisten.

Auf noch tieferer Ebene besitzt jedes `(i/o)stream` Objekt einen Zeichenpuffer zum Lesen bzw. Schreiben (`streambuf`), in den die Ausgabe der formatierten Daten erfolgt. Das `streambuf` Objekt abstrahiert sozusagen das physikalische Medium, über das die Ein- und Ausgabe abgewickelt wird, während die von `i/ostream` abgeleiteten Klassen die Formatierung besorgen. Zum binären, unformatierten Schreiben lassen sich Variableninhalte auch direkt (und sehr schnell) in diesen Puffer schreiben.

2.7.2 Wie funktioniert die Ein- oder Ausgabe in C++

Der Programmierer schreibt etwa:

```
#include <iostream>

int main(){
  int gehalt = 65000;
  std::cout << "Gehalt: "  << gehalt;
}
```

Die Programmzeile in `main` wird auf Grund der Assoziativität vom Compiler interpretiert als:

```
 (std::cout << "Gehalt: ") << gehalt;
```

Der Compiler sucht jetzt die verfügbaren Versionen von `operator<<` auf. Das Objekt `std::cout` ist vom Typ `std::ostream` (bzw. ist eine Referenz auf ein Objekt solchen Typs) und daher wird für die Ausgabe der Zeichenkette `"Gehalt:"` der `std::ostream & operator<<(std::ostream &, const char *)` aufgerufen. Der Rückgabetyp dieses Operators ist wieder ein `std::ostream` Objekt. Tatsächlich wird das gleiche `std::ostream`-Objekt, das als Argument übergeben wurde, in die aufrufende Umgebung zurückgegeben. Für die weitere Ausgabe verwendet der Compiler daher `operator<<(std::ostream &, int)` findet, diesmal mit `gehalt` als zweitem Argument. Aneinandergereihte Ausgabe oder Eingabeanweisungen in C++ werden so als sukzessive Funktionsaufrufe behandelt, bei denen Typsicherheit gewährleistet ist.

Wie das folgende Beispiel zeigt, ist es erforderlich, die Argumente zu klammern, wenn die auszugebenden Größen zusammengesetzte Ausdrücke sind, die Operatoren geringer Priorität als `<<` enthalten:

```
#include <iostream>

int main(){
  int number = 5;
  // Vorzeichen auch ausgeben, wenn number > 0 ist:
  std::cout << number >= 0 ? '+' : ' ' << number;   // LOGISCHER FEHLER
  std::cout << (number >= 0 ? '+' : ' ') << number; // ok
}
```

Dieses Programm kann übersetzt werden und produziert die Ausgabe `5+5` ohne führendes Leerzeichen. Es sei hier nur gesagt, dass nur die zweite Ausgabeanweisung das gewünschte Verhalten zeigt.[4]

2.7.3 Ein- und Ausgabe selbstdefinierter Typen

Für die bequeme Ein- und Ausgabe eigener Typen definiert man den entsprechenden Ein- und Ausgabeoperator in der folgenden Weise.

```
#include <iostream>

struct Kugel{
   double r; // Radius
   double x; // Mittelpunkt
};

inline
std::ostream &
operator<<(std::ostream & out, const Kugel& rhs){
   out << rhs.r << " " << rhs.x;
   return out;
}

std::istream &
operator>>(std::istream & in, Kugel& rhs){
   in >> rhs.r >> rhs.x;
   return in;
}
```

Mit diesen Definitionen verhält sich die Ein- und Ausgabe für diesen Datentyp `Kugel` im Programm wie die eines eingebauten Typs.[5] Damit der Compiler die Ein- und Ausgabeoperatoren findet, sollten sie im gleichen Namensraum wie `Kugel` definiert sein. Der Compiler sucht Funktionen oder Operatoren immer im globalen Namensraum sowie in den Namensräumen der Argumente (König-Lookup). Damit müssen Funktionen oder Operatoren nicht mit den Namensraumbezeichnern gekennzeichnet werden.

2.7.4 Wichtige Elementfunktionen der iostreams

Der folgende Abschnitt erfordert Wissen, das erst im nächsten Kapitel systematisch erarbeitet wird. Er befindet sich hier zu Referenzzwecken und kann beim ersten Lesen übersprungen werden.

2.7.5 Formatierung

Die Formatierung in den `i/ostream`-Klassen wird mittels Kontrollflags in deren gemeinsamer Basisklasse `ios_base` kontrolliert. Diese Flags und ihre Bedeutung sind im Folgenden aufgelistet. Einige von ihnen sind nicht unabhängig voneinander, sondern in Gruppen zusammengefasst:

[4] Mit dem Wissen, dass `i/ostream` eine implizite Umwandlung nach `void *` unterstützt, die den Zustand des Streams widerspiegelt (der Zeiger 0 steht für einen Fehler) überlege man sich, warum die erste Ausgabezeile syntaktisch korrekt ist und warum in der Ausgabe kein Leerzeichen und kein `+` auftaucht.

[5] Ein (kleines) Problem ergibt sich, falls private Datenelemente von Klassen eingelesen oder ausgegeben werden sollen. In diesem Fall verweisen wir auf Abschnitt 3.1.8.

Flag	Gruppe	Funktion
left	adjustfield	links ausrichten im Ausgabefeld
right	adjustfield	rechts ausrichten
internal	adjustfield	+/- links, Zahl rechts
dec	basefield	Dezimaldarstellung
hex	basefield	Hexadezimaldarstellung
oct	basefield	Oktaldarstellung
fixed	floatfield	Festkommazahl
scientific	floatfield	wissenschaftliche (Exponential-) Darstellung
boolalpha		Textrepräsentation von `true` und `false`
showbase		zeigt Präfix vor hex- und oct-Zahlen
showpos		'+' vor Zahl, wenn positiv
showpoint		zeigt den Dezimalpunkt immer
uppercase		E, X, A-F statt e, x, a-f
skipws		führende Leerzeichen oder TABs überlesen
unitbuf		Stream nach jeder Operation mit Medium synchronisieren

Die Klasse `std::ios_base` definiert Elementfunktionen `setf` und `unsetf` bzw. "Manipulatoren" `setiosflags` und `resetiosflags`, mit deren Hilfe sich diese Kontrollflags einzeln setzen oder rücksetzen lassen, ggf. unter Angabe der Bitgruppe als symbolischer Konstanten `basefield`, `floatfield`, `basefield`, die für eine Bitmaske steht. Die Funktionen, die ein zusätzliches Bitmaskenargument zulassen, beschränken das erste Argument auf die so gekennzeichnete Bitgruppe. Der Manipulator-Trick erlaubt auch, das Bit zu setzen, indem einfach dessen Name auf dem Stream ausgegeben wird. Um es zurückzusetzen, wird ein `no` vorangestellt.

Die überladenen Funktionen `fmtflags std::ios::flags()` und `fmtflags std::ios::flags(fmtflags)` erlauben das Auslesen bzw. das Überschreiben des gesamten Statuswortes.

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

int main(){
  double  d = 123.45;
  std::cout.setf(std::ios_base::showpos);
  std::cout << d << '\n';  // Ausgabe: +123.45
  std::cout.setf(std::ios_base::scientific);
  // alternativ und sicherer:
  std::cout.setf(std::ios_base::scientific,std::ios_base::floatfield);
  std::cout << d << '\n';  // Ausgabe: +1.234500e+02
  // Ruecksetzen der Flags durch Manipulatoren
  std::cout << std::noshowpos
            << std::resetiosflags(std::ios_base::floatfield);
  std::cout << d << '\n';  // Ausgabe: 123.45
}
```

Manipulatoren sind entweder in der Klasse `std::ios_base` definierte Zeiger auf eine Funktion, die auf einem `i/ostream` Objekt operiert und das dem Namen entsprechende Flag setzt oder rücksetzt (`std::noshowpos`) oder es handelt sich wie im Fall von `std::resetiosflags` um spezielle Funktionen im Namensraum `std`, die zusätzliche Argumente akzeptieren.

Für solche Funktionszeiger (oder die von den Funktionen gelieferten temporären Objekte) gibt es eine entsprechend überladene Version des `operator<<`, deren Aufgabe es ist,

die durch den Zeiger gegebene Funktion aufzurufen bzw. die dem Objekt zugeordneten Operationen selbst durchzuführen.

Bei Verwendung von Manipulatoren, die ein Argument erwarten, muss wie im obigen Beispiel gezeigt die Datei `iomanip` eingeschlossen werden. Die Manipulatoren ohne Argument sind bereits durch Einschließen von `iostream` deklariert.[6]

Neben den erwähnten Statusflags gibt es weitere Datenelemente zur Kontrolle der Ein- und Ausgabe. Diese kontrollieren die Feldweite die zur Ausgabe der Zahl genutzt wird, die Zahl der relevanten Stellen, oder Füllzeichen. Ihre Werte werden durch Elementfunktionen oder durch Manipulatoren gesetzt:

Manipulator	Wirkung
`setw(int width)`	setzt die minimale Feldweite für die direkt folgende Ausgabe, Defaultwert ist 0
`setprecision(int prec)`	setzt die Zahl der Stellen für Festkomma- oder die Zahl signifikanter Stellen bei Gleitkommadarstellung, ansonsten die maximale Anzahl der Stellen, Defaultwert ist 6.
`setfill(char)`	Zeichen, um ansonsten leer bleibende Positionen im direkt folgenden Ausgabefeld aufzufüllen.
`setbase(int base)`	mit base= 8, 10, 16 setzt die Basis der Ganzzahldarstellung wie mit den entsprechenden Flags im `basefield`
`(re)setiosflags`	wie oben beschrieben

Sie werden verwendet wie wir das bereits im vorigen Beispiel gesehen haben:

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

int main(){
  std::cout << std::setfill('*') << std::setw(11)
            << 1234.56 << '\n';  // Ausgabe: ****1234.56
  std::cout << std::setprecision(2)
            << 1234.56 << '\n';  // Ausgabe: 1.2e+03
  std::cout << std::setprecision(3) << std::fixed
            << 1234.56 << '\n';  // Ausgabe: 1234.560
  std::cout << std::setfill('*') << std::setw(11)
            << 1234.56 << '\n';  // Ausgabe: ***1234.560
}
```

An diesem Beispiel wird deutlich, dass `setw` und `setfill` nur für die direkt darauf folgende Ausgabe Bedeutung haben. Im Beispiel zieht weiterhin `setprecision` die Verwendung des Exponentialformats nach sich; die Genauigkeit bleibt aber ansonsten für die folgenden Ausgaben erhalten.[7]

[6] Fortgeschrittene mögen sich den Inhalt der Datei `iomanip` auf ihrem System ansehen, um die Funktionsweise von Manipulatoren zu verstehen.

[7] Leider legt der Sprachstandard nicht eindeutig fest, welche Nebeneffekte das Setzen der `precision` haben soll. Das Umschalten auf Exponentialformat in der zweiten Ausgabe hat einen der Autoren ziemlich überrascht. Tatsächlich erhielten wir für die Gesamtheit der Ausgaben mit drei verschiedenen Compilern auch drei verschiedene Ergebnisse. Die angegebenen Ausgaben sind Resultat des Intel icc, Version 6.0.1.

2.7.6 Spezielle Ein- und Ausgabefunktionen der `i/ostream` Klassen

Die Klassen `istream` für Eingabe und `ostream` für Ausgabe dienen als Basisklassen, deren Eigenschaften an alle Ein- und Ausgabeklassen weitergegeben werden. Dazu gehören die oben eingeführte Status- und Formatiermaschinerie der Klasse `std::ios_base` und die Bereitstellung grundsätzlicher Ausgabefunktionalität durch Umsetzung von Funktionen wie den im Folgenden aufgelisteten in Aufrufe der passenden Elementfunktionen ihres `streambuf` Elementes. Hier sind auch die überladenen Versionen von `operator<<` und `operator>>` für Standardtypen definiert. Das Folgende ist eine nicht vollständige Liste von Funktionen, um spezielle Anforderungen an Ein- und Ausgabe zu bewerkstelligen.

`ostream & ostream::operator<<(ostream &,double)`
schreibt eine doppelt genaue Zahl in formatierter (ASCII) Darstellung in den Puffer. Analog für `int`, `bool`, etc. Der Puffer schreibt dabei ggf. auf das repräsentierte physikalische Medium.

`istream & ostream::operator>>(istream &,double &)`
liest eine doppelt genaue Zahl in formatierter (ASCII) Darstellung aus dem Puffer. Analog für `int`, `bool`, etc. Führende Leerzeichen werden dabei ggf. je nach Wert von `std::ios_base::skipws` ignoriert. Der Puffer liest dazu aus dem repräsentierten physikalische Medium.

`ostream & ostream::put(char)`
gibt ein einzelnes Zeichen aus.

`int istream::get()`
entnimmt ein Zeichen vom Eingabepuffer und liefert es an den Aufrufer.

`istream & istream::getc(char &c)`
schreibt das nächste Zeichen im Eingabepuffer nach `c`

`istream & operator>>(istream &,char &c)`
liest ein Zeichen `c` ein und ignoriert dabei je nach Wert von `skipws` die *whitespace* Zeichen SPACE, TAB, EOL.

`istream & istream::get(char * c, int anzahl=1, char terminator='\n')`
liest maximal 'anzahl' Zeichen vom stream ein oder endet, wenn das Terminierungszeichen eingelesen wurde. Dieses Zeichen wird im stream belassen.

`istream  istream::getline (char * c, int anzahl, char terminator)`
dito, jedoch wird das Terminierungszeichen gelöscht.

`istream  istream::putback(char c)`
schreibt das vorher mit `get` aus dem Eingabepuffer gelesene Zeichen 'c' wieder in den Stream zurück. Diese Funktionalität ist nur für das letzte Zeichen garantiert, so dass sich ganze Zeichenketten nicht zurückschreiben lassen.

`int istream::peek()`
liest das nächste Zeichen vom Eingabepuffer, belässt es aber darin

`istream & istream::read(char* buf,int anz)`
kopiert höchstens *anz* Zeichen direkt nach `buf`.

`int istream::gcount()`
Anzahl der zuletzt unformatiert aus dem Puffer extrahierten Zeichen.

`ostream & ostream::write(const char* buf,int anz)`
für binäres Schreiben, schreibt *anz* Zeichen in den Ausgabepuffer.

2.7.7 Schreiben und Lesen von Dateien

Für Dateien gibt es die beiden Klassen `std::ofstream` und `std::ifstream` zum Lesen und Schreiben. Alle oben beschriebenen Elementfunktionen von `std::ostream` und

`std::istream` stehen durch den Vererbungsmechanismus zur Verfügung. Dazu kommen noch einige dateispezifische Operationen.

```
#include <fstream>

int main(){
  std::ifstream ins ("filename"); // oeffnet Datei 'filename' zum Lesen
  std::ofstream outa("filename"); // oeffnet Datei 'filename' zum Schreiben
      // Die Dateien werden durch  ins.close() / outs.close()
      // geschlossen oder spaetestens, wenn der Destruktor bei
      // Verlassen des Blocks aufgerufen wird.
  outa << 'a';
  outa.close();
  std::ofstream outb("filename",std::ios_base::out | std::ios_base::app);
  outb << 'b';                   // 'filename' enthaelt jetzt 'ab'
}
```

Wie in C lassen sich Dateien in verschiedenen Modi öffnen, die in C++ als Bitmuster in einem zweiten Argument des Konstruktors übergeben werden können. Diese sind im Bitfeld `openmode` der Klasse `ios_base` zusammengefasst:

`app`	Dateiende vor jedem Schreibversuch aufsuchen
`ate`	Dateiende beim Öffnen der Datei aufsuchen
`trunc`	Datei leeren beim Öffnen, implizit bei `ofstream`
`binary`	Binärmodus (kein Unterschied zu Text bei UNIX)
`in`	Öffnen zum Lesen, implizit bei `ifstream`
`out`	Öffnen zum Schreiben, implizit bei `ofstream`

Nicht alle dieser Modi lassen sich beliebig miteinander kombinieren. Garantiert ist jedoch, das jede Kombination erlaubt ist, die einem aus C bekannten Modus äquivalent ist. Man beachte, dass `app` das Dateiende vor jedem Schreibversuch aufsucht und damit u. U. sehr langsam ist. Der Modus `ate` ist ggfs. die bessere Wahl.

Fehlerzustände bei Ein- und Ausgabe

Spätestens bei der Bearbeitung von Dateien benötigt man Informationen darüber, ob man am Ende der Datei angelangt ist, ob das Öffnen der Datei erfolgreich war, aber letztlich auch, ob bei der formatierten Eingabe ein Fehler aufgetreten ist — wenn man etwa versucht hat, die Zeichenkette `"siebzehn"` als `double` einzulesen. Dafür stellen die Streams Statusinformationen zur Verfügung. Die Implementierung dieser Funktionen erfolgt bereits in `std::ios_base`, so dass für Ein- und Ausgabe die gleichen Funktionen vorhanden sind.

```
bool good()  // == true, wenn keinerlei Fehler aufgetreten sind
bool eof()   // == true, wenn das Ende der Datei/des Streams erreicht ist
bool bad()   // == true, wenn ein Hardwarefehler aufgetreten ist
bool fail()  // == true, wenn ein logischer Fehler aufgetreten ist
             //      (Eingabe passt nicht zum Datentyp) oder bad()==true

ios_base::iostate rdstate()   // liest Statuswort
bool clear(ios_base::iostate state = goodbit ) // setzt state "manuell"
bool clear()                  // Effekt:  good() == true
bool clear(ios_base::failbit) // setzt fail Status, analog badbit, eofbit.
```

Lesen bis Ende der Datei

Beim typischen Fall "Lesen bis Ende der Datei" sind `eofbit`, `badbit`, und `failbit` beteiligt. Angenommen, es treten keine logischen Fehler bei der Eingabe auf, dann wird die letzte im Stream befindliche Information noch gelesen, ohne dass eines dieser Bits gesetzt wird. Erst der darauf folgende Leseversuch schlägt fehl und setzt sowohl das `eofbit` als auch das `failbit`. Ein weiterer Versuch wird auch das `badbit` setzen.

`eof()` tritt also erst *nach* der letzten erfolgreichen Eingabe auf. Im folgenden Beispiel wird also der letzte Leseversuch einen `fail()` Zustand des Streams erzeugen und `n` enthält eine ungültige Zahl.

```
#include <iostream>

int main(){
  int n;
  while( !std::cin.eof() ){
    std::cin >> n;  // PROBLEM beim letzten Leseversuch
    std::cout << n; // undefiniert fuer letzten Durchlauf
  }
}
```

Das folgende Beispiel hingegen gibt nur erfolgreich gelesene Zahlen auch wieder aus.

```
#include <iostream>

int main(){
  int n;
  do {
    std::cin >> n;
    if ( std::cin.good() )
      std::cout << n; // Stream-Ende abgefangen
  } while ( std::cin.good() );
}
```

Die Streamobjekte definieren eine automatische Typumwandlung in den Typ `void *`. Diese liefert im wesentlichen einen Null-Zeiger zurück, wenn der Stream-Zustand `! good()` ist, und einen Wert ungleich Null sonst. Diese Eigenschaft lässt sich häufig in Schleifenbedingungen verwenden, um die Eingabe zu kontrollieren. Unser obiges Beispiel lässt sich damit so verkürzen:

```
#include <iostream>

int main(){
  int n;
  while( std::cin >> n )
    std::cout << n;
}
```

Hier wird in der Schleifenbedingung zunächst die Eingabe versucht und dann der Zustand des Eingabeobjektes auf `good()` getestet. Erst dann wird der Schleifenkörper durchlaufen, der also nur erfolgreich eingelesene Zahlen ausgibt.

2.8 Dynamische Speicherverwaltung

Zum Anfordern von Speicher dienen in C++ die Operatoren `new` und `new[]` für einzelne Datenelemente bzw. Felder. Das inverse Freigeben des Speichers erfolgt mit `delete` bzw.

`delete []`. Ein Anwendungsbeispiel für die Feldversion von `new []` haben wir bereits in Abschnitt 2.3.4 gesehen. Um zu verhindern, dass ein Programm während seiner Laufzeit immer mehr Speicher anfordert, der nicht wieder freigegeben wird *(memory leak)*, müssen diese Operatoren immer in `new/delete`-Paaren angewendet werden.

```
int    *p_i  = new int;
Person *p_p  = new Person("B.Stroustrup",45);
int n = 40;
int *pa_i = new int[n];
//...
delete p_i;
delete p_p;
delete[] pa_i;
```

Die Anforderung geschieht durch Angabe des Typs oder des Klassennamens nach `new`. In diesem Fall wird ein Datenelement der Klasse oder des Typs mit Hilfe des zugehörigen Defaultkonstruktors im Speicher angelegt, und der Operator gibt einen Zeiger auf dieses Element zurück. Um einen anderen als den Defaultkonstruktor zu verwenden, benutzt man nach dem `new` den entsprechenden Konstruktoraufruf, wie oben für die Klasse Person gezeigt.

Für Felder wird `new[]` verwendet. Hier wird Speicherplatz für die Anzahl von Speicherelementen angefordert, die innerhalb der eckigen Klammern angegeben ist. Für jedes Element wird der Defaultkonstruktor aufgerufen, der existieren muss. Der Operator `new[]` gibt einen Zeiger auf das erste Datenelement zurück. Zum Zugriff auf die Elemente jenseits des ersten verwendet man pointer-Arithmetik oder Indizierung mit `[]`.

```
Person  *p = new Person[100];
p[0]       = Person("Matthias",30);
p[1]       = Person("Stefan",32);
// ...
```

`new` kann eine Ausnahmebehandlung (`bad_alloc`) auslösen, falls die Speicheranforderung fehl schlägt. Sollte diese unbehandelt bleiben, ruft das Programm letztlich eine Abbruchroutine (`terminate()`) auf, die ein Speicherabbild schreibt und den Programmlauf abbricht.

2.8.1 Überladen des `operator new`

Für fortgeschrittene Anwendungen erlaubt C++ auf Klassenbasis das Überladen des so genannten `operator new`. Um zu verstehen, wie dieser Mechanismus funktioniert, müssen wir beachten, dass dieser ein interner Bestandteil des `new operator` ist (man beachte den Unterschied in der Reihenfolge der Worte!), der in den obigen Beispielen direkt vom Programm aufgerufen wird. Der `new operator` ruft zunächst intern den `operator new` auf, um uninitialisierten Speicher anzufordern. Dieser spielt also etwa die Rolle eines Aufrufs von `malloc` oder `calloc` in üblichen C-Programmen. Nach erfolgreicher Rückkehr initialisiert dann der `new operator` durch Aufruf des Klassenkonstruktors das (oder die) angeforderten Speicherelement(e). Durch Überladen von `operator new` kann man also die Speicherallozierung selbst erledigen, was von Vorteil sein kann, wenn beispielsweise bekannt ist, dass eine Klasse immer Objekte der gleichen Größe anfordert.

Als Gegenstück zu `operator new` muss dann auch der `operator delete` überladen werden, um den Speicher wieder konsistent freizugeben.

2.9 Programmorganisation: Implementierung und Deklaration

Es ist schwierig, ein großes Programm in einer einzigen Datei zu verwalten. Dies führt zu langen Übersetzungszeiten, unübersichtlicher Anordnung der verschiedenen Teile und zu großen Problemen, die Datei konsistent zu halten, wenn mehrere Personen an einem Projekt arbeiten.

Man trennt Programme daher meist in mehrere Dateien auf, die getrennt voneinander compiliert werden. Übersetzbare (Implementierungs-) C++-Dateien erhalten meist das Suffix `.cc` aber je nach Plattform oder Compiler sind auch andere Endungen üblich: `.cpp`, `.C`). Dateien, die nur Deklarationen enthalten, heißen auch Header-Dateien und dienen dazu, Funktionen und Daten aus einem Programmteil in anderen bekannt zumachen. In aller Regel (eine Ausnahme könnte z.B. `main.cc` sein) existiert zu jeder Implementierungsdatei mindestens eine Deklarationsdatei. Zwei Deklarationsdateien können gerechtfertigt sein, wenn man sicherstellen will, dass nur Funktionen nach außen hin bekannt gemacht werden, die auch von anderen Programmteilen benutzt werden sollen. Implementierungsspezifische Funktionen können dann in einer weiteren Deklarationsdatei aufgelistet sein.

Der Sprachstandard stellt einige Forderungen, die das Übersetzen in mehreren Einheiten erlauben und nötig sind, um die Konsistenz des Gesamtprogrammes sicherzustellen. So darf jede Definition (Funktionskörper, Reservierung von Speicherplatz für eine Variable) nur genau einmal im Gesamtprogramm erfolgen. Die Definition von `inline`-Funktionen muss für alle Übersetzungseinheiten identisch sein, Deklarationen dürfen nur einmal innerhalb der gleichen Übersetzungseinheit auftreten.

2.9.1 Deklarationsdateien

sollten daher enthalten:

1. include-guards, die das mehrfache Einschließen von Header-Dateien und damit die Mehrfachdeklaration von Funktionen, Klassen oder Variablen verhindern

   ```
   #ifndef HEADER_H
   #define HEADER_H
     // Deklarationen
     // ...
   #endif
   ```

2. Deklarationen von Funktionen, die modulübergreifend verwendet werden sollen. Zur Deklaration von Funktionen, die nur im jeweiligen Programmteil Verwendung finden sollen, können weitere Header-Dateien verwendet werden;

3. Definition von `inline` Funktionen

4. Deklaration von Klassen und deren Elementfunktionen

5. `extern`-Deklarationen globaler Variablen, Deklaration statischer Klassenvariablen.

6. Deklaration *und Definition* von `template` - Funktionen und Klassen.

2.9.2 Implementierungsdateien

sollten enthalten:

1. Implementierung von Elementfunktionen von Nicht-Template-Klassen,

2. Implementierung regulärer Funktionen,
3. Definition statischer Variablen in Datei-, Klassen- und globalem Bezugsrahmen.

2.10 Übungen

Ziel dieser ersten Übung ist das Kennenlernen grundlegender Sprachelemente und des Programmentwicklungszyklus.

2.10.1 Das Hello-World-Programm

- Schreiben Sie das C++ Hello-World-Programm, kompilieren Sie es und lassen Sie es ausführen (Abschnitt 2.1).
- Überprüfen Sie Portabilitätsfragen wie z.B.:
  - Unterstützt der Compiler die (veraltete) `iostream.h` Implementierungsdatei?
  - Darf man das `std::`-Präfix weglassen (wenn man z.B. die `iostream.h` Deklarationsdatei verwendet)? Welche Fehlermeldung erhält man anderenfalls vom Compiler?
  - Was passiert wenn der Rückgabetyp der Funktion `main` nicht zu `int`, sondern z.B. zu `void` gewählt wird?
- Modifizieren Sie das Programm so, dass es alle Kommandozeilenelemente in der Reihenfolge ihres Auftretens auf dem Terminal ausgibt.

2.10.2 Ein- und Ausgabe

- Schreiben Sie in C++ ein Programm, das eine Reihe von Zahlen vom Terminal einliest, das Einlesen bei Eingabe der Zahl 999.9 abbricht und vor seiner Beendigung noch die Summe der bisher gelesen Zahlen ausgibt. Wie verhält sich Ihr Programm bei fehlerhafter Eingabe, etwa eines Buchstabens statt einer Zahl?
- Die Klasse `std::stringstream` erlaubt die Verwendung einer Zeichenkette als Basis eines Einlese- oder Ausgabevorgangs mit den `iostream`-Funktionen. Versuchen Sie grob zu verstehen, warum das folgende Programm eine Approximation an π auf dem Bildschirm ausgibt und benutzen Sie die verwendeten Mechanismen, um die Eingaberoutine Ihres Programmes zu verbessern:

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

int main(){
  std::string       pi_string = "3.1415";
  std::stringstream pi_stream(pi_string);
  double pi;
  pi_stream >> pi;
  std::cout << pi;
}
```

- Lagern Sie geeignete Funktionalität des Programmes in ein Unterprogramm aus, das ein `std::istream &`-Argument nimmt und die Summe der gelesenen Zahlen zurückgibt. Modifizieren Sie das Programm so, dass die Eingabe sowohl vom Terminal (ohne Kommandozeilenargumente) als auch aus einer Datei (1 Kommandozeilenargument) erfolgen kann .

Kapitel 3

Klassen

3.1 Deklaration von Klassen

3.1.1 Datenelemente

Das Wort Klasse bezeichnet eine spezielle Sammlung von eng zusammenhängenden Funktionen und Daten, durch deren Zusammenwirken ein neuer, benutzerdefinierter Datentyp entsteht. Das Zusammenwirken hat viele Aspekte, wie etwa der priviligierte Zugang der Funktionen zum gemeinsamen Datenbestand, die Unterstützung von Funktionen wie gemeinsames, transparentes Kopieren oder Zuweisen der Datenelemente der Klasse durch den Compiler, die nahtlose Einbindung in das bestehende Typsystem durch das Überladen von Operatoren oder den Schutz der Datenelemente gegen ungewollte Manipulation durch Funktionen, die nicht zur Klasse gehören. In einer wohl durchdachten Klasse ist es zu jedem Zeitpunkt möglich, einen konsistenten Zustand der Daten zu erhalten, der nur durch die Wirkung der Klassenfunktionen verändert und in einen neuen konsistenten Zustand überführt werden kann.

Wir werden zunächst in Beispielen die Ziele und die technische Verwirklichung dieser Konzepte untersuchen um dann später abstraktere Methoden der Klassenkonzeption kennenzulernen.

Die Deklaration einer Klasse wird durch die Schlüsselwörter `struct` oder `class` eingeleitet[1]. Der engen Verwandtschaft dieser beiden Schlüsselworte liegt letztlich die Kompatibilität mit C zu Grunde. Streng genommen ist `class` in C++ sogar überflüssig, wurde aber sicherlich zur Demonstration des Willens zur Objektorientierung eingeführt. Historisch geht C++ eine erweiterte C-Version (C with classes) voraus, in der die Zusammenfassung von Funktionen und Datenelementen in erweiterten `struct`s realisiert war.

Aufgrund dieser Kompatibilität mit C haben Klassen zunächst einmal alle Eigenschaften, die auch C-Strukturen zukommen. Klassen sind daher insbesondere "Behälter" für bzw. Konglomerate verschiedener Datenelemente, die im Allgemeinen von verschiedenem Typ und insbesondere wiederum Strukturen oder Klassen sein können:

```
// person1.h:
#include <string>

struct Person {      // Klassenname
  int age;
```

[1] Diese Schlüsselworte `class` und `struct` haben weitgehend gleiche Bedeutung, unterscheiden sich aber dadurch, dass Datenelemente in Strukturen allgemein zugänglich (`public`), die in `class`es jedoch zunächst für Zugriffe in Funktionen außerhalb der Klasse nicht zugänglich (`private`) sind. Eine genauere Diskussion dieser Begriffe folgt in Abschnitt 3.1.7.

```
  std::string name;  // Klasse der C++ Standardbibliothek
};                   // Semikolon schliesst Dekl. ab

int main() {
  Person harald;     // Klasseninstanz 'harald' der Klasse
                     //  'Person'
}
```

Der hinter `struct` (oder `class`) angegebene Name wird vom Compiler wie ein neuer Datentyp behandelt. Man kann daher Variablen dieses neuen Typs deklarieren und spricht dann von den einzelnen Variablen als Instanzen der Klasse. Jede Klasseninstanz (jedes "Objekt") reserviert Speicherplatz für alle im Klassenkörper deklarierten Variablen und ermöglicht den Zugriff darauf unter dem in der Deklaration genannten Namen. Natürlich sind auch Felder von Strukturen oder Klassen und indirekte Zugriffe über Zeiger oder Referenzen möglich.

Der Compiler stellt die nötige Funktionalität bereit, um Klassen- oder Strukturinstanzen aufeinander zuzuweisen, sie an Unterprogramme zu übergeben oder sie aus Unterprogrammen zurück zu geben. Er sorgt auch dafür, dass der reservierte Speicher wieder freigegeben wird, wenn der Programmfluss die Grenze des Existenzbereiches einer Variable erreicht (die Variable geht *out of scope*).

Auf die Elemente von Klassen und Strukturen greift man im Allgemeinen mittels der Operatoren `.` und `->` zu. Der `->` Operator ist eine Abkürzung für die Kombination aus `*` und `.` für den Zugriff auf Elemente über einen Zeiger auf die Instanz.

```
// person1.cc
#include "person1.h"

void f(){
  Person p;
  Person *p_p = &p;  // Zeiger auf eine Instanz der Klasse Person
  p.age = 27;
  p.name = "Winnetou";

  p_p->age  = 32;    // ueberschreibt Winnetous age
  p_p->name = "Old Shatterhand";
  // aequivalent, aber umstaendlicher:
  (*p_p).age = 32;
  (*p_p).name = "Old Shatterhand";
}
```

Der mit einer Instanz einer Struktur oder Klasse assoziierte Speicherplatz richtet sich nach dem Speicherbedarf der einzelnen Datenelemente. Mittels `sizeof()` lässt sich der benötigte Platz in Einheiten der Größe eines `char` bestimmen. Ein Compiler kann, muss aber nicht, die Datenelemente in der Reihenfolge ihrer Deklaration in der Struktur im Speicher ablegen. Ebenso kann die Größe einer Struktur die Summe der Größe ihrer Elemente überschreiten, wenn der Compiler z.B. aus Gründen eines effizienten Speicherzugriffs die Datenelemente immer auf Speicherworten beginnen lässt. Weiterhin darf aus Gründen, die mit den Regeln für Speicherzugriffe auf Felder zusammenhängen, die Größe einer Struktur diejenige eines `char` nicht unterschreiten, selbst wenn sie, wie das bei Klassen manchmal vorkommt, keine Datenelemente enthält. Damit ist etwa garantiert, dass verschiedene Instanzen einer Klasse immer unterschiedliche Speicheradressen haben.

3.1.2 Elementfunktionen

Neu in C++ gegenüber C ist, dass auch Funktionen als Elemente einer Struktur oder Klasse erklärt werden können. Diese werden als *Elementfunktionen* oder *Methoden*, englisch *member functions*, bezeichnet. Von gewöhnlichen Funktionen unterscheidet sie ein privilegierter und syntaktisch vereinfachter Zugriff auf die Datenelemente der umschließenden Klasse. Zum Beispiel ließe sich die obige Struktur durch spezielle Funktionen zur Berechnung des Geburtstages oder der getrennten Ausgabe von Vor- und Nachnamen der Person erweitern:

```
// person2.h:
#include <string>

struct Datum {
  //...
};

struct Person {
  // member functions
  Datum  birthday() const;          // Geburtstag berechnen
  std::string first_name() const;   // Vorname bestimmen
  std::string surname() const;      // Nachname bestimmen
  void set_name(std::string a_name);// neuen Namen setzen

  // data
  Datum       birth_date;
  std::string name;
};
```

Im obigen Beispiel deutet `Datum` eine weitere benutzerdefinierte Klasse an, die in der Lage ist, ein bestimmtes Datum zu repräsentieren.

Das Schlüsselwort `const` hinter der Argumentliste der gewöhnlichen Elementfunktionen deutet an, dass diese nur lesend auf Datenelemente der Klasse zugreifen und die Klassenvariablen somit unverändert lassen. Dies erlaubt dem Compiler, gewisse Optimierungen bezüglich des Datenzugriffs vorzunehmen. Wichtig ist außerdem, dass auf Instanzen einer Klasse, die ihrerseits als `const` vereinbart wurden, nur konstante Elementfunktionen Anwendung finden können. Man sollte also, wo möglich, den ausschließlich lesenden Zugriff einer Funktion für den Compiler mit `const` kennzeichnen.

Die Definition von Elementfunktionen erfolgt üblicherweise in der Implementierungsdatei,

```
// person2.cc:
#include "person2.h"

Datum
Person::birthday() const {
  return birth_date;
}

std::string
Person::first_name() const {
  // suche das erste Segment, das durch ein Leerzeichen beendet wird
  return name.substr(0,name.find(' '));
}
```

```
std::string
Person::surname() const {
  // suche das letzte Segment, das durch ein Leerzeichen eingel. wird
  // diese einfache Version gibt das fuehrende Leerzeichen mit zurueck
  return name.substr(name.rfind(' '));
}

void Person::set_name(std::string name){
  this->name = name;  // Erlaeuterung siehe unten
}
```

oder in Ausnahmefällen im Klassenkörper selbst. Im letzteren Falle werden die Funktionen automatisch als `inline` behandelt, auch wenn sie nicht explizit so gekennzeichnet sind. Der Bezugsrahmen(*scope*)-Operator `::` identifiziert dabei die Elementfunktionen als zugehörig zur Klasse (`Person`). Er muss an dieser Stelle verwendet werden, weil Funktionen gleichen Namens mit gleichen Argumenten auch in anderen Klassen auftreten dürfen.

Innerhalb von Elementfunktionen wird auf die Datenelemente der Klasseninstanz mit den in der Klassendeklaration verwandten Namen zugegriffen, ohne dass dazu die Operatoren `.` oder `->` benötigt werden. Der Aufruf von Elementfunktionen erfolgt wie der Zugriff auf Datenelemente einer Klasse mittels der Operatoren `.` oder `->`,

```
#include "person2.h"
#include <iostream>

int main(){
  Person  p;   // Datenelemente noch undefiniert
  p.set_name("Stefan Schwarzer");
  std::cout << "Vorname:  '" << p.first_name() << "'\n"
            << "Nachname: '" << p.surname() << "'" << std::endl;
}
```

Zusammenfassung:
(i) Klassendeklaration vereinbaren Namen für neue Datentypen und werden durch die Schlüsselworte `class` oder `struct` eingeleitet.
(ii) Neben Datenelementen sind auch Elementfunktionen erlaubt, die ohne explizite Angabe eines Argumentes Zugriff auf die Datenelemente der Klasse haben.
(iii) Anderenfalls erfolgt der Zugriff auf die Elemente von Klasseninstanzen erfolgt mittels der Operatoren `.` oder `->`.

3.1.3 this - Zeiger

Um innerhalb von Elementfunktionen der Klasse `Person` auf die Klasse als Ganzes Bezug nehmen zu können, stellt C++ das Schlüsselwort `this` bereit. Im Körper von Elementfunktionen steht es stellvertretend für einen Zeiger auf die jeweilige Instanz der Klasse und kann in der für Zeiger üblichen Weise benutzt werden. Innerhalb von Elementfunktionen, die als `const` gekennzeichnet sind, hat `this` den Typ `const<Klasse> * const`, innerhalb anderer Elementfunktionen den Typ `<Klasse> * const`. Eine (sinnlose und gefährliche) Zuweisung auf den `this` Zeiger ist daher niemals möglich, sehr wohl jedoch der Zugriff auf Klassenelemente. Im vorhergehenden Beispiel dient in der Methode `Person::set_name` der `this`-Zeiger dazu, die Variable `name` als Elementvariable zu identifizieren und vom Argument der Elementfunktion zu unterscheiden. Insbesondere bei der Implementierung überladener Operatoren spielt der `this`-Zeiger eine wichtige Rolle (vgl. 3.1.10).

3.1.4 Geburt und Tod von Objekten

Bei der Deklaration einer Variablen vom Typ `Person` muss der Compiler Speicherplatz bereitstellen und diesen geeignet initialisieren. Dazu wird im Falle einer Klasse immer ein sogenannter Konstruktor aufgerufen. Konstruktoren sind spezielle Elementfunktionen, die als Namen den Klassennamen selbst tragen und keine Rückgabetyp, auch nicht `void` festlegen. Wir können beliebig viele Konstruktoren definieren, die sich allerdings, wie für überladene Funktionen typisch, in ihren Argumenten unterscheiden müssen. Ihre Aufgabe ist immer die Herstellung eines konsistenten Zustandes der Objektinstanz durch Initialisierung von Variablen oder Ressourcen, etwa die Initialisierung von Konstanten im Klassenkörper, die Anforderung von Ressourcen wie Speicher, das Öffnen einer Datei oder Netzwerkverbindung, usw. Eine besondere Rolle spielt der Defaultkonstruktor, dem kein Argument übergeben wird. Er wird, falls vom Benutzer kein anderer Konstruktor definiert ist, implizit vom Compiler bereitgestellt und seine Existenz ist für die Initialisierung von Feldern einer Klassenvariable notwendig.

Analog zum Konstruktor einer Klasse gibt es einen Destruktor, der nötigenfalls angeforderte Ressourcen wieder freigibt. Dieser trägt als Name den Klassennamen mit einem vorangestellten `~`.

```
// person3.h:
#include <string>

struct Datum {
  Datum();      // benutzerdefinierter Defaultkonstruktor
  Datum(int day, int month, int year); // Konstruktor
  // ...

  // Datenelemente
  int day;
  int month;
  int year;
};

struct Person {
  Person();     // benutzerdefinierter Defaultkonstruktor
  Person(Datum birthday, std::string name); // initialisiert Geb.tag, Name
  ~Person();    // Destruktor

  // weitere moegliche Elementfunktionen
  //  Datum  birthday() const;          // Geburtstag bestimmen
  //  std::string first_name() const;   // Vorname bestimmen
  //  std::string surname() const;      // Nachname bestimmen
  //  void set_name(std::string a_name);// neuen Namen setzen

  // Datenelemente
  Datum birth_date;
  std::string name;
};
```

Die *Definition* eines Konstruktors enthält noch eine spezielle Sektion zwischen Argumentliste und Beginn des Funktionskörpers, die durch einen Doppelpunkt `:` direkt hinter der Argumentliste eingeleitet wird und die in normalen Funktionen fehlt. Hier kann die Initialisierung der Klassenvariablen beeinflusst werden. Mit der Klammernotation kann für

jede Variable des Klassenkörpers eine Initialisierung durchgeführt werden, im Beispiel für `name` und `birth_date`. Es ist sinnvoll, sich hier den Aufruf einer der jeweiligen Konstruktoren vorzustellen.

```
// person3.cc:
#include "person3.h"

// Implementierung der Konstruktoren fuer Datum und Person

Datum::Datum()  // setzt ein sinnvolles Defaultdatum (POSIX Stunde 0)
: day(1), month(1), year(1970)
{ }

Datum::Datum(int a_day, int a_month, int a_year)
: day(a_day), month(a_month), year(a_year)
{ }

Person::Person(Datum birthday, std::string a_name)
: birth_date(birthday),
  name(a_name)  // benutzt Kopierkonstruktoren fuer birth_date, name
{ }

Person::Person()
:               // Defaultkonstruktor fuer Datum wird implizit aufgerufen
  name("")
{ }

// Destruktor fuer Person muss implementiert werden, da er
//  deklariert ist
Person::~Person(){}  // ruft implizit Destrukturen von Datum und string

// Implementierung weiterer Elementfunktionen
//  ...
```

Mehrere zu initialisierende Variablen können mit Kommata aneinandergereiht werden. Dabei ist zu beachten, dass der Compiler die Reihenfolge in dieser Liste ignoriert und die angegebenen Initialisierungen immer in der Reihenfolge vornimmt, wie sie der Definition im Klassen/Strukturkörper entspricht. Um das Programm verständlich zu gestalten, empfiehlt es sich daher, die Initialisierungsliste wie die Deklarationen zu sortieren.

Auch die nicht explizit in der Initialisierungsliste erwähnten Variablen werden vom Compiler (durch Aufruf des jeweiligen Defaultkonstruktors) initialisiert. Für die Initialisierung eines Datenelementes von Klassentyp kann natürlich jeder beliebige für diese Klasse definierte Konstruktor aufgerufen werden.

In einigen Fällen (Konstanten, Referenzen, `* const`-Zeiger) ist die Initialisierungssequenz vor Betreten des Funktionskörpers sogar die einzige Möglichkeit zur Initialisierung der Variablen.

Da der Wert einer Klassenvariablen im Körper des Konstruktors nur noch durch eine nachträglichen Zuweisung geändert werden kann, sollte man schon aus Effizienzgründen einen möglichst großen Teil der Arbeit des Konstruktors in der Initialisierungssequenz verrichten lassen.

Welcher Konstruktor an welcher Stelle im Programm verwendet wird, wird uns noch häufiger beschäftigen. Eine dieser Stellen ist die Deklaration einer neuen Instanz und der gewählte Konstruktor wird durch die ggf. übergebenen Argumente ausgewählt:

```
// main.cc:
#include "person3.h"

int main(){
  Datum   stefans_birthday(4,7,1963);
  Person  stefan(stefans_birthday,"Stefan Schwarzer");
  Person  p;  // Defaultkonstruktor; Achtung: keine Klammern ()!
}
```

So wie der Konstruktor dafür sorgt, dass Konstruktoren für alle Klassenbestandteile in der Reihenfolge ihres Auftretens in der Klassendeklaration aufgerufen werden, so sorgt der Destruktor für einen entsprechenden Aufruf der Destruktoren der Klassenbestandteile in umgekehrter Reihenfolge. Wie im Falle der Konstruktoren wird diese Eigenschaft sowohl für den benutzer- als auch für den ggf. compilergenerierten Destruktor garantiert. Im Gegensatz zu der möglichen Vielzahl von Konstruktoren gibt es nur einen einzigen Destruktor, der argumentlos und ohne Rückgabetyp deklariert werden muss. Information kann an den Destruktor daher nur in Form von Klassendatenelementen übergeben werden.

3.1.5 Defaultkonstruktor

Ein ggf. vorhandener Konstruktor mit einer leeren Argumentliste heißt Defaultkonstruktor. Wir müssen zwischen Defaultkonstruktoren einfacher Typen und solchen in Klassen unterscheiden. Für `static` deklarierte einfache Variablen nimmt der (compilergenerierte) Default"konstruktor" eine Initialisierung auf 0 vor, in allen anderen Fällen ist der Inhalt einfacher Variablen undefiniert, d.h. sie werden *nicht* initialisiert.

Die vom Compiler generierten Defaultkonstruktoren für Klassen rufen die Defaultkonstruktoren der Klassenelemente (in der Reihenfolge ihrer Deklaration) auf. Handelt sich es dabei um einfache Variablen, so werden diese nicht initialisiert. Statische, einfache Klassenvariablen werden einmal null-initialisiert, bevor die erste Instanz der Klasse erzeugt wird und werden im Folgenden von Konstruktoraufrufen ignoriert. Für Datenelemente, die selber Klassen sind, werden in allen Fällen die Defaultkonstruktoren aufgerufen. Es ist ein Fehler, wenn in diesem Falle gegebenenfalls keine Defaultkonstruktoren existieren.

In diesem Falle und wenn ein anderes als das beschriebene Verhalten gewünscht ist, ist es die Aufgabe des Programmierers, selber einen Defaultkonstruktor zu schreiben. Im untenstehenden Beispiel des Konstruktors für eine Person wird etwa ein leerer Name generiert und das Geburtsdatum auf den 1. Januar 1900 gesetzt, was eine mehr oder weniger sinnvolle Wahl sein kann. Gegebenenfalls könnte natürlich auch eine Variable gesetzt werden, die den etwas ärmlichen Zustand unserer Klasseninstanz markiert. Häufig ist es allerdings sinnvoller, in solchen Fällen ganz auf Defaultkonstruktoren zu verzichten. Sobald eine Klassendeklaration irgendeinen benutzerdeklarierten Konstruktor enthält, wird der Defaultkonstruktor nicht mehr automatisch vom Compiler erzeugt, da davon ausgegangen wird, dass zur Initialisierung der Klasse besondere Maßnahmen nötig sind.

```
// AnotherPerson.h:
#include <string>

struct Datum {
  Datum(int, int, int){ /* .. */ }
  // ...
};

class AnotherPerson {
public:
```

```
    AnotherPerson(const std::string & a_name, const Datum &a_birthday,
      int a_gender)
    : name(a_name), birthday(a_birthday), gender(a_gender) {}

    AnotherPerson()  // Defaultkonstruktor: laesst gender uninitialisiert
    : name(""), birthday(Datum(1,1,1900))
    { }

    enum { male=0, female };

  private:
    std::string name;
    const Datum birthday;
    int   gender;
  };


  // main.cc:
  static AnotherPerson global_person;
  static int global_i;

  int main(){
    int            i;
    AnotherPerson  ap("Stefan Schwarzer",Datum(4,7,1963),
                                          AnotherPerson::male);
    AnotherPerson  p;   // default ctor called
  }
```

Im Beispiel wird `global_person` als Variable vom Klassentyp defaultinitialisiert, dabei wird insbesondere das Element `gender` default-, d.h. für diesen Fall *nicht* initialisiert. Die statische Variable `global_i` wird als einfacher Datentyp hingegen null-initialisiert. Im Hauptprogramm werden `i` und `p` defaultinitialisiert.

Die Defaultkonstruktion eines einfachen Datentyps bewirkt nichts, der Wert der Variablen ist danach undefiniert. Dieses Verhalten kann aus Laufzeitgründen für numerische Typen erwünscht sein, die man in Feldern verwenden möchte.

3.1.6 Kopieren und Zuweisung von Objekten

Im Falle, dass Strukturen nur Elemente der eingebauten Typen enthalten (oder Strukturen, die ihrerseits aus solchen bestehen), reicht häufig eine bitweise Kopie der Einzelelemente, um eine Kopie der gesamten Struktur zu erstellen. Damit lassen sich solche Objekte auf dem Stack ablegen oder aufeinander zuweisen. Solche einfachen Klassen oder Strukturen kann man daher bereits an Unterprogramme übergeben oder als Resultate zurück erhalten.

```
  struct Fraction {             // "plain old data"
    int numerator;
    int denominator;
  };

  Fraction square(Fraction f){ // Kopierkonstruktor rufen, erzeugt Kopie
    // etwas mit f machen, das f veraendert
    f.numerator *= f.numerator;
    f.denominator *= f.denominator;
```

```
    return f;                  // Kopierkonstruktor wird aufgerufen
                               // Destruktor wird fuer f aufgerufen
  }

  int main(){
    Fraction half = {1, 2};    // Strukturinitialisierung wie in C
    Fraction other_half(half); // Kopierkonstruktor
    Fraction quarter;          // un-/defaultinitialisiert
    quarter = square(other_half); //Zuweisung des Resultats
  }
```

Für solche Fälle stellt der Compiler einen Default-Kopier-Konstruktor bereit, der im Falle solcher "POD" (*plain old data*) Strukturen bitweise Kopien anfertigt. Für benutzerdefinierte Klassen, die möglicherweise Datenelemente enthalten, deren Klasse einen eigenen Kopierkonstruktoren definiert, bewirkt der compilergenerierte Kopierkonstruktor den Aufruf aller Kopierkonstruktoren der Datenelemente der Klasse in der Reihenfolge ihrer Deklaration. Damit ist z.B. unsere `AnotherPerson` einwandfrei zu kopieren, obwohl zwei Datenelemente sogar benutzerdefinierte Klassen sind.

Man sollte sich aber in jedem Einzelfall überlegen, ob dieser vom Compiler bereitgestellte Mechanismus ausreichend ist, um das gewünschte Verhalten zu erzielen.

Typische Fälle, in denen der automatisch generierte Kopierkonstruktor nicht ausreicht, sind solche, in denen bereits der reguläre Konstruktor die Anforderung einer Ressource vornahm, etwa dynamisch Speicherplatz anforderte oder eine Datei öffnete. Hier entsteht beim Kopieren die Frage ob sich die Kopien auf dieselbe Datei und denselben Speicherbereich beziehen sollen und wie man sicherstellt, dass die Ressource genau einmal wieder freigegeben wird.

Ähnliche Überlegungen gelten für die Zuweisung. Auch hier stellt der Compiler einen Zuweisungsoperator bereit, der seinerseits die Zuweisungsoperatoren der Datenelemente der Klasse aufruft. Auch dieser ist in der Regel dann nicht mehr ausreichend, wenn in den Konstruktoren ungewöhnliche Ressourcen angefordert werden.

Der Sprachstandard fordert, dass benutzerdefinierte Zuweisungoperatoren oder Kopierkonstruktoren nichtstatische Elementfunktionen der Klasse sein müssen, d.h.

```
  class Homework {
    //...
    Homework(const Homework &other);  // Kopierkonstruktor
    Homework &operator=(const Homework &other);  // Zuweisung
    //...
  };
```

Der Kopier-Konstruktor trägt wie jeder Konstruktor den Namen der Klasse und hat keinen Rückgabetyp. Der Zuweisungsoperator sieht aus wie eine gewöhnliche Elementfunktion wobei der Rückgabetyp eine Referenz auf das Element der linken Seite der Zuweisung sein sollte; beide erfordern als Argument eine konstante Referenz auf ein bereits existierendes Objekt der Klasse.

In der Sektion 3.1.10, in der das Überladen von Operatoren angesprochen wird, werden auch Zuweisungsoperator und Kopierkonstruktoren nochmals eingehender diskutiert.

3.1.7 Datenkapselung und Zugriffsprivilegien: `private`, `public`

Datenelemente und Elementfunktionen einer Klasse können in einer von drei Typen von Sektionen (`public`, `private` oder `protected`) des Klassenkörpers spezifiziert werden. Die Bedeutung von `protected` soll weiter unten im Zusammenhang mit Vererbung (3.2.6) besprochen werden; sie ermöglicht Elementfunktionen in abgeleiteten Klassen den Zugriff auf

Variablen. Diese Sektionen werden durch das jeweilige Schlüsselwort, gefolgt von einem Doppelpunkt eingeleitet und können beliebig häufig durch die Angabe eines anderen Zugriffsstatus "umgeschaltet" werden. Der Status `public` ist default in C-typischen Strukturen, die durch `struct` definiert werden. Er bedeutet, dass alle Datenelemente des Aggregates frei zugänglich sind und aus beliebigen Funktionen heraus verwendet werden dürfen. Dagegen bedeutet eine Deklaration in der `private`-Sektion, dass ausschließlich Elementfunktionen der Klasse selbst Zugriff auf diese Daten haben. Das `private` Verhalten ist der Default in durch `class` eingeleiteten Klassendeklarationen.

Die Verwendung dieser Schlüsselwörter erlaubt dem Programmierer, die zur Repräsentation der Klasse nötigen Daten vor dem Anwender zu "verstecken" und zu verhindern, dass solche Daten direkt, ohne Vermittlung von Elementfunktionen, manipuliert oder gelesen werden können. Es kann daher nicht zu unerwünschten Wechselwirkungen zwischen Programmteilen aufgrund einer Änderung der inneren Repräsentation der Klasse oder durch inkonsistent genutzte Datenelemente kommen.

Die Bedeutung der Verwendung von Elementfunktionen zur Bereitstellung der Schnittstelle zu einer Klasse anstelle der direkten Verwendung der Klassenvariablen (wie in den C typischen `struct` üblich) kann nicht genügend betont werden. Sie erlaubt es, Änderungen des Klassendesigns durchzuführen, ohne dabei andere Teile des Programmes ändern zu müssen, die auf die Klasse und deren Funktionalität zurückgreifen. Die Lösung eines Jahr-2000-Problem bedeutet für ein so geschriebenes Programm die Änderung der inneren Repräsentation des Datums (beispielsweise der Jahreszahl durch einen `long int` statt durch zwei `char`, aber keine verzweifelte Suche nach Programmteilen, die sich auf diese zwei `char` als Repräsentation verlassen haben. In einem solchen Programm muss nur die Datumsklasse neu kompiliert werden, der übrige Programmcode bleibt völlig unbeeinflusst!

Als Beispiel der späteren möglichen Änderung einer Repräsentation ziehen wir auch das Beispiel unserer `Person` heran. Hier könnten wir uns überlegen, vielleicht aufgrund einer häufig nötigen getrennten Nutzung des Vor- und Nachnamens, den Namen nicht als eine, sondern als zwei getrennte Zeichenketten zu speichern. Die Auftrennung erfolgt in Konstruktor und in `set_name()` ohne dass sich die Benutzung der Klasse ändert.

```
// person4.h:
#include <string>

class Datum { };

class Person {
public: // Beginn der allgemein zugaenglichen Sektion
  // Konstruktoren befinden sich ueblicherweise in der public Sektion,
  //  ausser im Falle, dass die Konstruktion der Klasse explizit
  //  verhindern werden soll.
  // Ein Konstruktor in der private Sektion kann durch eine
  //  Freundfunktion oder -klasse oder eine Elementfunktion
  //  aufgerufen werden.
  // Der compilergenerierte Default- oder Kopierkonstruktor ist
  //  immer 'public':
  // Person(){}
  // Person(const Person &rhs){}

  Person(std::string a_name, Datum a_birthday);

  Datum  birthday() const;
  std::string first_name() const;
  std::string surname() const;
```

```
    void set_name(std::string a_name);

  private: // ab hier sind die Elemente gegen Zugriff
           //  durch Nicht-Elementfunktionen geschuetzt
    Datum       my_birthday;
    std::string my_surname;
    std::string my_firstname;
  };

  // person4.cc:

  void
  Person::set_name( std::string a_name ){
    // Vor- und Nachname trennen (hier vereinfacht)
    my_surname   = a_name.substr(a_name.rfind(' '));
    my_firstname = a_name.substr(0,a_name.find(' '));
  }

  Person::Person(std::string a_name, Datum a_birthday )
  : my_birthday(a_birthday)
  {
    set_name(a_name);  // Elementfunktionsaufruf zur Namenstrennung
  }

  // ...
```

Als Faustregel sollten Datenelemente einer Klasse immer `private` deklariert sein. Ist ein direkter Zugriff von außerhalb der Klasse auf ein solches Element nicht vermeidbar, so sollte man sich mit einer `inline` deklarierten bzw. zu einer diesem Zweck im Klassenkörper definierten Elementfunktion behelfen, die den Wert des Elementes zurückliefert. Eine zweite Funktion kann dazu verwendet werden, den Wert der Klassenvariablen zu ändern.

```
  class AnotherPerson {
  public:
    // ...
    Datum get_birthday() const { return my_birthday; }

    void set_birthday(const Datum &a_birthday)
    {
      my_birthday = a_birthday;
    }
  private:
    // ...
    Datum my_birthday;
  };
```

Ein guter Compiler wird in diesem Fall Code generieren, der dem direkten Zugriff auf die Klassenvariable äquivalent ist. Diese Vorgehensweise hält die Option offen, die Implementierung zu einem späteren Zeitpunkt doch noch zu ändern. Sollte ein Anwender der Klasse jedoch erst einmal `public`-deklarierte Datenelemente benutzen, dann gibt es meist kein (einfaches) Zurück mehr.

Funktionen, die die Schnittstelle der Klasse zur Außenwelt darstellen, gehören in die `public` Sektion. Hilfsfunktionen, die bestimmte interne Operationen durchführen und dazu wieder von Elementfunktionen aufgerufen werden, sollten in der `private`-Sektion deklariert

sein, oder, wenn sie auch für die Benutzung durch abgeleitete Klassen sinnvoll sind, durch `protected` geschützt werden.

3.1.8 `friend`-Deklaration

In einigen Fällen ist es erforderlich, auch gewöhnlichen Funktionen oder Methoden einer anderen Klasse Zugriff auf die privaten Daten einer Klasse zu gestatten. In diesem Fall muss der Klassenentwickler diese Funktion als `friend` der Klasse deklarieren. Häufig kann eine `friend` Deklaration einer Funktion dabei helfen, eine symmetrische Beziehung zwischen den Objekten einer Klasse auszudrücken, die mit einer Methode nur unintuitiv unsymmetrisch wiedergegeben werden kann.

```
class Person {
  // ...
  friend void marry(Person &p1, Person &p2);
};

//  marry(..) kann Daten sowohl in p1 und p2 aendern
void marry(Person &p1, Person &p2);
```

Eine solche Freundfunktion sollte wie eine Elementfunktion als Bestandteil der Schnittstelle einer Klasse aufgefasst werden. Daher kann auch nur der Klassenentwickler entscheiden, eine Funktion zu einem `friend` zu machen und diese Tatsache durch eine Deklaration im Klassenkörper zu kennzeichnen. Damit gibt der Klassenprogrammierer sein Versprechen, bei Änderung der Klassenimplementierung nötigenfalls nicht nur die Elementfunktionen, sondern auch diese Freundfunktion zu modifizieren.

Die häufigste Anwendung von Freundfunktionen findet sich wohl bei der Definition von Ein- und Ausgabeoperatoren von Klassen, sofern diese auf private Datenelemente der Klasse zurückgreifen müssen.

Da das erste Argument des überladenen Operators `<<` vom Typ `std::ostream &` sein muss (linker Operand), kann der Operator nicht als Elementfunktion eigener Klassen definiert werden. Um trotzdem Zugang zu deren privaten Datenelementen zu erhalten, kann eine `friend`-Deklaration verwendet werden.

```
// person5.h:
#include <iostream>

class Person {
  // ...
  friend std::ostream & operator>>(std::ostream &, const Person &);
}

// person5.cc:
#include "person5.h"

std::ostream &operator>>(std::ostream &out, const Person &rhs){
  // .. use rhs
  return out;
}
```

Wenn private Elemente höchstens eines Argumentes benötigt werden, dann kann alternativ zu einer `friend`-Funktion eine Elementfunktion in der `public`-Sektion der Klasse deklariert werden, auf die externe Funktionen (etwa ein potentieller `operator<<`) ohne Einschränkungen zugreifen können:

```
// person5.h:
#include <iostream>
// ...

class Person {
public:
  // ...
  std::ostream & print(std::ostream &);
};
```

Diese Elementfunktion hat vollen Zugriff auf die privaten Datenelemente der Klasse und kann ihrerseits ohne Zugriffsprobleme von einem überladenen `operator<<` aufgerufen werden, der so seine Arbeit ohne `friend`-Deklaration delegieren kann.

```
// person5.cc:
#include "person5.h"

std::ostream & operator<<(std::ostream & o, const Person & p){
  return p.print(o);
}

// ...
```

Es ist auch möglich, eine ganze Klasse als `friend` zu erklären. Damit wird der Zugriff auf die eigenen Datenelemente durch alle Elementfunktionen der Freund-Klasse erlaubt. Diese Technik verwendet man häufig bei so genannten inneren Klassen, um Zugriff auf deren Datenelemente zu erlangen.

```
class Outer {

  class Inner {
    int answer;
    friend class Outer;
  };

  public:
  void do_something(){
    Inner life;
    life.answer = 42;   // ok
    // ...
  }
};
```

3.1.9 Temporäre Objekte

Temporäre (anonyme, unbenannte) Objekte treten auf als Zwischenergebnisse bei der Auswertung von Ausdrücken, bei der Übergabe von Argumenten an Funktionen und bei der Rückgabe von Werten aus Funktionen. Man bezeichnet solche unbenannten Objekte auch als *rvalues,* um anzudeuten, dass sie nur auf der rechten Seite von Zuweisungen auftreten können. Benannte Objektinstanzen, Referenzen darauf oder dereferenzierte Zeiger sind *lvalues* und können dagegen sowohl auf der linken als auch auf der rechten Seite von Zuweisungen auftreten.

In Ausdrücken können unbenannte, temporäre Objekte auch "durch Aufruf eines Konstruktors" oder auch als Zwischenergebnisse auftauchen:

```
#include <complex>

int main(){

  // - Konstrukturaufruf
  std::complex<float> i(0.,1.);
  // - temporaere Variable erzeugt, Kopierkonstruktor
  std::complex<float> i2 = 2.f * i;
  // - Defaultkonstruktor wird aufgerufen (!)
  std::complex<float> result;

  // - temporaere Variable complex(2.,3) wird erzeugt
  // - 2 temporaere Variablen fuer die Resultate
  //    der Addition, Summation werden erzeugt,
  // - Zuweisungsoperator wird aufgerufen
  result = i * std::complex<float>(2.,3.) + i2;

  // ...
}
```

In der letzten Anweisung wird zunächst der temporäre komplexe Wert $(2, 3)$ auf dem Stack (oder in Prozessorregistern) abgelegt, danach mit i multipliziert und das temporäre Zwischenergebnis dieser Operation wird wieder auf dem Stack oder in Registern abgelegt. Dann wird in der gleichen Weise $2i$ addiert bevor schließlich die Zuweisung auf `result` erfolgt. Das in spitzen "Klammern" stehende `float` ist ein sogenannter Template-Parameter der Klasse `std::complex`, der in diesem Fall den Typ der für Real- und Imaginärteil verwendeten Variablen bestimmt (siehe Kapitel 4).

Da für Klassentypen die Erzeugung eines temporären Objektes den Aufruf eines Konstruktors bedeutet, ist ein Verständnis dieser Vorgänge wichtig, um die Effizienz eines Programmes zu optimieren.

Funktionen, die konstante Referenzen als Funktionsargumenttypen enthalten, dürfen an dieser Stelle mit Ausdrücken aufgerufen werden. Die Referenz zeigt dann für die Dauer des Funktionsaufrufes auf das temporäre Objekt. Für nichtkonstante Referenzen dürfen nur *lvalues* — d.h. "permanente" Objektinstanzen oder Verweise darauf — übergeben werden.

```
double cref_increment(const double &a){ return a + 1; }

void   ref_increment (double &a)      { a += 1; }

int main(){
  double  b=3.;
  double  result;

  result = cref_increment(b);   // ok, b bleibt 3., result ist 4
  result = cref_increment(b+5); // ok, result ist 8
  ref_increment(b);             // ok, b ist jetzt 4.
  // ref_increment(b+5);        // ERROR
}
```

3.1.10 Überladen von Operatoren

Zu einer vollständigen Schnittstelle einer Klasse gehören üblicherweise überladene Operatoren. Viele Klassen werden etwa den Ein- und Ausgabeoperator `>>` und `<<` überladen,

um ihren "Inhalt" einzulesen bzw. auszugeben. Bei anderen Klassen, wie etwa solchen, die dynamisch Speicherverwaltung betreiben, ist es unerlässlich, z.B. den Zuweisungsoperator zu überladen, um eine korrekte Funktionsweise zu gewährleisten. Insbesondere für Klassen, die arithmetische Funktionen erfüllen, ist auch das Überladen "klassischer" arithmetischer Operatoren sinnvoll, um eine intuitive Benutzung der Klasse zu ermöglichen. Dies sei hier am Beispiel einer Klasse zur Repräsentation komplexer Zahlen demonstriert:

```
// Complex.h:

class Complex {
public:
  // Konstruktor, kann benutzt werden als
  //  Complex(), Complex(re), Complex(re,im)
  Complex(double re=0., double im=0.): m_real(re), m_imag(im) {}

  Complex &
  operator+=(const Complex &rhs){
    m_real += rhs.m_real; m_imag += rhs.m_real;
    return *this;
  }

  Complex &
  operator*=(const Complex &rhs){
    m_real = m_real * rhs.m_real - m_imag * rhs.m_imag;
    m_imag = m_real * rhs.m_imag + m_imag * rhs.m_real;
    return *this;
  }

  //...

  double imag()const{ return m_imag; }
  double real()const{ return m_real; }

private:
  double m_real;
  double m_imag;
};

inline const Complex
operator+(const Complex &lhs, const Complex &rhs){
  Complex tmp(lhs); // Kopierkonstruktor
  tmp += rhs;       // operator+= im Klassenkoerper public definiert
  return tmp;
}

inline const Complex
operator*(const Complex &lhs, const Complex &rhs){
  Complex tmp(lhs); // Kopierkonstruktor
  tmp *= rhs;       // operator*= im Klassenkoerper public definiert
  return tmp;
}
```

Man sieht am Beispiel, dass es ausreicht, von den Operatorpaaren `+`, `+=` bzw. `*`, `*=` nur jeweils die Zuweisungsversion zu implementieren, auf die man sich dann in der zuweisungs-

losen Version beziehen kann. Die Rückgabetypen sind so gewählt, dass sich die Operatoren in der Weise verhalten, wie man das von den einfachen Typen der Sprache C gewohnt ist: So kann auf das Resultat einer Zuweisung wieder zugewiesen werden, während Produkte- oder Summenterme nur als *rvalues*[2] auftauchen, auf die nicht zugewiesen werden kann.

Im Allgemeinen sollte der Argumenttyp `const TYPE &` sein, um ein unnötiges Kopieren der Argumentwerte zu verhindern. Weiterhin sollte man auch untersuchen, ob man eine `const` Qualifikation des `this` Argumentes erreichen kann, etwa bei Operatoren, die ein Objekt nur auslesen oder bei Elementfunktionen, die nur Lesezugriff benötigen; im Beispiel oben träfe das etwa auf die `real()` und `imag()` Elementfunktionen zu. Dieses Versprechen, das Objekt nicht zu verändern, wird durch das `const` hinter dem Namen der Elementfunktion angezeigt.

Operatoren, deren linke Seite vom Typ der Klasse ist, können wahlweise als Elementfunktion der Klasse oder als externe Funktion überladen werden. Der `operator+` im obigen Beispiel hätte also auch als Elementfunktion implementiert werden können. Bei der Implementierung als Elementfunktion entfällt dann die explizite Angabe des linken Arguments in der Argumentliste, welches beim Aufruf als implizites Klassenargument übergeben wird.

```
// OtherComplex.h:

class OtherComplex {
public:
  OtherComplex(double re=0., double im=0.): m_real(re), m_imag(im)
  { }

  const OtherComplex operator+(const OtherComplex & rhs) const
  {
    return OtherComplex(m_real + rhs.m_real, m_imag + rhs.m_imag);
  }

private:
  double m_real;
  double m_imag;
};
```

Weitere Beispiele finden sich im Anhang A.3.

Bis auf die Operatoren `::` (Bezugsrahmen), `.` (Elementauswahl), `.*` (Elementauswahl durch einen Zeiger auf ein Element) und dem ternären Operator `?:` lassen sich alle Operatoren aus der Liste in Sec. 2.5.9 überladen. Man sollte aber beachten, dass man die Priorität und die Assoziativität der Operatoren nicht verändern kann. Viele denkbare Möglichkeiten sind daher durch den von C her vorgegebenen Rahmen nicht sinnvoll zu verwirklichen.

BASIC-Programmierer sind zum Beispiel gewohnt, die Potenzierung durch Benutzung des Symbols `^` auszudrücken, das als Operator in C die bitweise Bildung des logischen ausschließenden Oders der Argumente bewirkt. Allerdings ist es keine gute Idee, `operator^` zu diesem Zweck etwa für Paare zweier Fließkommazahlen oder einer Fließkommazahl und einer ganzen Zahl zu überladen, denn dessen Priorität ist geringer als die von Multiplikation oder Addition. Daher müssten die entstehenden Ausdrücke unnatürlich geklammert werden, um das erwartete Ergebnis zu liefern: `3*4.^2` würde als `(3*4.)^2` und nicht als `3*(4.^2)` interpretiert werden.

Sogar die Verwendung von `operator<<` in C++ für die Ausgabe ist in dieser Hinsicht problematisch, denn die Präzedenz ist für einige Fälle zu hoch. Der Ausdruck

```
std::cout << 3 < 4 ? true : false;
```

[2]Das "r" in *rvalue* deutet an, dass solche Ausdrücke Werte repräsentieren. Im Gegensatz zu *lvalues* können sie nicht als linke Seite einer Zuweisung Verwendung finden.

schreibt nicht etwa `true` bzw. `1` auf das Terminal, sondern gibt einen Syntaxfehler, da der Compiler versucht, zunächst die 3 auszugeben und erfolglos nach einem `operator<` sucht, der für `std::ostream` und `int` überladen ist. Im diesem Fall muss geeignet geklammert werden:

```
std::cout << (3 < 4 ? true : false);
```

Tipps für das Überladen von Operatoren und spezieller Elementfunktionen

In vielen Fällen haben sich für das Überladen von Operatoren bestimmte Muster bewährt, die garantieren, dass sich die überladenen Versionen nahezu wie die eingebauten Versionen verhalten und sich keine unliebsamen Überraschungen einstellen.

`operator<<,>>` für Ein-/und Ausgabe muss ein "externer" Operator sein, möglicherweise `friend` deklariert, denn auf der linken Seite des Operators steht keine Instanz der Klasse. Das erste Argument ist `std::ostream &` bzw. `std::istream &`, das zweite eine Kopie oder eine konstante Referenz auf die Klasse im Falle der Ausgabe und eine nicht-konstante Referenz im Falle der Eingabe. Der Operator gibt die Referenz auf den `ostream` oder `istream` im Argument als Rückgabewert zurück. Dies garantiert die Aufreihbarkeit von `<<` und `>>`

```
// Complex.h:
#include <iostream>

class Complex {
public:
  //...
  friend std::ostream & operator<<(std::ostream &, const Complex &);

private:
  double m_real, m_imag;
};

// Complex.cc:

// ...

std::ostream & operator<<(std::ostream &out, const Complex &c){
  out << "(" << c.m_real << "," << c.m_imag << ")";
  return out;
}
```

`operator=` muss als Elementfunktion der Klasse implementiert sein. Aus Effizienzgründen kann der Operator auf Selbstzuweisung prüfen und dann im Zweifel auf weitere Aktionen verzichten. Man kann ihn im Kopierkonstruktor verwenden, wenn in irgendeiner Weise geprüft wird, ob die linke Seite der Zuweisung eine bereits korrekt initialisierte Datenstruktur enthält (der Kopierkonstruktor kann dazu eine Statusvariable setzen oder Zeiger, die auf noch nicht angeforderten Speicher verweisen, auf Null setzen). Danach müssen Datenelemente überschrieben, Speicher angefordert und kopiert werden. Der Rückgabewert sollte eine Referenz auf die linke Seite sein, also auf das von `this` angesprochene Objekt. Dies garantiert einerseits wie im Fall der Aus- und Eingabeoperatoren die Aufreihbarkeit der Zuweisungsoperatoren und andererseits die *lvalue*-Eigenschaft des Ergebnisses der Zuweisung; d.h., dass auf Ergebnisse von Zuweisungen (wie in C) wieder zugewiesen werden kann.

```
// myclass.h:

class MyClass {
public:
  MyClass() : size(0), buffer(0) {}
  MyClass(const MyClass & rhs);             // Kopierkonstruktor
  MyClass & operator=(const MyClass & rhs); // Zuweisungsoperator
  // ...
  int size;
  char *buffer;
};

// myclass.cc:

MyClass &
MyClass::operator=(const MyClass & rhs){
  // aus Effizienzgruenden wird auf Selbstzuweisung geprueft
  if ( this == &rhs ) return *this;

  // diese Sequenz funktioniert auch wenn in new eine Ausnahme
  //  auftritt oder fuer moegliche Selbstzuweisungen (wenn der
  //  Eingangstest nicht gemacht wuerde)
  char * tmp = new char[rhs.size];
  //  ... now copy data to tmp
  for ( int i=0; i < rhs.size; ++i )
    tmp[i] = rhs.buffer[i];

  // bisherigen Puffer loeschen (funktioniert auch bei buffer==0
  //  wie etwa im Kopierkonstruktor
  delete buffer;

  // erst jetzt wird 'this' auf den neuen Stand gebracht
  buffer = tmp;
  size   = rhs.size;
  return *this;
}
```

Kopierkonstruktor Es passt in diesen Zusammenhang, auch die Implementierung eines Kopierkonstruktors zu zeigen, der mit dem obigen Zuweisungsoperator zusammenarbeitet. Auch er trägt die `const` Qualifizierung des Argumentes. In einigen Fällen muss das Argument verändert werden, obwohl es konzeptionell konstant ist. Zum Beispiel könnte eine Zeichenkette kopiert werden, indem einfach ein Zeiger auf die konstante Zeichenkette kopiert wird und dann ein dazugehöriger Referenzzähler erhöht wird. Der Zugriff auf diesen Zähler wird durch eine `const` Qualifikation des Argumentes verhindert. In solchen Fällen kann für einzelne Datenelemente der Klasse durch das `mutable` Schlüsselwort eine Ausnahme vom Schutz durch `const` erwirkt werden.

```
// myclass.cc:

MyClass::MyClass( const MyClass & rhs ): buffer(0) {
  *this = rhs;     // benutzerdefinierten Zuweisungoperator benutzen
}
```

```
// main.cc:
#include "myclass.h"

int main(){
  MyClass a, b, c;
  (a = b) = c; // legal wie in C: a==c, b unveraendert
  a = b = c;   // b==c, a==c
}
```

`operator +=, -=, *=, /=, ...` Die Implementierung sollte als Elementfunktion erfolgen, der Rückgabewert sollte dann eine nichtkonstante Referenz auf die Klasseninstanz sein, um wie im Falle des Zuweisungsoperators in C auch Zuweisungen auf das Ergebnis dieser Operationen zu erlauben. Im Beispiel `Complex` ist demonstriert, wie `operator+=` und `operator+` zusammenarbeiten können.

`operator +, -, *, /, ...` können dann mittels `+=, -=, ...` entweder als Element- oder Nichtelementfunktionen implementiert werden. Die Nichtelementfunktionen haben den Vorteil, beide Argumente insbesondere bezüglich möglicher Typumwandlungen des Compilers gleichberechtigt zu behandeln, ein Verhalten, das insbesondere bei arithmetischen Typen erwünscht ist. In der Regel sollte die Rückgabe als temporäres, konstantes Objekt erfolgen—dies drückt die aus C bekannte Tatsache aus, dass das unbenannte Ergebnis entsprechender Ausdrücke nur auf der rechten Seite von Zuweisungen auftreten darf. Dabei kann der Aufruf des Kopierkonstuktors zumindest bei der Rückgabe nicht verhindert werden (er kann jedoch für `inline` Implementierungen dieser Operatoren vom Compiler eliminiert werden).

Für zeitkritische Anwendungen gibt es Techniken wie (i) die verzögerte Auswertung [18] von Ausdrücken, d.h. die Erzeugung von Zwischenobjekten, die nur Verweise auf die Operanden speichern und die Operation nur dann durchführen, wenn deren Ergebnis letztlich wirklich benötigt wird, bzw. wie (ii) die Template-Metaprogrammierung [20]), die es erlauben, auch in komplexeren Ausdrücken ohne temporäre Zwischenobjekte auszukommen, aber dafür einen sehr hohen Programmieraufwand bedeuten.

```
// myclass.h:

inline const MyClass
operator+(const MyClass & lhs, const MyClass & rhs){
  MyClass tmp(lhs);    // copies lhs to temporary
  return  tmp += rhs;  // works because += returns *this
}
```

`operator ++, --` Das Resultat der Anwendung des Prefix-Operators `operator++` ist ein modifizierbarer Wert (*lvalue*), auf den zugewiesen werden kann. Wie für andere unäre Operatoren empfiehlt sich die Implementierung als Elementfunktion, die die modifizierte Instanz als nicht-konstante Referenz zurückgibt.

Um den Postfix-Operator von der Prefixversion zu unterscheiden, erhält er ein weiteres `int`-Argument, das aber nicht ausgewertet wird. Der Postfix-Operator gibt einen *rvalue* zurück (den Wert des Arguments vor der Modifikation), der am besten durch eine konstante Kopie des Arguments repräsentiert wird.

```
// MyInt.h:

struct MyInt {
  // ...
```

```
    // prefix version:
    int & operator++(){ val += 1; return val; }
    // postfix version:
    const int operator++(int) { int tmp(val); val += 1; return tmp; }
  private:
    int val;
  };
```

Für die Dekrementoperatoren gilt Entsprechendes.

3.1.11 Typumwandlung

C++ benutzt sehr komplexe Regeln, um Ausdrücken einen Sinn geben zu können, der mit der intuitiven Vorstellung des Programmierers übereinstimmt. Betrachten wir etwa folgendes Programmsegment mit der weiter oben definierten Klasse `Complex` zur Repräsentation komplexer Zahlen:

```
#include "Complex.h"

void f(){
  Complex c, I(0,1);
  c = 5 * I; // (A)
}
```

Um den Ausdruck (A) auszuwerten, bestimmt der Compiler zunächst den Typ der Operanden zu `double` und `Complex`. Er findet jedoch keinen `operator*`, der passende Argumente hätte, um das Produkt auszuwerten. Er sammelt jetzt aus Deklarationsdateien und aus der Liste eingebauter Funktionen alle in Frage kommenden `operator*` zusammen. Im Beispiel wären das unter anderem

```
Complex operator*(const Complex &, const Complex &);
double  operator(double, double);   // eingebaut
int     operator(int, int);         // eingebaut
```

In einem zweiten Schritt erwägt der Compiler dann Typumwandlungen, um die im Programm angetroffenen Operanden für eine dieser Funktionen passend zu machen. Dabei findet er keine Möglichkeit, die `Complex` Instanz in einen `double` oder `int` zu verwandeln, jedoch wird er den `Complex`-Konstruktor erwägen, um aus dem `int` zunächst einen `double` und dann einen temporären `Complex`en Wert zu machen, auf den er eine Referenz erzeugen und diese dann an den `Complex operator*(...)` reichen kann. Das Ergebnis dieser Betrachtungen ist nicht immer eindeutig und unter Umständen sind mehrere Wege über verschiedene Typumwandlungen möglich. In diesem Fall treten recht komplexe Regeln in Kraft, die die "besten" Umwandlungssequenzen bestimmen (eine ausführliche Diskussion findet sich in [18], siehe auch Abschnitt 4.7). Falls diese Regeln nicht zu einem eindeutigen Ergebnis führen, beschwert sich der Compiler über eine Nichteindeutigkeit und bricht den Übersetzungsvorgang ab. In einem solchen Fall muss man "per Hand" dem Ausdruck einen Sinn geben, etwa:

```
Complex   c, I(0,1);
c = Complex(5.,0.) * I;
```

Wie wir im vorhergehenden Beispiel gesehen haben, spielen Konstruktoren daher auch die Rolle von Typumwandlungsoperatoren. Manche Konstruktoren sind für diese Rolle allerdings nicht geeignet:

```
struct String {
  String(char c); // String aus einzelnem Buchstaben
  String(int n);  // leerer String with n alloziierten Bytes
  //...
};

// Verbindung zweier Zeichenketten
const String operator+(const String&,const String&);

String append5(String s){
  s = s + '5';
  s = s + 5;  // ???
  return s;
}
```

Die `append5` Funktion kompiliert klaglos, die zweite Zeile ist allerdings wohl ein Tippfehler, der in diesem Fall den Compiler dazu bewegt, einen leeren `String` mit 5 alloziierten Zeichen zu erzeugen und zu `s` hinzuzufügen. Dieses Verhalten lässt sich durch das Schlüsselwort `explicit` abstellen, das alle automatischen Typumwandlungen verbietet, die einen solchen Konstruktor erfordern würden.

```
struct String {
  String(char c);   // String mit einzelnem Buchstaben
  explicit String(int n); // String mit n alloziierten Bytes
  //...
};

// concatenate two strings
String operator+(const String&,const String&);

String append(String s){
  s = s + '5';
  // s = s + 5;  jetzt ein Fehler
  s = s + String(5); // aber moeglich falls noetig
  return s;
}
```

Eine zweite, in der Praxis weniger benötigte Form der Typumwandlung ist die, in einer Klasse eine entsprechende Elementfunktion bereitzustellen. Diese Elementfunktion hat die spezielle Form `operator <TYPE>()` wobei `<TYPE>` ein beliebiger eingebauter Typ oder eine (selbstdefinierte) Klasse sein kann. Im folgenden Beispiel wird beispielsweise ein Typumwandlungsoperator für `Complex` definiert, der es erlaubt, gemischte Ausdrücke aus der selbstdefinierten Klasse und dem Datentyp `std::complex` aus der Standardbibliothek zu schreiben (die Syntax `std::complex<double>` greift dabei den im nächsten Kapitel vorgestellten `templates` voraus):

```
// Complex.h:
#include <complex>

class Complex {
public:
  //...
  operator std::complex<double>(){
     return std::complex<double>(m_real,m_imag);
```

```
  }

private:
  double m_real;
  double m_imag;
};

std::complex<double> f(Complex a){
  return a;  // ruft Complex::operator std::complex<double>()
}
```

3.1.12 Hintergründe zum Überladen von Operatoren

Nachdem wir uns durch alle diese technischen Details gearbeitet haben, mag man sich fragen, wozu diese anscheinende Komplexität nützlich oder notwendig ist. So kommt etwa JAVA völlig ohne das Überladen von Operatoren aus und kennt auch keine Unterscheidung zwischen Zeiger- und Referenztypen.

Das Überladen von Operatoren ist für die harmonische Einbindung neuer Klassen als Typen in die bestehende Sprachstruktur unter Verwendung der aus C bekannten Struktur von Anweisungen macht die Überladbarkeit von Operatoren erforderlich. Die aus C geerbten Eigenschaften von `struct`s, die initialisiert, aufeinander zugewiesen und kopiert werden können, spiegeln sich in kompilergenerierten und benutzerdefinierten Kopier- und Defaultkonstruktoren und Zuweisungoperatoren wider. Wie wir gesehen haben, ist es in diesem Zusammenhang häufig nicht vermeidbar, der entsprechenden Operation die richtige Semantik zu geben.

So ist mit überladenen Operatoren etwa eine Klasse `complex` definierbar, die sich in ihrer Anwendung nicht von den eingebauten Typen unterscheidet. Insbesondere bei Klassen, die mathematische Konzepte repräsentieren sollen (komplexe Zahlen, Matrizen, Gruppen) kann durch geeignet überladene Operatoren eine intuitive Benutzung erreicht werden.

Wichtig und notwendig wird das Überladen dort, wo bei der Anwendung von Templatefunktionen entsprechende Elemente in der Klassenschnittstelle erwartet werden (siehe Kapitel 4 und 5).

3.1.13 Fragen

1. Welche Aussagen können Sie über den Wert des Ausdruck `sizeof(EmptyArray)` im folgenden Programmkontext machen?

   ```
   class  Empty {};
   static Empty EmptyArray[10];

   int main() {
     return sizeof(EmptyArray);
   }
   ```

2. Welchen syntaktischen Fehler enthält der folgende Programmcode?

   ```
   #include <iostream>

   class Number {
     double m_representation;
   public:
     Number(double n) : m_representation(n) {}
   ```

```
  friend std::ostream &
    operator<<(std::ostream & o, const Number &n);
  Number & operator*=(const Number &lhs) {
    m_representation *= lhs.m_representation;
    return *this;
  }
  // ...
};

std::ostream &
operator<<(std::ostream & o, const Number & n){
  return o << n;
}

Number debug_square(const Number &n) {
  std::cout << n << std::endl;
  return n *= n;
}

int main() {
  Number  n(1.4142);
  std::cout << debug_square(n);
}
```

3.2 Vererbung

Die Implementierung einer Klasse ist ein nicht zu unterschätzender Aufwand. Dieser mag sich lohnen angesichts der Garantien, die man über die Konsistenz der Daten des Objektes während seiner Lebenszeit machen kann, angesichts des verminderten Fehlersuchaufwandes, weil die Objektdaten nur über das (hoffentlich wohldefinierte) Klasseninterface aus Elementfunktionen Veränderungen erfahren können und angesichts der zusätzlichen Benutzungsicherheit und –freundlichkeit, die aus einer sauber definierten und intuitiven Schnittstelle aus Elementfunktionen hervorgeht.

Wie wir bereits oben gesehen haben, ist es nützlich, über Klassenbibliotheken zu verfügen, die grundlegende Konzepte durch intuitiv bedienbare Klassen ausdrücken (Beispiele sind aus der Standardbibliothek die `string` und `complex`-Klassen für Zeichenketten und komplexe Zahlen oder auch die eigene Datumsklasse). Solche Klassen lassen sich dann in vielfältiger Weise als Bausteine in eigenen, mächtigeren Klassen verwenden, wie wir das in den vorangegangenen Beispielen der Verwendung der `std::string` und `Datum` Klassen in `Person` kennen gelernt haben.

Über diese Form der (Wieder)Verwendbarkeit hinaus erlauben objektorientierte Sprachen, die Funktionalität von Klassen systematisch und hierarchisch zu erweitern, ohne dass dabei der bereits für die "alte" Klasse geschriebene Programmcode geändert werden müsste. Dabei ist das Hinzufügen von Datenelementen und Methoden möglich (das Objekt erhält neue, speziellere Eigenschaften), aber auch das Überschreiben bereits bestehender Methoden, welches erlaubt, unter Beibehaltung der Programmierschnittstelle die Eigenschaften des Objektes zu ändern. Damit können "neue" Objekte in "altem" Code verwendet werden. Diese Art der Wiederverwendung bestehender Programmstrukturen und -algorithmen nennt man "Vererbung."

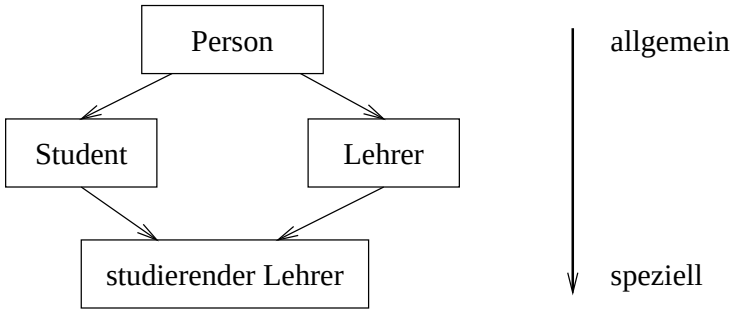
Vererbung dient

1. der Erweiterung vorhandener Klassen hinsichtlich zusätzlicher Datenelemente oder Elementfunktionen,

2. zur Wiederverwendung von Programmteilen aus fundamentaleren (Basis)klassen, insbesondere das Ersetzen (Überschreiben) von Elementfunktionen und

3. zur Festlegung von Programmierschnittstellen, d.h. den Deklarationen von Elementfunktionen, die erst in hierarchisch höher spezialisierten Klassen definiert oder neu festgelegt werden.

Vererbung führt stufenweise von allgemeineren zu spezielleren Klassen, wie dies etwa in der folgenden Skizze dargestellt ist.



Hier wird angedeutet, wie die Eigenschaften einer Klasse Person an die Klasse Student weitergegeben (vererbt) werden. Die Vererbungsbeziehung zwischen zwei Klassen drückt aus, was man umgangssprachlich vielleicht auf zwei Weisen sagen kann: (i) Ein Student kann alles, was eine Person kann, aber zusätzlich noch weitere Dinge, z.B. ein Fach studieren; oder (ii) ein Student "ist eine besondere" Person. Im gleichen Sinne "ist" ein Lehrer eine "besondere" Person und ein studierender Lehrer "ist" sowohl "ein besonderer Lehrer" als auch "ein besonderer Student." Vererbung fügt Klassen immer neue Eigenschaften hinzu und schreitet daher vom Allgemeinen zum Speziellen fort.

3.2.1 Syntax

Dem obigen Schema entsprechen die folgenden C++-Deklarationen:

```
#include <string>

class Person {
public:
  Person(std:string name);
  // ...
};

class Student : public Person {
public:
  Student(std::string name, int matrikelnummer);
  //...
};

class Lehrer : public Person {
public:
  Lehrer(std::string name, std::string fach);
  //...
};
```

```
class StudierenderLehrer : public Student, public Lehrer {
public:
  StudierenderLehrer(std::string name,
                     std::string fach, int matrikelnummer);
  //...
};
```

3.2.2 Zugriffsprivilegien

Die Vererbung einer Klasse wird dabei durch den Doppelpunkt hinter dem Namen der neu definierten Klasse angezeigt. Nach dem Schlüsselwort `public`, `protected` oder `private`, das wie im Fall der Elementfunktionen und -variablen die Zugriffsrechte des Klassenbenutzers regelt, schließt sich der Name der so genannten Basisklasse an, deren Eigenschaften geerbt werden. Dabei kann es sich wie bei `StudierenderLehrer` ggf. um eine ganze Liste von Basisklassen handeln. Eine `public`-Spezifikation erlaubt die Benutzung der Elementfunktionen der Basisklasse durch Benutzer der erbenden Klasse. Einschränkungen der Benutzbarkeit von Elementfunktionen durch `private` oder `protected` in der Basisklasse bleiben davon unberührt. Die `public`-Vererbung ist die bei Weitem wichtigste und häufigste Form.

Mittels `private` oder `protected` kann die Benutzung der Elementfunktionen der Basisklasse auf die erbende Klasse oder zumindest auf die hierarchisch folgenden Klassen (`protected`) beschränkt werden. Für den Benutzer der erbenden Klasse ist die Vererbung dann nicht als solche erkenn- oder nutzbar.

Als Faustregel kann man sich merken, dass man (`public`) Vererbung dort einsetzen sollte, wo die Beziehung zwischen den Klassen durch eine "ist ein(e)" Beziehung charakterisiert werden kann: Ein Student ist eine (spezielle) Person.

Im Gegensatz dazu lassen sich Datenelemente von Klassen häufig durch eine "hat ein(e)" Beziehung identifizieren. Beispielsweise ist ein `Student` durch eine Matrikelnummer charakterisiert (er "hat" eine solche), legt das nahe, diese als ein Datenelement der Klasse Student zu realisieren, und da er auch eine Person ist, sollte die Klasse von `Person` abgeleitet werden.

Vererbung mittels `private` und in ähnlicher Weise durch `protected` drückt "Implementierung durch" aus.

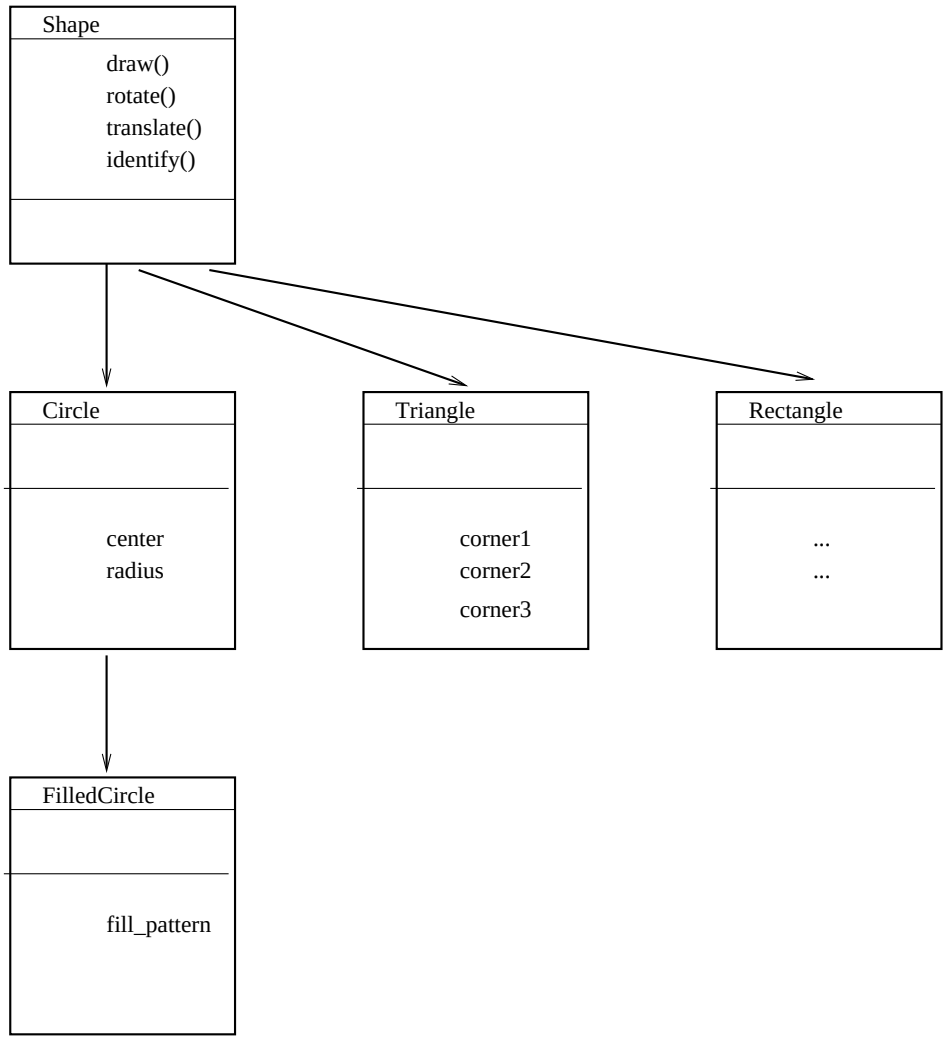
3.2.3 Abstrakte Basisklassen

Als weiteres Beispiel soll ein Grafikprogramm dienen, welches mit Objekten umgehen soll, die etwa an einer bestimmten Position gezeichnet werden müssen und die für den Benutzer durch eine Zeichenkette identifiziert werden sollen.

Wenn wir versuchen, die oben genannten Prinzipien anzuwenden, dann sollten geeignete Basisklassen möglichst einfach geartet sein und wenige Datenelemente enthalten und erbende Klassen sollten mehr und mehr Eigenschaften repräsentieren, was durch immer weitere zusätzliche Datenelemente realisiert wird.

In unserem Fall wäre eine geeignete Basisklasse eine, die gar keine Datenelemente enthält und etwa die abstrakte Eigenschaft "Form" grafischer Objekte repräsentieren könnte, die allen Objekten gemein ist. Eine Form lässt sich u.a. zeichnen, drehen, verschieben und ihr Typ, falls spezifiziert, identifizieren. Formen scheinen also alle Eigenschaften zu besitzen, die wir in unserem Programm von grafischen Objekten erwarten. Spezielle "Formen", wie Kreise oder Rechtecke, entstehen aus der allgemeinen Form durch Hinzufügen von Eigenschaften. Ein Klassendiagramm für unser Programm könnte also etwa so aussehen:

Wir haben unserer Basisklasse keine Datenelemente gegeben, sie jedoch mit Elementfunktionen ausgestattet. Man kann sich jetzt berechtigterweise fragen, was insbesondere die Elementfunktionen `draw`, `rotate`, `translate` leisten sollen, denn das Fehlen von Datenelementen heißt ja letztlich, dass diese Funktionen gar nichts Sinnvolles zeichnen, drehen, oder



verschieben können. Die Identifikation als Zeichenkette "Form" ist hingegen wohl vorstellbar und kann im Rahmen einer "normalen" Elementfunktion ausgedrückt werden. Sobald jedoch Klassen, die von `Shape` erben, Datenelemente vorhanden sind, die die Position und die geometrischen Eigenschaften des Objektes festlegen, können auch `draw`, `rotate` und `translate` sinnvoll implementiert werden.

Um dem Compiler mitzuteilen, dass eine Funktion des Klasseninterfaces ein abstraktes Konzept ausdrückt, das die Basisklasse mit allen abgeleiteten Klassen teilt, benutzt man das Schlüsselwort `virtual`. Man kann `virtual` auch als eine Mitteilung an den Compiler und andere Programmierer auffassen, dass man eine so gekennzeichnete Elementfunktion in einer erbenden (abgeleiteten) Klasse überschreiben möchte. Der Aufruf einer virtuellen Funktion ist mit einem geringfügig höheren Aufwand verbunden als der einer nicht-virtuellen Funktion.[3] Man kann zwar auch nicht-virtuelle Elementfunktionen überschreiben, wir werden aber weiter unten sehen, dass das unabsichtlich zu recht subtilen Fehlern führen kann. Bis man ein genaues Verständnis dieser Problematik hat, sollte man alle Elementfunktionen außer den Konstruktoren und überladenen Operatoren zu `virtual` erklären.

[3] Genauer gesagt, bewirkt die Verwendung von `virtual`, dass immer diejenige Elementfunktion aufgerufen wird, die dem dynamischen Typ des Objektes entspricht, während das Weglassen das Aufrufen der dem statischen Typ entsprechenden Funktion bewirkt. Diese beiden Funktionen können verschieden sein, wenn Zeiger oder Referenzen auf Objekte abgeleiteter Klassen verweisen.

Um dem Compiler mitzuteilen, dass er sich noch bis zur Definition einer abgeleiteten Klasse mit seiner ansonsten berechtigten Forderung nach Implementierung einer Elementfunktion gedulden soll, schreibt man hinter eine als `virtual` deklarierte Funktion an Stelle eines Semikolons oder eines Funktionskörpers die Zeichen `=0;` Eine derartig deklarierte Funktion nennt man häufig "rein" virtuell und Klassen, die solche Funktionsdeklarationen enthalten, heißen abstrakte oder Schnittstellenklassen.

Um dem Compiler also mitzuteilen, dass wir die Funktion `identify`, die in der Basisklasse den Namen `"Shape"` zurückliefern sollte, in erbenden Klassen überschreiben wollen, wo sie `"Circle"`, `"Triangle"`, etc. liefern soll, setzen wir das Schlüsselwort `virtual` vor ihre Deklaration. Das nachgestellte `=0` besagt, dass `draw`, `rotate`, `translate` erst in Klassen implementiert werden, in denen uns die nötigen Informationen zur Verfügung stehen.

```
// shape.h:
#include <string>

class Shape{
public:
  virtual std::string identify();
                          // implementiert in dieser Klasse (.cc Datei)
  virtual void draw()=0;  // implementiert in erbender Klasse
  // ...
  virtual ~Shape(){}      // virtueller Destruktor
};

class Circle: public Shape {
public:
  // Konstruktoren werden niemals vererbt
  Circle(double radius, Point center) : r(radius), c(center){}
  virtual std::string identify();  // ueberschreibt Basisklassenversion
  virtual void draw();     //   ... implementiert in der .cc Datei
  // ...
private:
  double radius;
  point  center;
};

class FilledCircle: public Circle {
public:
  // expliziter Aufruf des Basisklassenkonstruktors
  FilledCircle(double r, Point center) : Circle(r,center) {}
  virtual std::string identify();  // ueberschreibt Circle::identify()
  virtual void draw();      // ueberschreibt Circle::draw()
  // ...
  // no new data section
};
```

In einer Implementierungsdatei wird das Schlüsselwort `virtual` nicht wieder aufgenommen.

```
// shape.cc

#include "shape.h"

// ...
```

```
std::string
FilledCircle::identify()
{
  return "FilledCircle";
}

// ...
```

Ferner gilt: Einmal virtuell, immer virtuell. Eine in einer Basisklasse als `virtual` deklarierte Funktion ist auch in allen abgeleiteten Klassen virtuell. Damit erhält man die gleichen Ergebnisse, wenn man auf die Elementfunktion einer Klasse einmal über ein konkretes Objekt und ein anderes Mal über einen Basisklassenzeiger zugreift.

3.2.4 Eigenschaften spezieller Elementfunktionen

Wie im Falle von `FilledCircle` angedeutet, kann in den Initialisierungslisten der Konstruktoren der erbenden Klassen ein Konstruktor der Basisklasse aufgerufen werden, um deren direkte Initialisierung zu erreichen. Steht dort kein Konstruktor, so wird vom Compiler automatisch der Defaultkonstruktor der Basisklasse eingefügt. Sollte die Basisklasse keinen Defaultkonstruktor aufweisen, so ist die Nennung in der Initialisierungsliste die einzige Möglichkeit zur Initialisierung. Danach schließt sich wie üblich die Initialisierung der eigenen Datenelemente an.

```
// shaded_circle.h:
#include "shape.h"

class ShadedCircle : public FilledCircle {
public:
  ShadedCircle(double r, Point center, int a_pattern):
        FilledCircle(r,center), pattern(a_pattern) {}
  // ...
protected:
  int pattern;
};
```

Für compilergenerierte Default- oder Kopierkonstruktoren gilt sinngemäß das bereits in Abschnitt 3.1.4 Gesagte. Der Compiler wird grundsätzlich versuchen, Defaultversionen zu generieren, die ihrerseits zunächst wieder, soweit vorhanden, die Versionen der Default- und Kopierkonstruktoren der Basisklassen aufrufen. Danach schließt sich dann die Default- bzw. Kopierinitialisierung der hinzugekommenen Klassenvariablen an. Bei Destruktoren wird gerade umgekehrt verfahren, also zunächst die Destruktoren der Klassenvariablen, danach derjenige der Basisklasse aufgerufen.

Für automatisch generierte Zuweisungsoperatoren wird, falls vorhanden, der Zuweisungsoperator der jeweiligen Basisklasse aufgerufen und dann jeweils die Zuweisungsoperatoren der Datenelemente.

3.2.5 Virtuelle Funktionen

Verwendung und Wirkungsweise

Die Wirkungsweise virtueller Funktionen lässt sich am besten durch ein Beispiel erläutern. C++ erlaubt keine Instanzen von abstrakten Klassen wie `Shape`, wohl aber Zeiger auf solche abstrakten Klassen. Diese Beschränkung ist wichtig, da ja in abstrakten Klassen nicht alle

Elementfunktionen definiert sind. Wir deklarieren daher ein (nicht-abstraktes) Objekt der Klasse `FilledCircle`. Auf dieses verweisen wir mittels eines Zeigers auf ein `Shape`-Objekt.

Dass dies überhaupt möglich ist, garantiert uns der Vererbungsmechanismus. Dieser erlaubt, dass Zeiger nicht nur auf Objekte vom Typ des Zeigers, sondern auch auf davon abgeleitete, speziellere verweisen können. Ähnlich können (über den Aufruf des Kopierkonstruktors oder des Zuweisungsoperators der Basisklasse) auch abgeleitete Klassen auf deren Basisklassen kopiert oder zugewiesen werden. Dabei geht allerdings während des Kopierens Information verloren (s.u.).

Man kann jetzt die Elementfunktion `draw()` des `FilledCircle` in der gleichen Art und Weise aufrufen, wie man die entsprechende Funktion der Basisklasse `Shape` angesprochen hätten. Wäre die Funktion nicht als virtuell erklärt worden und sowohl in `Shape` als auch in `FilledCircle` implementiert, so riefe der Compiler hier die Elementfunktion von `Shape` auf. Bei virtuell deklarierten Funktionen ruft der Compiler also trotz des Zugriffs über den Basisklassenzeiger die "richtige" Funktion auf dem "richtigen" Objekt (dynamischer Typ) auf, während bei nicht-virtuellen Funktionen das so genannte Slicing auftritt (siehe 3.2.5), bei dem nur der Basisklassenanteil (statischer Typ) des möglicherweise abgeleiteten Objektes verwendet wird.

```
Shape *p_s;

FilledCircle c(2.3,Point(2.,3.));
p_s = &c;
(*p_s).draw();  // zeichnet gefuellten Kreis
p_s->draw();    // aequivalent
c.draw();       // aequivalent
(&c)->draw();   // aequivalent

Rectangle    r(Point(1.,1.),Point(2.,2.));
p_s = &r;
(*s).draw();    // zeichnet Rechteck
s->draw();      // aequivalent
r.draw();       // aequivalent

Shape *a[20];
a[0] = &c;
a[1] = &r;

// zeichnet Objekte verschiedenen Typs
for(int i = 0; i < 2; ++i){
  (*a[i]).draw();
}
```

Diese Eigenschaft, dass ein Zeiger eines Typs auf Objekte anderer (abgeleiteter) Typen mit (sehr) unterschiedlichen Eigenschaften verweisen kann, bezeichnet man als Polymorphismus. Polymorphismus ist ein Bestandteil "echter" Objektorientierung einer Programmiersprache. So kann durchaus eine Funktion, die vor langer Zeit für `Shape *` geschrieben wurde, mit den allerneusten Objekten (etwa von `Shape` abgeleitete grüngestreifte Teddybären) umgehen, ohne dass der entsprechende Code auch nur neu kompiliert werden müsste.

In einer Klasse mit mindestens einer virtuellen Funktion sollte der Destruktor ebenfalls virtuell deklariert sein. Nur so kann garantiert werden, dass, falls in der Klassenhierarchie nichttriviale Destrukturen vereinbart werden, diese über einen Basisklassenzeiger erreicht werden können.

Zusammenfassung:

(i) Adressen einer abgeleiteten Klasse können auf Zeiger der Basisklasse zugewiesen werden.
(ii) Im Unterschied zu nichtvirtuellen Funktionen kann die Implementierung einer virtuellen Funktion in einer abgeleiteten Klasse erfolgen.
(iii) Virtuelle Funktionen sind dafür gemacht, dass sich Klassen bei Referenzierung durch Basisklassenzeiger polymorph verhalten, d.h., die jeweils zum "richtigen" (dynamischen) Typ gehörende Funktion aufgerufen wird.
(iv) Die Verwendung nicht-virtueller Funktionen kann dazu führen, dass zur Programmlaufzeit die zusätzlichen Eigenschaften abgeleiteter Klassen scheinbar "verloren gehen".
(v) Virtuelle Funktionen, die erst in abgeleiteten Klassen implementiert werden, werden durch `=0` als "rein virtuell" gekennzeichnet.

Nicht-virtueller Aufruf von Elementfunktionen

Es steht uns natürlich frei, Elementfunktionen auch nicht-virtuell aufzurufen. Allerdings müssen wir dies dem Compiler explizit mitteilen, indem wir mit den Bezugsrahmenoperator `::` angeben, auf welche Basisklasse wir uns beziehen möchten. Diese Möglichkeit ist häufig hilfreich bei der Implementierung komplexer Funktionen, die in Basisklassen vielleicht schon teilweise implementiert sind:

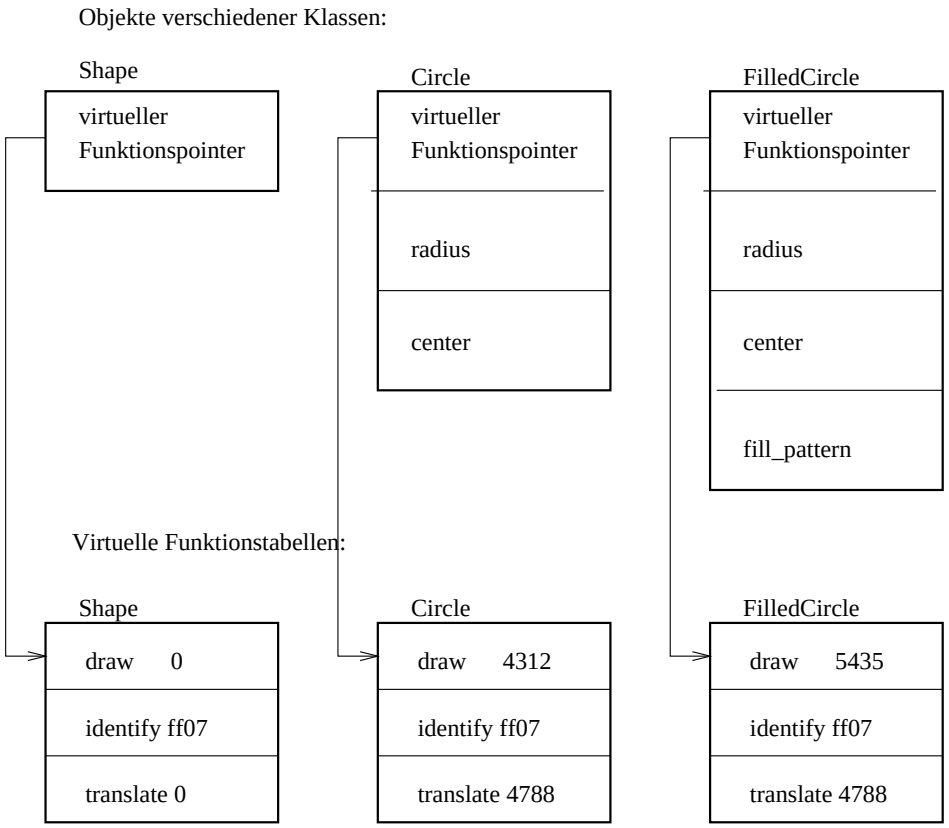
```
// shape.cc:

#include "shape.h"

// ...

void
Circle::draw(){ /* ... Kreis zeichnen */ }

void
FilledCircle::draw(){
  Circle::draw(); /* Aufruf der Basisklassenfunktion */
  this-> Circle::draw(); /* aequivalent */
  /* ... */       /* Anweisungen zum Fuellen */
}

// ...
```

Implementierung virtueller Funktionen

Man versteht die Wirkungsweise virtueller Funktionen besser, wenn man sich eine mögliche Implementierung in einem fiktiven Compiler anschaut. Dieser könnte den zu einem Objekt einer Klasse mit virtuellen Funktionen gehörigen Speicher etwa so aufteilen:

Die Tabellen enthalten Einträge (Funktionseinsprungadressen), die entweder mit denen der Basisklasse übereinstimmen, wenn die virtuelle Funktion nicht überschrieben wurde, oder anderenfalls die Adressen klasseneigener Funktionen. Abstrakte Klassen haben an einigen Stellen keine (0) Einträge für diejenigen Funktionen, die `=0` erklärt wurden.

Diese Funktionen bekommen bei Aufruf noch die Lage des Objektes im Speicher in Form des `this` Zeigers mitgeteilt und greifen dann in ihrer Codierung auf die jeweiligen Datenelemente zu. Um eine Funktion virtuell aufzurufen, benötigt der Compiler daher (i) einen Zeiger auf die virtuelle Funktionentabelle der "Original"Klasse und (ii) einen Zeiger auf z.B. den Beginn der Datensektion des Objektes. Wenn die Lage des Zeigers auf die virtuellen Funktionstabellen relativ zum Datensegment festliegt, reicht sogar eine einzige



Adresse aus. Diese kann dann beispielsweise auf einen Zeiger vom Typ *Zeiger auf Basisklasse* zugewiesen werden. Damit wird klar, wie die Information darüber, welchen Typ ein gegebenes Objekt "wirklich" (dynamischer Typ) hat, beim Umgang mit Zeigern erhalten bleiben kann.

Im Programmieralltag braucht man von diesem Mechanismus nichts zu wissen. Wichtig ist nur, dass man versteht, wann man virtuelle Funktionen benötigt und wann nicht. Als Faustregel braucht man sie immer. Man sollte nur dann keine virtuellen Elementfunktionen verwenden, wenn man (i) garantiert nur mit Klasseninstanzen arbeiten will, wenn (ii) nicht abzusehen ist, dass von der betroffenen Klasse jemals abgeleitet werden wird und wenn (iii) es auf das letzte Quäntchen Laufzeiteffizienz und Speicherplatz ankommt.

Slicing

Polymorphes Verhalten (d.h. Auswahl von Elementfunktionen aufgrund des "dynamischen" Originaltyps einer Variablen) erhält man bei Verwendung von Referenzen oder Zeigern für den Zugriff auf Objekte. Über den Umweg des Zeigers auf die Elementfunktionstabelle bleibt dabei die Typinformation erhalten. Bei einer Zuweisung oder Kopie hingegen geht die ursprüngliche volle Typinformation verloren. Es verbleibt der Kopie nur noch die Datensektion und die Funktionstabelle des (Basis)Klassentyps, der Ziel des Kopier- oder Zuweisungsvorgangs war.

```
void f1(Circle c){
  c.draw();
}

void f2(Circle& c){
```

```
  c.draw();
}

void f3(Circle* c){
  c->draw();
}

int main(){
  FilledCircle fc;
  f1(fc);
  f2(fc);
  f3(&fc);
}
```

In diesem Beispiel wird beim Aufruf von `f1()` das Argument `c` erst einmal in einen `Circle` konvertiert. Das heißt, dass das zusätzliche Datensegment, das den `FilledCircle` auszeichnete, gewissermaßen abgeschnitten wird und nur das des `Circle` übrig bleibt. Gleichermaßen wird die Tabelle der virtuellen Funktionen durch die des `Circle` ersetzt. Dieses Verhalten bezeichnet man als "slicing". In der Funktion `f1()` wird daher nur ein Kreis gezeichnet. Im Fall von `f2()` und `f3()` bleiben die Informationen von `FilledCircle` vollständig erhalten, da letztlich Zeiger auf das ursprüngliche Objekt übergeben werden.

Fragen und Antworten
F: Braucht man kein `const &`, damit beim Erzeugen von temporären Objekten nichts schief geht?
A: An `f2()` können nur Variablen ("lvalues") übergeben werden, aber keine temporären Objekte. Falls dies eine Einschränkung der Funktionalität bedeutet, so kann man das Argument von `f2()` auch als `const &` übergeben sofern die folgenden Bedingungen erfüllt sind. Alle benutzten Funktionen, die auf `c` zugreifen, Elementfunktionen oder nicht, sowie Zugriffe auf Datenelemente, dürfen nur lesend erfolgen. Bei (Element-)Funktionen verlangt das, dass diese entsprechend deklariert und implementiert sein müssen (s.u.).

F: Muss man, wenn man in einer Elementfunktion von `Circle` eine Funktion von `Shape` aufruft, Circle::Shape::draw() schreiben?
A: Nein, innerhalb der Elementfunktion ruft `Shape::draw()` die Funktion der Basisklasse auf.

F: Was passiert mit normalen Funktionen (nicht `virtual`)?
A: Beim Aufruf nichtvirtueller Funktionen wird die Information nicht ausgenutzt, auf welche abgeleitete Klasse der Pointer zeigt. Es werden dann immer die Funktionen der Basisklasse aufgerufen und entsprechend nur die Datenelemente verwendet, die sich in der Basisklassensektion befinden.

Überladen von Elementfunktionen

Der Typ einer Funktion, die eine virtuelle Funktion überschreiben soll, muss im Allgemeinen die gleiche Signatur (Argumente und deren Qualifizierung, Rückgabetyp)[4]aufweisen. Definiert man eine Funktion mit anderer Signatur in einer abgeleiteten Klasse, so verdeckt man dadurch die Funktion der Basisklasse. Diese Funktion ist dann keine, die die Basisklassenfunktion virtuell überschreibt. Um die Basisklassenfunktion wieder zugänglich zu machen, muss man sie in den Klassenbezugsrahmen explizit importieren, z.B. indem man von einer `using` Direktive Gebrauch macht.

[4]Für die Rückgabetypen gelten etwas weniger starke Anforderungen: diese müssen nur kovariant sein, d.h. es kann sich auch um Referenzen oder Zeiger handeln, die auf gemeinsame Basis- oder abgeleitete Klassen verweisen.

```
class Shape {
public:
  // erklaere die draw Funktion als nur lesend
  virtual void draw() const=0;
  // ...
};

class Circle: public Shape {
public:
  // diese muss das dann auch in abgeleiteten Klassen bleiben
  virtual void draw() const { /* ... */ };
  // ...
};

struct Pattern {
  // ...
};

class FilledCircle: public Circle {
public:
  // auf Grund des Argumentes nicht ueberschreibend:
  virtual void draw( Pattern p ) const { /* ... */ };

  // benoetigt, um die Basisklassenfunktion wieder zugaenglich zu machen
  using Circle::draw;
  // ...
};

int main()
{
  FilledCircle  fc;
  Circle        &c = fc;
  Pattern       p;

  fc.draw();    // ruft Circle::draw()
  fc.draw(p);   // ok, ruft FilledCircle::draw(Pattern)
  // c.draw(p); // Fehler - draw(p) existiert nicht in Circle
  c.draw();     // ok, virtueller Aufruf von FilledCircle::draw()
}
```

Fragen und Antworten

F: Wieso braucht man die Zuweisung `=0`? Reicht nicht `virtual`?
A: Mit der Zuweisung `=0` erklärt man die Funktion zu einer *rein* virtuellen. Man zwingt damit den Implementierer einer abgeleiteten Klasse dazu, diese Funktion zu schreiben. Die Basisklasse selbst wird damit automatisch zu einer abstrakten Klasse. Eine solche Funktion dient damit der Schnittstellendefinition. Es kann keine Objektinstanzen vom Typ dieser Klasse geben. Jedoch kann man Zeiger auf die Basisklasse haben, die auch auf Objekte ihrer abgeleiteten Klassen zeigen können.

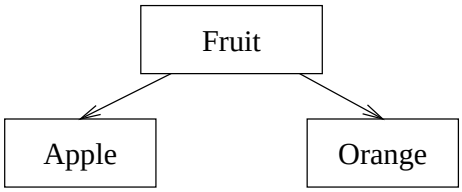
F: Muss eine rein virtuelle Funktion sofort in der nächsten abgeleiteten Klasse implementiert werden?
A: Nein. Sie muss erst in einer Klasse implementiert werden, von der man Objekte instantiieren möchte. Man kann also z.B. zuerst eine Interface-Definition erweitern.

F: Wenn bei einer Funktion nicht `=0` steht, ist sie dann noch virtuell?
A: Ja, siehe oben.

F: Bedeutet eine virtuelle Funktion in der Basisklasse, dass man sie in der abgeleiteten Klasse implementieren muss?
A: Nein. Dies ist nur bei den so genannten rein virtuellen Funktionen der Fall (erkennbar am =0 in der Deklaration).

F: Gibt es virtuelle Konstruktoren und Destruktoren?
A: Virtuelle Konstruktoren gibt es nicht, da bei der Konstruktion eines Objektes genau bekannt sein muss, wieviel Platz im Speicher belegt wird und wie dieser initialisiert werden muss. Ein virtueller Destruktor sollte immer dann deklariert werden, wenn man in der Klasse auch virtuelle Funktionen verwendet. Wenn man von einer Klasse ableitet, die keinen virtuellen Destruktor hat, wird das Verhalten des Programmes undefiniert, wenn ein Objekt mit nichttrivialen Datenelementen durch einen Basisklassenzeiger vernichtet wird.

Hier ist ein weiteres Beispiel, in dem von einer allgemeineren Basisklasse zu spezielleren Klassen hin vererbt wird und einige Basisklassenzeiger in einem Feld verwaltet werden. Die Funktion `identify()` wird dann virtuell aufgerufen und ist immer die zu dem im Speicher befindlichen Objekt gehörende (dynamischer Typ) und nicht die der Basisklasse (statischer Typ).




```
#include <iostream>

struct Fruit{
  virtual void identify(){std::cout << "Fruit\n";}
};

struct Apple : public Fruit {
  virtual void identify(){std::cout << "Apple\n";}
};

struct Orange : public Fruit {
  virtual void identify(){std::cout << "Orange\n";}
};

int main(){
  Apple  a;
  Orange o;
  Fruit  f;
  Fruit *list[] = {&a, &f, &o};

  list[0]->identify();   // druckt  Apple
  list[1]->identify();   //         Fruit
  list[2]->identify();   //         Orange
}
```

3.2.6 protected

Eine Deklaration im `protected`-Teil einer Klasse bedeutet, dass die Funktion oder das Datenelement sich für einen "normalen" Benutzer der Klasse wie ein(e) private(s) verhält, also nicht zugänglich ist. Allerdings bleibt der Zugriff von abgeleiteten Klassen aus erlaubt. Man sollte mit `protected` für Datenelemente ähnlich sparsam umgehen wie mit `public`, denn auch wenn sich die Benutzung eines Datenelementes "nur" durch die Klassenhierarchie zieht, kann dadurch eine Änderung der Klassenimplementierung extrem erschwert werden (siehe auch 3.2.2)

3.3 Mehrfachvererbung

Das Beispiel des studierenden Lehrers aus Abschnitt 3.2 ist eines für Mehrfachvererbung, denn eine solche Klasse ist sowohl von `Student` als auch `Lehrer` abgeleitet. An den Namen der abgeleiteten Klasse schließt sich eine durch Kommata getrennte Liste der Basisklassen an. Ähnlich werden die Namen der Basisklassen bei der Definition der Konstruktoren aufgereiht. Basisklassen, die in Konstruktoren nicht explizit erwähnt werden, werden durch Aufruf des Defaultkonstruktors initialisiert.

Probleme ergeben sich für den Compiler, wenn in den mehreren Basisklassen Elementfunktionen mit gleichem Namen auftreten. So kollidieren im `StudierenderLehrer` die Elementfunktionen der Basisklasse `Person`, die sowohl von `Student` als auch von `Lehrer` geerbt wird. Ein solcher Konflikt muss explizit aufgelöst werden. Im Beispiel geschieht dies durch die Neudeklaration der Funktion, die dann einfach eine der Basisklassenfunktionen mit Hilfe des Scope-Operators auswählt:

```
#include <string>

class Person {
public:
  std::string surname() const {
    std::string result;
    /* find surname */
    return result;
  }

  void set_name(std::string){ /* ... */ }

  // ...
};

class Student: public Person {
  // ...
};

class Lehrer: public Person {
  // ...
};


class StudierenderLehrer: public Student, public Lehrer {
public:
  // Basisklasseninitialisierung nach Doppelpunkt
  StudierenderLehrer(std::string name)
```

```
    : Lehrer(), Student(){
    set_name(name);
  }

  std::string surname() const { return Lehrer::surname(); }

  void set_name(std::string s) {
    Lehrer::set_name(s);
    Student::set_name(s);
  }

  // ...
};
```

Beim Zusammenführen von Lehrer und Student entsteht natürlich eine ähnliche Vieldeutigkeit bezüglich der Datenelemente von Person (oder i. A. wenn Datenelemente mit gleichem Namen aus verschiedenen Basisklassen in einer abgeleiteten Klasse zusammengeführt werden.) In diesem Fall muss bei Ansprechen eines solchen Datenelementes mittels Bezugsrahmenoperator der Klassenname vorangestellt werden.

3.3.1 `virtual`-Vererbung

Die virtuelle Vererbung dient zur Vermeidung der Duplikation von Datenelementen aus mehrfach geerbten Basisklassen. Wieder im obigen Beispiel erben sowohl `Student` als auch `Lehrer` die Datensektion von `Person`, und diese liegt daher in `StudierenderLehrer` zweimal vor. Nun kann diese Duplikation sinnvoll sein, ist es aber in vielen Fällen nicht, so wie im obigen Fall, weil ja in beiden Klassen letztlich die gleiche Person angesprochen wird.

Werden nun beide Klassen *virtuell* von Person abgeleitet, so ist garantiert, dass auch in weiter abgeleiteten Klassen immer nur eine Kopie des Datensegmentes angelegt wird:

```
#include <string>

class Person{
public:
  Person(std::string name){ /* ... */ }
  // ...
};

class Student : public virtual Person {
public:
  Student(std::string name, int matrikelnummer) :
    Person(name){ /* ... */ }
  //...
};

class Lehrer : public virtual Person {
public:
  Lehrer(std::string name, std::string fach) :
    Person(name){ /* ... */ }
  //...
};

class StudierenderLehrer : public Student, public Lehrer {
```

```
public:
  StudierenderLehrer(std::string name,
                     std::string fach, int matrikelnummer) :
    Person(""), Student(name,matrikelnummer), Lehrer(name,fach) {}
  //...
};
```

Die Initialisierung der virtuellen Basisklasse erfolgt immer in derjenigen Klasse, die in der Hierarchie am weitesten von der Basis entfernt ist und zwar entweder explizit wie im Beispiel oder implizit über einen Defaultkonstruktor.

Wir verweisen für weitere Eigenschaften der virtuellen Vererbung auf die Diskussion in Stroustrup [18].

F: Wie verhindere ich, dass andere Programmierer von meiner Klasse ableiten?
A: In sehr seltenen Fällen kann es nötig sein, weiteres Ableiten von einer Klasse zu vermeiden. Hierzu kann man benutzen, dass Klassen mit privaten Destruktoren oder Konstruktoren nur von `friend`-Funktionen bzw. -Klassen instantiiert werden können:

```
class CantInstantiate
{
  friend class DontDerive;
private:
  CantInstantiate(){}
};

// Von dieser Klasse kann nicht abgeleitet werden
class DontDerive : public virtual CantInstantiate {
public:
  DontDerive() { /* ... */ }
  //...
};
```

3.4 Übungen

Ziel dieser Übung ist das Kennenlernen grundlegender Klasseneigenschaften wie Vererbung und Polymorphismus sowie das Benutzen des Überladens von Operatoren zur Gestaltung der Klassenschnittstelle oder das Überladen von numerischen Operatoren.

3.4.1 Operatoren und Elementfunktionen

(a) Kopieren Sie die in in `EXERCISES/CLASSES/EXAMPLE1` befindlichen Dateien und erzeugen Sie ein ausführbares Programm aus `person_main.cc`. Da es sich um mehrere Dateien handelt, haben wir zur Vereinfachung eine Projektdatei (`Makefile`) bereitgestellt. Sie können durch einfache Eingabe von `make` das Programm übersetzen und die Schritte sehen, die "per Hand" für die Übersetzung erforderlich gewesen wären.

(b) Überladen Sie den Operator `<<`, um eine Ausgaberoutine für die Klasse `Person` bereitzustellen (vgl. Abschnitt 2.7.3). Übergeben Sie die auszugebende Instanz der Klasse `Person` als Referenz einer Konstanten. *Frage:* Was passiert und warum, wenn sie in der Klasse Person die Elementfunktion `age()` nicht als `int age() const` sondern nur als `int age()` deklarieren?

(c) Schreiben sie eine Funktion `marry`, die zwei Personen miteinander verheiratet und entsprechend den Nachnamen verändert (erfinden Sie eine geeignete Regel). Sie haben die Wahl, diese entweder als Elementfunktion `person1.marry(person2)` der Klasse zu implementieren oder als externe Funktion `marry(person1,person2)`. **Frage:** Wie ist der Zugriff auf die privaten Datenelemente der Person im Argument im Falle der Elementfunktion geregelt?

3.4.2 Vererbung

In `EXERCISES/CLASSES/EXAMPLE2` finden Sie ein Beispiel, in dem von einer Klasse `Shape` ein Rechteck `Rectangle` abgeleitet wird. Das Hauptprogramm legt dann ein Feld von `Shape` `*`s an, die es durch Aufruf der virtuellen Funktion `draw(...)` zeichnet. Im Beispiel handelt es sich dabei um 2 Rechtecke, die mit ASCII Grafik auf dem Bildschirm dargestellt werden.

(a) Erweitern Sie das `Rectangle` durch Ableiten zu einer `TextBox`, sodass es auch noch zentriert Text ausgeben kann. Verwalten Sie eine `TextBox` in der gleichen Liste wie ein `Rectangle`.

(b) Untersuchen Sie, was passiert, wenn Sie statt `Shape *`'s `Rectangle *`'s verwalten. Verwenden Sie jetzt ein Feld von `Rectangle`s statt eines Feldes von Zeigern. Ändert sich das Verhalten?

(c) Entfernen Sie jetzt die Basisklasse `Shape` aus `TextBox` und `Rectangle`. Welche Änderungen sind nötig, damit Ihr Programm wieder compiliert? Verwalten Sie wiederum Zeiger und Instanzen der Klassen und untersuchen Sie welchen Einfluss die Qualifizierung von `draw` mit dem Schlüsselwort `virtual` in `Rectangle` auf das Verhalten Ihres Programmes hat.

3.4.3 Klassenschnittstellen, spezielle Funktionen

1. Schreiben Sie eine eigene Klasse `Complex` zur Repräsentation komplexer Zahlen.

   - Real- und Imaginärteil sollten jeweils Zahlen vom Typ `double` sein. Überlegen Sie, welche Elementfunktionen und Konstruktoren für die Benutzung der Klasse notwendig und/oder hilfreich sind. Welche interne Repräsentation ist zur Implementierung am sinnvollsten und effizientesten?

- Überladen Sie den + und/oder * Operator für zwei komplexe Zahlen.
- Schreiben Sie eine Funktion, die die Wurzel aus einer komplexen Zahl zieht: `Complex sqrt(Complex)`

Implementieren Sie, auch wenn das in Ihrem Design nicht nötig sein sollte, Konstruktoren und Destruktor:

- Defaultkonstruktor, Kopierkonstruktor, Destruktor.
- Fügen Sie Ausgabeanweisungen zu den Konstruktoren und dem Destruktor hinzu, um feststellen zu können, wann diese aufgerufen werden. Schreiben Sie ein kurzes Beispielprogramm und testen Sie eigene Ausdrücke und den Aufruf Ihrer `sqrt`-Funktion.

2. Benchmarken Sie Ihre Klasse gegen die von der Standardbibliothek zur Verfügung gestellte Klasse. Hinweis: Unter UNIX kann die `time`-Funktion der Shell genutzt werden:
   `time <command>`

3.4.4 Virtuelle Funktionen (Fortgeschrittene)

Konstruieren Sie zwei Klassen als Basis- und abgeleitete Klasse mit virtuell deklarierten Zuweisungsoperatoren. Die abgeleitete Klasse soll zusätzliche Datenelemente enthalten. Schreiben Sie ein kurzes Hauptprogramm, mit dem Sie nachweisen, dass tatsächlich ein virtueller Aufruf des Zuweisungsoperators generiert wird.
Hinweise: Überlegen Sie sich, welches der Argumenttyp des Zuweisungsoperators in der abgeleiteten Klasse sein muss. Wie können Sie sicherstellen, dass der Zuweisungsoperator der abgeleiteten Klasse wirklich zwei Objekte passenden Typs aufeinander zuweist?
Weisen Sie nach, dass sich Ihre Implementierung im Falle
`a = b = c;`
korrekt verhält.

Kapitel 4

Templates

4.1 Grundidee

Viele Algorithmen lassen sich so formulieren, dass ihre Struktur in weiten Grenzen vom Datentyp unabhängig sind, auf dem sie operieren. So sind beispielsweise Addition und Multiplikation für komplexe Zahlen (oder auch quadratische Matrizen) genauso wie für reelle Zahlen oder ganze Zahlen erklärt. Damit lässt sich beispielsweise eine Exponentialfunktion für jeden Datentyp erklären, für den neben diesen Operationen geeignete neutrale Elemente (die "0" und "1") existieren, sowie eine sinnvolle Multiplikation mit einem reellen Skalar:

$$\exp(\mathbf{x}) = \sum_{n=0}^{\infty} \frac{1}{n!}\mathbf{x}^n \tag{4.1}$$

Zur Berechnung der Partialsummen wird die Addition benötigt, zur Berechnung der einzelnen Beiträge die wiederholte Multiplikation des Elementes $\mathbf{x}$ mit sich selbst sowie die Multiplikation (von links) mit dem Skalar $1/n!$. Die neutralen Elemente werden für die Initialisierung der Partialsumme und des sukzessive berechneten Ausdrucks $\mathbf{x}^n$ benötigt.

Algorithmen oder auch Datenstrukturen, die nur auf solchen gemeinsamen Eigenschaften aufbauen, sollten sich auch *unabhängig* vom speziellen Datentyp formulieren lassen. Erst bei der tatsächlichen Verwendung einer solchen Klasse oder Funktion, nicht bereits aber bei deren Definition oder Deklaration, muss dann der Datentyp festgelegt werden. Die Unterstützung solcher typunabhängiger Programmierung ist eine der herausragenden Eigenschaften von C++, die ein viel allgemeineres Programmdesign erlaubt, als das beispielsweise in C oder FORTRAN möglich ist.[1] Insbesondere in der Numerik ist diese Eigenschaft von C++ wichtiger als der Vererbungsmechanismus, dessen Facetten wir im Zusammenhang mit dem Klassenbegriff im vorigen Kapitel beleuchtet haben.

4.2 Template-Funktionen

Als Beispiele erwähnen wir zunächst so genannte `template`-Funktionen. Bei solchen Funktionen kann der Compiler den benötigten Datentyp durch die Betrachtung des Argumentes bestimmen, mit dem die Funktion aufgerufen wird. Ein typisches Beispiel ist gegeben durch den Programmtext, der das Vertauschen der Werte zweier Variablen bewirkt, gemeinsame Eigenschaft der Variablen muss die Existenz eines Zuweisungsoperators sein. Im zweiten Beispiel wird das Quadrat einer Zahl berechnet, das die Existenz der Multiplikation erfordert. In beiden Fällen ist überdies der Kopieroperator nötig.

[1] Die typunabhängige Programmierung hat C++ von Simula übernommen. Auch ADA verfügt über Unterstützung für generische Programmierung.

```
template<typename T>
void swap(T & a, T & b){
  T tmp = a;
  a = b;
  b = tmp;
}

template<typename T>
inline T sqr(T a){
  return a*a;
}
```

Definition und Deklaration von `template`-Funktionen und Klassen werden durch das Schlüsselwort `template` eingeleitet. Danach erfolgt in spitzen Klammern die Angabe der Templateparameter, in den obigen Beispielen sind das Datentypen, angezeigt durch `typename` (oder auch `class`), für die der (frei wählbare) Platzhalter `T` für den folgenden Programmblock vereinbart wird.

Die Benutzung einer Templatefunktion erfolgt wie die einer normalen Funktion.

```
int main(){
  // ...
  int a=4;
  int b=5;
  swap( a, b );
}
```

Im vorstehenden Beispiel sieht der Compiler, dass das Muster der `swap` Funktion für den Datentyp `T=int` instantiiert werden muss und erzeugt dann den nötigen Objektcode.

Diese Funktionskörper müssen bei der Übersetzung des Programms für den Compiler sichtbar sein. Dazu werden sie üblicherweise in der Deklarationsdatei sowohl deklariert als auch definiert. Im Gegensatz zu Nicht-Template-Funktionen ist dabei kein `inline` Zusatz erforderlich. Die Last, mehrfache Symbole in den Objektdateien zu eliminieren oder gar nicht erst entstehen zu lassen, liegt hier vollständig auf dem Compiler und Lader.

In einigen Fällen ist es dem Compiler nicht möglich, alle benötigten Datentypen an der Argumentliste abzulesen. Dies ist dann der Fall, wenn ein `template` Parameter nur im Rückgabewert oder innerhalb des Funktionskörpers auftritt. In diesem Fall muss man dem Compiler mittels spitzer Klammern `<>` mitteilen, für welchen Typ man eine Instantiierung wünscht.

```
template<typename T>
T random(){
  T  number;
  // compute random number
  return number;
}

int main(){
  double number = random<double>();
  // ...
}
```

Natürlich können auch mehrere Parameter auftreten, oder auch Mischungen von `typename` und nicht-`typename`-Parametern.

```
#include <iostream>
```

```
template <typename S, typename T>
S convert( const T &a ){

  S result(a);
  return result;
}

template<typename T, int N>
T sum(T (&a)[N]){ // liest Zahl der Elemente vom
                  // Feld-Argument ab
  double sum = 0;
  for ( int i=0; i < N; ++i )
    sum += a[i];
  return sum;
}

int main(){
  int    a(3);
  double b = convert<double>(a);  // convert

  double c[] = { 1., 2., 3., 4.};
  std::cout << sum(c);
}
```

4.3 Template-Klassen

Als Beispiel für die Parametrisierung einer Klasse mit einem Typ werden wir eine so genannte "Container"klasse betrachten, die wir zunächst in herkömmlicher Weise programmieren und dann unter Verwendung von `template`s verallgemeinern werden.

Ein Stack ist ein "Stapel" von Datenelementen. Neue Elemente werden "oben" auf dem Stapel abgelegt (push Operation) und auch von oben wieder entfernt (pop Operation). Stacks sind last-in first-out (LIFO) Puffer. Diese werden benötigt etwa bei der Auswertung arithmetischer Ausdrücke, zur Verwaltung temporärer Datenobjekte beim (rekursiven) Funktionsaufruf, bei Programmen, die Benutzereingaben interpretieren müssen (Taschenrechner), oder häufig auch in Simulationen, in denen dynamisch wachsende und kleiner werdende Listen von Daten verwaltet werden müssen.

4.3.1 Herkömmliche Programmierung

Die Schnittstelle der Stack-Klasse ergibt sich aus der Überlegung, wie man die Klasse letztlich benutzen will; d.h., dass wir die Programmierschnittstelle festlegen, bevor wir über deren Implementierung nachdenken.

```
// main.cc:

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

int main(){
  Stack queue;     // leeren Stack erzeugen
  queue.push(1);   // Elemente einfuegen...
  queue.push(2);
```

```
  std::cout << queue.pop() << "\n"; // ...und wiederholen
  std::cout << queue.pop() << "\n";
}
```

Im obigen Falle speichert `Stack` Variablen vom Typ `int` und eine einfache Implementierung, die die obigen Schnittstellenfunktionen realisiert, könnte etwa wie folgt aussehen:

```
// stack.h:
#ifndef STACK_H  // Konstruktion, um mehrfaches
#define STACK_H  //  'include' zu verhindern

class Stack {
public:
  Stack(): top(0){}
  ~Stack(){}
  void push(int a);
  int  pop();
private:
  static const int max_size = 17;
  int data[max_size];
  int top;
};

#endif
```

Zur Speicherung der mit `push` übergebenen Variablen kommt ein gewöhnliches C-Feld mit einer festen Anzahl von Feldelementen zum Einsatz.[2] Die Anzahl der zwischengespeicherten Werte wird in `top` festgehalten.

```
// stack.cc:
#include "stack.h"
#include <assert.h>

void Stack::push(int d){
  assert(top < max_size );
  data[top] = d;
  ++top;
}

int Stack::pop(){
  assert(top > 0 );
  --top;
  return data[top];
}
```

Das assert-Makro hilft bei logischen Fehlern im Programm, die eigentlich nicht auftreten sollten. Falls die genannte Bedingung nicht erfüllt ist, bricht das Programm an dieser Stelle mit einem Coredump ab und kann mit einem Debugger untersucht werden. Dieses Verhalten ist natürlich nur während der Debugphase erwünscht. Später in Produktionsläufen soll dieser Testcode nicht mehr ausgeführt werden. Dazu kann das Programm später unter Setzen des Makros `NDEBUG` übersetzt werden.

[2] Diese Anzahl ist im Beispiel mittels einer als `static const` deklarierten *ganzzahligen* Variablen festgelegt worden, und zwar für alle Klasseninstanzen (`static`) und unveränderbar (`const`). Für `static const` deklarierte ganzzahlige Variablen (`char`, `int`, `enum`) darf die Initialisierung direkt im Klassenkörper erfolgen. Ansonsten muss die Variable wie jede andere statische Klassenvariable nochmals in einer Implementierungsdatei definiert werden und kann an dieser Stelle dann auch initialisiert werden.

```
g++ -DNDEBUG stack.cc
```

Der Programmcode, der durch das assert-Makro geklammert ist, wird dann vollständig aus dem Programm entfernt.

Die Behandlung von laufzeitspezifischen Fehlern, wie fehlerhaften Benutzereingaben, nicht vorhandenen Dateien, etc., sollte *nicht* mit `assert` behandelt werden, sondern mit Programmiermethoden, die wie `if`-Tests, Exceptions, Aufruf einer Fehlerbehandlungsroutine oder die Rückgabe eines speziellen Wertes das Auftreten eines Fehlers anzeigen.

4.3.2 Typunabhängige Programmierung

Der Stack des letzten Abschnitts behandelt nur `int` als Datentyp, neue Stacks für andere Daten müssen durch Umbenennung des `int` an den entsprechenden Stellen, sowie durch den Einbau des Typs in den Klassennamen erreicht werden. C++ nutzt wie bei Funktionen auch hier `template`s, um diese zusätzliche Arbeit im Editor zu sparen und eine kompaktere Notation zu erreichen. Im Programmbeispiel des Stack wird dazu der Typ `int` an den entsprechenden Stellen durch einen benutzergewählten Namen ersetzt, der dem Compiler am Beginn des Programmsegmentes durch die `template`-Anweisung bekannt gemacht wird:

```
// stacktemplate.h:
#ifndef STACKTEMPLATE_H
#define STACKTEMPLATE_H

#include <assert.h>

template<typename T>
class Stack {
  public:
    Stack(): top(0){}
    ~Stack(){}
    void push(T);
    T  pop();
  private:
    static const int max_size = 17;
    T data[max_size];
    int top;
};

template<typename T>
void Stack<T>::push(T d){
  assert(top < max_size );
  data[top] = d;
  ++top;
}

template<typename T>
T Stack<T>::pop(){
  assert(top > 0 );
  --top;
  return data[top];
}
#endif
```

Statt `typename` darf bei der Deklaration des Template-Argumentes auch `class` verwendet werden, was die Schreibarbeit etwas verringert; beide Schlüsselworte deuten an, dass es sich

bei dem Template-Argument um einen Datentyp handeln muss.

Da es sich bei Template-Klassen nur um Muster (oder Familien) für die Erzeugung wirklicher Klassen handelt, muss für den Compiler bei der Übersetzung eines Programmteiles, das Templates benutzt, der gesamte Template-Quellcode sichtbar sein. Am einfachsten wird das garantiert, indem aller zur Klasse gehöriger Code in der Header-Datei untergebracht wird. Eine Trennung in Header- und Implementierungsdatei ist möglich, sofern der Compiler *externe* Template-Instantiierung unterstützt, d.h. in der Lage ist, die Template-Implementierung selbstständig aus den in Frage kommenden `.cc` Dateien zu extrahieren. Diese Funktionalität wird jedoch zur Zeit nur von wenigen Compilern angeboten. Sicher ist daher nur, die Template-Implementierungen in die Header-Dateien zu schreiben.[3]

Um unsere oben definierte Stack-Templateklasse zu benutzen, muss in dem Programmteil, das diesen `Stack` benutzt, das Template-Argument durch einen wirklichen Typ oder eine benutzerdefinierte Klasse ersetzt werden. Dies geschieht, indem wir den benötigten Typ in spitzen Klammern `<>` hinter dem Klassennamen aufführen:

```
// main.cc:

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

int main(){
  Stack<int> queue;
  queue.push(1);
  std::cout << queue.pop() << '\n';
}
```

Ein Beispiel für eine `template`-Klasse, die mehr als ein `template`-Argument hat ist die `map`-Klasse, die in der Standardbibliothek definiert ist. Diese greift über einen Schlüssel vom Typ `T` auf ein Datenelement vom Typ `U` zu und realisiert so eine Abbildung der Schlüssel- auf die Datenelemente.

Die `map`-Klasse ist in der folgenden Weise definiert:

```
// map:

namespace std {

  template<typename T, typename U>
  class map {
    // ...
  };
}
```

und kann bei Bedarf durch

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

[3]Die dabei auftretende Schwierigkeit ist, bei Übersetzung einer eventuellen Templateimplementierungsdatei zu wissen, für welche Typen Templates instantiiert werden müssen. Für dieses Wissen muss aber der Compiler bereits den gesamten übrigen Programmcode kennen, denn Templates können ja überall verwendet werden. Weiterhin muss bekannt sein, in welcher Implementierungsdatei ein Template definiert wird, damit diese Datei gefunden werden kann, wenn aufgrund neu hinzugekommenen Programmcodes eine Neuinstantierung erforderlich wird. Auch wenn wenige Compiler die notwendendigen Strategien implementieren, so wird doch meist ein Modus unterstützt, in dem für jede Header-Datei, in denen Templates deklariert werden, automatisch auch eine gleichnamige Implementierungsdatei eingeschlossen wird. Dieses Vorgehen verfrachtet die Templateimplementierungen in die Deklarationsdatei und wirkt dadurch ähnlich wie eine "echte" externe Instantiierung.

```
main(){
  std::map<std::string,int>  amap;
  // ...
}
```

instantiiert werden.

4.3.3 Ein Stack mit dynamischer Speicherverwaltung

Eine weitere wesentliche Einschränkung der oben angeführten `Stack<T>`-Klasse ist die feste Größe des Feldes, das zur Aufnahme der Datenelemente dient. Wir wollen jetzt dem Benutzer beim Anlegen des Stacks ermöglichen, dessen Maximalkapazität festzulegen.

Wir müssen dazu den Konstruktor und den Destruktor der Klasse abändern und dort `new[]` und `delete[]` aufrufen, um Speicher anzufordern oder wieder freizugeben (siehe 2.8).

```
// stackdynamic.h:
#ifndef STACKDYNAMIC_H
#define STACKDYNAMIC_H

#include <assert.h>

template<typename T>
class Stack {
public:
  Stack(int size=17);
  Stack(const Stack &);
  virtual ~Stack();

  Stack &operator=(const Stack &);

  void push(T);
  T  pop();
private:
  int max_size;
  T*  data;
  int top;
};

template<class T>
Stack<T>::Stack(int size) : max_size(size), top(0){
  data = new T[max_size];
}

template<class T>
Stack<T>::~Stack(){
  delete[] data;
}

template<typename T>
void Stack<T>::push(T d){
  assert(top < max_size );
  data[top] = d;
  ++top;
```

```
}

template<typename T>
T Stack<T>::pop(){
  assert(top > 0 );
  --top;
  return data[top];
}
#endif
```

Um das Verhalten eingebauter Typen zu erhalten, müssen jetzt zusätzlich noch der Zuweisungsoperator und der Kopierkonstruktor überladen werden. Der automatisch generierte Zuweisungsoperator würde bei Zuweisung einer Stack-Instanz auf eine andere die Zuweisungsoperatoren für die einzelnen Elemente aufrufen und die Daten der Zielklasse, auf die zugewiesen wird, überschreiben. Insbesondere würden auch die Zeiger-Datenelemente der Zielklasse einfach überschrieben. Die Information der Lage des in der Zielklasse angeforderten Speichers ginge daher bei dieser Operation verloren. Dieser könnte dann nicht wieder freigegeben werden. Weiterhin würden jetzt auch die `push` und `pop` Operationen der beiden Stacks auf dem gleichen Speicherbereich arbeiten, was zu Datenverlust führen würde, weil verschiedene Elemente über `push`-Funktionen verschiedener `Stack`-Instanzen in denselben Speicherbereich geschrieben würden.

Der Zuweisungsoperator muss daher sicherstellen, dass tatsächlich alle Daten der Ausgangsklasse und nicht einfach der Zeiger auf diese kopiert werden. Wir behalten den allozierten Speicher des linken Operanden bei, wenn dieser groß genug ist, um alle Daten aufzunehmen. Der Code ruft `new` vor `delete` auf, um zu garantieren, dass das Objekt in einem konsistenten Zustand bleibt, falls durch eine Ausnahme (exception) in `new` die Abarbeitung der Funktion abgebrochen wird.[4]

```
template<class T>
Stack<T> &
Stack<T>::operator=(const Stack<T> &rhs){

  T * n_data; // Zielbereich fuer Daten festlegen
  if ( max_size < rhs.max_size )
    n_data = new T[rhs.max_size];
  else
    n_data = data;

  for ( int i=0; i < rhs.top; i++ )
    n_data[i] = rhs.data[i];

  // data loeschen, falls n_data neu ist
  if ( max_size < rhs.max_size ) {
    delete [] data;
  }
  // restliche Variablen kopieren
  data = n_data;
  top  = rhs.top;
  max_size = rhs.max_size;
  return *this;
```

[4] Allerdings kann eine Ausnahmebehandlung in diesem Programmcode auch noch beim Kopieren der einzelnen Datenelemente, die ja selbst wiederum Speicher anfordern könnten. In diesem Fall müssen intelligente Zeigerklassen zum Einsatz kommen, deren Funktion in Abschnitt 6.3 erläutert wird.

```
}

template<class T>
Stack<T>::Stack(const Stack<T> &rhs): max_size(0), data(0), top(0){
  *this = rhs;   // benutzt ueberladenen Zuweisungsoperator
}
```

Wenn man die Zuweisung benutzt, um wie oben den Kopierkonstruktor zu definieren, muss darauf geachtet werden, dass `data` ein bisher noch nicht initialisierter Zeiger ist. Dieser muss auf 0 initialisiert werden, damit der Aufruf von `delete []` in `operator=` nicht mit undefiniertem Ergebnis versucht, einen undefinierten Speicherbereich freizugeben.

Wir weisen noch darauf hin, dass sich beim Übergang von unserer einfachen zu einer Klasse mit dynamischer Speicherverwaltung für den Anwender die Programmierschnittstelle der Klasse nicht geändert hat. Damit haben wir sichergestellt, dass keiner der Programmteile, die `Stack` verwenden, geändert werden muss.

4.4 Template Spezialisierung

In großen Projekten tritt eine Klasse wie die oben implementierte `Stack`-Klasse an vielen Stellen auf und wird mit vielen verschiedenen Typen `T` instantiiert. In den Objektdateien wird dann jedes mal eine Kopie der Klasse für den jeweiligen Typ erzeugt. Häufig ist dieser Programmcode über große Strecken gleich, wenn z.B. Versionen von `Stack` für Datentypen erzeugt werden, die wie Zeiger auf einen Datentyp alle die gleiche Größe aufweisen. Man wünscht sich dann, dass auch nur eine Version der Klasse im Objektcode erscheint.

Weiterhin sollte es möglich sein, für einzelne Datentypen, für die eine besonders laufzeit- oder speichereffiziente Implementierung möglich ist (etwa Stacks von einzelnen Bits, die in Speicherworten — d.h. in der Regel Variablen vom Typ `unsigned int` — zusammengefasst werden können), spezielle Versionen der Templateklasse zur Verfügung zu stellen.

Beide Probleme lassen sich durch *Template-Spezialisierung* lösen. Diese erlaubt entweder für beliebige einzelne Datentypen (wie `bool`) oder für bestimmte Mengen von Datentypen (Zeiger `T*`, Templates wie `std::vector<T>`, Felder `T[]`) Template-Implementierungen, die vom Mutter-Template vollständig unabhängig sind. Im ersten Fall spricht man von vollständiger Spezialisierung, im zweiten von teilweiser (partieller) Spezialisierung.

Wir wollen die nötigen Techniken am Beispiel einer `Stack`-Klasse demonstrieren, die Zeigertypen verwalten soll. Wir stellen dafür zunächst mittels vollständiger Spezialisierung eine Version für `void *` zur Verfügung. Später werden wir diese einsetzen, um per partieller Spezialisierung wie gewünscht eine Klasse für alle Zeigertypen zu konstruieren. Das Vorliegen einer vollständigen Spezialisierung eines Templates wird dem Compiler durch leere spitze Klammern `<>` hinter dem Schlüsselwort `template` angezeigt.

```
// stackpvoid.h:
#include "stackdynamic.h"

// vollstaendige Spezialisierung fuer void *
template<>
class Stack<void *> {
public:
  Stack(int size=17);
  Stack(const Stack &);
  virtual ~Stack();

  Stack &operator=(const Stack &);
```

```
  void push(void *);
  void * pop();
private:
  int max_size;
  void * data;
  int top;
};

template<>
Stack<void *>::Stack(int size) : max_size(size), top(0){
  data = new T[max_size];
}

template<>
Stack<void *>::~Stack(){
  delete[] data;
}

template<>
void Stack<void *>::push(void * d){
  assert(top < max_size );
  data[top] = d;
  ++top;
}

template<>
void * Stack<void *>::pop(){
  assert(top > 0 );
  --top;
  return data[top];
}

template<>
Stack<void *> &
Stack<void *>::operator=(const Stack<void *> &rhs){
  // Implementierung wie fuer Stack<T>
  // ...
  return *this;
}

template<>
Stack<void *>::Stack<void *>(const Stack<void *> &rhs)
  : data(0), max_size(0), top(0)
{
  *this = rhs;
}
```

Jetzt benutzen wir partielle Spezialisierung, um eine Schnittstelle bereitzustellen, das die Funktionen von `Vector` für alle Zeigertypen bereitstellt, intern jedoch mit `void *` arbeitet. Um die Elementfunktionen von `Stack<void *>` nutzen zu können, kann entweder `private` vererbt werden, was die Benutzung der Funktionen der Basisklasse auf die direkt abgeleitete Klasse beschränkt oder ein `Stack<void *>` kann als `private`s Datenelement eingeführt werden. Wir benutzen Vererbung, die in diesem Beispiel die enge Verbindung zwischen

`Stack<T *>` und `Stack<>` noch betont.

```
#include "stackdynamic.h"
#include "stackpvoid.h"

// partielle Spezialisierung fuer alle Zeigertypen
template<class T>
class Stack<T *> : private Stack<void *> {
public:
  Stack(int size=17) : Stack<void *>(size) {}
  virtual ~Stack(){}

  void push(T* p){ Stack<void *>::push(static_cast<void *>(p)); }
  T * pop(){ return static_cast<T *>(Stack<void *>::pop()); }
};
```

Das obige Template definiert nur reine so genannte *forwarding* Funktionen, die während der Compilation sicherstellen, dass die statische Typüberprüfung der Argumente zu Fehlern führt, falls die falschen Zeigertypen an das `Stack<>`-Interface übergeben werden. Die Definition im Klassenkörper bewirkt, dass der Compiler die Elementfunktionen als `inline`-Funktionen sieht und den enthaltenen Code direkt in der aufrufenden Umgebung einfügt. Da die Typumwandlung zweier Zeiger ineinander (hier von `T *` auf `void *` und umgekehrt) in der Regel keine Modifikation der internen Repräsentation erfordert, wird letztlich nur ein Aufruf der entsprechenden Funktionen der Basisklasse verbleiben. Wir erhalten so durch die Mithilfe des Compilers Typsicherheit für alle Typen, verhindern die Duplizierung des Programmcodes und letztlich die gleiche Laufzeiteffizienz wie in der Basisklasse `Stack<void *>`.

Eine partielle Spezialisierung kann ganz allgemein über eine speziellere Kennzeichnung des Typs `T` des "Muttertemplates" erfolgen: So sind etwa `T*`, `T**`, `T[]`, `std::vector<T>`, `std::vector<T*>` Beispiele für eine solche Spezialisierung. Eine partielle Spezialisierung von Nicht-Typ-Templateargumenten ist unmöglich, hier bleibt nur die Möglichkeit der vollständigen Spezialisierung.

```
template<int N, int M, typename T>
Matrix {
  // ...
  T  array[N][M];
};

template<int N, typename T>
Matrix<N,1,T> {
  // ...
  T  vector[N];
};

template<typename T>
Matrix<1,1,T> {
  // ...
  T element;
};
```

Es sei nochmals deutlich darauf hingewiesen, dass zwischen "Muttertemplate" und Spezialisierung keine weitere spezielle Beziehung besteht. Insbesondere wird keine Elementfunktion vererbt und es besteht auch keine "Freundschaft" zwischen den Klassen. Weiterhin ist

es auch erforderlich, externe Funktionen wie z.B. überladene Operatoren für die Spezialisierung neu zu implementieren oder als Alternative dem Compiler geeignete Typumwandlungen anzubieten. Wie bei der Überladung von Operatoren liegt es in der Verantwortung des Programmierers, sicherzustellen, dass sich eine Spezialisierung so verhält, wie das durch das Muttertemplate impliziert wird.

4.5 Template `friends`

Wir haben bereits im Kapitel über Klassen und Vererbung gesehen, dass es manchmal erforderlich ist, Zugriffe auf interne Variablen einer Klasse über eine `friend` Deklaration für eine externe Funktion zuzulassen. Eine solche Funktion muss in der Regel auch als Template implementiert werden, das dann mit der entsprechenden Template-Klasse zusammenarbeiten kann. Beispielsweise könnte ein Ausgabeoperator für unseren `Stack` so aussehen:

```
template<class T>
std::ostream&
operator<<(std::ostream& out, const Stack<T> &stack ){
  for ( int i=0; i < stack.top; ++i )
    std::cout << stack.data[i] << '\n';
  return out;
}
```

Um die Instantiierung dieses Operators für `T` zum Freund der Stack-Klasse zu machen, muss man Folgendes tun. Man sieht zunächst, dass die Implementierung der Templatefunktion schon die Deklaration der Stack-Klasse sehen muss, so dass die Klassendefinition *vor* der Funktionsimplementierung erfolgen muss. Andererseits muss bekannt sein, dass es einen `operator<<` als Template gibt, um diesen in der Klasse als `friend` erklären zu können.

Um diesen Zirkel zu durchbrechen, benötigt man das Mittel der Vorwärtsdeklaration einer Klasse, die im Wesentlichen wie bei einer Funktion dem Compiler nur den Namen einer Klasse bekannt macht. Man kann dann bereits Funktionen oder Klassen deklarieren, deren Argumente nur Zeiger oder Referenzen auf die so bekannt gemachte Klasse enthalten. Unser Problem lässt sich dann so lösen:

```
// Vorwaertsdeklaration der Stack Klasse
template<class T>
class Stack;

// Deklaration des Template-Ausgabeoperators
template<class T>
std::ostream&
operator<<(std::ostream& out, const Stack<T>& stack);

// Definition der Ausgabe als 'friend' in Stack
template<class T>
class Stack {  // Klassendefinition
  // ...
  friend std::ostream &
  operator<< <>(std::ostream &, const Stack<T>& );
  // ...
};
```

Die spitzen Klammern nach den Funktionsnamen sind dabei nötig, um die Funktion als Template-Funktion kenntlich zu machen. In die spitzen Klammern hätte man noch `T`

schreiben können, um anzudeuten, dass es sich um die Instantiierung für `T` handeln soll, was aber bereits aus dem zweiten Argument klar wird.

Man mache sich auch den Unterschied zu den folgenden zwei anderen möglichen Deklarationen klar:

```
template <class T>
class Stack {
  //...
  friend std::ostream&
  operator<< (std::ostream &, const Stack<T> &);

  template<class S>
  friend std::ostream&
  operator<< (std::ostream &, const Stack<S> &);
};
```

Die erste macht eine reguläre Funktion zum Freund, die keine Template-Funktion ist, sondern als zweites Argument "zufällig" einen `const Stack<T>&` erwartet, etwa

```
std::ostream&
operator<<(std::ostream& out, const Stack<double>& stack);
```

Die zweite Deklaration macht gleich eine ganze Familie von Ausgabeoperatoren zu Freunden, nämlich alle Instantiierungen des Template-Ausgabeoperators, den wir oben beschrieben haben, auch zu anderen Typen als `T`, möglicherweise solchen, die zu `T` in keiner Beziehung stehen.

4.6 Traits

Schreibt man numerische Templateprogramme oder Templatecode im Allgemeinen, so muss man oft auf typabhängige Konstanten wie `DBL_MAX` oder `INT_MAX` oder `FLT_MAX` zugreifen, deren Namen vom verwendeten Typ abhängen. Wie kann man in Templatecode, bei dem der Typ der Variablen, für den instantiiert wird, ja noch gar nicht feststeht, auf solche typabhängigen Informationen zugreifen?

Eine Lösung für dieses Problem stellen die so genannten `traits` dar. Traits sind eine allgemeine Technik, um Typinformation auf Werte, andere Typen, Funktionen, etc. abzubilden. Die grundsätzliche Idee greift wieder auf Template-Spezialisierung zurück und sei hier am Beispiel der mit `#include <limits>` bereitgestellten `std::numeric_limits` erläutert.

Zunächst wird ein allgemeines Template für alle Typen bereitgestellt:

```
namespace std {

template<class T>
struct numeric_limits {
  inline static T max();
  static const bool is_integer = false;
  // ...
};

}
```

Das Fehlen der Implementierung für die deklarierte Funktion bewirkt, dass das Programm nicht vollständig kompiliert werden kann, sollte jemals für `T` ein Typ eingesetzt werden, der

nicht — wie im Folgenden beschrieben — unterstützt wird. Der Benutzer muss dann selbst Sorge tragen, dass eine geeignete Spezialisierung bereitgestellt wird, in Analogie zu der in der Standardbibliothek. Für konkrete Typen wird jetzt spezialisiert und der Maximalwert für den jeweiligen Typen zurückgeliefert. Im Beispiel ist noch ein logischer Datentyp vereinbart, um zwischen Fließkomma- und ganzzahligen Typen unterscheiden zu können.

```
#include <float.h>  // systemabhaengige Konstanten
#include <limits.h>

namespace std {

// Spezialisierung fuer alle bekannten Typen:
template<>
struct numeric_limits<double>
  inline static double max(){ return DBL_MAX; }
  static const bool is_integer = false;
  // ...
};

template<>
struct numeric_limits<float>
  inline static float max(){ return FLT_MAX; }
  static const bool is_integer = false;
  // ...
};

template<>
struct numeric_limits<int>
  inline static int max(){ return INT_MAX; }
  static const bool is_integer = true;
  // ...
};

}

template<class T>
T max_representable(){
  return std::numeric_limits<T>::max();
}
```

Für nicht-integrale Konstanten bevorzugen wir die Substitution durch `inline`-Funktionen, weil sie keine vom Klassenkörper getrennte Definition erfordern, wie es für konstante aber nicht-integrale Datentypen nötig wäre. Trotzdem wird letztlich durch den Compiler aufgrund der einfachen Struktur der Funktionen direkt der benötigte Wert im aufrufenden Code eingesetzt.

4.7 Überladen von Template-Funktionen

Templatefunktionen lassen sich mit regulären und mit anderen Templatefunktionen überladen. Der Compiler folgt zwar recht komplexen Regeln, um zu bestimmen, welche Funktion letztlich aufgerufen wird, aber das Ergebnis stimmt in den allermeisten Fällen mit den intuitiven Erwartungen überein.

```
#include <complex>
```

```
using std::complex;

double exp(double); // regulaere Exponentialfunktion
float  exp(float);  // ueberladene Version fuer float

template<class T>
T exp(const T &);   // Templatefunktion

template<class T>   // Templatefunktion
complex<T> exp(const complex<T> &);

complex<double>     // regulaere Funktion
exp( const complex<double> & );

int main(){
  complex<float> a;
  exp(a);           // ruft complex<T> exp(const complex<T> &);
                    //    mit T=float
  exp(1.2);         // ruft double exp(double)
  exp(1);           // ruft T exp(const T&); T=int
  exp<double>(1);   // ruft T exp(const T&); T=double
  return 0;
}
```

Zur Bestimmung, welche Funktion bei jedem Funktionsaufruf letztlich aufgerufen wird, geht der Compiler in drei Schritten vor:

Auswahl der in Frage kommenden Funktionen Hierzu wird eine Liste sogenannter *viable functions* erstellt, die zunächst alle regulären Funktionen mit passendem Namen umfassen, deren Argumente mit maximal einer Standard- (d.h. compilerinternen oder C-) und einer benutzerdefinierten Typumwandlung oder -anpassung aus den aktuellen Argumenten erreichbar sind. Im Falle des Aufrufs `exp(a)` im obigen Beispiel ist dies nur `exp(complex<float>)`, da `complex<float>` keine Konversion nach `float`, `double` oder `complex<double>` erlaubt. Dann werden die Templatefunktionen daraufhin geprüft, ob sich das Template-Argument aus dem aktuellen herleiten lässt. Bei "spezialisierten" Argumenten wird so die Menge der möglichen Funktionen eingeschränkt. Alle Funktionen werden dann mit dem jeweilig deduzierten Argumenttyp in die Liste der *viable functions* eingetragen. Hier kann es durchaus vorkommen, dass die Argumente verschiedener Versionen der Templatefunktionen gleich sind:

```
template<class T>
  void f( T ) { /* ... */ }

template<class T>
  void f( T* ){ /* ... */ }

int main(){
  int i;
  f( &i );  // f<int *>(int *) or f<int>(int *)?
}
```

Auswahl der besten Funktion Danach wird die "beste" Funktion aus dieser Liste bestimmt. Dazu werden die Konversionen verglichen, die an den aktuellen Parametern

durchgeführt werden müssen, damit sie mit den Argumenttypen der oben gefundenen Funktionen übereinstimmen.

Dazu werden die Typumwandlungen und -anpassungen in Klassen eingeteilt: es handelt sich um eine *Übereinstimmung (exact match)*, wenn (i) keine Umwandlung nötig ist, (ii) nur `const` oder `volatile` hinzugefügt werden müssen, oder (iii) nur eine Umwandlung eines Feldtyps oder einer Funktion in einen Pointertyp erforderlich ist. Es handelt sich um eine *Typanpassung (type promotion)*, wenn ein kurzer integraler Datentyp ohne Genauigkeitsverlust in entweder `int` oder `unsigned int` bzw. `float\ nach \verb`double— überführt wird. Von einer *Typumwandlung (conversion)* sprechen wir, wenn das Bitmuster des Typs geändert beim Übergang in den Zieldatentyp geändert werden muss bzw. möglicherweise ein Genauigkeitsverlust auftritt, wie etwa bei dem Umwandlungen von längeren in kürzere Datentypen, von und nach `bool` oder bei Umwandlungen von `double` nach `int`. Es zählen hierzu auch die Umwandlungen von einen Zeigertyp in einen anderen.

Eine Typumwandlungssequenz besteht aus einer oder mehreren Umwandlungen oder Anpassungen aus den genannten Klassen. Eine Sequenz ist "besser" als eine andere, wenn sie weniger Umwandlungen benötigt oder die beteiligen Umwandlungen aus "genaueren" Kategorien stammen. So ist die Sequenz `const char []`, `const char *` (Feld-Pointer), `char *` (`const`-Anpassung) besser als `const char []`, ..., `void *`, weil die letztere Sequenz sowohl länger ist, als auch eine Umwandlung zweier Zeigertypen ineinander verlangt, die den Rang einer Typumwandlung besitzt.

Die beste Funktion ist gefunden, wenn keine nötige Argumentkonversion schlechter ist als die zu einer anderen Kandidatfunktion und mindestens eine besser.

Nicht-Template-Funktionen werden gleichermaßen passenden Templatefunktionen vorgezogen. Passen mehrere Template-Funktionen, so wird diejenige ausgewählt, deren Argument weitergehend spezialisiert ist. Im Beispiel des vorigen Abschnitts wird etwa `void f(T*)` instantiiert und aufgerufen.

Zugänglichkeit Sollte keine "beste" Funktion gefunden werden können, so meldet der Compiler ein nicht-eindeutiges Ergebnis und bricht den Übersetzungslauf ab. Anderenfalls schaut sich der Compiler erst jetzt an, ob ein Funktionsaufruf letztlich wirklich erlaubt oder möglich ist. Es kann so z. B. vorkommen, dass die ausgewählte Funktion eine private Elementfunktion einer Klasse ist und gar nicht aufgerufen werden darf, obwohl vielleicht eine andere Kandidatfunktion existiert, die vielleicht zugänglich wäre, aber deren Argumenttypen nicht so gut passen.

Man beachte, dass in dieser Auswahl Funktionen mit Referenzargumenten nicht von solchen mit "by value" Argumenten unterschieden werden. Der folgende Code produziert eine Fehlermeldung des Compilers bezüglich nicht eindeutiger Überladung von Funktionen:

```
#include <math.h>

double exp( double a );
double exp( double &a );

int main(){
  double a;
  exp( a );  // ERROR
}
```

4.8 Template-Elementfunktionen

In einigen Fällen ist es praktisch, die Möglichkeit zu haben, Elementfunktionen einer Klasse als Template-Funktionen zu gestalten. Beispiele sind häufig Typumwandlungsfunktionen, entweder als Konstruktoren einer Klasse oder als "Extraktoren" von Information. Nehmen wir beispielsweise die Definition eines Vektors fester Länge, der hier einfach aus 2 `doubles` bestehen soll. Ein solcher Vektor kann im Prinzip aus jedem Datentyp konstruiert werden, der Indizierung mit `[]` zulässt. Eine entsprechende Klasse könnte dann so aussehen:

```
class AShortVec{
public:
  template<class T>
  AShortVec( const T& init ): x(init[0]), y(init[1]){}

   // ... Elementfunktionen
private:
   double x, y;
};
```

An der Position von `T` kann sowohl ein Zeiger als auch ein Feld oder auch eine Klasse mit überladenem `operator[]` stehen. Trotz der Flexibilität des obigen Konstruktes wird der Compiler die Erzeugung von Code verweigern, wenn der Typ der Elemente von `T` nicht kompatibel mit dem der Elemente von `ShortVec` ist.

Weiterhin ist wichtig zu beachten, dass Template-Elementfunktionen nicht virtuell sein können. Dies hängt i.W. damit zusammen, dass virtuelle Funktionstabellen nur Funktionen aufnehmen können, deren Typen zur Übersetzungszeit feststehen. Da der Compiler jedoch bei Template-Elementen in abgeleiteten Klassen unter Umständen zusätzliche Elemente generieren muss, würde dies auch Instantiierungen entsprechender Template-Elemente der Basisklasse nach sich ziehen. Der zur Basisklasse gehörige Programmcode müsste dann neu erzeugt werden und es müsste über Übersetzungseinheiten hinweg sichergestellt sein, dass die virtuelle Funktionstabelle konsistent ist. Dieses Problem scheint zwar lösbar, aber trotzdem wurde in den gegenwärtigen C++-Implementierungen vor dieser Komplexität zurückgeschreckt.

4.9 Expression Templates

Expression Templates sind der Kern einer Technik, die es erlaubt, den Compiler zur Übersetzungszeit als Interpreter zu betreiben und Berechnungen ausführen zu lassen. Das Ergebnis dieser Berechnungen sind Zahlen oder Programmcode. Daher heißen die zugehörigen Template Deklarationen häufig auch Template-Metaprogramme. Diese Technik ist komplex, aber extrem interessant, weil sich damit die Erzeugung von temporären Objekten zur Laufzeit vermeiden lässt und sehr effizientes Kompilat entstehen kann. Es versteht sich allerdings, daß diesem Vorteil hoher Laufzeitperformance (Abb. 4.9) als Nachteil die extrem komplexe Bibliotheksprogrammierung und die langen Übersetzungszeiten (siehe Abb. 4.9) gegenüberstehen.

Für eine Einführung in diese Art der Programmierung ist der Artikel von Todd Veldhuizen [20] gut geeignet. Aus diesem Artikel sind auch die folgenden Beispiele entnommen.

4.9.1 Template Rekursion

Hier ist zunächst ein einfaches Beispiel, das die Fakultät einer ganzen Zahl zur Übersetzungszeit berechnet. Man beachte, dass neben Datentypen auch Zahlen oder andere zur Übersetzungzeit feststehende Ausdrücke als Template Parameter auftreten dürfen.

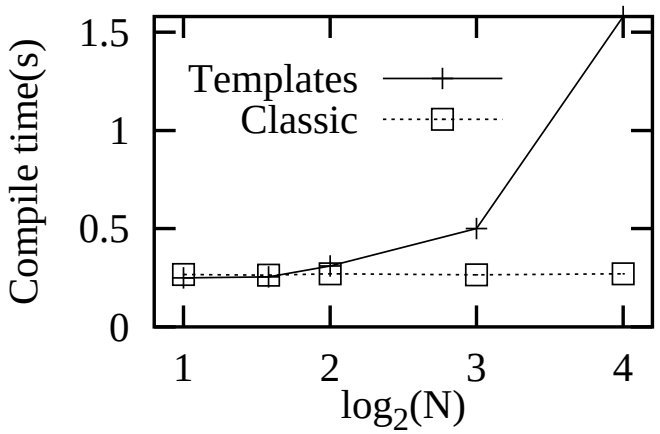


Abbildung 4.1: Compile Zeiten für Compiler "cxx" auf einer DEC Alpha Maschine

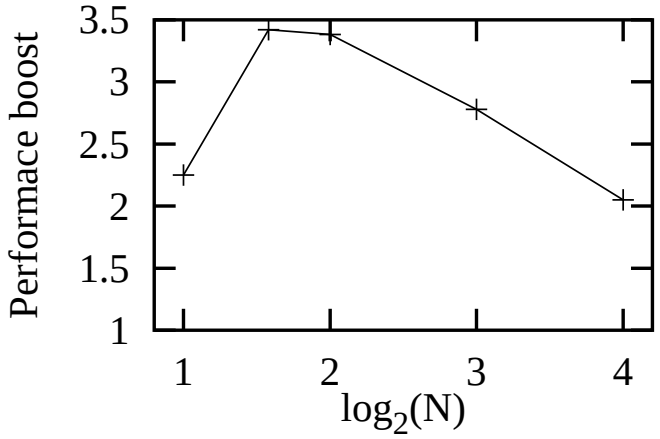


Abbildung 4.2: Performancegewinn einer Template-Version des Bubblesort Algorithmus im Vergleich zu einer "klassischen" Doppelschleifen-Lösung. Es werden 10^6 Sortierungen eines Integer Feldes mit einer Länge von $N = 2,4,8$ und 16. Für $N = 16$, war die Zeit (1.01 ± 0.04) s. Für $N = 2$, spielt die Laufzeit der äußeren `for`-Schleife wahrscheinlich eine wichtige Rolle.

```
template<int N>
class Factorial {
public:
   enum { value = N * Factorial<N-1>::value };
};

template<>
class Factorial<1> {
public:
  enum { value = 1 };
};
```

Die Template-Klasse `Factorial` stellt den Wert über das Element `value` zur Verfügung. Bei der Auswertung des `enum` wird auf die um eins kleinere Fakultät zurückgegriffen. Damit die Rekursion abbricht, ist die Klasse für den Wert 1 spezialisiert. Im Programmtext kann man durch `const int fak5 = Faktorial<5>::value` die Fakultät berechnen lassen.

Schleifenoptimierung

Als Beispiel für die Ziele der Metaprogrammierung in der Numerik sei hier die Optimierung von Schleifen demonstriert. Ein Typ, der "physikalische" Vektoren, etwa 2 oder 3 Gleitkommazahlen für Ort oder Geschwindigkeit und deren vektorielle Addition in Simulationen repräsentiert, könnte vielleicht so definiert werden:

```
template <class T, int DIM>
struct AShortVec {
  AShortVec(){}
  AShortVec(const T& default){
    for(int i=0; i <DIM; ++i)
      data[i] = def;
  }

  inline AShortVec &
  operator+=(const AShortVec &rhs);

private:
  T data[DIM];
};

template <class T, int DIM>
AShortVec<T,DIM> &
AShortVec<T,DIM>::operator+=(const AShortVec<T,DIM> &rhs){
  for (int i=0; i < DIM; ++i )
     data[i] += rhs.data[i];
  return *this;
}
```

Ein Programm, das diesen Typ nutzt, um damit etwa die diskretisierte Bewegung vieler Teilchen zu simulieren, wird die Orte von Zeitschritt zu Zeitschritt ändern müssen.

```
void timestep(int n, AShortVec<double,3> pos[],
                     AShortVec<double,3> delta[] ){
  for( int i=0; i < n; ++i )// i bezeichnet Teilchen
    pos[i] += delta[i];     // implizite Schleife ueber Komponenten
}
```

In dieser Routine werden vektorielle Werte `delta[i]` auf die Teilchenpositionen `pos[i]` aufaddiert. Die Addition ist "versteckt" in der Verwendung des `operator+=`.[5] Vorteil der obigen Routine ist die klare Form, die durch die Verwendung von `AShortVec` erzielt wird. Ihr Nachteil liegt darin, dass `operator+=` für jede einzelne Vektoraddition die Verwaltung einer Schleifenvariablen erfordert. Aufgrund der kurzen Schleifenlängen muss diese Variable sehr häufig initialisiert werden, was möglicherweise langsameren Code bewirkt als das folgende, nicht-objektorientierte Vorgehen, das die Schleifenreihenfolge umdreht.

```
const int N = 42;

void timestep(int ndim, double pos[][N], double delta[][N]){
  for ( int idim=0; idim < ndim; ++idim )
    for ( int i=0; i < N; ++i )
      pos[idim][i] += pos[idim][i];
}

int main(){
  double pos[3][N], delta[3][N];
  //...
  timestep(3, pos, delta);
}
```

In beiden Fällen ist der Feldzugriff so gestaltet, dass die Datenelemente in lückenloser Reihenfolge nacheinander angesprochen werden, was für die Vektorisierung des Codes (auf Großrechnern) bzw. den Zugriff auf Cache-Speicher (auf Arbeitsplatzrechnern) vorteilhaft ist. Im nicht-objektorientierten Vorgehen ist allerdings die Zahl der Schleifeninitialisierungen deutlich geringer.

Die kurze Schleife im objektorientierten Code ließe sich allerdings eliminieren, indem wir `AShortVec` für beispielsweise `N=3` spezialisieren. Dann könnte der Additionsoperator so implementiert werden:

```
template <class T>
AShortVec<T,3> &
AShortVec<T,3>::operator+=(const AShortVec<T,3> &rhs){
  data[0] += rhs.data[0];
  data[1] += rhs.data[1];
  data[2] += rhs.data[2];
  return *this;
}
```

Durch das Inlining würde die Rückgabe von `*this` eliminiert. Diese Vorgehensweise ergibt dann zwar potentiell schnellen Code, aber die Template-Klasse muss für alle denkbaren Werte von `N` spezialisiert werden.

Hier ist jetzt ein abgekürztes Beispiel für einen kurzen numerischen Vektortyp, der ohne Schleifen in Operatoren auskommt und der für beliebige Dimension benutzbar ist. Wir realisieren ihn durch rekursives Einfügen von Datenelementen.

```
template<class T, int DIM>
struct ShortVec {
  ShortVec(){}
  ShortVec(const T& def ): data(def), more_data(def){}
private:
```

[5] Die Verwendung des `operator+=` erspart dabei die Erzeugung temporärer Werte, die bei `operator+` nötig wäre (vgl. 71)

```
  T data;
  ShortVec<T, DIM-1> more_data;
};

template<class T>
struct ShortVec<T,1> {
  ShortVec(){}
  ShortVec(const T& default): data(default){}
private:
  T data;
};
```

Diese Rekursion bricht ab, weil der Compiler für `DIM`=1 auf eine Spezialisierung stößt. Dem Datentyp wurde in jeder Dimension genau ein Datenelement hinzugefügt. Der Default-Konstruktor erlaubt uns, Felder von `ShortVec` anzulegen und der Konstruktor mit einem Element erlaubt Initialisierung der Datenelemente auf 0 oder einen anderen festen Wert. Die Implementierung des `operator+=` kann jetzt so erfolgen:

```
template<class T, int DIM>
struct ShortVec {
  // ..
  ShortVec &
  operator+=( const ShortVec & rhs ){
    data      += rhs.data;
    more_data += rhs.more_data;
    return *this;
  }
};

template<class T>
struct ShortVec<T,1> {
  // ..
  ShortVec &
  operator+=( const ShortVec & rhs ){
    data      += rhs.data;
    return *this;
  }
};
```

Der Aufruf von `operator+=` für die Vektoraddition von `more_data` bewirkt, da dieser Operator `inline` definiert ist, sozusagen ein Einsetzen des Operators der Klasse zur um 1 verminderten Dimension. Der Compiler sieht, dass außer des `return` im ersten Aufruf keines benötigt wird und kann die entsprechenden Anweisungen eliminieren. Effektiv entsteht Programmcode, in dem für jedes skalare Datenelement der Vektoren ein `+=` ausgeführt wird. Die sonst übliche Schleife ist hier durch eine rekursive, zur Übersetzungszeit stattfindende Inline-Expansion von kurzen Funktionen ersetzt worden.

Die Implementierung weiterer Funktionalität folgt dem obigen Beispiel.

4.9.2 Metaprogramm-Kochrezept

Für eine "echte" Programmierung, braucht man neben der Rekursion natürlich noch weitere Kontrollstrukturen. Hier folgt eine kurze Liste, wie sich z.B. die üblichen Laufzeitkonstrukte `if else` und `do while` in Metaprogrammen ausdrücken lassen, die zur Compilezeit ausgewertet werden.

if-Anweisung Die einfache Alternative, deren "Laufzeitvariante" in C++-Code etwa so ausgedrückt würde

```
if (condition)
  statement1;
else
  statement2;
```

lässt sich in einem Metaprogramm in der folgenden Form programmieren:

```
// Klassendeklarationen

template<bool C>
class _name { };

template<>
class _name<true> {
public:
  static inline void f()
  { statement1; }         // true case
};

template<>
class _name<false> {
public:
  static inline void f()
  { statement2; }         // false case
};
```

Damit kann dann Code, der bereits zur Compilezeit ausgewertet wird, durch die entsprechende Spezialisierung der Klasse `_name` ausgedrückt werden. Anstelle von `f()` wird dann entsprechend dem Template-Argument die passende Funktion eingesetzt:

```
// Ersatz fuer eine 'if/else' Anweisung:
  _name<condition>::f();
```

Schleifen: Schleifen mit zur Compilezeit bekannter Obergrenze

```
int i = N;
do {
  statement;
} while (--i > 0);
```

lassen sich übersetzen in eine Form, die ohne das Schleifenkontrollelement auskommt und stattdessen `statement` mit entsprechend variierter Schleifenvariable wiederholt ausführt:

```
// Class declarations
template<int I>
class _loop {
private:
  enum { go = (I-1) != 0 };
```

```
public:
  static inline void f()
  {
    statement;
    _loop< go ? (I-1) : 0 >::f();
  }
};

template<>
class _loop<0> {
public:
  static inline void f(){}
};
```

Die Spezialisierung der Klasse `_loop` für das Argument 0 liefert wieder die Basis für den Abbruch dieser Rekursion zur Übersetzungszeit. Die Anweisung

```
_loop<N>::f();
```

wird dann durch eine Aufreihung von `statement;` ersetzt, wie sie sonst zur Laufzeit durch das o.g. `while`-Konstrukt der Fall wäre.

4.10 Template-Basisklassen

Prinzipiell ist es möglich, Templateklassen auch als Basisklassen zu verwenden.

```
template <class T>
struct A {
  // ...
};

template <class T>
struct B : public A<T> {
  // ...
};

template <class T>
struct C : public T {
  // ...
};
```

Zur Motivation und zur Demonstration von Problemen greifen wir auf eines der bereits diskutierten Beispiele zurück. Wie wir oben bereits erwähnt haben, handelt es sich bei der Spezialisierung einer Templateklasse um eine völlig neue Klasse, die für den Compiler außer der Tatsache der Spezialisierung keine besondere Beziehung zum "Muttertemplate" hat. Manchmal wünscht man sich jedoch, nur kleine Teile einer Template-Klasse neu implementieren zu müssen. Wir haben in Abschnitt 4.9.1 am Beispiel der Vektoraddition gesehen, dass möglicherweise nur eine Funktion, dort `operator+=` aus Performancegründen, für eine Spezialisierung, dort `N=3` implementiert werden muss.

Leider ist es nicht möglich, die Spezialisierung durch Ableitung aus dem Muttertemplate zu gewinnen

```
template <class T, int DIM>
```

```
struct AShortVec {
  // ...
};

template <class T>
struct AShortVec<T,3> : public AShortVec<T,3> {  // FEHLER
  // ...
};
```

denn die Spezifikation der Basisklasse ist nicht eindeutig. Ist `AShortVec<T,3>` eine Spezialisierung, die hier dann rekursiv in die Typdeklaration einginge, oder um die Instantiierung des Muttertemplates für `DIM=3`?

Man kann jedoch die Gemeinsamkeiten in eine Basisklasse ausfaktorisieren und dann versuchen, durch Ableitung sowohl in das Muttertemplate als auch in die Spezialisierung einzubringen.

```
template <class T, int DIM>
struct AShortVecBase {
private:
  T data[DIM];
};

template <class T, int DIM>
struct AShortVec : public AShortVecBase<T,DIM> {
  AShortVec(const T &def){
    for (int i=0; i < DIM; ++i )
      AShortVecBase<T,DIM>::data[i] = def;
  }

  AShortVec &
  operator+=(const AShortVec &rhs){
    for (int i=0; i < DIM; ++i )
       AShortVecBase<T,DIM>::data[i] += rhs.data[i];
    return *this;
  }
};

template <class T>
struct AShortVec<T,3> : public AShortVecBase<T,3> {
  AShortVec(const T &def){
    AShortVecBase<T,3>::data[0] =
        AShortVecBase<T,3>::data[1] =
        AShortVecBase<T,3>::data[2] = def;
  }

  AShortVec &
  operator+=(const AShortVec &rhs){
    AShortVecBase<T,3>::data[0] += rhs.data[0];
    AShortVecBase<T,3>::data[1] += rhs.data[1];
    AShortVecBase<T,3>::data[2] += rhs.data[2];
    return *this;
  }
};
```

Diese Lösung ist möglich, hat aber leider den Nachteil, dass die Symbole der Template-Basisklasse mittels Scope-Operator importiert werden müssen, denn der Compiler ist bei Template-Basisklassen nicht dazu gezwungen, die Template-Basisklasse zu instantiieren, um zu sehen, welche Symbole bereitgestellt werden. Das bedeutet, dass innerhalb der abgeleiteten Klasse nicht nur Datenelemente aus der Basisklasse, sondern auch Elementfunktionen mittels Scope-Operator angesprochen bzw. per `using`-Direktive in den Namensraum importiert werden müssen:

```
template <class T, int DIM>
struct AShortVecBase {
  double norm(){ /* compute norm */ }  ;
private:
  T data[DIM];
};

template <class T>
struct AShortVec<T,3> : public AShortVecBase<T,3> {
  // keine () noetig, importiert alle Symbole norm
  using AShortVecBase<T,3>::norm;

  // using noetig
  double length(){ return sqrt(norm()); }  // ok
  // alternativ:
  // double length(){ return sqrt( AShortVecBase<T,3>::norm() ); }
};

void f(){
  AShortVec<T,3> a;
  a.norm();  // ok auch ohne using
}
```

Dieses Verfahren ist nur nötig, wenn Funktionen und Datenelemente aus der Basisklasse innerhalb der abgeleiteten Klasse angesprochen werden. Nachdem die abgeleitete Klasse instantiiert wurde, etwa durch den Aufruf in der Funktion `f()`, sind alle Symbole, auch die vererbten, bekannt und können angesprochen werden. Viele Compiler unterstützen in Erweiterung des C++-Standards allerdings auch die Instantiierung der Basisklasse bei der Symbolsuche, die die unschöne Asymmetrie zum Verhalten von Nicht-Templateklassen bei Ableitung beseitigt.

4.11 Übungen

Ziel dieser Übungen ist das Erlernen elementarer Template-Techniken und der Erwerb grundlegender Kenntnisse über die Verwendung dynamischer Speicherverwaltung.

4.11.1 Stack/-Stapelspeicher

In `EXERCISES/TEMPLATES/STACK` finden Sie Programmcode für eine Stack-Klasse, die `int`s verwaltet.

(a) Ändern Sie den Programmcode so, dass beliebige Typen verwaltet werden können und testen Sie die Klasse für `int` und `double` wie in `main.cc` für die reine Integer-Version angedeutet.

(b) Führen Sie in der Funktion `push()` Code ein, der dafür sorgt, dass sich der Stack geeignet vergrößert, wenn ein Element eingefügt werden soll, die Kapazität jedoch bereits erschöpft ist.

(c) *bei Interesse:* Definieren Sie eine neue Klasse ohne Datenelemente, deren Zuweisungsoperator Sie durch eine Deklaration in der `private:` Sektion unzugänglich machen. Studieren Sie die Fehlermeldungen, die auftreten, wenn Sie versuchen, einen Stack von solchen Objekten anzulegen.

4.11.2 Templatefunktionen

(a) Schreiben Sie eine Template-Exponentialfunktion `exp`, indem Sie die Reihe

$$\exp(x) = 1 + \frac{1}{1}x + \frac{1}{1 \cdot 2}x^2 + \frac{1}{1 \cdot 2 \cdot 3}x^3 + \dots \tag{4.2}$$

bis zum zehnten Glied aufsummieren. Schreiben Sie ein Hauptprogramm, das diese Funktion für `complex<float>` instantiiert und vergleichen Sie das Ergebnis mit der eingebauten komplexen Exponentialfunktion `std::exp` für die Werte i und $0.1 + 2i$.
(b) Stellen Sie sicher, dass Ihre Version der Exponentialfunktion die minimal möglichen Annahmen über die strukturellen Eigenschaften von `T` macht. Es sollte ausreichen, dass für `T` eine Multiplikation von links mit Skalaren und mit anderen `T` sowie eine Addition mit den üblichen neutralen Elementen definiert ist.
(c) Lösen Sie das System gewöhnlicher Differentialgleichungen

$$y' = Ay, \quad y = (y_0(x), y_1(x)), \quad y(0) = (1, 0), \quad A = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \tag{4.3}$$

direkt durch Anwendung der Lösungsformel

$$y = y(0)\exp(Ax) \tag{4.4}$$

für repräsentative Werte von x. Hierzu können Sie entweder die "fertige" Matrixklasse `PTZ_Matrix` verwenden, die aus einem der letzten Kurse hervorgegangen ist, oder Sie können eine "kleine" Matrixklasse, die die Bedingungen unter (a) erfüllt, selber schreiben (s.u.)

4.11.3 Matrix-/Vektorklasse

Implementieren Sie eine Klasse, die kurze physikalische Vektoren (etwa Ort oder Geschwindigkeit) mit fester Dimension repräsentieren kann und die sich weitgehend intuitiv benutzen lässt. Der interne Datentyp soll sich wie die Dimension der Vektoren zur Compilezeit frei wählen lassen.

Kapitel 5

Die C++ Standardbibliothek

5.1 Standard Template Library

Die Standard Template Library (STL) ist Bestandteil der C++ Standardbibliothek. Als zentrale Klassen bietet sie Container-Datenstrukturen an, die als Templateklassen implementiert sind. Dies sind Klassen, die in der Lage sind, Objekte zu speichern und die Schnittstellen zur Verfügung stellen, die es den STL-Algorithmen erlauben, effizient zu sortieren, wiederaufzufinden, einzufügen, zu löschen etc. Jeder Container bietet darüber hinaus spezielle eigene Stärken, die über Elementfunktionen bereitgestellt werden. Der konzeptionell einfachste Container `vector<>` ist einem dynamisch alloziierten Feld nachempfunden, das nicht immer vollständig genutzt wird, so dass in einfacher Weise Elemente angefügt werden können. Das Einfügen von Elementen erfolgt über Elementfunktionen von `vector<>`, so dass im Bedarfsfall zusätzlicher Speicher alloziert werden kann. Wir werden sehen, dass die Container eine einheitliche Schnittstelle bereitstellen, die zur einer vom speziellen Container und dem verwalteten Datentyp unabhängigen Implementierung von Algorithmen (als Templatefunktionen) dient (`find`, `sort`, `remove`, etc.).

Nicht alle Objekte können in Containerklassen verwaltet werden. Geeignete Objekte müssen bestimmte (geringe) Anforderungen an die mit ihnen zugelassenen Operationen erfüllen. Die Benutzung der meisten Container geht von einer "by value"-Semantik aus, was erfordert, dass sich die verwalteten Objekte wie die eingebauten einfachen Datentypen defaultkonstruieren, kopieren und aufeinander zuweisen lassen müssen. Container wie `set<>` oder `map<,>` erfordern weiterhin, dass sich durch Auswertung einer Kleiner-Relation (ausgedrückt durch `operator<`) eine Anordnung der Elemente erreichen lässt.

Die Standardbibliothek unterteilt in sequentielle Container, die Daten im Speicher im Wesentlichen linear aneinanderreihen und in assoziative Container, die zusätzlich erfordern, dass Daten (oder assoziierte Schlüsselwerte) mittels `operator<` anordenbar sein müssen, daher dann aber sehr schnell wiedergefunden werden können. Insbesondere existieren folgende Containerklassen:

vector<T> ist ein dynamisch wachsendes eindimensionales "Feld", das schnelles Einfügen und Entfernen am Ende des `vector` erlaubt; Einfügen oder Löschen an einer anderen Stelle erfordert Umkopieren aller Elemente bis zum Ende. Default- und Kopierkonstruktor sowie Zuweisungsoperator für Elemente sind erforderlich (Sequenz);

list<T> ist eine doppelt verkettete Liste, die schnelles Einfügen und Entfernen an beliebiger Stelle ermöglicht, ohne dass Elemente umkopiert werden müssen (Sequenz);

deque<T> (double ended queue) für z.B. die Warteschlangensimulation, die Verwendung ist ähnlich der eines `vector`, erlaubt aber schnelles Einfügen und Entfernen sowohl am Anfang als auch am Ende des Containers (Sequenz);

set<T> ist eine Menge von Datenelementen, die eine Anordnung mit `operator<` zulassen. Jedes Element ist nur einmal vorhanden, der Zeitaufwand für Einfügen und Finden von Elementen sind i. W. gleich und hängen nur logarithmisch von der Größe des Containers ab (assoziativer Container);

multiset<T> wie set, aber Datenelemente können mehrfach vorhanden sein (assoziativer Container);

map<T> speichert Schlüssel-Wert-Paare. Der Schlüssel muss eindeutig sein und wird nur einfach vorgehalten. `map<T>` ist häufig als balancierter Baum implementiert und daher kann in logarithmischer Zeit ein neues Element eingefügt bzw. ein altes entfernt werden (assoziativer Container);

multimap<T> verhält sich ähnlich wie `map<T>`. Jeder Schlüssel kann jedoch mit mehreren Werten assoziiert sein, Beispiel: eine Person mit mehreren Telefonnummern (assoziativer Container).

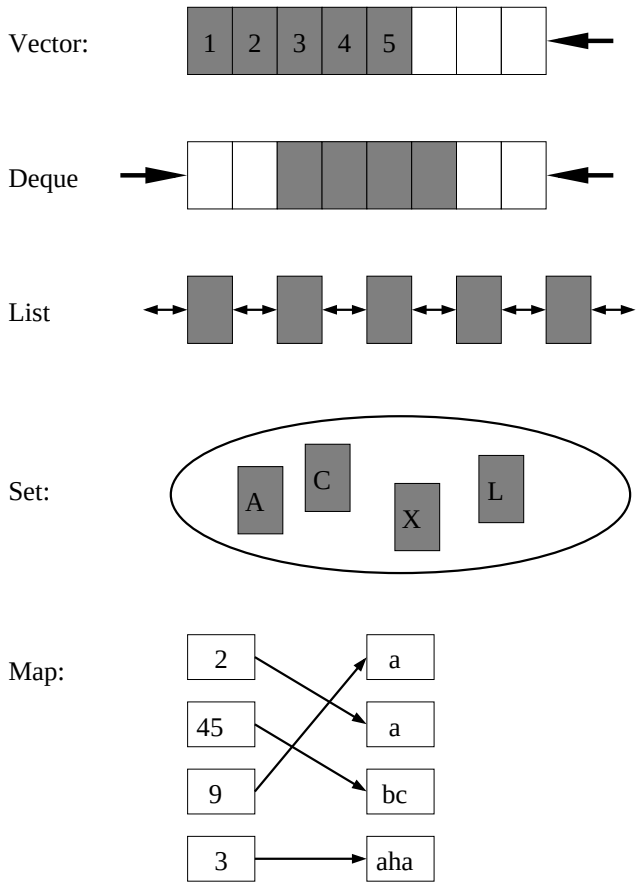


Abbildung 5.1: Grapfische Darstellung einiger Container von STL

Für die in der Numerik häufig benötigten (mehrdimensionalen) Felder, die für einfache algebraische Manipulationen der Elemente vorbereitet sind, gibt es die `valarray`-Klasse (siehe Abschnitt 5.3). Ansonsten ist `vector` der für die sequentielle Speicherung von Daten gleichen Typs am besten geeignete Container. Das folgende Beispiel zeigt seine Benutzung:

```
#include <vector>
```

```
#include <iostream>

int main(){
  std::vector<double> v; // leerer vector<double>

  v.push_back(1.);       // Einfuegen eines Elementes am Ende
  v.push_back(2.);

  v[0] = 47.11;          // existierendes Element überschreiben

  // alle Elemente ausgeben
  for ( unsigned i=0; i < v.size(); ++i ){
    std::cout << v[i] << "\n";
  }
  // alternativ unter Nutzung von Iteratoren
  for ( std::vector<double>::iterator i=v.begin(); i != v.end(); ++i ){
    std::cout << *i << "\n";
  }
}
```

5.1.1 Zugriff auf Elemente beliebiger Container

Die `for`-Schleife des obigen Beispiels demonstriert wie der Zugriff auf Elemente von Container erfolgen kann. Insbesondere für Vektoren ist der Zugriff mittels des gewohnten `operator[]` möglich. Die in der STL dafür vorgesehene allgemeine Schnittstelle nutzt jedoch das allgemeinere so genannte Iteratorkonzept. Iteratoren bilden die grundlegende Schnittstelle, über die Algorithmen den Inhalt eines beliebigen Containers abfragen oder modifizieren können. Sie sind von der Idee her verallgemeinerte Zeiger auf die Elemente, die im Container gespeichert sind. Jeder Container hat die Freiheit, eine eigenen Iteratorklasse zu implementieren, oder wenn sich das anbietet, einfach auf die spracheigenen Zeiger zurückzugreifen. Iteratoren unterstützen Operationen wie `*` und `++`, die auf die Elemente selbst verweisen bzw. das Ansprechen sukzessiver Elemente erlauben. Die verschiedenen Eigenschaften der Container finden sich in der unterschiedlichen Intelligenz der zugeordneten Iteratoren wieder: Da z.B. ein `vector<>` seine Elemente in der Regel konsekutiv im Speicher ablegt, kann der zugeordnete Iterator mit `++` und `--` manipuliert werden, aber auch durch Addition eines Integers eine bestimmte Zahl von Elementen einfach überspringen. Der Iterator einer `list` oder eines `set` bietet die letztere Eigenschaft nicht an, man kann sich im Allgemeinen nur mittels `++` oder `--` von einem Element zum nächsten weiterhangeln. Noch komplexere Container stellen möglicherweise nur die `++` Operation zur Verfügung.

Jeder Container muss seine Iteratoren als Klassen `iterator`, `const_iterator`, `reverse_iterator` und `const_reverse_iterator` bereitstellen, wobei es sich dabei entweder um eine oder mehrere eigenständige Klassen oder um einfache lokale `typedef`s auf eingebaute Typen handelt. Eine Implementierung von `vector` könnte etwa die folgenden Deklarationen vornehmen:

```
template <typename T>
class vector {
  public:
    vector();              // legt Daten linear im Speicher ab
    // ...
    typedef T*       iterator;
    typedef T*       reverse_iterator;
    typedef const T* const_iterator;
```

```
    typedef const T* const_reverse_iterator;

    typedef T value_type;
    typedef int size_type; // implementierungsabhaengig
    //  ...
    iterator begin();
    iterator end();
    const_iterator begin() const;
    const_iterator end() const;
    reverse_iterator rbegin();
    reverse_iterator rend();
    const_reverse_iterator rbegin() const;
    const_reverse_iterator rend() const;

    size_type size() const;
    size_type max_size() const;
  private:
    T *data;              // privates "Feld" fuer Daten,
                          // implementierungsabhaengig
};
```

Die `const`-Versionen der Iteratoren müssen immer dann verwendet werden, wenn der Container selbst `const`-Eigenschaften hat (etwa weil er als konstante Referenz an ein Unterprogramm übergeben wurde) und somit auch über Iteratoren keine Modifikation des Containerinhaltes möglich sein soll. Ignoriert man diese Regel, so folgt die Strafe in Form eines nicht kompilerenden Programmes auf den Fuß.

Die weiteren `typedef`'s für `value_type` und `size_type` sind insbesondere in Templateprogrammen wichtig, die weitere Einzelheiten über die verwalteten Datentypen oder den Container kennen müssen.

Die Elementfunktion `begin()` liefert einen Iterator (immer mindestens mit den Eigenschaften eines Vorwärtsiterators), der auf das erste Element des Containers verweist. Mittels `operator++` lässt sich dieser bis `end()` fortschalten, der den ersten ungültigen Iterator dieser Sequenz liefert, d.h. einen, der auf kein Datenelement verweist. Im Zusammenspiel mit `operator*`, der den Wert liefert, auf den verwiesen wird, können wie in folgenden Beispiel alle Elemente einer Liste angesprochen werden:

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

int main(){
  std::list<double>   v;
  v.push_back(1.);
  v.push_back(2.);
  // ...

  // nutzt Vorwaertsiteratoreigenschaften:
  std::list<double>::iterator i;
  for ( i=v.begin(); i != v.end(); ++i ){
    std::cout << *i << "\n";  // gibt alle Elemente aus
  }
  // nutzt Eigenschaften bidirektionaler Iteratoren:
  while ( i != v.begin() ) {
    std::cout << *(--i) << "\n"; //gibt alle Elemente rueckwaerts aus
  }
```

```
  // Rueckwaertsiterator:
  for ( std::list<double>::reverse_iterator j = v.rbegin();
        j != v.rend(); ++j ) {
    std::cout << *j << "\n";  // gibt alle Elemente rueckwaerts aus
  }
}
```

Zusammenfassend unterscheidet die STL folgende Iteratortypen, die sich in der unterstützten Funktionalität unterscheiden.

Vorwärts- Iteratoren (forward iterator) sind auf das elementweise Voranschreiten beschränkt. Sie unterstützen nur `++` als arithmetische Operation, den Elementzugriff mit `*` und `->` sowie Vergleiche auf Gleichheit und Ungleichheit. Jeder durch `begin()` gelieferte Iterator ist mindestens ein Vorwärtsiterator.

Rückwärts- Iteratoren (reverse iterator) unterstützen sinngemäß ebenfalls das Fortschreiten mit `++`, bewegen sich allerdings in umgekehrter Richtung durch den Container. Vergleiche auf Gleichheit und Ungleichheit sind möglich. Man erhält solche Iteratoren über die Elementfunktion `rbegin()` und kann das Ende der gültigen Sequenz durch Vergleich mit `rend()` erkennen.

Bidirectionale Iteratoren sind solche, die in einer Klasse die Eigenschaften von Vorwärts- als und die von Rückwärtsiteratoren vereinen. Rückwärtsiteration erfolgt mit `--`. Die Iteratoren einer doppelt verketteten Liste `list` sind bidirektionale Iteratoren. `begin()` und `rbegin()` liefern in diesem Falle den gleichen Typ (aber natürlich verschiedene Werte) zurück.

Random Access Iteratoren unterstützen die volle übliche Zeigerarithmetik, also insbesondere die Manipulation durch `--`, `++`, die Addition oder Subtraktion ganzer Zahlen und auch die Indizierung durch `[]`. Zugriffe auf den bezogenen Wert oder dessen Elemente erfolgen mit `*` bzw. `->`. Vergleiche solcher Iteratoren sind möglich mit `==, <, >, !=`. Iteratoren der `vector<>` Klasse, die Daten i.W. linear im Speicher ablegt und als Iteratoren einfach Zeiger auf diesen Speicher verwenden kann, sind Random Access Iteratoren.

Je nachdem, wieviel "Intelligenz" ein Algorithmus der STL dem Container abverlangt, sind als Argumente mehr oder weniger "intelligente" Iteratortypen erforderlich. Da es sich bei den Algorithmen um Templatecode handelt, macht sich die Übergabe eines "zu dummen" Iterators durch Fehlermeldungen des Compilers über fehlende Operatoren bei der Instantiierung des Algorithmus bemerkbar.

Beim Umgang mit Iteratoren ist zu beachten, dass sie (abhängig vom Containertyp) in der Regel ungültig werden, wenn der Inhalt des zu Grunde liegenden Containers verändert wird. Während dies beim Löschen von Elementen klar ist, kann es beim Einfügen vom Container abhängig sein (Einfügen in eine Liste wird häufig bisherige Iteratoren nicht ungültig machen). Wird etwa ein Element an einen `vector` angehängt, so kann es zu einer Neuallokation von Speicherplatz kommen. Damit sind natürlich alle "alten" Verweise auf Elemente des `vector` hinfällig.

5.1.2 Eigenschaften der STL-Container-Klassen

Allgemein arbeiten die STL Container mit Objekten, die sich kopieren und zuweisen lassen. Bei großen Objekten ist es daher sinnvoll, nicht die Objekte selbst, sondern Zeiger auf die Objekte mit STL-Containern zu verwalten.

Die Container selbst weisen Funktionen auf, mit deren Hilfe sich Elemente einfügen und wieder auslesen lassen. Container können als Ganzes kopiert oder zugewiesen werden, wobei Kopien der enthaltenen Elemente angefertigt werden. Beim Schreiben von Funktionen sollte man also beachten, dass bei Übergabe "by value" eine vergleichsweise aufwendige Kopieraktion stattfindet. In vielen Fällen ist es ausreichend und angebracht, Container als Referenzen zu übergeben und so das Kopieren zu umgehen.

Welche Funktionen im Einzelnen angeboten werden, ist vom Typ des Containers abhängig. Sequenzen und assoziative Container haben jeweils spezielle zusätzliche Funktionen, um die Container optimal nutzen zu können.

```
#include <list>
#include <map>
#include <vector>

struct A {
  // ...
  bool operator<(const A&){return true;}
  bool operator==(const A&){return true;}
};

int main(){

  // gemeinsame Eigenschaften:

  std::list<A>             m, m2;
  std::list<A>::value_type element;
  std::list<A>::size_type  number;
  std::list<A>::iterator   pos1=m2.begin(), pos2=m2.end();
  std::list<A>::const_iterator cpos; // verbietet, Elemente zu aendern

  m.size();           // Zahl der gespeicherten Elemente
  m.max_size();       // size() des groesstmoeglichen Containers
  m.empty();          // bool, true falls leer
  m.swap(m2);         // Inhalte von m und m2 vertauschen (schnell)
  m.begin();          // Iterator auf erstes Element
  m.end();            // Iterator jenseits des letzten Elements

  // 1. spezielle Funktionen fuer reversible Container:
  std::list<A>::reverse_iterator rpos;
  rpos = m.rbegin();  // letztes Element
  m.rend();           // "Zeiger" vor das erste Element

  // 2. spezielle Funktionen fuer Sequenzen
  std::list<A>           nlist(3,A());  // 3 Default A's
  std::list<A>::iterator npos1=nlist.begin(), npos2=nlist.end();

  m.insert(pos1,element); // element vor pos1 einfuegen, Iteratoren
                      //  jetzt ungueltig
  pos1 = m.begin();
  m.insert(pos1,npos1,npos2); // fuegt Elemente im Interval
                      //  npos1..npos2 vor pos1 ein, pos1 wird
                      //  ungueltig
  pos1 = m.begin();
```

```
  m.erase(pos1);      // loescht Element an Stelle pos1
  pos1 = m.begin(), pos2 = m.end();
  m.erase(pos1,pos2); // loescht Elemente im Bereich pos1..pos2
  m.clear();          // m.erase(m.begin(),m.end());

  // 2.1. spezielle Funktionen fuer vector, list, deque

  std::vector<A>            v(10,A());
  std::vector<A>::size_type n(3);

  m.front();          // alle: erstes Element
  m.back();           // alle: letztes gueltiges Element
  m.push_front(element);  // list, deque: als erstes Element einfuegen
  m.push_back(element);   // alle: als letztes Element anfuegen
  m.pop_front();      // list, deque: m.erase(a.begin())
  m.pop_back();       // alle:        m.erase(--a.end())
  v[n];               // vector, deque: Zugriff auf *(v.begin()+n)
  v.at(n);            // vector, deque: wie [] mit Indextest

  //     nur list:
  m.sort();           // list: sortiert die Elemente
  m.unique();         // list: eliminiert mehrfache Elemente
  m.merge(m2);        // list: verschmilzt 2 sortierte Listen
  m.reverse();        // list: kehrt Reihenfolge um
  m.remove(element);

  //     nur vector:
  v.capacity();       // elements w/o allocation
  v.resize(n,element);// erase or insert elements down/up to n
  v.reserve(n);       // tell vector that we want it to reserve
                      //  space for n objects

  // 3. spezielle Funktionen fuer assoziative Container

  std::map<int,A>            ma;
  std::map<int,A>::key_type  k(15);   // Typ des Schluessels, hier int
  std::map<int,A>::value_type map_element(42,A()); //Schluessel-Wert-Paar

  ma.insert(map_element); // gibt Iterator auf Position im Container
  ma.erase(42);       // loescht alle Elemente mit Schluessel 42
  ma.find(42);        // gibt Iterator auf ein Element mit Schluessel
                      //  42 oder ma.end() falls nicht vorhanden
  ma.count(k);        // Zahl der Elemente mit Schluessel k
  ma.lower_bound(k);  // Iterator auf erstes Element mit
                      //  Schluessel nicht kleiner als k
  ma.upper_bound(k);  // Iterator auf erstes Element mit
                      //  Schluessel groesser als k
}
```

Hier sind zwei Beispiele für die Implementierung eines Algorithmus, in denen einmal Elementfunktionen und lokale `typedef`'s eines STL Containers und zum Zweiten nur Iteratoren verwendet werden. Die Funktion gibt die zwei ersten Elemente eines Containers in umgekehrter Reihenfolge aus, indem sie das erste Element (`m.begin()`) zwischenspeichert,

den Iterator (von dem nur Vorwärtsiterator-Eigenschaften verlangt werden) fortzählt, um das zweite Element zu erhalten und dann beide ausgibt.

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

template <class CONTAINER>
void print_first_two_reverse(const CONTAINER & c)
{
  if ( c.size() < 2 ) return;
  // containerspezifischer Iterator (const wegen Uebergabetyp):
  typename CONTAINER::const_iterator it = c.begin();
  // Elementtyp des Containers
  typename CONTAINER::value_type first(*it);
  std::cout << *(++it) << '\n'
            << first << std::endl;
}

template <class Iterator>
void print_first_two_reverse(Iterator begin, Iterator end )
{
  Iterator current(begin);
  if ( current != end )   // erstes Element gueltig?
    ++current;
  if ( current != end ){  // zweites Element gueltig?
    std::cout << *current << '\n'
              << *begin << std::endl;
  }
}

int main()
{
  double   a[] = {1., 2.};
  std::vector<double> s(a,a+2); // Konstruktor mit Iteratoren

  print_first_two_reverse(s);   // Container benoetigt
  print_first_two_reverse(s.begin(),s.end()); // ok
  print_first_two_reverse(a,a+2); // normale Zeiger funktionieren auch!
}
```

Das Schlüsselwort `typename` findet hier Verwendung, um dem Compiler bei der Instantiierung des Templatecodes zu helfen. Es zeigt an, dass es sich bei `CONTAINER::iterator` um einen Typ handelt. Es wäre ja auch möglich, dass es sich um einen Variablennamen oder eine andere Größe handelt, die im gegebenen Zusammenhang syntaktisch fehlerhaft wäre. Es ist immer dann nötig, wenn die angegebene Größe ein Name aus einer Templateklasse ist, die im Code noch nicht soweit spezialisiert ist, dass die Bedeutung des Namens für den Compiler ersichtlich ist.

Viele Containeroperationen, wie etwa die Benutzung von `operator[]` auf `vector<>` oder Zugriffe über Iteratoren, werden aus Gründen der Performance von STL Implementierungen in der Regel nicht auf die Überschreitung von Bereichsgrenzen, der Gültigkeit des Iterators, etc. geprüft. Daher muss die Gültigkeit von Iteratoren durch Vergleich mit dem jeweils ersten ungültigen Wert oder in anderer Form sichergestellt werden. Die "safe STL" [8] stellt bei Bedarf (für Debugzwecke) entsprechende Funktionalität zur Verfügung.

Man sieht oben, dass sich Programmcode, der für Iteratoren geschrieben wurde, auch mit normalen C-Zeigern verwenden lässt. Der für Container-Argumente geschriebene Code hingegen funktioniert nur mit Containern. Um in flexiblerer Art und Weise Algorithmen mit Iteratoren schreiben zu können, bietet die Bibliothek die so genannten Iterator-Traits an (siehe Abschnitt 4.6).

5.1.3 Container-Adapter-Klassen

Einige Datenstrukturen, die man prinzipiell als Container erwarten könnte (etwa unseren Stack aus dem Template Kapitel in Abschnitt 4.3.2), sind in der STL nicht als eigenständige Container verwirklicht, sondern als ein eingeschränktes, spezielles Interface zu einem beliebigen Container. Dieser zugrunde liegende Container kann vom Benutzer als Template-Argument bei der Deklaration spezifiziert werden. Wer sich dabei ertappt, einen dieser Adaptoren verwenden zu wollen, der sollte sich auch die Eigenschaften der Algorithmen `make_heap`, `sort_heap`, `pop_heap`, `push_heap` anschauen, wozu wir mangels Platz allerdings auf andere Dokumentation verweisen müssen [23, 24].

queue

`queue` simuliert eine Warteschlange (FIFO: first in, first out-Speicher), bei der Elemente immer am Ende mit `push(...)` angefügt und vorn mit `pop()` entfernt werden. Die Funktionen `front()` und `back()` erlauben das Betrachten von Elementen an Anfang bzw. Ende der queue. Spezielle Versionen dieser Funktionen existieren für konstante Objekte, wie sie bei Parameterübergabe an Unterfunktionen per konstanter Referenz häufig entstehen.

```
template <class T, class Sequence = std::deque<T> >
class queue {
public:
  explicit queue(const Sequence &s) : m_s(s){};

  typedef typename Sequence::value_type value_type;
  typedef typename Sequence::size_type  size_type;
  typedef Sequence                      container_type;

  value_type & front();              // Blick auf erstes Element
  const value_type & front() const;  // noetig fuer const queue (&)
  value_type & back();               // Blick auf letztes Element
  const value_type & back() const;
  void push(const value_type &);     // Element hinten anfuegen
  void pop();                        // Element vorne entfernen
  // ...
protected:
  Sequence m_s;
};
```

stack

Ein Stapelspeicher (deutsches Wort für Stack) liefert die zuerst eingespeicherten Elemente als letzte wieder ab (ganz wie ein Stapel Papier auf dem Schreibtisch, der nach oben wächst und von oben wieder kleiner wird (LIFO: last in, first out-Speicher).

Der STL `stack` benutzt `top()`, um sich das Datenelement auf der obersten Position des Stapels anzusehen. Erst dann wird das Element mit `pop()` entfernt. Der Vorteil dieser kompliziert scheinenden Vorgehensweise ist, dass das oben liegende Element nicht kopiert werden muss, sondern mit Referenzen gearbeitet werden kann.

Zum Anfügen von Elementen dient `push(...)`

```
template <class T, class Sequence = std::deque<T> >
class stack {
public:
  explicit stack(const Sequence &s) : m_s(s){};

  typedef typename Sequence::value_type value_type;
  typedef typename Sequence::size_type  size_type;
  typedef Sequence                      container_type;

  value_type & top();              // Blick auf erstes Element
  const value_type & top() const;  // noetig fuer const queue (&)
  void push(const value_type &);   // Element oben ablegen
  void pop();                      // Element oben entfernen
  // ...
protected:
  Sequence m_s;
};
```

priorityqueue

Zusätzlich zu den Datenelementen werden noch Prioritätswerte zugelassen. Innerhalb der durch einen Prioritätswert festgelegten Klasse werden die Elemente wie bei einer queue behandelt. Elemente mit höherer Priorität stehen allerdings immer *vor* den Elementen mit niedriger Priorität in der priority queue.

5.1.4 Algorithmen

Vorbemerkungen

In der `<algorithm>` Header-Datei befinden sich Template-Funktionen, die mittels Iteratoren auf Containerelemente zugreifen. Dabei werden die Containerinhalte entweder nur gelesen, wie etwa bei Algorithmen, die Elemente suchen, zählen oder vergleichen *(non-modifying sequence operations*, oder die Elementwerte werden aktiv modifiziert *(modifying sequence operations)* wie bei Algorithmen, die Bereiche kopieren oder löschen. Zwar sind sortierende und verwandte Algorithmen eigentlich auch *(modifying sequence operations)*, aber so wichtig, dass sie in der Bibliothek als eigene Kategorie gelten.

Funktionsobjekte

Um Algorithmen sinnvoll nutzen zu können, benötigen wir noch ein weiteres wichtiges Konzept, dass es uns erlaubt, gewissermaßen "Aktionen" an den Algorithmus zu übergeben. Damit ist etwa eine Funktion gemeint, die in der Lage ist, zwei Elemente zu vergleichen, wenn dies nicht bereits durch einen `operator<` geleistet wird oder eine Funktion, die das Maximum oder Minimum einer Reihe von Werte bestimmen kann, die ihr nacheinander übergeben werden.

Wir wollen das Prinzip hier am `for_each` Algorithmus demonstrieren, der eine explizit geschriebene `for`-Schleife über die Containerelemente ersetzen kann. Angenommen, wir wollen `for_each` dazu bringen, alle im Container gespeicherten Werte auf der Konsole auszugeben. Dazu müssen wir eine Funktion schreiben, die dies für ein Element erledigt und dann diese Funktion an `for_each` übergeben. Wir benutzen Template-Funktionen, um uns nicht von vornherein auf einen Typ von Elementen festzulegen:

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

template <typename T>
void print_me(const T &elem) {
  std::cout << elem << " ";
}

int main(){
   std::vector<int> v;
   // ...
   std::for_each(v.begin(), v.end(), print_me<int>);
}
```

Gegenüber der expliziten Implementierung einer Schleife besteht der Vorteil dieser Vorgehensweise darin, dass man die direkte Manipulation von Iteratoren vermeidet und so Fehlermöglichkeiten verringert (etwa Bereichsgrenzenprobleme). Bei `for_each` ist dieser Vorteil sicher nicht unmittelbar ersichtlich, aber in vielen sortierenden Algorithmen sind die Operationen auf den Iteratoren genügend komplex, um den zusätzlichen Aufwand mehr als zu rechtfertigen.

Darüber hinaus ist beim Design der STL darauf geachtet worden, dass so genannte "light weight" Objekte als Kopien übergeben werden. Damit vermeidet man das Aliasing und erleichtert dem Compiler das Optimieren des Codes. [1]

Die Implementierung der Algorithmen als Templates erlaubt anstelle von Funktionszeigern wie im obigen Beispiel auch so genannte Funktionsobjekte oder Funktoren zu verwenden. Die Syntax eines Funktionsaufrufs lässt sich durch Überladen des `operator()` einer Klasse imitieren. So kann das obige Beispiel auch in der folgenden Art und Weise implementiert werden.

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

template<class T>
struct PrintMe {
  void operator()(const T &elem) {
      std::cout << elem << " ";
  }
};

int main() {
  std::vector<int> v;
  //...

  PrintMe<int>     print_me;  // eine Instanz des Funktionsobjektes
  std::for_each( v.begin(), v.end(), print_me );

  // temporaere Objekte sind auch moeglich:
  std::for_each( v.begin(), v.end(), PrintMe<int>() );
```

[1] Falls das dabei häufig nötige Kopieren der Objekte vermieden werden muss, so kann man in dem Container statt der Objekte selbst nur Zeiger auf mit `new` erzeugte Instanzen speichern. Vorsicht ist allerdings bei Löschen oder anderen Operationen geboten, bei denen Zeigerwerte verloren gehen können. Hier muss man möglicherweise mit einer eigenen Zeigerverwaltungsklasse (*smart pointer*, siehe 6.3) Abhilfe schaffen.

```
}
```

Im Falle, dass der `operator()` des Funktionsobjektes eine `inline` Funktion ist, kann der Compiler den Funktionsaufruf eliminieren, was mit reinen (nicht-inline) Funktionspointern nicht möglich ist. Dies bedeutet, dass bei Benutzung der Algorithmen der STL mit Funktionsobjekten oder Zeigern auf inline-Funktionen keinerlei Verlust an Laufzeiteffizienz gegenüber dem expliziten Ausschreiben der Schleifenoperationen eintritt.

Die Verwendung von Funktionsobjekten ist jedoch flexibler als diejenige von Funktionen, weil als Elemente der Klasse noch Datenelemente zulässig sind, die während der Abarbeitung des Algorithmus erhalten bleiben. Ein ähnliches Verhalten lässt sich mit statischen Variablen in Funktionen erreichen, die allerdings als globale Variablen bei mehrmaliger Verwendung der Funktion zu unerwünschten Wechselwirkungen Anlass geben können (etwa weil sie sich nicht reinitialisieren lassen oder weil bei Threadprogrammierung gleichzeitig aus mehreren Aufrufen der Funktion auf sie zugegriffen werden kann).

Bei der Arbeit mit Funktionsobjekten muss man beachten, dass diese bei Parameterübergabe und bei Rückgabe kopiert werden, was manchmal zu unerwartetem Verhalten führt. Ein standardkonformer `for_each` Algorithmus (im globalen Namensraum) könnte folgendermaßen implementiert sein,

```
  template<class ITERATOR, class FUNCTOR>
  FUNCTOR
  for_each(ITERATOR begin, ITERATOR end, FUNCTOR functor){
    ITERATOR it = begin;
    while ( it != end )
      functor(*it++);
    return functor;
  }
```

und ein Funktionsobjekt zum Aufsummieren von Integer-Werten könnte so formuliert werden:

```
  class SumIt {
  public:
    SumIt(): my_sum(0){}
    void operator()( int e ){ my_sum += e; }
    int sum(){ return my_sum; }
  private:
    int my_sum;
  };
```

Damit führt die naive Vorgehensweise

```
void f(){
  std::vector<int> v;
  v.push_back(3);
  v.push_back(4);
  SumIt mysum;
  ::for_each(v.begin(),v.end(),mysum);  // Auswahl unserer Version
  std::cout << " Summe= " << mysum.sum() << std::endl;
}
```

allerdings nicht zum gewünschten Ergebnis, sondern gibt immer 0 als Summe aus. Der Grund ist, dass nicht `mysum` selbst, sondern eine Kopie an den Algorithmus übergeben wird. Aus diesem Grund kann in `for_each` das Objekt `mysum` nicht verändert werden. Man erhält das gewünschte Resultat, wenn man wie im folgenden Beispiel den Rückgabewert von `for_each` auswertet, der eine Kopie des geänderten Objektes ist.

```
void g(){
  std::vector<int> v;
  v.push_back(3);
  v.push_back(4);
  SumIt mysum = ::for_each(v.begin(),v.end(),mysum);
  std::cout << " Summe= " << mysum.sum() << std::endl;
}
```

Nichtmodifizierende Algorithmen

Wir wollen jetzt die wichtigsten Algorithmen der STL kennenlernen. Für ausführlichere Informationen verweisen wir auf die Dokumentation der Bibliothek eines Compilers, den Quelltext der Deklarationsdateien der Bibliothek oder auf Ressourcen im Internet [24].

Mittels `find` oder `find_if` lässt sich das erste Element in einem durch Iteratoren charakterisierten Bereich finden, auf das eine bestimmte Bedingung zutrifft. Der Suchbereich wird, wie wir das bereits oben am Beispiel von `for_each` gesehen haben und wie es auch sonst in der STL üblich ist, durch ein Paar von Iteratoren festgelegt. Dabei wird der erste Iterator immer als dem Bereich zugehörig gerechnet, während der zweite Iterator bereits gerade außerhalb liegt. Das bedeutet, dass die von den Elementfunktionen `begin()` und `end()` der Container gelieferten Iteratoren direkt verwendet werden können. Falls beide Iteratoren gleich sind, kennzeichnet das einen leeren Bereich. Das Ergebnis von `find` wird als Iterator, der auf das gefundene Element verweist, an die aufrufende Umgebung zurückgegeben.

```
#include <vector>
#include <algorithm>

int main(){
  std::vector<int>  collection;
  std::vector<int>::iterator it;
  // ...
  // erste 5 suchen
  it = std::find(collection.begin(), collection.end(), 5);
  if ( it != collection.end() )   // ja, gefunden
     // im eingeschraenkten Bereich weitersuchen
     it = std::find(it,collection.end(),42);
  // ...
}
```

Wird kein passendes Element gefunden, so wird der als zweites Argument übergebene Iterator zurückgegeben. Eigene Bedingungen lassen sich durch Funktionen/Funktionsobjekte und `find_if` behandeln, im folgenden Beispiel liefert `is_odd` den Wert `true` wenn das Argument eine ungerade Zahl ist. Die Routine `find_if` liefert einen Iterator auf das erste Element, das diese Bedingung erfüllt oder sonst den Wert `collection.end()`.

```
#include <vector>
#include <algorithm>

bool is_odd( int elem ){
  return elem % 2;
}

int main(){
  std::vector<int> collection;
  std::vector<int>::iterator it;
```

```
  // ...
  // erstes ungerades Element suchen
  it = std::find_if(collection.begin(), collection.end(), is_odd);
  if ( it != collection.end() )   // ja, gefunden
    ; // ...
}
```

In Tabelle 5.1 sind die meisten nicht-modifizierenden Algorithmen aufgelistet. Dort stehen `b<n>` und `e<N>` für Iteratoren, die einen Bereich `[b<n>,e<n>)` begrenzen, `p` für einen Iterator, der das Ergebnis enthält, `v` für einen Wert, `pred` für eine logische Funktion oder Funktor, `fct` für einen Funktor ohne spezifischen Rückgabewert und `cmp` für einen Funktor, der zwei Werte vergleicht.

Tabelle 5.1: **Nicht sequenzmodifizierende STL Algorithmen.**

`fct for_each(b,e,fct)`	lesender Zugriff auf Elemente durch Funktor
`int count(b,e,v)`, `int count_if(b,e,pred)`	zählt passende Elemente
`p find(b,e,v)`, `p find_if(b,e,pred)`	Suche eines bestimmten Elementes
`p find_first_of(b,e,b2,e2)`	sucht in `[b,e)` eines von der Elemente aus `[b2,e2)`. `p` zeigt auf das erste passende Element
`p (min/max)_element(b,e)`	p verweist auf kleinstes/größtes Element, auch `cmp` als drittes Argument möglich
`bool equal(b1,e1,b2,e2)`	testet zwei Bereiche auf Gleichheit, auch `cmp` als fünftes Argument möglich
`p adjacent_find(b,e)`	findet erstes Wertepaar, auch `cmp` als drittes Argument möglich
`p find_end(b,e,b2,e2)`	sucht letzte Teilfolge in `[b,e)`, die mit `[b2,e2)` übereinstimmt, auch `cmp` als fünftes Argument möglich. `p` zeigt auf die letzte gefundene passende Teilfolge
`search(b,e,b2,e2)`	wie `find_end`, sucht aber die erste Teilfolge, auch `cmp` möglich

Modifizierende und Sortierende Algorithmen

Modifizierende Algorithmen verändern den Inhalt des Containers über die zur Verfügung gestellten Iteratoren. Dabei muss beachtet werden, dass das in der Regel nur ein Umkopieren der vorhandenen Elemente innerhalb des Containers erlaubt und kein Löschen oder Hinzufügen, denn diese Operationen würden ja die Iteratorargumente ungültig werden lassen.

Löschen Daher ist klar, dass die STL Löschen in zwei Schritten vollzieht. Im ersten Schritt, ausgeführt durch `remove`, werden nur Werte von Elementen umkopiert und die "entfernten" Werte an das "Ende" des Containers gebracht. `remove` liefert das neue Ende des Containers als Iterator zurück. Der Benutzer muss dann mit der Elementfunktion `erase` des Containers dafür Sorge tragen, dass die Elemente aus dem Container entfernt werden. `erase` kann dabei ein einzelnes oder einen durch Iteratoren eingegrenzten Bereich löschen. Im folgenden Beispiel werden alle Elemente, die gleich 42. sind, aus einem Container entfernt:

```
#include <vector>
```

```
#include <algorithm>

int main(){
  std::vector<double>  v(10,1.); //10 Elemente, alle 1.
  v[5] = 42.;
  std::vector<double>::iterator new_end;
  new_end = remove( v.begin(), v.end(), 42. );
  v.erase( new_end, v.end() );
}
```

Ähnlich funktioniert `remove_if`, das wie `find_if` statt eines Wertes einen Funktionspointer oder ein Funktionsobjekt als Argument erwartet, welches einen `value_type` nimmt, eine Bedingung berechnet und dann einen `bool` liefert.

Ebenso kollabiert `unique` Abfolgen gleicher Elemente zu einem einzigen Element. Für sortierte (siehe unten) Container entfernt dies so alle Duplikate. `Unique` erwartet Vorwärtsiteratoren als Argumente und liefert einen Iterator, der auf das erste zu löschende Element verweist.

```
  std::vector<double>::iterator it;
  new_end = std::unique( v.begin(), v.end() );
  v.erase( new_end, v.end() );
```

Kopieren Das Kopieren von Elementen von einem Bereich in einen anderen wird durch `copy` erledigt. `copy` erwartet nur drei Argumente, deren drittes ein Iterator ist, auf den zugewiesen und der sukzessive inkrementiert wird. Dieser Mechanismus lässt sich auch verwenden, um die Elemente eines Bereiches in einen anderen Container einzufügen, anstatt nur dort vorhandene Elemente zu überschrieben. Dazu bedient man sich sogenannter Einfügeiteratoren, die bei genauerem Hinsehen über die Iteratorschnittstelle nur die Elementfunktionen `push_back` oder `push_front` der Container zur Verfügung stellen (vgl. Abschnitt 5.1.5). Die Bibliothek bietet die fertigen Klassen `front_insert_iterator` und `back_insert_iterator` zu diesem Zweck an.

```
#include <vector>
#include <algorithm>
#include <iterator>

int main(){
  std::vector<double>  v(10,1.);
  std::vector<int>     w;  // leer
  // w mit Elementen aus v fuellen
  std::copy(v.begin(), v.end(), std::front_insert_iterator(w) );
  w[3] = 42.;              // jetzt legal
  // Zurueckkopieren durch Ueberschreiben der Elemente in v
  std::copy(w.begin(), w.end(), v.begin());
}
```

Sortieren Mittels `sort` lässt sich ein durch zwei Iteratoren charakterisierter Bereich von Datenelementen eines Containers sortieren, der Bereich ist wie gewohnt dabei inklusive des ersten und exklusive des zweiten Iterators,

```
#include <vector>
#include <algorithm>

int main(){
```

```
  std::vector<int>  v;
  // ... beliebige Elemente einfuegen

  std::sort( v.begin(), v.end() );
}
```

Falls vorhanden und zugänglich, sollte man zum Sortieren allerdings lieber statt dieser generischen Sortierfunktion die Elementfunktion des Containers verwenden, die bedeutend schneller sein kann.

Übersicht Eine Übersicht über die vorhandenen mutierenden und sortierenden Algorithmen gibt die Tabelle 5.2. In Ergänzung der Abkürzungen aus dem letzten Abschnitt steht hier `o` für einen Ausgabeiterator (ein Iterator, der mit `operator++` bzw. `operator--` fortgeschaltet und auf den mit `operator*` und `=` zugewiesen werden kann.) `op` ist ein Funktor (unärer Operator), der ein modifiziertes Argument zurückgibt, ebenso `binop` für zwei Argumente (binärer Operator). `gen` ist ein Funktor, der kein Argument erwartet aber einen Wert liefert. Für Vergleiche soll das binäre Funktionsobjekt `comp` (im Gegensatz zu `cmp`) wahr ergeben, wenn das erste Argument kleiner ist als das zweite.

Tabelle 5.2: **Modifizierende/Sortierende STL Algorithmen**

`o copy(b,e,o)`	kopiert Bereich `[b,e)` nach `o` durch sukzessives Inkrementieren von `o`, Rückgabewert: nächste nicht zugewiesenen Position
`o swap(b,e,o)`	vertauscht Elemente in Bereichen (wie `copy`)
`o transform(b,e,o,op)`, `o transform(b1,e1,b2,o,binop)`	`op` wird auf das Element aus `[b,e)` angewendet und das Resultat auf `o` zugewiesen. Bei der zweiten Version kommen die Argumente von `binop` aus Bereich `[b1,e1)` bzw. `[b2,...)`
`replace(b,e,vold,vnew)`, `replace_if(b,e,pred,vnew)`	ersetzt Werte in Bereich durch `vnew`, wenn sie gleich `vold` sind bzw. `pred()` wahr ist
`o replace_copy(b,e,o,vold,vnew)`, `o replace_copy_if(...,pred,.)`	gibt dem Zielbereich einen neuen Wert, wenn `=vold` oder `pred` wahr
`fill(b,e,v)`	gibt Bereich einen neuen Wert `v`
`generate(b,e,gen)`	wie `fill()`, zugewiesen wird aber der Rückgabewert des Funktors `gen` (ohne Argument)
`p remove(b,e,v)`, `p remove_if(b,e,pred)`	löscht Elemente falls `=v` oder `pred` wahr, ggf. .erase(p,e) nötig
`o remove_copy(b,e,o,v)`, `o remove_copy_if(...,pred)`	kopiert Bereich außer Elementen, für die `=v` oder `pred` wahr ist
`p unique(b,e,v)`	löscht aufeinander folgende Duplikate, `p` zeigt auf das Ende der entstandenen Sequenz, `cmp` statt `v` möglich. Ggf. `...erase(p,e)` nötig.
`unique_copy(b,e,o)`, `unique_copy(b,e,o,cmp)`	kopiert stattdessen nach `o` mit binärem Prädikat
`reverse(b,e)`	paarweises `swap` von Elementen zur Umkehr der Reihenfolge
`o reverse_copy(b,e,o)`	wie reverse, aber Ausgabe nach `o`

Tabelle 5.2: **Modifizierende/Sortierende STL Algorithmen**

`random_shuffle(b,e)`	Durcheinanderwürfeln (optional kann als drittes Argument kann ein Zufallszahlengenerator übergeben werden)
`partition(b,e,pred)`	alle Elemente, die `pred` erfüllen, werden vor diejenigen platziert, die es nicht erfüllen
`stable_partition(b,e,pred)`	wie `partition`, aber die vorhandene relative Anordnung der Elemente bleibt erhalten
`sort(b,e,comp)`	Sortieren von `[b,e)`, ohne `comp` wird `operator<` benutzt
`stable_sort(b,e,comp)`	wie `sort()`, ändert aber nicht die relative Reihenfolge von Elementen
`nth_element(b,p,e)`	Nach Aufruf zeigt `p` auf das Element, das an dieser Stelle stünde, wäre die gesamte Sequenz sortiert worden (nützlich zur Bestimmung des Median)
`bool binary_search(b,e,v,comp)`	wahr wenn ein Element `el` existiert, für das `!comp(*el,v) && !comp(v,*el)`
`o merge(b1,e1,b2,e2,o,comp)`	verschmilzt zwei Bereiche in einen dritten (`o`), nicht überlappenden. Ohne `comp` nutzt der Algorithmus `operator<`
`inplace_merge(b,p,e,comp)`	verschmilzt die vorsortierten Bereiche `[b,p)` und `[p,e)` desselben Containers in `[b,e)`. Ohne `comp` wird `operator<` genutzt
`bool includes(b1,e1,b2,e2,comp))`	wahr wenn jedes Element des sortierten Bereichs `[b2,e2)` im sortierten Bereich `[b1,e1)` enthalten ist. Ohne `comp` wird `operator<` verwendet.
`o set_union(b1,e1,b2,e2,o,comp)`	Vereinigungsmenge sortierter Bereiche. Ohne `comp` wird `operator<` genutzt
`o set_intersection (b1,e1,b2,e2,o,comp)`	dto. für Schnittmenge
`o set_difference (b1,e1,b2,e2,o,comp)`	dto. für Differenzmenge
`bool prev_permutation(b,e,comp)` `bool next_permutation(b,e,comp)`	Permutationen: wahr, wenn eine lexikografische Ordnung aller Permutationen existiert und daher ein eindeutiges Resultat möglich ist. Nutzt `operator<` falls `comp` nicht angegeben ist

Elementfunktionen spezieller Container

Manche Algorithmen lassen sich für bestimmte Containertypen in effizienterer Weise als Elementfunktionen der Container implementieren. Ein Beispiel ist der `sort()` Algorithmus für Listen. Falls maximale Performance erforderlich ist, sollten diese an Stelle der allgemeineren Algorithmen verwendet werden.

5.1.5 Iterator-Adaptoren

Iteratoradaptoren sind Hilfsklassen, die in den Algorithmen der STL an die Stelle von Iteratoren treten können, jedoch eine Funktionalität jenseits der Designvorstellung "ver-

allgemeinerter Zeiger" besitzen. So gibt es viele Algorithmen, die auf Bereichen arbeiten und beispielsweise mit zwei Iteratoren eine Quelle und mit einem weiteren Iterator eine Daten"senke" bereitstellen (`copy()`, `transform()`, `unique_copy()`, etc.). Der als Datensenke dienende Iterator wird dabei üblicherweise mit `operator++` weitergezählt und es erfolgt eine Zuweisung mittels `operator*` und Zuweisung `=`. Ein Iterator, der in einen Container verweist, wird also sukzessive Elemente überschreiben.

Man kann jedoch auch eine Iteratorklasse konstruieren, die anstelle des Überschreibens von Elementen in einen Container einfügt oder auf die Konsole bzw. in eine Datei schreibt. Dazu müssen die entsprechenden Operatoren überladen werden. Die Standardbibliothek stellt die Klassen `back_insert_iterator`, `front_insert_iterator`, `insert_iterator`, `istream_iterator`, `ostream_iterator`, `istreambuf_iterator`, `ostreambuf_iterator` für solche Zwecke zur Verfügung. Ein Beispiel für die Anwendung eines Einfügeiterators haben wir bereits auf Seite 135 gesehen.

Das Prinzip der Implementierung sei hier am Beispiel eines Ausgabeiterators gezeigt, der Elemente in einen `ostream` ausgibt, der als Argument übergeben wird. Der Ausgabeiterator überlädt dazu `operator*` und gibt eine Referenz auf sich selbst zurück. Weiterhin wird `operator=` überladen und bewirkt den gewünschten Aufruf von `operator<<` für den auszugebenden Typ. Daher kann eine Zuweisung die rechte Seite in das Stream-Objekt ausgeben.

```
#include <iostream>
#include <algorithm>

template <class T>
struct OStreamIterator {
public:
  OStreamIterator(ostream &o) : stream(o){};

  OStreamIterator & operator++(){
    return *this;
  }

  void operator=(const T & element){
    stream << element << " ";
  }

  OStreamIterator & operator*(){ return *this; }
private:
  ostream &stream;
};

int main(){
  OStreamIterator<int>  print_me(std::cout);

  int a[] = {1, 4, 9, 16};
  // gib Elemente in umgekehrter Reihenfolge aus
  std::reverse_copy(a,a+4,print_me);
}
```

Den Ausgabeiteratoren der Standardbibiothek kann man noch ein zweites Argument mitgeben, das als Trennzeichen zwischen zwei Elementen ausgegeben wird. Als Templateargument ist wie oben der Typ der auszugebenden Klasse erforderlich.

```
#include <iterator>
```

```
//...

void f(){
  //...
  // Leerzeichen als Trennung
  std::ostream_iterator<int>  print_me(cout," ");
  //...
}
```

5.1.6 Nachtrag zur Motivation

Der bisher beschriebenen Konzepte der STL sind beim ersten Blick recht komplex. Auf den zweiten Blick ist der Aufbau der Bibliothek aber nicht kompliziert: Wir haben es mit einer Ansammlung von Containern und Algorithmen zu tun, die mittels verallgemeinerter Zeigertypen auf die Containerelemente zugreifen. Alles funktioniert zusammen, weil sich alle diese Template-Funktionen und -Klassen an einige gemeinsame Regeln halten (im Wesentlichen die Iteratoreigenschaften und das gemeinsamen Containerklasseninterface).

Wir sind jetzt auch in der Lage, diese grundsätzliche Konstruktionsidee der STL besser zu verstehen. Sie löst nämlich ein fundamentales Problem von Bibliotheken mit einen gewissen Anspruch an universelle Verwendbarkeit. Seien etwa n verschiedene Datentypen gegeben (`int, double, void *,` ...), m verschiedene Container (`vector<>, list<>,` ...) und k Algorithmen (`sort, unique,` ...). Bei herkömmlicher Programmierung müsste man $n \times m \times k$ verschiedene Implementierungen schreiben, um alle möglichen Kombinationen abzudecken. Templates erlauben, die Algorithmen für beliebige Typen zu implementieren. Wenn man weiterhin vereinbart, dass alle Container und Iteratoren eine gewisse Minimalschnittstelle unterstützen, so brauchen wir jeden Algorithmus auch wirklich nur einmal zu implementieren. Diese Minimalanforderungen an die beteiligten Klassen sind in verschiedene Gruppen gegliedert, so dass nicht immer alle Algorithmen auch mit allen Containern zusammenarbeiten.

Allerdings existieren in solchen Fällen in Abweichung vom heiligen Prinzip häufig zusätzliche Elementfunktionen der Container mit verwandter oder gleicher Funktionalität (z.B. funktioniert `sort` nur mit random access Iteratoren, die z.B. von `list` nicht bereitgestellt werden; dafür hat `list` die Element-Funktion `sort`).

5.2 Benutzung der Klasse `std::string`

Die Darstellung und Manipulation von Zeichenketten erfolgt in C durch Anlegen und spätere Bearbeitung von Puffern, auf die mit `char *` verwiesen wird. Bei deren Bearbeitung werden Funktionen verwendet, an die die Puffergröße übergeben werden muss oder die korrekt mit `\0` terminierte Pufferinhalte voraussetzen. Insbesondere bei Einlesen von Zeichenketten ergibt sich jedoch häufig das Problem, dass der Puffer von vorneherein auf eine bestimmte Größe limitiert werden muss und dass damit bei längerer Eingabe entweder Zeichen verloren gehen oder ein Pufferüberlauf mit möglicherweise schlimmen Folgen (etwa Sicherheitslücken bei Systemprogrammen) eintritt.

Als Alternative bietet C++ daher zur Repräsentation von Zeichenketten die Klasse `std::string` an, die alle Speicherverwaltung intern erledigt, Kenntnis über die Zahl der gespeicherten Zeichen hat und viele Bearbeitungsfunktionen in den Elementfunktionen vereint. Darüberhinaus stellen `string`s wie die bisher besprochenen Container der STL eine Iterator-Schnittstelle bereit, so dass sich die Algorithmen der STL auch auf `string`s anwenden lassen.

Intern ist die Implementierung von `string`s komplex und basiert zunächst auf einer Templateklasse `basic_string<>`, die zur Instantiierung mit den Datentypen `char` oder `w_char`

vorgesehen ist, die an sich entweder 8 oder 16 bit Zeichensätze repräsentieren. Alle die für `string` relevanten Eigenschaften dieser Zeichensätze sind in der Klasse `char_traits<>` beschrieben, die ebenfalls bei der `string`-Instantiierung angegeben werden muss. Diese Klasse enthält neben `typedef`s wie `pos_type` oder `char_type` auch Funktionen, die einzelne Zeichen oder auch Zeichenketten anordnen (Gleichheit und Kleiner-Relation), die Zeichen und Zeichenketten effizient kopieren, verschieben (wie etwa `memcopy` oder `memmove` aus der C-Bibliothek) und auffinden, die die Länge von Zeichenketten bestimmen, die nach bzw. von `int` wandeln und die das spezielle Zeichen `EOF` (end of file) definieren. Diese Klasse abstrahiert die Eigenschaften von Zeichensätzen so weit, dass ein `string` letztlich auch Multibyte-Text repräsentieren kann, in dem verschiedene Zeichen durch ein oder mehrere "bytes" vertreten sein können. Die Manipulation der Zeichenketten erfolgt immer über diese `char_traits<>` Klasse, so dass die binäre Repräsentation der Zeichen für den Benutzer der Klasse versteckt wird.

Hinter einem `std::string` versteckt sich ein `typedef`, der `basic_string<>` für `char` mit einer Traitsklasse koppelt, die bewirkt, dass `std::string` im Vergleich mit C-Zeichenketten letztlich keine (unangenehmen) Überraschungen liefert. Auf `std::string` wollen wir uns im Folgenden konzentrieren und für eine weitergehende Diskussion aus Platzgründen z.B. auf das Buch von Josuttis [8, 9] verweisen.

5.2.1 Benutzungbeispiel: Manipulation von Dateinamen

Als einfaches Beispiel zeigen wir hier die Manipulation von Dateinamen mit `string`, das wir einem Beispiel in [9] angelehnt haben.

Ein Programm, das auf der Kommandozeile z.B. mit `prog name1.dat name2.dat` aufgerufen wird, stellt diese Informationen intern in einem Feld von Zeigern auf `char` zur Verfügung (Typ `char *[]`). das als optionales zweites Argument der `main` Funktionen erreichbar ist. Der erste (nullte) Eintrag in diesem Feld ist der Programmname selbst, danach folgen die Kommandozeilenargumente der Reihe nach. Das erste Argument von `main` ist ein `int`, das die Anzahl der Einträge in diesem Feld liefert.

```
#include <string>
#include <fstream>

int main( int argc, char *argv[] )
{
  // Erzeugung eines temporaeren Dateinamens aus dem ersten
  // Argument

  // Kommandozeilenargumente sind regulaere C-Zeichenketten
  // Konversion von char * in einen string:
  if ( argc < 2 ) return 1; // wir brauchen ein Argument...
  const std::string filename(argv[1]);
  const std::string suffix("tmp");

  // find findet die Position eines Zeichens, rfind sucht
  // ab Zeichenkettenende
  std::string::size_type pos = filename.find('.');

  // default ctor: Zeichenkette leer
  std::string basename, extname, tmpname;
  // string::npos kennzeichnet eine unmoegliche Position
  if ( pos == std::string::npos ) {
    // '.' nicht gefunden, keine Endung, + fuegt zusammen
```

```
    tmpname  = filename + '.' + suffix;
  }
  else {
    // '.' gefunden
    // Teilzeichenkette exkl. zweitem Argument (hier ohne '.')
    basename = filename.substr(0,pos);
    // fehlendes zweites Argument entspricht Ende des string
    extname  = filename.substr(pos+1);
    if ( extname.empty() ) { // hoppla, nur '.', keine Endung
      tmpname = filename;
      tmpname += suffix;     // Anhaengen
    }
    else if ( extname == suffix ) { // endet auf .tmp, == ueberladen
      tmpname = filename;
      // Ersetzen der existierenden Endung, die ersten
      //  zwei Argumente kennzeichnen den zu ersetzenden Teil
      tmpname.replace( pos+1, extname.size(), "xxx" );
    }
    else {
      // jede andere Endung bis Stringende durch suffix ersetzen:
      tmpname = filename;
      tmpname.replace(pos+1, std::string::npos, suffix);
    }
  }
  // Oeffnen einer backup Datei, c_str() ist \0 terminiert
  std::ofstream tmpfile(tmpname.c_str());
  // .. use it
  tmpfile << tmpname;
}
```

Der Zugriff auf die Einzelzeichen ist oben nicht gezeigt, lässt sich aber wie in C über den `operator[](size_type)` erreichen. Die Elementfunktion `at(size_type)` hat den gleichen Zweck, führt aber vor dem Zugriff einen Test auf die Gültigkeit des Indexes aus und wirft eine Exception, wenn dieser Test fehlschlägt (vgl. Kapitel 6).

Die Elemente `find`, `replace` und andere Elementfunktionen von `string` arbeiten auf der gesamten Zeichenkette oder auf Teilzeichenketten, die entweder in C-Weise durch einen Zeiger auf das erste Element, oder bei `string`s durch eine Position (d.h. durch ein `string`-Objekt und einen Wert des Typs `std::string::size_type`) und in beiden Fällen die Länge der Kette repräsentiert werden. Die erste Position entspricht der 0, die Position nach der letzten gültigen und die Länge einer möglichst langen Kette wird durch den Wert `std::string::n_pos` repräsentiert. Bei Beschreibung einer Teilzeichenkette durch einen einzelnen Wert ist als zweiter (Längen-)Wert immer `std::string::n_pos`, also die restliche Teilzeichenkette, impliziert.

Operatoren, die wie `operator==` auf ganzen Zeichenketten arbeiten, sind in der Regel auch für nullterminierte C-Zeichenketten als linker oder rechter Operand überladen.

Die folgende Liste von Elementfunktionen zeigt die wichtigsten, aber nicht alle überladenen Formen. Diese lassen sich aber in der Regel durch Analogien erschließen oder durch Graben in der Deklarationsdatei `string` und den eingeschlossenen Dateien finden.

Tabelle 5.3: **Elementfunktionen der Klasse `std::string`.**

`string(size_type n, charT c)`	string aus n Einzelzeichen c
`string(const charT *)`	string aus C-string

Tabelle 5.3: **Elementfunktionen der Klasse `std::string`.**

`string(const charT *, size_type n)`	string aus ersten n Zeichen
`string(const string &, size_type b, size_type e)`	aus Zeichen zwischen `b` und `e`
`iterator begin()`	Iterator-Interface
`iterator end()`	Iterator-Interface
`const_iterator begin() const`	Interator-Interface
`const_iterator end()const`	Iterator-Interface
`bool empty()`	Zeichenkette leer?
`size_type size()`	Anzahl der Zeichen
`size_type length()`	Anzahl der Zeichen
`size_type max_size()`	implementierungsabhängige Konstante
`size_type capacity()`	alloziierter Speicher
`void reserve(size_type n)`	Vor-Anforderung für `n` Zeichen
`void clear()`	alle Zeichen löschen
`reference operator[](size_type pos)`	Zugriff auf Einzelzeichen
`const_reference operator[](...) const`	dto., nur lesend
`reference at(size_type pos)`	Zugriff auf Einzelzeichen
`operator+=(const string &)`	Anhängen einer zweiten Kette
`operator+=(charT *s)`	Anhängen
`append(const string &)`	Anhängen
`append(const charT *s, size_type n)`	`n` Zeichen von s
`push_back(charT c)`	Anhängen des Zeichens `c`
`size_type find(const string &s, size_type pos=0)`	sucht Kette `s` ab `pos`
`size_type find(charT c, size_type pos=0)`	sucht Zeichen `c` ab `pos`
`insert(size_type pos, const string &s)`	Einfügen von `s` vor `pos`
`erase(size_type pos)`	Löschen des Zeichens an `pos`
`erase(size_type p0, size_type p1)`	Löschen der Zeichens von `p0` bis ausschließlich `p1`
`replace(size_type p0, size_type p1, const string &s)`	Ersetzen der Teilkette zwischen `p0` und `p1` durch `s`
`substr(size_type p0=0, size_type p1=npos)`	Kopie der Teilkette zwischen `p0` und `p1`

Für `string`s sind die Ein- und Ausgabeoperatoren `>>` und `<<` in der gewohnten Weise überladen. Bei der Ausgabe wird dabei die gesamte Zeichenkette ausgegeben, bei der Eingabe werden normalerweise führende Leerzeichen (whitespace) übersprungen (außer es wird das Flag `skipws` im Stream gesetzt) und es wird bis zum ersten Leerzeichen nach einer Kette von Nicht-Leerzeichen (also ein Wort) gelesen.

Häufig ist es empfehlenswert, bei Einlesen aus einem Stream die Information zeilenweise einzulesen und zu verarbeiten. So kann in einfacher Weise Fehlerbehandlung erfolgen oder mehrfache Versuche unternommen werden, den Inhalt einer Zeile zu interpretieren. Zum Einlesen einer Zeile dient die Funktion `std::getline(std::istream &, std::string &, charT separator='\n')`, die als Zeilenende `'\n'` erwartet. Zulässig sind aber beliebige Zeichen, die beim Einlesen automatisch entfernt werden.

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

void f(){
```

```
  std::string s;
  while ( getline(std::cin,s0 ) {
    // liest zeilenweise von std::cin
    // ...
  }
}
```

F: Ich benötige für die Dateinamen in einem gewissen Betriebssystem einen Zeichensatz, der keinen Unterschied bezüglich Groß- und Kleinschreibung macht.
A: Dies kann erreicht werden, indem man eine eigene `char_traits<char>`-Klasse implementiert, die gegenüber der Standardversion eine spezielle Funktion zum Vergleich von Zeichen enthält. Ein `basic_string<>`, der mit `char` und dieser Traits-Klasse instantiiert wird, verhält sich wie gewünscht.

5.3 valarray

Die Standardbibliothek enthält zur Unterstützung numerischer Operationen auf großen Feldern typgleicher Elemente den Container `valarray`. Man kann ihn sich wie eine Version von `vector` vorstellen, dessen Länge sich einmalig dynamisch festlegen lässt, der aber daraufhin optimiert ist, dass sich diese Länge später nicht mehr ändert.

```
#include <valarray>

int main(){

  std::valarray<double>   A(0.,10); // Anfangswert 0., 10 Elemente
  std::valarray<double>   B;        // uninitialisiert;
  B.resize(10);                     // 10 Elemente, default initialisiert
  int a[] = {0,1,2,3,4,5,6,7,8,9,10,11,12};
  std::valarray<int> C(a,10);       // erste Elemente von a uebernommen
  std::valarray<int> D(C);          // Kopie der Elemente in C

}
```

Ähnlich wie für FORTRAN90-Felder und viele Datenmanipulationssprachen (Matlab, IDL) sind alle Standardoperatoren und Funktionen, die sinnvoll auf den Feldelementen ausgeführt werden können, als Funktionen des Feldes selbst überladen. Die Schleifen über die Elemente des `valarray`, die zur Abarbeitung der Operationen nötig sind, werden in der Implementierung versteckt. Dabei lassen Funktionen und Operatoren das Feld in der Regel unverändert und man muss vielmehr durch Zuweisungen klarmachen, was mit dem Resultat geschehen soll.

```
  A    = 2.;      // weist 2. auf alle Elemente zu
  A   *= 0.1;     // multipliziert alle Elemente mit 0.1
  A[9] = 42.;     // ueberladener operator[] zum Zugriff auf Elemente
  B    = A;       // Zuweisung von Feldinhalten,
                  // VORSICHT: kein Fehler bei unpassender Groesse

  A.sum();        // Summe aller Elemente
  A.max();        // Maximum aller Elemente
  A.min();        // Minimum aller Elemente
  A = std::cos(A); // Jedes Element wird durch seinen cos ersetzt
```

```
  B.shift(2);     // Verschieben aller Elemente,
                  //  Einfuegen von Nullen am Rand
  B.cshift(2);    // Zyklisches Verschieben
                  //  negative Argumente schieben in Gegenrichtung

  B += A;         // elementweise Addition
  B += 2.;        // Addition einer Konstanten
  D  = C << 2;    // Bitmuster der Werte in C um je 2 Bits verschoben
  D  = 2 << C;    // neues valarray aus 2en, die mit Werten aus C
                  //  verschoben wurden
  D  = D << C;    // jeder Wert aus D mit dem korrespondierenden Wert
                  //  aus C verschoben.
  D  <<= C;       // aequivalent, aber im Zweifel effizienter
```

5.4 Anwendungsbeispiele für STL Klassen

5.4.1 Zweidimensionale Felder

Wegen ihrer Bedeutung in numerischen Berechnungen wollen wir hier auf verschiedene Möglichkeiten eingehen, in C++ mehrdimensionale Felder zu verwirklichen. Dabei demonstrieren wir jeweils das Prinzip am zweidimensionalen Beispiel. Jede Methode hat Vor- und Nachteile, die der Programmierer im Einzelfall abwägen muss. Wir wollen für das Folgende vereinbaren, dass der erste Feldindex als x aufgefasst werden soll, dessen Variation uns durch die "Spalten" des Feldes fortbewegt.

C-Felder

Die einfachste Möglichkeit, zweidimensionale Felder zu deklarieren, ist die aus C bekannte, die wir bereits in Abschnitt 2.3.4 angesprochen haben:

```
const int xsize = 20;
const int ysize = 30;

int main(){
  double   array[xsize][ysize];

  for ( int x=0; x < xsize; x++ )
    for ( int y=0; y < ysize; y++ )
       array[x][y] = x * y;
}
```

Allerdings müssen auch in C++ dafür die Feldgrenzen bereits bei der Übersetzung des Programmes festliegen, was häufig eine nicht akzeptable Einschränkung bedeutet.

Mit der `valarray<>`- oder auch der `vector<>` Klasse der STL lässt sich ein dynamisches zweidimensionales Feld in der folgenden Form "rekursiv" definieren.

```
#include <valarray>
using std::valarray;

int main(){
  int xsize, ysize;
  //... get size from somewhere
  xsize = 20, ysize = 30;
```

```
  valarray<int>  tmp(ysize);  // Spaltenprototyp, Default-Werte
  // Wiederholung von 'tmp' insgesamt xsize mal
  valarray< valarray<int> > array(tmp, xsize);

  for ( int x=0; x < xsize; x++ )
    for ( int y=0; y < ysize; y++ )
      array[x][y] = x * y;
}
```

Der aus C gewohnte Zugriff auf Elemente in der Form `[][]` ist möglich, weil der erste Zugriff als Ergebnis einen Referenz auf ein `valarray<>` zurückliefert, das eine Spalte repräsentiert. Dieses kann dann nochmals mit `[]` indiziert werden, was schließlich eine Referenz auf ein Feldelement liefert. Gegenüber C-Feldern ergibt sich der Vorteil, dass die Größe zur Laufzeit festgelegt werden kann. Bei Verwendung von `valarray` stehen weiterhin alle arithmetischen Operationen auch als Operationen auf den Feldelementen zur Verfügung. Als möglicher Nachteil für einige Anwendungen muss allerdings festgehalten werden, dass das Speicherlayout nicht festliegt, da es sich bei `vector<>` und `valarray<>` um Klassen handelt, die in der Wahl ihres Speicherlayouts keinerlei Beschränkungen unterliegen. Weiterhin wird die Größeninformation getrennt für jede vektorwertige Spalte gespeichert, was in einem geringfügig höherem Speicherbedarf gegenüber der C-Lösung resultiert.

Realisierung mit Zeigern

Mit Zeigern lassen sich dynamisch mehrdimensionale Felder verwalten, die sich wie die eingebauten Felder mit konstanten Bereichsgrenzen verhalten. Nachdem das Programm ermittelt hat, wieviel Speicher benötigt wird, kann man einen zusammenhängenden Speicherbereich anfordern und ein Feld von Zeigern der um eins verminderten Dimension benutzen, um die mehrfache Anwendung von `[]` zu unterstützen. Dies ist unten am Beispiel eines zweidimensionalen Feldes demonstriert. Diese nachträgliche Ver-Zeiger-ung eines zusammenhängenden Speicherbereiches garantiert volle Kontrolle über die Anordnung der Feldelemente im Speicher. Der Zugriff kann wahlweise durch `[][]` oder über Indexarithmetik und einfaches `[]` erfolgen.

```
int main(){

  int  xsize, ysize;
  // kommen von irgendwoher...
  xsize = 20, ysize = 30;

  // Speicher am Stueck holen
  int *fp1    = new int [xsize*ysize];
  // zweidimensional machen
  int **array = new int *[xsize];
  for ( int i = 0; i < xsize; i++ )
    array[i] = fp1 + i * ysize;

  // array laesst sich wie ein 2D Feld verwenden
  for ( int x=0; x < xsize; ++x )
    for ( int y=0; y < ysize; ++y )
      array[x][y] = x * y;
}
```

Diese Lösung erlaubt die volle Kontrolle über das Speicherlayout, erfordert allerdings gegenüber der reinen C-Lösung zusätzlichen Speicher für die Verzeigerung der Spalten des

Feldes. Besser wäre natürlich auch das Verpacken der unschönen Speicherzugriffsarithmetik in eine Klasse.

Realisierung mit automatisierter Index-Arithmetik

Um auch diesen letzten Schritt zu gehen und keinen zusätzlichen Speicheraufwand bei voller Kompatibilität zu C-Feldern zu haben, muss man letztlich eine eigene Klasse implementieren. Dabei muss der Zugriffsoperator `operator[]` der Klasse überladen und ein Trick angewandt werden, um für den letztendlichen Speicherzugriff die Werte beider Indizes zur Verfügung zu haben. Information über Schrittweite und Wert des Index speichern wir in einem Stellvertreter(Proxy-)objekt. Dieses kann dann seinerseits den `operator[]` überladen und in einem zweiten Schritt die notwendige Indexarithmetik durchführen und eine Referenz auf ein Feldelement liefern:

```
#include <iostream>

template <class T>
struct Array2D {  // 2D array class

  Array2D(int a_xsize, int a_ysize) : ysize(a_ysize) {
    // ... Argumenttests
    mem = new T[a_xsize*a_ysize]; // Speicherblock holen
  }
  ~Array2D(){
    delete[] mem;                 // Loeschen im Destruktor
  }

private:
  // Hilfsklasse zum Zwischenspeichern der
  //  Information im ersten Index
  struct Proxy {
    Proxy(int a_offset, T* &a_array):
       offset(a_offset), array(a_array){}

    T & operator[](int j){
      return array[offset + j];
    }
  private:
    int offset; // springe ueber Elemente der 1. Dimension
    T* &array;  // Feld fuer letztendlichen Zugriff
  };

public:
  Proxy operator[](int i){      // x-index sagt wieviele Spalten
    return Proxy(ysize*i,mem);  //  uebersprungen werden muessen
  }

private:
  int ysize;
  T *mem;
};

int main(){
```

```
  int xsize = 5;
  int ysize = 7;

  Array2D<int>  v(xsize,ysize);
  for (int x=0; x < 5; ++x )
    for ( int y=0; y < 7; ++y )
      v[x][y] = x * y;
}
```

Mit den Techniken der Template-Metaprogrammierung lässt sich das oben stehende Konzept in beliebige Dimensionen erweitern.

Für numerische Zwecke sollte man je nach Problemklasse auch entsprechende Bibliotheken in Erwägung ziehen, die viel Implementierungsarbeit ersparen können und häufig hervorragende Laufzeiteigenschaften haben. Für Matrix-Vektor-Operationen wäre in diesem Zusammenhang insbesondere die *blitz++*-Bibliothek zu erwähnen, die Methoden der Template-Metaprogrammierung anwendet, um sehr schnelle, cacheoptimierte Speicherzugriffe und arithmetische Operationen zu erreichen.

Übergabe als Argument

Wie wir gesehen haben, gibt es eine Vielzahl von Möglichkeiten, das Konzept Feld bzw. seinen Teilaspekt als Größe, die sich mehrfach durch `operator[]` indizieren lässt, zu implementieren. Es ist daher sicherlich nicht sinnvoll, Algorithmen zu implementieren, die sich speziell auf die eine oder andere Art und Weise der Felddeklaration festlegen. Es bieten sich daher Templates als Funktionsargumente an,

```
template <class T>
int rank( T &array ){
  int rank;
  // Rang der Matrix berechnen, die durch array repraesentiert wird
  // ...
  return rank;
}
```

die die Verwendung mit allen obigen Deklarationsweisen zulassen.

5.5 Übungen

Ziel dieser Übung ist das Kennenlernen einiger Eigenschaften der *Standard Template Library* `STL`. Da die `STL` fertige Bausteine für die Übungsaufgaben enthält, geben wir keine Programmbausteine vor.

5.5.1 Die Klasse `vector`

a) Schreiben Sie ein Programm, das solange ganze Zahlen vom Terminal einliest, bis der Anwender eine negative Zahl eingibt. Danach gibt es die Zahlen in derselben Reihenfolge aus. Optional können Sie dieses Programm auch mit herkömmlicher Programmierung schreiben. Stellen Sie aber sicher, dass der Anwender *beliebig viele* Zahlen eingeben darf.

b) In einer verbesserten Version sollen die Zahlen aufsteigend sortiert ausgegeben werden.

5.5.2 Der `map`-Container

Schreiben Sie mit Hilfe des `map` Containers ein kleines Telefonbuchprogramm.

a) Im einfachsten Fall enthält das Programm ein paar feste Einträge und gibt bei Eingabe eines Namens die zugehörige Telefonnummer aus, oder eine Meldung, dass der Teilnehmer nicht eingetragen ist. Erinnerung: `map.find(`*`key`*`)` liefert den Iterator `map.end()` falls *key* nicht gespeichert ist.

b) Geben Sie auf Wunsch auch eine komplette Telefonliste aus.

5.5.3 Algorithmen

Zum Testen einiger Algorithmen der `STL` erweitern wir das Programm aus Aufgabe 1.

a) Schreiben Sie ein Funktionsobjekt `SumIt`, dass mit Hilfe des `for_each` Algorithmus die Summe der gespeicherten Zahlen bildet. Geben Sie die Summe zu Kontrollzwecken aus und bilden Sie die Summe auch auf "klassische Art".

b) `remove(`*`anfang`*`,`*`ende`*`,`*`wert`*`)` entfernt alle Elemente mit dem Wert *wert*. Analog vereinigt `unique(`*`anfang`*`,`*`ende`*`)` alle *aufeinanderfolgenden* identischen Elemente zu einem Element. Verändern Sie das Programm, damit es alle mehrfach auftretenden Zahlen nur einmal ausgibt.

c) Lösen Sie die Aufgabe des Ausgebens der Zahlen auf mindestens drei verschiedene Weisen, die entweder eine `for(;;)` Schleife, den `for_each(,,)`-Algorithmus oder Ausgabeiteratoren verwenden.

5.5.4 Iterator-Gültigkeit

Schreiben Sie einen Algorithmus im Sinne der STL, der alle Elemente mit einem bestimmten Wert `<value>` beliebigen Typs aus einem Container entfernt. Dabei ist zu beachten, dass eine Modifikation des Containers üblicherweise zur Ungültigkeit der in den Container verweisenden Iteratoren führt. Testen Sie Ihren Algorithmus mit folgenden Beispielen:

- einem `vector`, der 10 gleiche Elemente des Wertes `<value>` enthält.

- einem `set`, mit 20 zufällig angeordneten Elementen von 2 Werten `<value1>` und `<value2>`, indem Sie zunächst alle Elemente des Wertes `<value1>` und dann die des Wertes `value2>` entfernen. Lassen Sie sich zwischenzeitlich den Inhalt des Containers zur Kontrolle ausgeben.

Kapitel 6

Exceptions

6.1 Exceptions zur Fehlerbehandlung

Das bei der Fehlerbehandlung auftretende Grundproblem ist, dass Fehler häufig auf einer anderen Ebene auftreten als der, auf der das Wissen vorhanden ist, wie man mit dem Fehler umgehen kann. Zum Beispiel kann eine typische Bibliotheksroutine zwar einen Fehler oder eine Inkonsistenz in den Argumenten einer Funktion erkennen oder etwa das Nichtabbrechen einer numerischen Iteration feststellen, aber nur wenig zu deren Korrektur tun, denn nur die aufrufende Routine kann wissen, in welcher Weise auf einen Fehler reagiert werden kann: etwa durch Programmabbruch oder die Wahl eines anderen Parametersatzes.

Man begegnet diesem Problem üblicherweise mit einer der folgenden Methoden, die alle eigene Vor- und Nachteile haben.

Folgende Ansätze sind denkbar:

Ignorieren des Fehlers: Rückgabe irgendeines (legalen) Wertes und Hoffen (auf die Nachsicht der Rechenzeitkommission oder auf den hoffentlich erfolgenden Programmabsturz).

Programmabbruch: nicht optimal, aber immerhin verhindert man bei numerischen Programmen die möglicherweise sinnlose Nutzung wertvoller Ressourcen. Bei kommerziellen Programmen wird man aber sicherlich seine Umsatzzahlen im Auge behalten müssen.

Rückgabe eines Fehlerwertes: problematisch, weil häufig keine speziellen Werte vorhanden sind, die sich zur Anzeige eines Fehlers benutzen ließen oder aber der Rückgabewert vom aufrufenden Programm gar nicht überprüft wird. Ein typisches Beispiel sind numerische Routinen wie `log`, die etwa bei negativen Argumenten ein Bitmuster zurückgeben, das keine Zahl repräsentiert. Allerdings wird dann häufig (und die Mathematikroutinen erlauben das) mit diesem Wert weitergerechnet. Also: besser für den Umsatz, aber nicht unbedingt für die Autos, die über die so berechnete Brücke fahren.

Aufruf einer globalen Fehlerbehandlungsroutine: dies widerspricht oft den Grundprinzipien einer modularen oder objektorientierten Programmierung, denn diese Routine benötigt ja Information darüber, wo und in welchem Kontext der Fehler aufgetreten ist. Häufig ist das Resultat, das man eine unverständliche Fehlermeldung erhält und den Hinweis, entweder den Systemadministrator oder den Autor des Programmes zu kontaktieren.

Setzen einer globalen Fehlervariablen: problematisch, weil dieser Ansatz nicht "skaliert:" Wer entscheidet welcher Wert für welchen Fehler steht? Was passiert bei Pro-

grammen, die mit mehreren Threads arbeiten? Weiterhin wird nicht genügend nachdrücklich auf den Programmierer eingewirkt, den Wert dieser Variablen wirklich zu prüfen.

Exceptions: In C++ (und nicht nur dort) gibt es als Mittel zur Fehlerbehandlung zusätzlich Ausnahmen (Exceptions), die wir im Folgenden besprechen wollen.

Das Grundprinzip der Ausnahmebehandlung wird an folgendem Beispiel klar:

```
#include <cmath>
#include <iostream>

double safe_sqrt(double a ){
  if ( a < 0 ) throw a;
  return std::sqrt(a);
}

int main() {
  double b(42.), c(3.1415);
  // ...
  try{
     double a = safe_sqrt(b) + safe_sqrt(c);
     std::cout << "result = " << a;
  }
  catch(double &a){
     std::cerr << "invalid argument " << a << " occured";
  }
  catch(...){
     std::cerr << "unknown error occured";
  }
  // ...
}
```

Hier wird im so genannten `try`-Block versucht, eine Berechnung durchzuführen. Wird der Funktion `safe_sqrt` ein negatives Argument übergeben, so stößt diese eine Ausnahmebehandlung an (*throwing an exception*). Dazu wird ein Objekt einer beliebigen Klasse hinter der `throw`-Anweisung angegeben, von dem durch die `throw`-Anweisung eine Kopie angelegt wird. Dieses (Exception-)Objekt dient zur Findung der richtigen Fehlerbehandlungsroutine und enthält ggf. weitere Information, die für Aufräumarbeiten nötig sind. Bei Ausführung des `throw` wird die Abarbeitung der Routine an dieser Stelle unterbrochen, und es werden die Destruktoren aller Objekte aufgerufen, die seit Funktionsaufruf in allen lokalen Bezugsrahmen bis dorthin konstruiert wurden. Dieses Vorgehen setzt sich nötigenfalls durch mehrere Ebenen von Unterprogrammen fort (*stack unwinding)*, solange bis ein einschließender `try`-Block gefunden wird. Im Fehlerfall wird der `try`-Block selbst auch nicht zu Ende ausgeführt, d. h. im obigen Beispiel wird die Anweisung `std::cout << "result + " << a;` nicht ausgeführt. Existiert kein solches umschließendes `try`, so wird in letzter Konsequenz die Programmausführung beendet (s. u.).

Jetzt wird der Typ der Ausnahme der Reihe nach mit den Argumenten der mit dem `try` assoziierten `catch`-Anweisungen verglichen. Sobald dort ein passender Typ gefunden wird, wird die Verarbeitung im zugehörigen Block fortgesetzt — das Exceptionobjekt wird "gefangen". In diesem Block kann jetzt mit der dort zur Verfügung stehenden Information eine Fehlerbehandlung stattfinden. Es ist daher klar, dass man bei der Wahl der Argumente der `catch`-Anweisungen vom speziellen zum allgemeinen Fall vorgehen muss. Die `catch(...)`-Anweisung fängt alle Ausnahmen ab. Anschließende `catch`-Anweisungen würden niemals ausgeführt.

Eine Ausname verhält sich in vieler Hinsicht (Aufräumen des Stacks, Übertragung der Kontrolle an einen anderen Programmteil) wie ein kontrolliertes `goto` (vgl. auch `setjmp`, `longjmp`) aus der C-Bibliothek), mit dem sich die Programmflusskontrolle von einem beliebigen Punkt aus verschachtelten Unterprogrammen heraus an den Aufrufer übertragen lässt.

Ausnahmeobjekte lassen sich durch wiederholtes `throw` weiterreichen, etwa wenn der Fehler im `catch` Block nicht vollständig behandelt werden kann. Im obigen Beispiel etwa ließe sich schreiben,

```
b = -1.;
try{
   double a = safe_sqrt(b) + safe_sqrt(c);
   std::cout << "result = " << a;
}
catch(double &a){
  std::cerr << "invalid argument " << a << "\n";
  throw;  // altes Exception Objekt wird weitergereicht
}
catch(...){
  std::cerr << "unknown error occured";
}
```

um eine solche Weitergabe zu erreichen.

Sollte während der Stackaufräumarbeiten bei der Ausführung eines Destruktors in der Ausnahmebehandlung eine weitere Ausnahme auftreten, so bricht das Programm ab. Programmteile die potentiell Ausnahmen erzeugen, können allerdings durch Aufruf von `uncaught_exception()` prüfen, ob gerade eine unbehandelte Ausnahme im System "hängt".

Daraus ergeben sich einige Vorsichtsregeln für Programme mit Ausnahmebehandlung: Ausnahmen sollten immer als Referenzen gefangen werden, um ggf. Probleme bei der Speicheralloziierung für zu kopierende Objekte zu umgehen. Gleichzeitig ermöglicht dies die Nutzung von polymorphen Fehlerobjekten. Weiterhin sollte jede Form expliziter Nutzung von `new` und `delete` mit Argwohn betrachtet werden, denn ein explizites `delete` wird möglicherweise bei Eintreten einer Ausnahmebehandlung nicht mehr ausgeführt. Dieses Problem erfordert spezielle Objekte, die `delete` "automatisch" aufrufen können, wenn der Programmfluss den definierenden Block verlässt (siehe Abschnitt 6.3). Ähnliche Einschränkungen gelten für andere Formen der Ressourcenanforderung und -freigabe.

In manchen Situationen lässt sich die Ausnahmebehandlung vereinfachen, indem man die Klassen möglicher Fehler hierarchisch organisiert. Man kann sich dann bei der Fehlerbehandlung des Polymorphismus bedienen und muss so den Fehlerbehandlungscode bei Einführung neuer Fehler nicht ändern.

```
#include <limits>
#include <iostream>

 struct MathErr {
   virtual void debug_print(){
     std::cerr << "Math error";
   }
 };

 struct IntOverflow: public Matherror {
   virtual void debug_print(){
     std::cerr << "Int Overflow error";
   }
```

```
};

struct DivByZero: public MathErr {
  virtual void debug_print(){
    std::cerr << "Division by Zero";
  }
};

int add(int a, int b){
  if (  (a > 0 && b > 0 && a > std::numeric_limits<int>::max - b )
     || (a < 0 && b < 0 && a < std::numeric_limits<int>::min + b ) )
    throw IntOverflow();
  return a + b;
}

int divide(int a, int b){
  if ( b == 0 ) throw DivByZero();
  return a / b;
}

main(){
  // ...
  try{
    result = div(a, add(a,b));
  }
  catch ( MathErr &m ) {   // passt auf alle MathError Ausnahmen
    m.debug_print();
  }
}
```

In den Ausnahmeobjekten lässt sich Information verpacken, die ggf. im aufrufenden Programm benötigt wird, um den Fehler zu behandeln.

F: Kann man die Information, in welcher Klasse der Fehler aufgetreten ist, in der Exception speichern?
A: Ausnahmen können natürlich nicht nur in Elementfunktionen, sondern an beliebigen Stellen des Programmes generiert werden. Wenn Typinformation benötigt wird und zugänglich ist (wie z.B. in der Elementfunktion einer Klasse), dann kann diese natürlich in einer speziellen Ausnahme gespeichert werden und zwar auf vielerlei Weise: (i) im Typ der Ausnahme selbst (diese könnte ein Template sein, das einen `typedef` enthält), (ii) in einem `std::string`, der den Klassennamen speichert, (iii) in einem `std::typeinfo` Objekt, das einen Typnamen in möglicherweise kodierter, aber eindeutiger Form enthält. Eine Referenz auf ein solches Objekt wird durch den Operator `typeid` bereitgestellt, der wie `sizeof` auf ein Objekt, einen Typ oder einen Ausdruck angewendet werden kann. `Typeinfo` Objekte können mit `==` verglichen werden.

6.1.1 Exceptions in Ein- und Ausgabe

Die Ein- und Augabeklassen kennen die `std::ios_base::failure` Ausnahme. Genauere Informationen lassen sich aus den Statusbits der ios-Klasse ableiten. Das Auswerfen von Exceptions lässt sich durch Aufrufen der Elementfunktion `exceptions` dieser Klassen aktivieren. Als Argument gibt man ihm das Bitmuster der Statusbits, die eine Exception

auslösen sollen, wenn sie gesetzt werden. So werden nach dem Aufruf `std::cin.exeptions (std::ios::eofbit | std::ios::failbit)` Ausnahmen generiert, falls das `eofbit` oder das `failbit` gesetzt werden. Die Elementfunktion `exceptions()` ohne Argumente liefert als Rückgabewert das Bitmuster der Bits, die eine exception auslösen.

6.1.2 Exception des `operator new()`

Der `operator new()` wirft die Ausnahme `bad_alloc`, falls eine Speicheranforderung fehlschlägt. Sollte diese unbehandelt bleiben, schreibt das Programm eine core-Datei und bricht ab. In C und frühen C++-Versionen hat man statt der Ausnahme einen Null-Pointer als Rückgabewert erhalten. Oft sieht man daher noch Code der Form

```
// VORSICHT: fehlerhafter Code
double *p0 = new double[256];
if ( p0 )
  ; // verwende p0
else
  std::cerr << "Probleme bei der Speicheranforderung \n";
```

Da laut neuem C++-Standard im Fehlerfall eine Ausnahme erzeugt wird, wird der `else`-Zweig hier niemals ausgeführt. Vielmehr wird letztlich `terminate()` aufgerufen, wenn kein spezielle Fehlerbehandlungsroutine mit `std::set_unexpected( void(*)() )` (s. u.) gesetzt wurde.

```
double *p1 = 0;
int size = 256;
while ( !p1 ) {
  try { p1 = new double[size]; }
  catch ( std::bad_alloc & ) {
    size /= 2;  // uh-oh, Gier etwas zuegeln
  }
}
// verwende p1
```

6.2 Exceptions in Funktionsdeklarationen

Man kann eine Funktion explizit mit der Information versehen, welche Ausnahmen sie generieren kann:

```
void f() throw( MathError, bad_alloc );
void g() throw();
void h();
```

Die Funktion `f()` verspricht, außer `MathError` und `bad_alloc` keine andere Ausnahmen zu werfen. Dies umfasst natürlich auch von `MathError` oder `bad_alloc` abgeleitete Ausnahmen. Die gewöhnliche Signatur von `h()` bedeutet, dass prinzipiell alle Ausnahmen auftreten können.

Im Gegensatz dazu meint `g()`, dass sie niemals eine Ausnahme auswirft. Sollte das dennoch passieren, so wird eine Funktion aufgerufen, die man im Programm durch Aufruf von `std::set_unexpected( void (*)() )` spezifizieren kann. Diese Funktion nimmt keine Argumente und darf nicht zurückkehren. Als Default wird `terminate()` aufgerufen, was eine `core` Datei schreibt und das Programm beendet.

Es ist dabei zu beachten, dass nicht nur diejenigen Ausnahmen spezifiziert werden müssen, die die Funktion selbst möglicherweise generiert, sondern auch diejenigen, die in

expliziten oder impliziten Funktionsaufrufen entstehen können. Im Beispiel kann etwa im Konstruktor der Klasse `string` durch Speichermangel der `new operator` eine `bad_alloc`-Ausnahme auslösen.

```
void g() throw()  // VORSICHT: problematische Deklaration
{
 std::string s("a potential problem");
 // ...
 return;
}
```

6.3 Vermeiden von Resource Leaks mit Auto Pointern

Wie lässt sich in Gegenwart möglicher Ausnahmen erreichen, dass einmal angeforderte Ressourcen wieder korrekt freigegeben werden? Nun, einerseits garantiert uns der Compiler, dass bereits initialisierte automatische Variablen korrekt durch Destruktoraufruf vernichtet werden. Probleme entstehen, wenn wir vielleicht in der Zwischenzeit Speicher durch `new` angefordert haben, oder etwa eine Datei durch den Aufruf einer C-Bibliotheksfunktion geöffnet wurde. Dann wird während des Stack-Aufräumprozesses weder der Speicher wieder freigegeben noch die Datei geschlossen.

Nun, hätten wir zum Lesen aus der Datei ein `ifstream` Objekt verwendet, so wäre das Schließen kein Problem gewesen, denn das passiert als Bestandteil des Destruktors des Objektes automatisch, der ja garantiert aufgerufen wird. Dies ist ein allgemeines Muster (Konstruktor fordert Ressourcen an, Aktionen werden durchgeführt, Destruktor gibt Ressourcen wieder frei), das sich in vielen Klassen findet und uns auch hier einen Hinweis gibt, wie wir das Problem der Freigabe des Speichers lösen können.

Statt direkt einen Zeiger zu verwalten, können wir ein Hilfsobjekt erzeugen, das sich den Zeiger merkt und "automatisch" ein `delete` aufruft, wenn das Hilfsobjekt, z.B. während des Stack-Aufräumens, vernichtet wird. Die Standard C++-Bibliothek stellt dazu die Hilfsklasse `auto_ptr` bereit. Wird ein solches Objekt vernichtet, wird dabei auch das verwaltete Objekt zerstört, auf das verwiesen wird.

```
// memory.h:
namespace std {

 template <class T>
 class auto_ptr {
 public:
   explicit auto_ptr(T* aptr=0) throw() : ptr(aptr){}
   ~auto_ptr(){ delete ptr; }
   auto_ptr & operator=(auto_ptr &rhs){
      ptr = rhs.ptr;
      rhs.ptr = 0;
   }

   //...

 private:
   T *ptr;
 };

 // ...
}
```

```
// main.cc:
#include <memory>

class A {
  // ...
};

int main()
{
  std::auto_ptr<A> ptr(new A);
  //...
  bool condition;
  //...
  if ( condition ) throw 42;   // ok, delete wird aufgerufen
  //...
}
```

Die für das Arbeiten mit Zeigern auf skalare Objekte nötigen Operatoren `*`, `->` sind überladen, so dass sich `auto_ptr` im Allgemeinen wie ein regulärer Zeiger verhält.

Eine weitere Verwendung von `auto_ptr` findet sich in allen Objekten, die andere Ressourcen über Zeiger verwalten. Um in deren Konstruktoren zu erreichen, dass bei Auftreten einer Ausnahme richtig aufgeräumt wird, müssen `auto_ptr` oder ähnliche Hilfsobjekte verwendet werden.

6.3.1 Kopieren von auto_ptr

Ein Problem entsteht beim Kopieren der `auto_ptr`, denn es muss sichergestellt sein, dass das verwaltete Objekt nur durch genau einen Aufruf von `delete` zerstört wird, d. h. dass sich der Speicher immer nur im Besitz genau eines `auto_ptr` Objektes befindet.

`auto_ptr` übergibt das verwaltete Objekt beim Kopieren in den Besitz der frisch erstellten Kopie; bei Zuweisungen wird analog die linke Seite Besitzerin des verwalteten Objektes. Als Konsequenz muss bei der Übergabe von Auto-Pointern *by value* in Funktionsaufrufen vorsichtig vorgegangen werden, denn wenn der Auto-Pointer nicht wieder als Rückgabewert an die aufrufende Umgebung zurückgegeben wird, so wird das bezogene Objekt mit dem Funktionsargument bei Rückkehr der Funktion zerstört. Es ist sicher, eine Referenz auf einen `auto_ptr` zu übergeben oder einen `auto_ptr` als Rückgabetyp zu verwenden.

Hier ist ein Beispiel für das Verhalten von Auto-Pointern beim Kopieren oder Zuweisen:

```
#include <memory>
#include <iostream>

std::auto_ptr<int> f()
{
  // p1 ist Eigentuemer nach Konstruktion des Objektes
  std::auto_ptr<int> p1(new int(0));
  std::auto_ptr<int> p2;    // p2 verwaltet nichts

  p2 = p1;     // p2 verwaltet jetzt das Objekt, p1 ist NULL
  (*p2)++;     // Wert um 1 erhoehen
  return p2;   // Eigentum an den Aufrufer weiterreichen
}

std::auto_ptr<int> g(std::auto_ptr<int> i)
```

```
{
  (*i)++;
  // argument must be returned
  return i;
}

void h(std::auto_ptr<int> &i)
{
  (*i)++;
}

int main(){
 std::auto_ptr<int> p = f();   // ok, kopiert Rueckgabewert
 std::cout << *p << '\n';;
 h(p);
 std::cout << *p << '\n';
 p = g(p);                     // ok
 std::cout << *p << '\n';
 g(p);                         // delete wird aufgerufen!
 std::cout << p.get() << '\n'; // NULL Zeiger
 }
```

Weitere Eigenschaften von Auto-Pointern sind:

1. Auto-Pointer sind für einzelne Objekte gedacht, es wird niemals `delete[]` aufgerufen,
2. Die "versehentliche" Umwandlung eines Auto-Pointer in einen regulären Zeiger wird durch die Eigenschaften von Auto-Pointern unmöglich gemacht,
3. `T* auto_ptr::get()` liefert die Adresse des Objekts, auf das der Auto-Pointer verweist (ein normaler Zeiger ohne Verantwortung),
4. `T* auto_ptr::release()` liefert die Adresse des bezogenen Objekts und hebt gleichzeitig den Besitz des Objekts durch den Auto-Pointer wieder auf. Das Objekt muss jetzt explizit zerstört werden.
5. `auto_ptr` verändert bei Kopie oder Zuweisung das Objekt der rechten Seite der Zuweisung. Dieses Verhalten macht die Verwendung mit Containern der STL unmöglich, denn dort wird einerseits an Einfügefunktionen wie `push_back` per konstanter Referenz übergeben, andererseits aber auch ein Kopierkonstruktor zur Einfügung in den Containerspeicherbereich aufgerufen. Daher schlägt die Instantiierung dieser Funktionen fehl. Darüber hinaus sind natürlich viele Algorithmen der STL von der *by value* Semantik der verwalteten Objekte abhängig, die `auto_ptr` gerade nicht implementiert.

6.4 Übungen

Ziel dieser Übung ist das Kennenlernen der Ausnahmebehandlung in C++, die auf den so genannten `exceptions` basiert.

6.4.1 Verwendung von Exceptions

Eigenes Programm:
Schreiben Sie ein Programm, das eine "sichere" arithmetische Operation ausführt. Das heisst, es sollte das Argument bzw. der Wertebereich des Arguments überprüft werden. Bei Überschreiten der zulässigen Werte sollte eine `exception` ausgeworfen werden. Diese sollte im Hauptprogramm abgefangen und mit einer Ausgabe kommentiert werden (nehmen sie z. B. das `safe_sqrt` oder die `add` Funktion). Geben Sie optional bei der Fehlerbehandlung mit aus für welche Operation mit welchen Operanden der Fehler auftrat (Tipp: Sie können dazu ihre `exception` Klasse mit Elementen versehen, die im Konstruktor gesetzt werden).

6.4.2 Exception des operator new

Schreiben Sie ein Programm, das den Anwender fragt, wieviel Speicher es anfordern soll. Wird zuviel Speicher angefordert, soll die Anforderung solange halbiert werden, bis die Speicheranforderung gelingt.

6.4.3 Exception oder spezieller Fehlerwert?

Erweitern Sie Ihre `Stack`-Klasse, damit sie eine `exception` wirft, wenn `pop` aufgerufen wird obwohl der stack leer ist. Da `pop` einen Wert zurückliefert, dessen Wertebereich vorher nicht bekannt ist, ist dies im Prinzip die einzige Möglichkeit einen Fehler zu signalisieren.

6.4.4 Exceptions bei der Ein- und Ausgabe

Schreiben Sie ein Programm, das Daten aus einer Datei einliest, deren Name vom Anwender angegeben wird. Die Fehlerbehandlung der Streams sollte dabei mittels Exceptions erfolgen. Falls ein Fehler beim Einlesen auftritt, sollte der Anwender die Möglichkeit erhalten, einen anderen Dateinamen anzugeben.

6.4.5 Ein intelligenter Zeiger

Wir haben gesehen, dass sich `auto_ptr` nicht mit Containern der STL verwenden lässt. Verifizieren Sie dies mit einem kurzen Programmbeispiel. Konstruieren Sie einen eigenen *smart pointer*, der dieser Beschränkung nicht unterliegt und der folgende Eigenschaften hat:
(i) Konstruktor, dem ein Zeiger auf eine Objektinstanz übergeben wird
(ii) Kopierkonstruktor und Zuweisungsoperator, die auf einen Referenzzähler zurückgreifen, der ihnen sagt, ob eine Objektinstanz die letzte verwaltete ist, ggf. inkrementiert oder dekrementiert und bei Überschreiben der letzten Objektinstanz ein nicht mehr bezogenes Objekt löscht.
(iii) Der Destruktor soll ebenfalls den Referenzzähler aktualisieren und für den letzten Zeiger auf ein Objekt ein `delete` aufrufen. (iv) Überladen Sie die Operatoren `*`, `->`, `bool`, um das Verhalten einfacher Zeiger zu simulieren.
(v) Haben Sie Ideen, was man machen könnte, um die versehentliche Konstruktion zweier solcher Zeiger zu verhindern, die auf das gleiche Objekt verweisen?
(v) Wie könnte man Zuweisungsoperator und Kopierkonstruktor implementieren, um auch Zuweisung oder Kopieren von kompatiblen Zeigertypen zu erlauben?

Machen Sie sich klar, welche Vorteile und Nachteile die Verwendung eines solchen Zeigers gegenüber expliziter Verwendung von `new` und `delete` hat.

Kapitel 7

Programmentwicklung

Programme, die das Schreiben von Programmen unterstützen, gibt es in großer Zahl. Am bekanntesten sind wohl die in der Windows-Welt verbreiteten grafischen integrierten Programmierumgebungen, die Compiler, Debugger und ggf. Versionsverwaltung in ein Paket integrieren. Wir werden im Folgenden nur einige einfache Pakete und Ideen erwähnen, die bestimmte Teilaspekte der Programmentwicklung ansprechen und die als freie Software für viele Systeme zur Verfügung stehen. Die größte Lücke in diesem Kapitel stellt vermutlich dar, dass wir nicht auf das Arbeiten mit Debuggern eingehen. Um logische Programmfehler oder auch Speicherzugriffsfehler zu finden, sind Debugger unentbehrlich. Weitere Werkzeuge helfen dabei, Speicherlecks zu finden, die hauptsächlich bei der Entwicklung von Klassen mit dynamischer Speicherverwaltung auftreten. Auch auf solche Werkzeuge können wir hier nicht eingehen, sondern können nur vorerst nur auf das WWW als Ressource verweisen.

7.1 Versionsverwaltung mit CVS

Werden wichtige Daten über längere Zeit oder von mehreren Personen bearbeitet, dann wird es häufig unerlässlich, die Zeitpunkte wichtiger Änderungen oder auch nur alte Versionen wieder rekonstruieren zu können. Nur so lassen sich kritische Änderungen identifizieren, die vielleicht in subtiler Weise zum Nichtfunktionieren eines Paketes geführt haben oder auch die Aufgabe lösen, ein großes Paket z.B. für zwei Plattformen oder in mehreren parallelen Versionen zu pflegen. Nur so ist man auch sicher, zu jedem Zeitpunkt zumindest eine lauffähige Version eines Programmes zu besitzen (nämlich dessen letzte "offizielle" Version).

Um nicht für jede kleine Änderung eine Kopie des gesamten Projekts anlegen zu müssen, wurden Programme entwickelt, die jeweils nur die gemachten Änderungen ablegen und die alle Versionen aus sukzessiver Kombination der abgelegten Änderungen mit einer Referenzversion rekonstruieren können.

Für Open Source Projekte hat sich weitgehend CVS (*concurrent version system*) durchgesetzt. CVS ist eine umfangreiche Erweiterung des Programmes `rcs`, das sich in der UNIX Welt für die Versionsverwaltung einzelner Dateien durchgesetzt hat, auf ganze Verzeichnisbäume. Während CVS in der UNIX Welt häufig im Kommandozeilenmodus verwendet wird, gibt es auch freie Versionen für Windows, die dieser Ebene eine grafische Benutzeroberfläche hinzugefügt haben. Viele UNIXe bringen CVS bereits mit und die Windows-Versionen lassen sich über Suchmaschinen leicht finden.

CVS speichert Referenzkopie und Änderungen in einem zentralen Verzeichnis, dem sogenannten *repository*. Die Konzeption von CVS ermöglicht es, mit mehreren Programmierern gemeinsam an einem Projekt zu arbeiten. Diese beziehen über CVS Kopien der im Zentralverzeichnis befindlichen Dateien, können diese dann lokal bearbeiten und schließlich wieder an CVS zurückgeben. CVS verschmilzt dabei Dateiversionen, solange die Änderungen ver-

schiedener Programmierer sich nicht überlappen. Ansonsten wird bei der Rückgabe der Datei an CVS ein Konflikt gemeldet, den der jeweils letzte Bearbeiter einer Datei per Hand auflösen muss.

7.1.1 Einrichtung eines CVS Zentralverzeichnisses

Als CVS Zentralverzeichnis kann ein beliebiges leeres Unterverzeichnis in der UNIX Dateisystemstruktur dienen. Dieses wird mit

```
cvs -d <repository> init
```

initialisiert. Dabei werden von CVS Verwaltungsdateien im Unterverzeichnis CVSROOT abgelegt. Dort kann man später spezielle Konfigurationsoptionen einstellen bzw. durch das Einrichten von "Modulen" die Kommunikation mit CVS erleichtern. Um bei zukünftigen Kommandos an CVS nicht wiederholt das Zentralverzeichnis angeben zu müssen, kann dieses in der Umgebungsvariable CVSROOT gespeichert werden. Diese wird sinnvollerweise in der Konfigurationsdatei der Shell (.bashrc, .cshrc, etc.) gesetzt.

Als abschließender Schritt wird noch ein Unterverzeichnis für Daten im Repository benötigt, das man am einfachsten mit dem Befehl `mkdir` erzeugt:

```
cd <repository>
mkdir <Projektverzeichnis>
```

Jetzt kann jeder Programmierer mit entsprechenden Zugriffsrechten im Repository dieses zunächst leere Verzeichnis "auschecken". Durch Ausführen von

```
cvs -d <repository> checkout <Projektverzeichnis>
```

wird eine lokale Kopie dieses (leeren) Verzeichnisses zusammen mit Verwaltungsinformation (im Unterverzeichnis CVS) erzeugt. Diese kann dann in vielfältiger Weise genutzt werden.

7.1.2 Benutzung

In dem ausgecheckten, anfänglich leeren Arbeitsverzeichnis können in beliebiger Weise (Kopie, Editor) Dateien oder neue Unterverzeichnisse erzeugt werden. Nach dem Erstellen der anfänglichen Version einer Datei oder eines neuen Unterverzeichnisses wird deren Existenz und die Absicht, diese im Repositry zu hinterlegen, dem CVS mit

```
cvs add <Verzeichnis oder Dateiname>
```

mitgeteilt. Bei einem nachfolgenden

```
cvs commit <Dateiname>
```

übernimmt CVS die Datei oder spätere Änderungen in das Repository. Der Dateiname kann auch eine Liste von Dateinamen sein oder sogar leer verbleiben, womit er sich dann auf alle Dateien im aktuellen Verzeichnis und in Unterverzeichnissen bezieht.

7.1.3 Beschreibung der Kommandos

Die CVS-Kommandos und ihre Funktion sind im einzelnen:

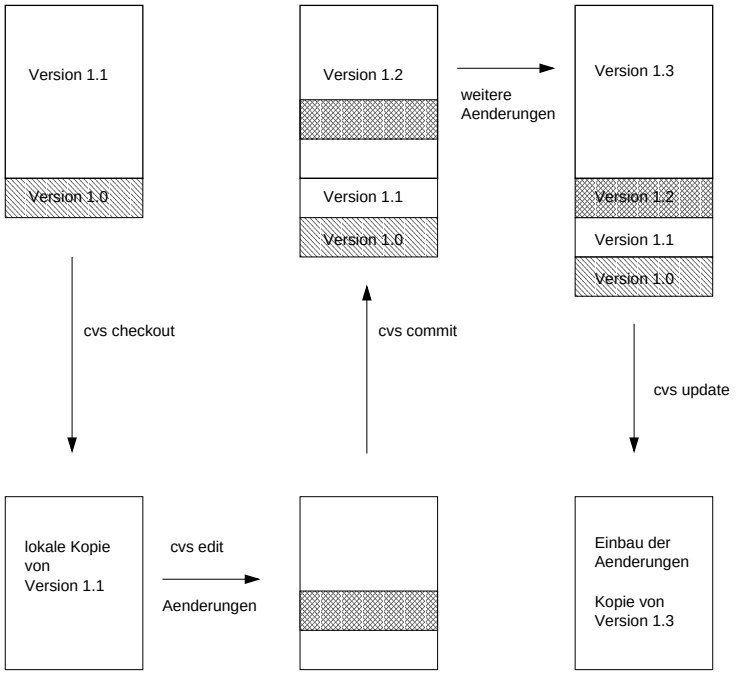


Abbildung 7.1: Grundkonzepte von CVS: Durch das "Auschecken" einer Datei wird eine lokale Arbeitskopie erstellt. Diese wird durch "cvs edit" schreibbar gemacht. Nach Durchführung von Änderungen werden diese dem CVS durch "cvs commit" mitgeteilt. Mittels "cvs update" kann man jederzeit die lokale Arbeitskopie mit dem Repository abgleichen, falls dort inzwischen andere Programmierer ihrerseits Änderungen eingetragen haben.

Übersicht über die `cvs` Kommandos

add	Add a new file/directory to the repository
admin	Administration front end for rcs
annotate	Show last revision where each line was modified
checkout	Checkout sources for editing
commit	Check files into the repository
diff	Show differences between revisions
edit	Get ready to edit a watched file
editors	See who is editing a watched file
export	Export sources from CVS, similar to checkout
history	Show repository access history
import	Import sources into CVS, using vendor branches
init	Create a CVS repository if it doesn't exist
log	Print out history information for files
login	Prompt for password for authenticating server.
logout	Removes entry in .cvspass for remote repository.
rdiff	Create 'patch' format diffs between releases
release	Indicate that a Module is no longer in use
remove	Remove an entry from the repository
rtag	Add a symbolic tag to a module
status	Display status information on checked out files
tag	Add a symbolic tag to checked out version of files
unedit	Undo an edit command
update	Bring work tree in sync with repository
watch	Set watches
watchers	See who is watching a file

Hilfe

Neben der obligatorischen Manualseite `man cvs` enthält CVS eine Online-Hilfestellung, die Kommandos auflistet `cvs --help-commands`, globale `cvs --help-options` oder kommandospezifische Optionen `cvs --help <command>` auflistet, bzw. diese oben aufgelistete Information ausgibt `cvs --help`.

Dateien/Verzeichnisse hinzufügen oder entfernen

Eine neue Datei oder ein neues Unterverzeichnis wird mit

```
cvs add main2.cc
```

bei CVS zur Verwaltung angemeldet. Es muss danach noch eingecheckt werden:

```
cvs commit -m 'Kommentar zum neuen file' main2.cc
```

Gibt man die `-m` Option nicht an, so startet CVS einen Editor, in dem zur Eingabe des Kommentars aufgefordert wird.

Um eine Arbeitskopie zu löschen, benuzt man

```
cvs release main1.cc        (Loeschen der Arbeitskopie)
```

Dieses Kommando entfernt nur die lokale Kopie, aber nicht die Repository-Version der Datei. Um den Status im Repository zu ändern, muss man mit `cvs remove` ankündigen, dass in zukünftigen Versionen diese Datei nicht mehr benötigt wird. Der Geschichtsinformation der Datei wird nun durch remove eine Version zugeordnet, die als nicht existent gilt. Frühere Versionen der Datei bleiben erhalten und können weiterhin ausgecheckt werden.

```
cvs remove main1.cc          (Anmeldung des Loeschens)
cvs commit main1.cc          (Vollzug des Loeschens)
```

Erstellen einer Arbeitskopie - Checkout bestehender Dateien/Verzeichnisse

Das folgende Kommando legt, falls nicht vorhanden, im aktuellen Verzeichnis das Unterverzeichnis `<Projektverzeichnis>` an, in dem sich Dateien und alle Verzeichnisstruktur befindet, die von CVS im entsprechenden Unterverzeichnis des Repository verwaltet werden. Die Angabe des Repository ist nur beim ersten Checkout nötig, danach entnimmt CVS diese Information den Verwaltungsdateien im ausgecheckten Verzeichnis. Durch Setzen der Umgebungsvariable CVSROOT kann die Angabe des Repository generell eingespart werden. Checkout Kommandos in bereits ausgecheckten Verzeichnissen bewirken, dass die dort vorhandenen Dateien an das Repository angeglichen werden bzw. lokal fehlende Dateien aus dem Repository entnommen werden.

```
cvs -d <cvsroot> checkout project_1
```

Alle Dateien/Unterverzeichnisse liegen nun als lokale, schreibgeschützte Kopien vor. Wenn nichts anderes angegeben wird, beziehen sich cvs-Kommandos immer auf das aktuelle Verzeichnis und bearbeiten rekursiv auch alle Unterverzeichnisse.

Mit der Option `-r` lassen sich beliebige ältere Versionen auschecken:

```
cvs checkout -r 1.3 main1.cc  (Version 1.3 von main1.cc)
```

In ähnlicher Weise können statt Versionsnummern auch Zeitangaben (siehe Abschnitt Änderungsliste/Log) oder symbolische Namen für Versionen stehen (dazu siehe die CVS Dokumentation).

Editieren einer ausgecheckten Datei

Das Defaultverhalten von cvs-1.12 ist, dass frisch ausgecheckte Dateien schreibbar sind. Sie können unmittelbar editiert und danach die Änderungen mit `commit` wieder in das Repository übergeben werden.

Arbeitet man mit vielen Programmierern an einem Projekt, so kann es sinnvoll sein, CVS mitzuteilen, dass man über Änderungen an einer Datei informiert werden möchte. Diesen Wunsch teilt man CVS mit dem `watch` Kommando mit, in dem man auch die überwachte Aktion (etwa das commit oder bereits das Editieren einer Datei) angibt. Dazu begibt man sich in ein ausgechecktes Verzeichnis und führt dort

```
cvs watch add -a commit
```

aus. Man kann dem CVS global mitteilen, dass ausgecheckte Dateien nur lesbar sein sollen, indem man die Umgebungsvariable CVSREAD setzt. Das erinnert Programmierer daran, auch das Editieren einer Datei dem CVS bekanntzumachen:

```
cvs edit main1.cc
```

Diese Datei kann nun bearbeitet werden, denn der Schreibschutz wurde mit diesem Kommando aufgehoben. Gleichzeitig versendet CVS e-mail an Programmierer, die das Editieren dieser Datei durch CVS überwachen lassen. Überlegt man es sich später anders, teilt man CVS mit, dass man die Datei nicht mehr editieren will:

```
cvs unedit main1.cc
```

Information darüber, welcher Programmierer gerade die Datei `main1.cc` mit der Absicht, sie zu verändern, ausgecheckt hat, liefert

```
cvs editors main1.cc
```

Benötigt man die ausgecheckte Datei nicht mehr, so kann sie CVS-verträglich gelöscht werden:

```
cvs release main1.cc
```

Aktualisieren ausgecheckter Dateien

Um die ausgecheckten Dateien oder Verzeichnisse zu aktualisieren, was nötig ist, wenn sich die Versionen im Repository durch das `cvs commit` anderer Programmierer zwischenzeitlich geändert haben, benutzt man das Kommando `cvs update`.

```
cvs update            (alle Dateien des aktuellen Verzeichnisses)
cvs update main1.cc   (nur main1.cc)
cvs update -d         (checkt auch neue Unterverzeichnisse aus)
```

Ist die Version im Repository aktueller als die lokale Version, so integriert CVS die Änderungen in die lokalen Kopien. Treten dabei Konflikte auf, weil die lokalen Kopien selbst an Stellen verändert wurden, die mit den Änderungen im Repository überlappen, so warnt CVS, und der Programmierer muss die entstandenen Konflikte "per Hand" (und Editor) auflösen.

Eine neue Version einchecken

Um die editierte Datei als neue Version durch CVS verwalten zu lassen, wird sie mit `cvs commit` an CVS übergeben und somit für andere Programmierer als neue Version sichtbar:

```
cvs commit -m 'Kommentar zu den Aenderungen' main1.cc
```

Vor jedem commit erfordert CVS ein update, damit Änderungen aus dem Repositry in die Datei integriert werden können. Gibt man die `-m` Option nicht an, so startet CVS einen Editor, der die Eingabe eines Kommentartextes ermöglicht.

Änderungsliste/Log

Informationen zu bisherigen Änderungen einer Datei `xyfile.cc` findet man in den Kommentaren, die die Bearbeiter beim Einchecken hinterlassen. Diese ruft man mit

```
cvs log xyfile.cc
```

ab. Es ist auch in einfacher Weise möglich, sich die Änderungen in der Datei selber anzeigen zu lassen. In Anlehnung an das UNIX Kommando `diff` wird von

```
cvs diff xyfile.cc
```

die Differenz der Arbeitskopie zu der im Repository angezeigt. Die Änderungen von einer Version zu einer anderen findet man mit

```
cvs diff -c -r 1.3 -r 1.8 xyfile.cc
```

Die `-c` Option erhöht die Deutlichkeit der Ausgabe. Man kann sich auch die Änderung bezüglich eines Zeitpunktes ausgeben lassen:

```
cvs diff -D "one month ago" xyfile.cc
```

Das Datum kann dabei auf verschiedene Weisen angegeben werden. Legale Möglichkeiten sind:

```
3/31/92
3/31/92 10:00:07 PST
January 23, 1987 10:05pm
1972-09-24 20:05
2 hours ago
400000 seconds ago
last year
last Monday
yesterday
22:00 GMT
a fortnight ago
```

Verzweigungen im Versionsbaum erzeugen

Mit `cvs` sind wie mit `rcs` nicht nur lineare, sondern auch verzweigte Versionsbäume möglich:

Solche Verzweigungen können in großen Projekten dazu dienen, in Entwicklung befindliche Teile, in denen noch "Experimente" gemacht werden, von Versionen zu trennen, die bereits getestet sind und freigegeben wurden. Eine weitere Motivation besteht darin, wichtige Fehler in alten Versionen noch beheben zu können, ohne den Hauptzweig antasten zu müssen. Eine solche Verzweigung wird mit einem symbolischen Namen verknüpft kann dann als Grundlage einer Arbeitskopie dienen. Im Beispiel wird der Name `quickfix` für alle Dateien im Verzeichnis `project` vereinbart:

```
cvs rtag -r 1.3 quickfix projekt
```

Wenn man mit

```
cvs checkout -r quickfix projekt
```

eine frische Kopie auscheckt, dann beziehen sich `commit` Anweisungen in diesem Verzeichnis immer nur auf den neuen Zweig.

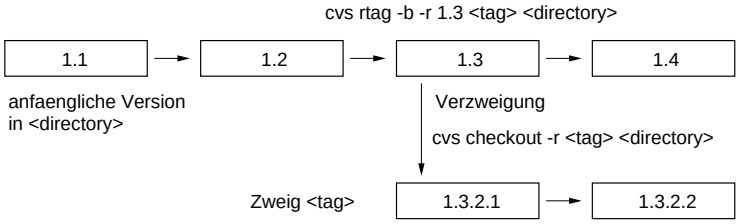


Abbildung 7.2: Die Erzeugung neuer Zweige im Versionsbaum ist möglich mit einem der `tag` Kommandos des CVS. Im Beispiel wird an der existierenden Version 1.3 ein Zweig mit Namen `<tag>` erzeugt. Indem dann mit diesem Namen ausgecheckt wird, erzeugt man eine Arbeitskopie der Version 1.3, die intern mit einer neuen Versionsnummer (1.3.2.1) belegt wird. Änderungen, die an dieser Datei gemacht und ins Repository eingetragen werden, werden nicht am Hauptzweig durchgeführt, sondern in dem neu erzeugten Nebenzweig. Diese Eigenschaft von CVS ermöglicht das Beheben von Fehlern oder das gemeinsame Arbeiten an experimentellen Versionen, die keinen Eingang finden sollen in den Hauptzweig des Programmes.

7.2 Coding Standards

Coding Standards sind Regeln, die vor allem von Firmen, Universitäten oder andere Organisationen aufgestellt werden, um die Zusammenarbeit zwischen vielen Programmierern handhabbar zu machen. Sie spiegeln häufig spezielle Erfahrungen oder Ziele dieser Gruppen wider und beabsichtigen unter anderem:

1. die Lesbarkeit des Programmcodes durch sinnvolle Namensgebungen und einheitliche Strukturierung zu erhöhen,
2. die Portabilität auf andere Rechner oder Plattformen als die Entwicklungsplattform sicherzustellen, sowie
3. die Erweiterbarkeit und Wiederbenutzbarkeit der Programme,
4. die Fehlerfreiheit und
5. die Leistungsfähigkeit — insbesondere bei numerischen Anwendungen.

So sollen Programme auch für Mitarbeiter am gleichen Projekt "bekannt" aussehen — häufig wird dann auch als Bestandteil der Qualitätssicherung ein Code-Review durchgeführt, bei dem Mitarbeiter gegenseitig ihre Programme auf Probleme hin untersuchen.

Beispiele für solche Coding Standards findet man im Internet. Es ist nicht ungewöhnlich, auch Widersprüche zwischen verschiedenen Versionen zu finden, die sich aus der Setzung unterschiedlicher Schwerpunkte erklären. Ein Beipiel sind die GNU Coding Standards der Free Software Foundation, die Linux Coding Standards, die sich zum Beispiel in der Dokumentation befinden, die mit dem Linux-Quelltext verteilt wird. Man sehe sich auch z.B. die Deklarationsdateien der C++-Bibliothek an, die im Quelltext mit Compilern verfügbar sind. Auch in der C++-FAQ

`http://www.informatik.uni-konstanz.de/~kuehl/cpp/cppfaq.htm`

befindet sich ein Abschnitt zu Coding Standards.

7.2.1 Dateinamen

Aufgrund der Eigenschaften der Compiler empfiehlt es sich Deklarationsdateien mit `.h` und Definitionsdateien mit Suffix `.cc` (UNIX) oder `.cpp` (Windows) anzulegen.

7.2.2 Variablennamen

Oft wird der Typ der Variablen in einem Suffix (z.B. `_ptr` für Pointer) angezeigt.

Eine andere Möglichkeit ist die Verwendung von Präfixes wie z.B. in der von Microsoft verwendeten ungarischen Notation etwa mit `c` für `char class`, `i` für `integer`, `s` für `short` oder `static`, `p` für Zeigervariablen. Damit könnte etwa `ps_maxwindows` einen Zeiger auf ein Variable vom Typ `short` kennzeichnen, deren Bedeutung im Namen kodiert ist. Bei weitgehend lokaler Definition von Variablen, die von C++ sehr durch die beliebige Stellung von Variablendeklarationen im Programm unterstützt wird, ist die Verwendung Ungarischer Notation weitgehend entbehrlich (und wird in den GNU Coding Standards abgelehnt).

Alternativ oder zusätzlich gibt es Konventionen für Klassennamen (Wortbeginn mit Großbuchstaben), lokale Variablen (Wortbeginn mit Kleinbuchstaben); für globale oder statische Variablen findet man häufig eine Großschreibung des ersten Buchstabens wie bei Klassennamen. Die besondere Rolle von Konstanten, Makros und Aufzählungsvariablen wird durch durchgängige Großschreibung angezeigt.

Um Elementvariablen von lokalen Variablen zu unterscheiden, bietet sich ebenfalls eine spezielle Kennzeichnung an. Häufig findet man den Präfix `m`, `my` oder `_`. Für statische Klassenvariablen findet man `our`, um die Zugehörigkeit zu allen Klasseninstanzen zu markieren.

Wichtiger als solche formalen Präfix-Regeln ist allerdings die Wahl aussagekräftiger und prägnanter Namen für Variablen und Prozeduren.[1]

7.2.3 Variablendeklaration

Pro Zeile sollte nur eine Deklaration stattfinden. Erstens eignet sich diese Form besser zur Dokumentation, zweitens vermeidet man Fehler wie

```
int* a, b, c; // a ist pointer auf int, b und c sind int!
```

Variablen sollten möglichst lokal deklariert werden und an der Stelle ihrer Initialisierung, insbesondere sollte man von der Möglichkeit Gebrauch machen, Schleifenvariablen in `for`-Schleifen in der `for`-Anweisung selbst zu deklarieren.

7.2.4 Performance, const, Zeiger oder Referenz

Verwende `const` wenn immer möglich, insbesondere bei der Übergabe von Referenzen an Unterfunktionen. Beispiel:

```
const int N=100;
```

hier hat der Compiler die Möglichkeit, bestimmte Berechnungen schon zur Compilezeit auszuführen. Das führt dazu, dass die Verwendung dieser Variablen an den meisten Stellen vollständig wegoptimiert werden kann.

Bevorzuge bei "kleinen" Klassen die Übergabe *by value* anstatt der Übergabe von Referenzen:

```
class Small{
  // keine dynamische Speicherwerwaltung
  // 3-4 normale Datenelemente als Member
 ...
};

class Big{
```

[1] Einer der Autoren ist der Meinung, dass in einer Zeit des globalen Austausches von Information die Wahl englischer Variablennamen guter Stil ist, denn früher oder später wird das Programm mit einem Kollegen oder Kommilitonen ausgetauscht, der kein Deutsch spricht...

```
 ...
};

void f(Small s, const Big & b);
```

Dieses Vorgehen ermöglicht dem Compiler, das Nichtvorhandensein von Aliasing zu erkennen. Bei der Übergabe zweier Variablen mittels Referenz oder mittels Zeigern kann es dazu kommen, dass mit verschiedenen Namen derselbe Speicherplatz angesprochen wird. Dann darf der Compiler keine Optimierungen ausführen, die sich darauf verlassen, dass die angesprochenen Speicherbereiche nicht überlappen. Im unteren Beispiel dürfen also die Anweisungen in der Schleife nur in der angegebenen Reihenfolge durchlaufen werden.

```
f(int *a, int *b){
  for(int i=0;i<100; i++){
    a[i]=b[i+5]*3;
  }
}
```

Ist für den Compiler erkennbar, dass kein Aliasing auftritt, dann kann auf bestimmten Architekturen die Schleife "parallelisiert" werden. In einigen numerisch orientierten Sprachen wie FORTRAN verlangt man daher *vom Programmierer*, bei Aufruf einer Routine mit verschiedenen Feldparametern sicherzustellen, dass kein solches Aliasing auftritt. In der Zukunft von C++ ist dafür das Schlüsselwort `restrict` bei Übergabe eines Feldtyps vorgesehen, das dem Compiler anzunehmen erlaubt, dass kein aliasing stattfindet.

Die Übergabe per `const &` wird bei größeren Klassen effektiver, da keine Kopie angelegt wird. Damit dies funktioniert, müssen Elementfunktionen konsequent `const` deklariert und gegebenenfalls zwei Versionen (eine `const` "zum Lesen," die andere nicht `const` auch zum Schreiben) bereitgestellt werden.

7.2.5 #include Direktiven

Systemheader sollten mit `#include < >`, eigene Header mit `#include " "` eingefügt werden, da im ersten Fall zunächst Systemverzeichnisse, im zweiten zunächst Benutzerverzeichnisse durchsucht werden. Relative Pfadangaben in `include` Direktiven (`#include "../myheader.h"`) sollte man im Hinblick auf mögliche Umstrukturierungen der Verzeichnisstruktur vermeiden. Compiler bieten dafür Erweiterungen der Suchpfade an, meist durch Verwendung der `-I` Option: `g++ -I../ myfile.cc`

Jede Headerdatei muss eine `#ifndef/#define/#endif`-Kombination enthalten, um zu verhindern, dass Deklarationen pro Übersetzungseinheit mehr als einmal sichtbar sind:

```
header.h

#ifndef HEADER_H   /* eindeutiger Name fuer header.h */
#define HEADER_H
// ...
#endif
```

7.2.6 Präprozessoranweisungen

Das Ausblenden von langen Codeblöcken mit

```
#if 0
  ...
#endif
```

umgeht mögliche Probleme in der Kombination von C++-Kommentaren und Präprozessoranweisungen sowie das der geschachtelten `/* */` Kommentare in C-Code. Aus dem gleichen Grund sollte, wo möglich, die C++-Kommentarsyntax `//` bevorzugt werden.

Die in C übliche Definition von Konstanten durch Präprozessoranweisungen sollte in C++ zugunsten von `const`, möglicherweise `static` oder in unbenannten Namensräumen verwendeten Variablen aufgegeben werden.

Die in C übliche Verwendung von Makros zur Emulation von inline-Funktionen sollte in C zugunsten von wirklichen inline-Funktionen bzw. inline Template-Funktionen aufgegeben werden.

7.2.7 Kommentare

- Verwende `//` wo möglich statt `/* */` (s.o.)
- Am Anfang jeder Datei ein ausführlicher und sinnvoller Kommentar, was hier gemacht wird
- Vor jeder Funktion eine Beschreibung, was die Funktion macht
- Kommentare sollen "allgemein" formuliert sein, sie sollen keine Wiederholung des Algorithmus in Worten darstellen.
- Kommentare in der gleichen Sprache wie Variablennamen
- Kommentare gehören an jede Zeile, die einen "genialen" Programmiereinfall beinhaltet. Für andere Leute oder zu einer späteren Zeit ist dieser Einfall meist nur schlicht unverständlich.

Kommentare werden von einigen Programmierhilfen (javadoc, kdoc, doxygen) dazu verwendet, automatisch Dokumentation zu erzeugen. Dazu ist jeweils eine spezielle Syntax der Kommentare zu verwenden. Insbesondere bei Klassenbibliotheken, die von mehreren Programmierern genutzt werden, ist der Nutzen dieser Dokumentation nicht zu unterschätzen — und sei es nur wegen der übersichtlichen Auflistung der Methoden in den Klassen.

7.2.8 Formatierung

- Um gute Lesbarkeit des Programmtextes mit gängigen Texteditoren zu gewährleisten, sollte man nicht mehr als 80 Zeichen pro Zeile verwenden und dann einen Zeilenumbruch einfügen.
- Leerzeichen vor und nach binären Operatoren erhöhen die Lesbarkeit.
- Bei Zeilenumbrüchen in Ausdrücken, die Operatoren enthalten, sollte die Fortsetzungszeile durch ein Operatorsymbol eingeleitet werden.
- Geschweifte Klammern sollten konsistent gesetzt werden. Hier gibt es viele verschiedene Konventionen. Klammern können etwa paarweise untereinander stehen oder (kompakter) die öffnende Klammer steht nach dem einleitenden Ausdruck (`if(...) else`, `for(...)`, etc.) und die schließende unter dem erstem Buchstaben des einleitenden Ausdrucks. Man sollte sich hier für einen Stil entscheiden und diesen dann durchhalten.
- Bei längeren Blöcken die schließende Klammer kommentieren um den Bezug zur öffnenden herzustellen.
- Funktionen kurz halten und wenn nötig durch die Einführung weiterer Funktionen kürzen. Idealerweise passt der Inhalt einer Funktion immer auf einen Bildschirm.

- Deklaration des Destruktors direkt nach den Konstruktoren, um die Konsistenz beider schnell beurteilen zu können.

7.2.9 Editorunterstützung

Einige Editoren unterstützen automatisch das Einrücken und Klammern gewisser syntaktischer Einheiten. Im *emacs* lässt sich etwa mit

```
M-x c-set-style <ENTER> <Stil> <ENTER>
```

einer der Stile `gnu, linux, k&r, stroustrup` einstellen.

7.3 Objektorientiertes Design

7.3.1 Beziehung zwischen Objekten

Am Anfang des Designs einer Klasse, oder von Softwaredesign überhaupt, ist es wichtig, sich darüber ganz klar zu sein, was das geplante Programm oder die zu implementierende Klasse leisten soll. Die exakte Implementierung sollte am Beginn der Designphase keine Rolle spielen und sich nur in Ausnahmefällen im Benutzerinterface der Klasse widerspiegeln. Wichtiger ist es, die Benutzung intuitiv und einfach zu gestalten, ein Klasseninterface sollte minimal, aber vollständig sein.

Im Zweifel sollte man immer versuchen, lieber viele kleine Komponenten zu entwerfen anstelle von wenigen, aber umfangreichen oder großen, die gewöhnlich nur für einen entsprechend speziellen Zweck dienlich sind. Kleine Komponenten sind besser und leichter überschaubar und können leichter und flexibler zu neuen Strukturen zusammengesteckt werden als das mit größeren der Fall ist. Jede Komponente sollte einem klaren Konzept entsprechen, das sich im Interface widerspiegelt. So sind etwa komplexe Zahlen arithmetische Objekte und erhalten dementsprechend auch die Eigenschaften der eingebauten arithmetischen Typen. Ein Timer könnte die von einer Stoppuhr gewohnte Schnittstelle aufweisen: Elementfunktionen `start`, `stop`, `reset`, `read`, die sich wie die Knöpfe einer mechanischen Uhr verhalten und die Uhr entweder (wiederholt) starten, anhalten oder zurücksetzen und mit deren Hilfe sich der aktuelle Stand abfragen läßt.

Die objektorientierte Programmierung verspricht bessere Wartbarkeit und bessere Wiederverwendbarkeit von Programmkomponenten. Dies wird einerseits erreicht durch das Konzept der Vererbung; andererseits sollte man aber auch wo möglich auf die bereits in der Standardbibliothek implemetierten Klassen zurückgreifen und nicht versuchen, "das Rad neu zu erfinden".

Die von der Standardbibliothek vorgegebenen Kategorien der Trennung von Datenstrukturen und Algorithmen können häufig Anregungen geben, wie Probleme aufgespalten werden können.

Um ein Design zu testen, bietet es sich an, zuerst eine rudimentäre Implementierung zu erstellen. Z.B. könnten Funktionsaufrufe zunächst einfach leere Funktionskörper aufweisen, nur default-Werte zurückgeben oder durch warnende Ausgaben dokumentiert sein. Ziel sollte immer sein, die Klassen einfach und intuitiv bedienbar zu machen, nicht jedoch, die Implementierung zu vereinfachen.

Als Faustregel bietet sich zur Übersetzung eines Problems in eine objektoriente Programmstruktur an, zunächst eine verbale Analyse durchzuführen. Als Anhaltspunkt kann gelten, dass Substantive als Klassen und Verben als Elementfunktionen oder Funktionen realisiert werden können.

Die wichtigsten Beziehungen zwischen Klassen sind

is a : Vererbung drückt aus, dass es sich bei der abgeleiteten Klasse um eine spezielle Version der Basisklasse mit ggf. zusätzlichen oder jedenfalls geänderten Eigenschaften handelt. Die Basisklasse stellt in der Regel die Programmierschnittstelle bereits in vereinfachter Form (möglicherweise als rein virtuelle Funktionsprototypen) zur Verfügung. Umgangssprachlich drückt sich diese Beziehung häufig durch *is a* aus.

has a : ist der Test, ob eine Eigenschaft im Gegensatz zur Ableitung lieber durch Einschluss als Datenelement erworben werden soll. Ein Auto *hat einen* Motor: in einer objektorientierten Umgebung bedeutet das die Repräsentation des Motors durch ein Datenelement der Klasse Auto. Ein Lastwagen *ist ein* spezielles Auto—hier ist Ableitung erfolgversprechend.

is implemented in terms of : kann bedeuten, dass `private` abgeleitet werden sollte. Das bedeutet, dass die Klasse zwar die Funktionen der Basisklasse zur Realisierung ihrer eigenen Funktionalität verwenden kann, aber diese Tatsache nicht für den Benutzer sichtbar ist. So kann z.B. eine Stack-Klasse mittels `std::vector` und dessen Elementfunktionen implementiert werden.

7.3.2 Klassenschnittstellen

Klassenschnittstellen sollten gleichzeitig minimal und vollständig sein. Vollständigkeit bedeutet, dass ich per Aufruf einer Elementfunktion alle Operationen durchführen kann, die sich aus der gedachten Rolle der Klasse möglicherweise ergeben. Damit die Notwendigkeit und Versuchung eines externen Zugriffs auf Datenelemente der Klasse minimiert. Die Minimalität der Klassenschnittstelle bedeutet, dass für die Erledigung dieser Aufgaben nicht mehr Elementfunktionen als unbedingt nötig zur Anwendung kommen sollen. Funktionsnamen wie `clear()` und `reset()`, die beide in der Schnittstelle auftauchen, können bewirken, dass sich der Anwender beginnt, darüber Gedanken zu machen, ob deren Wirkungen wirklich die gleichen sind. Weiterhin sind große Schnittstellen unübersichtlich und fordern mehr Zeit zum Verständnis.

Wie bei dem bereits oben erwähnten Beispiel der Stoppuhr macht es Sinn, sich bei der Definition der Klassenschnittstelle an realen Gegenständen zu orientieren, deren Benutzung intuitiv erfassbar ist.

Auf jeden Fall ist es erstrebenswert, die Schnittstelle weitgehend nach Präzedenzen zu gestalten: Die Kompatibilität mit C-Strukturen, Schnittstellennamen wie in der STL, das Überladen von geeigneten Operatoren für "arithmetische" Typen wie Matrizen, Vektoren, komplexe Zahlen, etc. können dabei Leitlinien sein.

7.3.3 Allgemein oder speziell?

Häufig findet man sich bei der Implementierung einer Funktion oder Klasse vor der Frage, wie "allgemein" die Implementierung sein sollte. Soll ich etwa ein Histogramm berechnen, das (i) Werte aus einem vorgefertigten `std::vector<>` einliest, sollten (ii) die Werte aus einem C-Feld `a[]` kommen oder sollte man STL-ähnliche `push_back` oder `insert` Funktionen oder gar Eingabeiteratoren verwenden, um die Information von außen elementweise an die Klasse zu übergeben? Die Antwort darauf kann leider nicht allgemein gegeben werden, sondern hängt immer vom Anwendungsfall ab. Die Übergabe von Feldern oder Vektoren kann eine notwendige Optimierung gegenüber einer Lösung mit Eingabeiterator oder einer`push_back` oder `insert`-Funktion sein. Erfahrungsgemäß vorzuziehen ist aber der Ansatz, die allgemeinere Funktion zu verwenden und sich auf den Compiler für Optimierungen zu verlassen. Dies gilt insbesondere, wenn das verwendete Konzept bereits in der STL Ausdruck gefunden hat.

Für Felder als Argumente ist häufig ein Template eine gute Wahl, das nur die Eigenschaft des Feldes im folgenden Programmcode voraussetzt, mit `operator[]` bearbeitbar zu sein.

Kapitel 8

Systemspezifische Fragen und Bibliotheken

Moderne Betriebssysteme stellen Funktionalität bereit (etwa die grafische oder akustische Ein- und Ausgabe), die im Sprachstandard nicht enthalten sind. Wie diese Funktionen anzusprechen sind, hängt vom Betriebssystem ab. Auf vielen Systemen stehen Bibliotheken zur Verfügung, die in der Sprache C geschrieben sind und daher auch aus C++ angesprochen werden können.

Im Folgenden sollen einige Konzepte und Bibliotheken sowie deren Integration in ein C++-Programm vorgestellt werden.

8.1 Vorüberlegungen

Die Verwendung von Bibliotheken stellt immer ein Risiko für die "Lebenszeit" von Programmcode dar. Insbesondere sollte beachtet werden

1. dass Bibliotheken weiterentwickelt werden und damit ggf. eine Anpassung des Programmes nötig wird, um es mit einer neuen Bibiotheksversion lauffähig zu halten.

2. dass die Anbindung an eine Bibliothek häufig zur Programmlaufzeit erfolgt (dll's in der Microsoft-Welt oder shared libraries in UNIX-Derivaten). Dies bedeutet, dass man sein Programm gar nicht mit der endgültig verwendeten Bibiotheksversion testen kann. Die Konsequenzen sind aus der Microsoft-Welt hinlänglich bekannt: Installationsprogramme bringen ggf. ihre dynamischen Bibliotheken mit und ersetzen deren Systemversionen, mit unvorhersehbaren Effekten für die Systemstabilität oder auf andere Programme. Abhilfe schafft nur eine konsequente Versionsverwaltung dynamisch gelinkter Bibliotheken (Beispiel: `ldconfig` in Linux) oder statisches Anbinden der Bibliothek mit der Konsequenz recht großer Programme.

3. dass Bibliotheken häufig nicht zwischen verschiedenen Rechnerarchitekturen portabel sind. So bietet etwa die Microsoft Visual C++ Umgebung eine umfangreiche Klassenbibliothek zur Anbindung an das Betriebssystem und zur Erstellung einer grafischen Benutzeroberfläche an. Unglücklicherweise ist diese Bibliothek nur unter Windows-Betriebssystemen verfügbar. Portable grafische Oberflächen lassen sich mit wxWindows, gtk++ oder (halb-)kommerziellen Lösungen wie Qt programmieren.

4. dass bei der Benutzung kommerzieller Bibliotheken Lizenzierungfragen entstehen.

8.2 Threads

8.2.1 Grundprinzip

Thread heißt wörtlich Faden. Ein Thread ist eine Sequenz von Instruktionen, die von der CPU ausgeführt werden. Dabei kann es sich, wie bei einfachen Programmen üblich, um eine einzige Sequenz handeln, die der Abarbeitung des Programmes entspricht, oder um mehrere Sequenzen, die parallel auf mehreren CPUs oder in verschiedenen Zeitscheiben auf einer CPU auf einem gemeinsamen Datenbestand arbeiten. Natürlich muss das Betriebssystem (oder eine geeignete Bibliothek) in der Lage sein, die Kontrolle entsprechend den Bedürfnissen des Programms auf eine oder mehrere CPUs zu verteilen, und es muss automatisch auch Threadwechsel auf einer CPU vornehmen können.

Alle gleichzeitig ablaufenden Threads teilen sich den gleichen Adressraum ihres "Mutter"programmes, so dass Datenaustausch zwischen Threads auf das Lesen und Schreiben von gemeinsamen Speicherbereichen (Feldern und Variablen) reduziert werden kann. Es sind keine Softwareprotokolle (wie etwa das Message Passing Interface (MPI)) oder Kopiervorgänge wie bei Interprozesskommunikation mit Pipes erforderlich. Wir werden jedoch sehen, dass dabei einige wichtige Fragen der Synchronisierung geklärt werden müssen, um voraussagbares Programmverhalten zu erzeugen.

Als Beispiel betrachten wir ein Programm, das auf eine Benutzeraktion (etwa einen Mausklick) schnell reagieren muss, obwohl gleichzeitig umfangreiche Berechnungen durchzuführen sind. Dieses Problem lässt sich mit einem Design lösen, in dem man die Benutzeranfragen in einem Thread bearbeitet und die Berechnungen in einem zweiten.

```
---------------------------------------------- > t

Programm mit einem Thread:

--- main {                                } ------

Programm mit zwei Threads:

                      Thread 1
--- main { --- create --------- join --- }  ------
                      Thread 2
                      ---------
```

Wie lassen sich nun Threads erzeugen und verwalten? Wie erwähnt, ist dazu entweder die Mithilfe des Betriebssystems erforderlich (auf dessen Funktionalität wir in der Regel wieder über eine geeignete Systembibliothek zugreifen) oder die Nutzung einer geeigneten Bibliothek. Wir werden hier nur das Beispiel der standardisierten POSIX Threadbibliothek erläutern. POSIX-Threads sind für viele Betriebssysteme verfügbar, inklusive der Windows-NT Nachfolger und die überwiegende Mehrheit der uns bekannten UNIX-Systeme. Diese Bibliothek definiert einen Datentyp `pthread_t`, der einen Thread eindeutig identifiziert. Unter Linux handelt es sich dabei um einen einfachen Integer-Datentyp. Die wichtigsten Funktionen zur Erzeugung und Beendigung von Threads sind

```
// pthread.h:

int  pthread_create(pthread_t * thread_id, pthread_attr *attributes,
                    (void *) (start_fct *)(void *), void *start_fct_arg);
void pthread_exit  (void *return_value);
int  pthread_join  (pthread_t thread_id, void *return_value);
int  pthread_cancel(pthread_t thread_id);
int  pthread_detach(pthread_t thread_id);
```

`pthread_create` erzeugt dabei einen (Tochter-)Thread, der mit dem Aufruf der Funktion, auf die `start_fct` verweist, abzulaufen beginnt. Dabei wird `start_fct_arg` an `*start_fct` als Argument übergeben. Die Thread-Attribute umfassen die Priorität, mit der der Thread Rechenzeit zugeteilt bekommt, sowie Variablen, deren Zustand das Verhalten bei Threadende und bei Versuchen definieren, den Thread von außen zu beenden. Es kann statt einer Liste von Attributen ein Null-Zeiger übergeben werden, der ein durch POSIX bestimmtes Defaultverhalten einstellt.[1] `pthread_exit` kann vom Tochterthread gerufen werden, wenn sich dieser beenden möchte (`pthread_exit` ist eine Funktion, die wie `exit()` nicht in die aufrufende Funktion zurückkehrt). Dieser Aufruf dient zur Synchronisation z.B. mit `pthread_join`, das dem Mutterthread erlaubt, auf die Beendigung des Tochterthreads zu warten. `pthread_join` sorgt auch dafür, dass Ressourcen des Tochterthreads wieder freigegeben werden. Natürlich wird der Thread auch beendet, wenn die anfangs aufgerufene Funktion ein `return` ausführt, wobei die Wirkung wie diejenige eines `pthread_exit` ist. Mittels `pthread_cancel` kann der Mutterthread versuchen, die Beendigung des Tocherthreads zu erzwingen; das genaue Verhalten des Tochterthreads wird dabei durch die anfangs übergebenen Attribute geregelt, möglicherweise ignoriert dieser eine cancel Aufforderung. Nach `pthread_cancel` muss noch `pthread_join` erfolgen, um die Thread-Ressourcen freizugeben (außer man hat vorher `pthread_detach` aufgerufen, was diese Ressourcen automatisch freigibt, wenn der Thread seine Ausführung beendet). Die Verwendung von `pthread_cancel` in objektorientiertem Kontext sollte mit äußerster Vorsicht erfolgen, da es dazu führen kann, dass ggf. Destruktorcode nicht ausgeführt wird oder auch Locks nicht sauber weider freigegeben werden.

Wir wollen eine C++-Klasse konstruieren, die die als Thread auszuführende Funktion als virtuelle Methode implementiert, den Thread explizit durch Aufrufen einer `start`-Elementfunktion startet und durch Aufruf von `pthread_join` (Wiederzusammenführen des Mutter- mit einem beendeten Tochterthread) in entsprechenden Elementfunktionen bzw. im Konstruktor beendet:

```
// thread.h:

#include <pthread.h>

extern "C" {
  // Typ einer C-function, die sich als start_fct eignet
  typedef void * (*thread_fct)(void *);
}

class Thread {
public:
  Thread() : m_id(0), m_running(false) {
  }

  ~Thread(){
    // join wird i.d.R. benoetigt zur Ressourcenfreigabe
    join();
  }

  // wartet auf Beendigung des Threads, gibt Ressourcen frei
  void * join(){
    if ( m_running ){
      void *return_value;
```

[1] Nähere Informationen findet man unter UNIX z.B. durch das textorientierte Manual mit dem Kommando `man pthread_create`

```
      pthread_join(m_id, &return_value); // m_running == false
      return return_value;
    }
    else
      return 0;
  }

protected:
  // 'richtiger' Code in der abgeleiteten Klasse
  virtual void * thread()=0;

  // Thread starten
  void start(){
    if ( ! m_running ){
      m_running = true;
      // unkritischer cast einer statischen C++-Elementfunktion auf eine
      // "normale" C-Funktion (Vorsicht, wenn die Plattform
      // Formen des Funktionsaufrufs nutzt), verwendet Default-Attribute
      pthread_create(&m_id,0,(thread_fct) runThread, this);
    }
  }

private:
  static void *runThread(Thread *a){
    void * retval = a->thread();
    a->m_running = false;  // unsauber: Race-condition
    pthread_exit( retval );
    return 0; // nie ausgefuehrt
  }

  pthread_t     m_id;
  volatile bool m_running;
};
```

Mit diesem Design können wir den Thread leider noch nicht einem (potentiellen) Konstruktor von `Thread` starten, weil die rein-virtuelle Funktion `thread` zu dem Zeitpunkt noch nicht bekannt ist. Der Start des Threads kann jedoch im Konstruktor der ableitenden Klasse nachgeholt werden. Dadurch dass `start` als `protected` deklariert wurde, ist die abgeleitete Klasse gezwungen, den Thread entweder im Konstruktor zu starten oder `start` in der `public` Sektion der Klasse an einen Benutzer herauszureichen.

Man beachte die Benutzung des Schlüsselwortes `volatile` für den Threadzustand, welches dem Compiler verbietet Optimierungen durchzuführen, die auf der Annahme beruhen, dass sich `m_running` nicht ändert, wenn keine Elementfunktionen ausgeführt werden.

Im folgenden Beispiel implementieren wir einen Thread, der eine Folge von Integern ausgibt und durch eine Eingabe auf `std::cin` im Mutterthread beendet werden kann.

```
// count.cc:

#include "thread.h"
#include <iostream>
#include <unistd.h>  // sleep

class Count : public Thread {
public:
```

```
  Count(): m_go(true) { start(); }

  void *thread(){
    int i = 0;
    while ( m_go ){
      std::cout << i++ << std::endl;
      sleep(1);
    }
    return 0;
  }

  void stop(){
    m_go = false;
  }
private:
  volatile bool m_go;
};

int main(){
  Count cnt;  // starts counting immediately
  char terminate = 'n';
  while ( terminate != 'y' )
    std::cin >> terminate;
  cnt.stop();
  // join is called in the Thread dtor here
}
```

8.2.2 Synchronisierung

Im obigen Beispiel erfolgte eine einfache Kommunikation zwischen den Threads über die Elementvariable `m_go`, die in einem Thread gelesen und im anderen geschrieben wurde. Wenn zwei Threads auf die gleiche Ressource zugreifen, ergeben sich häufig Probleme dadurch, dass die Threads vom Betriebssystem an einer beliebigen, nicht vorhersehbaren Stelle im Maschinencode unterbrochen werden können. Ein einfaches Beispiel ist eine Funktion, die in einer statischen Variable zählt, wie häufig sie aufgerufen wurde:

```
int count(){
  static calls;
  calls = calls + 1; // kritisch bei Unterbrechung
  return calls;
}
```

Wird hier etwa bei Ausführung der Berechnung `calls + 1` der bearbeitende Thread gerade vor der Speicherung des Ergebnisses unterbrochen und ein zweiter Thread übernimmt die Kontrolle, dann wird dieser in der Variable `calls` noch den alten Wert finden, diesen um 1 erhöhen und in `calls` speichern. Wird der erste Thread dann wieder aktiv, so speichert er die gleiche Zahl in `calls`, damit hat der Zähler jetzt einen Aufruf der Funktion nicht registriert. Solche Programmteile, die wie die Zuweisung `calls = calls + 1` nur von einem Thread allein ausgeführt werden dürfen, um korrektes (d.h. deterministisches) Programmverhalten zu erzielen, bezeichnet man auch als kritische Programmabschnitte (*critical sections*).

Prozeduren, die Synchronisationsmechanismen enthalten, die solche (und ähnliche) Mehrfachzugriffe verhindern, heißen *threadsicher*. Die meisten Funktionen, die nur lokale Variablen bearbeiten, sind von sich aus threadsicher, so z.B. die mathematischen Funktionen

wie exp(), log(), etc., die Zwischenergebnisse immer auf dem Stack ablegen und immer auf Kopien der Originalvariablen arbeiten. Ist der gleichzeitige Zugriff mehrerer Threads auf eine Funktion möglich, so bezeichnet man die Funktion als *reentrant*.

Um dem Programmierer die Kontrolle über den Programmausführung in kritischen Programmabschnitten zu geben, werden betriebssytemabhängig verschiedene Mechanismen zur Verfügung gestellt. Wir werden hier allerdings nur die Mittel der POSIX Threadbibliothek erläutern. POSIX Threads bieten zum Schutz der *critical sections* sogenannte *mutex* (mutual exclusion) Objekte an. Vor dem Eintritt in einen kritischen Abschnitt muss sich ein Thread alleinigen Zugriff verschaffen und den Zugriff anderer Threads verbieten. Man spricht in diesem Zusammenhang vom Anlegen eines *Locks* . Ist der Bereich bereits gelockt, so muss der anfragende Thread warten, bis es wieder möglich wird, ein Lock anzulegen. Die wichtigen Funktionen sind

```
// pthread.h:

pthread_mutex_init   (pthread_mutex_t *, const pthread_mutexattr_t *);
pthread_mutex_destroy(pthread_mutex_t *);
pthread_mutex_lock   (pthread_mutex_t *);
pthread_mutex_unlock (pthread_mutex_t *);
pthread_mutex_trylock(pthread_mutex_t *);
```

`pthread_mutex_t` ist eine Datenstruktur, die alle zum Mutex gehörende Information aufnimmt. `pthread_mutex_init` initialisiert diese Struktur unter Berücksichtigung der angegebenen Attribute, die das Mutex-Verhalten insbesondere bezüglich des Verhaltens bei weiteren Lock-Versuchen aus dem gleichen Thread detailliert regeln. Als Defaultverhalten bei Übergabe eines Null-Zeigers wird sich ein Thread selbst blockieren, wenn er mehr als einmal versucht, einen Mutex mit `pthread_mutex_lock` zu holen, ohne ihn zwischenzeitlich mit `pthread_mutex_unlock` wieder freigegeben zu haben. `pthread_mutex_trylock` holt einen Mutex, wenn dieser vorher frei war oder gibt einen Fehlercode zurück, wenn der Mutex vorher bereits im Eigentum eines anderen Threads war. `pthread_mutex_destroy` gibt alle mit einer Mutex-Datenstruktur verbundenen Ressourcen frei. Ein Mutex sollte freigegeben sein, damit dieser Aufruf Erfolg hat.

Wie im Beispiel der Threads schreiben wir eine kurze Wrapper-Klasse um diese Aufrufe herum

```
// mutex.h:

#include <pthread.h>

class Mutex {
public:
  WTR_Mutex() {
    pthread_mutex_init( &m_Mutex, NULL);
    release();
  }

  ~WTR_Mutex () {
    pthread_mutex_destroy( &m_Mutex );
  }

  // schliesst/holt einen freien Mutex
  void lock() {
    pthread_mutex_lock( &m_Mutex );
  }
```

```
  // gibt einen Mutex wieder frei
  void release() {
    pthread_mutex_unlock( &m_Mutex );
  }

  // holt einen mutex, wenn dieser frei ist und gibt ansonsten
  //  false zurueck
  bool try() {
    return pthread_mutex_trylock( &m_Mutex ) == 0;
  }

private:
  pthread_mutex_t m_Mutex;
};
```

Mit dieser Klasse ließe sich unsere `count()` Funktion folgendermaßen threadsicher(er) machen:

```
#include "mutex.h"

Mutex m;

// count() zaehlt Aufrufe korrekt, aber garantiert nicht,
//  dass verschiedene Threads auch verschiedene
//  Rückgabewerte erhalten:
int count(){
  m.lock();
  static calls;
  calls = calls + 1;
  m.release();
  // mögliche Unterbrechung durch zweiten Thread
  return calls;
}
```

In dieser Weise wird die Sektion, in der die Variable `calls` modifiziert wird, gesichert. Es kann allerdings immer noch vorkommen, dass zwei parallel laufende Threads die gleiche Zahl zurückgeliefert bekommen (wenn der zweite Thread die Kontrolle bekommt gerade bevor der erste das `return` ausführen will), aber es wird kein Aufruf unterschlagen.

Ein weiterer Ressourcen-Schutz sind die sogenannten *Semaphoren,* die im Wesentlichen Integer-Variablen sind, die aus verschiedenen Threads heraus inkrementiert und dekrementiert werden können. Es existieren "wait" Funktionen, mit denen ein Thread darauf warten kann, dass eine Semaphore von Null verschieden wird. Semaphoren werden eingesetzt, wenn man mehrere begrenzte Ressourcen eines Typs hat, z.B. fünf Netzwerkkarten in einem Server, die auf die Threads aufgeteilt werden müssen. Semaphoren implementieren gewissermaßen ein "reference counting" Paradigma für die verwalteten Ressourcen.

Sowohl im Falle von Mutex-Variablen als auch von Semaphoren ist es von äußerster Wichtigkeit, sicher zu stellen, dass die Anforderung einer Ressource und deren Freigabe immer paarweise vorgenommen werden. Wird etwa im letzten Beispiel das Lock um `calls = calls + 1` herum nicht wieder freigegeben (der Aufruf an `release()` erfolgt nicht), so wird sich das Programm selbst blockieren (dead-lock), wenn beim nächsten Eintritt in die Funktion `count()` der aufrufende Thread auf ein Lock wartet, das niemals freigegeben wird...

Conditions werden zur Synchronisierung eingesetzt, um das Eintreffen einer bestimmten Bedingung an einen oder mehrere Threads zu melden. Dazu wird eine Condition-Variable

von einem Thread vorbereitet und andere Threads können auf das Eintreffen bzw. den Zustandswechsel dieser Variablen warten.
F: Woran erkennt man critical sections?
A: Allgemein gilt, dass alle Operationen auf globale und statische Variablen und die Modifikation aller sonstigen nichtstatischen Objekte, die sich zwei oder mehrere Threads teilen, geschützt werden müssen. Falls die Operation auf dem Objekt atomar ist (wie z.B. einzelne Assembleranweisungen, die eine Variable im Speicher inkrementieren), kann man im Prinzip auf einen Schutz verzichten, aber in der Praxis, beim Umgang mit Hochsprachen wie C oder C++, weiß man nie, welche Operationen vom Compiler erzeugt und ggf. atomar realisiert werden.

Kapitel 9

C++ und die weite Welt

9.1 Entwicklungsziele, Eigenschaften und Geschichte von C++

C++ entwickelte sich aus Programmiererwünschen heraus kontinuierlich aus C. Zunächst wurden Strukturen zu Klassen erweitert, das Vererbungskonzept und virtuelle Funktionen eingeführt (C with classes). Die Zugriffskontrolle auf Datenelemente sollte die nötige Kommunikation zwischen Designer und Anwender einer Klasse unterstützen. Referenzen waren nötig, um die Eigenschaften überladener Operatoren mit denen der eingebauten vergleichbar zu machen.

Frühe Implementierungen von C++ wiesen eine Deklarationsdatei namens *generic.h* auf, die eine Makrosammlung anbot, um bestimmte Aspekte der generischen Programmierung, insbesondere die Implementierung von Containertypen, mit Hilfe des Präprozessors zu imitieren. Die generische Programmierung mit Templates wurde mit der Entscheidung für eine weitgehende Unterstützung der Standardbibliothek eingeführt, um Container-Typen zu unterstützen, deren Inhalte zur Compilezeit überprüfbar sein sollten und die keine Laufzeitnachteile nach sich ziehen sollten. Sie entstanden auch aus dem Wunsch, Präprozessor-Makroanweisungen und die mit deren Verwendung verbundenen, meist unerwünschten Nebeneffekte zu vermeiden.

Einen weiteren nicht unerheblichen Einfluss bei der Auswahl und Entwicklung weiterer Spracheigenschaften hatte die Standardbibliothek, die zu ihrer Implementierung Eigenschaften wie Default-Template-Argumente, Template-Elementfunktionen und Templates als Template-Argumente, sowie neue Schlüsselworte wie `mutable` und `explicit` verlangte.

Ausnahmebehandlung (`try{} catch(){}`) und Namensräume kamen zum Sprachumfang hinzu, um wichtige Probleme bei der Entwicklung von Bibliotheken, insbesondere die konsistente Fehlerbehandlung, anzugehen. Relativ spät wurde die Sprache um Laufzeittypinformation (`typeid`, `dynamic_cast<>`) ergänzt.

Dem Design von C++ liegen die folgenden Prinzipien zu Grunde:

1. Kompatibilität mit C, d. h. dass alle C-Programme auch gültige C++ Programme sind. Bis auf wenige Ausnahmen, die insbesondere durch die strengere Typüberprüfung in C++ bedingt sind, ist dies der Fall.

2. Statische Typüberprüfung, d.h. dass das Zusammenpassen von Funktionsdeklarationen und -aufrufen bereits zur Compilezeit überprüft wird, wo immer das möglich ist.

3. Vermeidung von Spracheigenschaften, die die Laufzeiteffizienz in Frage gestellt hätten. So wurden z.B. wurden Konstrukte vermieden, die als Default Information in jedem

Objekt gespeichert hätten (wie etwa der Zeiger auf eine Tabelle virtueller Funktionen), so dass die Eigenschaft, ein Objekt einer Klasse zu sein, keinen zusätzlichen Speicherplatzbedarf impliziert. Gleichzeitig sind auch alle Elementfunktionen als Default nicht-virtuell, um den geringfügig kostspieligeren Aufruf einer virtuellen Methode zu vermeiden. Aus eben diesem Grund wurden auch Laufzeittypinformation und die Fehlerbehandlung mit Exceptions erst sehr spät und kurz vor der Standardisierung in die Sprache integriert.

4. Benutzbarkeit in traditionellen Programmierumgebungen, d.h. C++ greift zurück auf systemeigene Linker, Loader-Programme und Laufzeitumgebungen.

5. Vermeidung von high-level Konstrukten wie Matrix-Typen, komplexen Zahlen, Zeichenketten, flexible Feldvariablen, etc. als Sprachbestandteilen. Die Sprache C++ stellt vielmehr die Mittel zur Verfügung, damit ein Benutzer oder eine Bibliothek solche Typen selbst definieren und problemlos in die existierende Umgebung integrieren kann.

C++ erzwingt keine Objektorientierung, es bietet "nur" die Sprachmittel an, die erforderlich sind, um objektorientierte Programme zu schreiben. C++ unterstützt vielmehr eine Vielzahl von Programmierparadigmen: prozedurale, modulare, objektorientierte, und generische (typunabhängige) Programmierung.

9.2 Java — zum Vergleich mit C++

Java ist durch die Bereitstellung einer plattformübergreifenden Klassenbibliothek, die neben algorithmischen Komponenten auch grafische Benutzeroberflächen anbietet und durch die anschließende Implementierung in populärer Internetsoftware (netscape, Explorer) zu der wichtigsten universellen Programmiersprache geworden, um Applikationen für das WWW zu schreiben. Die Definition der Klassenbibliothek(en) ist Bestandteil von Java und garantiert daher durch ihre Einheitlichkeit über verschiedene Plattformen hinweg ein ungewöhnlich hohes Maß an Portabilität.

Java-Sourcecode wird zunächst in plattformunabhängigen Java-Bytecode "kompiliert", d.h., nach Optimierungen im Wesentlichen in einfach weiterverarbeitbare Tokens übersetzt. Der Java-Bytecode kann dann portabel auf verschiedenen Plattformen mit Hilfe eines Interpreters, der unterstützenden Klassenbibliothek und eines "just in time compilers" (JIT) ablaufen. Die Kombination aus Interpreter und JIT trägt den Namen "Java virtual machine" (JVM). Die JVM unterzieht den auszuführenden Code noch weiteren Validitäts- und Sicherheitstests: etwa der Verifikation von Zertifikaten, der Verifikation des Bytecodes vor der Ausführung bezüglich unimplementierter Codes und Speicherzugriffsverletzungen; letztere werden grundsätzlich auch während der Laufzeit des Programmes geprüft.

Der hohen Portabilität[1] und Sicherheit steht der Nachteil des Interpreterkonzeptes entgegen, das durch seine Allgemeinheit eine verhältnismäßig langsamere Programmausführung im Vergleich zu Programmcode bedingt, der direkt für die entsprechende Plattform kompiliert wurde. Je nach Testprogramm und Tester zwischen läuft Java-Bytecode 1 bis 10-mal langsamer als äquivalenter kompilierter Code.

Die Java-Syntax hat im Bereich elementarer Funktionalität Ähnlichkeiten mit der von C/C++. Die wichigsten Eigenschaften und Unterschiede von Java zu diesen Sprachen sind im Folgenden zusammengefasst.

[1] In der Praxis ergeben sich allerdings Einschränkungen auf Grund von Fehlern in den Implementierungen des Java-Systems, das neben dem Interpreter auch umfangreiche Bibliotheken umfasst. Weiterhin existieren mehrere Versionen, so dass neuere Programme auf älteren Systemen ggf. nicht mehr lauffähig sind.

1. gemäß Sun Microsystems ist Java *a simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language.*

2. Organisation des Programmcodes: Java erzwingt, dass jeder Programmcode in Klassen definiert wird. Jede (als `public` deklarierte) Klasse erfordert eine eigene Quelldatei, deren Name sich aus dem der Klasse mit nachgestelltem ".java" ergibt. Eine der Klassen enthält eine Methode mit dem Namen `main` als Programmstartpunkt. Java und Zeichenketten in Java basieren auf Unicode, einer Zwei-Byte-Darstellung für die Schriftzeichen der meisten Sprachen der Welt.

   Das Java "Hello World!"-Programm sieht so aus:

```
public class HelloWorld {
  public static void main(String[] arg){
    System.out.println("Hello World!");
  }
}
```

   Nach Übersetzung mit `javac`, wobei eine Datei `HelloWorld.class` erzeugt wurde, lässt man das kompilierte Programm durch den Interpreter `java` ausführen; hier als Beispiel die UNIX-übliche Befehlssequenz:

```
bash> javac HelloWorld.java
bash> java HelloWorld
Hello World!
```

   Da Klassen- und Dateinamen übereinstimmen, sind keine Hinweise (include-Direktiven) an den Compiler nötig (Java kennt keinen Präprozessor), wo weiterer Programmcode zu finden ist. Weiterhin gibt es keine Trennung von Implementierungs- und Deklarationsdateien, jeglicher Code findet sich in der `.java` Datei. Weitere wichtige Entwicklungstools sind:

   javadoc zur Erstellung von Dokumentation aus Kommentaren im Programm,

   javah zur Erzeugung von Header-Dateien für die Anbindung von Programmen in C an java,

   jdb zum Debuggen von Java-Programmen,

   appletviewer

3. Elementare Datentypen

Typ	Inhalt	Default	Größe
`boolean`	true/false	false	1 bit
`char`	Unicode	`\u0000`	16 bit
`byte`	signed Integer	0	8 bit
`short`	signed Integer	0	16 bit
`int`	signed Integer	0	32 bit
`long`	signed Integer	0	64 bit
`float`	IEEE Fließkommaformat	0.0	32 bit
`double`	IEEE Fließkommaformat	0.0	64 bit

   Die elementaren Typen werden als einzige Datentypen als Werte an Funktionen übergeben. Nach der Default-Initialisierung enthält die Variable immer den Wert 0 und

verbleiben nicht uninitialisiert wie in C oder C++. Die bitweise Repräsentierung der Datentypen ist bei allen Java-Implementierungen gleich.

Im Gegensatz zu C++ sind alle elementaren Typen auch in umschließenden Klassen verfügbar, die von `Object` und mit Ausnahme von `boolean` auch von `Number` abgeleitet sind und so über zusätzliche Elementfunktionen (Umwandlung in String, Zuordnung eines HashCodes, etc.) `Boolean`, `Integer`, ..., usw. verfügen. Da es sich um echte Objekte handelt, kann von ihnen abgeleitet und neue Funktionalität durch weitere Elementfunktionen implementiert werden. Umgekehrt gibt es in Java keine Möglichkeit, die Klassenschnittstelle durch das Überladen von Operatoren der der eingebauten Typen anzunähern.

Alle anderen so genannten Referenztypen werden mit `new` auf dem Heap alloziert. Nicht initialisierte Referenzvariablen haben den speziellen Wert `null`, gegen den auch Tests durchgeführt werden können. Syntaktisch verhalten sie sich bei Zugriff auf Datenelemente und Elementfunktionen wie C++-Referenzen, bezüglich der Variablenübergabe an Unterfunktionen wie Zeiger:

```
// Wizard.java:
public class Wizard {
  private String broom;
  private double speed;

  /** Konstruktoren */
  public Wizard(){};
  public Wizard(String abroom){
     setBroom(abroom);
  }

  public void setBroom(String atype){ broom = atype; }
  public String getBroom(){ return broom; }

  /** versuchter Besentausch */
  public void swapBrooms(String otherBroom){
    String tmp = broom; // tmp ist Referenz, keine Kopie!
    broom = otherBroom;
    otherBroom = tmp; // ueberschreibt temporaeres Argument
                      // alter Besen geht hier verloren
  }
  public void accelerate( double increment ){ // by value-Uebergabe
    if ( increment > 1. )
       broom = "broken";
    else
       speed += increment;
  }
}

// HarryPotterGo.java:
public class HarryPotterGo {
  static Wizard harry_potter = new Wizard();
  static String his_broom = "Firebolt 42";

  public static void main(String args[]){
    harry_potter.setBroom(his_broom);
```

```
    harry_potter.accelerate(1.1);
    System.out.println(harry_potter.getBroom() + " " + his_broom);
    harry_potter.swapBrooms(his_broom);
    System.out.println(harry_potter.getBroom() + " " + his_broom);
  }
}
```

Dieses Programm erzeugt die Ausgabe

```
broken Firebolt 42
Firebolt 42 Firebolt 42
```

Methoden sind per Default virtuell, außer sie werden mit `final` als nicht überschreibbar deklariert. `const` hat die aus C++ bekannte Bedeutung.

4. Garbage Collection. Im obigen Beispiel werden Referenzvariablen mit `new` initialisiert, aber nicht mit einer dazu inversen Operation wieder freigegeben. Das in C++ explizit erforderliche "delete" wird in Java durch Teile der Laufzeitumgebung durchgeführt, wenn ersichtlich ist, dass auf ein Objekt keine Referenz mehr verweist und das Objekt demzufolge aus dem Speicher entfernt werden kann. Diese Technik der Speicherverwaltung heißt "Garbage Collection". Die Freigabe des Speichers kann nötigenfalls durch `a = null` erzwungen werden. Die Rolle eines Destruktors zur Freigabe anderer Ressourcen als Systemspeicherplatz wird durch eine Methode mit dem Namen `finalize()` übernommen.

5. Java-Felder sind deutlich flexibler als ihre C und C++-Verwandten angelegt. Beliebige mehrdimensionale Felder können in beliebiger Form (Felder von Feldern unterschiedlicher Größe) zur Laufzeit angelegt werden. Ihre aktuelle Größe kann jederzeit durch `length()` abgefragt werden.

```
int      var = 23;
double[]   d = new double[var];   /* eindimensional */
Object[][] O = new Object[var][]; /* zweidimensional */
O[1][]       = new Object[42];
int size     = O[1].length();
```

6. Zuweisungen sind entweder als Referenz möglich (= Operation) oder durch Kopieroperationen, die entweder als gewöhnliche Elementfunktionen oder vorzugsweise durch Implementierung der `Cloneable`-Schnittstelle implementiert sind.

```
Vector v(12);           /* 12 Elemente anfaengliche Kapazitaet */
Vector w = v;           /* w, v zeigen auf denselben Speicherbereich */
Vector u = v.clone();   /* werden bitweise kopiert */
```

Eine ähnliche Unterscheidung wird bei Vergleichen gemacht

```
if (w == v) {...}       /* identische Speicherbereiche? */
if (w.equals(v)) {...}  /* identische Inhalte?          */
```

7. Java kennt keine Mehrfachvererbung, dafür kann eine Schnittstelle wie in einer abstrakten C++-Basisklasse in einem so genannten Interface festgelegt werden. Jede Klasse kann beliebig viele dieser Interfaces implementieren.

```
public class WizardAtHogwarts extends Wizard
                              implements Cloneable {
  String gadgets[];

  public WizardAtHogwarts(){
    super("Hogwart Broom 2000"); // expliziter Aufruf des
                                 //  Basisklassenkonstruktors
  }

  public WizardAtHogwarts(String gadgets[])
  {
    this();                      // rufe Konstruktor ohne Argumente
    this.gadgets = new String[gadgets.length];
    for ( int i=0; i < gadgets.length; ++i )
      this.gadgets[i] = gadgets[i];
  }

  public Object clone(){
    WizardAtHogwarts another = new WizardAtHogwarts(gadgets);
    another.setBroom(this.getBroom());
    return another;
 }

}
```

Im Beispiel wurde das Interface Cloneable implementiert. Da der Rückgabetyp dort als `Object` deklariert ist, verliert man die Information, welche Klasse kopiert wurde und muss diese Information durch einen Cast wiederherstellen. Weiterhin merkt man bei der nötigen Implementierung von Cloneable, dass es nicht möglich ist, einen sauberen Clone zu erhalten, wenn nicht auch schon die Basisklasse über eine solche Funktion verfügt (was analog dem C++ Kopierkonstruktur ist, der dort allerdings für jede Klasse existiert) — im obigen Beispiel lässt sich das Datenelement `speed` von `Wizard` nicht kopieren, das in der Basis(Super)klasse privat ist. Unten zeigen wir, dass sich `Wizard` und `WizardAtHowarts` polymorph verhalten: Ob eine Klasse von einer anderen abgeleitet ist oder ein Interface implementiert, kann zur Laufzeit mit dem Operator `instanceof` geprüft werden.

8. Packages dienen der Gruppierung von Klassen und entsprechen in ihrer Funktion etwa den Namensräumen von C++. Package-Namen sind beliebig tiefe, durch Punkt getrennte Namensanreihungen, die Verzeichnispfaden innerhalb der jeweiligen Betriebssystemsumgebung entsprechen. Diese Pfade sind relativ zum aktuellen Verzeichnis oder dem durch die Umgebungsvariable CLASSPATH vorgegebenen Verzeichnis.

```
package meinPaket;            // optionale Anweisung am Dateianfang
import  com.sun.java.swing.*; // erlaubt abgekürzte Verwendung von
                              //  Klassennamen aus diesem Paket.
                              //  '*' ist nicht rekursiv
```

Die Sichtbarkeitsmodifikatoren für Klassenelemente entsprechen etwa denen, die auch in C++ verwendet werden, der in Java vorhandene `package`-Modifikator erklärt gewissermaßen Freundschaft für alle anderen Klassen im Paket:

Status	Modifikator
`public`	alle anderen Klassen
`protected`	Subklasse in gleichem oder anderem Paket
`package`	beliebige Klasse in gleichem Paket
`private`	gleiche Klasse

9. Sonstiges zur Objektorientierung: Interfaces, abstrakte, innere, lokale und anonyme Klassen.

   Interfaces sind rein virtuellen Klassen in C++ vergleichbar und definieren eine Programmierschnittstelle mit Methoden oder/und Konstanten. Sie sind, wie Klassen, Datentypen und können daher zum Beispiel in Feldern verwaltet werden. Alle Klasseninstanzen, die das Interface implementieren, können auf die implementierte Instanz zugewiesen werden.

   Abstrakte Klassen sind wie in C++ solche, die mindestens eine abstrakte Methode enthalten, d.h. eine als `abstract` deklarierte Methode ohne Implementierung. Sowohl die Methode(n) als auch die Klasse muss als `abstract` erklärt werden und kann damit wie in C++ nicht instantiiert werden.

   Innere Klassen sind als Elemente der umschließenden Klasse deklariert und haben Zugriff auch auf die `private` erklärten Elemente der umschließenden Klasse.

   Lokale Klassen sind in beliebigen Blöcken von Java-Code definiert und nur in diesem Block sichtbar. Sie sind praktisch, wenn für einen bestimmten Zweck (etwa zur Registrierung eines "Callbacks") lokal nur eine einzige Instanz einer Klasse benötigt wird, die Zugriff auf die (als `final` erklärten) Blockvariablen hat.

   Anonyme Klassen sind lokale Klassen ohne Namen, die direkt an der Stelle ihrer Verwendung deklariert werden. C++ kennt keine anonymen Klassen.

10. Die Schleifenkonstrukte und `if`-Ausdrücke sind äquivalent zu denen in C/C++ und verwenden den Typ `boolean`.

11. Exception-Handling erfolgt durch das Konstrukt
    `try{...} catch(Exception e){...} finally{...}`
    Der nach beliebig vielen `catch(..){}` Anweisungen folgende `finally{}`-Teil wird dabei immer ausgeführt, egal ob eine Ausnahme auftritt oder nicht. Er enthält Code, der Ressourcen freigibt, die bei Auftreten einer Ausnahme sonst nicht freigegeben würden. In C++ steht der entsprechende Code üblicherweise in den Destruktoren und es obliegt dem Compiler, deren Ausführung bei Auftreten einer Ausnahmebehandlung sicherzustellen.

12. Nicht vorhanden sind in Java:

    - `enum`, `union`, `struct`
    - Präprozessor und die zugehörigen Direktiven
    - Zeigervariablen (nur Referenzen (wie FORTRAN) und die spezielle Rolle der elementaren Datentypen)
    - `sizeof()`, da die Größe jedes Typs plattformunabhängig ist und daher vom Programmierer bestimmt werden kann.
    - Templates (möglicherweise in Java 2)
    - Überladen von Operatoren (mit Ausnahme von `+` für `String`)
    - Defaultargumente für Funktionen, da ihre Funktionalität leicht durch Überladen der Funktion nachgebildet werden kann.

Hier ist ein weiteres, komplexeres Programmbeispiel, das zeigt, wie ein Feld von `Buttons` erzeugt wird, die bei Aktivierung ein Fenster darstellen, das über ihre Position informiert.

```
// default-Package

import com.sun.java.swing.*;    // JFrame, JButton
import java.awt.*;              // GridLayout
import java.awt.event.*;        // ActionListener

public class FrameTest extends JFrame implements ActionListener
{
   int dim = 3;

   public static void main(String[] args)
   {
      FrameTest ft = new FrameTest("Hauptfenster");
   }

   public FrameTest(String title)
   {
      super(title);
      JPanel panel = new JPanel();
      panel.setLayout(new GridLayout(dim,dim));
      JButton[][] button = new JButton[dim][dim];  // Anlegen des Arrays
      for (int i=0; i<dim; i++)
      {
         for (int j=0; j<dim; j++)
         {
            // Einfuegen der Buttons
            button[i][j] = new JButton(new Integer(j+i*dim+1).toString());
            button[i][j].addActionListener(this);
            panel.add(button[i][j]);
         }
      }
      WindowListener l = new WindowAdapter()
      {
         public void windowClosing(WindowEvent e) {System.exit(0);}
      };
      addWindowListener(l);
      getContentPane().add(panel);
      pack();
      show();
   }

   public void actionPerformed(ActionEvent e)
   {
      if (e.getSource() instanceof JButton)
      {
         JButton button = (JButton)(e.getSource());
         JOptionPane.showMessageDialog(this, "pressed: " +
          button.getText(), "Dialogfenster",JOptionPane.INFORMATION_MESSAGE);
      }
   }
}
```

Mehr Informationen zu Java erhält man z.B. in den folgenden Referenzen im WWW (Stand 1998),

`http://java.sun.com/docs/books/jls/html/index.html` Java Sprachdefinition,

`hppt://java.sun.com/docs/books/tutorial` Tutorial,

oder in gedruckter Form etwa in [5] (Java 1.1).

Anhang A

Beispiele

A.1 Einführendes Beispiel

Ein etwas komplexeres Hello-World-Programm als Beispiel für ein sehr kurzes, untypisches aber korrektes C++ Programm, das viele syntaktische Strukturen eines C-Programms nutzt.

```
13:16 boa_sts:Material/programs> cat sqrt_ex.cc

#include <iostream>   /* C++ Standardbibliothek, kein .h */
#include <math.h>     /* C Standardbibliothek */

// Startpunkt main ist Schluesselwort, immer vom Typ int
int main()
{
  // Variablendeklarationen sind moeglich irgendwo vor erster Benutzung
  // 'i' ist nur innerhalb der for-Schleife sichtbar
  for (int i=0; i < 10; ++i) {

    // jedes Mal, das der'for'-Block durchlaufen wird, wird
    //  Speicherplatz fuer d neu reserviert und mit -1 initialisiert.
    double d = -1.;

    // Vom Terminal lesen, bis d positiven Wert hat
    while ( d < 0. ) {

      std::cout << "enter a positive number: ";
      std::cin  >> d;
      if ( d < 0. )
        std::cout << d << " is < 0; please ";
    }
    // Wurzel berechnen ...
    double sqrt_d = sqrt(d);

    // ... und ausgeben
    std::cout << "     sqrt(" << d << ") = " << sqrt_d << "\n";
  }

}

13:16 boa_sts:Material/programs> g++ sqrt_ex.cc
13:17 boa_sts:Material/programs> a.out
enter a positive number: 2.
     sqrt(2) = 1.41421
```

A.2 Programmierstile

In Folgenden sollen am Beispiel eines Zufallszahlengenerators unterschiedliche Programmierstile dargestellt werden.

A.2.1 Prozedurale/Modulare Programmierung

Das folgende C-Programm ist ein typisches Beispiel für die Erzeugung von Zufallszahlen in einer prozeduralen Weise. In der Deklarationsdatei werden zwei Funktionen zur Initialisierung des Generators bzw. zur Erzeugung von Zufallszahlen vorgesehen.

```
// rand1.h:

#ifndef RAND1_H  /* include guards - in Deklarationsdateien erfordert */
#define RAND1_H

void rand1_seed(int seed);
int  rand1_int(void);

#endif
```

In der Implementierungsdatei finden sich Konstanten, die das Verhalten des Zufallszahlengenerators kontrollieren und eine Variable, die die jeweils letzte erzeugte Zufallszahl speichert. Diese ist gleichzeitig der Status des Generators.

```
/* rand1.c: */

#include <stdio.h>
#include "rand1.h"

static int       g_last  = -1;  /* globale Variable auf Dateibezugsebene */
static const int gc_add  = 51349;
static const int gc_mult = 4561;
static const int gc_max  = 243000;

void rand1_seed(unsigned seed){
  if ( seed >= 0 )
    g_last = seed;
  else
    printf( "seed value < 0" );
}

int rand1_int(void) {
  if ( g_last == -1 ){   /* nicht initialisiert */
    printf( "warning: calling rand1_seed(0)" );
    rand1_seed(0);
  }
  g_last = (g_last * gc_mult + gc_add) % gc_max;
  return g_last;
}
```

Die Benutzung des Generators erfolgt durch Aufruf der deklarierten Funktionen, wobei auf die richtige Reihenfolge geachtet werden sollte.

```
/* main.c: */
#include <stdio.h>
#include "rand1.h" /* to use rand1 functions */

int main(){
  int i;

  rand1_seed(10);
  for ( i=0; i < 2; ++i )
    printf ( "%d\n", rand1_int() );
}

14:18 boa_sts:Material/programs> g++ main_rand1.cc rand1.c -o rand
14:18 boa_sts:Material/programs> rand
96959
21348
```

Es fällt auf, dass bei der Erzeugung von Zufallszahlen immer geprüft werden muss, ob der Generator bereits initialisiert wurde. Will man diese Abfrage eliminieren, riskiert man die Verwendung eines nicht-initialisierten Generators. Weiterhin lässt sich gleichzeitig

nicht mehr als eine unabhängige Sequenz von Zufallszahlen erzeugen, denn der Generator kann ja nur genau einmal initialisiert werden. Die Verwendung unabhängiger Generatoren ist wünschenswert, wenn in verschiedenen Programmteilen Generatoren verwendet werden und man diese unabhängig voneinander verifizieren möchte. Die Verwendung eines globalen Generators ändert die Sequenz von Zufallszahlen und damit numerischen Resultate, wenn solche Programmteile zusammengefügt werden.

A.2.2 Objektorientiertes C++-Programm zur Erzeugung von Zufallszahlen

In der Deklarationsdatei erfolgt die Festlegung einer Schnittstelle für die Benutzung einer Klasse, die Daten und Algorithmen für den Zufallszahlengenerator zusammenfasst und zwar sowohl die Teile `(public)`, die für die Benutzung durch andere Programmteile gedacht sind als auch diejenigen, die nur für die Implementierung des Zufallszahlengenerators genutzt werden `(private)`.

```
// rand3.h:

#ifndef RAND3_H
#define RAND3_H

struct Random {
public:
  Random(int seed);  // Konstruktor, immer ohne Rückgabetyp
  int Int();         // normale Elementfunktion

private:
  int m_last;        // interne (Zustands-) Variablen
  // Konstanten, die sich alle Instanzen teilen:
  static const int m_add;  /*  = 51349; */
  static const int m_mult; /*  = 4561;  */
  static const int m_max;  /*  = 243000;*/
};

#endif
```

Die Implementierung des Generators sieht jetzt etwa so aus:

```
// rand3.cc :

#include "rand3.h"

Random::Random(int seed){
  if ( seed < 0 ) throw;  // kein konsistente Zustand moeglich
  m_last = seed;
}

int Random::Int(){
  return m_last = ( m_last * m_mult + m_add) % m_max;
}

const int Random::m_add  = 51349;
const int Random::m_mult = 4561;
const int Random::m_max  = 243000;
```

Die Benutzung des Generators gestaltet sich jetzt wie die Benutzung jedes “normalen”, eingebauten Datentyps. Das Hauptprogramm `main.cc` könnte vielleicht so aussehen:

```
// main.cc:

#include <iostream>
#include <rand3.h>
```

```
int main(){
  Random  r1(10);
  Random  r2(12);

  for ( int i=0; i < 10; ++i )
    std::cout << r1.Int() << " " << r2.Int() << "\n";
}
```

Durch die Objektorientierung und die Verwendung eines Konstruktors wird eine korrekte Initialisierung der beiden Generatoren vor ihrer ersten Verwendung erzwungen. Die Routine zur Erzeugung der Zufallszahlen vereinfacht sich entsprechend. Das Programm kann eine beliebige Anzahl von Zufallsgeneratoren verwenden, denn jeder bringt seine eigene Zustandsvariable mit.

A.2.3 Objektorientiertes C-Programm zur Erzeugung von Zufallszahlen

Objektorientierung ist ein Programmierstil, er kann mit Abstrichen an Eleganz und Lesbarkeit des Programms in jeder Programmiersprache verwirklicht werden. Als Beispiel soll hier ein C Programm dienen, das die Funktionen des obigen C++-Programms imitiert. Die im C++-Programm nicht sichtbaren Konstruktoren und Destruktoren müssen jetzt natürlich explizit gemacht werden. Die Klasse muss durch eine Struktur ersetzt werden.

```
/* rand2.h: */

#ifndef RAND2_H
#define RAND2_H

typedef struct {
  int last;
} RandomInternal;

typedef RandomInternal *Random;

Random rand2_create ( int seed );
int    rand2_int    ( Random rep );
void   rand2_destroy( Random rep );

#endif

#include <stdio.h>
#include "rand2.h"

int main(){
  Random r1 = rand2_create(10);
  int    i;

  for ( i=0; i < 10; ++i )
    printf ( "%d\n", rand2_int(r1) );

  rand2_destroy(r1);
}
```

Die Implementierung erfolgt in einer getrennten Datei und muss für den Benutzer nicht sichtbar sein – sie kann, z.B. in kompilierter Form als Objektdatei oder in einer Bibliothek vorliegen.

```
/* rand2.c: */

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

```
#include "rand2.h"

static const int gc_add  = 51349;
static const int gc_mult = 4561;
static const int gc_max  = 243000;

Random rand2_create(int seed) {
  Random new_rnd = (Random) malloc ( sizeof(RandomInternal) );
  if ( seed > 0 )
     new_rnd->last = seed;
  else {
     printf( "seed value < 0" );
     new_rnd->last = 0;
  }
  return new_rnd;
}

int rand2_int(Random rep) {
  if ( ! rep->last )      /* null pointer */
    printf( "call rand2_create before rand2_int!" );
  rep->last = ( rep->last * gc_mult + gc_add) % gc_max;
  return rep->last;
}

void rand2_destroy(Random rep) {
  free (rep);
}
```

A.2.4 Vererbung

Eine Klasse lässt sich durch Vererbung/Ableitung einfach erweitern. Die Deklaration erfolgt durch Angabe der Basisklasse nach einem Doppelpunkt und Zugriffsspezifikation nach dem Klassennamen. Die Initialisierung der Basisklasse erfolgt ebenfalls in der durch einen Doppelpunkt eingeleiteten Initialisierungssequenz im Konstruktor bevor dessen Funktionskörper betreten wird.

```
// rand4.h:

#ifndef RAND4_H
#define RAND4_H

#include "rand3.h"

struct DRandom: public Random {
public:
  // Konstruktor, Initialisierung der Basis nach :
  DRandom(int seed): Random(seed){};

  // 'inline' Elementfunktion für Zufallszahlen im Bereich 0..1
  double Double(){
    return Int()/static_cast<double>(m_max);
  }

  // Destruktor nicht erforderlich
};

#endif
```

A.2.5 Typunabhängige (Generische) Programmierung: Feldgrenzentests

```
// array.h:

#ifndef ARRAY_H
#define ARRAY_H

namespace sts {
```

```
struct BadRange{};

template<class TYPE, int N>
class Array {
public:
  Array(){};

  // [] access
  TYPE &operator[](int i);

  ~Array(){};
protected:
  TYPE  a[N];
};


template<class T, int N> inline T &
Array<T,N>::operator[](int i){
  // overload to obtain a safe []

  if( i < 0 || i >= N ) throw BadRange();
  return a[i];
}
}

#endif



//  main.cc:

#include ``array.h"

main(){

  sts::Array<double,10> a;
  a[9]  = 9.;
  a[10] = 10.;        // will terminate here due to bad range problem

}
```

A.3 Eine Datumsklasse

A.3.1 Schnittstellendefinition in `Date.h`

```
// date.h:

#ifndef DATE_H
#define DATE_H

#include <iostream>
#include <string>

namespace sts {

class Date {
public:
  // Konstruktor fuer Date aus Tag/Monat/Jahr Information
  Date(int day, int month, int year);
  // Kopierkonstruktor
  Date(const Date &);
  // Defaultkonstruktor
  Date();

  void Print(std::ostream &out = std::cout);
  Date Add(int number_of_days);
  int  Diff(const Date & other_date) const;
  void SetDate(int day, int month, int year);
```

```
  ~Date();

private:
  // number of days since 01.01.0001
  int days_since_0;
};

inline
std::ostream & operator<<(std::ostream &o, const Date &d){
  return d.Print(o);
}

}
#endif
```

A.3.2 Implementierung Date.cc

```
// Date.cc:

#include "Date.h"

namespace sts {

Date::Date(int day, int month, int year ){
  // error handling missing
  // implements a 360 day year of 12 months with 30 days each
  SetDate(day, month, year);
}

void Date::SetDate(int day, int month, int year ){
  // error handling omitted
  days_since_0 = (day-1) + ( month-1) * 30 + (year-1) * 360;
}

Date::Date(const Date &date){
  *this = date; // uses the builtin assignment
  std::cout << " -copy ctor called- ";
}

Date::Date()
: days_since_0(0)
{ }


void
Date::Print(ostream &o){
   int day, month, year;
   year  =  days_since_0 / 360 + 1;
   month = (days_since_0 - (year-1) * 360) / 30 + 1;
   day   =  days_since_0 - (year-1) * 360 - (month-1) * 30 + 1;

   o << day << "." << month << "." << year;
}

Date
Date::Add(int number_of_days){
  days_since_0 += number_of_days;
  return *this;
}

int
Date::Diff(const Date & other_date) const {
  return other_date.days_since_0 - days_since_0;
}

} // close namespace sts
```

A.3.3 Hauptprogramm main.cc

```
// main.cc:

#include "Date.h"

using sts::Date;
using std::cout;

int main(){
  Date  date(28, 9, 98);     // some date
  date.Print();              // printed
  cout << "\n";              // next line

  Date  date1(0, 13, 10);    // should give a problem but doesn't
  date1.Print(); cout << "\n";

  date.Print();
  date.Add(14);              // add 14 days to date
  date.Print();  cout << "\n";

  Date  date2(28,10,98);     // yet another date
  cout << "\ndifference from ";  date.Print();
  cout << " to ";  date2.Print();
                             // check difference; calls copy ctor implicitly
  cout << " is " << date.Diff(date2) << "\n";
}
```

Literatur

[1] *Programming languages - C++, ISO/IEC 14882*, ISO/ANSI, 1998.

[2] Andrei Alexandrescu, *Modern C++ Design - Generic Programming and Design Patterns Applied*, Addison Wesley, 2001.

[3] Grady Booch, *Object-Oriented Analysis and Design with Applications*, The Benjamin/Cummings Publication Company, 1994.

[4] Ulrich Breymann, *Designing Components with the C++ STL*, Addison Wesley Longman, 1998.

[5] David Flanagan, *Java in a Nutshell*, O'Reilly, 2nd edition, Köln, 1997.

[6] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Pattern*, Addison Wesley, 1995.

[7] Cay S. Horstmann, *Mastering C++*, An Introduction to C++ and Object-Oriented Programming For C and Pascal Programmers, 2nd edition, John Wiley & Son.

[8] Nicolai Josuttis, *Die C++ Standardbibliothek*, Addison Wesley, 1996.

[9] Nicolai Josuttis, *The C++ Standard Library*, 2nd edition, Addison Wesley, 1999.

[10] Brian W. Kernighan and Dennis M. Ritchie, *The C-Programming Language*, 2nd edition, Prentice Hall, 1988.

[11] Lippman, *C++ Primer*, 3rd edition, Addison-Wesley, 1998.

[12] Steven C. McConnel, *Code Complete: A Practical Handbook of Software Construction*, Microsoft Press.

[13] Scott Meyers, *Effektiv C++ programmieren*, 2. Auflage, Addison Wesley, 1995.

[14] Scott Meyers,*More Effective C++*, Addison Wesley, 1996.

[15] Scott Meyers,*Effective STL*, Addison Wesley, 2001.

[16] David Musser and Atul Saini, *STL Tutorial and Reference Guide*, Addison-Wesley, 1996.

[17] Robert Robson, *Using the STL*, Springer-Verlag, 1998.

[18] B. Stroustrup, *The C++ Programming Language*, 3rd edition, Addison Wesley, 1997.

[19] David Vandevoorde, Nicolai M. Josuttis *C++ Templates The Complete Guide*, Addison-Wesley, 2003.

[20] Todd Veldhuizen, *Template Metaprograms*, C++ Report, **7**(4), 36–43 (May 1995).

WWW

Im Netz findet man viele Quellen, die zu C++-spezifischen Fragen Auskunft geben.

Zuerst sind die usenet-Diskussionsgruppen zu nennen, die sich mit Fragen des Sprachstandards (`comp.std.c++`) oder Problemen mit vorhandenen Sprachfeatures (`comp.lang.c++.moderated`) auseinandersetzen. Bei `www.dejanews.com` findet man häufig bereits Antworten, ohne eine Frage geposted zu haben.

Darüberhinaus gibt es viele nützliche Adressen im Web, die html-Dokumente über spezielle Sprachaspekte oder Bibliotheken bereitstellen. Die folgenden Adressen waren im Frühjahr 2003 aktuell:

[21] `http://www.jamesd.demon.co.uk/csc/faq.html`
Administrativa und die FAQ der Gruppe `comp.std.c++`, wahrscheinlich auch in Zukunft zu erreichen über die Homepage von Matt Austern (`http://www.lafstern.org/matt/`), der an der Standardbibliothek und im Standardisierungskomittee mitgewirkt hat.

[22] `http://www.informatik.uni-konstanz.de/~kuehl/c++-faq`
Eine (aktuelle) Kopie der C++ FAQ von Marshall Cline

[23] `http://www.itga.com.au/~gnb/wp`
Eine Seite, die den gesamten Vorentwurf (committee draft 2) des C++-Standards bereitstellt. Der Standard selbst kann in elektronischer Form bei ANSI (`www.ansi.org`) als `pdf`-Dokument bezogen werden (etwa 20 EUR).

[24] `http://www.dinkumware.com/`
Referenzseiten zur Standardbibliothek (P. J. Plauger).

[25] `http://www.oonumerics.org`
Leitseite für Referenzen zur objektorientierten Numerik. Der Autor hat sich einen Namen auf dem Gebiet der Expression-Templates gemacht, die in seiner blitz++ Bibliothek zur linearen Algebra zum Einsatz kommen.

[26] `http://www.boost.org`
Source Code für Beispielimplementierungen vieler objektorientierter Konzepte mit hohem Wert für die Weiterbildung über Möglichkeiten von C++.

[27] `http://www.gotw.ca`
Herb Sutter's Sammlung von C++-Rätseln und Fragen, die jeweils (auch problematische) Aspekte der Syntax und Verwendung bestimmter Konzepte ansprechen. Empfehlenswert für jeden, der glaubt, die Programmierung in C++ durchdrungen zu haben.

[28] `http://www.research.att.com/~bs, http://www.cs.tamu.edu/people/faculty/bs`
Bjarne Stroustrups Homepage(s)

Index

Überladen, 66
    Auswahl der Funktion, 106
    Funktion, 33
    von Operatoren, 66
    von Template-Funktionen, 106

ableiten, 76
Ableitung, 77
algorithm, 130
Algorithmus, 121
Aliasing, 131, 169
Argument
    by value, 31, 168
    default, 32
    Feld, 172
    Referenz, 168
    variable Anzahl, 34
assert, 96
Assoziativität, 35
at, 141
Ausgabe, 41
Ausgabeoperator, 104
auto, 17
auto_ptr, 156

Bezugsrahmen, 37
Bibliothek, 173
Bitmanipulationsoperatoren, 36
Block, 17, 41
boolalpha, 44
break, 28, 30

case, 30
catch, 152, 153
class, 53
Coding Standards, 167
const, 17
    Elementfunktion, 55
Container, 121
    Adapter, 129
    Elementfunktionen, 127
continue, 28
critical section, 177
cv-qualifier, 18
cvs
    Kommandos, 163

Datenkapselung, 62
dec, 44
default, 30
Defaultargumente, 32
Defaultkonstruktor, 59
Deklaration
    Vorwärts-, 104
Deklarationsdatei, 13
delete ([]), 48
deque, 121
Destruktor, 59
    virtuell, 81
do while, 26

Eingabe, 41
Elementfunktion, 55
    Aufruf, 56
    const, 55
    get, 63
    inline, 56
    set, 63
    Template-, 109
        virtuell, 109
    virtuelle, 78, 80
Elementvariable
    static, Initialisierung, 96
Ellipsis, 35
EOF, 140
erase, 134
erben, 76
Exception
    Funktionsdeklaration mit, 155
Exceptions, 151
explicit, 73
Expression Template, 109
extern, 39

Fehlerbehandlung, 151
    assert, 96
Feld
    Realisierung mit Zeigern, 145

zweidimensionales, 144
Felder, 21
FIFO, 129
find, 133
fixed, 44
flags, 44
for, 26
friend, 64, 104
Funktion, 30
Überladen, 33
Argument, 31
by value, 31, 168
default, 32
Referenz, 168
variable Anzahl, 34
Definition, 30
Deklaration, 30
friend, 64
inline, 31, 50
rein virtuelle, 79
Template-, 93
virtuelle, 78, 80
virtuelle, Überladen, 84
Funktionsobjekt, 131
Funktor, 131

Garbage Collection, 185
goto, 28

hex, 44

if, 25
Initialisierung
Basisklasse, 80
Default-, 16, 59
im Konstruktor, 57
Klassen-, 57
Null-, 16, 59
POD-Struktur-, 60
static, 17, 96
inline, 31
Instanz, 54
internal, 44
iomanip, 45
iostream, 42
Elementfunktionen, 43
Formatierung, 43
Iterator, 123
bidirectional, 125
const, 124
Einfüge-, 135
Rückwärts-, 123, 125
random access, 125
Vorwärts-, 125
Iterator-Adaptoren, 137

Klasse, 7, 53
abstrakte, 79
friend, 65
Größe, 54
innere, 65
nicht ableitbare, 89
Template-, 95, 97
Kommandozeile, 140
Kommentare, 14
Konstruktor, 57
Default-, 57, 59
Initialisierung von Variablen, 57
Kopier-, 60, 61, 70
Kopieroperator
via operator=, 100

Label, 28
left, 44
LIFO, 129
list, 121
lock, 178
Lookup
König-, 43
lvalue, 65

main, 34
Argumente, 140
Manipulator, 44
map, 121
Marke, 28
Mehrfachvererbung, 87
memory leak, 49
Metaprogramme, 113
Methode, 7, 55
multimap, 121
multiset, 121
mutable, 17, 70
Mutex, 178

name mangling, 33
Namensraum, 37
namespace, 37
NDEBUG, 96
new ([]), 48
new [], 49
numeric limits, 105

Objekt, 54
Größe, 54
temporär, 65
oct, 44

Operator, 30, 35
  (), 36
    Überladen von, 131
  -¿, 54
  ., 54
  ::, 56
  ¡¡, 69, 104
    friend, 64
  =, 61, 100
  ¿¿, 69
  Überladen, 66
  als Elementfunktion, 68
  Assoziativität, 37
  Bindungsstärke, 37
  bitweise logischer, 35
  delete, 22
  Komma-, 35
  new, 22, 155
  Rückgabetyp, 67
  Scope-, 41, 56
  ternärer, 36
  Typumwandlung, 72
  Vergleichs-, 36
  Zuweisungs-, 61, 69, 70, 100
operator++, 71
operator++(int), 71
operator[], 146

Pointer, 19
pointer
  smart, 159
Polymorphismus, 81
priorityqueue, 130
private, 61
protected, 61, 87
Proxy-Objekt, 146
public, 61

queue, 129

rcs, 161
reentrant, 178
Referenzen, 23
register, 17
remove, 134
repository, 161
return, 31
reverse_iterator, 123
right, 44
rvalue, 65

scientific, 44
Scope, 37
set, 121
set_unexpected(), 155
setf, 44
setfill, 45
setprecision, 45
setw, 45
showbase, 44
showpoint, 44
showpos, 44
sizeof, 15
sizeof(), 54
skipws, 44
Slicing, 83
sort, 135
Speicherverwaltung, 48
  dynamische, 22, 99
Spezialisierung
  partielle, 102, 103
  vollständige, 101, 105
Sprungmarke, 28
Stack, 95
stack, 129
Standard Template Library, 121
Stapelspeicher, 129
static, 17, 39, 96
STL, 121
  Algorithmus, 121
  Container, 121
  safe, 128
string, 139
  find(), 56
  substr(), 56
stringstream, 52
struct, 53
Struktur
  Initialisierung, 60
switch, 29
Synchronisierung
  Thread-, 177

Template, 93, 109
  Basisklasse, 115
  Elementfunktion, 109
    virtuell, 109
  Fakultät, 109
  friend-Deklaration, 104
  Funktion, 93
  Klasse, 95
  Spezialisierung, 101, 115
    partielle, 102, 103
    vollständige, 101, 105
Template Rekursion, 109
terminate(), 155

this, 56
Thread, 174
    POSIX-, 174
threadsicher, 177
throw, 152, 155
Traits, 105
traits, 105
try, 152, 153
typedef, 24
typeid, 154
typeinfo, 154
typename, 94, 128
Typumwandlung, 72

unitbuf, 44
unsetf, 44
uppercase, 44
using, 38, 84
using-Direktive, 117

valarray, 143
Variable
    auto
        in Funktion, 17
    const, 17
    Element
        static, Initialisierung, 96
    globale, 17
        Deklaration von, 50
    Initialisierung
        static const, 106
    lokale, 41
    register, 17
    static
        in Funktion, 17
    volatile, 18
Variablenattribute, 17
vector, 121
Vererbung, 75, 88
    mehrfache, 87
    private, 77, 102
    protected, 77
    public, 76
    virtuelle, 88
virtual, 78
    bei Vererbung, 88
void, 24
volatile, 17, 176
Vorwärtsdeklaration, 104

Warteschlange, 129
while, 26

Zeiger, 19
    auf Templatefunktion, 130
    intelligenter, 159
    this, 56
Zuweisungsoperator, 61, 100