

Einführung in C++

Inhalt

1	Einführung.....	1-1
1.1	Datenorientierte oder hierarchische Softwareentwicklung.....	1-1
1.2	Objektorientierte Softwareentwicklung.....	1-1
1.3	Begriffe der objektorientierten Programmierung.....	1-2
1.3.1	Klasse und Objekt.....	1-3
1.3.2	Objektdefinition.....	1-4
1.3.2.1	Objektstatus.....	1-4
1.3.2.2	Objektverhalten.....	1-4
1.3.2.3	Objektidentität.....	1-5
1.3.3	Abstraktion.....	1-6
1.3.4	Kapselung.....	1-7
1.3.5	Modularisierung.....	1-8
1.3.6	Hierarchie.....	1-9
1.3.6.1	Allgemeine Definition.....	1-9
1.3.6.2	Vererbung.....	1-10
1.3.7	Typenprüfung.....	1-11
1.3.8	Gleichzeitigkeit.....	1-12
1.3.9	Objektspeicherung.....	1-13
1.4	Mischung der Konzepte.....	1-14
1.5	Konzepte der objektorientierten Sprache C++.....	1-15
1.5.1	Klassenbildung.....	1-16
1.5.2	Zugriffsrechte.....	1-16
1.5.3	Modulbildung.....	1-16
1.5.4	Klassenableitung, Vererbung.....	1-16
1.5.5	Aggregationen.....	1-17
1.5.6	Polymorphismus.....	1-17
1.5.7	Parametrisierbare Klassen.....	1-17
1.5.8	Exceptionhandling.....	1-17
2	Syntaktische Beschreibung von C++.....	2-1
2.1	Lexikalische Konventionen.....	2-1
2.1.1	Trennzeichen.....	2-1
2.1.2	Kommentare.....	2-1
2.1.3	Bezeichner.....	2-1
2.1.4	Schlüsselwörter.....	2-2
2.1.5	Literale.....	2-4
2.1.5.1	Integer-Literal.....	2-4
2.1.5.2	Character-Literal.....	2-5
2.1.5.3	Floating-Literal.....	2-5
2.1.5.4	String-Literal.....	2-6
2.1.5.5	Boolean-Literal.....	2-6
2.1.6	Operatoren.....	2-6
2.2	Datentypen.....	2-8

2.2.1	Grundlegende Datentypen	2-8
2.2.1.1	Ganzzahlige Datentypen	2-8
2.2.1.2	Gleitkommatypen	2-8
2.2.1.3	Leerer Datentyp	2-9
2.2.1.4	Boolscher Datentyp	2-9
2.2.1.5	Datentyp für erweiterten Zeichensatz	2-9
2.2.2	Abgeleitete Datentypen	2-9
2.2.2.1	Felder	2-9
2.2.2.2	Zeiger	2-10
2.2.2.3	Referenz	2-11
2.2.2.4	Zeichenketten (Strings)	2-12
2.2.3	Benutzerdefinierte Datentypen	2-14
2.2.3.1	Strukturen	2-14
2.2.3.2	Eigene Datentypen	2-14
2.2.3.3	Klassen	2-14
2.3	Definition und Deklaration von Variablen	2-16
2.3.1	Symbolische Konstanten	2-16
2.3.2	Dynamische Speichernutzung	2-16
2.4	Ausdrücke und Typenumwandlung	2-17
2.4.1	Schreibweise von Ausdrücken	2-17
2.4.2	Auflösung von Ausdrücken (Prioritäten)	2-18
2.4.3	Umwandlung von Standardtypen	2-19
2.4.4	Casting (explizite Typenumwandlung)	2-20
2.4.4.1	Funktionales Casting	2-20
2.4.4.2	Cast Notation	2-20
2.4.4.3	Casting Operatoren	2-21
2.5	Anweisungen	2-22
2.5.1	Selektionen	2-22
2.5.1.1	if-else	2-22
2.5.1.2	switch - case	2-23
2.5.2	Iterationen	2-24
2.5.2.1	while	2-24
2.5.2.2	do while	2-24
2.5.2.3	for	2-25
2.5.3	Sprunganweisungen	2-26
2.5.3.1	break	2-26
2.5.3.2	continue	2-26
2.5.3.3	return	2-26
2.5.3.4	goto	2-26
2.6	Funktionen	2-27
2.6.1	Aufbau und Wirkungsweise von Funktionen	2-27
2.6.1.1	Generelle Betrachtung	2-27
2.6.1.2	Syntax von Funktionen	2-27
2.6.1.3	Spezifizierer	2-28
2.6.1.4	Rückgabotyp	2-28
2.6.1.5	Argumente	2-28
2.6.1.6	Standardargumente	2-29
2.6.1.7	Zeiger auf Funktionen	2-30
2.6.1.8	Funktionsüberladung	2-31
2.6.1.9	Rekursive Funktionen	2-32
2.6.2	Standardfunktionen	2-32
2.6.3	Klassenbibliotheken (Foundation Classes)	2-32
2.7	Geltungsbereich und Lebensdauer	2-33
2.7.1	Spezifikation von Speicherklassen	2-33
2.8	Modularisierung	2-34
2.8.1	Trennung von Interface und Implementation	2-34
2.8.2	Einbindung von C Code in C++	2-35

2.8.3	Einsprungpunkt in ein C++ Programm	2-36
3	Klassen und Objekte	3-1
3.1	Was ist eine Klasse	3-1
3.2	Interface der Klasse	3-2
3.3	Implementation der Klasse	3-3
3.4	Instanzenbildung (Herstellung von Objekten)	3-3
3.4.1	Statische Instanziierung	3-3
3.4.2	Dynamische Instanziierung	3-4
3.5	Nutzung von Instanzen	3-5
3.5.1	Ansprechen statischer Objekte	3-5
3.5.2	Ansprechen dynamischer Objekte	3-6
3.5.3	Implizites Member-Argument (this)	3-6
3.6	Zugriffsrechte und Geltungsbereich	3-7
3.7	Static Daten-Member	3-9
3.7.1	Klassendeklaration mit einem static Daten-Member	3-9
3.7.2	Klassenimplementation mit einem static Daten-Member	3-9
3.8	Static Funktionen	3-10
3.9	Inline Funktionen	3-12
3.10	Operatoren für Klassen	3-13
3.11	Überladen von Funktions-Member (statische Polymorphie)	3-16
3.12	Spezielle Funktions-Member	3-17
3.12.1	Konstruktoren	3-17
3.12.2	Der Copy-Konstruktor	3-18
3.12.2.1	Verhalten des Standard-Copy-Konstruktor mit Daten-Member by value	3-19
3.12.2.2	Verhalten des Standard-Copy-Konstruktor mit Daten-Member by reference	3-19
3.12.2.3	Verhalten spezifischer Copy-Konstruktor: Daten-Member by reference	3-22
3.12.3	Der Zuweisungsoperator	3-25
3.12.4	Destruktoren	3-28
3.13	Friend Funktionen	3-29
4	Verhältnisse zwischen Klassen	4-1
4.1	Grundsätzliche Überlegungen	4-1
4.2	Vererbung	4-1
4.2.1	Ableiten von Klassen	4-2
4.2.2	Vererbung von Zugriffsrechten	4-4
4.2.3	Mehrfachvererbung	4-7
4.2.4	Funktionsüberschreibung und -überladungen in Vererbungen	4-9
4.2.4.1	Statische Funktionsüberladung in Vererbungen	4-9
4.2.4.2	Funktionsüberschreibung	4-9
4.2.4.3	Dynamische Funktionsüberladung in Vererbungen	4-10
4.2.5	Konstruktoren und Destruktoren abgeleiteter Klassen, Initialisierlisten	4-13
4.2.6	Abstrakte Klassen	4-14
4.2.7	Virtuelle Basisklassen	4-14
4.3	Assoziationen (Beziehungsstrukturen)	4-16
4.3.1	Einfache Beziehungen	4-16
4.3.2	Konditionelle Beziehungen	4-17
4.3.3	Komplex konditionelle Beziehungen	4-17
4.4	Aggregationsstrukturen (Containing Relationships)	4-18

4.5	Using (Verwendung)	4-20
5	Templates (Parametrisierte Klassen).....	5-1
5.1	Template Funktionen	5-1
5.1.1	Automatische Typenumwandlung bei Template-Funktionen	5-1
5.1.2	Überladen von Template-Funktionen.....	5-2
5.1.3	Lokale Variablen	5-2
5.1.4	Mehrere Typenparameter	5-2
5.2	Template-Klassen	5-3
5.2.1	Beispiel:	5-4
6	Ausnahmebehandlung	6-1
6.1	Fehlerbehandlung	6-1
6.2	Das Konzept	6-1
6.3	Schlüsselworte	6-2
6.4	Definition von Fehlerklassen	6-2
6.5	Auslösen einer Ausnahme	6-3
6.6	Auswerten einer Ausnahme	6-3
6.7	Weitergabe von Fehlern.....	6-4
6.8	Behandlung nicht aufgefangener Exceptions	6-4
6.9	Extensions für Schnittstellendeklarationen	6-4
7	Standardbibliothek	7-1
7.1	Konzepte der C++ Standardbibliothek	7-1
7.1.1	Namespaces	7-1
7.1.2	Definieren eines eigenen Namenraumes.....	7-1
7.1.3	Benutzen eines Namensraumes	7-2
7.1.4	namespace std.....	7-3
7.1.5	Header Dateien	7-3
7.1.6	Exceptions	7-4
7.2	Utilities	7-5
7.2.1	Vergleichsoperatoren	7-5
7.2.2	Hilfsfunktionen	7-5
7.2.2.1	min() und max() Funktionen.....	7-5
7.2.2.2	Vertauschen von Werten mit swap().....	7-6
7.2.2.3	cstdint und cstdlib Header	7-6
7.2.3	Der Datentyp pair	7-7
7.2.4	Auto-Pointer	7-8
7.2.5	Typ spezifische Definitionen (limits).....	7-9
7.3	Die Standard Template Library	7-10
7.3.1	Allgemeines Konzept	7-10
7.3.2	STL-Container.....	7-10
7.3.2.1	Sequentielle Container.....	7-12
7.3.2.2	Assoziative Container	7-14
7.3.3	STL-Iteratoren.....	7-19
7.3.4	STL-Algorithmen	7-20
7.3.4.1	Nicht modifizierende Algorithmen	7-20
7.3.4.2	Modifizierende Algorithmen	7-21
7.3.4.3	Löschende Algorithmen	7-21
7.3.4.4	Mutierende Algorithmen	7-21
7.3.4.5	Sortierende Algorithmen	7-21

7.3.4.6	Algorithmen für sortierte Bereiche	7-22
7.3.4.7	Numerische Algorithmen	7-22
7.4	Strings	7-23
7.4.1	Konstruktion, Kopieren und Destruktion von Strings	7-23
7.4.2	Nicht modifizierende Zugriffe	7-23
7.4.3	Modifizierende Zugriffe	7-24
7.4.4	Such-Funktionen	7-24
7.4.5	Vergleichsoperatoren	7-24
7.4.6	Direkter Zeichenzugriff	7-24
7.4.7	Teil- und Summen-Strings	7-25
7.4.8	Ein- und Ausgabe-Funktionen	7-25
7.5	Ein- und Ausgabe mit StreamKlassen	7-27
7.5.1	Allgemeine I/O Klassen und Objekte	7-27
7.5.2	Standardoperationen	7-28
7.5.3	Funktionen für die Eingabe	7-29
7.5.4	Funktionen für die Ausgabe	7-30
7.5.5	Streamstatus	7-30
7.5.5.1	Konstanten für den Zustand des Streams	7-30
7.5.5.2	Funktionen für die Kontrolle des Streams	7-30
7.5.6	Formatdefinitionen	7-31
7.5.6.1	Feldbreite, Füllzeichen und Ausrichtung	7-31
7.5.6.2	Vorzeichen und Grossbuchstaben	7-31
7.5.6.3	Zahlensysteme	7-33
7.5.6.4	Gleitkommanotation	7-33
7.5.6.5	Allgemeine Flags	7-35
7.5.6.6	Setzen und Abfragen von Flags	7-35
7.5.7	Dateizugriff	7-36
7.5.7.1	Klassen und Objekte für Filezugriff	7-36
7.5.7.2	Dateien öffnen	7-36
7.5.7.3	Wahlfreie Positionierung in einem File	7-37
7.5.7.4	Dateien lesen und schreiben	7-38
7.5.7.5	Dateien schliessen	7-38
7.5.7.6	Umleiten der Standardkanäle	7-38
7.6	Weitere Klassen	7-39
8	Anhang	8-1
8.1	Geschichtliche Entwicklung der Sprache C++	8-2
8.2	Literaturhinweise	8-3
8.3	Stichwortverzeichnis	8-4

Vorwort

C++ ist im wesentlichen eine Weiterentwicklung der Sprache C. In einer ersten Betrachtung erleichtert dieser Umstand dem C-Programmierer den Umstieg in die neue Sprache. In einer zweiten, genaueren Betrachtung stellen wir fest, dass der Umstieg meistens doch nicht so einfach ist. Dies weil ...

- ... die Analyse und der Design von objektorientierter Software gegenüber der herkömmlichen Softwareentwicklung ebenfalls überdenkt werden muss,

sowie ...

- ... die Runtime-Libraries eines bestimmten Zielsystems andere Funktionen aufweisen, und durch das, der Einarbeitungsaufwand nicht zu unterschätzen ist.

Somit ist der Aufwand für die Umstellung im Softwareentwicklungsbereich von C auf C++ grösser als zunächst angenommen!

Selbstverständlich gesellen sich zu den Unterschieden wie sie in den normierten Versionen von C++ festgehalten sind, analog der herkömmlichen C-Welt, die Unterschiede aufgrund verschiedener Compiler und verschiedener Herstellern.

Die Sprache C++ ist nach ANSI normiert (ANSI X3J16 Version 3). Als Grundlage diente dazu der C++ Standard der AT&T. Die Kursunterlage basiert auf den im Markt gängigsten C++ Compilern.

Als roter Faden wird im vorliegenden Kurs ein kleines Zeichnungsprogramm entwickelt. In der Kursunterlage "Einführung in C++" wird dabei vor allem auf die Aspekte der Datenhaltung und Speicherung eingegangen, während die Aspekte der Benutzerschnittstelle nur soweit betrachtet werden, wie Sie für das Testen des Codes absolut notwendig sind. Der Design einer Benutzerschnittstelle (Userinterface) und die Einbindung in eine moderne grafische Umgebung wird im Kurs "Softwareentwicklung für Windows" betrachtet.

Im Rahmen zahlreicher Kurse, die ich in den letzten Jahren im Bereich C++ und objektorientierte Softwareentwicklung erteilt habe, machte ich die Erfahrung, dass die alleinige syntaktische Schulung von C++ unbedingt von theoretischen Betrachtungen für den Aufbau von objektorientierten Systemen ergänzt werden sollte. Die optimalsten Lernfortschritte haben aus meiner Sicht diejenigen Personen erfahren, die die investierte Zeit in ca. drei gleich grosse Teile für Syntax, OOA/D und Übungen aufgeteilt haben.

Für die Übungen ist zudem zu bemerken, dass man sich vor allem am Anfang mit sehr kleinen Schritten begnügen sollte. Je klarer und einfacher das Übungsziel gesteckt wird, desto klarer sind auch die Lernfortschritte zu erkennen.

Rolf Wenger

1 Einführung

1.1 Datenorientierte oder hierarchische Softwareentwicklung

Die datenorientierte oder die hierarchische (top-down, bottom-up) Softwareentwicklung orientieren sich zunächst an den Objekten der Realität. Der Entwickler sucht zuerst die zu verarbeitenden Informationen und analysiert die Beziehungen zwischen den Informationen. Anschliessend beschreibt er Funktionalitäten zur Bearbeitung der Informationen (Design). Er wird dabei Gemeinsamkeiten in einem Softwaresystem suchen, damit er keinen, effizienten Code realisieren kann. Als Resultat erhält der Entwickler zwei Dinge:

- eine mehr oder weniger organisierte Datenhaltung
- ein Satz von Funktionen zum Bearbeiten der Daten

Für Analyse, Design und Konstruktion datenorientierter Softwaresysteme sind verschiedenste Verfahren (Methoden) und Tools entwickelt worden (Yourdon, Constantin, Jackson ...). Mit Hilfe dieser Methoden können kleinste bis grösste Softwaresysteme auf herkömmlichen prozess-, daten- und charakterorientierten DV-Systemen effizient hergestellt werden. Mit dem Einzug neuer Informationstechnologien wie die grafische Benutzeroberflächen, die integrierte Datenhaltung (Multimedia) oder verteilten Anwendungen (CORBA- oder DCOM-Technologie), kommen diese "herkömmlichen" Methoden jedoch zunehmend in Schwierigkeiten. Es fehlen den Methoden Elemente zur Darstellung und Handhabung von "Messaging", Informationskapselung und Abstraktion. Diese fehlenden Elemente werden bei der objektorientierten Softwareentwicklung berücksichtigt.

1.2 Objektorientierte Softwareentwicklung

Wie beim datenorientierten oder hierarchischen Vorgehen orientiert sich auch der Entwickler objektorientierter Softwaresysteme zunächst an den Objekten der Realität. Der Unterschied zum datenorientierten Vorgehen besteht darin, dass mit dem Erkennen der eigentlichen Objekte auch die Funktionen für diese Objekte von Beginn weg gesucht und definiert werden. Der Entwickler bildet neue, "informationstechnologische Objekte", die Daten und Funktionen beinhalten. Mit diesem Vorgehen erhält er:

- Prozessmodelle, die die Interaktion zwischen den verschiedenen Objekten beschreiben
- "Objektmodelle" (resp. Klassenmodelle), die die eigentlichen Objekte mit Ihren Informationen und Funktionen beschreiben.

Auch die objektorientierte Softwareentwicklung wird durch Methoden und Tools unterstützt (Booch, Coad/Yourdon...). Da die Objektorientiertheit an sich eine einfachere Abstraktion von komplexen Zusammenhängen zulässt, sind naturgemäss auch die Methoden zur Darstellung von objektorientierten Systemen für die Entwicklung von Softwaresystemen mit messageorientierter Verarbeitung (GUI-Applikationen) oder integrierter Datenhaltung besser geeignet als datenorientierte Darstellungsmethoden.

Wie bei allen Methoden zur Bearbeitung von Gegebenheiten, bildet auch die objektorientierte Technik keine Lösung für alle Probleme der Softwareentwicklung.

oder

Wer einen neuen Hammer geschenkt bekommt, sollte nicht meinen, dass ab dem Moment alle Probleme wie ein Nagel zu behandeln sind.

1.3 Begriffe der objektorientierten Programmierung

Um sich in der objektorientierten Programmierung (OOP) zurecht zu finden, ist es unerlässlich einige Begriffe einzuführen, sowie die Basiskonzepte der OOP zu beleuchten. Das nächste Kapitel widmet sich deshalb ausschliesslich diesem wichtigen Thema.

1.3.1 Klasse und Objekt

Der objektorientierte Entwickler unterscheidet zwischen Klasse und Objekt. Die Klasse definiert dabei die Abstraktion¹ eines Gegenstandes, einer Beziehung oder einer sonstigen Sache ohne reale Manifestation. Das Denken in Klassen erfordert vom Entwickler entsprechendes Abstraktions- und allgemeines Denkvermögen.

Der Begriff „Klasse“ ist dem Begriff „Typ“ ähnlich. Da der Begriff „Typ“ jedoch in der Programmierung bereits vorgelegt ist, und damit einfach die Beschreibung eines Datenformats gemeint wird, ist es besser, den Begriff in der OO-Technik nicht noch einmal zu überlagern.

Die Klasse definiert im Gegensatz zum Typ nicht nur Informationsinhalte sondern auch Funktionalität.



Abbildung 1-1 Eine Abstraktion (Klasse) kann letztendlich nur in Form eines realen Objektes wiedergegeben werden.

¹ Siehe auch Kapitel „Abstraktion“

1.3.2 Objektdefinition

Eine Klasse kann detailliert anhand eines realen Objektes derselben studiert werden. Für die Betrachtung von Objekten einer bestimmten Klasse unterscheiden wir deren statisches und dynamisches Verhalten, sowie ihre Identifikationen.

1.3.2.1 Objektstatus

Das statische Verhalten beschreibt den zum Zeitpunkt der Betrachtung vorhandene Dateninhalt. Dabei kann der Dateninhalt aus beliebigen Datenstrukturen bestehen. In der Literatur wird dazu auch oft der Begriff „Objektstatus“ verwendet. Die Informationen eines Objektes werden mit sogenannten Attributen ausgedrückt.

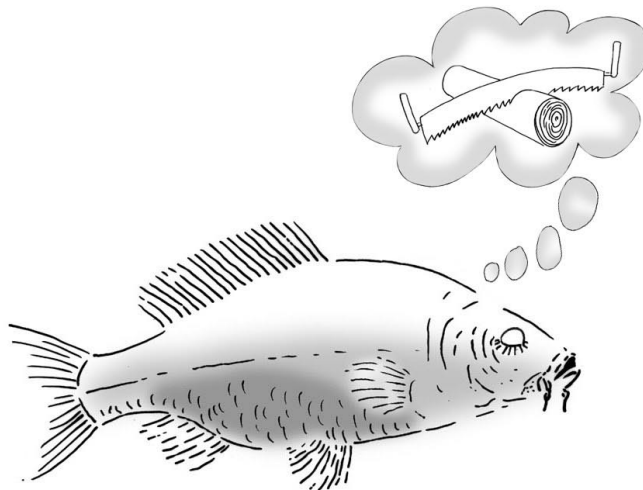


Abbildung 1-2 Ein Objekt verfügt zu jedem Zeitpunkt seines Lebenszyklus über einen Status. Der Status beschreibt den Informationsinhalt des Objektes.

1.3.2.2 Objektverhalten

Das dynamische Verhalten beschreibt die vorhandene Funktionalität eines Objektes. Ein Objekt im Sinn der Informatik verfügt im Regelfall über verschiedenste Funktionalitäten, von denen aber zu einem bestimmten Zeitpunkt nur eine aktiv ist (Singletasking). Objekte sind viel schwieriger zu beherrschen, wenn zu einem bestimmten Zeitpunkt gleichzeitig mehrere Funktionen aktiv sind (multitasking)².

Da reale Objekte in aller Regel viel komplizierter aufgebaut sind, erkennen wir, dass die Informatik einmal mehr nur eine kümmerliche Annäherung dessen ist, was sich in der realen Welt tatsächlich abspielt.

² Siehe auch Kapitel „Gleichzeitigkeit“

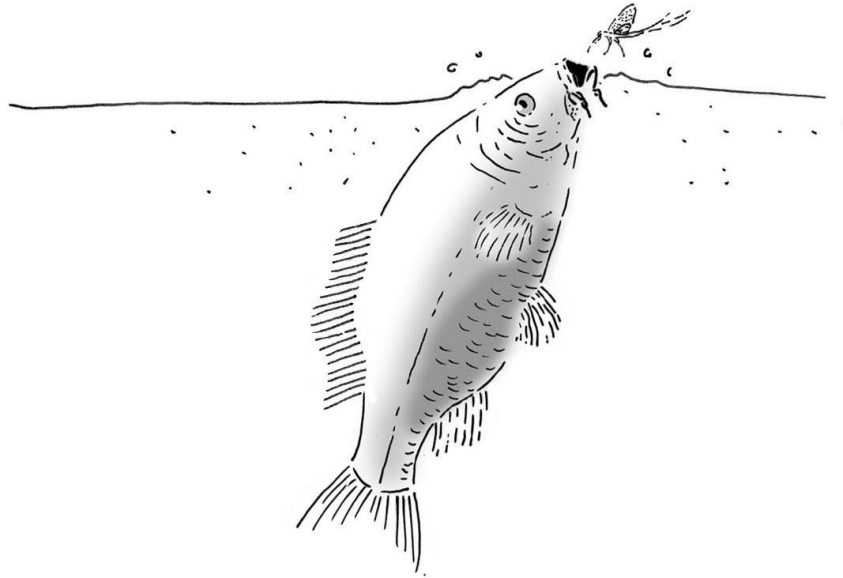


Abbildung 1-3 Ein Objekt führt zu jedem bestimmten Zeitpunkt eine oder mehrere Funktionen aus. Die Summe aller Funktionen beschreibt das Verhalten des Objektes.

1.3.2.3 Objektidentität

Es ist durchaus möglich, dass mehrere Objekte der gleichen Klasse mit dem selben Status existieren. Jedoch selbst dann muss es möglich sein, die einzelnen Objekte voneinander zu unterscheiden. Zu diesem Zweck benutzen wir die Identität eines Objektes. Die einzelnen Attribute der gleichen Objekte können den selben Inhalt haben und trotzdem sind die Objekte zu unterscheiden. In der Informatik führen wir zu diesem Zweck sehr oft eindeutige Nummerierungen. Diese entkoppeln die Erkennung der Identität von normalen beschreibenden Elementen.

In einer Datenbank für Personen verwenden wir kaum den Namen mit dem Vornamen zur Objektkennung, da es möglich ist, dass mehrere Personen mit dem selben Namen und Vornamen existieren.

Beim Design einer Klasse, sind den Möglichkeiten zur exakten Identifikation eines Objektes grösste Beachtung zu schenken. Es ist möglich, dass eine Klasse ohne Attribut für die Objektkennung definiert werden kann. Sollte sich jedoch später herausstellen, dass es trotzdem notwendig sein sollte, die besagte Klasse mit einer Objektkennung zu versehen, kann sich die Unterlassung zum Zeitpunkt des Design durch unverhältnismässig grossen Korrekturaufwand rächen.

Eine eindeutige Objektidentität kann nicht von einem Objekt auf ein anderes übertragen werden. Selbst wenn ein Objekt mit einer Identität X gelöscht wird und anschliessend ein neues Objekt mit der Identität X hergestellt würde, wäre die wahre Identität des neuen Objektes eine andere. Ob diese theoretische Betrachtung jedoch im Informatikalltag eine Rolle spielt, ist der Beurteilung des Entwicklerteams überlassen.

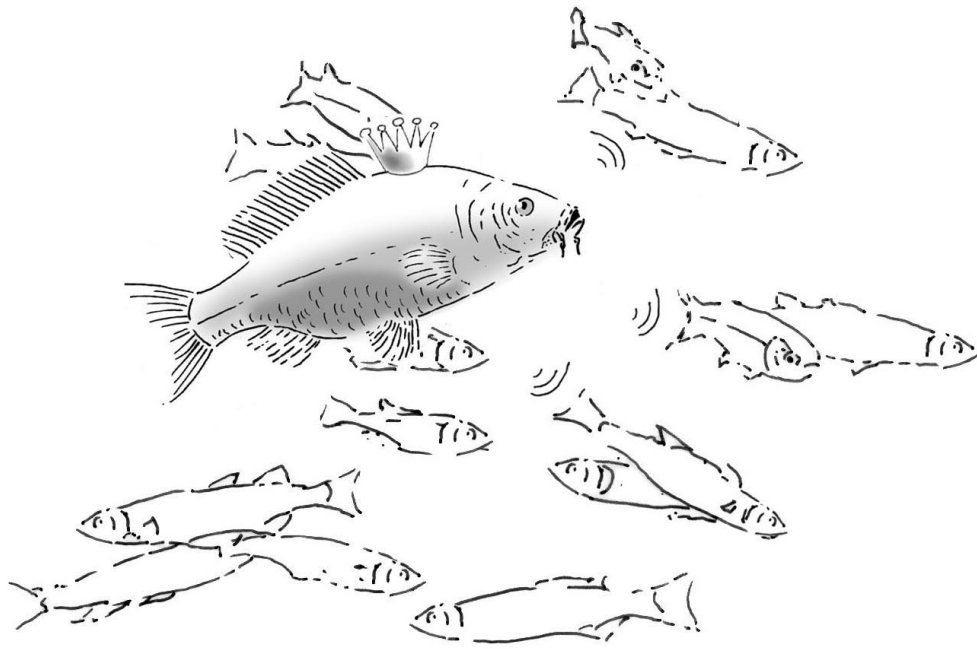


Abbildung 1-4 Ein Objekt verfügt über eine eindeutige Identität. Es ist in Realität nicht möglich die Identität eines Objektes zu wechseln.

1.3.3 Abstraktion

Wie wir bei der Begriffsdefinition im Kapitel „Klasse und Objekt“ bereits definiert haben, bildet die Abstraktion (engl. Abstraction) eine Abbildung der Realität, relativ zur Perspektive des Betrachters. Das heisst, dass eine Abstraktion durchaus verschieden interpretiert werden kann (und in Realität auch wird).

Die unterschiedliche Interpretation von Abstraktionen durch verschiedene Personen führt in einem Informatikprojekt immer zu Schwierigkeiten innerhalb des Projektteams. Diese Schwierigkeiten äussern sich in Form von nicht erwünschten Resultaten, gegenseitigem Unverständnis, Gesprächskonflikten oder gar nicht kompatibler Software. Es ist somit wichtig, dass die Abstraktion sehr genau definiert wird. Dazu gehört die Definition der statischen Informationen (Attribute) und des dynamischen Verhaltens (Funktionen oder Methoden).

Die richtige Abstraktion finden wir, indem wir immer wieder die Abstraktion mit den real abzubildenden Sachen vergleichen. Das heisst, dass beispielhafte Objekte von der gefundenen Abstraktion abgeleitet werden und betreffend ihrem Status und Verhalten geprüft werden.

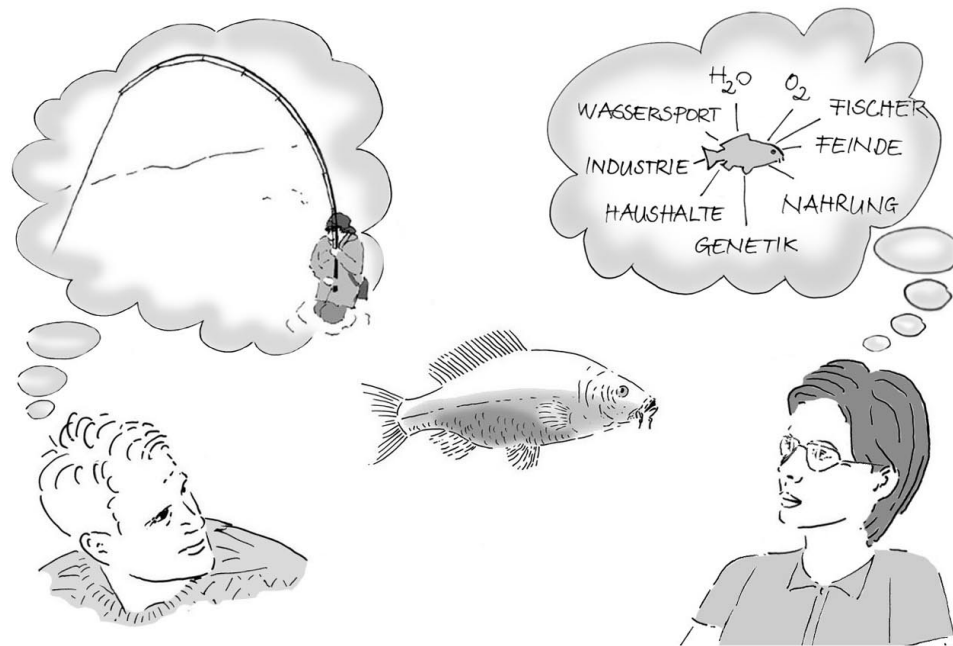


Abbildung 1-5 Eine Abstraktion beschreibt die Charakteristika einer realen Sache relativ zur Perspektive des Betrachters.

1.3.4 Kapselung

Ein Objekt kann intern viel komplizierter aufgebaut sein, als sich dies von aussen erkennen lässt. Diesen Umstand nennen wir Kapselung.

Mit der Kapselung (engl. Encapsulation) wird es möglich, interne Details gegen aussen zu schützen, respektive komplizierte Gebilde einfach erscheinen zu lassen. Der Hersteller des Objektes (eigentlich der Klasse) ist mit Vorteil bestrebt, die Kapselung vielfältig und feingliedrig auszugestalten. Dies fördert die vielzitierte Wiederverwendung des Codes, indem die Schnittstellen (die Ansicht von Aussen) nicht geändert werden muss, jedoch die Objekte über ganz andere Funktionalitäten verfügen können.

Die Kapselung kann soweit gehen, dass selbst einfachste Attribute nur von internen Funktionen angesprochen, resp. diese Attribute über sogenannte Schnittstellenfunktionen beeinflusst oder abgefragt werden können.

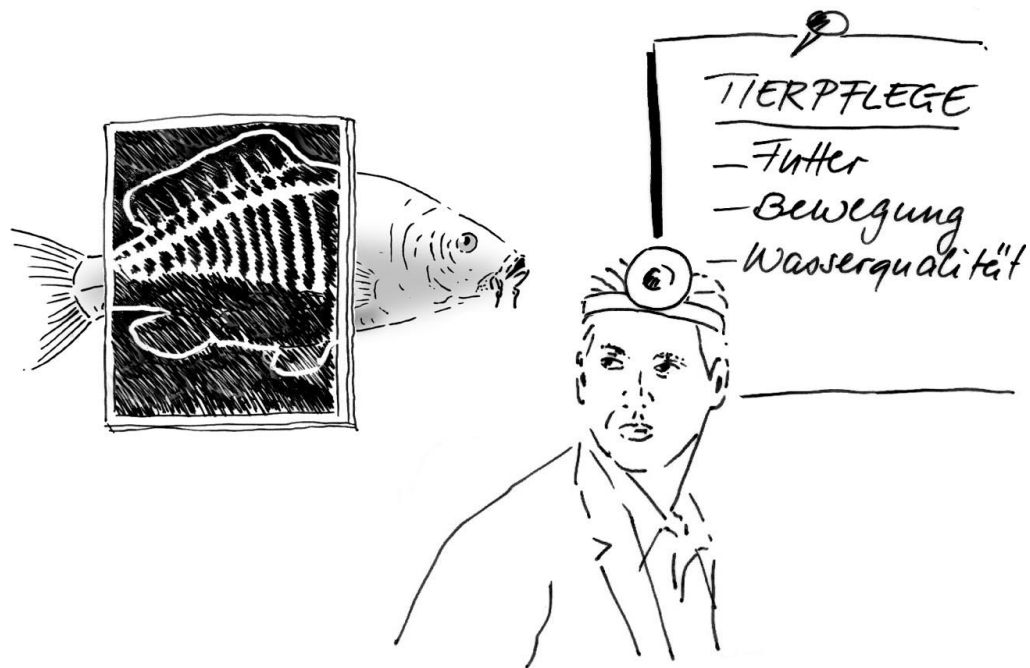


Abbildung 1-6 Die Kapselung beschreibt den inneren Aufbau der Objekte.

1.3.5 Modularisierung

Die Modularisierung (engl. Modularity) ist in der Informatikindustrie durchaus kein neues Konzept. Sie dient auch in der OOT zur Beschreibung der Aufteilung einzeln verwendbarer Bauteile. In Anlehnung an die Hardwareindustrie sprechen wir in Bezug auf Modularisierung auch von der Herstellung von Software-IC's. Im Gegensatz zur prozeduralen Softwaretechnik, welche die Modularisierung auf Funktionsstufe führt, definieren wir in der OOT die Modularisierung auf Klassen- resp. Objektstufe.

Die Modularisierung kennt verschiedene Arten der Implementation. Die einfachste Art ist die Modularisierung von Sourcecode. Hierbei geht es darum, dass die Wiederverwendung der einzelnen Module auf Sourcecodelevel statisch herbeigeführt wird. Diese Anwendung ist banal und alltäglich.

Die heute modernste Art der Modularisierung finden wir in der Client- / Server Komponententechnik, welche die Modularisierung soweit treibt, dass die einzelnen Module (Komponenten) unabhängig von einander fertig entwickelt werden, und zur Laufzeit über Grenzen eines Computersystems hinweg miteinander zusammenarbeiten können. Dabei ist es sogar möglich, dass die einzelnen involvierten Systeme auf unterschiedlichen Rechner- und Betriebssystemarchitekturen basieren.

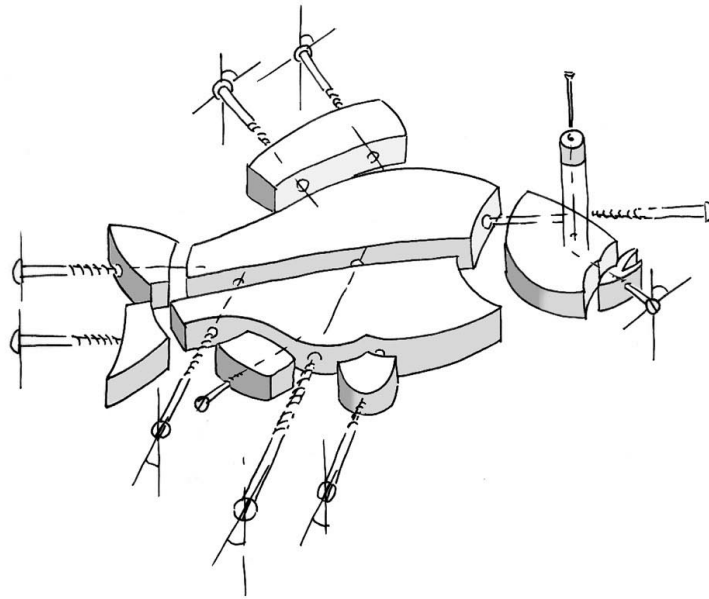


Abbildung 1-7 Die Modularisierung beschreibt die Aufteilung in einzeln verwendbare Bauteile.

1.3.6 Hierarchie

1.3.6.1 Allgemeine Definition

Die Hierarchie (engl. Hierarchy) beschreibt die Abhängigkeit der Klassen untereinander. Gemeinsamkeiten verschiedener Unterklassen werden in sogenannten Oberklassen zusammengefasst. Den Prozess der Begriffsbildung für eine Oberklasse zu einer oder mehreren Unterklassen nennen wir Spezialisierung. Der umgekehrte Prozess der Begriffsbildung von Unterklassen zu einer bestimmten Oberklasse nennen wir Generalisierung.

In der Regel verfügt eine Unterklasse über mehr Informationen und Funktionen als eine Oberklasse.

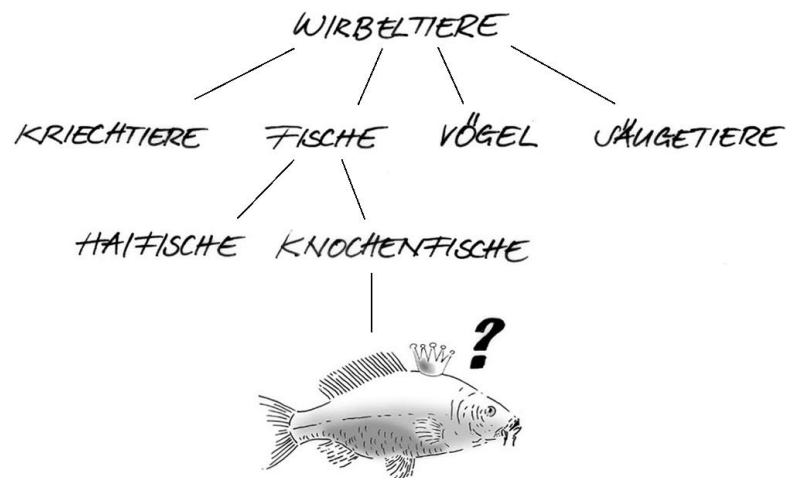


Abbildung 1-8 Die Hierarchie beschreibt die Abhängigkeit der Klassen untereinander.

1.3.6.2 Vererbung

Die Bildung von Hierarchien hat in der OOT die Vererbung (engl. Inheritance) zur Folge. Beim Prozess der Vererbung werden Informationen und Funktionen von der Oberklasse an die Unterklasse weitergegeben. Je nach Sprachimplementation ist es dabei möglich, die genauen Modalitäten der Vererbung zu beeinflussen. So ist es in C++ zum Beispiel möglich, neue Klassen durch Vererbung von einer oder mehreren Oberklassen herbeizuführen, oder die Weitergabe von Attributen und Funktionen an die Unterklasse zu beeinflussen.

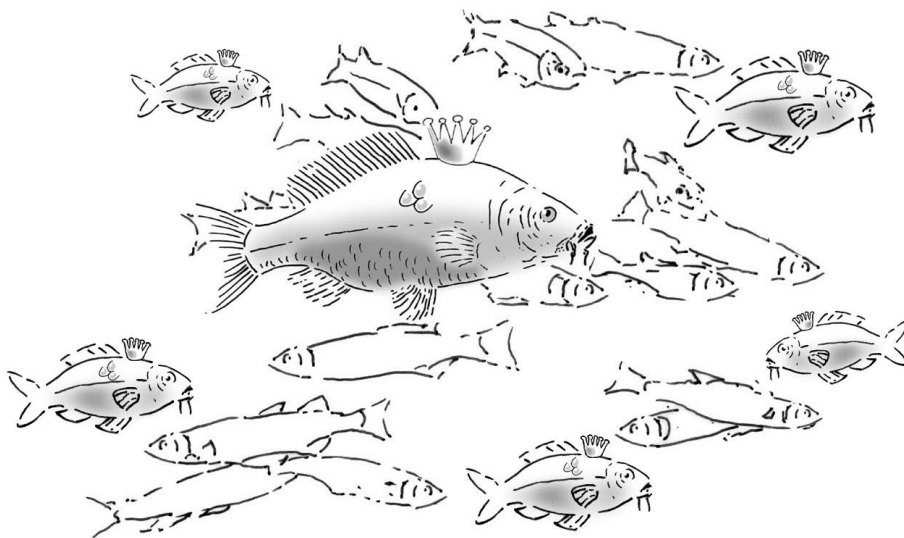


Abbildung 1-9 Die Vererbung beschreibt die Weitergabe von Status und Verhalten von Oberklasse(n) zu Unterklasse(n).

1.3.7 Typenprüfung

Die Typenprüfung (engl. Typing) verhindert die unbewusste Überführung eines Objektes einer Klasse in ein Objekt einer anderen Klasse. Der Ausspruch aus dem Volksmund „ein Fisch ist kein Vogel“ zeigt sehr typisch was hier gemeint ist.

Die unbewusste Typenüberführung birgt die Gefahr der falschen Behandlung des neuen Objektes (zum Beispiel Abfrage von Informationen die es nicht besitzt), und kann so zu echten betrieblichen Problemen führen (Softwareabstürze).

Was im Bereich der Typenüberführung erlaubt ist und was nicht, muss den jeweiligen Sprachimplementationen entnommen werden. In aller Regel sind Möglichkeiten der Objektüberführung innerhalb einer gültigen, definierten Hierarchie in Richtung Oberklasse möglich, jedoch nicht umgekehrt. Die Überführung der diskreten Informationstypen (Integer, Float ...) unterliegt meistens besonderen Regeln.

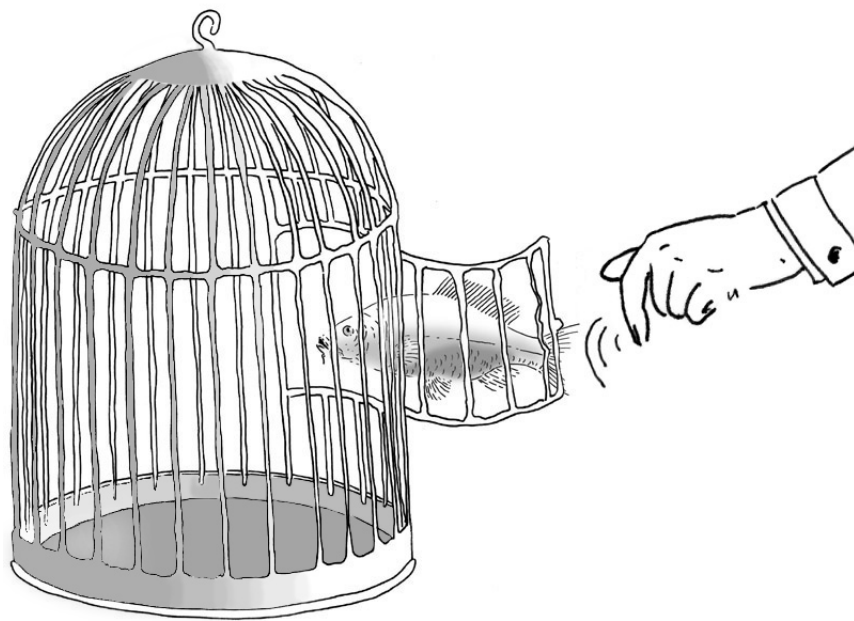


Abbildung 1-10 Die Typenprüfung verhindert die unbewusste Überführung einer Information vom einen Typ in den andern.

1.3.8 Gleichzeitigkeit

Die Gleichzeitigkeit (engl. Concurrency) definiert die Möglichkeit der Verrichtung mehrerer Aktionen zu einem bestimmten Zeitpunkt. Die OOT beinhaltet Möglichkeiten zum Entwurf von Gleichzeitigkeit. Die Möglichkeiten der Implementation der Gleichzeitigkeit ist jedoch wiederum abhängig von der verwendeten Sprache und dem verwendeten Betriebssystem.

Die Gleichzeitigkeit ist wohl das Gebiet der Informatikindustrie, welches die „kümmerlichste“ Umsetzung der Realität und gleichzeitig die grösste Herausforderung darstellt. Obschon die heutigen Computersysteme sehr ausgeklügelt sind und vor allem sehr schnell rechnen, findet die Implementierung der Gleichzeitigkeit immer wieder ihre Grenzen an der Kapazität und Komplexität der realen Welt.

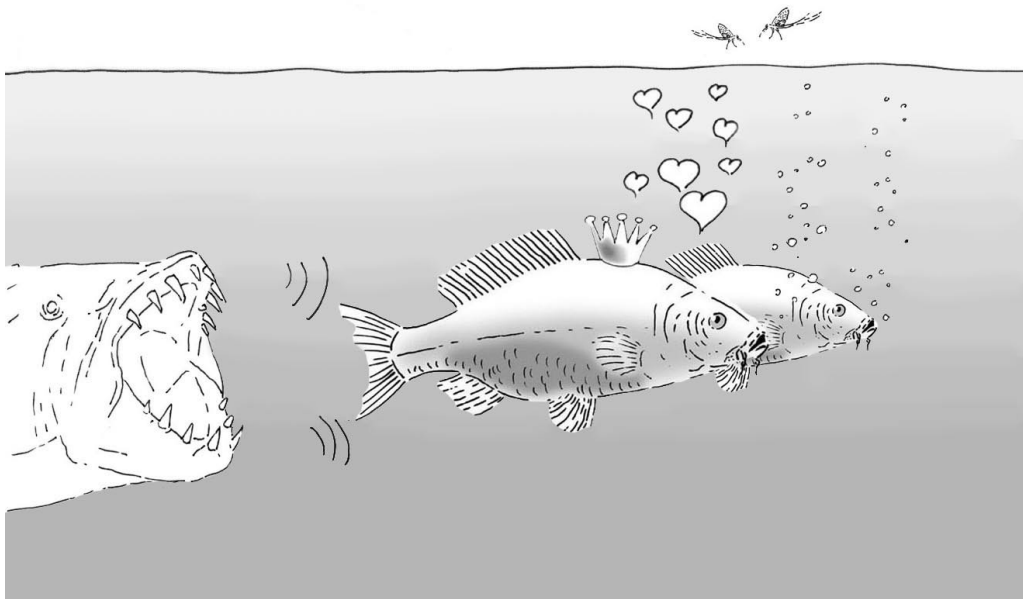


Abbildung 1-11 Die Gleichzeitigkeit beschreibt die Möglichkeit der Verrichtung von mehreren Aktionen zu einem bestimmten Zeitpunkt

1.3.9 Objektspeicherung

Die Objektspeicherung (engl. Persistenz) beschäftigt sich mit dem Umstand der Lagerung der Objekteigenschaften (Status und Identität) und der Beziehungen zwischen einzelnen Objekten.

Bemerkenswert ist in diesem Gebiet der OOT, insbesondere die Umsetzung der verschiedenen Algorithmen für die Speicherung von Beziehungen zwischen Objekten. So können zum Beispiel die Verknüpfungen von Objekten innerhalb des Arbeitsspeichers eines Rechners über Zeiger führen, währenddem die Verknüpfungen auf einem externen Datenträger über die Objektidentifikation geführt werden.

Ebenfalls wichtig ist die Erkenntnis, dass ein Objekt einmal auf einen externen Datenträger übertragen, als Objekt einer bestimmten Klasse wieder erkannt werden muss, da zur Laufzeit die Verbindung der Information auf dem Datenträger mit der Funktionalität über die Klassenangaben führen.

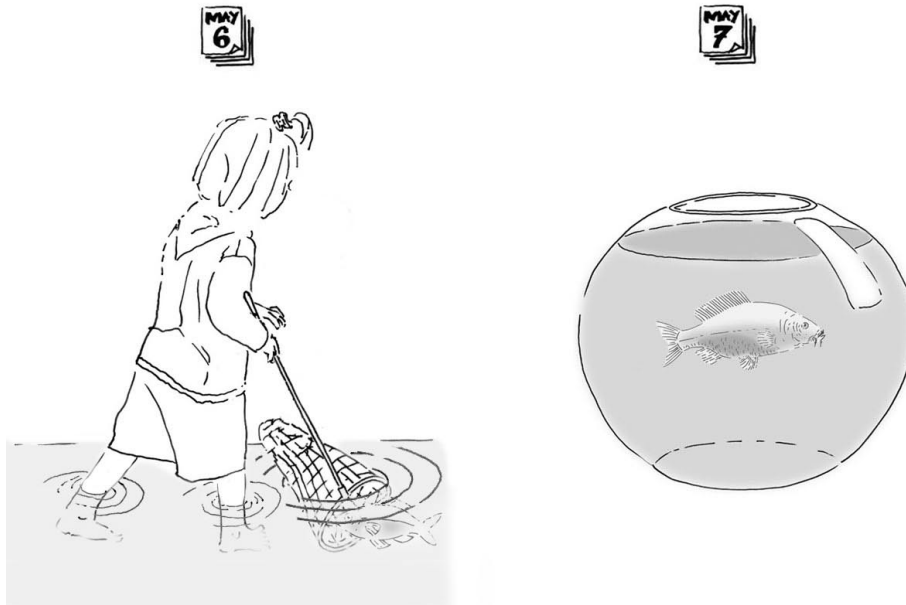


Abbildung 1-12 Die Persistenz beschäftigt sich mit der Erhaltung der Eigenschaften und Beziehungen der Objekte über Raum und Zeit.

1.4 Mischung der Konzepte

Die in diesem Kapitel dargestellten grundlegenden Konzepte werden in der Praxis natürlich nicht in sich geschlossen sondern nur in sich vernetzt angewendet. Dabei spielen einige Konzepte eine sogenannte Primärrolle, das heisst, dass sie in jedem OOT Projekt mit Sicherheit angewendet werden.

Dazu gehören:

- Abstraktion
- Kapselung
- Modularisierung
- Hierarchie

Die weitergehenden Konzepte ...

- Typenprüfung
- Gleichzeitigkeit und
- Objektspeicherung

... sind ergänzende Konzepte, die ein reales Projekt abrunden, jedoch nicht überall vorhanden sind.

Die Kunst der objektorientierten Analyse und des objektorientierten Design besteht nun darin, obige Konzepte auf eine reale Problemstellung anzuwenden und diese in einer „künstlichen“ Ausdruckswelt zu beschreiben, so dass der Programmierer anschliessend die Lösung in einer geeigneten Sprache wie C++ erstellen kann.

1.5 Konzepte der objektorientierten Sprache C++

Um die Orientierung beim Einstieg in die Programmiersprache C++ etwas zu erleichtern, betrachten wir zuerst einmal die Hauptgebiete, in die sich die Sprache einteilen lässt:

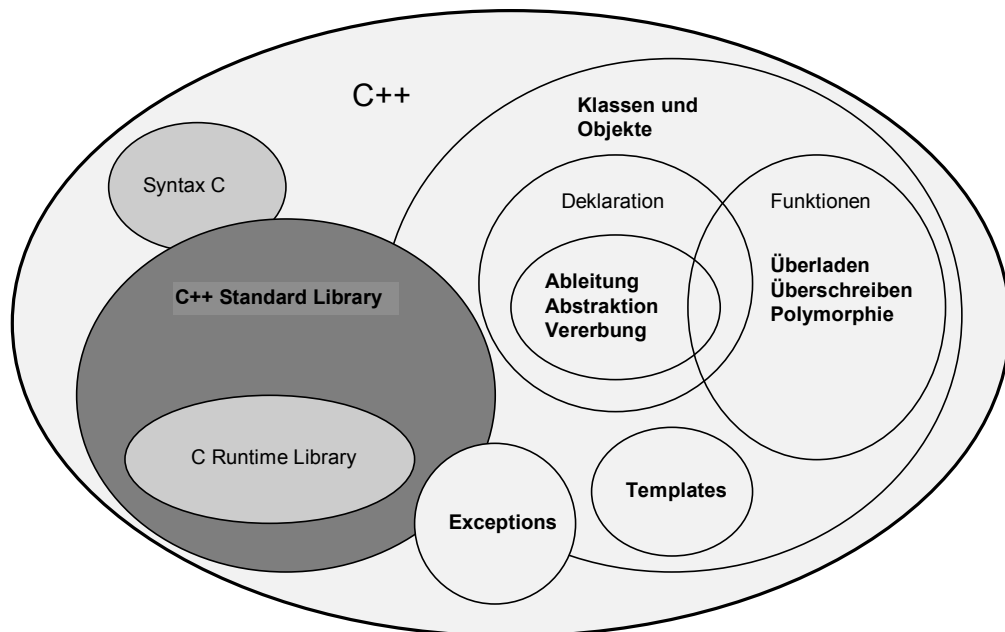


Abbildung 1-13 Menge der Wissensgebiete die im Umgang mit C++ beherrscht werden müssen.

C++ ist als Sprache eine Weiterentwicklung von C. Gemessen am Konzept der objektorientierten Programmierung unterstützt C++ nicht alle möglichen Modelldefinitionen. Doch sind die wesentlichen Kriterien erfüllt, so dass wir von einer "echten" objektorientierten Sprache reden dürfen.

Stufe	OOP Kriterium	C++ Unterstützung
MUSS	Abstraktion (Abstraction)	Klassenbildung
	Kapselung (Encapsulation)	Zugriffsrechte, Friend-Funktionen
	Modularität (Modularity)	Modulbildung
	Hierarchie (Hierarchy)	Klassenableitung (Vererbung), Aggregationen
WUNSCH	Typenprüfung (Typing)	Polymorphismus
		Parametrisierbare Klassen
	Gleichzeitigkeit (Concurrency)	- ³
	Speicherung über Raum und Zeit (Persistenz)	- ⁴
ZUSÄTZE	-	Exceptionhandling

³ Abhängig vom verwendeten Basissystem, Compiler und Bibliotheken

⁴ Muss manuell oder unter Zuhilfenahme eines OO-Subsystems (z.B. DB) implementiert werden

1.5.1 Klassenbildung

Die Klasse ist ein benutzerdefinierter Datentyp, der sich aus Datenmitgliedern (Variablen, Attributen) und Funktionsmitgliedern (Funktionen, Methoden) zusammensetzt. Währenddem die Datenmitglieder den Zustand eines Objektes beschreiben, definieren die Funktionsmitglieder das Verhalten eines Objektes.

Funktionsmitglieder dürfen unter bestimmten Umständen mehrfach definiert werden. Dies verfolgt den Zweck einer einfacheren Schnittstellenbildung. C++ spricht in diesem Fall von "function- and operatoroverloading".

Die Bildung von Klassen ist im wesentlichen ein designlastiges Unterfangen. Der Programmierer (hier als Codierer zu interpretieren) setzt eigentlich nur den Design in C++ Syntax um.

1.5.2 Zugriffsrechte

Innerhalb eines Objektes können alle Funktionsmitglieder auf alle Datenmitglieder des Objektes zugreifen. Hingegen können aussenstehende Objekte (selbst wenn sie von der selben Klasse abstammen) nur dann zugreifen, wenn die Klasse dies mit einer speziellen Zugriffsberechtigung vorgesehen hat. C++ implementiert drei verschiedene Stufen von Zugriffsberechtigung.

Datenmitglieder die normalerweise von aussen nicht zugänglich sind, können über speziell definierte Funktionen in bestimmten Fällen trotzdem erreicht werden (insofern vom Entwickler vorgesehen). C++ spricht in diesem Fall von "Friend Functions".

1.5.3 Modulbildung

Wie C unterstützt C++ das Konzept der Trennung von Interfacedefinition und Implementation. Die Interfaces werden im allgemeinen in *.h oder *.hxx Files abgelegt und als Source zur Verfügung gestellt. Die Implementationen wird gewöhnlich in *.cpp, *.cp oder *.c Files⁵ abgelegt.

Die Modulbildung erfolgt meist natürlich pro Klasse oder für mehrere gleichartige Klassen gemeinsam. Der Begriff Interface ist in diesem Zusammenhang so zu verstehen, dass es sich um eine Definition für C++ Module handelt. Diese Art von Interfaces kann von andern Sprachen wie Pascal oder VB nicht interpretiert werden und ist lediglich für die C++ Bindung vorgesehen.

1.5.4 Klassenableitung, Vererbung

Klassen können von Basisklassen einfach oder mehrfach abgeleitet werden. Dabei werden die Datenmitglieder und die Funktionsmitglieder der Superklassen an die Subklassen vererbt. Mit der Ableitung kann eine Zugriffsspezifikation angegeben werden. Dieses Prinzip ermöglicht die Kombination von Kapselung und Hierarchie.

In abgeleiteten Klassen dürfen Funktionen von Superklassen neu definiert werden. In diesem Fall wird beim Aufruf der Funktion in der Regel die lokal definierte Funktion verwendet. C++ spricht in diesem Fall von "function override"

1.5.5 Aggregationen

Objekte dürfen andere Objekte oder Zeiger auf andere Objekte beinhalten. Das Ansprechen der kaskadierten Klassen erfolgt transparent (gemäss Regeln C++).

Im Fall der Klassendeklaration innerhalb einer andern Klasse sprechen wir nicht von Aggregation sondern von Embedded Classes.

⁵ Abhängig vom verwendeten Compiler

1.5.6 Polymorphismus

Die Vieldeutigkeit tritt dann auf, wenn in einer Klasse mehrere Funktionen mit dem gleichen Funktionsnamen existieren (statische Polymorphie). Dynamische Polymorphie tritt dann auf, wenn zur Laufzeit ein Objekt über dessen Basisklasse durch Nutzung einer virtuellen Funktion angesprochen wird. C++ unterstützt beide Techniken.

1.5.7 Parametrisierbare Klassen

Mittels parametrisierbaren Klassen (Templates) unterstützt C++ eine erweiterte Typenprüfung. Dabei können Klassen zunächst typenunabhängig definiert werden. Erst bei der Verwendung einer solchen "typenlosen" Klasse muss dann der eigentliche Typ angegeben werden. C++ ist so in der Lage, zwei grundsätzlich gleichen Klassen, jedoch verschiedener Typen voneinander zu unterscheiden.

1.5.8 Exceptionhandling

Das exception handling bildet in C++ den Mechanismus zur Handhabung von Laufzeitfehlern. Das Konzept unterstützt das Auffangen von erwarteten und unerwarteten Fehlern zur Laufzeit eines Programms.

2 Syntaktische Beschreibung von C++

Dieses Kapitel stellt die Sprache C++ in der grundsätzlichen Schreibweise vor. Das Kapitel ist als Referenzkapitel ausgelegt und geht deshalb nicht auf die sprachlichen Zusammenhänge und Spezialitäten ein. Die genauen Erklärungen zu den speziellen OO-Techniken und die Verwendung der Konstrukte werden in den folgenden Kapiteln diskutiert. Die Beschreibung orientiert sich an der zur Zeit in Normierung begriffenen ANSI Version 3.0.

2.1 Lexikalische Konventionen

Ein C++ Programm besteht aus einem oder mehreren Files. Diese Files werden vom Compiler in mehreren Phasen übersetzt. Die erste Compilerphase bildet der Präprozessor. Er löst die #-Anweisungen und die Makrodefinitionen auf. Als Resultat entsteht aus dem Präprozessing eine Folge von Token. In einer weiteren Phase werden die Token aufgelöst. Dabei können folgende Typen von Token vorkommen:

- Trennzeichen (whitespaces)
- Kommentare (coments)
- Bezeichner (identifier)
- Schlüsselwörter (keywords)
- Literale (literals)
- Operatoren (operators)

2.1.1 Trennzeichen

Als Trennzeichen gelten Blanks, horizontale und vertikale Tabulatoren, Zeilenvorschübe, Zeilenendzeichen und Formfeed-Zeichen. Trennzeichen können im C++ Code beliebig vorkommen. Sie werden vom Compiler ignoriert.

2.1.2 Kommentare

Kommentare über mehrere Zeilen werden mit den Zeichen `/*` eingeleitet und mit den Zeichen `*/` beendet. Solche Kommentare können nicht geschachtelt werden. Kurzkommentierungen auf einer Zeile können mit `//` eingeleitet werden. Kurzkommentare gehen bis zum nächsten Zeilenende. Innerhalb eines mit `/*` eingeleiteten Kommentars werden die Zeichen `//` und `/*` ignoriert.

2.1.3 Bezeichner

Bezeichner werden für die programminterne Erkennung von Namen verwendet (Variablen, Funktionsnamen). Es sind Folgen von Buchstaben und Ziffern zugelassen, wobei das erste Zeichen ein Buchstabe sein muss (das Zeichen `_` gilt als Buchstabe). Bezeichner sind casesensitiv und beliebig lang. Sie dürfen zudem nicht exakt der Schreibweise eines Schlüsselwortes entsprechen.

2.1.4 Schlüsselwörter

Die folgenden Schlüsselwörter sind in C++ reserviert und dürfen nicht als Bezeichner benutzt werden¹:

¹ Die fett dargestellten Schlüsselwörter bilden gegenüber C eine Erweiterung oder verhalten sich anders.
Die Auflistung wurde dem X3J16 Draft Standard Stand April 95 entnommen

Schlüsselwort	Bedeutung / Verwendung
asm	Leitet Assemblercode im C++ Code ein (Inlining)
auto	Definition des Gültigkeitsbereiches
bool	Datengrundtyp
break	Verarbeitungsabbruch einer Mehrfachverzweigung oder einer Schleife
case	Einzelner Fall in einer Mehrfachverzweigung (switch - case)
catch	Exception Handling
char	Datengrundtyp
class	Datengrundtyp
const	Modifizierer
const_cast	Explizite Typenumwandlung
continue	Sprunganweisung in Schleifen zur Prüfung der Abbruchbedingung
default	Defaultelement in einer Mehrfachverzweigung (switch - case)
delete	Löschen dynamisch angelegter Objekte im Speicher
do	Schleifensteuerung
double	Datengrundtyp
dynamic_cast	Explizite Typenumwandlung
else	Alternativer Weg bei einfacher Fallverzweigung (if - else)
enum	Datengrundtyp
explicit	Verhindert implizite Typenumwandlung bei Konstruktoren
extern	Bezug auf Deklaration von Variablen und Funktionen
false	Boolsche Aussage „falsch“
float	Datengrundtyp
for	Schleifensteuerung
friend	Deklaration von Funktionen in Klassen
goto	Unbedingte Sprunganweisung (wenn möglich nicht verwenden)
if	Einfache Fallverzweigung
inline	Compileranweisung für Funktionssubstitution im Code
int	Datengrundtyp
long	Attribut eines Datengrundtypes
mutable	Übersteuerung von const Definitionen

Schlüsselwort	Bedeutung / Verwendung
namespace	Definieren von Gültigkeitsbereichen für Bezeichner
new	Dynamische Erzeugung eines Objektes im Speicher
operator	Überladung von Operatoren
private	Spezifizierung der Zugriffsrechte auf Elemente in Klassen
protected	Spezifizierung der Zugriffsrechte auf Elemente in Klassen
public	Spezifizierung der Zugriffsrechte auf Elemente in Klassen
register	Attribut eines Datengrundtypes
reinterpret_cast	Explizite Typenumwandlung
return	Funktionsbeendigung mit Wertrückgabe
short	Attribut eines Datengrundtypes
signed	Attribut eines Datengrundtypes
sizeof	Grösse eines Datentyps ermitteln
static	Definition des Gültigkeitsbereiches
static_cast	Explizite Typenumwandlung
struct	Datengrundtyp
switch	Leitet eine Mehrfachverzweigung ein
template	Parametrisierung von Funktionen und Klassen
this	Zugriff auf eine Elementfunktion
throw	Exception Handling
true	Boolsche Aussage „wahr“
try	Exception Handling
typedef	Definiert eigene Datentypen
typeid	Klassenidentifikation
union	Datengrundtyp
unsigned	Attribut eines Datengrundtypes
using	Definieren von Gültigkeitsbereichen (default) für Bezeichner
virtual	Überschreibbare Funktionen von Klassen
void	Datengrundtypes
volatile	Attribut eines Datengrundtypes
wchar_t	Datengrundtyp
while	Schleifensteuerung

Tabelle 2-1

C++ Schlüsselworte

Neben obigen Schlüsselwörtern sind Bezeichner, die mit doppeltem Unterstrich `__` oder einem einfachen Unterstrich und einem folgenden Grossbuchstaben beginnen reserviert. Sie werden für Funktionen in Standardbibliotheken benutzt.

Zusätzlich sind alle untenstehenden alternativen Repräsentationen von Operatoren in Form von Schlüsselwörtern reserviert:

Schlüsselwort	Normale Schreibweise
bitand	&
and	&&
bitor	
or	
xor	^
compl	~
and_eq	&=
or_eq	=
xor_eq	^=
not	!
not_eq	!=

Tabelle 2-2 C++ Schlüsselworte die aus alternativer Schreibweise für Operatoren entstehen. Achtung: Die Nutzung dieser Schlüsselwörter wird in Visual C++ nur mittels `#include <iso646.h>` unterstützt.

2.1.5 Literale

Literale sind besser unter dem Begriff Konstanten bekannt. Wir unterscheiden folgende Typen:

- Ganzzahlkonstanten (integer-literal)
- Zeichenkonstanten (character-literal)
- Fließkommakonstanten (floating-literal)
- Textkonstanten (string-literal)
- Booleankonstante (boolean-literal)

2.1.5.1 Integer-Literal

Integer-Literale werden zur Basis 8, 10 oder 16 definiert. Massgebend ist dabei die erste oder die ersten beiden Ziffern. Wird ein Integer-Literal mit 0 begonnen, wird damit ein oktaler Wert (Basis 8) definiert. Beginnt das Literal mit den Zeichen 0x oder 0X, wird ein hexadezimaler Wert (Basis 16) definiert. Alle anderen Werte gelten als dezimal (Basis 10).

```
1 #define FF_OCT    014
2 #define FF_DEZ    12
3 #define FF_HEX     0XC
```

Der jeweilige Integertyp des Literals kann zusätzlich mit den Suffix U oder u (unsigned) und L oder l (long) beeinflusst werden. Bei entsprechend grossen Werten wird der Typ u.U. automatisch angenommen.

```

1  #define LIT_NORMAL      256
2  #define LIT_LONG       256L
3  #define LIT_UNSIGNED   256U
4  #define LIT_ULONG      256UL
5  #define LIT_ULONG_AUTO 4294967256

```

2.1.5.2 Character-Literal

Ein Character-Literal besteht aus einem oder mehreren Textzeichen, die in Hochkomma eingeschlossen sind. Die Zeichencodierung wird vom Compiler in den Zeichensatz des Runtime-Systems übersetzt. Nicht darstellbare Zeichen (Steuerzeichen) können mittels untenstehender Codierung im Sourcecode eingefügt werden. Character-Literale können auch in Strings eingebettet verwendet werden.

Zeichen	Codierung
newline	<code>\n</code>
horizontal tab	<code>\t</code>
vertical tab	<code>\v</code>
backspace	<code>\b</code>
carriage return	<code>\r</code>
form feed	<code>\f</code>
alert	<code>\a</code>
backslash	<code>\\</code>
question mark	<code>\?</code>
single quote	<code>\'</code>
double quote	<code>\"</code>
octal number	<code>\ooo</code>
hexadecimal number	<code>\xhhh</code>

Tabelle 2-3 Characterliterals

2.1.5.3 Floating-Literal

Floating-Literale bestehen aus einem Integerteil, einem Dezimalpunkt und einem fraktionalen Teil. Hinzu kann ein vorzeichenbehafteter Exponent und ein Typsuffix definiert werden. Es ist zu beachten, dass entweder nur ein Ganzzahlteil oder nur ein Exponent, oder beides definiert werden kann. Hingegen ist es nicht zulässig, den Dezimalpunkt und den Exponenten wegzulassen. Der Typ eines Floating-Literals ist `double` solange kein Suffix definiert ist. Mit dem Suffix `f` oder `F` wird `float` und mit dem Suffix `l` oder `L` `long double` erzwungen.

```

6  #define FAKTOR_DOUBLE_1  256.
7  #define FAKTOR_DOUBLE_2  256.856
8  #define FAKTOR_DOUBLE_3  256.856E-64
9  #define FAKTOR_DOUBLE_4  -256.856E64
10 #define FAKTOR_LONG_DOUBLE 256.856L
11 #define FAKTOR_FLOAT_1    256.856F

```

2.1.5.4 String-Literal

String-Literale sind Folgen von Textzeichen. Sie werden vom Compiler als `array of character` mit Speicherklasse `static` erzeugt. Nebeneinander liegende Literale werden zu einem Literal zusammengefügt. Alle Literale werden mit ei-

nem '\0' Zeichen beendet. Damit ist sichergestellt, dass die String-Literale wie normale Strings gelesen werden können.

```
1 #define VERSION      "Version 2.00.000"
2 #define APP_NAME     "MyApplication"
3 #define SPEZ_TEXT_1  "First Line \nSecond Line\n"
4 #define SPEZ_TEXT_2  "This is a" " compound " ":\n" "B\n"
```

Jedoch Vorsicht:

Die Auswirkungen des Versuches ein String-Literal zu ändern sind nicht definiert.

2.1.5.5 Boolean-Literal

Die beiden Schlüsselworte `false` und `true` bilden die Boolean-Literale. Durch untenstehende Definition können die beiden Literale definiert werden, wenn sie in einer Compilerimplementation nicht vorhanden sind.

```
5 #define false        0
6 #define true         1
```

2.1.6 Operatoren

Operatoren dienen zur Verknüpfung von Ausdrücken. Nach Anwendungsgebiet werden folgende Operatoren unterscheiden:

Anwendungsgebiet	Bezeichnung	Operator	Beispiel	Alternative Schreibweise
Multiplikativ	Multiplikation	*	<code>a * b</code>	n.A. ¹⁾
	Division	/	<code>a / b</code>	n.A. ¹⁾
	Modulo (Rest)	%	<code>a % b</code>	n.A. ¹⁾
Additiv	Addition	+	<code>a + b</code>	n.A. ¹⁾
	Subtraktion	-	<code>a - b</code>	n.A. ¹⁾
	Inkrement 1	++	<code>a++</code>	n.A. ¹⁾
	Dekrement 1	--	<code>a--</code>	n.A. ¹⁾
Shift (Bitverschiebung)	Shift links	<<	<code>a << 1</code>	n.A. ¹⁾
	Shift rechts	>>	<code>b >> 1</code>	n.A. ¹⁾
Relational	Kleiner	<	<code>a < b</code>	n.A. ¹⁾
	Grösser	>	<code>a > b</code>	n.A. ¹⁾
	Grösser gleich	>=	<code>a >= b</code>	n.A. ¹⁾
	Kleiner gleich	<=	<code>a <= b</code>	n.A. ¹⁾

n.A. ¹⁾ keine alternative Schreibweise vorhanden

Anwendungsgebiet	Bezeichnung	Operator	Beispiel	Alternative Schreibweise
Vergleich	Gleich	==	a == b	n.A. ¹⁾
	Nicht gleich	!=	a != b	n.A. ¹⁾
Bitweise Verknüpfung	UND	&	a & b	n.A. ¹⁾
	ODER (inklusive)		a b	n.A. ¹⁾
	Exklusive ODER	^	a ^ b	n.A. ¹⁾
Logische Verknüpfung	UND	&&	a && b	n.A. ¹⁾
	ODER		a b	n.A. ¹⁾
Kondition	Kondition	? :	c ? a : b	n.A. ¹⁾
Zuweisung	Normale Zuweisung	=	a = b	n.A. ¹⁾
	Multiplikative Zuweisung	*=	a *= b	a=a * b
	Dividierende Zuweisung	/=	a /= b	a=a / b
	Restzuweisung	%=	a %= b	a=a % b
	Additive Zuweisung	+=	a += b	a=a + b
	Subtrahierende Zuweisung	-=	a -= b	a=a - b
	Shiftende Zuweisung rechts	>>=	a >>= b	a=a >> b
	Shiftende Zuweisung links	<<=	a <<= b	a=a << b
	Binäre Zuweisung UND	&=	a &= b	a=a & b
	Binäre Zuweisung ODER	=	a = b	a=a b
	Binäre Zuweisung exklusive ODER	^=	a ^= b	a=a ^ b
Elementzeiger	Dereferenzierung	->	ptr->e	(*ptr).e
	Separator	.	struct.e	n.A. ¹⁾
Trennungen	Kommaoperator	,	a,b	n.A. ¹⁾
Memorygrösse	-	sizeof()	sizeof(a)	n.A. ¹⁾
Speicherzuweisung	-	new	new int	n.A. ¹⁾
Speicher freigeben	-	delete	delete	n.A. ¹⁾

Tabelle 2-4 C++ Operatoren mit ihren alternativen Schreibweisen (n.A. ¹⁾ keine alternative Schreibweise vorhanden)

Operatoren sind hierarchisch geordnet. Dies spielt insbesondere bei Verknüpfungen von Operatoren und Ausdrücken eine besondere Rolle. Bestehen Unsicherheiten in der Auswertungsreihenfolge, sind die Ausdrücke durch Klammerung eindeutig zu schreiben. Die Klammerung hat zudem den Vorteil, dass die Lesbarkeit des Codes erhöht wird (siehe auch Kapitel "Auflösung von Ausdrücken (Prioritäten)").

2.2 Datentypen

2.2.1 Grundlegende Datentypen

2.2.1.1 Ganzzahlige Datentypen

Alle ganzzahligen Datentypen lassen sich explizit als signed bzw. unsigned deklarieren. Die Wertebereiche und die Kompatibilität der verschiedenen Typen sind abhängig von den entsprechenden Compilern.

Typ	Bezeichnung	möglicher Wertebereich
char	Zeichen	-128 ... 127
signed char	Zeichen	-128 ... 127
unsigned char	Zeichen	0 ... 255
short int	Integerwert	-32768 ... 32767
signed short int	Integerwert	-32768 ... 32767
unsigned short int	Integerwert	0 ... 65535
int	Integerwert	-2147483648 ... 2147483647
signed int	Integerwert	-2147483648 ... 2147483647
unsigned int	Integerwert	0 ... 4294967295
long int	Integerwert	-2147483648 ... 2147483647
signed long int	Integerwert	-2147483648 ... 2147483647
unsigned long int	Integerwert	0 ... 4294967295

Tabelle 2-5 Ganzzahlige Datentypen

Wenn die Attribute signed und unsigned nicht explizit definiert sind, wird signed angenommen.

2.2.1.2 Gleitkommatypen

Die verschiedenen Gleitkommatypen lauten:

Typ	Grösse [Bit]	Vorzeichen [Bit]	Exponent [Bit]	Mantisse [Bit]	Wertebereich	Präzision [Stellen]
float	32	1	8	23	$\pm 3,4 \cdot 10^{38}$	7
double	64	1	11	52	$\pm 1,7 \cdot 10^{308}$	15
long double	80	1	15	64	$\pm 1,2 \cdot 10^{4932}$	17

Tabelle 2-6 Gleitkommatypen

Die exakten, der jeweiligen Entwicklungsumgebung angepassten Grössen für obigen arithmetischen Datentypen, können den Dateien `<limits.h>` und `<float.h>` entnommen werden. Die darin befindlichen Definitionen von Konstanten können in Programmen natürlich auch für Auswertungen verwendet werden.

2.2.1.3 Leerer Datentyp

Der Datentyp **void** definiert einen leeren, allgemeinen Typ. Er dient dazu dem Compiler mitzuteilen, dass kein Wert in eine Funktion oder aus einer Funktion übergeben wird, oder dass Daten unbekannten Typs angesprochen werden sollen.

```
1 void MyFunction( void ); // Deklariert eine Funktion ohne
2                          // Argument und ohne Rückgabewert
3 void *MyPointer;        // Definiert einen Zeiger auf ein
4                          // Objekt unbekannten Typs
```

2.2.1.4 Boolescher Datentyp

Der Wertebereich vom Datentyp **bool** ist `true` oder `false`. Das Standardverhalten der booleschen Werte (automatische Typenumwandlung) entspricht demjenigen eines vorzeichenbehafteten Integerwertes (signed integer).

2.2.1.5 Datentyp für erweiterten Zeichensatz

Der Datentyp **wchar_t** unterstützt die erweiterten Zeichensätze. Dabei wird pro Zeichen mehr als ein Byte verwendet. Die effektive „Breite“ des Datentypes ist abhängig von der jeweiligen Implementation des Compilers (idR 16 Bit pro Zeichen).

In der Microsoft Entwicklungsumgebung existieren mehrere Zeichencodierungen:

- Standardcodierung (8 Bit)
- Multibyte Codierung (MBCS 16 Bit)
- Unicode Codierung (16 Bit)

Man beachte die Onlinehilfe der Entwicklungsumgebung für die Unterscheidung der Funktionalitäten.

2.2.2 Abgeleitete Datentypen

Aus den elementaren Datentypen (`char`, `int`, `double`...) können mit den Operatoren `[]`, `*` und `&` die Datentypen Feld, Zeiger und Referenz abgeleitet werden.

2.2.2.1 Felder

Felder dienen zur Aufnahme einer bestimmbar Anzahl Informationen vom gleichen Typ. Ein Feld wird wie folgt definiert:

Datentyp Feldname[Anzahl Elemente]

```
5 int  alZufallsZahl[ 250 ]; // Feld mit 250 Zufallszahlen
6 char achZeile[ 1000 ];    // Feld mit 1000 Zeichen
```

Wenn bei einer Felddefinition nur eine Anzahl angegeben wird, spricht man von einem eindimensionalen Feld. Bei zwei Grössenangaben spricht man von zweidimensionalen Feldern und bei `n` Grössenangaben von `n`-dimensionalen Feldern.

Die Summe der Elemente entspricht dem Produkt der Grössenangaben. Für die Berechnung des belegten Speicherplatzes muss die Summe der Elemente mit der Anzahl Byte des entsprechenden Datentypes multipliziert werden.

```
1 long int alZahlen[250][250]; // Feld mit 250 x 250 Elementen à
2                             // 4 Byte ( 250'000 Byte )
3 char  achZeile[50][1000];    // Feld mit 50 x 1000 Elementen à
4                             // 1 Byte ( 50'000 Byte )
```

Für die Bearbeitung eines bestimmten Elementes eines Feldes ist zu beachten, dass die Elementnummer nullbasierend ist. Das heisst, dass das erste Element mit der Indexnummer 0 angesprochen wird, das zweite Element mit der Indexnummer 1 usw.. Besonders beim Arbeiten mit Algorithmenliteratur ist dem Umstand grösste Beachtung zu schenken.

```

5  achZeile[ 0 ] = 'I'; // erstes Zeichen der Zeile ist I
6  achZeile[ 1 ] = 'n'; // zweites Zeichen der Zeile ist n
7  achZeile[ 2 ] = 'd'; // drittes Zeichen der Zeile ist d
8  achZeile[ 3 ] = 'e'; // viertes Zeichen der Zeile ist e
9  achZeile[ 4 ] = 'x'; // fünftes Zeichen der Zeile ist x
10 achZeile[ 5 ] = '\0'; // sechstes Zeichen der Zeile ist
11                      // ein "Nullzeichen"

```

2.2.2.2 Zeiger

Zeiger (engl. Pointer) dienen zur Aufnahme von Speicheradressen. Die Adressen dienen dazu, Positionen von Daten im Speicher festzuhalten. Über die Adressen kann auf die Daten zugegriffen werden.

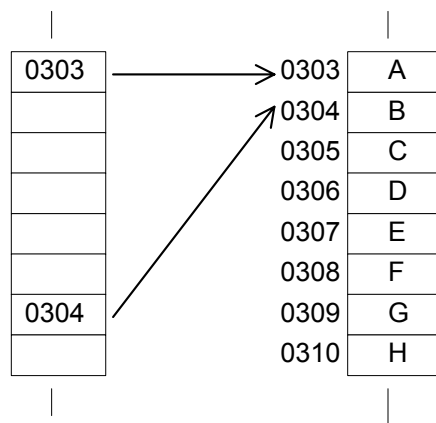


Abbildung 2-1 Schematische Darstellung der Wirkungsweise eines Zeigers

Ein Zeiger beansprucht im Memory nur soviel Platz wie die Maschine für die Darstellung einer Adresse benötigt (idR 4 Byte). Ein Zeiger wird wie folgt definiert:

Datentyp *Variablenname;

```

12 char    *pchPosition;           // Zeiger auf einen char
13 int     *piXKoordinate;         // Zeiger auf einen Integerwert

```

Bei einer Mehrfachdeklaration von Variablen mit einem Datentyp, muss der Operator * vor jedem Variablennamen geschrieben werden:

```

1  int  *p,  y;    // bedeutet int *p; int y;
2  int  *p, *y;    // bedeutet int *p; int *y;
3  int  p, *y;     // bedeutet int p; int *y;

```

Für die Arithmetik von Zeigern gelten spezielle Regeln. Grundsätzlich berechnet die sog. Zeigerarithmetik eine Einheit nicht als ein Byte, sondern als Anzahl Byte des Datentypes eines Zeigers.

```

4  int  *piZeiger;      // Deklaration des Zeigers
5  piZeiger = 3036;     // Zuweisung Speicheradresse 3036 in Zeiger
6  piZeiger++;          // Inkrementierung des Zeigers ergibt 3040
7                          // (auf einer 32 Bit Maschine)

```

... aber

```

8  char*pchZeiger;     // Deklaration des Zeigers
9  pchZeiger = 3036;   // Zuweisung der Speicheradresse 3036 in Zeiger
10 pchZeiger++;        // Inkrementierung des Zeigers ergibt 3037
11                      // (auch auf 32 Bit Maschinen)

```

2.2.2.3 Referenz

Als Referenz wird ein alternativer Name für ein Objekt bezeichnet. Über den alternativen Namen kann das ursprüngliche Objekt verändert werden. Dies innerhalb einer Funktion, aber auch über Funktionsgrenzen hinweg. Die Schreibweise `X&` bedeutet Referenz auf `X`.

```

12 int  x;              // Definition der Variablen x
13 int  i = 1;          // Definition und Initialisierung der Variablen i
14 int& r = i;          // Definition einer Referenz r auf i. Somit
15                      // zeigen r und i auf dasselbe integer Objekt
16 int  x = r;          // x = 1
17
18 ...
19
20 r = 2;               // i, r = 2, x bleibt 1 da x keine Referenz ist

```

Eine Referenz kann dazu dienen Funktionen zu implementieren, die den Wert der Argumente verändern soll. Im Gegensatz zu Pointerargumenten ist der Vorteil hier darin, dass innerhalb der Funktion keine Dereferenzierungen programmiert werden müssen.

```

21 // Implementation mit Pointer
22 // -----
23 void Increment( int *iValue ) { (*iValue)++; }
24 void MyFunction() {
25     int iLocalValue = 1;
26     Increment( &iLocalValue );          // iLocalValue = 2
27 }
28
29 // Implementation mit Referenz
30 // -----
31 void Increment( int& iValue ) { iValue++; }
32
33 void MyFunction() {
34     int iLocalValue = 1;
35     Increment( iLocalValue );           // iLocalValue = 2
36 }

```

2.2.2.4 Zeichenketten (Strings)

Zeichenketten werden in C++ als Array von Zeichen definiert. Dabei soll das letzte Zeichen in einem String immer das Nullzeichen sein (`'\0'`). Die Kennzeichnung mit dem Nullzeichen dient nicht nur den eigenen Funktionen zur Erkennung des Stringendes, sondern auch den Standard-Bibliotheksfunktionen. Bei der Verarbeitung von Strings in C++ Programmen muss Routine oder Vorsicht vorhanden sein. Es gibt für die korrekte Behandlung von Strings, resp. deren Längen, keine Compilerkontrollen. Das selbe gilt für die Bibliotheksfunktionen der Stringverarbeitung (siehe Kapitel 5). Letztere gehen davon aus, dass Strings plausibilisiert und mit entsprechenden Nullzeichen am Ende versehen sind.

In C++ sind für die Behandlung von Strings neben den bekannten C Runtimefunktionen für Stringhandhabung auch die String-Klassen zu verwenden (siehe Kapitel Stringklasse).

```
1 // Definition eines statischen Strings
2 char *String1 = "Ein String";
3
4 // Definition eines Strings ohne Speicherzuweisung
5 char *String2;
6
7 // Definition String mit Speicherzuweisung
8 char String3[255];
9
10 // weist die Anfangsadresse von String1 an String2 zu
11 String2 = String1;
12
13 // Weist einen String einem Zeichenarray zu
14 strcpy( String3, "Das ist ein neuer String");
15
16 // Dies ist ein Zeichen und kein String
17 char c = 'a';
18
19 // Dies ist nur ein Nullzeichen und kein String
20 char c = '\0';
```

Statische Strings erhalten vom Compiler automatisch ein Nullzeichen.

Zwischen Pointern und Arrays bestehen enge Zusammenhänge. Der Name eines Arrays kann nämlich gleichzeitig als Pointer auf das erste Arrayelement verwendet werden.

```
21 // Definition eines statischen Strings
22 char String[] = "Dies ist ein String";
23
24 // Definition eines Pointers auf das erste Element im String
25 char *pString = String;
26
27 // Gleiche Definition in anderer Schreibweise
28 char *pString = &String[ 0 ];
```

Pointer können mehrfach verschachtelt werden. Dabei ist der Auflösung und Zuweisung von Pointern grösste Beachtung zu schenken.

```

1 // Definition eines ersten statischen Strings
2 char String1[] = "String1";
3
4 // Definition eines zweiten statischen Strings
5 char String2[] = "String2";
6
7 // Definition Array von pointer auf Typ char mit Grösse 2
8 char *pszStringTable[ 2 ];
9
10 // pointer auf einen pointer auf char
11 char **apszString;
12
13 // Zuweisung Adresse des ersten Strings in Tabellenelement 0
14 pszStringTable[ 0 ] = String1;
15
16 // Zuweisung Adresse des zweiten Strings in Tabellenelement 1
17 pszStringTable[ 1 ] = String2;
18
19 // Zuweisung des ersten Eintrages der Stringtabelle
20 apszString = pszStringTable;
21
22 // Ausgabe des ersten Tabelleneintrages
23 printf( "String 1 direct over array of char : %s\n", String1 );
24 printf( "String 1 over array of pointer to char: %s\n", pszStringTable[0]);
25 printf( "String 1 over pointer to pointer to char: %s\n\n", *apszString );
26
27 // Ausgabe des zweiten Tabelleneintrages
28 printf( "String 2 direct over array of char : %s\n", String2 );
29 printf( "String 2 over array of pointer to char: %s\n", pszStringTable[1]);
30 printf( "String 2 over pointer to pointer to char: %s\n", *(++apszString));

```

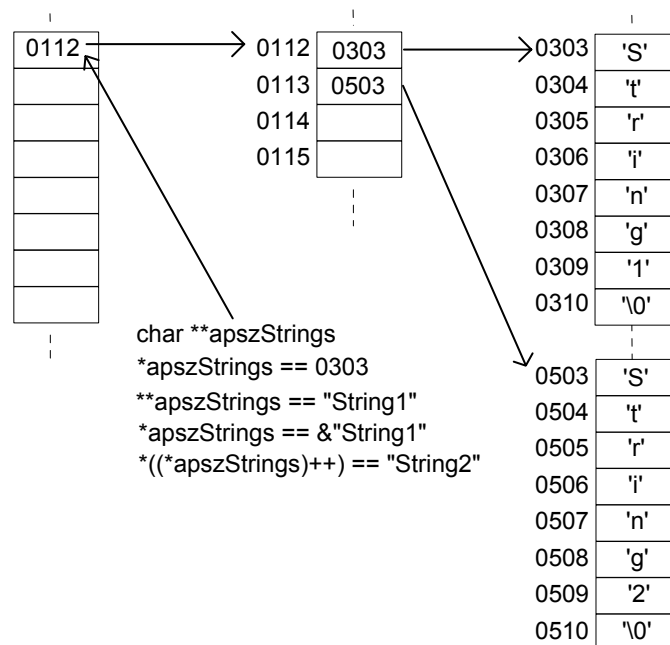


Abbildung 2-2 Schematische Darstellung von Zeiger auf Zeiger auf String

2.2.3 Benutzerdefinierte Datentypen

2.2.3.1 Strukturen

Strukturen definieren einen Datenverbund aus mehreren einzelnen Elementen gleicher oder verschiedener Typen.

```
struct tagname {      Typ Var;  
                      [Typ Var;]  
                      [...];  
                      Typ Var } Strukturname;
```

Der Zugriff auf ein Element erfolgt über den Struktur- und Elementnamen. Abhängig von der Definition einer Struktur wird zu diesem Zweck der Zugriff über eine Dereferenzierung oder einen absoluten Namen erfolgen.

```
1  struct tagDatum {      // Strukturdeklaration  
2      int tag;  
3      int monat;  
4      int jahr;  
5  };  
6  
7  tagDatum heute;        // Variablendefinition  
8  tagDatum *tag;         // Zeigerdefinition  
9  
10 heute.tag = 12;        // Wertzuweisung Element tag  
11 heute.monat = 10;      // Wertzuweisung Element monat  
12  
13 tag = &heute;          // Zeiger tag zeigt auf heute  
14 tag->jahr = 1994;       // Wertzuweisung über Dereferenzierung;  
15                        // Andere Schreibweise: (*tag).jahr = 1994
```

2.2.3.2 Eigene Datentypen

Mit Hilfe des Schlüsselwortes *typedef* können beliebige, eigene Datentypen geschaffen werden. Dabei lautet die allgemeine Syntax:

```
typedef  Definition  Neuer_Datentyp;  
  
16 typedef unsigned char BYTE;      // Typendeklaration Typ BYTE  
17 typedef struct {                 // Typendeklaration Typ POINT  
18     int iX_Coord;  
19     int iY_Coord;  
20 } POINT;  
21  
22 BYTE    byteSpezInfo;             // Variable vom Typ Byte  
23 POINT    ptUrsprung;              // Variable vom Typ POINT
```

2.2.3.3 Klassen

Klassen werden für die eigentliche objektorientierte Technik verwendet. Sie ermöglichen die Instanzierung von Objekten, die Abstraktion, die Kapselung, die Vererbung von Informationen und Funktionen sowie spezielle Handhabungen von Funktionen.

Im Gegensatz zu den bis jetzt betrachteten syntaktischen Elementen, vereinen Klassen sowohl Daten als auch Funktionen zu eigentlichen Objekten. Trotzdem gelten Klassen als benutzerdefinierter Datentyp.

Klassenschlüssel Bez : [Zugriffsspez.][Basisbez.],[Zugriffsspez.][Basisbez.][...] {

```

        [Zugriffsspez.:]
            [Datenmember;]
        [Zugriffsspez.:]
            [Funktionsmember;]
};

```

Klassenschlüssel sind:

- class
- struct
- union

Zugriffsspezifizierer sind:

- public
- protected
- private

Eine einfache Klasse:

```

1  // Deklaration einer Klasse CSWS_Point
2  class CSWS_Point {
3      int  iX_Coord;
4      int  iY_Coord;
5
6  public:
7      CSWS_Point();
8      CSWS_Point( int iX, int iY );
9      void SetPoint( int iX, int iY );
10 };

```

Eine abgeleitete Klasse:

```

11 // Deklaration einer abgeleiteten Klasse
12 class CMyWindow : public CWindow {
13 private:
14     int ixKoord;
15     int iyKoord;
16 public:
17     CMyWindow();
18     CMyWindow( int xKoord, int yKoord );
19     void SetWindowPos( int xKoord, int yKoord );
20 };

```

Die Schreibweise für die Implementation von Memberfunktionen kann dem Kapitel "Funktionen" entnommen werden.

2.3 Definition und Deklaration von Variablen

Bevor eine Variable in einem Programm verwendet werden kann, muss sie definiert werden. Bei der Definition wird sie mit einem frei wählbaren, zu diesem Zeitpunkt im Programm verfügbaren Datentyp, verknüpft.

[Speicherklasse] Typ Variablenname [Zuweisungsoperator Literal];

```

1      float      fResult;
2      int        iCounter;
3      long double ldExactResult = 0;
4      bool       bWindowIsOn = true;

```

In C++ müssen die Variablen nicht zwingend am Anfang eines Blockes deklariert werden. Grundsätzlich kann eine Variable immer dann deklariert werden wenn sie gerade gebraucht wird.

```
5      ...
6      // in C++ zulässig
7      for ( int iCounter = 0; iCounter > 25; iCounter++ ) {
8          ...;
9      }
```

Variablendeklarationen sind meistens auch Definitionen, da mit der Deklaration auch der Speicher alloziert wird. Nur bei Deklarationen mit dem Modifizierer "extern" wird kein Speicher alloziert. Variablen sollten immer durch den Programmierer initialisiert werden.

2.3.1 Symbolische Konstanten

Wird bei einer Variablendeklaration vor den Typspezifizierer das Schlüsselwort *const* vorangestellt wird die Deklaration nicht zu einer Variablen sondern zu einer symbolischen Konstante. Symbolische Konstanten besitzen gleich wie Variablen einen Speicherplatz. Auf Sie besteht jedoch ein eingeschränktes Zugriffsrecht (read only). Eine Deklaration einer symbolischen Konstante ist keine Definition wenn zusätzlich der Spezifizierer *extern* verwendet wird.

```
10 const double PI = 3.14159;      // Definition
11 extern const CRLF = "\r\n";    // Definition
12 extern const int MaxCount;      // Deklaration
13 const int MaxValue;             // Fehler, keine Initialisierung
```

Bei mehrfacher Deklaration für ein und dieselbe symbolische Konstante müssen die aufgeführten Typen jeweils identisch sein.

2.3.2 Dynamische Speichernutzung

Daten deren Grösse zum Zeitpunkt der Entwicklung nicht bekannt sind, werden mit Vorteil als dynamische Daten ausgelegt. Das heisst, dass zum Zeitpunkt der Programmentwicklung nur ein Variable vom Typ Pointer auf einen bestimmten Datentyp angelegt wird. Dieser Variable wird dann zur Laufzeit des Programms der wahre Pointer auf einen Datenblock zugewiesen. Für die dynamische Erstellung resp. Freigabe eines Datenblockes werden in C die Bibliotheksfunktionen *malloc()* und *free()* benutzt. C++ stellt für die selben Aufgaben die zwei Operatoren *new* und *delete* zur Verfügung.

Mit *new* wird dynamisch Speicher alloziert. Dabei verwendet *new* im Gegensatz zu *malloc()* nicht die Anzahl anzulegende Byte, sondern akzeptiert dynamisch den entsprechenden Typ. Mit ...

```
1 char *str = malloc( 20 );      // in C
2 char *str = new char[20];      // in C++, Array von 20 char
3 char *ch = new char;           // in C++ ein Objekt erzeugen
```

... wird zwar das selbe Resultat erzeugt, jedoch nur darum weil der Typ *char* ein Byte Platz beansprucht.

Untersuchen wir eine etwas komplizierte Struktur:

```
4 struct Node {
5     int    item;
6     Node  *left;
7     Node  *right;
8 };
9
10 ...
```

```

11 // in C
12 struct Node *node = ( struct Node * )malloc( sizeof( struct Node ) );
13
14 ...
15 // in C++
16 Node* node = new Node;

```

Der Zeigertyp der von new zurückgegeben entspricht dem konkreten Typ der angelegt wurde. Kann new keinen Block der geforderten Grösse anlegen, gibt der Operator 0 zurück. Gemäss Definition ist ein Zeiger des Types void* kompatibel mit jedem beliebigen Zeiger. Null (0) ein gültiger Zeiger, der ebenfalls mit jedem anderen Zeigertyp kompatibel ist. C++ garantiert, dass bei der Adresse 0 nie ein Objekt angelegt wird.

Zur Freigabe eines Speicherbereiches verwenden wir den Operator *delete*.

```

17 free( object );      // C
18 delete object;       // C++ für das Löschen einfacher Objekte
19 delete[] objects;    // für das Löschen von Arrays

```

Die Anwendung von delete auf einen Zeiger mit dem Wert 0 hat keine Wirkung.

2.4 Ausdrücke und Typenumwandlung

2.4.1 Schreibweise von Ausdrücken

Ein Ausdruck ist eine Verknüpfung von Literalen, Operatoren und Bezeichnern. Ausdrücke werden gebildet um Resultate zu errechnen. Syntaktisch werden Ausdrücke durch Interoperationszeichen (sog. Punctuators) getrennt. Das Strichpunkt bildet dabei das wichtigste Zeichen. Mehrere Ausdrücke werden mit geschweiften Klammern zu einem Block zusammengefasst. Parameter zu Schlüsselwörter und Bezeichner werden innerhalb von runden Klammern den Bezeichnern nachgestellt geschrieben.

```

1 // Ausdrücke werden durch Interoperationszeichen getrennt.
2 a = b * x * x;
3 y = x * g;
4 if ( y < a ) {
5     y = MyFunc( a, g );
6     cout << "y wurde über MyFunc berechnet und lautet:" << y << endl;
7 }
8 cout << "Ende der Berechnung" << endl;

```

2.4.2 Auflösung von Ausdrücken (Prioritäten)

Ausdrücke müssen u.U. in der richtigen Reihenfolge aufgelöst werden, weil sonst die Resultate falsch sein können.

```

9 // pro Memoria: wie lautet das Resultat untenstehender Rechnung
10 a = 3 + 4 * 5 - 2           // ist a = 33 oder a = 21

```

C++ löst Ausdrücke gemäss der untenstehenden Tabelle auf. Dabei besitzen Gruppen oben in der Tabelle eine höhere Priorität als Gruppen, die in der Tabelle unten angesiedelt sind. Kann ein Ausdruck mit der Nutzung der Prioritäten der Auswertung nicht richtig berechnet werden (z.B. eine mathematische Formel), muss der Programmierer C++ anweisen, wie die Reihenfolge der Auflösung zu geschehen hat. Zu diesem Zweck wird der Ausdruck durch setzen von Klammern ergänzt.

```

11 // Beispiel für die Nutzung zusätzlicher Klammerung
12
13 // Fläche eines Kreisringes berechnen
14 // Formel:  $A = \pi * (R^2 - r^2)$ 
15 Area = Pi * R * R - r * r;           // Falsch
16 Area = Pi * ( R * R - r * r );      // Richtig

```

Operator	Bedeutung	Auswertung
(..)	Klammerung von Ausdrücken	von rechts her
::	Globaler Geltungsbereich	von rechts her
.	Elementauswahl	von links her
->	Elementauswahl	von links her
[]	Indexoperator	von links her
()	Funktionsaufruf, Typenkonversion (funktionale Notation)	von links her
++ --	Postfix Inkrement und Dekrement	von rechts her
++ --	Präfix Inkrement und Dekrement	von rechts her
~	Bitweise Negation	von rechts her
!	Logische Negation	von rechts her
+ -	Vorzeichen	von rechts her
* &	Inhaltsoperator, Adressoperator	von rechts her

Fortsetzung siehe nächste Seite ...

Operator	Bedeutung	Auswertung
(type)	Typkonversion (casting Notation)	von rechts her
new delete	Dynamische Speicherverwaltung	von rechts her
sizeof	Grösse	von rechts her
.*	Komponente über Komponentenzeiger	von links her
->*	Komponente von Zeiger über Komponentenzeiger	von links her
* / %	Multiplikative Operatoren	von links her
+ -	Additive Operatoren	von links her
<< >>	Shift Operatoren	von links her
< <= > >=	Relationale Operatoren	von links her
== !=	Gleichheits Operatoren	von links her
&	Bitweises UND	von links her
^	Bitweises exklusiv-ODER	von links her
	Bitweises inklusiv-ODER	von links her
&&	Logisches UND	von links her
	Logisches ODER	von links her
?:	Konditional Operator	von rechts her
=	Zuweisungsoperator	von rechts her
*=	Zuweisung mit gleichzeitiger Multiplikation	von rechts her
/=	Zuweisung mit gleichzeitiger Division	von rechts her
%=	Zuweisung mit Rest nach ganzzahliger Division	von rechts her
+=	Zuweisung mit gleichzeitiger Addition	von rechts her
-=	Zuweisung mit gleichzeitiger Subtraktion	von rechts her
>>= <<=	Zuweisung mit gleichzeitiger Shift-Operation	von rechts her

&=	Zuweisung mit gleichzeitiger binärer UND Verknüpfung	von rechts her
^=	Zuweisung mit gleichzeitiger binärer exklusiv-ODER Verknüpfung	von rechts her
=	Zuweisung mit gleichzeitiger binärer inklusiv-ODER Verknüpfung	von rechts her
,	Kommaoperator, Trennung von Angaben	von links her

Tabelle 2-7 Reihenfolge der Auswertung von Ausdrücken.

2.4.3 Umwandlung von Standardtypen

Elementare Datentypen können in Ausdrücken frei verwendet und kombiniert werden. C++ konvertiert die Typen wenn immer möglich so, dass keine Informationen verloren gehen. Informationen können dann verloren gehen, wenn eine Zuweisung von einer Variablen in eine mit weniger Speicherplatz repräsentierte Variable erfolgt.

Bei Anwendung der meisten Operatoren erfolgt die Ermittlung von Wert und Typ einheitlich. Das von C++ verwendete Schema wird arithmetische Standardkonversion genannt. Diese Methode gleicht generell den "kleineren" Typen dem "grösseren" an. Resultate haben den grösseren Typ.

1. Ist einer der Operanden vom Typ long double, wird der andere nach long double konvertiert.
2. Andernfalls wird, sofern einer der Typen double ist, der andere in double konvertiert.
3. Andernfalls wird, sofern einer der Typen float ist, der andere nach float konvertiert.
4. Danach wird einer der Operanden in unsigned long konvertiert, wenn der andere Operator bereits den Typ unsigned long trägt.
5. Andernfalls wird ein unsigned int in einen long int konvertiert, wenn der andere Operand den Typen long int trägt und der Typ alle Werte eines unsigned int repräsentieren kann; andernfalls werden beide Operanden zu unsigned long int konvertiert.
6. Andernfalls wird ein Operand zu long konvertiert, wenn der andere Operand bereits den Typ long trägt.
7. Andernfalls wird ein Operand zu unsigned konvertiert, wenn der andere Operand bereits den Typ unsigned trägt.
8. Andernfalls sind beide Operanden vom Typ int.

2.4.4 Casting (explizite Typenumwandlung)

Casting bedeutet explizite Typenumwandlung. Da C++ strenge Typenprüfung aufweist, ist das Casting oft notwendig um Typenumwandlungen gezielt durchzuführen. Dazu kann das Casting auch benutzt werden um Standardumwandlungen von Datentypen zu verändern, oder einfach um sicherzustellen, welche der Standardumwandlung verwendet werden soll (Übersicht).

Syntaktisch existieren in C++ mehrere Alternativen für das Casting:

2.4.4.1 Funktionales Casting

Das funktionelle Casting ermöglicht eine Typenumwandlung mit einem oder mehreren Argumenten.

NewType(OldTyp)

```

1  double( 32 )           // nur ein Beispiel
2  int( 'a' - 'z' + 1 )   // auch nur ein Beispiel

```

2.4.4.2 Cast Notation

Die sogenannte Cast Notation erlaubt die Umwandlung eines Argumentes in einen angegebenen einfachen oder zusammengesetzten Datentyp.

(NewType) OldTyp

```

3  (double) 32           // nur ein Beispiel
4  (int)( 'a' - 'z' + 1 ) // auch nur ein Beispiel

```

2.4.4.3 Casting Operatoren

Für die Typenumwandlung von Objekten oder fundamentalen Datentypen existieren in C++ spezielle Operatoren:

- `dynamic_cast`
- `static_cast`
- `reinterpret_cast`
- `const_cast`

Der "edelste" der oben aufgeführten Operatoren ist der `dynamic_cast`. Mit Hilfe dessen ist eine Umwandlung mit Typenprüfung zur Laufzeit möglich. Das heisst, dass ein Typ nur dann umgewandelt wird, wenn auch entsprechend eine gültige Umwandlung möglich ist.

```

5  class A {};
6  class B : public A {};
7
8  ...
9      A *pA = new A;
10     B *pB = new B;
11
12     pA = dynamic_cast< A *>(pB);

```

Bei ungültiger Umwandlung zur Laufzeit wird entweder ein Null-Zeiger zurückgegeben oder eine `casting exception` ausgelöst.

Der `static_cast` Operator definiert eine Typenumwandlung für den Compiler. Er kann mit dem funktionellen Casting verglichen werden. Im Gegensatz zum `dynamic_cast` ist der `static_cast` nicht typsicher.

Ebenso sind die beiden letzten Casts nur in besonderen Fällen anzuwenden, da sie auch unsichere Umwandlungen durchführen. Der `reinterpret_cast` ermöglicht eine Umwandlung von einem beliebigen Typ in einen andern beliebigen Typ und der `const_cast` ermöglicht die Aufhebung von `const` definierten Objekten.

2.5 Anweisungen

2.5.1 Selektionen

2.5.1.1 if-else

Mit den if-else Anweisungen werden Entscheidungen formuliert.

Syntax: **if** (*expression*)
 statement 1;
 [else
 statement 2;]

oder: **if** (*expression*) {
 statement 1;
 statement 2;
 :
 statement n;
 }
 [
 else {
 statement 1
 statement 2;
 :
 statement n;
 }
]

Bedingungen: *expression* kann arithmetischer Typ, pointer Typ oder Objekt einer definierten Klasse sein.

Funktionsweise: Zuerst wird *expression* berechnet. Ergibt die Berechnung einen von 0 verschiedenen Wert (TRUE) wird *statement 1*, ansonsten *statement 2* ausgeführt (FALSE).

Bemerkungen: Bei verschachtelten if-else Anweisungen ist zu berücksichtigen, dass ein else-Teil immer mit dem letzten if-Teil verbunden wird, dem kein else-Teil zugeordnet ist. Klammern hilft auch hier die Übersicht zu erlangen.

2.5.1.2 switch - case

Die switch - case Anweisung ermöglicht eine Auswahl unter mehreren Alternativen.

Syntax:

```
switch ( expression ) {  
    case const-expression:  
        statement 1;  
        [statement 2;  
        :  
        statement n;  
        [break;]  
  
    case const-expression:  
        statement 1;  
        [statement 2;  
        :  
        statement n;  
        [break;]  
  
    ...  
  
    [default:  
        statements;  
    ]  
}
```

- Bedingungen: *expression* kann ein arithmetischer Typ oder ein Objekt einer definierten Klasse sein. Jedes case-Label muss eindeutig sein. Der Ausdruck default darf nur einmal vorkommen.
- Funktionsweise: Zuerst wird *expression* ausgewertet. Das Resultat wird mit den *const-expression* der case verglichen. Bei Gleichheit werden die *statements* hinter dem entsprechenden case bis zum nächsten break oder bis zum Ende der switch Anweisung ausgeführt.
- Bemerkungen: Switch-Anweisungen können geschachtelt werden. Verschiedene case mit gleicher Behandlung können hintereinander geschrieben werden.

```
1  switch( i ) {  
2      case 1:  
3          // Anweisungen  
4          break;  
5  
6      case 2:  
7      case 3:  
8      case 4:  
9          // Anweisungen  
10         break;  
11  
12     case 5:  
13         // Anweisungen  
14         break;  
15  
16     default:  
17         // Anweisungen  
18 }
```

2.5.2 Iterationen

2.5.2.1 while

While erlaubt eine Iteration mit Prüfung einer Anfangsbedingung.

Syntax: **while** (*expression*) *statement*;

oder: **while** (*expression*) {
 statement 1;
 [*statement 2*;
 :
 statement n;]
}

Bedingungen: *expression* kann arithmetischer Typ, pointer Typ oder Objekt einer definierten Klasse sein.

Funktionsweise: Zuerst wird *expression* berechnet. Ergibt die Berechnung einen von 0 verschiedenen Wert wird *statements* ausgeführt. Nach jeder Ausführung von *statements* wird *expression* erneut berechnet. Die Schleife wird erst beendet wenn *expression* 0 wird.

Bemerkung: *expression* muss vorsichtig definiert werden, so dass der Wert 0 werden kann. Ansonsten besteht die Gefahr eines Endlosloop. Mit while(1) kann bewusst eine Endlosschleife programmiert werden. While führt die Schleife solange aus wie *expression* zutrifft. Meistens wird ein Bestandteil von *expression* (Abbruchkriterium) innerhalb der *statements* inkrementiert oder dekrementiert.

2.5.2.2 do while

Do while erlaubt eine Iteration mit mindestens einer Durchführung.

Syntax: **do** *statement* **while**(*expression*);

oder: **do** {
 statement;
 ...;
}
while (*expression*);

Bedingungen: *expression* kann arithmetischer Typ, pointer Typ oder Objekt einer definierten Klasse sein.

Funktionsweise: Zuerst wird *statement* einmal ausgeführt, anschliessend *expression* berechnet. Ergibt die Berechnung einen von 0 verschiedenen Wert wird *statement[s]* erneut ausgeführt. Nach jeder Ausführung von *statement[s]* wird *expression* erneut berechnet. Die Schleife wird erst beendet wenn *expression* 0 wird.

Bemerkung: *expression* muss vorsichtig definiert werden, so dass der Wert 0 werden kann. Ansonsten besteht die Gefahr einer Endlosschleife. Mit do .. while(1) kann bewusst eine Endlosschleife programmiert werden.

2.5.2.3 for

For erlaubt eine Iteration mit Initialisierung einer Anfangs- und Definition einer Endbedingung für die Schleifenausführung.

Syntax: **for** (*init-expression*; *end-expression*; *iterator-expression*) *statement*;

oder: **for** (*init-expression*; *end-expression*; *iterator-expression*) {
 statement;
 ...
}

Bedingungen: *init-expression* und *end-expression* kann arithmetischer Typ, pointer Typ oder Objekt einer definierten Klasse sein.

Funktionsweise: Zuerst wird *init-expression* ausgeführt. Anschliessend wird *end-expression* ausgewertet. Trifft *end-expression* zu wird *statement* ausgeführt, anschliessend wird *iterator-expression* ausgewertet und dann *end-expression* erneut berechnet. Trifft *end-expression* zu, wiederholt sich der "*statement* / *end-expression*" Zyklus so lange bis *end-expression* nicht mehr zutrifft (FALSE liefert).

Bemerkung: *end-expression* muss vorsichtig definiert werden, so dass der Wert zutreffen kann. Ansonsten besteht die Gefahr einer Endlosschleife. Mit *for(;;)* kann bewusst eine Endlosschleife programmiert werden.

```
1  #include <iostream>
2  using namespace std;
3
4  int main(int argc, char* argv[])
5  {
6      // while loop
7      int x = 1;
8      while ( x<=10 ){
9          cout << x << ". Schlaufendurchgang" << endl;
10         x++;
11     }
12
13     // do while loop
14     int y = 1;
15     do {
16         cout << y << ". Schlaufendurchgang" << endl;
17         y++;
18     } while ( y<=10 );
19
20     // for loop
21     for ( int z=1; z<=10; z++ ){
22         cout << z << ". Schlaufendurchgang" << endl;
23     }
24 }
```

2.5.3 Sprunganweisungen

2.5.3.1 break

`break` erlaubt in einer Iteration- oder switch-Anweisung einen gezielten Sprung auf die erste Anweisung hinter der Iteration oder der switch-Anweisung.

Syntax: **`break;`**

2.5.3.2 continue

`continue` erlaubt in einer Iterations-Anweisung einen unbedingten Sprung an das Iterationsende. Als nächste Verarbeitung wird nach einem *continue-expression* oder *end-expression* der Schleifenkontrolle berechnet. Je nach Resultat wird dann die Schleife erneut ausgeführt oder nicht.

Syntax: **`continue;`**

2.5.3.3 return

Eine `return`-Anweisung kehrt aus einer Funktion in den aufrufenden Kontext zurück. Es wird anschliessend die unmittelbar nächste Codezeile nach dem Funktionsaufruf ausgeführt, respektive der Ausdruck in dem der Funktionsaufruf steht gemäss den Prioritätsregeln weiter aufgelöst.

Syntax: **`return expression;`**

Bedingungen: *expression* muss vom selben Typ sein wie der definierte Rückgabewert im Funktionsprototyp.

Bemerkung: *expression* wird nur bei Funktionen angegeben, die nicht als `void` definiert sind. Wenn die Funktion mit Rückgabewert `void` definiert ist, darf *expression* fehlen.

2.5.3.4 goto

Eine `goto`-Anweisung bewirkt einen unbedingten Sprung auf ein Label (identifier).

Syntax: **`goto identifier ;`**
...
identifier:

Bedingungen: *identifier:* muss in der selben Funktion wie das `goto`-Statement definiert sein.

Bemerkung: `goto` ist aus Sicht der strukturierten Programmierung tunlichst zu vermeiden!!!

2.6 Funktionen

Funktionen ergänzen die bis jetzt kennengelernten Sprachelemente. Wir unterscheiden Standardfunktionen und Benutzerfunktionen zur Auswertung von Ausdrücken. Sind die Funktionen klassenbehaftet, das heisst nur in Verwendung einer bestimmten Klasse benutzbar, sprechen wir von Funktions-Mitgliedern einer Klasse. Standardfunktionen

und Klassen (resp. Klassen und deren Funktionen) können in sogenannten Funktions- respektive Klassenbibliotheken zusammengefasst werden.

2.6.1 Aufbau und Wirkungsweise von Funktionen

2.6.1.1 Generelle Betrachtung

Funktionen sind in sich geschlossene Verarbeitungseinheiten denen als Input 0 bis n Parameter übergeben werden, und von denen genau ein Wert zurück erhalten werden kann. Funktionen werden mit einem Bezeichner benannt. Sie können fast beliebig verschachtelt werden. Ruft eine Funktion sich selber auf, sprechen wir von einer rekursiven Funktion.

Bei der Verwendung von Funktionen ist generell zu beachten, dass die Funktion vor deren Anwendung im Softwarekontext definiert werden muss. Ist das nicht möglich ist die Funktion im Kontext für den Compiler zu deklarieren. Predeklaration heisst, dass dem Compiler der Funktionsname, der Rückgabotyp sowie die Typen der Argumente, die die Funktion verwendet angegeben werden müssen, bevor die Funktion selber benutzt oder definiert wird.

2.6.1.2 Syntax von Funktionen

Globale Funktionen:

```
[Spezifizierer] [Rückgabotyp] Bezeichner([Typ Arg1][[, Typ Arg2]..[, Typ Arg_n]])
{
    statements;
    [return Rückgabewert;]
}
```

oder

```
[Spezifizierer] [Rückgabotyp] ::Bezeichner([Typ Arg1][[, Typ Arg2]..[, Typ Arg_n]])
{
    statements;
    [return Rückgabewert;]
}
```

oder bei Funktions-Membren (Klassenfunktionen):

```
[Spezifizierer] [Rückgabotyp] Klasse::Bezeichner([Typ Arg1]..[, Typ Arg_n])
{
    statements;
    [return Rückgabewert;]
}
```

Funktionsweise: Bezeichner respektive Klassen::Bezeichner definieren den Funktionsnamen der als Einsprungpunkt in die Funktion dient. Beim Aufruf wird ein Stack aufgebaut, auf dem die Argumente zu liegen kommen. Nach dem Beenden der Funktion wird der Rückgabewert mit der Anweisung *return* an den Aufrufer zurückgegeben.

2.6.1.3 Spezifizierer

Die beiden Spezifizierer "inline" und "virtual" können nur in Verbindung mit Funktionen verwendet werden. Die beiden Spezifizierer bedeuten:

inline	bedeutet, dass der Compiler die Funktion nicht effektiv als Funktion im Code belässt, sondern den Funktionskörper an jeder die Funktion aufrufender Stelle hinein kopiert. Dies führt bei kleinen Funktionen zu besserer Performance, da der Overhead für einen Funktionsaufruf (Stackauf- und abbau) nicht betrieben werden muss. Dem gegenüber steht jedoch der Umstand, dass Coderedundanz entsteht, und durch das, der zu ladende Code grösser wird. Der Spezifizierer inline garantiert jedoch nicht für eine inline implementierte Funktion. Diese Technik setzt voraus, dass die entsprechende Funktion in der Headerdatei codiert ist. Ob und wann diese Technik sinnvoll eingesetzt wird, sei dem Leser überlassen.
virtual	kann nur in Definitionen nichtstatischer Elementfunktionen in einer Klassendeklaration vorkommen. Dieser Spezifizierer deutet an, dass die betroffene Funktion in einer abgeleiteten Klasse überschrieben werden kann und so eine andere Bedeutung erfährt als bereits in der Basisklasse implementiert ist.

2.6.1.4 Rückgabetyt

Eine Funktion hat genau einen Rückgabewert. Der Rückgabewert kann von elementarem Typ, leerem Typ (void), vom Typ Pointer oder Referenz aber auch ein ganzes Objekt sein. Der Rückgabewert kann in Ausdrücken gemäss den Umwandlungsregeln in einen anderen Typ überführt werden.

2.6.1.5 Argumente

An Funktionen können Argumente übergeben werden. Sie dienen als Eingabe in die Funktion, und können von elementarem Typ, Typ Objekt, Typ Pointer oder Typ Referenz sein. Wird der Inhalt eines elementaren Typs oder eines Objektes selber direkt übergeben, sprechen wir von "call by value". Wird hingegen ein Zeiger auf einen beliebigen Datentyp (inkl. Objekte) übergeben, sprechen wir von "call by reference".

```

1 // Deklaration der Funktion
2 int GenerateMyWindow( HWND, int, int, int, int, char * );
3
4 // -----
5 // Hauptfunktion
6 // -----
7 int main()
8 // -----
9 {
10 // Variablen deklarieren
11     int iReturnCode;
12     HWND hWnd;
13     ...
14
15 // Funktion anwenden
16 iReturnCode = GenerateMyWindow( hWnd, 5, 5, 200, 350, "Fenstertitel" );
17
18 // Wertrückgabe
19 return iReturnCode;
20 }
21
22
23 // -----
24 // Funktion: Eigenes Fenster erstellen
25 // -----
26 int GenerateMyWindow( HWND hWnd,          // Handle auf Elternfenster
27 int iXKoord,          // X-Koord. l.o. Ecke Fenster
28 int iYKoord,          // Y-Koord. l.o. Ecke Fenster
29 int iBreite,          // Fensterbreite
30 int iHoehe            // Fensterhöhe
31 char *pszWinText ) // Fenstertext
32 // -----
33 {
34 // lokale Variablen
35     int iErrorCode = 0;
36     ....
37
38 // Verarbeitung
39     ....
40
41 // Wertrückgabe
42     return iErrorCode;
43 }

```

2.6.1.6 Standardargumente

In C++ kann die Argumentenliste von Funktionen mit Standardwerten definiert werden. Dies hat zur Folge, dass für die aufrufende Instanz die Angaben der Argumente optional erfolgen kann.

Es ist jedoch zu beachten, dass in der Funktionsdeklaration die Argumente mit Standardwerten von rechts her lückenlos definiert werden.

```

1 // Deklaration der Funktion (korrekte Version)
2 // -----
3 int GenerateMyWindow( char *,
4 HWND hWnd = NULLHANDLE,
5 int iXKoord = 0,
6 int iYKoord = 0,
7 int iBreite = 200,
8 int iHoehe = 100 );
9
10 // Deklaration der Funktion (falsche Version)
11 // -----
12 int GenerateMyWindow( char *,
13 HWND hWnd = NULLHANDLE,
14 int iXKoord = 0,
15 int iYKoord = 0,
16 int iBreite,
17 int iHoehe );
18
19 // -----
20 // Hauptfunktion
21 // -----
22 main ()
23 // -----
24 {
25     ...
26     // Funktion anwenden
27     HWND hWnd = ....;
28     iReturnCode = GenerateMyWindow( "Fenstertitel", hWnd, 50, 50 );
29     ...
30 }
31
32 // -----
33 // Funktion: Eigenes Fenster eröffnen
34 // -----
35 int GenerateMyWindow( char *pszWindowText, // Fenstertext
36 HWND hWnd, // Handle auf Elternfenster
37 int iXKoord, // X-Koord. l.o. Ecke Fenster
38 int iYKoord, // Y-Koord. l.o. Ecke Fenster
39 int iBreite, // Fensterbreite
40 int iHoehe ) // Fensterhöhe
41 // -----
42 {
43     ....
44 }

```

2.6.1.7 Zeiger auf Funktionen

Bei sehr dynamischen Vorgängen kann es vorkommen, dass Funktionsadressen zum Zeitpunkt des Compile-Link nicht bekannt sind, sondern erst "at runtime" errechnet werden können. In diesem Fall muss eine Funktion über einen Funktionspointer aufgerufen werden. Um einen Funktionspointer zu deklarieren gilt folgende Syntax:

[Spezifizierer] [Rückgabebetyp] (*Bezeichner) ([Arg1][,Arg2]..[,Arg_n]);

```

1 // Deklaration des Funktionspointers
2 typedef int (*Makro)( char * );
3
4 ...
5 char *pszString = "Test";
6 Makro fpMakro;
7 fpMakro = GetProcAddress( "TheFunction" );
8
9 ...
10 // Aufruf der Funktion
11 iReturn = (*fpMakro)( pszString );
12 ...

```

2.6.1.8 Funktionsüberladung

Funktionen werden in C++ aufgrund ihres Bezeichners sowie deren Typen und Argumentenzahl unterschieden. Die Deklaration von Funktionen mit dem selben Bezeichner jedoch verschiedenen Inputtypen wird Funktionsüberladung genannt.

Es gelten dabei folgende Regeln:

Funktionen die sich nur durch den Funktionswert (Rückgabetyt) unterscheiden, dürfen nicht überladen werden.

"Typ" und "Typ &" sind als Typen formaler Argumente nicht unterschiedlich genug.

"Typ" und "const Typ" sind als Typen formaler Argumente nicht unterschiedlich genug.

"Typ *" und "Typ []" sind als Typen formaler Argumente nicht unterschiedlich genug.

Unterscheidungen von Typen in formalen Argumenten, die lediglich auf einem typedef basieren, sind nicht unterschiedlich genug.

Kombinationen von obigen Definitionen führen ebenfalls zu Fehldefinitionen.

```

13 // Typendefinition
14 typedef char * PSZ;
15
16 // Gültige Funktionsüberladung
17 void print( const char *str, int breite );
18 void print( const char *str );
19 void print( const double Wert, int breite );
20 void print( const int Wert, int breite );
21
22 // Ungültige Funktionsüberladungen - nur Funktionswert
23 int print( const char *str, int breite );
24
25 // Unterschied erstes Argument nicht ausreichend
26 void print( char *str, int breite );
27
28 // Unterschied erstes Argument nicht ausreichend
29 void print( PSZ str, int breite );

```

Damit das Entwicklungssystem die verschiedenen Funktionen mit gleichem Namen unterscheiden kann (Linkage), erzeugt der Compiler aus der Funktionssignatur (Name und Argumente) einen neuen, codierten Funktionsnamen. Dieser Vorgang wird Dekoration von Funktionsnamen (engl. namemangling) genannt. Leider ist der dazu notwendige Algorithmus nicht normiert, so dass Module verschiedener Hersteller untereinander nicht zwingend linkbar sind.

2.6.1.9 Rekursive Funktionen

Eine Funktion die sich selber aufruft, wird rekursive Funktion genannt. Bei der Konstruktion rekursiver Funktionen sind zwei Dinge immer abzuschätzen:

- Die erwartete Rekursionstiefe liefert uns die zu erwartende Stackgrösse. Jeder Funktionsaufruf hat einen Stackaufbau zur Folge. Ist nun das zu erwartende Stackaufkommen pro Funktionsaufruf sehr gross, so bedeutet das, dass der Stack sehr rasch sehr gross werden kann.
- Absicherung, dass ein Abbruchkriterium erreicht werden kann, oder künstliche Absicherung durch einen "Rekursionsoverflow" der in der Funktion geführt wird.

2.6.2 Standardfunktionen

Die Standardfunktionen sind für verschiedenste Compiler und Maschinen verfügbar. Kapitel 7 zeigt eine Übersicht der gebräuchlichsten Standardfunktionen. Bei der Verwendung von Standardfunktionen ist generell zu beachten, dass die entsprechenden Definitionsfiles am Anfang eines C++ Moduls deklariert werden. Ansonsten können unliebsame Überraschungen erlebt werden.

```

1 // Deklarationsfiles der Standardbibliotheken einbinden
2 #include "stdafx.h"
3 #include <iostream>
4
5 using namespace std;
6
7 int main(int argc, char* argv[])
8 {
9     // while loop
10    int x = 1;
11    while ( x<=10 ){
12        cout << x << ". Schlaufendurchgang" << endl;
13        x++;
14    }
15
16    ...
17 }
```

2.6.3 Klassenbibliotheken (Foundation Classes)

Klassenbibliotheken sind Sammlungen von Klassen zu bestimmten Themengebieten. Mit den meisten Compilern und spezifischen Entwicklungsumgebungen werden heute Klassenbibliotheken ausgeliefert. Ab Version 3 des Sprachstandards sollte mindestens die Standard Template Library (sog. STL) im Lieferumfang eines Compilers enthalten sein.

Der grosse Vorteil von solchen Bibliotheken besteht darin, dass eine breite Grundfunktionalität angeboten wird. Der Entwickler kann bei guten Kenntnissen der Bibliothek extrem rasche Programmentwicklungen durchführen. Genau darin liegt jedoch der Nachteil der Klassenbibliothek. Die Effizienz ist abhängig vom Kenntnisgrad der vordefinierten Klassen und Klassen-Membere. Bei grossen Bibliotheken ist es oft nicht einfach eine Übersicht zu bekommen, da die Vielfalt extreme Dimensionen annehmen kann.

So besteht die Microsoft Foundation Class (MFC) von Visual C++ für Windows aus ca. 100 Klassen mit insgesamt ca. 1500 Funktions-Membere. Für die Nutzung einer solchen Klassenbibliothek sind zu den Kenntnissen über Existenz der Klassen und deren Funktionen, die Zusammenhänge und Kenntnisse von "Sideeffects" für eine effiziente Implementation besonders wichtig.

2.7 Geltungsbereich und Lebensdauer

Als Geltungsbereich wird der Programmbereich genannt indem ein Bezeichner oder ein Literal gültig ist. Der Geltungsbereich hängt meistens mit der Lebensdauer derselben Angaben zusammen. Grundsätzlich wird zwischen globalem und lokalem Geltungsbereich unterschieden. Bezeichner mit globalem Geltungsbereich besitzen in der Regel während der gesamten Ausführung des Programms einen Speicherplatz, der zudem von allen Funktionen eingesehen werden darf. Daten mit globalem Geltungsbereich werden im Datenteil der Applikation geführt. Bezeichner mit lokalem Geltungsbereich können nur innerhalb des definierten Blockes (in der Regel eine Funktion) angesprochen werden. Daten mit lokalem Geltungsbereich werden auf dem Stack angelegt.

2.7.1 Spezifikation von Speicherklassen

Mit der Definition der Speicherklasse kann der Geltungsbereich respektive die Lebensdauer der Variablen beeinflusst werden. Es werden folgende Speicherklassen unterschieden:

auto	Defaultspeicherklasse für lokale Daten (wird benutzt wenn nichts anderes spezifiziert ist).
register	Werte werden in einem geeigneten Prozessorregister geführt. Ist vor allem bei Iterationen als Zähler zu verwenden (Performance).
static	Bewirkt dass lokale Daten den Inhalt während der ganzen Programmdauer beibehalten (Verlängerung der Lebensdauer). Defaultspeicherklasse für globale Daten (wird benutzt wenn nichts anderes spezifiziert ist).
extern	Gibt an, dass kein Speicherplatz für die Variable oder das Objekt reserviert werden muss. Bei extern-Angabe wird die Referenz der Variablen vom Linker auf eine Variablen gleichen Namens in einem anderen Modul aufgelöst. Extern bedeutet Deklaration aber nicht Definition.

	Deklaration ausserhalb sämtlicher Blöcke	Deklaration innerhalb eines Blockes, mit Spezifizierer static	Deklaration innerhalb eines Blockes, ohne Spezifizierer static
Geltungsbereich	global	lokal	lokal
Lebensdauer	gesamte Laufzeit des Programms	gesamte Laufzeit des Programms	Laufzeit des Blockes
Speicherklasse	static	static	auto
Initialisierung	implizit zu 0	implizit zu 0	keine
Speicherung	im Datenteil	im Datenteil	auf dem Stack

Tabelle 2-8 Auswirkungen der Spezifikation von Speicherklassen.

2.8 Modularisierung

2.8.1 Trennung von Interface und Implementation

Zur besseren Trennung und einfacheren Verwendung werden in C++ die Interfacespezifikationen von den Implementierungen physisch getrennt. Das Interface wird in einer sogenannten Headerdatei gespeichert, währenddem die Implementation im Codefile untergebracht ist.

Die Headerdatei beinhaltet²:

- Definitionen von Literalkonstanten
- Verweise auf andere Headerdateien
- Globale Variablen (wenn nötig)
- Externe Variablen (wenn nötig)
- Typdefinitionen
- Klassendeklaration mit Daten-Member und Prototypen der Funktions-Member
- Prototypen von allgemeinen Funktionen

Die Reihenfolge der Deklarationen spielt, solange keine Abhängigkeiten vorhanden sind, keine Rolle. Eine Headerdatei darf nicht zyklisch im Compilationsprozess vorkommen. Sie kann aber von beliebig vielen Compilationsprozessen benutzt werden.

Die Implementation beinhaltet:

- Implementation der allgemeinen Funktionen
- Implementation der Funktions-Member

Die Reihenfolge der Funktionsimplementation einer Klasse spielt keine Rolle. Das Modul soll jedoch nur einmal kompiliert und darf nur einmal in die Software eingebunden werden.

In C++ werden die Trennungen sehr oft so gemacht, dass pro Klasse ein File für das Interface und ein File für die Implementation erstellt wird. Insbesondere moderne Tools wählen diesen Weg der Modularisierung. Diese Aufteilung ist nicht immer sinnvoll und kann zu einer wahren "Modul-Inflation" führen.

Bei komplexen Projekten empfiehlt sich unbedingt eine Modulplanung. Die Modulplanung hilft:

- die Abhängigkeiten zwischen den Modulen aufzuzeigen
- die Realisierungsreihenfolge der Module festzulegen
- das Testen der Module zu planen

² C++ schreibt keine wirkliche Ordnung vor. Die hier abgedruckte Aufteilung stammt aus der praktischen Erfahrung.

2.8.2 Einbindung von C Code in C++

Sowohl für das eigentliche Anwendungsprogramm als auch für die einzelnen Module (Klassen) können C Funktionen eingebunden werden. Dieser Fall tritt oft ein, wenn bereits vorhandene Funktionen aus Bibliotheken aufgrund von Investitionsschutz wiederverwendet werden sollen. Zur korrekten Steuerung des Compile und Linkvorganges sind dabei die Funktionsdeklarationen im Ausdruck `extern "C" {}` zu klammern.

```
1 // Verwendung von C Funktionen
2 extern "C" {
3     int    printf( char *, ...);
4     void   exit( int );
5 }
```

Für die Einbindung ganzer C-Header in C++ dient folgende Schreibweise:

```
6 // Verwendung von C Funktionen in einer C++-Header Datei
7 extern "C" {
8     #include< MyOldCLib.h >
9 }
```

Beim Erstellen von neuen C-Header Dateien kann unter Berücksichtigung der vordefinierten Konstanten `__cplusplus` die Datei gleich für die Verwendung in beiden Sprachen erzeugt werden:

```
10 // situative Verwendung der Header je nach Compiler
11 #ifdef __cplusplus
12     extern "C" {
13 #endif
14
15     int    printf( char *, ...);
16     void   exit( int );
17
18 #ifdef __cplusplus
19     }
20 #endif
```

2.8.3 Einsprungpunkt in ein C++ Programm

Der Startpunkt eines C++ Programm bildet die Funktion *main()*.

main() ist in einem Programm genau einmal vorhanden. Die Funktion ist vom Compiler nicht vordefiniert.

main() darf programmintern nicht als Funktion aufgerufen werden.

main() darf nicht überladen werden und hat einen implementationsabhängigen Typ.

main() darf nicht static oder inline definiert werden.

Die Adresse von main() kann programmintern nicht ermittelt werden.

Der Funktion werden folgende Daten übergeben:

- argc Anzahl übergebene Argumente
- argv Array von Vektoren auf die einzelnen Argumente
- env Array von Vektoren auf die Systemumgebung

Das Argument argv[0] enthält in der Regel den Namen, der für den Aufruf des Programms verwendet wurde. Es ist garantiert, dass argv[argc] = 0 ist. Dies ist insbesondere bei Funktionen für die Auswertung der Übergabeargumente von Bedeutung.

```
1  // -----
2  // Function: main
3  //
4  // Return:   int to operatingsystem
5  //           0   run without error
6  // -----
7  int main( int argc,          // count of given arguments
8  char *argv[],              // vector to pointer to arguments
9  char *envp[] )             // vector to pointer to environment
10 // -----
11 {
12     // local variables and objects
13     int          iError = 0;          // internal errorcode
14     :
15
16     // Initiate data
17     if ( argc > 1 ) {
18         // filename for input has not been update on command line
19         strcpy( szInputFile, argv[ 1 ] );
20     }
21
22     if ( argc > 2 ) {
23         // filename for output has not been update on command line
24         strcpy( szOutputFile, argv[ 2 ] );
25     }
26     :
27
28     // give back the errorcode
29     return iError;
30 }
```

3 Klassen und Objekte

Dieses Kapitel zeigt den Aufbau von Klassen und den Unterschied zu den Objekten die aus Klassen gebildet werden. Objekte werden mit Funktionen der Klasse bearbeitet. Spezielle Typen von Funktionen ergänzen die Technik der objektorientierten Programmierung. Das Kapitel erläutert diese Spezialitäten.

3.1 Was ist eine Klasse

Eine Klasse ist funktionell mit einer Typendeklaration einer Struktur in C vergleichbar. In Ergänzung zur Deklaration der Struktur, werden innerhalb der selben nicht nur Daten deklariert, sondern auch Funktionen. Für Daten und Funktionen können mittels Berechtigungen verschiedene Gültigkeitsbereiche deklariert werden.

Eine Klasse wird im Code nur als Typ deklariert. Sie besitzt somit keinen physischen Speicher.

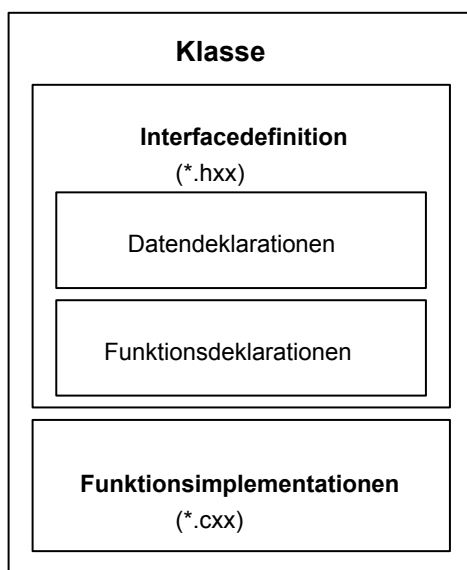


Abbildung 3-1 Schematische Darstellung der Elemente einer kompletten Klassenimplementation

Die Daten und Funktionen einer Klasse werden auch Member, Elemente oder Attribute einer Klasse genannt. Die Verwendung der Begriffe ist abhängig von der verwendeten Programmiersprache. Im Zusammenhang mit der Sprache C++ hat sich der Begriff Member etabliert.

3.2 Interface der Klasse

Die Deklaration einer Klasse kann mit einem der folgenden Schlüsselworte *class*, *struct* oder *union* eingeleitet werden. Mit Hilfe der Klassendeklaration wird der Compiler in die Lage versetzt, diese zu kennen. Mit der reinen Deklaration einer Klasse kann grundsätzlich nicht gearbeitet werden (ausser der Wiederverwendung zur Datendefinition). Im Interface einer Klasse werden die Daten- und Funktions-Member deklariert.

```

1  class CSWS_Point {
2      int iX_Coord,          // X Koordinate des Punktes
3      int iY_Coord;          // Y Koordinate des Punktes
4
5  public:
6      void Set( int, int); // Deklaration der Funktion Set
7  };

```

Der Bezeichner nach dem Schlüsselwort heisst "Klassenname".

Variablen in Klassendeklarationen dürfen nicht initialisiert werden, da die Klassendeklaration lediglich einer Typendeklaration entspricht und somit kein Speicher zur Verfügung steht, der initialisiert werden kann. Die Funktionsdeklaration bildet das Prototyping der Funktion (siehe auch "Aufbau und Wirkungsweise von Funktionen").

```

8  class CSWS_Point {
9      int iX_Coord = 0; // Fehler: keine Initialisierung zulässig
10     int iY_Coord;     // korrekte Deklaration
11     ...

```

Bei der Verwendung des Schlüsselwortes *struct* ist zu berücksichtigen, dass die Zugriffsrechte nicht identisch wie bei der Verwendung von *class* sind. Untenstehendes Beispiel zeigt die Schreibweisen beider Formen zum Erlangen einer identischen Deklaration (siehe auch Zugriffsrechte und Geltungsbereich).

```

12 struct CSWS_Point {
13     int iX_Coord,          // X Koordinate des Punktes
14     int iY_Coord;          // Y Koordinate des Punktes
15     void Set( int, int); // Deklaration der Klassenfunktion Set
16 };
17
18 // Identische Definition, jedoch mit Schlüsselwort class (ist
19 // vorzuziehen)
20 class CSWS_Point {
21 public:
22     int iX_Coord,          // X Koordinate des Punktes
23     int iY_Coord;          // Y Koordinate des Punktes
24     void Set( int, int); // Deklaration der Klassenfunktion Set
25 };

```

3.3 Implementation der Klasse

Um die Deklaration einer Klasse zu vervollständigen, müssen die deklarierten Funktions-Member noch implementiert werden. Dabei ist zu beachten, dass der Funktionsname in sich aus Klassenname und Member-Name zusammengesetzt ist.

```
1 void CSWS_Point::Set(int xWert, int yWert )
2 {
3     iX_Coord = xWert;
4     iY_Coord = yWert;
5 }
```

Die Deklaration einer Klasse wird sinnvollerweise in einem Headerfile (*.h) und die Implementation in einem C++ File (*.cpp) realisiert. Diese Konvention bringt den Vorteil, dass Benützer (Programmierer) der Klasse zwar das Interface sehen (benutzen kann), jedoch bleiben ihm die Details der Implementation verborgen.

3.4 Instanzenbildung (Herstellung von Objekten)

3.4.1 Statische Instanziierung

Aus den Klassendeklaration können Objekte (einer bestimmten Klasse) definiert (instanziert) werden (analog der Definition von Variablen eines bestimmten Basistypes). Erst mit der Definition eines Objektes wird der Compiler für die Daten der Klasse Memory bereitstellen. Selbstverständlich können in einem Programm so viele Objekte einer bestimmten Klasse erzeugt werden, wie nötig sind.

```
6 CSWS_Point swsptStart, swsptEnde; // es werden 2 Objekte der Klasse
7                                 // CSWS_Point definiert
```

Nach der Definition von Objekten ist eine Klasse zum eigentlichen Leben erweckt worden. Jedes erzeugte Objekt erhält seine eigenen Daten. So wird der Speicher für obenstehendes Codefragment mit swsptStart und swsptEnde wie folgt organisiert:

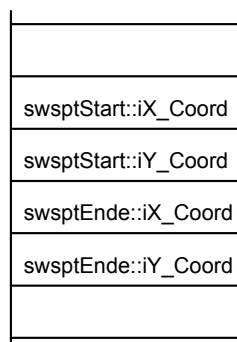


Abbildung 3-2 Schematische Darstellung der Datenspeicherung für zwei Objekte gleichen Typs.

Nebst der oben dargestellten Instanziierung ist es auch möglich, Objekte direkt mittels der Klassendeklaration zu definieren. Diese Form der Objektinstanziierung wird jedoch in der Praxis wenig angewendet, da sie der Modularität nicht förderlich ist.

```
1 struct CSWS_Point {
2     int    iX_Coord,          // X Koordinate des Punktes
3     int    iY_Coord;         // Y Koordinate des Punktes
4     void Set( int, int);      // Deklaration der Klassenfunktion Set
5 } swsptStart, swsptEnde;     // es werden 2 Objekte der Klasse
6                             // CSWS_Point definiert
```

3.4.2 Dynamische Instanziierung

Sind die Objekte zum Zeitpunkt der Codierung nicht aus dem Code ersichtlich, sondern werden erst zur Laufzeit des Programms erzeugt, sprechen wir von dynamischer Instanziierung. Dabei wird mittels dem Operator `new` ein Objekt erzeugt. Der resultierende Pointer auf das Objekt wird einer entsprechend definierten Variablen übergeben. Die dynamische Handhabung von Objekten kommt überall dort vor, wo die Anzahl zu benutzender Objekte zum Zeitpunkt der Programmerstellung noch nicht bekannt ist (Listen, Bäume...).

```
7 CSWS_Point *pwsptP1, *pwsptP2; // Definition 2er Pointer auf
8                               // Objekte der Klasse CSWS_Point
9
10 // dynamische Erzeugung der Objekte
11 pwsptP1 = new CSWS_Point;
12 pwsptP2 = new CSWS_Point;
13
14 // Nutzung der dynamisch angelegten Objekte
15 ...
16
17 // Löschen der Objekte
18 delete pwsptP1;
19 delete pwsptP2;
```

Dynamisch erzeugte Objekte werden auf dem Heap angelegt. Der Handhabung der Objekte muss grösste Aufmerksamkeit geschenkt werden. Insbesondere das "Freigeben" des Speichers ist von grösster Wichtigkeit, sollen nicht unliebsame Nebeneffekte auftreten (Langzeitverhalten eines Programmes).

3.5 Nutzung von Instanzen

3.5.1 Ansprechen statischer Objekte

Ist ein Objekt vorhanden, kann es mit den Klassenfunktionen sofort bearbeitet werden. Ein Daten-Member wird innerhalb eines Ausdrucks in Bezug auf den Objektname verwendet. Die volle Schreibweise in allgemeingültiger Form lautet:

Objektname.Klassenname::Datenmember

Normalerweise benutzt man jedoch folgende, abgekürzte Schreibweise:

Objektname.Datenmember

```
1  CSWS_Point swsptStart, swsptEnde;    // es werden 2 Objekte der Klasse
2                                          // CSWS_Point definiert
3  int ixWert;                          // Hilfswert
4
5  ...
6  swsptStart.iX_Coord = 25;
7  swsptStart.iY_Coord = swsptStart.iX_Coord;
8  ixWert = swsptStart.iX_Coord;
9  ...
```

Ein Funktions-Member wird innerhalb eines Ausdrucks in Bezug auf den Objektname angesprochen. Die volle Schreibweise in allgemeingültiger Form lautet:

Objektname.Klassenname::Memberfunktion(Arg 1, ... , Arg n);

Normalerweise benutzt man folgende, abgekürzte Schreibweise:

Objektname.Memberfunktion(Argument 1, ... , Arg n);

```
10 CSWS_Point   swsptEnde;
11 ...
12
13 // normale Schreibweise
14 swsptEnde.Set( 100, 100 );
15
16 // explizite Schreibweise
17 swsptEnde.CSWS_Point::Set( 100, 100 );
```

Im gezeigten Beispiel sind ausschliesslich klassenspezifische Funktions-Member verwendet worden. Es kann vorkommen, dass globale Funktionen und Funktions-Member mit gleichen Namen vorhanden sind. Für diesen Fall muss vorgesehen werden, dass auch globale Funktionen explizit aufgerufen werden können.

```
1  struct C_SESSION {
2      long lPID;          // Datenmember long
3      char *szSessionName; // Datenmember pointer auf char
4      ...
5      long fork( .... ); // Deklaration der Klassenfunktion fork
```

```

6  };
7
8  ...
9  C_SESSION MySessionObject;    // Instanziierung eines Objektes
10
11 ...
12 MySessionObject.fork( ... ); // Aufruf der Funktion fork eines Obj.
13                               // der Klasse C_SESSION
14 ...
15 ::fork(...);                 // expliziter Aufruf der globalen Funktion
16                               // fork durch verwendung des Scopeoperators
17 fork(...);                   // gleichwertiger Funktionsaufruf

```

3.5.2 Ansprechen dynamischer Objekte

Dynamisch instanzierte Objekte werden über die normale Zeigerdereferenzierung angesprochen.

```

18 CSWS_Point *pswsptP1, *pswsptP2; // Definition 2er Pointer auf
19                                   // Objekte der Klasse CSWS_Point
20
21 // dynamische Erzeugung der Objekte
22 pswsptP1 = new CSWS_Point;
23 pswsptP2 = new CSWS_Point;
24
25 // Nutzung der dynamisch angelegten Objekte
26 pswsptP1->Set( 0, 0 );
27 pswsptP2->Set( 100, 53 );
28
29 // Löschen der Objekte
30 delete pswsptP1;
31 delete pswsptP2;

```

3.5.3 Implizites Member-Argument (this)

Ein Funktions-Member einer Klasse erhält bei dessen Aufruf ein implizites Argument, welches das Objekt bezeichnet, für das die Funktion aufgerufen wurde. Das übergebene Argument kann innerhalb des Rumpfes des Funktions-Members explizit unter dem Namen *this* angesprochen werden. Diese Massnahme kann notwendig werden, wenn der Name des anzusprechenden Members nicht mehr eindeutig ist, oder wenn in eine Funktion der Pointer der gerade aktiven Instanz an eine andere Funktion übergeben werden muss.

```

32 void CSWS_Point::Set( int xWert, int yWert )
33 {
34     this->iX_Coord = xWert; // identisch mit iX_Coord = xWert
35     this->iY_Coord = yWert; // identisch mit iY_Coord = yWert
36     func( this );          // Ruft Funktion func(CSWS_Point *) auf
37 }

```

3.6 Zugriffsrechte und Geltungsbereich

In obigen Beispielen sind alle Daten- und Funktions-Member jeweils zu jedem Zeitpunkt von aussen erreichbar (Verwendung der *struct*-Definition der Klasse CSWS_Point). Die Klassen-Member sind nicht gekapselt, und die Funktion Set() scheint einen "Durchlauferhitzer" darzustellen. Dieses allgemeine Zugriffsrecht wird *public*-Zugriff genannt. Wäre nur diese Stufe des Zugriffes auf Daten und Funktionen definiert, würde das wichtige Konzept der Kapselung unterlaufen.

C++ definiert folgende Zugriffsrechte:

Zugriffsrecht	Bedeutung
public	Auf ein <i>public</i> Klassenelement kann aus jeder Funktion eines Programms zugegriffen werden. In der Regel werden nur diejenigen Klassenelemente als <i>public</i> definiert, die den Zugang zu den Daten und Funktionen ermöglichen sollen.
protected	Auf <i>protected</i> Klassenelemente kann selbstverständlich aus den eigenen Klassenfunktionen aber auch aus den Funktionen vererbter Klassen zugegriffen werden. Für Zugriffe von Aussen (globaler Scope oder nicht vererbte Klassen) besteht jedoch keine Zugriffsmöglichkeit.
private	Klassenelemente mit dem Spezifizierer <i>private</i> können nur aus den Element- und Friend-Funktionen der eigenen Klasse angesprochen werden (gegen aussen gekapselt).

Tabelle 3-1 Zugriffsspezifizierer für Klassen-Member.

Ein Zugriffsspezifizierer stellt den Zugriff auf die ihm folgenden Member bis zum Ende der jeweiligen Klassendefinition oder bis zum nächsten Zugriffsspezifizierer fest. Innerhalb einer Klassendefinition können beliebig viele Zugriffsspezifizierer in beliebiger Reihenfolge definiert werden.

Das Standard-Zugriffsrecht für Klassen die mittels *class* definiert werden ist *private*, für Klassen die mittels *struct* definiert werden *public*.

```
1 // KLassendeklaration
2 class CSWS_Point {
3     int    iX_Coord,           // X Koordinate des Punktes
4           iY_Coord;           // Y Koordinate des Punktes
5 public:
6     void Set( int, int);       // Deklaration der Klassenfunktion Set
7     void Move( int, int );    // Deklaration der Klassenfunktion Move
8 };
9
10 // Klassenimplementation
11 void CSWS_Point::Set( int X, int Y ) {
12     iX_Coord = X;
13     iY_Coord = Y;
14 }
```

```

15 // Nutzendes Programm
16 main () {
17     CSWS_Point    P1;    // Definition von einem Punkt-Objekt
18
19     P1.iX_Coord = 5;    // Fehler! Private Member können nur von Funktions-
20     P1.iY_Coord = 10;    // member der gleichen Klasse angesprochen werden
21     P1.Set( 5, 10 );    // Ok! Daten werden via Funktionsmember übermittelt
22 }

```

Die Namen von Klassenelementen haben den Geltungsbereich Klasse. Das hat zur Folge, dass der selbe Name für ein Daten- oder Funktions-Member in mehreren Klassen definiert werden kann. Es ist Sache der Sprache, jeweils pro Objekt die richtige Funktion im Speicher zu adressieren.

```

1  // -----
2  // Klassendeklaration Punkt
3  // -----
4  class CSWS_Point {
5      int    iX_Coord,            // X Koordinate des Punktes
6          iY_Coord;            // Y Koordinate des Punktes
7
8  public:
9      void Set( int, int);        // Deklaration der Klassenfunktion Set
10     void Move( int, int );    // Deklaration der Klassenfunktion Move
11 };
12
13 // -----
14 // Klassendeklaration Linie
15 // -----
16 class CSWS_Line {
17     int    iX1_Coord,            // X Koordinate des Punktes 1
18         iY1_Coord;            // Y Koordinate des Punktes 1
19     int    iX2_Coord,            // X Koordinate des Punktes 2
20         iY2_Coord;            // Y Koordinate des Punktes 2
21
22 public:
23     void Set( int, int);        // Deklaration der Klassenfunktion Set
24     void Move( int, int );    // Deklaration der Klassenfunktion Move
25 };
26
27 // -----
28 // Testprogramm
29 // -----
30 void main()
31 {
32     CSWS_Point swsptEnde, swsptStart;
33     CSWS_Line  swslnLine;
34
35     ...
36     // Nutzung der Movefunktion für Objekt der Klasse Punkt
37     swsptEnde.Move( 100, 100 );    // implizite Schreibweise
38     swsptEnde.CSWS_Point::Move( 100, 100 ); // explizite Schreibweise
39
40     // Nutzung der Movefunktion für Objekt der Klasse Linie
41     swslnLine.Move( 100, 100 );    // implizite Schreibweise
42     swslnLine.CSWS_Line::Move( 100, 100 ); // explizite Schreibweise
43     ...
44 }

```

3.7 Static Daten-Member

Daten-Member die als *static* deklariert werden, sind nicht an ein bestimmtes Objekt einer Klasse gebunden, sondern repräsentieren ihren Wert einmal für alle Objekte einer bestimmten Klasse. Sie sind mit externer Bindung im Datenteil einer Applikation angelegt.

Die Deklaration eines *static* Daten-Members innerhalb einer Klassendefinition ist noch keine Definition! Diese muss an separater Stelle des Programms vorgenommen werden (Schreibweise in untenstehendem Beispiel beachten).

3.7.1 Klassendeklaration mit einem static Daten-Member

Die Deklaration erfolgt in der Regel im File für die Interfacebeschreibung (*.h, *.hxx):

```
1 // Klassendeklaration im Interfacefile
2 class CSWS_Point {
3     int          iX_Coord,      // X Koordinate des Punktes
4                 iY_Coord;      // Y Koordinate des Punktes
5     static bool fVisible;      // alle Punkt sichtbar (TRUE)
6
7 public:
8     void Set( int, int );      // Dekl. der Klassenfunktion Set
9     void Move( int, int );     // Dekl. der Klassenfunktion Move
10    void SetVisibility( bool ); // Dekl. der Klassenfunktion SetVisibility
11 };
```

3.7.2 Klassenimplementierung mit einem static Daten-Member

Die Definition (und eventuell auch die Initialisierung) der static Daten-Member erfolgt in der Regel am Anfang der Funktionsimplementationen der entsprechenden Klasse im Implementationsfile (*.cxx, *.cpp):

```
12 // -----
13 // Implementation der Klasse CSWS_Point
14 // -----
15
16 // Definition der statischen Datenmember
17 // -----
18 bool CSWS_Point::fVisible = true;
19
20 ...
21 // -----
22 // Funktion Set
23 // -----
24 void CSWS_Point::Set( int iX, int iY )
25 // -----
26 {
27 ...
28 }
```

Im Speicher werden bei Instanzierung zweier Objekte der Klasse CSWS_Point aus obiger Klassendeklaration folgende Strukturen angelegt:

CSWS_Point::fVisible
swsptStart.CSWS_Point::iX_Coord
swsptStart.CSWS_Point::iY_Coord
swsptEnde.CSWS_Point::iX_Coord
swsptEnde.CSWS_Point::iY_Coord

Abbildung 3-3 Schematische Darstellung der Datenspeicherung für zwei Objekte gleichen Typs mit einem statischen Daten-Member.

3.8 Static Funktionen

Ein statisches Funktions-Member ist eine Funktion die in der Klasse deklariert ist, jedoch unabhängig von Objekten arbeitet. Sie wird für den Zugriff auf statische Daten-Member verwendet. Das statische Funktions-Member besitzt keinen this-Pointer. Somit kann innerhalb einer solchen Funktion nicht auf Elemente eines Objektes zugegriffen werden.

Ein statisches Funktions-Member darf nicht als virtuell deklariert werden, noch darf der selbe Name für ein statisches und dynamisches Klassenmember verwendet werden.

Anwendung einer statischen Funktion für die Objektzählung

```

1 // static.cpp : show the behavior and usage of static members
2 //
3 #include "stdafx.h"
4 #include <iostream>
5
6 using namespace std;
7
8 // -----
9 // class declaration
10 // -----
11 class CSWS_Point {
12
13 // static datamember
14 private:
15     static int        iCount;        // count of used Objects
16
17 // datamember
18 public:
19     int iX_Coord;        // x coordinate of point
20     int iY_Coord;        // y coordinate of point

```

```

21 // class interface - functionmember
22 public:
23     CSWS_Point() { iCount++; }
24     ~CSWS_Point() { iCount--; }
25     static int    GetCount( void );
26 };
27
28 // -----
29 // class implementation
30 // -----
31 int CSWS_Point::iCount = 0;
32
33 int CSWS_Point::GetCount( void )
34 {
35     // try to access non static elements is not allowed
36     // Error: Ungueltige Ref. auf Datenelement 'CSWS_Point::iX_Coord' in
37     // statischen Member-Funktion
38     // CSWS_Point::iX_Coord = 0;
39     return iCount;
40 }
41
42 // -----
43 // using the static function
44 // -----
45 void main(int argc, char* argv[])
46 {
47     // define pointer to a point
48     CSWS_Point *pptPoint;
49
50     // print out the count of points
51     cout << "Count of points is: " << CSWS_Point::GetCount() << endl;
52
53     // create a point ( dynamic ) and print out the count of points
54     pptPoint = new CSWS_Point;
55     cout << "Count of points is: " << CSWS_Point::GetCount() << endl;
56
57     // create a point ( static ) and print out the count of points
58     CSWS_Point P1;
59     cout << "Count of points is: " << CSWS_Point::GetCount() << endl;
60     cout << "Count of points is: " << P1.GetCount()
61         << "(output usage a object)" << endl;
62
63     // delete the dynamic created point and print out the count of points
64     delete pptPoint;
65     cout << "Count of points is: " << CSWS_Point::GetCount() << endl;
66 }

```

Resultat des obigen Programmes (Konsolenausgabe):

```

67 Count of points is: 0
68 Count of points is: 1
69 Count of points is: 2
70 Count of points is: 2(output usage a object)
71 Count of points is: 1

```

3.9 Inline Funktionen

Bei sehr einfachen Funktions-Memberrn wirkt der Aufruf einer Funktion im Vergleich zur Aufgabe als komplett überflüssig. Unter Umständen ist zur Laufzeit eines Programms eine solche Funktion viel wirksamer wenn sie nicht als eigentliche Funktion, sondern als direkter Programmschritt eingebettet ist. Dies hat zwar bei mehrfacher Verwendung der selben Funktion den Nachteil von Redundanzen, bietet aber unter Umständen extremen Performancegewinn.

Um den Gewinn der Performance zu haben und trotzdem ein sauber strukturiertes Programm schreiben zu können, bietet C++ die Möglichkeit der sogenannten InlineGenerierung von Funktionen. Dies bedeutet für den Programmierer, dass er die Funktion normal schreiben kann, und dabei den Compiler anweist, anstelle eines jeweiligen Funktionsaufrufes den Code direkt in die benutzende Routine zu kopieren. Dies wird erreicht indem die Funktion mit dem Spezifizierer inline dekoriert wird.

Inlinedefinition einer Funktion (explizite Schreibweise):

```
1  inline void CSWS_Point::Set( int xWert, int yWert )
2  {
3      iX_Coord = xWert;
4      iY_Coord = yWert;
5  }
```

Bei der Verwendung von C++ Klassen kann die Benützung von Inlinefunktionen auch erfolgen, indem die Funktion direkt in der Klassendeklaration implementiert wird.

Inlinedefinition einer Funktion (implizite Schreibweise):

```
6  class CSWS_Point {
7  private:
8      int iX_Coord,    // X Koordinate des Punktes
9      int iY_Coord;    // Y Koordinate des Punktes
10 public:
11     // inline Implementation der Klassenfunktion Set
12     void Set( int xWert, int yWert ) {iX_Coord=xWert; iY_Coord=yWert; }
13 };
```

Bei der Verwendung der impliziten Inline-Implementation (letzte Schreibweise), muss jeweils darauf geachtet werden, dass das Makefile alle Abhängigkeiten auf das entsprechende Includefile vermerkt hat. Ansonsten können unter Umständen lästige Fehlersuchen vorkommen.

Eine implizit erstellte Inlinefunktion erfordert normale Schreibweise. Das heisst:

- Jede Codezeile wird mit Strichpunkt beendet
- Der Block der Funktion wird nicht mit Strichpunkt beendet

3.10 Operatoren für Klassen

Neben Funktionen können für Klassen auch Operatoren definiert werden. Als Operatorzeichen können zu diesem Zweck die im Sprachstandard definierten Zeichen verwendet werden. Die Operatoren werden analog den Funktionen als Prototypen in der Klassendeklaration geführt und als separat implementiert (inline ist selbstverständlich auch möglich). Neben dem eigentlichen Operatorzeichen besteht der "Funktionsname" für Operatoren aus dem Schlüsselwort "operator" und dem eigentlichen Operatorzeichen. Bei der Verwendung des Operators wird nur das Operatorzeichen angegeben.

Für die Definition von Operatoren gelten folgende Regeln:

Es ist nicht möglich neue Operatorzeichen zu definieren oder bestehende Operatorzeichen zu erweitern (Beispiel `**` für Potenzierung ist nicht zulässig).

Die Prioritäten für die Auflösung von Ausdrücken ist ebenfalls festgelegt und kann nicht verändert werden (Sprachstandard).

Die für die fundamentalen Datentypen vordefinierten Operatoren werden nicht automatisch auf abstrakte Datentypen (Klassen) übertragen.

Die Operatoren für die fundamentalen Datentypen sind festgelegt und können weder umdefiniert noch erweitert werden.

Folgende Operatoren können nicht überladen werden:

- Operator `sizeof` (Grösse)
- Operator `.` (Elementzugriff)
- Operator `*` (Komponentenzeiger)
- Operator `::` (Scope)
- Operator `?:` (Konditionaloperator)

Die Auswahl der Operatorzeichen soll natürlich erfolgen. Das heisst, dass der Programmierer das zu verwendende Zeichen seiner Absicht in Bezug auf Usancen beurteilen soll. In untenstehendem Beispiel werden zwei Linien zusammengezählt (Vektorgeometrie) entsprechend ist das `+` Zeichen das logisch zu verwendende Zeichen.

```
1 // operatoren.cpp : Definiert den Einsprungpunkt für die Konsolenanwendung.
2 #include "stdafx.h"
3 #include <iostream>
4
5 using namespace std;
6
7 // -----
8 // class: CSWS_Line
9 // purpose: declare a two dimensional line
10 // -----
11 class CSWS_Line {
12 private:
13     CSWS_Point P1;
14     CSWS_Point P2;
15
16 private:
17     static unsigned long ulCounter;
```

```

18
19 // functions
20 public:
21     ...
22     CSWS_Line operator+( CSWS_Line &AddLine );
23     void Set( int x1, int y1, int x2, int y2 );
24     ...
25     static unsigned long GetCount( void ) { return ulCounter; }
26 };
27
28 // -----
29 // class implementation
30 // -----
31 ...
32 // -----
33 // class:    CSWS_Line
34 // function: assignment operator
35 // purpose:  constructs a object using a other object of same class
36 // -----
37 CSWS_Line CSWS_Line::operator+( CSWS_Line &AddLine )
38 // -----
39 {
40     // build local object
41     CSWS_Line lineTemp;
42
43     // calculate components of temporary line
44     lineTemp.P1 = P1;
45     lineTemp.P2 = P2 + AddLine.P2 - AddLine.P1;
46     return lineTemp;
47 }
48
49 // -----
50 // using the static function
51 // -----
52 void main(int argc, char* argv[])
53 {
54     CSWS_Line    L1;
55     CSWS_Line    L2;
56     CSWS_Line    L3;
57     L1.Set( 10, 10, 20, 100 );
58     L2.Set( 10, 10, 200, 15 );
59
60     // add vectors
61     L3 = L1 + L2;
62     ...
63 }

```

Die Operatoren new und delete bilden beim Definieren von Operatoren zwei Spezialfälle. Die beiden Operatoren sind als externe Bibliotheksfunktionen definiert. Die beiden Funktionen werden zwar in den meisten Fällen das gewünschte Resultat erzielen. Trotzdem könnte es vorkommen, dass die Operatoren im Rahmen eines Projektes neu definiert werden sollen.

new und delete lassen sich global oder spezifisch für eine Klasse überladen. Bei der globalen Art werden die entsprechenden Standardfunktionen ausser Kraft gesetzt. Bei der klassenspezifischen Überladung gilt die Definition natürlich nur für die betroffene Klasse.

Zum globalen Überladen werden die Funktionen als extern im globalen Gültigkeitsbereich definiert:

```
1  extern void * operator new( long bytes )
2  // -----
3  {
4      // Hier kommt der Code zum allozieren von Speicher. Es muss beachtet
5      // werden, dass die Funktion einen void * Pointer zurückgeben muss
6  }
7
8  extern void operator delete( void * ptr )
9  // -----
10 {
11     // Hier kommt der Code um den Memorybereich auf den ptr zeigt
12     // freizugeben
13 }
```

Da für die dynamische Memoryverwaltung pro aloziertem Block immer ein gewisser Overhead betrieben werden muss, ist es für zahlreiche kleine Datenobjekte oft nicht sinnvoll, jedes Objekt mit den Standardfunktionen zu erstellen. Zudem kann es vorkommen, dass das globale Überschreiben der Operatoren *new* und *delete* zu keinem besseren Ergebnis führt. In diesem Fall kann das klasseninterne Überschreiben der Operatoren allenfalls einen Ausweg aus dem Dilemma darstellen:

```
14 class CSWS_Point {
15 // -----
16     int xKoord, yKoord;
17     ...
18
19 public:
20     ...
21     void * operator new( long byte );
22     void operator delete( void * ptr );
23     ...
24 }
25
26 void * CSWS_Point::operator new( long byte )
27 // -----
28 {
29     // Code für die alozierung von Speicher für ein Point-Objekt
30 }
31
32 void CSWS_Point::operator delete( void * ptr )
33 // -----
34 {
35     // Code für die Freigabe von Speicher eines Objektes der Klasse Point
36 }
```

Nach einer nach obenstehendem Muster definierten Überladung der Operatoren, werden für die Objekte der Klasse Point nur noch die intern definierten Operatoren verwendet. Für alle anderen Objektklassen werden automatisch die global gültigen Operatoren *new* und *delete* verwendet.

```
37     ...
38     // CSWS_Point::operator new aufrufen
39     CSWS_Point *ptr = new CSWS_Point( 1, 1 );
40
41     // Aufruf ::operator new (globale Version), da nicht überladen
42     char *str = new char[20];
```

3.11 Überladen von Funktions-Member (statische Polymorphie)

Beim Design von Klassen kann die Situation entstehen, dass einer Funktion die Verarbeitung von mehreren Datentypen mehrfach implementiert werden müsste. Zur eindeutigen Unterscheidung der Funktionen musste der C-Programmierer in dieser Situation die Funktionen über den Namen "typisieren", was dazu führte, dass mehrere Funktionen mit verschiedenen Namen eigentlich die selbe Funktionalität aufwiesen.

C++ bietet hier die elegante Möglichkeit der statischen Polymorphie. Mit diesem Mechanismus bleibt der Funktionsname konstant, jedoch muss die Funktionssignatur eindeutig bleiben. Die Funktionssignatur setzt sich zusammen aus Funktionsnamen sowie Anzahl und Typ der Argumente.

Beispiel einer 5-fach definierten Funktionsüberladung:

```
1 void print( const char *str, int breite); // #1
2 void print( double d, int breite);      // #2
3 void print( long l, int breite );       // #3
4 void print( int i, int breite );        // #4
5 void print( const char *str );          // #5
```

Folgende Einschränkungen sind bei den Überlagerungen von Funktionen zu beachten:

Funktionen, die sich nur durch den Funktionswert unterscheiden, dürfen nicht überladen werden:

```
6 void Funktion( const char *str, int breite); // #1
7 int  Funktion( const char *str, int breite); // #2 -> Fehler!
```

Typ und Typ & sind als Typen formaler Argumente nicht unterschiedlich genug.

```
8 void Funktion( double ); // #1
9 void Funktion( double & ); // #2 -> Fehler!
```

Typ und const Typ sind als Typen formaler Argumente nicht unterschiedlich genug.

```
10 void Funktion( int ); // #1
11 void Funktion( const int ); // #2 -> Fehler!
```

Unterscheidungen von Typen in formalen Argumenten, die lediglich auf einem typedef basieren, führen ebenfalls nicht zu überladbaren Funktionen.

```
12 typedef unsigned char BYTE;
13 void Funktion( unsigned char ); // #1
14 void Funktion( BYTE ); // #2 -> Fehler!
```

Formale Argumente des Types T* und T[] werden in C++ nicht unterschieden.

```
15 void Funktion( char * ); // #1
16 void Funktion( char[] ); // #2 -> Fehler!
17 void Funktion( char[20] ); // #3 -> Fehler!
```

Selbstverständlich führen auch Kombinationen von obigen Fehldeklarationen zu unzulässigen Versuchen von Funktionsüberladung.

Alle deklarierten Funktionen müssen auch vollständig implementiert werden. Somit führt jede Überladung einer Funktion zu einer eigenständigen Funktion.

3.12 Spezielle Funktions-Member

Bei der Deklaration einer Klasse werden von C++ automatisch folgende vier Funktions-Member für diese Klasse deklariert:

- Standardkonstruktor
- Copy-Konstruktor
- Zuweisungsoperator
- Standarddestruktor

Die aufgezeigten Funktions-Member spielen für die Nutzung eines Objektes eine sehr wichtige Rolle, und werden deshalb in den nächsten Kapiteln näher beschrieben.

3.12.1 Konstruktoren

Jedesmal wenn ein Programm ein Datenelement anlegt, sollte es dieses auch initialisieren. C++ stellt für die Initialisierung während der Instanzierung von Objekten spezielle Konstruktorfunktionen zur Verfügung. Die Konstruktorfunktion (kurz: der Konstruktor) wird automatisch aufgerufen, wenn ein Objekt angelegt wird. Der Aufruf erfolgt unabhängig davon ob das Objekt statisch oder dynamisch instanziiert wird und grundsätzlich auch unabhängig davon ob ein Konstruktor vom Programmierer definiert wird oder nicht. Daraus kann geschlossen werden, dass C++ bei Nichtimplementierung eines Konstruktors selber einen Konstruktor verwendet (scheinbarer Defaultkonstruktor):

Allgemeine Schreibweise: **T::T()**

Für die Implementation von Konstruktoren sind folgende Regeln von Bedeutung:

Der Funktionsname des Konstruktors ist identisch mit dem Klassennamen.

Eine Klasse kann mehrere Konstruktoren besitzen. Diese müssen aber in Bezug auf den Typ bzw., die Anzahl Argumente verschieden sein (siehe auch Kapitel Funktionsoverload), damit der Compiler den richtigen Konstruktor wählen kann.

Ein Konstruktor darf keinen Rückgabetyt besitzen (auch nicht void).

Wird für eine Klasse kein Konstruktor definiert, wendet C++ einen sogenannten Defaultkonstruktor (auch Standardkonstruktor) an. Der Defaultkonstruktor ist ein Konstruktor der ohne Argumente aufgerufen wird.

Wird für eine Klasse ein Konstruktor definiert, wird der Defaultkonstruktor von C++ nicht gebildet. In diesem Fall muss die Implementation prüfen, ob der Standardkonstruktor als Overload realisiert werden muss.

```
1 // -----
2 // Deklaration der Klasse CSWS_Point
3 // -----
4 class CSWS_Point {
5     int  iX_Coord,           // X Koordinate des Punktes
6         iY_Coord;           // Y Koordinate des Punktes
7 public:
8     CSWS_Point( int, int ); // Konstruktor der Klasse Point
9     void Set( int, int );   // Deklaration der Klassenfunktion Set
10 };
11
12 // -----
13 // Implementation der Klasse CSWS_Point
14 // -----
15 CSWS_Point::CSWS_Point( int iX, int iY )
```

```

16 {
17     iX_Coord = iX;
18     iY_Coord = iY;
19 }
20
21 // Aufgrund obiger Klassendefinition können nun Objekte der Klasse
22 // CSWS_Point gleich mit gültigen Werten angelegt werden:
23 // -----
24 // Testprogramm
25 // -----
26 void main()
27 {
28     CSWS_Point ptPoint1( 100, 50 );
29 }

```

Untenstehendes Beispiel zeigt eine Klassendefinition mit 2 Konstruktoren:

```

30 class CSWS_Point {
31     int    iX_Coord,        // X Koordinate des Punktes
32     int    iY_Coord,        // Y Koordinate des Punktes
33     int    iZ_Coord,        // Z Koordinate des Punktes
34 public:
35     CSWS_Point( int, int );    // Zweidimensionale Version
36     CSWS_Point( int, int, int ); // Dreidimensionale Version
37     void Set( int, int );      // Deklaration der Klassenfunktion
38     void Set( int, int, int ); // Deklaration der Klassenfunktion
39 };

```

3.12.2 Der Copy-Konstruktor

Ein Copy-Konstruktor ist ein Konstruktor der aufgerufen werden kann, um ein Objekt zu kopieren. Dieser Fall tritt immer dann auf, wenn ein Objekt "by value" an eine Funktion übergeben wird, oder wenn ein Objekt bei der Instanzierung direkt durch eine Objektzuweisung initialisiert wird.

Der Copy-Konstruktor wird mit mindestens einem Argument aufgerufen. Das Minimalargument bildet eine Referenz auf das zu kopierende Objekt. Ein Copy-Konstruktor wird nur dann automatisch erzeugt, wenn kein Copy-Konstruktor explizit definiert wurde.

Allgemeine Schreibweise: **T::T(const T&)**

Solange die Daten in einem Objekt "by value" gespeichert sind, kann der default Copy-Konstruktor benutzt werden. Dieser kopiert den Inhalt des Quellobjektes ohne Modifikation in das Zielobjekt. Dies geht in der Regel solange gut, wie nur Objekte von Klassen kopiert werden in denen keine Referenzen oder Pointer definiert sind.

Trifft letztere Situation ein, muss beim Klassendesign geprüft werden, ob das Verhalten des Copy-Konstruktor abgeändert werden muss.

Werden innerhalb einer Klasse Daten per Pointer geführt, muss die Erstellung eines Copy-Konstruktors geprüft werden.

Wird für eine Klasse ein spezifischer Zuweisungsoperator implementiert, muss die Erstellung eines Copy-Konstruktors geprüft werden.

Bei Deklaration mit Zuweisung wird ein Copy-Konstruktor aufgerufen (Typ ObjB = ObjA).

Der default Copy-Konstruktor kopiert komponentenweise.

3.12.2.1 Verhalten des Standard-Copy-Konstruktor mit Daten-Member by value

Wird kein Copy-Konstruktor definiert, kopiert C++ bei einer Objektzuweisung die enthaltenen Daten 1:1 pro Daten-element vom Quellenobjekt in das Zielobjekt.

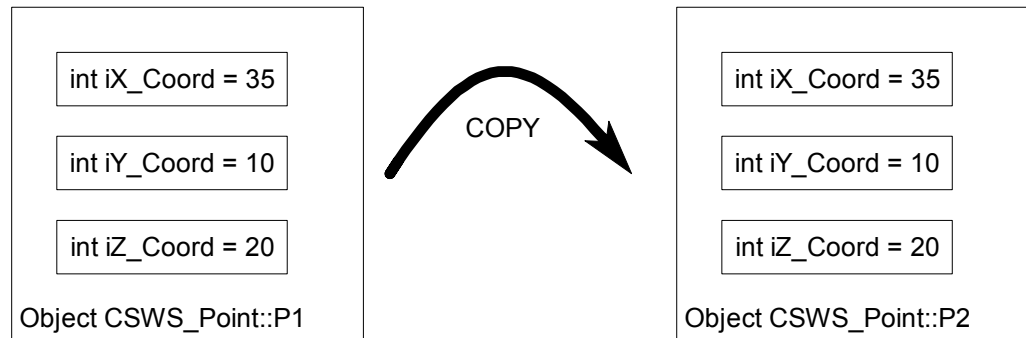


Abbildung 3-4 Schematische Wirkungsweise des Standard-Copy-Konstruktors für Objekte, die Daten-Member nur "by value" enthalten.

3.12.2.2 Verhalten des Standard-Copy-Konstruktor mit Daten-Member by reference

Wird kein Copy-Konstruktor definiert, kopiert C++ bei einer Objektzuweisung die enthaltenen Daten 1:1 pro Daten-element vom Quellenobjekt in das Zielobjekt. Dies führt bei Objekten mit enthaltenen Daten-Membere mit Zeigern dahin, dass sowohl das Original als auch das kopierte Objekt auf ein und den selben Speicherblock zeigen.

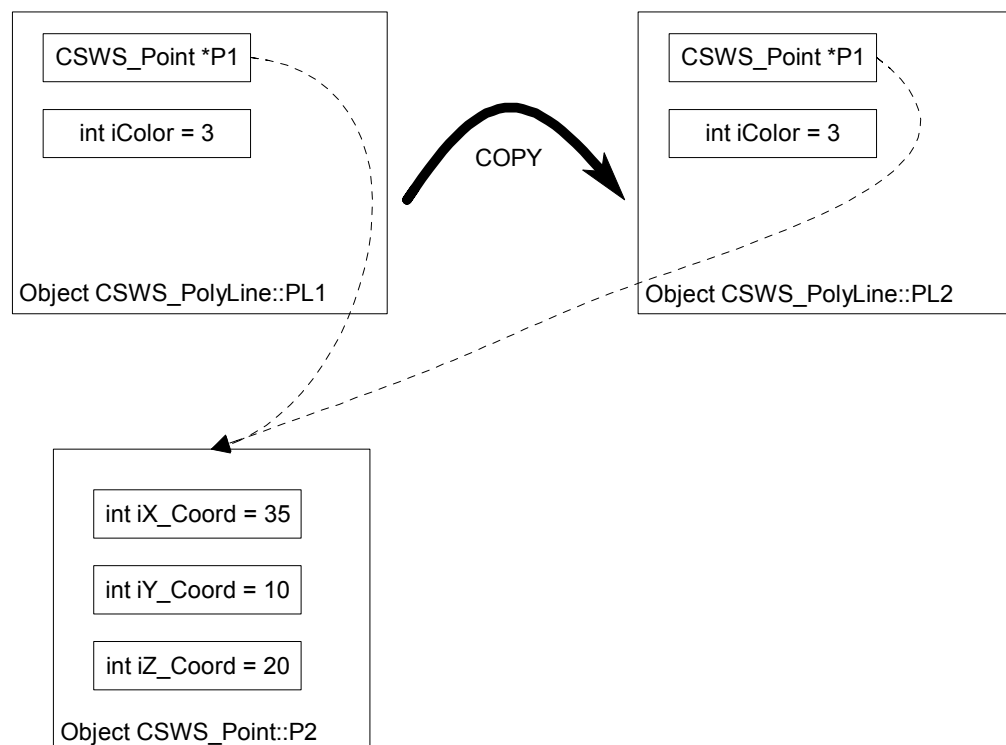


Abbildung 3-5 Schematische Wirkungsweise des Standard-Copy-Konstruktors für Objekte, die auch Daten-Member "by reference" enthalten.

Betrachten wir uns zu obiger Problematik folgenden Sourcecode:

```
1  // -----
2  // class declaration and implementation of point
3  // -----
4  class CSWS_Point {
5  // -----
6  public:
7      int iX_Coord;
8      int iY_Coord;
9      int iZ_Coord;
10
11     // Construction
12     CSWS_Point() : iX_Coord( 0 ), iY_Coord( 0 ), iZ_Coord( 0 ) {}
13 };
```

```

14 // -----
15 // class declaration and implementation of polyline
16 // -----
17 // -----
18 class CSWS_PolyLine {
19 // -----
20 public:
21     CSWS_Point *P1;
22     int         iColor;
23     int         iID;
24
25 public:
26     CSWS_PolyLine() { iColor = 0; P1 = new CSWS_Point(); }
27
28     // Funktionen
29     void SetID( int iID ) { CSWS_PolyLine::iID = iID; }
30     void List( void );
31     void Move( int x, int y, int z = 0 );
32 };
33
34 // -----
35 void CSWS_PolyLine::List( void ) {
36 // -----
37     cout << "Adress to Point is " << P1 << endl;
38     cout << "Punkt " << iID << ".x " << P1->iX_Coord << endl;
39     cout << "Punkt " << iID << ".y " << P1->iY_Coord << endl;
40     cout << "Punkt " << iID << ".z " << P1->iZ_Coord << endl << endl;
41 }
42
43 // -----
44 void CSWS_PolyLine::Move( int x, int y, int z ) {
45 // -----
46     P1->iX_Coord += x;
47     P1->iY_Coord += y;
48     P1->iZ_Coord += z;
49 }
50
51 // -----
52 // testprogramm
53 // -----
54 void main () {
55
56     // construct the first object
57     CSWS_PolyLine PL1;
58     PL1.SetID( 1 );
59     PL1.List();
60     PL1.Move( 10, 20 );
61     PL1.List();
62
63     // construct the second object using direct initiation
64     CSWS_PolyLine PL2 = PL1;
65     PL2.SetID( 2 );
66     PL1.List();
67     PL2.List();
68
69     // now move the line 1
70     PL1.Move( 50, 50 );
71     PL1.List();
72     PL2.List();
73 }

```

```

1  // -----
2  // generated output
3  // -----
4  Adress to Point is 0x00420E30
5  Punkt 1.x 0
6  Punkt 1.y 0
7  Punkt 1.z 0
8
9  Adress to Point is 0x00420E30
10 Punkt 1.x 10
11 Punkt 1.y 20
12 Punkt 1.z 0
13
14 Adress to Point is 0x00420E30
15 Punkt 1.x 10
16 Punkt 1.y 20
17 Punkt 1.z 0
18
19 Adress to Point is 0x00420E30
20 Punkt 2.x 10
21 Punkt 2.y 20
22 Punkt 2.z 0
23
24 Adress to Point is 0x00420E30
25 Punkt 1.x 60
26 Punkt 1.y 70
27 Punkt 1.z 0
28
29 Adress to Point is 0x00420E30
30 Punkt 2.x 60
31 Punkt 2.y 70
32 Punkt 2.z 0

```

3.12.2.3 Verhalten spezifischer Copy-Konstruktor: Daten-Member by reference

Sollte der oben aufgezeigte Effekt der gemeinsamen Referenzierung gewünscht sein, ist natürlich alles in Ordnung. Jedoch ist in der Praxis meistens nicht diese Konstellation als Resultat einer Kopie gewünscht, sondern die Konstellation, dass sowohl das Objekt als auch der im Objekt referenzierte Speicher kopiert wird. Wir nennen dies eine sogenannte "Deep Copy" (Kopieren von Datenstrukturen über Referenzgrenzen und Vererbungsstufen hinweg).

C++ ermöglicht das Erstellen von "Deep Copies" indem die Copy-Konstruktoren der Klassen überschrieben werden.

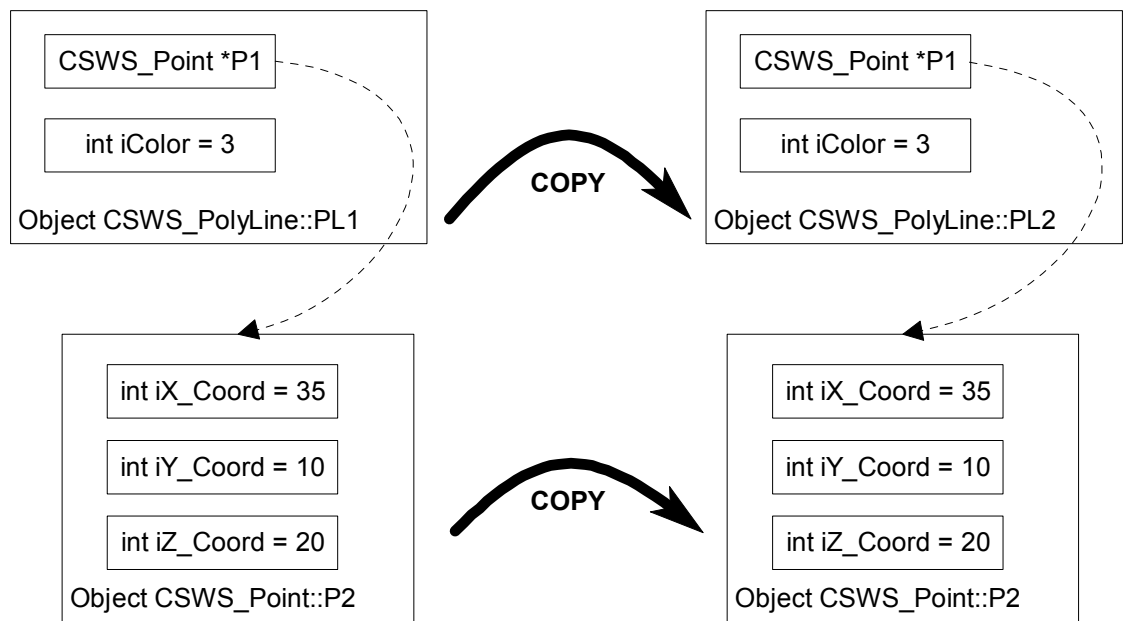


Abbildung 3-6 Schematische Darstellung der meistens gewünschten Wirkungsweise des Copy-Konstruktors für Objekte, die auch Daten-Member "by reference" enthalten.

```

1  // -----
2  // class declaration of point
3  // -----
4  class CSWS_Point {
5  public:
6      int iX_Coord;
7      int iY_Coord;
8      int iZ_Coord;
9
10     // Construction
11     CSWS_Point() : iX_Coord( 0 ), iY_Coord( 0 ), iZ_Coord( 0 ) {}
12 };
13
14 // -----
15 // class declaration and implementation of polyline
16 // -----
17 class CSWS_PolyLine {
18 public:
19     CSWS_Point *P1;
20     int         iColor;
21     int         iID;
22
23 public:
24     CSWS_PolyLine() { iColor = 0; P1 = new CSWS_Point(); }
25     CSWS_PolyLine( CSWS_PolyLine &plSource );
26
27     // Funktionen
28     void SetID( int iID ) { CSWS_PolyLine::iID = iID; }
29     void List( void );
30     void Move( int x, int y, int z = 0 );
31 };

```

```

32
33 // -----
34 CSWS_PolyLine::CSWS_PolyLine( CSWS_PolyLine &plSource ) {
35 // -----
36     iColor = plSource.iColor;
37     iID     = plSource.iID
38     P1 = new CSWS_Point;
39     P1->iX_Coord = plSource.P1->iX_Coord;
40     P1->iY_Coord = plSource.P1->iY_Coord;
41     P1->iZ_Coord = plSource.P1->iZ_Coord;
42 }
43
44 // -----
45 void CSWS_PolyLine::List( void ) {
46 // -----
47     cout << "Adress to Point is " << P1 << endl;
48     cout << "Punkt " << iID << ".x " << P1->iX_Coord << endl;
49     cout << "Punkt " << iID << ".y " << P1->iY_Coord << endl;
50     cout << "Punkt " << iID << ".z " << P1->iZ_Coord << endl << endl;
51 }
52
53 // -----
54 void CSWS_PolyLine::Move( int x, int y, int z ) {
55 // -----
56     P1->iX_Coord += x;
57     P1->iY_Coord += y;
58     P1->iZ_Coord += z;
59 }
60
61 // -----
62 // testprogramm
63 // -----
64 void main ()
65 {
66     // construct the first object
67     CSWS_PolyLine PL1;
68     PL1.SetID( 1 );
69     PL1.List();
70     PL1.Move( 10, 20 );
71     PL1.List();
72
73     // construct the second object using direct initiation
74     CSWS_PolyLine PL2 = PL1;
75     PL2.SetID( 2 );
76     PL1.List();
77     PL2.List();
78
79     // now move the line 1
80     PL1.Move( 50, 50 );
81     PL1.List();
82     PL2.List();
83 }

```

```

1  // -----
2  // generated output
3  // -----
4  Adress to Point is 0x00420E30
5  Punkt 1.x 0
6  Punkt 1.y 0
7  Punkt 1.z 0
8
9  Adress to Point is 0x00420E30
10 Punkt 1.x 10
11 Punkt 1.y 20
12 Punkt 1.z 0
13
14 Adress to Point is 0x00420E30
15 Punkt 1.x 10
16 Punkt 1.y 20
17 Punkt 1.z 0
18
19 Adress to Point is 0x00420E60
20 Punkt 2.x 10
21 Punkt 2.y 20
22 Punkt 2.z 0
23
24 Adress to Point is 0x00420E30
25 Punkt 1.x 60
26 Punkt 1.y 70
27 Punkt 1.z 0
28
29 Adress to Point is 0x00420E60
30 Punkt 2.x 10
31 Punkt 2.y 20
32 Punkt 2.z 0

```

3.12.3 Der Zuweisungsoperator

Die Zuweisung an Objekte einer Klasse wird mittels dem Zuweisungsoperator definiert. Es gilt dabei grundsätzlich die selben Regeln wie bei dem Copy-Konstruktor zu berücksichtigen.

Allgemeine Schreibweise: **T& T::operator=(const T&)**

Werden innerhalb einer Klasse Daten per Pointer geführt, muss die Erstellung eines Zuweisungsoperators geprüft werden.

Wird für eine Klasse ein spezifischer Copy-Konstruktor implementiert, muss die Erstellung eines Zuweisungsoperators geprüft werden.

Bei Deklaration mit Zuweisung wird ein Copy-Konstruktor aufgerufen (Typ ObjB = ObjA):

```

Typ ObjA;           // normaler Konstruktor
Typ ObjB = ObjA;    // Copy-Konstruktor

```

Bei getrennter Deklaration und anschliessender Zuweisung (Initialisierung), werden zuerst ein normaler Konstruktor und anschliessend ein Zuweisungsoperator benutzt:

```

Typ ObjA;           // normaler Konstruktor
Typ ObjB;           // normaler Konstruktor
ObjB = ObjA;        // Zuweisungsoperator

```

Der default Zuweisungsoperator weist komponentenweise zu.

```

1  // -----
2  // class declaration of point
3  // -----

```

```

4  class CSWS_Point {
5  public:
6      int iX_Coord;
7      int iY_Coord;
8      int iZ_Coord;
9
10     // Construction
11     CSWS_Point() : iX_Coord( 0 ), iY_Coord( 0 ), iZ_Coord( 0 ) {}
12 };
13
14 // -----
15 // class declaration and implementation of polyline
16 // -----
17 class CSWS_PolyLine {
18 public:
19     CSWS_Point *P1;
20     int         iColor;
21     int         iID;
22
23 public:
24     CSWS_PolyLine() { iColor = 0; P1 = new CSWS_Point(); }
25     CSWS_PolyLine( CSWS_PolyLine &plSource );
26     CSWS_PolyLine &operator= ( CSWS_PolyLine & plSource );
27
28     // Funktionen
29     void SetID( int iID ) { CSWS_PolyLine::iID = iID; }
30     void List( void );
31     void Move( int x, int y, int z = 0 );
32 };
33
34 // -----
35 CSWS_PolyLine::CSWS_PolyLine( CSWS_PolyLine &plSource ) {
36 // -----
37     iColor = plSource.iColor;
38     P1 = new CSWS_Point;
39     P1->iX_Coord = plSource.P1->iX_Coord;
40     P1->iY_Coord = plSource.P1->iY_Coord;
41     P1->iZ_Coord = plSource.P1->iZ_Coord;
42 }
43
44 // -----
45 CSWS_PolyLine & CSWS_PolyLine::operator= ( CSWS_PolyLine & plSource ) {
46 // -----
47     iColor = plSource.iColor;
48     iID     = plSource.iID;
49     delete P1;
50     P1 = new CSWS_Point;
51     P1->iX_Coord = plSource.P1->iX_Coord;
52     P1->iY_Coord = plSource.P1->iY_Coord;
53     P1->iZ_Coord = plSource.P1->iZ_Coord;
54     return *this;
55 }
56
57 // -----
58 void CSWS_PolyLine::List( void ) {
59 // -----
60     cout << "Adress to Point is " << P1 << endl;
61     cout << "Punkt " << iID << ".x " << P1->iX_Coord << endl;
62     cout << "Punkt " << iID << ".y " << P1->iY_Coord << endl;
63     cout << "Punkt " << iID << ".z " << P1->iZ_Coord << endl << endl;
64 }
65
66 // -----
67 void CSWS_PolyLine::Move( int x, int y, int z ) {
68 // -----
69     P1->iX_Coord += x;
70     P1->iY_Coord += y;

```

```

71     P1->iZ_Coord += z;
72 }
73
74 // -----
75 // testprogramm
76 // -----
77 void main () {
78
79     // construct the first object
80     CSWS_PolyLine PL1;
81     PL1.SetID( 1 );
82     PL1.List();
83     PL1.Move( 10, 20 );
84     PL1.List();
85
86     // construct the second object using indirect initiation
87     CSWS_PolyLine PL2;
88     PL2 = PL1;
89     PL1.List();
90     PL2.List();
91
92     // now move the line 1
93     PL2.SetID( 2 );
94     PL1.Move( 50, 50 );
95     PL1.List();
96     PL2.List();
97 }

98 // -----
99 // generated output
100 // -----
101 Adress to Point is 0x00420E30
102 Punkt 1.x 0
103 Punkt 1.y 0
104 Punkt 1.z 0
105
106 Adress to Point is 0x00420E30
107 Punkt 1.x 10
108 Punkt 1.y 20
109 Punkt 1.z 0
110
111 Adress to Point is 0x00420E30
112 Punkt 1.x 10
113 Punkt 1.y 20
114 Punkt 1.z 0
115
116 Adress to Point is 0x00420E90
117 Punkt 1.x 10
118 Punkt 1.y 20
119 Punkt 1.z 0
120
121 Adress to Point is 0x00420E30
122 Punkt 1.x 60
123 Punkt 1.y 70
124 Punkt 1.z 0
125
126 Adress to Point is 0x00420E90
127 Punkt 2.x 10
128 Punkt 2.y 20
129 Punkt 2.z 0

```

3.12.4 Destruktoren

Die Umkehrfunktion eines Konstruktors kann mit Hilfe des Destruktors eines Objektes erreicht werden. Analog des automatischen Aufrufes des Konstruktors beim Erstellen eines Objektes, wird der Destruktor beim Eliminieren eines Objektes durch C++ automatisch aufgerufen. Die Funktionalität des Destruktors wird sinngemäss durch den Programmierer bestimmt.

Allgemeine Schreibweise: `T::~~T()`

Der Funktionsname des Destruktors ist identisch mit dem Klassennamen mit einem Tilde-Zeichen davor (~ wird auch als Komplement des Konstruktors bezeichnet).

Ein Destruktor erhält keine Argumente und es wird ebenfalls kein Funktionswert angegeben (nicht einmal void).

Destruktoren können nicht vererbt werden.

Da der Rumpf eines Destruktors ausgeführt wird bevor die Destruktoren der eingebetteten Datenelemente aufgerufen werden, können die Datenelemente innerhalb des Destruktors noch verwendet werden.

In Vererbungshierarchien sind Destruktoren als virtuell zu deklarieren.

Untenstehendes Beispiel zeigt eine Klassendeklaration mit einem Destruktor.

```
1  class CSWS_PolyLine {
2  public:
3      CSWS_Point *P1;
4      int         iColor;
5      int         iID;
6
7  public:
8      CSWS_PolyLine() { iColor = 0; P1 = new CSWS_Point(); }
9      CSWS_PolyLine( CSWS_PolyLine &plSource );
10     CSWS_PolyLine & operator= ( CSWS_PolyLine & plSource );
11     ~CSWS_PolyLine() { delete P1; }
12     ...
13 };
```

Konstruktoren und Destruktoren finden insbesondere bei komplizierteren Klassendefinitionen mit dynamischer Speicherbelegung eine sehr sinnvolle Anwendung. So kann zum Beispiel mit dem Konstruktor automatisch ein sinnvoller Anfangszustand für die Erstellung einer verketteten Liste erstellt werden. Der Destruktor kann aufgrund dessen so hergestellt werden, dass er eine beliebige verkettete Liste dynamisch angelegter Objekte wieder löscht.

Selbstverständlich müssen sowohl Konstruktor als auch Destruktor als normale C++ Funktion implementiert werden.

3.13 Friend Funktionen

Im untenstehenden Codefragment ist ein Operator + definiert. Mit dessen Hilfe kann ein Punkt (Objekt der Klasse CSWS_Point) um Distanz X verschoben werden.

```
14 // friend.cpp : Definiert den Einsprungpunkt für die Konsolenanwendung.
15 // -----
16
17 #include "stdafx.h"
18 #include <iostream>
19 using namespace std;
20
21 // -----
22 class CSWS_Point {
23 // -----
24     int iX_Coord;
25     int iY_Coord;
26
27 public:
28     CSWS_Point operator+ ( CSWS_Point A );
29     CSWS_Point( int iX_Coord = 0, int iY_Coord = 0 )
30     {
31         CSWS_Point::iX_Coord = iX_Coord;
32         CSWS_Point::iY_Coord = iY_Coord;
33     }
34
35     void List( void );
36 };
37
38 // -----
39 void CSWS_Point::List( void )
40 // -----
41 {
42     cout << "X Koordinate = " << iX_Coord << endl;
43     cout << "Y Koordinate = " << iY_Coord << endl << endl;
44 }
45
46 // -----
47 CSWS_Point CSWS_Point::operator+ ( CSWS_Point A )
48 // -----
49 {
50     return CSWS_Point( iX_Coord + A.iX_Coord, iY_Coord + A.iY_Coord );
51 }
52
53 // -----
54 void main()
55 // -----
56 {
57     CSWS_Point P1;
58     CSWS_Point P2;
59
60     P1.List();
61     P2 = P1;
62     P2.List();
63     P2 = P1 + 25;
64     P2.List();
65 }
```

In Zeile 63 des letzten Codefragmentes wird der Umstand des Standardcastings genutzt. Der Compiler führt eigentlich die Zeile ...

P2.operator=(P1.operator+(CSWS_Point(25));²

... aus. Aufgrund der reinen Logik sollte die Zeile auch als ...

P2 = 25 + P1;

... codiert werden können. Dies würde folgende explizite Zeile ergeben:

P2.operator=(25.operator+(P1);

Leider ergibt diese Art der Codierung nur Compilerfehler, da ein fundamentaler Datentyp wie int keine Elementfunktion haben kann.

Das Problem liegt darin, dass eine automatische Typenumwandlung nur für Parameter möglich ist. In der vorliegenden Form der Implementierung des Operators + ist der erste Wert das Objekt selber und nicht ein Parameter. Würden wir den Umkehrversuch explizit casten, würde es natürlich auch funktionieren:

P2 = CSWS_Point(25) + P1;

Um das Problem aber allgemein zu lösen, kann mit einer globalen Funktion, die beide Objekte als Parameter übernimmt, gearbeitet werden.

CSWS_Point operator+ (CSWS_Point A, CSWS_Point B);

Eine solche Funktion verliert nun aber den Zugriff auf die internen, privaten Variablen der Objekte. Um die Funktion effizient zu gestalten, kann diese in der eigentlichen Klasse als Friend-Funktion deklariert werden. Diese Deklaration öffnet der globalen Funktion den Zugriff auf die Internas der Klasse.

Die gesamte Implementation präsentiert sich wie folgt:

```
1 // friend.cpp : Definiert den Einsprungpunkt für die Konsolenanwendung.
2 // -----
3 #include "stdafx.h"
4 #include <iostream>
5 using namespace std;
6
7 // -----
8 // Predeclarations
9 // -----
10 class CSWS_Point;
11 CSWS_Point operator+ ( CSWS_Point &A, int B );
12 CSWS_Point operator+ ( int A, CSWS_Point &B );
13
14 // -----
15 // class declaration
16 // -----
17 class CSWS_Point {
18     int iX_Coord;
19     int iY_Coord;
20
21 public:
22     CSWS_Point( int iX_Coord = 0, int iY_Coord = 0 );
23     void List( void );
24     friend CSWS_Point operator+ ( CSWS_Point &A, int B );
```

2 ² Nur erklärende Art und Weise der Codedarstellung. Kann nicht explizit so geschrieben werden.

```

25     friend CSWS_Point operator+ ( int A, CSWS_Point &B );
26 };
27
28 // -----
29 // class implementation
30 // -----
31 CSWS_Point::CSWS_Point( int iX_Coord, int iY_Coord )
32 // -----
33 {
34     CSWS_Point::iX_Coord = iX_Coord;
35     CSWS_Point::iY_Coord = iY_Coord;
36 }
37
38 void CSWS_Point::List( void )
39 // -----
40 {
41     cout << "X Koordinate = " << iX_Coord << endl;
42     cout << "Y Koordinate = " << iY_Coord << endl << endl;
43 }
44
45 // -----
46 // global implementations
47 // -----
48 CSWS_Point operator+ ( CSWS_Point &A, int B)
49 // -----
50 {
51     return CSWS_Point( A.iX_Coord + B, A.iY_Coord );
52 }
53
54 CSWS_Point operator+ ( int A, CSWS_Point &B)
55 // -----
56 {
57     return CSWS_Point( B.iX_Coord + A, B.iY_Coord );
58 }
59
60 // -----
61 // testprogramm
62 // -----
63 void main()
64 {
65     CSWS_Point P1( 1, 2 );
66     CSWS_Point P2;
67
68     P1.List();
69     P2 = P1;
70     P2.List();
71     P2 = P1 + 25;
72     P2.List();
73     P2 = 50 + P1;
74     P2.List();
75 }

```

Nebst obiger funktionsweiser Definition von Freunden, können ganze Klassen als Friends definiert werden, so dass von der einen Klasse direkt in den gekapselten Bereich der andern Klasse durchgegriffen werden kann.

Syntax zur Deklaration einer Friend-Klasse:

```
1  class CSWS_Point {
2  // Class datamember
3  private:
4      int iX_Coord, // X coordinate of point
5          iY_Coord, // Y coordinate of point
6          iZ_Coord; // Z coordinate of point
7  static int iCount; // count of used Objects
8
9  // class friend definitions
10     friend class CSWS_Line;
11
12 // private functionmember
13 public:
14     CSWS_Point operator+ ( CSWS_Point P )
15     { return CSWS_Point( iX_Coord + P.GetX(),
16                         iY_Coord + P.GetY(),
17                         iZ_Coord + P.GetZ()); }
18     CSWS_Point operator- ( CSWS_Point P )
19     { return CSWS_Point( iX_Coord - P.GetX(),
20                         iY_Coord - P.GetY(),
21                         iZ_Coord - P.GetZ()); }
22
23 // class interface - functionmember
24 public:
25     // Constructors with inline implementation
26     CSWS_Point( void ) { iX_Coord=0; iY_Coord=0; iZ_Coord=0; iCount++; }
27
28     // Interfacefunctions for manipulate data
29     void Set( int X, int Y ){ iX_Coord = X; iY_Coord = Y; iZ_Coord = 0; }
30     void Set( int X, int Y, int Z ){ iX_Coord=X; iY_Coord=Y; iZ_Coord=Z; }
31     int  GetX  ( void ) { return iX_Coord; }
32     int  GetY  ( void ) { return iY_Coord; }
33     int  GetZ  ( void ) { return iZ_Coord; }
34     CSWS_Point Get ( void ){ return CSWS_Point( iX_Coord,
35                                                  iY_Coord,
36                                                  iZ_Coord); }
37 };
```

Ein Funktions-Member arbeitet mit dem Objekt, von dem es aufgerufen wurde. Eine Friend-Funktion dagegen arbeitet mit dem Objekt oder den Objekten, die an sie als Argument übergeben wurden.

Wenn eine Friend-Funktion nicht Member einer Klasse ist, so darf in der Funktion das implizite Member-Argument this nicht benutzt werden. Es ist dann undefiniert!

Friend-Definitionen werden durch Zugriffsspezifizierer nicht beeinflusst.

Friend-Beziehungen werden weder vererbt noch sind sie transitiv.

4 Verhältnisse zwischen Klassen

In der realen Welt lassen sich Klassen nur sehr selten komplett isolieren. Meistens ist es schon gar nicht sinnvoll, Versuche zur absoluten Isolierung von Klassen durchzuführen. Die Beziehungen die Klassen untereinander verbinden, können von sehr verschiedenartiger Natur sein. C++ unterstützt methodisch und syntaktisch mehrere Beziehungsarten. Dieses Kapitel zeigt welche Verhältnisse mit C++ wie abgedeckt werden können.

4.1 Grundsätzliche Überlegungen

Bei der informationstechnischen Analyse einer Aufgabenstellung erkennt der Spezialist oft sehr früh, dass sich die Daten nicht in einer Tabelle oder Klasse versorgen lassen. Aufgabenstellungen ohne Anwendung der Relationstheorie zur Isolierung sinnvoller Klassen werden wohl nur in den Lehrbüchern anzutreffen sein. Das Kapitel zeigt zwar die C++ spezifischen methodischen und syntaktischen Verhältnisse bei gegebener Klassifizierung von Daten, ich erinnere jedoch daran, dass die Anwendung der unten beschriebenen Verhältnisse erst zum Zug kommt, wenn der grundsätzliche Analyse- und Design-Prozess eines Projektes fertig abgeschlossen ist. Selbstverständlich können Klassen jederzeit erweitert werden, ohne dass die ganzen Anwendungen, die die Klassen benützen, neu geschrieben werden müssen. Jedoch dürfte eine einfache Erweiterung einer Klassensammlung schwierig sein, wenn von Anfang an grobe Designfehler mit eingebaut wurden.

Der Ansatz der objektorientierten Softwareentwicklung zur Darstellung von Beziehungen kann am ehesten mit der Relationenbildung verglichen werden. Der wesentliche Unterschied besteht dabei darin, dass die OO-Technik auch die Funktion einer Information in die Relation mit einbezieht. So spricht der OO-Designer eben von einer Klasse (die in sich Daten und Funktionen hat) die in Relation zu einer anderen Klasse (mit anderen Daten und Funktionen) steht.

Eine Sammlung von Klassen die in mehr oder weniger engen Relationen zueinander stehen, wird als Klassenbibliothek bezeichnet. Klassenbibliotheken können selbstverständlich selber hergestellt werden. Sie dienen zur Lösung eines spezifischen Problems. Immer öfter werden jedoch auch Klassenbibliotheken am Markt beschafft, die bestimmte, oft wiederkehrende Aufgabenstellungen in einer allgemeinen Art und Weise lösen.

4.2 Vererbung

Die revolutionärste Möglichkeit der objektorientierten Technik ist die Vererbung von Information und Funktion einer Klasse an eine andere. Die Technik der Vererbung ist sehr mächtig und oft angewendet. Keine mir bekannte Klassenbibliothek kommt ohne die Vererbung aus.

Beim Vorgang "Vererbung" wird eine Eltern - Kind Beziehung hergestellt. Dabei spricht der C++ Entwickler von der Basisklasse (auch Elternklasse oder Oberklasse) und der abgeleiteten Klasse (auch Kindklasse oder Unterklasse). Der Vererbungsvorgang kann über die Zugriffsberechtigung so gesteuert werden, dass die abgeleitete Klasse alle oder nur Teile der Eigenschaften der Basisklasse erbt.

Besteht eine Vererbungsbeziehung aus einem Elternteil und einem Kindteil, sprechen wir von einer sogenannten einfachen Vererbungsstruktur. Eine Beziehung zwischen mehreren Elternteilen und einem Kindteil, wird eine Mehrfachvererbung genannt. Vererbungen können sich über mehrere Generationen hinziehen.

Die Technik der Vererbung birgt folgende Vorteile in sich:

- Basisklassen können einzeln implementiert und getestet werden
- Grössere Wiederverwendbarkeit von Code
- Raschere Implementationszeit

4.2.1 Ableiten von Klassen

Für die Ableitung einer Klasse von einer Basisklasse gilt folgende Syntax:

```
class AbgeleiteteKlasse : [ Zugriffsspezifizierer ] Basisklasse
{
    Klassendeklaration
};
```

dabei gelten:

Abgeleitete Klasse:	Bezeichner der neuen, abgeleiteten Klasse
Zugriffsspezifizierer:	Definition der Umsetzung von Zugriffsrechten (optional)
Basisklasse:	Bezeichner der Basisklasse
Klassendeklaration:	Klassendeklaration der neuen abgeleiteten Klasse, diese geschehen zusätzlich zu den Definitionen der Basisklasse.

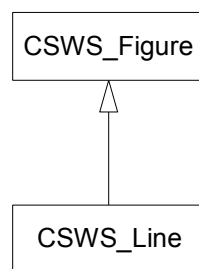


Abbildung 4-1 Ableitung einer Klasse von einer Basisklasse (Darstellung nach UML).

Umsetzung obiger Darstellung in C++ Code:

```
1  // -----
2  // Class CSWS_Figure
3  // -----
4  class CSWS_Figure {
5  // Datamember
6  protected:
7      int iLineColor; // color of the border or line for the element
8      int iObjectType; // objecttype used to build persistance
9
10 // Function Member
11 public:
12     // constructor
13     CSWS_Figure();
14     ...
15 };
16
17 // -----
18 // Class CSWS_Line
19 // -----
20 class CSWS_Line : CSWS_Figure {
21
22 // Class datamember
23 private:
24     int iX1_Coord;
25     int iY1_Coord;
```

```

26     int        iX2_Coord;
27     int        iY2_Coord;
28
29 // class interface - functionmember
30 public:
31
32     // Constructors
33     ....
34 };

```

Die kaskadierte Ableitung von Klassen ist möglich. Insbesondere bei Klassenbibliotheken von Compiler- oder Drittherstellern wird von der Ableitung von ganzen Klassenhierarchien rege Gebrauch gemacht. Für den Entwickler werden die genauen Kenntnisse der Möglichkeiten und Verknüpfungen beim Gebrauch von solchen Bibliotheken zur absoluten Notwendigkeit, um eine Implementation fachgerecht und rationell durchzuführen.

Untenstehende Abbildung zeigt ein Beispiel einer Klassenhierarchie (Application Architecture Classes of Visual C++ 2.2) mit einfacher Vererbung.

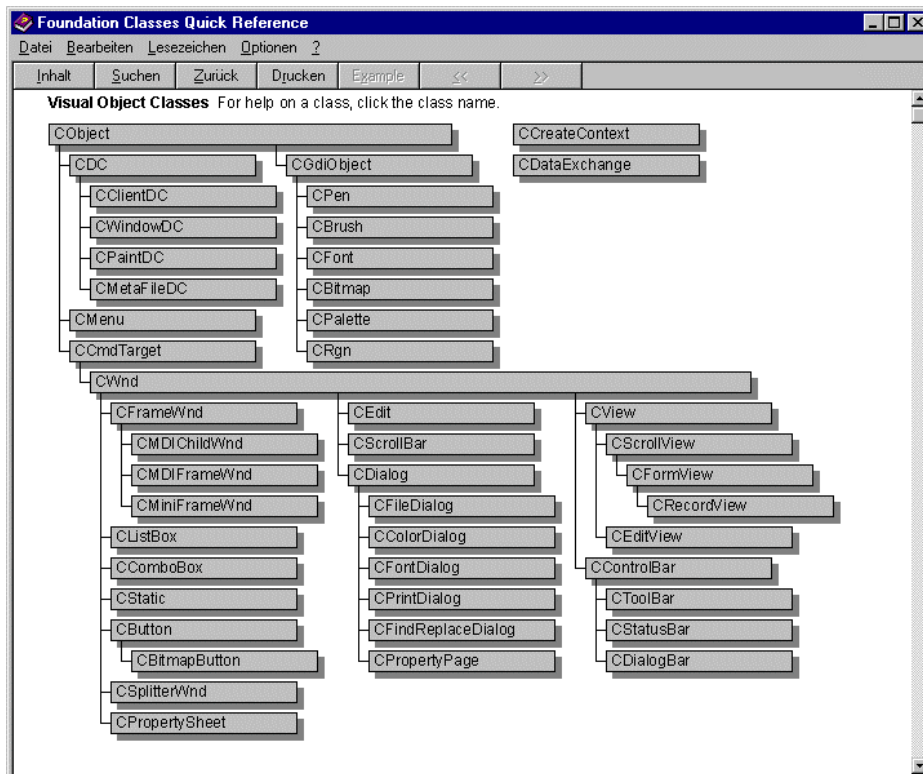


Abbildung 4-2 Abbildung eines Ausschnittes aus einer Klassenbibliothek (Sicht Vererbung).

4.2.2 Vererbung von Zugriffsrechten

Beim Ableiten einer Klasse von einer Basisklasse werden die Daten- und Funktions-Member an die abgeleitete Klasse vererbt. Bei diesem Vorgang entstehen eigentlich zwei verschiedene Mengen von Informationen und Funktionen:

- Die Menge der in der abgeleiteten Klasse deklarierten Member
- Die Menge der von der oder den Basisklassen geerbten Member

Für die in der abgeleiteten Klasse deklarierten Member gelten die normalen Regeln für Zugriffsrecht (Siehe Kapitel 3: Zugriffsrechte und Geltungsbereich). Für die Menge der in der Basisklasse deklarierten Member gelten spezielle Regeln, die über den Zugriffsspezifizierer bei der Vererbung beeinflusst werden.

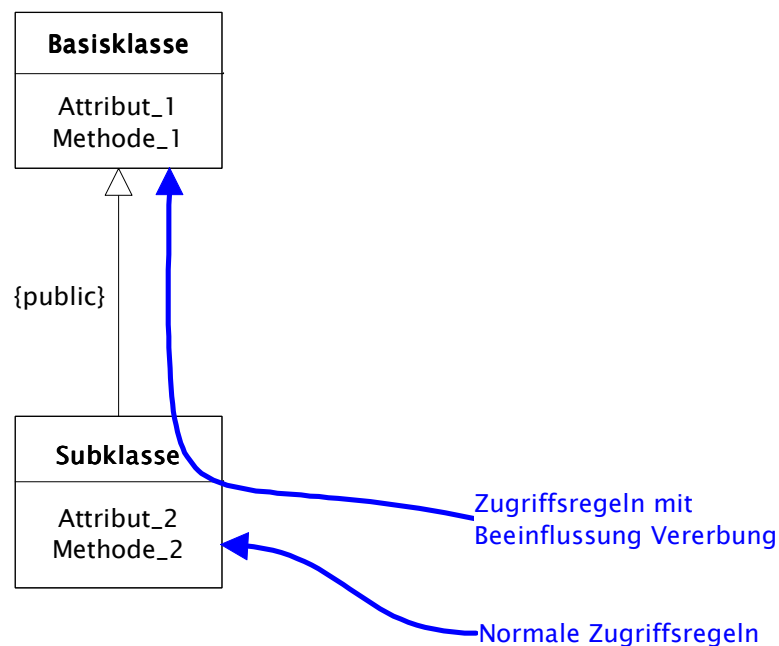


Abbildung 4-3 Prinzip der Zugriffsregeln bei der Vererbung.

Das untenstehende Codefragment zeigt einen Zugriffsspezifizierer der bei der Deklaration der Vererbung eingesetzt ist. Die hier gezeigte Art entspricht gleichzeitig der am häufigsten vorkommenden Art der Ableitung (public ist Standardfall).

```

1 // Class CSWS_Line
2 // -----
3 class CSWS_Line : public CSWS_Figure {
4
5 // Class datamember
6 private:
7     int iX1_Coord, iY1_Coord, iX2_Coord, iY2_Coord;
8
9 // class interface - functionmember
10 public:
11     // Constructors
12     ....
13 };

```

Der in der Vererbung eingebrachte Zugriffsspezifizierer nimmt Einfluss auf das Zugriffsrecht der Member in den Basisklassen. Dabei wird das effektive Zugriffsrecht aus einer logischen Verknüpfung des Rechtes des Members in der Basisklasse selber und des Rechtes der Ableitung. Es sind folgende Regeln definiert:

Bei einer Ableitung einer Klasse "Ableitung" von einer Klasse "Basis" mit dem *public* Spezifizierer, werden die *public* Member von "Basis" so nach "Ableitung" vererbt, als seien sie in "Ableitung" *public* definiert, respektive werden die *protected* Member von der Klasse "Basis" so nach "Ableitung" vererbt, als seien sie *protected* Member in "Ableitung".

Bei einer Ableitung einer Klasse "Ableitung" von einer Klasse "Basis" mit dem *protected* Spezifizierer, werden die *public* und die *protected* Member von "Basis" so nach "Ableitung" vererbt, als seien sie in "Ableitung" *protected* definiert.

Bei einer Ableitung einer Klasse "Ableitung" von einer Klasse "Basis" mit dem *private* Spezifizierer, werden die *public* und die *protected* Member von "Basis" so nach "Ableitung" vererbt, als seien sie in "Ableitung" *private* definiert.

In allen obigen Fällen sind die *private* Member von "Basis" für alle abgeleiteten Klassen nicht zugänglich (Ausnahme: Mittels Friend-Funktion in Basis).

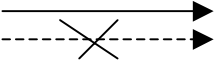
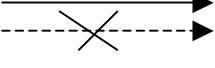
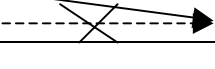
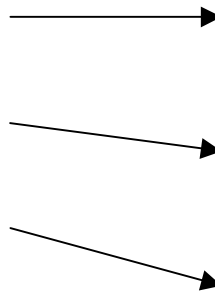
Ableitung	Basisklasse		abgeleitete Klasse
public	public protected private		public protected no access
protected	public protected private		protected no access
private	public protected private		private

Tabelle 4-1 Darstellung der Zusammenhänge beim Ableiten von Klassen mit Zugriffsspezifikation.



4.2.3 Mehrfachvererbung

Analog den obigen Erklärungen zur Ableitung von Klassen, können Klassen von mehreren Basisklassen abgeleitet werden. Die allgemeine Syntax dazu lautet:

```
class AbgeleiteteKlasse :      [ Zugriffsspezifizierer ] Basisklasse_1
                               [, [ Zugriffsspezifizierer ] Basisklasse_2 ..]
                               [, [ Zugriffsspezifizierer ] Basisklasse_N]
{
    Klassendeklaration
};
```

dabei gelten:

Abgeleitete Klasse:	Bezeichner der neuen, abgeleiteten Klasse
Zugriffsspezifizierer:	Zugriffsspezifikation zur Basisklasse
Basisklasse_1:	Bezeichner der ersten Basisklasse
Basisklasse_2:	Bezeichner der zweiten Basisklasse
Basisklasse_N:	Bezeichner der n-ten Basisklasse
Klassendeklaration:	Klassendeklaration der neuen abgeleiteten Basisklasse, diese geschehen zusätzlich zu den Definitionen der Basisklassen.

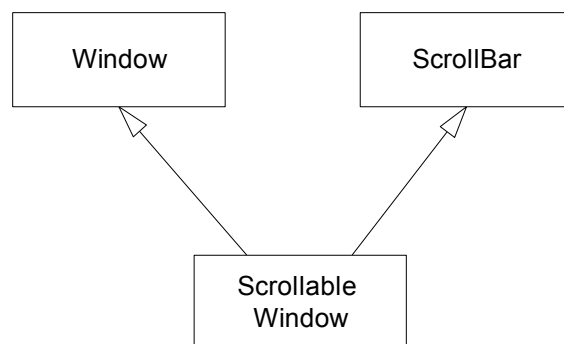


Abbildung 4-4 Beispiel einer Klassenhierarchie mit Mehrfachvererbung.

Umsetzung obiger Darstellung in C++ Code:

```
1 class ScrollableWindow : public Window, public ScrollBar
2 {
3     public:
4         ScrollableWindow( ... );
5         ~ScrollableWindow();
6
7     protected:
8         ....
9 };
```

Die Mehrfachvererbung kompliziert die Regeln zum Ansprechen von Funktions-Member dadurch, dass Aufrufe von Membern unter Umständen nicht mehr eindeutig sind. Dies ist oft eine Frage des Design. Beim Einsatz von Klassenbibliotheken lässt sich dieser Umstand jedoch nicht zwingend vermeiden.

Mehrdeutigkeit von Funktions-Member:

```
10 // Definition einer ersten Basisklasse
11 class Window {
12     // ...;
13
14 public:
15     void Move ( int x, int y );
16 };
17
18 // Definition einer zweiten Basisklasse
19 class ScrollBar {
20     ...
21 public:
22     void Move ( int x, int y );
23 };
24
25 // Definition einer abgeleiteten Klasse
26 class ScrollableWindow : public Window, public ScrollBar {
27     // ...
28 };
```

Die abgeleitete Klasse ScrollableWindow enthält nun beide Move-Funktionen der Basisklassen. Beim Aufruf nach folgendem Muster

```
29 // Definition eines Objektes der Klasse: ScrollableWindow
30 ScrollableWindow Scr1Wnd;
31
32 // verschieben des Fensters
33 Scr1Wnd.Move( 10, 10 );
```

... ist für den Compiler nicht klar, welche der beiden Funktionen Move gemeint ist. Diesem Umstand kann Rechnung getragen werden, indem ein expliziter Aufruf der Funktion durchgeführt wird.

```
34 // verschieben des Fensters
35 Scr1Wnd.Window::Move( 10, 10 );
36 Scr1Wnd.ScrollBar::Move( 10, 10 );
```

Einfacher ist es in der abgeleiteten Klasse eine neue Funktion zu definieren, die obige Schreibweise in sich vereint.

```
37 // Definition einer abgeleiteten Klasse
38 class ScrollableWindow : public Window, public ScrollBar {
39     ...
40 public:
41     // ...
42     void Move ( int x, int y ){ Window::Move( x, y );
43                               ScrollBar::Move( x, y ); }
44 };
```

4.2.4 Funktionsüberschreibung und -überladungen in Vererbungen

Es kann durchaus vorkommen, dass einzelne, vererbte Funktions-Member von abgeleiteten Klassen nicht exakt der geforderten Funktionalität der neu gebildeten Kindklasse entsprechen. Für diese Fälle bietet C++ die Möglichkeit, Funktions-Member von Basisklassen in den vererbten Klassen zu überladen oder zu überschreiben, so dass beim Gebrauch einer entsprechenden Funktion automatisch die tatsächliche, der abgeleiteten Klasse entsprechende Funktion, ausgeführt wird. Das Überschreiben wird in C++ von zwei verschiedenen Mechanismen unterstützt:

- das statische Überladen (statische Polymorphie)
- das statische Überschreiben (Override)

- das dynamische Überschreiben (dynamische Polymorphie)

4.2.4.1 Statische Funktionsüberladung in Vererbungen

Das Überladen von Funktionen in Vererbungsstrukturen ist den selben Regeln unterworfen wie das statische Überladen innerhalb einer Klasse. Mischungen zwischen Überladungen in der einen oder andern Klasse der Vererbung, respektive Kombinationen davon sind möglich.

4.2.4.2 Funktionsüberschreibung

Das statische Überschreiben einer Methode erfolgt dann, wenn die Funktionssignatur in der Subklasse exakt der Funktionssignatur in der Basisklasse entspricht¹.

```

1  // -----
2  class Base
3  // -----
4  {
5  ...
6  public:
7      void Func( int x, int y );
8      void FuncBase( void ){}
9  ...
10 };
11
12 void Base::Func( int x, int y ){
13 // -----
14 ...
15 }
16
17 // -----
18 class Inherit: public Base
19 // -----
20 {
21 ...
22 void Func( int x, int y );
23 void FuncInherit( void ){}
24 ...
25 };
26
27 void Inherit::Func( int x, int y ){
28 // -----
29 ...
30 }
31
32 // -----
33 // Testprogramm
34 // -----
35 void main () {
36 // -----
37 ...
38     Inherit objInherit;
39     objInherit.Func( 10, 23 );           // call Inherit::Func
40     objInherit.Base::Func( 10, 23 );    // call Base::Func (explicit)
41     objInherit.FuncBase();              // call Base::FuncBase
42     objInherit.FuncInherit();           // call Inherit::FuncInherit
43 }

```

¹ Entgegen der statischen Polymorphie, bei welcher die Funktionssignatur sich eindeutig von derjenigen der Basisklasse respektive der andern Funktionen mit dem selben Namen innerhalb der gleichen Klasse unterscheiden muss.

Bei der oben dargestellten Methode gibt es keine dynamisch zu interpretierenden Möglichkeiten einer Funktionsverwendung. Es ist zum Zeitpunkt der Codierung jeweils klar wie sich der Compiler und somit auch der fertige Code verhalten wird.

4.2.4.3 Dynamische Funktionsüberladung in Vererbungen

In Situationen in denen zum Zeitpunkt der Codierung zwar bekannt ist, dass Objekte verwendet werden, die von einer bestimmter Basisklasse abstammen, jedoch nicht genau bekannt ist, welcher konkreten Klasse sie angehören, reicht die statische Funktionsüberladung oft nicht aus.

Nehmen wir an, in unserem Zeichnungsbeispiel sollen die geometrischen Figuren (Linien, Rechtecke, Ellipsen) einer ganzen Zeichnung ausgegeben werden. Natürlich könnten wir bei der Erfassung der Figuren diese nach Ihrem Typ ordnen und dann entsprechend die Zeichnungsfunktionen sortiert nach Typ benutzen. Damit könnten wir Eindeutigkeit für die funktionelle Programmierung erreichen.

Aus anderer Sicht wäre es natürlich elegant, wenn die genaue Festlegung der Figurenreihenfolge aus Sicht des Codes nicht relevant wäre, sozusagen durch den Benutzer gesteuert in beliebiger Reihenfolge Figuren erzeugt werden könnten, und diese entsprechend ihrer Eingabefolge auch wieder ausgegeben werden könnten.

C++ unterstützt letztere Anforderung durch die Implementation von Code mittels dynamischer Polymorphie. Für die Realisierung dieser Technik müssen folgende Voraussetzungen geschaffen werden:

- Eine Vererbungsstufe
- Deklaration einer virtuellen Funktion in der Basisklasse
- Instanziierung der effektiven Nutzobjekte
- Ansprechen der gewünschten Funktionalität der Nutzobjekte über deren Basisklasse

```
1  // declaration of class CSWS_Figure
2  // -----
3  class CSWS_Figure {
4  ...
5  public:
6      virtual void Draw();
7      ...
8  };
9
10 // Implementation of draw a figures
11 // -----
12 void CSWS_Figure::Draw() {
13     // nothing to do, because it's only a abstract
14 }
```

```

15
16 // declaration of class CSWS_Ellipse
17 // -----
18 class CSWS_Ellipse : public CSWS_Figure {
19 ...
20 public:
21     void Draw();
22     ...
23 };
24
25 // Implementation of draw a ellipse
26 // -----
27 void CSWS_Ellipse::Draw() {
28     // code to draw a ellipse
29 }
30
31 // declaration of class CSWS_PolyLine
32 // -----
33 class CSWS_PolyLine : public CSWS_Figure {
34 ...
35 public:
36     void Draw();
37     ...
38 };
39
40 // Implementation to draw a PolyLine
41 // -----
42 void CSWS_PolyLine::Draw() {
43     // code to draw a polyline
44 }
45 // test code using dynamic polymorphic behaviour
46 // -----
47 void main () {
48     // create a pointer to figure
49     CSWS_Figure *pFigure;
50
51     // create a real object
52     pFigure = (CSWS_Figure *) new CSWS_PolyLine;
53
54     // draw the polyline using the pointer to a figure
55     pFigure->Draw(); // uses CSWS_PolyLine::Draw
56 }

```

Wie dem obenstehenden Codefragment entnommen werden kann, muss die Funktion Draw der Klasse CSWS_Figure (Basisklasse) normal implementiert werden. Dies bringt den Vorteil, dass ein Objekt von der Basis-klasse direkt auch instanziiert werden könnte. Wenn eine direkte Instanzierung nicht notwendig ist, kann die Funktion als rein virtuell bezeichnet werden. In diesem Fall wird der Prototyp der Funktion erstellt, jedoch die Implementation weggelassen.

Ein rein abstrakter Begriff kann in C++ mit einem rein abstrakten Typ abgebildet werden. Dieser zeichnet sich eben dadurch aus, dass er nicht zu einem Objekt instanziiert werden kann. Wir codieren eine abstrakte Klasse, indem wir der Funktionspredeklaration eine Nullzuweisung anschließen. Eine so deklarierte Funktion nennen wir rein virtuell (engl. pure virtual).

Man beachte, dass die Existenz einer rein virtuellen Funktion innerhalb einer Klasse ausreicht um die Klasse als abstrakt zu bezeichnen. Dies ist unabhängig von allfälligen Zugriffsspezifikationen für diese Klasse (private macht keinen Sinn) und auch unabhängig von der Menge der rein virtuellen Funktionen.

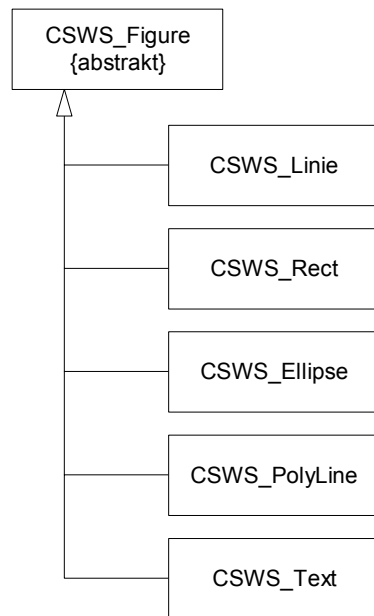


Abbildung 4-5 Beispiel einer Klassenhierarchie mit abstrakter Basisklasse.

```

1  // Definition of class CSWS_Figure
2  // -----
3  class CSWS_Figure {
4  ...
5  public:
6      ...
7      // pure virtual function for showing data
8      virtual void Draw( CSWS_GTBX *gtbxOutputDevice ) = 0;
9      ...
10 };
  
```

Mit Hilfe virtueller Funktionen kann die dynamische Polymorphie der OO-Technik implementiert werden.

Innerhalb der neu definierten "Funktion" in der abgeleiteten Klasse darf die virtuelle Funktion der Basisklasse selber aufgerufen werden. Zu diesem Zweck muss allerdings der Scope-Operator zur Auflösung der Namen verwendet werden, da sonst eine Rekursion entstehen würde.

Unterscheidet sich eine Funktion in einer abgeleiteten Klasse nur durch die Argumente von der gleichen Funktion in der Basisklasse, wird nicht der "Virtual-Mechanismus" ausgelöst, sondern nur eine "normale Funktionsüberladung" definiert.

Wird eine virtuelle Funktion mit "0" initialisiert entsteht eine "rein virtuelle Funktion" (engl. Pure Virtual Function). Eine rein virtuelle Funktion hat in der Klasse in der sie deklariert ist keine Implementation.

4.2.5 Konstruktoren und Destruktoren abgeleiteter Klassen, Initialisiererlisten

Wenn sowohl die Basisklasse als auch die abgeleitete Klasse über Konstruktoren und Destruktoren verfügt, werden diese bei der Erzeugung und Zerstörung der Objekte selbstverständlich aufgerufen. Dabei werden die Konstruktoren in der logischen Reihenfolge der Klassenhierarchie aufgerufen (Basisklasse zuerst), und die Destruktoren in umgekehrter Reihenfolge abgearbeitet (Basisklasse zuletzt).

Konstruktoren können nicht als virtuell deklariert werden. Destruktoren können als virtuell deklariert werden.

Der Konstruktor einer abgeleiteten Klasse, dessen Basiskonstruktor Argumente benötigt, sollte diese in der Definition seines Konstruktors angeben (via Initialisiererliste).

Argumentendeklaration über mehrere Konstruktoren hinweg:

```
11 // Definition der Basisklasse
12 // -----
13 class Basis {
14     int Wert1;
15
16 public:
17     Basis ( int Arg1, int Arg2 );
18     ~Basis ();
19 };
20
21 // Definition des Konstruktors der Basisklasse
22 // -----
23 Basis::Basis( int Arg1, int Arg2 )
24 {
25     ...
26 }
27
28 // Definition der abgeleiteten Klasse
29 // -----
30 class Inherit : Basis {
31     int Wert2;
32
33 public:
34     Inherit ( int Arg1, int Arg2, int Arg3 );
35     ~Inherit ();
36 };
37
38 // Definition des Konstruktors der abgeleiteten Klasse
39 // -----
40 Inherit::Inherit( int Arg1, int Arg2, int Arg3 ) :
41     Basis( Arg1, Arg2 ),      // Initialisierung Basisklasse
42     Wert2( Arg3 )            // Initialisierung Datenmember
43 {
44     ...
45 }
```

4.2.6 Abstrakte Klassen

In Klassenhierarchien ist es möglich, dass die Erzeugung von Instanzen einer Basisklasse ausgeschlossen werden soll, da mit ihr nur ein Konzept festgelegt wird, dessen verschiedene Ausprägungen in abgeleiteten Klassen spezifiziert werden müssen (wie im Figuren, Geometrieobjekt-Beispiel).

Diesen Effekt erzielt man, indem man die Basisklasse als abstrakt definiert. Eine abstrakte Klasse kann nur als Basisklasse einer anderen Klasse verwendet werden. Die abstrakte Klasse kann nicht zur Instanzierung von Objekten herangezogen werden. Eine Klasse ist in C++ als abstrakt definiert, wenn mindestens eines ihrer Funktions-Member als rein virtuell definiert ist (siehe Kapitel "Dynamische Funktionsüberladung in Vererbungen").

4.2.7 Virtuelle Basisklassen

Standardmässig sind die Daten einer Basisklasse so oft enthalten, wie Objekte von einer abgeleiteten Klasse instanziiert wurden. In speziellen Fällen kann es jedoch vorkommen, dass die Daten einer Basisklasse zwischen allen Klassen die von der Basisklasse abgeleitet wurden, geteilt werden sollen. Dies wird mit dem Spezifizierer *virtual* auf Stufe Klassenableitung erreicht.

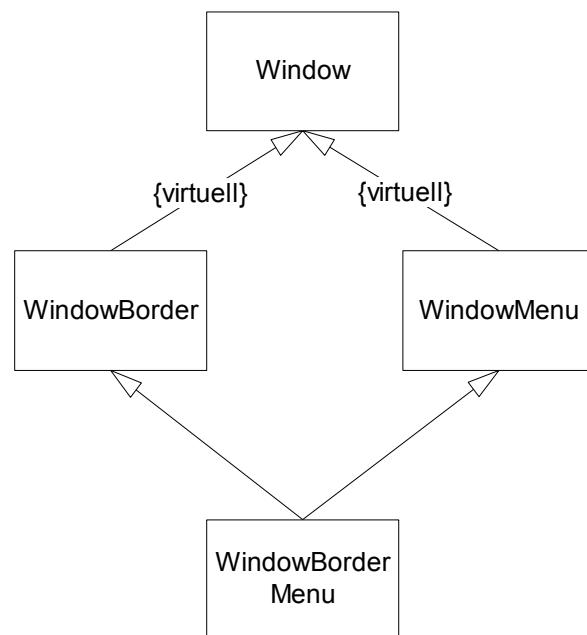


Abbildung 4-6

Beispiel der Klassenhierarchie, die zu meist zu virtuellen Basisklassen führen.

Umsetzung des obigen Diagramms in C++ Code:

```
1  // -----
2  // Declaration of class Window
3  // -----
4  class Window
5  // -----
6  {
7      ...
8      // pure virtual function for showing data
9      virtual void Draw( ) = 0;
10     ...
11 };
12
13 // -----
14 // Declaration of class WindowBorder
15 // -----
16 class WindowBorder: public virtual Window
17 // -----
18 {
19     ...
20     // function for showing data
21     void Draw( );
22     ...
23 };
24
25 // Declaration of class WindowMenu
26 // -----
27 class WindowMenu: public virtual Window
28 // -----
29 {
30     ...
31     // function for showing data
32     void Draw( );
33     ...
34 };
35
36 // Declaration of class WindowBorder
37 // -----
38 class WindowBorderMenu: public WindowBorder,
39                         public WindowMenu
40 // -----
41 {
42     ...
43     // function for showing data
44     void Draw( );
45     ...
46 };
```

Bei der Implementation von virtuellen Funktionen in virtuellen Basisklassen ist grösste Vorsicht geboten, da es leicht vorkommen kann, dass Funktionen mehrfach definiert werden und ablaufen.

```
47 // Definition of class-member WindowBorderMenu::Draw
48 // -----
49 void WindowBorderMenu::Draw( void )
50 {
51     WindowBorder::Draw();           // ok,
52     WindowMenu::Draw();             // Beachte die Veränderungen durch den
53                                     // vorangehenden call
54     // specific functionality for WindowBorderMenu
55     ...
56 };
```

4.3 Assoziationen (Beziehungsstrukturen)

Beziehungsstrukturen stammen typischerweise aus der Relationstheorie. Sie beschreiben die mengenmässigen Beziehungen zwischen Objekten einzelner Klassen. Je nach Anzahl der Objekte, die zu einem bestimmten Zeitpunkt eine Beziehung zu einander eingehen können, unterscheidet man verschiedene Kardinalitäten:

- | | |
|-------------------------------|--|
| - Einfache Beziehung: | Ein Objekt geht genau eine Beziehung ein |
| - Konditionelle Beziehung: | Ein Objekt geht keine oder eine Beziehung ein |
| - Komplexe Beziehung: | Ein Objekt geht eine oder mehrere Beziehungen ein |
| - Komplex konditionelle Bez.: | Ein Objekt geht keine, eine oder mehrere Beziehungen ein |

4.3.1 Einfache Beziehungen



Abbildung 4-7 Darstellung einer einfachen Beziehung.

Umsetzung obiger Darstellung in C++ Code mittels einem eingebundenen Objekt. Beachten Sie, dass diese Variante funktionell gleichwertig ist mit der Variante einer dynamischen Einbindung via Zeiger.

```
57 class CSWS_Text
58 {
59     CSWS_Point P1;
60     ...
61
62 public:
63     CSWS_Text( int, int, char * );
64     ...
65 };
```

4.3.2 Konditionelle Beziehungen

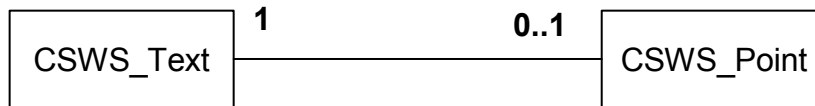


Abbildung 4-8 Darstellung einer konditionellen Beziehung.

Umsetzung obiger Darstellung in C++ Code mittels einem eingebundenen Objekt. Beachten Sie, dass diese Variante gegenüber der einfachen Beziehung dynamisch implementiert werden muss, da ansonsten immer ein eingebundenes Objekt vorhanden ist.

```
66 class CSWS_Text
67 {
68     CSWS_Point *P1;
69     ...
70
71 public:
72     CSWS_Text( int, int, char * );
73     ...
74 };
```

4.3.3 Komplex konditionelle Beziehungen

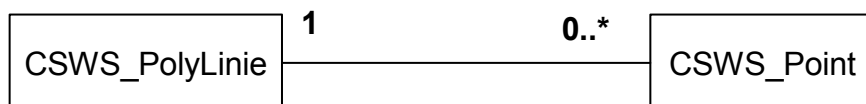


Abbildung 4-9 Darstellung einer einfach-komplexen Beziehung.

Umsetzung obiger Darstellung in C++ Code (mögliche Implementation). In der Realität wird neben der unten dargestellten "selfmade" Methode wohl eher auf ein Mechanismus unter Einbezug von Fremdmitteln wie die STL (siehe Kapitel 7) zurückgegriffen. Dazu empfehlen sich Klassen wie <list> oder <vector> besonders gut.

```
75 class CSWS_PolyLine
76 {
77     CSWS_Point **ptCenter;
78     ...
79
80 public:
81     CSWS_PolyLine( ... );
82     ...
83 };
```

Die konditionelle Beziehung bedarf einer speziellen Schreibweise der Konstruktoren. Ansonsten ändert gegenüber der komplex konditionellen Beziehung nichts grundsätzliches.

4.4 Aggregationsstrukturen (Containing Relationships)

Aggregationen bringen die Zusammensetzung eines Ganzen aus mehreren Teilen zum Ausdruck. Die Implementierung der Teilstrukturen richtet sich nach Verwendung und vorhandener Klassen respektiv Klassenbibliothek. Die Zusammensetzung von Aggregationsstrukturen kennt folgende Möglichkeiten:

- Verschachtelte Klassendefinition
- Benützung vordefinierter Klassen "by value"
- Benützung vordefinierter Klassen "by reference"



Abbildung 4-10 Darstellung einer Aggregation.

Umsetzungsmöglichkeiten obiger Darstellung in C++ Code:

```
1 // containment by nested classes
2 // -----
3 class CSWS_Text {
4 private:
5     class CSWS_Point {
6     private:
7         int iX_Coord;
8         int iY_Coord;
9     public:
10        ...
11    } P1;
12
13 public:
14     CSWS_Text( int, int, char * );
15     ...
16 };
17
18 // containment by value
19 // -----
20 class CSWS_Text {
21 private:
22     CSWS_Point P1;
23
24 public:
25     CSWS_Text( int, int, char * );
26     ...
27 };
28
29 // containment by reference
30 // -----
31 class CSWS_Text {
32 private:
33     CSWS_Point *P1;
34
35 public:
36     CSWS_Text( int, int, char * );
37     ...
38 };
```

In oben dargestellten Codefragment ist in der ersten Variante der Klassendeklaration eine eingebundene Klassen abgebildet (Zeile 1..16). Dem Scope und der Zugriffsberechtigungen sind bei dieser Art der Programmierung speziell Beachtung zu schenken. Im erwähnten Beispiel kann die Klasse CSWS_Point nur von Funktionen der Klasse CSWS_Text verwendet werden, da die gesamte Klassendeklaration mit Zugriff *private* deklariert ist. Wenn sowohl

die Klassendeklaration von CSWS_Point als auch die Daten-Member innerhalb der Klasse mit *public* deklariert würden, könnte von "ausser" mit dem Scope

CSWS_Text::CSWS_Point::int

zugegriffen werden (siehe nächstes Codefragment).

```
39 // containment by nested classes
40 // -----
41 class CSWS_Text {
42 public:
43     class CSWS_Point {
44     public:
45         int iX_Coord;
46         int iY_Coord;
47     public:
48         ...
49     } P1;
50
51 public:
52     CSWS_Text( int, int, char * );
53     ...
54 };
55
56
57 // testprogramm
58 // -----
59 void main () {
60     CSWS_Text text( 10, 10, "Testtext" );
61     ...
62     // check scope to direct access data in object of class CSWS_Point
63     text.P1.iX_Coord = 100;
64 }
```

4.5 Using (Verwendung)

Die einfache Verwendung einer Klasse ist eine lockere Bindung zwischen zwei Klassen. Sie ist gegeben, sobald die eine Klasse die andere Klasse als Signatur in Funktions-Membnern beinhaltet, oder innerhalb einer Funktionsimplementation als lokales Objekt benutzt wird.

Die Using-Beziehung ist wohl die häufigste Beziehungsart überhaupt. Sie wird insbesondere auch bei der Benützung vorgegebener Bibliotheken sehr oft verwendet.



Abbildung 4-11 Darstellung einer Verwendungsbeziehung (using).

Umsetzung obiger Darstellung in C++ Code:

```
65 // -----
66 // Class CSWS_Figure
67 // -----
68 class CSWS_Figure{
69     ...
70 public:
71     // constructor
72     CSWS_Figure();
73     // pure virtual function for showing data
74     virtual void Draw( CDC *pDC ) = 0;
75     ...
76 };
77
78 // -----
79 // Class CSWS_Rect
80 // -----
81 class CSWS_Rect{
82     ...
83 public:
84     // constructor
85     CSWS_Rect();
86
87     // virtual function for showing data
88     virtual void Draw( CDC *pDC );
89     ...
90 };
91
92 void CSWS_Rect::Draw( CDC *pDC ) {
93
94     // drawing the elements to the display
95     pDC->Rectangle( ...);
96     ...
97 };
```

5 Templates (Parametrisierte Klassen)

Bei der Repräsentation von Daten im Memory finden sich Aufgabenstellungen, die für alle Datentypen immer wieder die gleichen Herausforderungen darstellen, jedoch aufgrund verschiedener Typen, immer wieder neu implementiert werden müssen. Dazu gehören typischerweise:

- Arraybearbeitungen
- Sortierungen
- einfach und doppelt verkettete Listen
- Baumstrukturen
- Makrodefinitionen

Um die immer wiederkehrenden Funktionalitäten mit verschiedenen Typen nicht mehr implementieren zu müssen, stellt C++ die sogenannten Template-Funktionen zur Verfügung.

5.1 Template Funktionen

```
1 // Deklaration von Template-Funktion max
2 // -----
3 template <class T> // Definiert T als Typenparameter (wobei T
4                   // auch ein fundamentaler Datentyp sein kann)
5 T max( T, T );    // Verwendet T
6
7 // Implementation von Template-Funktion max
8 // -----
9 template <class T>
10 T max( T a, T b ) {
11     return ( a > b ? a : b );
12 }
```

Selbstverständlich können einfache Funktionen auch inline deklariert werden. Anwendung von max:

```
13 int a, b, c;
14 a = 25;
15 b = 30;
16 c = max( a, b );
```

5.1.1 Automatische Typenumwandlung bei Template-Funktionen

Für Templates ist keine automatische Typenumwandlung definiert.

```
17     int i, c;
18     long l;
19
20     c = max( i, l );// Fehler, keine automatische Typenumwandlung
```

Durch eine explizite Deklaration kann das Problem jedoch leicht behoben werden. Der Compiler generiert aus der untenstehenden long-Deklaration einen Funktionscode aus dem entsprechenden Template.

```
21 long max( long, long );
```

somit gilt:

```
22 int i, c;  
23 long l;  
24  
25 l= 25L;  
26 i= 13;  
27 c = (int) max( i, l ); // Zwar ok, aber gefährlich long --> int!
```

5.1.2 Überladen von Template-Funktionen

Template-Funktionen können überladen werden. Damit kann erreicht werden, dass ...

- Template-Funktionen für beliebige Datentypkombinationen gelten
- Optimierungen für spezifische Datentype implementiert werden können

Obiges Beispiel von max() funktioniert für Strings nicht, da nur die Startadressen der Strings verglichen werden (nicht sinnvoll):

```
28 char *s1, *s2;  
29 ...  
30 char *maxstr = max( s1, s2 ); // vergleicht nur Adressengrösse
```

Überladen des Templates für C-Strings

```
31 char * max( char *a, char *b ) {  
32     return ( strcmp( a, b ) > 0 ? a : b );  
33 }
```

5.1.3 Lokale Variablen

Template-Funktionen dürfen intern beliebige Variablen des Schablonentyps besitzen. Zudem können lokale Variablen statisch deklariert werden. Dabei wird für jeden Typ, für den die Funktion aufgerufen wird, eine statische Variable belegt.

5.1.4 Mehrere Typenparameter

Templates dürfen beliebig viele Typen in der Deklaration besitzen.

```
34 template <class T1, T2> // gemeint ist hier class T1 und class T2  
35  
36 void MyFunction( T1 x, T2 y )  
37 {  
38     cout << x << ' ' << y << endl;  
39 }
```

5.2 Template-Klassen

Template-Funktionen wie sie obenstehend beschrieben sind, können auch zu Klassen erweitert werden. Es entstehen somit generische Klassen (auch parametrisierbare Klassen genannt), die eine Art "vordefinierte aber typenlose Implementation" einer Problemstellung darstellen. Mit den Template-Klassen wird primär das Ziel verfolgt, den eigentlichen Datentyp erst zum Zeitpunkt des effektiven Gebrauchs der Implementation zu deklarieren.

Eine generische Klasse wird mit folgender Syntax definiert:

```
template <class T >
class Bezeichner {
    [Zugriffsspez.:]
        [Datenmember;]
    ...
    [Zugriffsspez.:]
        [Funktionsmember;]
};
```

Klassen können grundsätzlich als Template (allgemeingültig) implementiert werden.

Syntaktisch ist die Klasse mit der Logik der Templates zu ergänzen.

Innerhalb der Klasse kann der Datentyp des Templates (T) wie jeder andere Datentyp zur Deklaration von Klassenelementen (Daten und Funktionen) verwendet werden.

Inline-Implementationen sind zulässig.

Bei der Instanzierung einer Templateklasse muss der zu verwendende Datentyp deklariert werden.

Pro unterschiedlicher Nutzung eines Templates (unterschiedlicher Datentyp) wird durch den C++ Compiler dynamisch Code generiert.

In Template-Klassen ist das Überladen von Funktionen erlaubt.

Template-Klassen können geschachtelt werden (Template für Template), sowie in Vererbungen und Polymorphie genutzt werden.

5.2.1 Beispiel:

```
40 // Klassendeklaration
41 // -----
42 template <class T>
43 class Stack {
44     private:
45         T*   elems;
46         int  size;
47         int  NumElems;
48
49     public:
50         Stack(); // Konstruktor
51         Stack( const Stack<T> &); // Copy Konstruktor
52         const Stack<T>& operator= (const Stack<T>& ); // Zuweisung
53         ~Stack(); // Destruktor
54
55         void push( T& ); // element auf den Stack setzen
56         T pop(); // Element vom Stack nehmen
57 };
58
59 // Klassenimplementation
60 // -----
61
62 // Konstruktor
63 template <class T>
64 Stack<T>::Stack()
65 {
66     NumElems = 0;
67     size = 10;
68     elems = new T [ size ];
69 }
70
71 // Destruktor
72 template <class T>
73 Stack<T>::~~Stack()
74 {
75     ...
76 }
77
78 // Zuweisung
79 template <class T>
80 const Stack<T>& Stack<T>::operator= (const Stack<T>& s)
81 {
82     ...
83 }
84
85 // push
86 template <class T>
87 void Stack<T>::push( T& x)
88 {
89     ...
90 }
91
92 // pop
93 template <class T>
```

```

94  T Stack<T>::pop()
95  {
96      ...
97  }
98
99  // Anwendung des Templates für Klassen
100 // -----
101
102 // Statische Definition
103 Stack<int>      intStack;    // Stacks für int's
104 Stack<float*>   pfloatStack; // Stack für Pointer to float
105 Stack<String>   stringStack; // Stack für Strings (String muss
106                                     // als Klasse definiert sein)
107 // Dynamische Definition
108 Stack<int>      *intStack = new Stack<int>;
109
110 // via Typdef
111 typedef Stack<int> IntStack;
112 IntStack aintStack[ 10 ]; // Array von 10 int-Stacks
113
114 // Mehrfachnutzung
115 Stack< Stack<int> > intStackStack; // Leerzeichen beachten sonst
116                                     // wird >> interpretiert

```

Weitere Beispiele zu Nutzung von Templates können dem Kapitel "Standardbibliothek" (Containerklassen) entnommen werden.

6 Ausnahmebehandlung

Dieses Kapitel beschreibt den in C++ integrierten Mechanismus zur Erkennung von Ausnahmen und sowie deren Behandlung zur Laufzeit eines Programmes. Der Mechanismus beruht auf einem Ausdruck, der eine Ausnahme (exception) auswirft, die von einem Handler aufgefangen werden kann.

6.1 Fehlerbehandlung

Der Autor einer Bibliothek ist in der Lage, Laufzeitfehler zu entdecken, weiss im allgemeinen jedoch nicht, was er mit ihnen anfangen soll. Der Anwender einer Bibliothek weiss unter Umständen, wie mit solchen Fehlern umgegangen werden kann, ist jedoch nicht in der Lage sie zu entdecken - es sei denn, es handelt sich um Fehler, die dem Anwender-Code entspringen und mit der Bibliothek nichts zu tun haben (Zitat aus [3]).

Die elementare Idee der Ausnahmebehandlung besteht darin, dass eine Funktion, die einem Problem gegenübersteht das sie nicht lösen kann, eine Ausnahme (exception) auswirft (throw), und im Programm eine Funktion existiert, die sich dem Problem annimmt (catch) um eine definierte Fehlerbehandlung durchzuführen. Bei diesem Vorgang wird die Funktion, in der das Problem auftritt, kontrolliert verlassen.

Beim Exceptionhandling handelt es sich somit um ein Sprachmittel zur Beeinflussung von Fehlern und deren Auswirkungen im Programmfluss. Es ermöglicht die Trennung von Datenfluss und Fehlerbehandlung und benötigt dabei keine Parameter und Rückgabewerte (praktisch bei Konstruktoren und ähnlichem).

Beim Eintreten einer Exception wird ein Exception-Objekt erzeugt und anschliessend alle übergeordneten Blöcke geordnet verlassen, bis die Ausnahme behandelt werden kann. Dabei kann der Mechanismus hierarchisch definiert, verfeinert oder zusammengefasst werden.

Exceptions sind kein Mittel zur Kontrolle von Ausnahmesituationen die von aussen gesteuert werden (Signals, Messages, Pipes, Queues ...).

6.2 Das Konzept

Beim Eintreten von ungewöhnlichen Situationen (gezielt programmiert, aber auch bei betriebssystemnahen Funktionen wie Speicherfehler, Filefehler) werden alle aufgerufenen Blöcke bzw. Funktionen verlassen, bis der Block gefunden wird, in dem für die Ausnahme eine Behandlung existiert.

Beim Aufruf von Funktionen kann definiert werden, was passiert, wenn innerhalb der Funktion oder einer davon aufgerufenen Funktion eine Ausnahmesituation auftritt. Zu diesem Zweck wird ein eigener Block für die Behandlung von Ausnahmen in die Funktion eingesetzt. Im Gegensatz zum bekannten Fehler-Handling in C, wird in C++ die Situation also mindestens codemässig auf den Kopf gestellt, indem vor dem Ausführen der Aktion, die Fehlerbehandlung definiert wird (Antizipation). Fehler werden implementationstechnisch als Objekte angesehen. Somit erzeugt eine Ausnahme ein Fehlerobjekt.

Fehlerklassen (Klassen für Exception-Handling) beschreiben Fehlerobjekte. Für verschiedene Fehler gibt es verschiedene Fehlerklassen. Fehlerklassen können hierarchisch gegliedert werden.

6.3 Schlüsselworte

- throw "Wirft" ein entsprechendes Objekt in die Programmumgebung, wenn ein Fehler auftritt.
- catch Dient dazu ein Fehlerobjekt "einzufangen" und den Fehler zu behandeln.
- try Definiert den Gültigkeitsbereich, innerhalb dessen Fehler mit catch abgefangen und behandelt werden.

6.4 Definition von Fehlerklassen

Wenn eine Fehlerklasse nur für eine bestimmte Klasse verwendet wird, kann die Methode des "physical containment" angewendet werden.

```
1 class T {
2     // Datamember
3     private:
4         ...
5
6     // Errorclass
7     public:
8         // this class needs no member, only the existence of this class
9         //resp. a object is necessary to use the a exception-handling
10        // However, please remember that every class has a default
11        // constructor
12        class MyErrorClass {};
13
14        // other Functionmember
15        public:
16            T();
17            ...
18    };
```

Allgemeingültige Fehlerklassen werden analog "normaler" Klassen deklariert und verwendet.

```
class CError{;
```

Allgemeingültige Fehlerklassen werden oft auch Daten- und Funktions-Member beinhalten. In diesen Fällen gelten grundsätzlich die selben syntaktischen Regeln wie bei andern C++ Klassen.

```
1 class RangeError{
2     private:
3         int    iIndex;
4 }
```

```

5     public:
6         RangeError( int i) : iIndex(i) {};
7         int GetIndex( void );
8     };

```

6.5 Auslösen einer Ausnahme

Ausnahmen werden nach dem gezielten Erkennen mit den "throw" Statement ausgelöst. Man beachte, dass mit dem "throwing" der Exception effektiv der Konstruktor der Klasse aufgerufen wird. Bei komplexeren Fehlerobjekten mit entsprechendem Inhalt ist somit auch der entsprechende Konstruktor zu nutzen.

```

9 // pop-Funktion der Templateklasse Stack
10 // -----
11 template <class T>
12 T Stack<T>::pop()
13 {
14     // kein Element mehr vorhanden
15     if ( numElems == 0 ) {
16         throw CErrorStackRange();
17     }
18
19     // Element vom Stack nehmen
20     return elems[ --numElems ];
21 }

```

6.6 Auswerten einer Ausnahme

Bei der Auswertung der Ausnahme handelt es sich eigentlich eher um eine Antizipation. Das heisst, dass alle möglichen zu behandelnden Ausnahmen (fallbezogen), vor der eigentlichen Aktion definiert werden.

```

22 BOOL fLoop = TRUE;
23 while ( fLoop ) {
24     try {
25         cout << "Wert aus Stack = " << intStack.pop() << endl;
26     }
27     catch ( CErrorStackRange ) {
28         cout << "Stack ist leer" << endl;
29         fLoop = 0;
30     }
31     catch ( OtherException ) {
32         cout << "Der andere Fehler" << endl;
33     }
34     catch (...) {
35         cout << "Undefinierter Fehler im Stack" << endl;
36         fLoop = 0;
37     }
38 }

```

6.7 Weitergabe von Fehlern

Es ist möglich einen Fehler, der in einer *catch*-Anweisung behandelt wird, wieder auszuwerfen, so dass er in einer übergeordneten Behandlung bearbeitet wird.

```

1  BOOL fLoop = TRUE;
2  while ( fLoop ) {
3      try {
4          cout << "Wert aus Stack = " << intStack.pop() << endl;
5      }
6      catch ( CErrorStackRange ) {
7          cout << "Stack ist leer" << endl;
8          throw; // Fehler erneut in die Exceptionkette werfen
9      }
10 }

```

6.8 Behandlung nicht aufgefangener Exceptions

Nicht abgefangene Exceptions führen zu einem aussergewöhnlichen Programmabbruch. Zu diesem Zweck wird die Funktion *terminate()* aufgerufen, die wiederum die Funktion *abort()* aufruft. Im Gegensatz zur normalen Funktion *exit()*, verlässt die Funktion *abort()* das Programm sofort, ohne die Destruktoren der statischen Objekte aufzurufen.

Um allfällige Fehler einer Software so zu vermeiden (nicht richtig geschlossene DB's, Files ...) kann die *terminate()* Funktion mit *set_terminate()* durch eine eigene *terminate()* Funktion ersetzt werden. Die *terminate()* Funktion darf dabei weder Parameter noch Rückgabewert haben.

Die Funktion *set_terminate()* liefert die Adresse der alten *terminate()* Funktion zurück.

6.9 Extensions für Schnittstellendeklarationen

Die Menge möglicher Exceptions gehört zu einer Funktionsdeklaration wie deren Parameter. Um dies in der Sprache C++ zu unterstützen, wurde die Funktionsdeklaration um die Möglichkeit der Deklaration von Exceptions, die von der Funktion nach aussen gerichtet werden können, erweitert. Zu diesem Zweck wird die Parameterliste mit einer Liste der Exceptions ergänzt.

```

11 // Alle Exceptions aus einer Funktion möglichen:
12 void f();
13
14 // Definition bestimmter Exceptions aus einer Funktion:
15 void f() throw ( Exception-Class, Exception-Class );
16
17 // Keine Exceptions aus einer Funktion:
18 void f() throw ();

```

7 Standardbibliothek

C++ kennt eine Reihe von Standardklassen und -funktionen. Mit den Bibliotheken werden die C++ Schlüsselwörter um einen Deklarations- und Befehlssatz ergänzt, der die Sprache für einen breiten Einsatz öffnet. Viele Bibliotheken, insbesondere deren Einbindung in den Code sind analog den C Runtime-Bibliotheken anzuwenden. In diesem Kapitel werden die wichtigsten Standardfunktionen, die Standardklassen und deren Member-Funktionen sowie die Standard Template Library erklärt. Alle in diesem Kapitel vorhandenen Auszüge aus Headerfiles sind der Implementation Visual C++ von Microsoft (Implementation HP) entnommen.

Die Standardbibliothek organisiert sich in drei Hauptteilen:

- Die Standardbibliothek
- Utilities
- Die Standard Template Library

Mit der Standardbibliothek wurden ein paar weitergehende Konzepte und Grundbegriffe eingeführt, sowie die Kompatibilität zu den notwendigen Bibliotheksfunktionen wie sie von C her bekannt sind gepflegt. Da in dieser Bibliothek natürlich auch die bekannten I/O Funktionen für Files und Konsolen-Operationen vorhanden sind, ist es somit möglich diese prozedural orientierten Funktionen transparent mitzuverwenden. Später in diesem Kapitel werden wir sehen, dass diese Funktionen eigentlich in C++ obsolet sind. Demgegenüber existieren als Beispiel die mathematischen Funktionen nicht in einer speziellen C++ Library, was die eigentliche Standardbibliothek natürlich weiterhin nötig macht.

Utilities sind Hilfsmittel, die die Grundfunktionalität der Bibliothek mit kleinen aber äusserst sinnvollen Elementen ergänzt. Die Utilities sind in der Regel typenunabhängig und allgemein einsetzbar.

Die Standard Template Library (kurz STL; und nicht zu verwechseln mit Microsofts ATL = Automation Template Library) ist eine objektorientierte Ergänzung zur allgemeinen Library. In ihr sind neben einigen Basisklassen für I/O und String Behandlung insbesondere die Kontainerklassen von grosser Bedeutung. Mit diesen können Assoziationen realisiert werden.

7.1 Konzepte der C++ Standardbibliothek

7.1.1 Namespaces

Um Namenskonflikte in verschiedenen Modulen und Bibliotheken zu vermeiden, ist es möglich eigene Entwicklungen von Entwicklungen Dritter durch Namensräume zu schützen. Die Namensräume bilden logische "Gefässe" in denen die eigenen Klassen und andere Definitionen gekapselt werden.

7.1.2 Definieren eines eigenen Namenraumes

Mittels dem Schlüsselwort **namespace** wird ein neuer Namensraum definiert. Dabei kann die Definition des Namensraumes über mehrere Teile in einer oder mehreren Dateien erfolgen (zum Beispiel aufgeteilt in Interface und Implementation). Zusätzlich können Namensräume ineinander geschachtelt definiert werden.

Interface.h

```
1 namespace MyOwn {  
2     class A {  
3         ...
```

```

4     public:
5         static int getA( void );
6     ...
7     };
8 }

```

Interface.cpp

```

9 namespace MyOwn {
10     int A::getA( void )
11     {
12         return 0;
13     }
14 }

```

7.1.3 Benutzen eines Namensraumes

Ein Namensraum kann entweder implizit oder explizit verwendet werden. Die explizite Verwendung geschieht durch Nutzung des Scopeoperators, welcher den Bezeichner für den Namensraum vom Bezeichner für die Sache (Klasse oder Funktion) trennt.

```

15 int main(int argc, char* argv[])
16 {
17     MyOwn::A::getA();
18     MyOwn::A a;
19
20     a.getA();
21 }

```

Neben der expliziten Schreibweise ist die implizite Anwendung häufiger anzutreffen. Bei diesem Stil wird die Benutzung des Namensraumes mit dem Schlüsselwort **using** deklariert. Aufgrund dieser Deklaration kann anschliessend auf Definitionen im Namensraum zugegriffen werden, ohne dass dieser noch benannt wird.

```

22 using namespace MyOwn;
23
24 int main(int argc, char* argv[])
25 {
26     A::getA();
27     A a;
28
29     a.getA();
30 }

```

7.1.4 namespace std

Die Standard Library nutzt konsequent das Konzept der Namensräume. Alle Bezeichner sind im "namespace std" definiert. Bei der Verwendung der standard Streamklasse für Ein- und Ausgaben muss also statt ostream nun std::ostream verwendet werden und eine Ausgabeanweisung müsste demnach wie folgt geschrieben werden:

```

31 std::cout << std::hex << 34 << std::endl;

```

In Realität wird in der Regel die implizite Schreibweise auch hier angewendet.

```
32 using namespace std;
33 cout << hex << 34 << endl;
```

7.1.5 Header Dateien

Die Header Dateien werden in der Standard Bibliothek von C++ ohne jede Namenserverweiterung eingebunden.

```
34 #include <iostream>
35 #include <string>
```

Dies gilt für neue C++, aber auch für übernommene C Header Dateien. Zur Differenzierung der beiden Sprachen ist der Name der übernommenen C Header Dateien ein "c" vorangestellt.

```
36 #include <string>    // C++ String Klasse
37 #include <cstring>   // C String Funktionen
38 #include <cmath>     // Mathematische Funktionen
```

Dies bedeutet nicht, dass die Dateien ohne Namenserverweiterungen eingebunden werden. Der include-Name muss nicht mit dem Dateinamen übereinstimmen. So kann ein Compiler bzw. ein Präprozessor intern folgende Umwandlung definiert haben.

```
39 #include <cstdlib>
```

bedeutet intern

```
40 namespace std {
41     #include <stdlib.h>
42 }
```

Zwecks Kompatibilität werden die normalen "C-Header" auch noch zur Verfügung gestellt. Der wesentliche Unterschied ist, dass die C-Form die Identifier global und nicht im Namensraum std definiert.

7.1.6 Exceptions

Die Library unterstützt grundsätzlich das Exceptionkonzept von C++. Dabei steht eine Hierarchie von Exception-Klassen zur Verfügung. Wenn nichts anderes definiert ist, werden die Klassen mit den Headern <exception> und <stdexcept> eingebunden.

Aus der Hierarchie möglicher Fehlerobjekte kann die Bibliothek nur Fehler aus den Teilbereichen logic_error und runtime_error erzeugen. Die anderen Teilbereiche sind für Operatoren und spezielle Situationen reserviert.

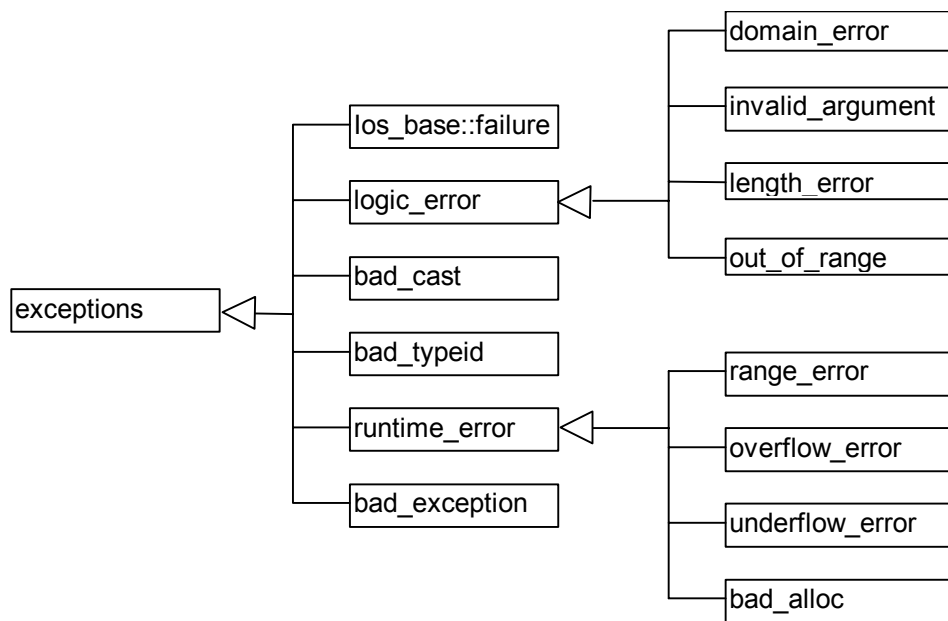


Abbildung 7-1 Hierarchie der Standard-Exceptions.

Insbesondere der Exception `bad_alloc` ist spezielle Aufmerksamkeit zu schenken. Sie kann immer dann vorkommen, wenn ein in den Bibliotheksfunktionen eingebundener `new`-Operator eine Speicherallozierung nicht erfolgreich durchführen konnte.

Um eine eigene, von einer standard Fehlerklasse abgeleitete Fehlerklasse zu konstruieren, ist es in der Regel notwendig die virtuelle Funktion `what()` zu überschreiben. Diese Funktion liefert den Grund der Fehlerauslösung.

Definition einer eigenen Fehlerklasse

```

1  class ReadEmptyQueue : public std::exception {
2
3      public:
4          virtual const char * what() const throw() {
5              return "read empty queue";
6          }
7  };
  
```

7.2 Utilities

In der Library sind diverse allgemeine Hilfsmittel verstreut. Gerade diese Hilfsmittel sind es aber, die einem Programmierer das Leben vereinfachen. Es handelt sich um folgende Dinge:

- Vergleichsoperatoren
- Hilfsfunktionen `min()`, `max()`
- Datentyp `pair`
- Datentyp `auto_ptr`
- Typenspezifische Limiten
- Portierte C-Funktionen

7.2.1 Vergleichsoperatoren

Vier Vergleichsfunktionen leiten die Vergleichsoperatoren `!=`, `>`, `<=` und `>=` aus den beiden Grundoperatoren `==` und `<` ab. Die Funktionen sind der Headerdatei `<utility>` wie folgt definiert:

```
8  template<class T>
9      bool operator!=(const T& x, const T& y);
10
11  template<class T>
12      bool operator<=(const T& x, const T& y);
13
14  template<class T>
15      bool operator>(const T& x, const T& y);
16
17  template<class T>
18      bool operator>=(const T& x, const T& y);
```

7.2.2 Hilfsfunktionen

7.2.2.1 `min()` und `max()` Funktionen

Im Gegensatz zu den Makroimplementationen `min` und `max` von C stellt die Standard Library von C++ `min` und `max` in Form von globalen Templatefunktionen zur Verfügung. Die Definitionen sind in der Datei `<algorithm>` untergebracht.

```
1  template<class T> inline
2      const T& min(const T& x, const T& y);
3
4  template<class T> inline
5      const T& max(const T& x, const T& y);
```

Zusätzlich zu den trivialen Varianten der Funktionen `min`, `max` ist noch ein Funktionspaar definiert, das zu den vergleichenden Werten auch noch einen Zeiger auf eine Vergleichsfunktion als Argument mitnehmen kann.

```
6  template<class T, class Pred> inline
7      const T& min(const T& x, const T& y, Pred pr);
8
9  template<class T, class Pred> inline
```

```
10      const T& max(const T& x, const T& y, Pred pr);
```

7.2.2.2 Vertauschen von Werten mit **swap()**

Die Funktion zum Vertauschen von zwei Werten wird ebenfalls in der Datei <algorithm> deklariert.

```
11  template<class T> inline
12      void swap(T& x, T& y);
```

Wie alle obigen Definitionen ist auch diese Funktion im "namespace std" deklariert. Das heisst, dass die allgemeingültige Syntax zum Aufruf dieser Funktion lautet:

```
13      ...
14      int x = 25;
15      int y = 17;
16
17      std::swap( x, y );
```

7.2.2.3 **cstdint** und **cstdlib** Header

Die beiden Header Dateien <cstdint> und <cstdlib> bilden die Umsetzungen der C-Header <stdint.h> und <stdlib.h>. In ihnen sind einige häufig verwendete Makros, Typen und Funktionen definiert.

Konstanten, Typen und Makros in <cstdint>

Identifizier	Bedeutung
NULL	Zeigerwert "nicht definiert" (In C++ ist dieser Wert schlicht int 0)
size_t	Datentyp für Grössenangaben
ptrdiff_t	Datentyp für Zeiger-Abstände
offsetof()	Offset einer Komponente in einer Struktur oder einem Union

Konstanten, Typen und Makros in <cstdlib>

Identifizier	Bedeutung
exit(int <i>status</i>)	Programm beenden
abort()	Programm abbrechen
atexit(void (* <i>function</i>)())	<i>function</i> bei exit() aufrufen

7.2.3 Der Datentyp pair

In der Bibliothek werden an einigen Stellen Wertepaare verwendet. In der Header-Datei <utility> ist dazu eine entsprechende Klasse deklariert.

```
1  template<class T, class U>
2      struct pair {
3
4          // Datenmember
5          T first;
6          U second;
7
8          // Default Konstruktor
9          pair();
10
11         // Konstruktor mit Initialisierung von first und second
12         pair(const T& x, const U& y);
13     };
```

Um die Benutzung des Datentyps pair abzurunden sind die beiden Vergleichsoperatoren == und < definiert.

```
14  template<class T, class U>
15      bool operator==(const pair<T, U>& x, const pair<T, U>& y);
16
17  template<class T, class U>
18      bool operator<(const pair<T, U>& x, const pair<T, U>& y);
```

Eine weitere Funktion make_pair() , dient dazu ein Wertepaar anzulegen, ohne dass die Typen explizit angegeben werden müssen.

```
1  template<class T, class U>
2      pair<T, U> make_pair(const T& x, const U& y);
```

Beispiel:

```
3      pair< int, char > objPair1( 65, 'A' );
4      pair< int, char > objPair2;
5      objPair2 = make_pair( 66, 'B' );
```

Bei dieser Anwendung unbedingt die Regeln der Standardtypenumwandlung beachten. Der Ausdruck pair<int, float>(46, 87.89) ist nicht identisch mit dem Ausdruck make_pair(46, 87.89), da die Standardtypenumwandlung beim zweiten Ausdruck einen Datentyp double (und nicht float) erzeugt.

7.2.4 Auto-Pointer

Wie der Name bereits vermuten lässt, bietet die Klasse auto_ptr einen weiteren Datentyp für Zeiger an. Jedoch im Gegensatz zu normalen Zeigern, zerstört ein Zeiger der Klasse auto_ptr bei Aufruf seines Destruktors auch das Objekt auf das er zeigt. Diese Technik wird auch Smart-Pointer genannt.

Die Deklaration der Klasse `auto_ptr` in der Headerdatei `<memory>` lautet wie folgt:

```
6  template<class T>
7      class auto_ptr {
8      public:
9          explicit auto_ptr(T *p = 0);
10         auto_ptr(auto_ptr& rhs);
11         auto_ptr<T>& operator=(auto_ptr& rhs);
12         ~auto_ptr();
13         T& operator*() const;
14         T *operator->() const;
15         T *get() const;
16         T *release();
17         T *reset(T *p = 0);
18     };
```

Mit Hilfe der Klasse `auto_ptr` kann somit Code nach folgender Form geschrieben werden:

```
1  void f()
2  {
3      auto_ptr< MYCLASS > ptr;
4
5      ptr = new MYCLASS; // Objekt der Klasse MYCLASS instanzieren
6      ...                // beliebige Aktionen durchführen
7  }                      // delete ptr ist nicht nötig, da auto_ptr
8                          // verwendet wurde
```

Da das Objekt auf das ein Zeiger vom Typ `auto_ptr` zeigt, bei dessen Zerstörung ebenfalls zerstört wird, hat das zur Konsequenz, dass nicht zwei Zeiger auf das gleiche Objekt zeigen dürfen. Diesem Umstand ist in der Standardbibliothek durch die Weitergabe des Besitzrechtes auf das Objekt das mit einem `auto_ptr`-Zeiger referenziert wird, Rechnung getragen.

Beim Kopieren eines Auto-Pointers geht das verwaltete Objekt in den Besitz der Kopie über.

Bei Zuweisung wechselt der Besitz des Objekts vom rechts des Zuweisungszeichens stehenden Auto-Pointers, zum zugewiesenen Auto-Pointer.

Auto-Pointer Parameter in Funktionsaufrufen sind in der Regel als Referenz auszulegen (siehe obige Kopierregel). Der Besitz des referenzierten Objektes würde ansonsten an die aufgerufene Funktion übergehen, welche am Funktionsende den eigentlichen besitzenden Auto-Pointer (auf dem Funktionsstack) zerstört, was wiederum ein Zerstören des referenzierten Objektes nach sich ziehen würde.

7.2.5 Typ spezifische Definitionen (limits)

Die Headerdatei `<limits>` stellt die ebenfalls von C her bekannten Daten betreffend plattformabhängigen Limiten zur Verfügung. Innerhalb der Datei werden in der Template-Klasse `numeric_limits<>` die Werte definiert.

```
9  template<class T>
10     class numeric_limits {
11     public:
12         static const bool has_denorm;
13         static const bool has_denorm_loss;
14         static const bool has_infinity;
15         static const bool has_quiet_NaN;
```

```

16         static const bool has_signaling_NaN;
17         static const bool is_bounded;
18         static const bool is_exact;
19         static const bool is_iec559;
20         static const bool is_integer;
21         static const bool is_modulo;
22         static const bool is_signed;
23         static const bool is_specialized;
24         static const bool tinyness_before;
25         static const bool traps;
26         static const float_round_style round_style;
27         static const int digits;
28         static const int digits10;
29         static const int max_exponent;
30         static const int max_exponent10;
31         static const int min_exponent;
32         static const int min_exponent10;
33         static const int radix;
34         static T denorm_min() throw();
35         static T epsilon() throw();
36         static T infinity() throw();
37         static T max() throw();
38         static T min() throw();
39         static T quiet_NaN() throw();
40         static T round_error() throw();
41         static T signaling_NaN() throw();
42     };

```

Diverse Grössenausgaben (Limits) mit der STL.

```

43     #include <limits>
44     ...
45
46     // Grössenausgaben
47     cout << "max(short):          "
48           << numeric_limits<short>::max()
49           << endl;
50     cout << "max(int):            "
51           << numeric_limits<int>::max()
52           << endl;
53     cout << "max(long):           "
54           << numeric_limits<long>::max()
55           << endl;
56     cout << "max(float):          "
57           << numeric_limits<float>::max()
58           << endl;
59     cout << "max(double):         "
60           << numeric_limits<double>::max()
61           << endl;
62     cout << "is char signed:      "
63           << (numeric_limits<char>::is_signed ? "true" : "false")
64           << endl;

```

erzeugt die Ausgabe:

```

65     max(short):          32767
66     max(int):            2147483647
67     max(long):           2147483647

```

```

68 max(float) :      3.40282e+038
69 max(double) :     1.79769e+308
70 is char signed:    true

```

7.3 Die Standard Template Library

7.3.1 Allgemeines Konzept

Die Standard Template Library entspricht in Ihrem Wesen nicht dem klassischen objektorientierten Lösungsansatz. Vielmehr sind die Daten und die Funktionen bewusst voneinander getrennt worden, damit ein allgemeingültiges und vor allem kräftiges Programmiermodell entsteht. Die Library unterscheidet :

- Container dienen dazu, Mengen von Objekten eines bestimmten Typs zu verwalten.
- Iteratoren dienen dazu, über eine Menge von Objekten zu wandern (iterieren).
- Algorithmen dienen dazu, Mengen als Ganzes und bestimmte Elemente in Mengen zu bearbeiten.
- Allocators dienen dazu, den Speicher bei der Verwendung von Containern zu verwalten.
- Funktionelle Objekte definieren Hilfsfunktionen wie less(), greater(), plus() usw. Die funktionellen Objekte werden anstelle von Funktionspointer verwendet.

Das Highlight der STL ist aber eindeutig ihre Typenfreiheit. Das heisst, dass die Gesamtheit von Klassen und Funktionen typenneutral in Form von Schablonen (Templates) realisiert ist. Man spricht in diesem Zusammenhang auch von der "generischen Programmierung".

7.3.2 STL-Container

Die Container-Klassen dienen dazu, eine Menge von Elementen in einer bestimmten Art und Weise zu verwalten. Um den verschiedenen Ansprüchen für die Verwaltung von Daten in Bezug auf Verhalten wie Sortierung, Zugriffsart oder Zugriffsgeschwindigkeit gerecht zu werden, stellt die STL mehrere Grundtypen von Container-Klassen zur Verfügung:

- Vektoren
- Deques
- Listen
- Maps und Multimaps
- Sets und Multisets

Allen Klassen sind bestimmte Eigenschaften gemeinsam. Dieser Umstand ist natürlich im Umgang mit den Container-Klassen sehr praktisch, kann doch damit sogar die spätere Änderung von der einen Klasse in eine andere erreicht werden.

Von den Elementen werden intern Kopien angelegt. Dies hat zur Folge, dass alle Elemente entsprechende Copy-Konstruktoren zur Verfügung stellen.

Die Elemente besitzen eine Reihenfolge, welche sequentiell abgearbeitet werden kann.

Die Operatoren sind nicht sicher, das heisst konkret, dass praktisch keine Fehlerprüfungen gemacht werden (keine Exceptions ausgelöst werden).

Verschiedenste Operatoren sind vordefiniert (siehe untenstehenden tabellarischen Auszug)

Ausdruck	Bedeutung
<code>ContTyp m</code>	Erzeugt einen leeren Container ohne Elemente

<code>ContTyp m1(m2)</code>	Erzeugt einen Container m1 als Kopie des Containers m2
<code>ContTyp m1(m2.anf, m2.end)</code>	Erzeugt einen Container m1 mit Kopien der Elemente des Containers m2 aus einem bestimmten Bereich
<code>m.~ContTyp()</code>	löscht alle Elemente und gibt den Speicher frei
<code>m.size()</code>	liefert die aktuelle Anzahl Elemente im Container
<code>m.empty()</code>	sagt aus ob der Container leer ist (entspricht dem Ausdruck <code>m.size()==0</code>)
<code>m.max_size()</code>	liefert die maximal mögliche Anzahl Elemente
<code>m1 == m2</code>	sagt aus ob m1 gleich m2 ist
<code>m1 != m2</code>	sagt aus ob m1 ungleich m2 ist
<code>m1 < m2</code>	sagt aus ob m1 kleiner m2 ist
<code>m1 > m2</code>	sagt aus ob m1 grösser m2 ist
<code>m1 <= m2</code>	sagt aus ob m1 kleiner oder gleich m2 ist
<code>m1 >= m2</code>	sagt aus ob m1 grösser oder gleich m2 ist
<code>m1 = m2</code>	weist m1 alle Elemente von m2 zu
<code>m1.swap(m2)</code>	vertauscht die Elemente von m1 mit denen von m2
<code>m.begin()</code>	liefert einen Iterator für das erste Element ¹
<code>m.end()</code>	liefert einen Iterator für die Position hinter dem letzten Element
<code>m.insert(pos, element)</code>	fügt an pos das element in m ein
<code>m.erase(pos)</code>	löscht das Element an pos
<code>m.erase(anfang, ende)</code>	löscht alle Elemente im Bereich von anfang bis ende
<code>m.clear()</code>	löscht alle Elemente (leert den Container)

7.3.2.1 Sequentielle Container

Sequentielle Container sind geordnete Mengen, bei denen jedes Element eine bestimmte Position besitzt, die durch den Zeitpunkt und den Ort des Einfügens definiert ist.

Vektor

Der Vektor-Container (kurz: der Vektor) verwaltet die Elemente in einem dynamischen Array. Auf die einzelnen Elemente kann mit Hilfe des Index direkt auf die Elemente zugegriffen werden (random access). Das Zeitverhalten des Vektors ist für die Mutation von Daten am Ende optimal, jedoch für Änderungen im Bereich der Mitte (Löschen neu einfügen) sehr langsam.

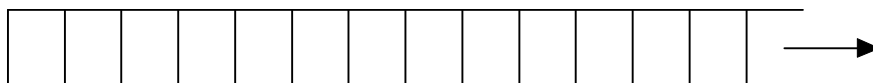


Abbildung 7- 2 Der Vektor ist ein "dynamisches Array"

Die Funktionalität des Vektors ist recht gross. Für eine genaue Beschreibung sollte unbedingt die Standardliteratur, resp. Die Beschreibung der aktuell Verwendeten Implementation konsultiert werden. Untenstehend sind lediglich die wichtigsten Funktionen wiedergegeben:

Integer Werte 1 bis 10 mittels eines Vektors.

¹ Siehe Kapitel Iteratoren

```

1  #include "stdafx.h"
2  #include <iostream>
3  #include <vector>
4  using namespace std;
5
6  int main(int argc, char* argv[]) {
7      // vector definieren
8      vector<int> iMenge;
9
10     // vector füllen
11     for ( int iCounter = 1; iCounter <=10; iCounter++ ) {
12         iMenge.push_back( iCounter );
13     }
14
15     // vector auslesen
16     for ( iCounter = 0; iCounter < iMenge.size(); iCounter++ ) {
17         cout << iCounter << ". Element: "
18             << iMenge[ iCounter ] << endl;
19     }
20     return 0;
21 }

```

Erzeugt die Ausgabe:

```

22     0. Element: 1
23     1. Element: 2
24     2. Element: 3
25     3. Element: 4
26     4. Element: 5
27     5. Element: 6
28     6. Element: 7
29     7. Element: 8
30     8. Element: 9
31     9. Element: 10

```

Deque

Die Deque ist ein dynamisches Array, das so implementiert ist, dass es in beide Richtungen wachsen kann (Deque steht für double ended queue). Das Zeitverhalten ist hier für Änderungen am Anfang und am Ende optimal. Hingegen ist die Deque für Einschieben und Löschen von Elementen im Bereich der Mitte nicht ideal.

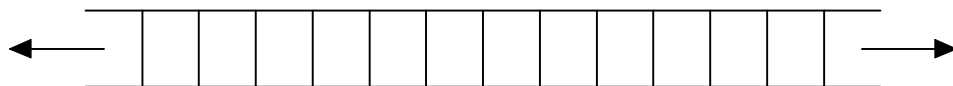


Abbildung 7- 3 Die Deque ist ein "dynamisches Array"

Integer Werte 1 bis 10 mittels einer Deque (2 mal).

```

1  #include "stdafx.h"
2  #include <deque>
3  #include <iostream>
4  using namespace std;
5
6  int main(int argc, char* argv[]) {
7
8      // deque

```

<code>ContTyp m1(m2)</code>	Erzeugt einen Container m1 als Kopie des Containers m2
<code>ContTyp m1(m2.anf, m2.end)</code>	Erzeugt einen Container m1 mit Kopien der Elemente des Containers m2 aus einem bestimmten Bereich
<code>m.~ContTyp()</code>	löscht alle Elemente und gibt den Speicher frei
<code>m.size()</code>	liefert die aktuelle Anzahl Elemente im Container
<code>m.empty()</code>	sagt aus ob der Container leer ist (entspricht dem Ausdruck <code>m.size()==0</code>)
<code>m.max_size()</code>	liefert die maximal mögliche Anzahl Elemente
<code>m1 == m2</code>	sagt aus ob m1 gleich m2 ist
<code>m1 != m2</code>	sagt aus ob m1 ungleich m2 ist
<code>m1 < m2</code>	sagt aus ob m1 kleiner m2 ist
<code>m1 > m2</code>	sagt aus ob m1 grösser m2 ist
<code>m1 <= m2</code>	sagt aus ob m1 kleiner oder gleich m2 ist
<code>m1 >= m2</code>	sagt aus ob m1 grösser oder gleich m2 ist
<code>m1 = m2</code>	weist m1 alle Elemente von m2 zu
<code>m1.swap(m2)</code>	vertauscht die Elemente von m1 mit denen von m2
<code>m.begin()</code>	liefert einen Iterator für das erste Element ¹
<code>m.end()</code>	liefert einen Iterator für die Position hinter dem letzten Element
<code>m.insert(pos, element)</code>	fügt an pos das element in m ein
<code>m.erase(pos)</code>	löscht das Element an pos
<code>m.erase(anfang, ende)</code>	löscht alle Elemente im Bereich von anfang bis ende
<code>m.clear()</code>	löscht alle Elemente (leert den Container)

7.3.2.1 Sequentielle Container

Sequentielle Container sind geordnete Mengen, bei denen jedes Element eine bestimmte Position besitzt, die durch den Zeitpunkt und den Ort des Einfügens definiert ist.

Vektor

Der Vektor-Container (kurz: der Vektor) verwaltet die Elemente in einem dynamischen Array. Auf die einzelnen Elemente kann mit Hilfe des Index direkt auf die Elemente zugegriffen werden (random access). Das Zeitverhalten des Vektors ist für die Mutation von Daten am Ende optimal, jedoch für Änderungen im Bereich der Mitte (Löschen neu einfügen) sehr langsam.

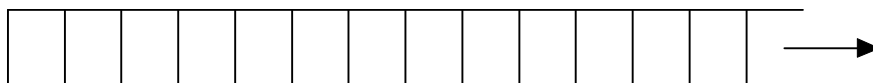


Abbildung 7- 2 Der Vektor ist ein "dynamisches Array"

Die Funktionalität des Vektors ist recht gross. Für eine genaue Beschreibung sollte unbedingt die Standardliteratur, resp. Die Beschreibung der aktuell Verwendeten Implementation konsultiert werden. Untenstehend sind lediglich die wichtigsten Funktionen wiedergegeben:

Integer Werte 1 bis 10 mittels eines Vektors.

¹ Siehe Kapitel Iteratoren

```

19         iListe.pop_front();           // erstes Element löschen
20     }
21     cout << endl;
22
23     return 0;
24 }

```

Erzeugt die Ausgabe:

```

25     1 2 3 4 5 6 7 8 9 10

```

7.3.2.2 Assoziative Container

Assoziative Container sortieren ihren Inhalt automatisch anhand einer Vergleichsfunktion, die von aussen definiert werden kann. Das Zeitverhalten der meist als Binärbäume implementierten Container ist gut, insbesondere die Suche von Elementen ist sehr schnell.

Set / Multiset

Ein Set-Container (kurz: ein Set) ist eine Menge, in der die Elemente jeweils nur einmal vorkommen dürfen und automatisch nach ihrem Wert sortiert werden. Im Gegensatz dazu können in einem Multiset die Elemente mehrfach vorkommen, sind jedoch auch sortiert.

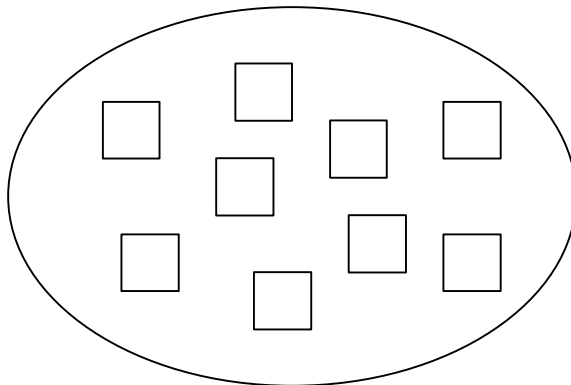


Abbildung 7- 5 Das Set als Menge.

Das Set definiert jeden Wert als Unikat in der Menge.

```

1  #include "stdafx.h"
2  #include <set>
3  #include <iostream>
4  using namespace std;
5
6  int main(int argc, char* argv[])
7  {
8
9      // set als Typ deklarieren, beachte die Benützung des
10     // funktionellen Objektes less<>
11     typedef set<int, less<int> > IntSet;
12     IntSet iSet;
13
14     // set 1. mal füllen
15     srand( (unsigned)0 );

```

```

16     for ( int iCounter = 1; iCounter <=10; iCounter++ ) {
17         iSet.insert( rand());
18     }
19
20     // set 2. mal füllen - mit den selben Werten
21     srand( (unsigned)0 );
22     for ( iCounter = 1; iCounter <=10; iCounter++ ) {
23         iSet.insert( rand());
24     }
25
26     // set auslesen
27     IntSet::iterator pos;
28     for ( pos = iSet.begin(); pos != iSet.end(); ++pos ) {
29         cout << *pos << ' ';
30     }
31     cout << endl;
32     return 0;
33 }

```

Erzeugt die Ausgabe:

```

34 38 2437 7719 8365 8855 10450 11797 21238 30612 32285

```

Das MultiSet lässt mehrfach Elemente zu.

```

35 #include "stdafx.h"
36 #include <set>
37 #include <iostream>
38 using namespace std;
39
40 int main(int argc, char* argv[])
41 {
42     // multiset
43     typedef multiset<int, less<int> > IntMultiSet;
44     IntMultiSet iMultiSet;
45
46     // set 1. mal füllen
47     srand( (unsigned)0 );
48     for ( iCounter = 1; iCounter <=10; iCounter++ ) {
49         iMultiSet.insert( rand() );
50     }
51
52     // set 2. mal füllen - mit den selben Werten
53     srand( (unsigned)0 );
54     for ( iCounter = 1; iCounter <=10; iCounter++ ) {
55         iMultiSet.insert( rand() );
56     }
57
58     // set auslesen
59     IntMultiSet::iterator mpos;
60     for ( mpos=iMultiSet.begin(); mpos!=iMultiSet.end(); ++mpos){
61         cout << *mpos << ' ';
62     }
63     cout << endl;
64
65     return 0;

```

```
66 }
```

Erzeugt die Ausgabe:

```
67 38 38 2437 2437 7719 7719 8365 8365 8855 8855 10450 10450
68 11797 11797 21238 21238 30612 30612 32285 32285
```

Map / Multimap

Ein Map-Container (kurz: Map) verwaltet Schlüssel-Wert-Paare. Das heisst, dass jedes Element ein Schlüssel und ein dazugehöriger Wert aufweist. Die Elemente werden nach Schlüssel sortiert, wobei in einem Map jeder Schlüssel nur einmal vorkommen darf. Im Gegensatz dazu, darf bei einem Multimap der Schlüssel mehrmals vorkommen.

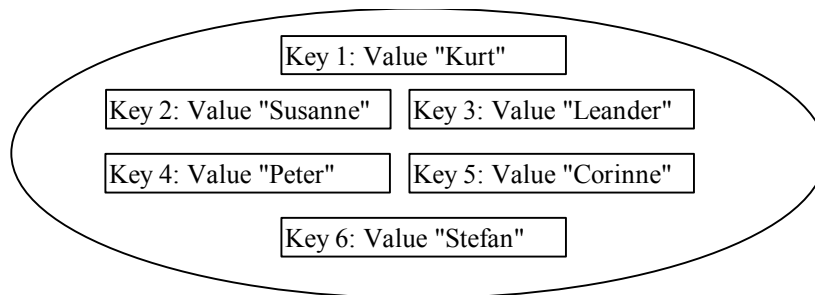


Abbildung 7-6 Der Map-Container als Menge.

```
69 #include "stdafx.h"
70 #include <map>
71 #include <string>
72 #include <iostream>
73 using namespace std;
74
75 typedef map<int, string > MYMAP;
76 typedef map<int, string >::iterator ITRMYMAP;
77
78 int main(int argc, char* argv[])
79 {
80     // definitionen für map
81     MYMAP map;
82
83     // map 1.mal füllen
84     map.insert( pair<int, string>( 1, "Kurt" ));
85     map.insert( pair<int, string>( 4, "Peter" ));
86     map.insert( pair<int, string>( 6, "Stefen" ));
87     map.insert( pair<int, string>( 2, "Susanne" ));
88     map.insert( pair<int, string>( 5, "Corinne" ));
89     map.insert( pair<int, string>( 3, "Leander" ));
90
91     // map 2.mal füllen
92     map.insert( pair<int, string>( 1, "A" ));
93     map.insert( pair<int, string>( 4, "B" ));
94     map.insert( pair<int, string>( 6, "C" ));
95     map.insert( pair<int, string>( 2, "D" ));
96     map.insert( pair<int, string>( 5, "E" ));
97     map.insert( pair<int, string>( 3, "F" ));
```

```

98
99      // map auslesen mit einem iterator
100      ITRMYMAP itrMap;
101      itrMap = map.begin();
102      while ( itrMap != map.end()) {
103
104          // Ausgabe der Werte
105          cout << "Element " << itrMap->first
106              << " "          << itrMap->second << endl;
107
108          // iterator weiterschalten
109          itrMap++;
110      }
111      cout << endl;
112
113      return 0;
114  }

```

Erzeugt die Ausgabe:

```
115 Element 1 Kurt
116 Element 2 Susanne
117 Element 3 Leander
118 Element 4 Peter
119 Element 5 Corinne
120 Element 6 Stefen
```

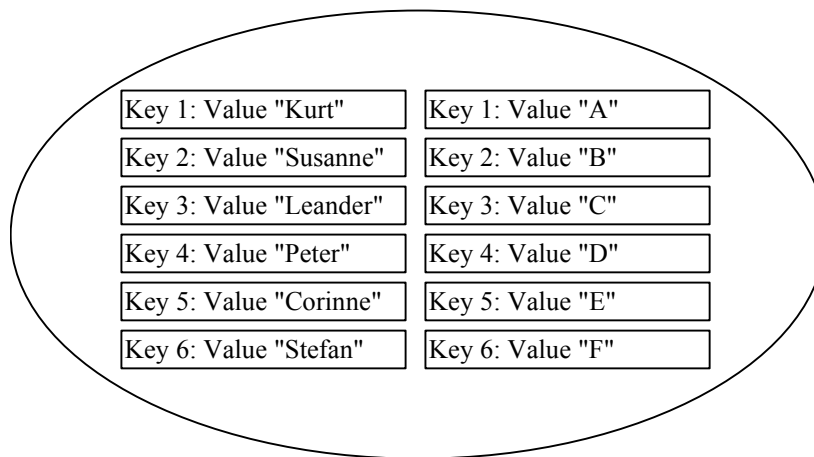


Abbildung 7-7 Der Multimap-Container als Menge.

```
121 #include "stdafx.h"
122 #include <map>
123 #include <string>
124 #include <iostream>
125 using namespace std;
126
127 typedef multimap<int, string > MYMMAP;
128 typedef multimap<int, string >::iterator ITRMMYMAP;
129
130 int main(int argc, char* argv[])
131 {
132     // definitionen für map
133     MYMMAP mmap;
134
135     // map 1.mal füllen
136     mmap.insert( pair<int, string>( 1, "Kurt" ));
137     mmap.insert( pair<int, string>( 4, "Peter" ));
138     mmap.insert( pair<int, string>( 6, "Stefen" ));
139     mmap.insert( pair<int, string>( 2, "Susanne" ));
140     mmap.insert( pair<int, string>( 5, "Corinne" ));
141     mmap.insert( pair<int, string>( 3, "Leander" ));
142
143     // map 2.mal füllen
144     mmap.insert( pair<int, string>( 1, "A" ));
145     mmap.insert( pair<int, string>( 4, "B" ));
146     mmap.insert( pair<int, string>( 6, "C" ));
147     mmap.insert( pair<int, string>( 2, "D" ));
148     mmap.insert( pair<int, string>( 5, "E" ));
```

```

149     mmap.insert( pair<int, string>( 3, "F" ) );
150
151     // map auslesen mit einem iterator
152     ITRMMYMAP itrMMap;
153     itrMMap = mmap.begin();
154     while ( itrMMap != mmap.end() ) {
155
156         // Ausgabe der Werte
157         cout << "Element " << itrMMap->first
158             << " " << itrMMap->second << endl;
159
160         // iterator weiterschalten
161         itrMMap++;
162     }
163     cout << endl;
164
165     return 0;
166 }

```

Erzeugt die Ausgabe:

```

167 Element 1 Kurt
168 Element 1 A
169 Element 2 Susanne
170 Element 2 D
171 Element 3 Leander
172 Element 3 F
173 Element 4 Peter
174 Element 4 B
175 Element 5 Corinne
176 Element 5 E
177 Element 6 Stefen
178 Element 6 C

```

7.3.3 STL-Iteratoren

Die Iteratoren sind Objekte, die es ermöglichen eine allgemeingültige Verbindung zwischen Algorithmen (siehe nächstes Kapitel) und den Containerklassen herzustellen. Die Iteratoren sind somit in allen Containerklassen als gemeinsame Elemente definiert, und werden entsprechend einheitlich verwendet.

Für die programmatische Bearbeitung von Containern werden oft Schleifensteuerungen eingesetzt. Dabei wird ein Iteratorobjekt als Pointer auf das gerade aktuelle Containerelement für die Verarbeitung benutzt. Dieses Objekt wird generell nach folgender Konstruktionsmethode erzeugt:

Containerklasse::iterator IteratorBezeichner;

Definition eines Iteratorobjektes.

```

1 // ohne typedef
2 set<int, less<int> >::iterator pos;
3
4 // mit typedef
5 typedef set<int, less<int> > IntSet;
6 ...
7 IntSet::iterator pos;

```

Um mittels des Iteratorobjekts die Schleife korrekt zu kontrollieren, müssen noch entsprechende Vergleichsobjekte erzeugt werden. Zu diesem Zweck sind in den Containerklassen Funktionsmember definiert, die temporäre Iteratorobjekte liefern. Diese können auch für implizite Schleifensteuerungen verwendet werden.

Ausdruck	Bedeutung
begin()	liefert einen Iterator für das erste Element
end()	liefert einen Iterator für die Position hinter dem letzten Element
rbegin()	liefert einen Iterator für das letzte Element
rend()	liefert einen Iterator für die Position hinter dem ersten Element

Das Iteratorobjekt für Ende zeigt jeweils hinter das letzte Element im Container in Lese-richtung.

Für die Iteratoren müssen keine Includedefinitionen gemacht werden.

Eine Codebeispiel für Iterator kann den Erklärungen zu Map und Multimap entnommen werden.

7.3.4 STL-Algorithmen

Die STL-Algorithmen bieten Unterstützung beim Finden, Vertauschen, Sortieren, Kopieren, Aufaddieren und Modifizieren von Containern. Die Algorithmen sind keine Funktionsmember der Container sondern globale Funktionen die mit Hilfe von Iteratoren gesteuert werden.

7.3.4.1 Nicht modifizierende Algorithmen

Funktion	Bedeutung
for_each()	ruft für jedes Element eine readonly Operation auf()
find(), find_if()	sucht bestimmtes Element
search()	sucht erste Teilfolge von bestimmten Werten
find_end()	sucht letzte Teilfolge von bestimmten Werten
find_first_of()	sucht eines von mehreren möglichen Elementen
adjacent_find()	sucht zwei benachbarte Elemente mit bestimmten Eigenschaften
min_element()	liefert das kleinste Element
max_element()	liefert das grösste Element
count(), count_if()	zählt Elemente
equal()	testet zwei Bereiche auf Gleichheit
lexicographical_compare()	testet zwei Bereiche lexikalisch auf <
mismatch()	liefert die ersten beiden unterschiedlichen Elemente zweier Bereiche

7.3.4.2 Modifizierende Algorithmen

Funktion	Bedeutung
copy()	kopiert einen Bereich
copy_backward()	kopiert einen Bereich, angefangen von hinten
swap_ranges()	vertauscht die Elemente zweier Bereiche
transform()	modifiziert die Elemente eines oder zweier Bereiche
fill(), fill_n()	weist mehreren Bereichen einen festen Wert zu
generate(), generate_n()	weist mehreren Elementen einen jeweils generierten Wert zu
replace(), replace_if()	ersetzt bestimmte Werte durch einen neuen festen Wert
replace_copy(), replace_copy_if()	kopiert einen Bereich und ersetzt dabei Elemente

7.3.4.3 Löschende Algorithmen

Funktion	Bedeutung
remove(), remove_if()	löscht bestimmte Elemente
remove_copy(), remove_copy_if()	kopiert einen Bereich und löscht dabei Werte
unique()	löscht aufeinanderfolgende Duplikate
unique_copy()	kopiert und löscht dabei aufeinanderfolgende Duplikate

7.3.4.4 Mutierende Algorithmen

Funktion	Bedeutung
reverse()	kehrt die Reihenfolge von Elementen um
reverse_copy()	kopiert und kehrt die Reihenfolge von Elementen um
rotate()	rotiert die Elemente
rotate_copy()	kopiert und rotiert dabei die Elemente
next_permutation()	permutiert die Reihenfolge in eine Richtung
prev_permutation()	permutiert die Reihenfolge in die andere Richtung
random_shuffle()	bringt die Reihenfolge der Elemente durcheinander
partition()	verschiebt bestimmte Elemente
stable_partition()	verschiebt bestimmte Elemente und behält dabei relative Reihenfolge

7.3.4.5 Sortierende Algorithmen

Funktion	Bedeutung
sort()	sortiert Elemente
stable_sort()	sortiert unter Beibehaltung der Reihenfolge von Elementen mit gleichen Werten
partial_sort()	sortiert die ersten n Elemente
partial_sort_copy()	kopiert die ersten n sortierten Elemente
nth_element()	sortiert in Hinsicht auf ein bestimmtes Element
make_heap()	macht einen Bereich zu einem Heap
push_heap()	integriert ein Element in einen Heap
pop_heap()	löst ein Element aus einem Heap
sort_heap()	sortiert einen Heap komplett

7.3.4.6 Algorithmen für sortierte Bereiche

Funktion	Bedeutung
lower_bound()	liefert das erste Element $\geq$ einem Wert
upper_bound()	liefert das erste Element $>$ einem Wert
equal_range()	liefert einen Bereich von Elementen mit einem bestimmten Wert
binary_search()	liefert, ob ein Element enthalten ist
includes()	liefert, ob alle Elemente einer Teilmenge enthalten sind
merge()	fasst die Elemente zweier Bereiche zusammen
set_union()	bildet die Vereinigungsmenge zweier Bereiche
set_intersection	bildet die Schnittmenge zweier Bereiche
set_difference()	bildet die Differenzmenge zweier Bereiche
set_symetric_difference()	bildet die Komplementärmenge zweier Bereiche
inplace_merge()	verschmilzt zwei direkt ineinanderliegende Teilbereiche

7.3.4.7 Numerische Algorithmen

Funktion	Bedeutung
accumulate()	verknüpft Elemente
partial_sum()	verknüpft ein Element jeweils mit allen Vorgängern
adjacent_difference()	verknüpft jeweils ein Element mit seinem Vorgänger
inner_product()	verknüpft alle Verknüpfungen von jeweils zwei Elementen zweier Bereiche

7.4 Strings

In der Headerdatei <string> werden mehrere Stringklassen definiert:

- `basic_string` ist die Basis-String-Templateklasse. Diese legt noch keinen Typ der zu verwenden-
den Zeichen fest, sondern definiert die grundlegenden Mechanismen.
- `string` ist die eigentliche Stringklasse mit den Zeichen vom Typ `char`.
- `wstring` ist eine weitere Stringklasse für Strings die aus Zeichen vom Typ `wchar_t` sind. Diese
Strings dienen der Internationalisierung.

7.4.1 Konstruktion, Kopieren und Destruktion von Strings

Vielfältig überladene Konstruktoren sowie ein Destruktor ermöglichen einfachste Erzeugung und korrektes Memory-
management bei der Zerstörung.

Funktion	Bedeutung
<code>string s</code>	erzeugt einen Leerstring
<code>string s1(s2)</code>	erzeugt eine String als Kopie eines andern Strings
<code>string s1(s2,3)</code>	erzeugt eine String als Kopie der Zeichen eines andern Strings, beginnend mit Index 3
<code>string s1(s2,3,2)</code>	erzeugt einen String als Kopie von max 2 Zeichen eines andern Strings, beginnend mit Index 3
<code>string s("hallo")</code>	erzeugt einen String als Kopie eines C-Strings
<code>string s "hallo", 4)</code>	erzeugt einen String als Kopie der ersten 4 Zeichen eines C-Strings
<code>string s(10, ' ')</code>	erzeugt eine String mit 10 Leerzeichen
<code>s.~string()</code>	Zerstört einen String

7.4.2 Nicht modifizierende Zugriffe

Funktion	Bedeutung
<code>size(), length()</code>	liefert die Anzahl Zeichen im String
<code>empty()</code>	liefert, ob String leer ist
<code>max_size()</code>	liefert die max. Anzahl Zeichen die der String enthalten kann
<code>capacity()</code>	liefert die Anzahl Zeichen, die ohne Realozierung von Speicher aufgenommen werden können
<code>reserve(anz)</code>	reserviert für den String Speicherplatz für min. anz Zeichen, sofern nicht schon genug Platz zur Verfügung steht

7.4.3 Modifizierende Zugriffe

Funktion	Bedeutung
operator = ()	weist einen String zu
assign()	Zuweisung eines Strings mit verschiedenen Varianten
swap(string)	Vertauscht den String für den die Funktion aufgerufen wurde mit dem String der als Argument übergeben wird
operator +=()	hängt am Ende eines bestehenden Strings einen String an
append()	verketten von Strings in verschiedenen Varianten
insert()	einfügen von Strings in verschiedenen Varianten
replace()	ersetzen von Strings in verschiedenen Varianten
remove()	löschen von Zeichen in verschiedenen Varianten
copy()	kopiert Zeichen des Strings, für den die Funktion aufgerufen wird, in eine Zeichenfolge die als Argument übergeben wird (ebenfalls in verschiedenen Varianten angeboten)
resize()	ändert die Grösse des Strings

7.4.4 Such-Funktionen

Funktion	Bedeutung
find(), rfind()	sucht einen Teilstring im String für den die Funktion aufgerufen wurde (in verschiedenen Varianten angeboten)
find_first_of(), find_last_of()	sucht die ersten bzw. letzten Zeichen im String (in verschiedenen Varianten angeboten)
find_first_not_of(), find_last_not_of()	sucht die ersten bzw. letzten Zeichen die nicht den als Argument definierten Zeichen im String entsprechen (in verschiedenen Varianten angeboten)

7.4.5 Vergleichsoperatoren

Für die Stringklasse stehen zunächst die standard Vergleichsoperatoren zur Verfügung (==, !=, <, >, <= und >=). Dabei können die zu vergleichenden Strings, Objekte der Klasse string (einer oder zwei) oder C-Strings (const char *, nur einer) sein. Daneben existieren verschiedene Varianten zum Testen der Gleichheit mit der Funktion compare().

7.4.6 Direkter Zeichenzugriff

Funktion	Bedeutung
at(index)	Zugriff auf das Zeichen an der Stelle index
operator[index]	Zugriff auf das Zeichen an der Stelle index
c_str()	liefert die Zeichen des Strings als Array, das mit einem String-Ende-Zeichen abgeschlossen ist
data()	liefert die Zeichen des Strings als Array, das jedoch nicht zwingend mit einem String-Ende-Zeichen abgeschlossen ist

7.4.7 Teil- und Summen-Strings

Funktion	Bedeutung
----------	-----------

operator +()	verketten von zwei Strings (in verschiedenen Varianten angeboten)
substr()	erzeugt einen Teilstring (in verschiedenen Varianten angeboten)

7.4.8 Ein- und Ausgabe-Funktionen

Funktion	Bedeutung
operator << ()	gibt den String auf dem Ausgabe-Strom aus
operator >> ()	liest einen String vom Eingabe-Strom bis zum ersten Trennzeichen (Whitespace) ein.
getline()	liest eine Zeichenfolge vom Eingabe-Strom bis zum ersten Newline ein (in verschiedenen Varianten angeboten)

Ein paar Stringobjekte bearbeiten.

```

1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  void main()
6  {
7      string stringText1;
8      string stringText2;
9      string stringText3;
10
11     stringText1 = "Mustertext";
12     cout << stringText1 << endl << endl;
13     cout << "Ist Text 1 leer? "
14          << (stringText1.empty() ? "Ja" : "Nein" )
15          << endl;
16     cout << "Ist Text 2 leer? "
17          << (stringText2.empty() ? "Ja" : "Nein" )
18          << endl;
19
20     stringText2 = stringText1;
21     stringText3 = stringText1;
22     stringText2 += string( 10, ' ' ) + stringText3;
23     cout << stringText2 << endl << endl;
24
25     if ( stringText1 == stringText2 ) {
26         cout << stringText1
27              << " == "
28              << stringText2
29              << endl << endl;
30     }
31     else {
32         cout << stringText1
33              << " != "
34              << stringText2
35              << endl << endl;
36     }
37
38     cout << "size():      " << stringText1.size() << endl;
39     cout << "length():    " << stringText1.length() << endl;
40     cout << "max_size(): " << stringText1.max_size() << endl;
41     cout << "capacity(): " << stringText1.capacity() << endl;

```

```
42     cout << endl;
43 }
```

liefert folgendes Ergebnis

```
44     Mustertext
45
46     Ist Text 1 leer? Nein
47     Ist Text 2 leer? Ja
48     Mustertext          Mustertext
49
50     Mustertext != Mustertext          Mustertext
51
52     size():      10
53     length():    10
54     max_size():  4294967293
55     capacity():  31
```

7.5 Ein- und Ausgabe mit StreamKlassen

7.5.1 Allgemeine I/O Klassen und Objekte

Die hauptsächliche Klassenhierarchie für die I/O-Operationen sieht wie folgt aus:

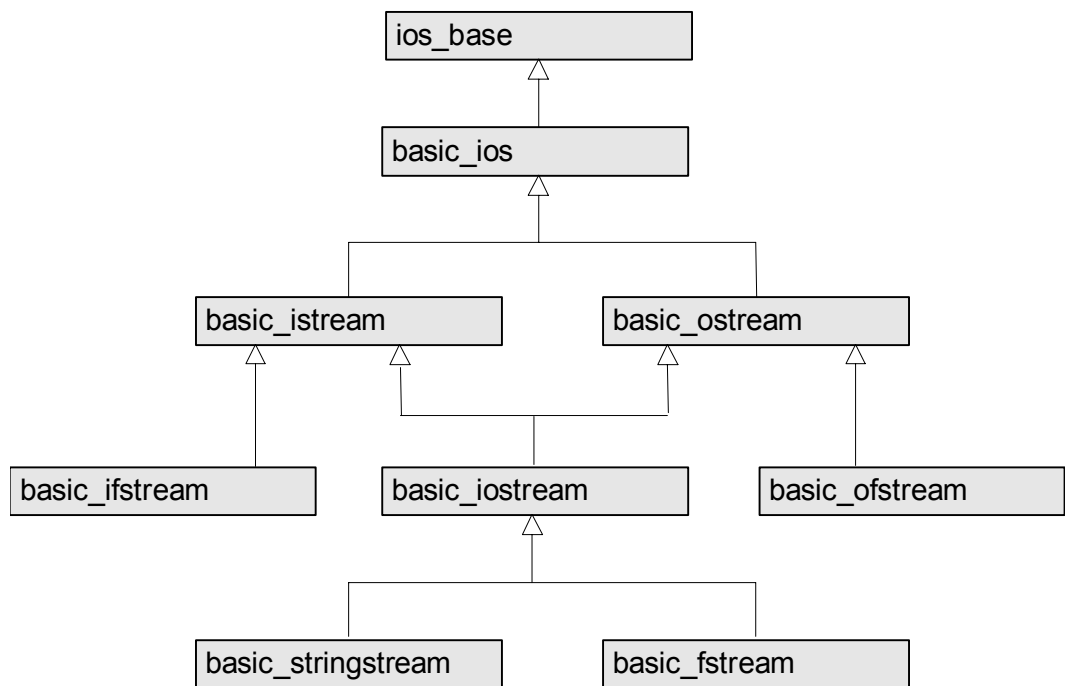


Abbildung 7- 8 Klassendiagramm der stream Klassen der STL.

In der Basisklasse `ios_base` sind die prinzipiellen Eigenschaften aller Stream-Klassen, wie Status- und Format-Flags sowie Funktionen zur Bearbeitung der Flags definiert.

In der davon abgeleiteten Template-Klasse `basic_ios` werden dagegen prinzipielle Eigenschaften aller Streamklassen, die vom Format der Zeichen abhängig sind, definiert. Dazu gehören die Definitionen der Standardbuffer.

Die Template-Klassen `basic_istream` und `basic_ostream` definieren Objekte von denen Zeichen gelesen bzw. geschrieben werden²:

- `cin`: Stellt das Objekt für den Standard-Eingabekanal dar (entsprechend der C-Variable `stdin`)
- `cout`: Stellt das Objekt für den Standard-Ausgabekanal dar (entsprechend der C-Variable `stdout`)
- `cerr`: Stellt das Objekt für den Standard-Ausgabekanal für Fehler dar (entsprechend der C-Variable `stderr`). Die Ausgaben auf `cerr` werden nicht gepuffert.
- `clog`: Analog `cerr`, jedoch erfolgen die Ausgaben gepuffert. (In C ist dafür keine entsprechende Variable definiert).

² Für internationalisierung sind entsprechende Objekte `wcin`, `wcout`, `wcerr`, `wclog` definiert.

7.5.2 Standardoperationen

Die Operatoren >> und << bilden die Standardoperationen für die Ein- resp. Ausgabe in Bezug auf die Stream-Objekte. Sie sind für alle fundamentalen Datentypen (inkl. char * und string) überladen. Somit steht eine einfache Art zur Ein- (Operator >>) resp. Ausgabe (Operator <<) von Daten zur Verfügung.

Ein- und Ausgabe verschiedener Datentypen

```
1      #include <iostream>
2
3      int    iWert = 100;
4      float fWert = 3.1415;
5      char   cZeichen = 'A';
6      char   *Zeichenkette = "Ausgabe einer Variablen";
7
8      ...
9      // diverse Ausgaben
10     cout << "Ausgabe eines statischen Textes" << endl;
11     cout << "Ausgabe von " << "zwei statischen Texten" << endl;
12     cout << Zeichenkette << endl;
13
14     // mögliche Darstellung einer langen Ausgabezeile
15     cout << "Mischen von Typen "
16           << Zeichenkette
17           << " "
18           << iWert
19           << " "
20           << cZeichen
21           << endl;
22
23     // weitere Ausgaben
24     cout << "Adresse von *Zeichenkette: "
25           << (void*)Zeichenkette << endl;
26
27     // Ein und Ausgaben
28     cout << "Bitte integer Wert eingeben: ";
29     cin >> iWert;
30     cout << "Eingegebener Wert: " << iWert << endl;
31     cout << "Bitte float Wert eingeben: ";
32     cin >> fWert;
33     cout << "Eingegebener Wert: " << fWert << endl;
34     ...
```

liefert folgendes Ergebnis

```
35     Ausgabe eines statischen Textes
36     Ausgabe von zwei statischen Texten
37     Ausgabe einer Variablen
38     Mischen von Typen Ausgabe einer Variablen 100 A
39     Adresse von *Zeichenkette: 45ed7c
40     Bitte integer Wert eingeben: 6
41     Eingegebener Wert: 6
42     Bitte float Wert eingeben: 5.7
43     Eingegebener Wert: 5.7
```

Mit der Klasse stringstream lassen sich recht praktische Zusammenstellungen von Strings aus heterogenen Daten machen.

```
44     #include "stdafx.h"
```

```

45 #include <string>
46 #include <sstream>
47 #include <iostream>
48 using namespace std;
49
50 int main(int argc, char* argv[])
51 {
52     basic_stringstream< char > strSQL_Statement;
53
54     // enhance precision, standard is 6
55     strSQL_Statement.precision(10);
56
57     // create SQL Statement into a stream
58     strSQL_Statement << "INSERT INTO TABLE ("
59                     << " FIELD1, FIELD2, FIELD3,"
60                     << " FIELD4, FIELD5, FIELD6 )"
61                     << " VALUES ( '"
62                     << "Text 1"           << "' , '"
63                     << "Text 1"           << "' , '"
64                     << "Text 1"           << "' , '"
65                     << 74.123456789012345 << " , '"
66                     << 38.55              << " , '"
67                     << "Text"             << "' ) ";
68
69     // define params to execute SQL
70     cout << "SQL: " << strSQL_Statement.str() << endl;
71
72     return 0;
73 }

```

7.5.3 Funktionen für die Eingabe

Funktion	Parameter	Verwendung
get()	void	Liefert als Rückgabewert das nächste gelesene Zeichen oder EOF zurück.
get()	char &	Weist das nächste Zeichen dem übergebenen Parameter vom Typ char zu, und liefert den Stream zurück. Nach der Operation kann dem Stream der Zustand über die Operation entnommen werden.
get()	char *s, int anz, char ende = '\n'	Liest bis zu anz-1 Zeichen in den String s ein. Spätestens mit dem Zeichen ende wird jedoch die Operation abgebrochen. Der String s wird durch die Operation mit '\0' terminiert. Das Zeichen ende wird nicht übernommen.
getline()	char *s, int anz, char ende = '\n'	Entspricht obiger Variante von get(), mit dem Unterschied, dass das Zeichen ende übernommen wird.
read()	char *s, int anz	Liest bis zu anz Zeichen in den String s ein.
gcount()	keine	Liefert die Anzahl gelesene Zeichen des letzten Lesebefehls.
ignore()	optional int anz = 1, optional int ende = EOF	Überliest max. anz Zeichen bis ende (Zeichen oder EOF) auftritt.
peek()	keine	Liefert das nächste Zeichen aus dem Stream oder EOF, ohne es effektiv auszulesen. Beim nächsten Lesen wäre das Zeichen also immer noch vorhanden.

putback()	char	Stellt ein Zeichen in den Eingabestrom zurück, damit es beim nächsten Mal erneut gelesen wird.
-----------	------	--

Bei allen obigen Funktionen werden Trennzeichen (Whitespaces) nicht überlesen.

7.5.4 Funktionen für die Ausgabe

Funktion	Parameter	Verwendung
put()	char	Gibt das übergebene Zeichen als nächstes Zeichen aus und liefert den Stream zurück. Nach der Operation kann dem Stream der Zustand des Schreibvorganges entnommen werden.
write()	char *s, int anz	Gibt anz Zeichen des Strings s aus.
flush()	keine	Leert den Ausgabepuffer, indem alle darin vorhandenen Zeichen ausgegeben werden.

7.5.5 Streamstatus

Streams verfügen über einen Zustand. Darin wird festgehalten ob die Ein- resp. Ausgabe sinnvoll ist oder allenfalls eine Fehlersituation eingetreten ist.

7.5.5.1 Konstanten für den Zustand des Streams

Für die Kontrolle des Zustandes sind folgende Konstanten definiert:

Konstante	Bedeutung
goodbit	alles in Ordnung
eofbit	Ende der Daten ist erreicht
failbit	es ist ein Fehler beim letzten Vorgang aufgetreten
badbit	es ist ein fataler Fehler eingetreten, der Streamstatus ist nicht definiert

7.5.5.2 Funktionen für die Kontrolle des Streams

Funktion	Bedeutung
good()	true wenn alles in Ordnung ist (ios::goodbit gesetzt)
eof()	true bei end of file (ios::eof gesetzt)
fail()	true bei Fehler (ios::failbit oder ios::badbit gesetzt)
bad()	true bei fatalem Fehler (ios::badbit gesetzt)
rdstate()	liefert die Kombination der gesetzten Flags.
clear()	löscht alle gesetzten Flags

7.5.6 Formatdefinitionen

Für die Definition verschiedener IO-Formate werden in der Klasse ios_base verschiedene Komponenten definiert. Eine der vordefinierten Komponenten dient dazu, die Formatflags zu führen. Innerhalb der Formatflags gibt es Gruppen von Flags, die so manipuliert werden können, dass jeweils nur ein Flag einer Gruppe gesetzt werden kann (Bitfelder).

Mit den Funktionen *setf()* und *unsetf()* können ein oder mehrere Flags gesetzt oder gelöscht werden, indem die Stati einzeln oder mit einer binären oder-Verknüpfung angegeben werden.

Setzen von Flags für die Formatdefinitionen

```
1 // Flags "showpos" und "uppercase" setzen
2 cout.setf( ios::showpos | ios::uppercase );
3
4 // Ausgabe in hex-Format als einziges Format setzen
5 cout.setf( ios::hex, ios::basefield );
6
7 // Flag "uppercase" zurücksetzen
8 // cout.unsetf( ios::uppercase );
```

7.5.6.1 Feldbreite, Füllzeichen und Ausrichtung

Die Feldbreite einer Ausgabe kann mit der Funktion *width()* definiert werden. Wird der Funktion ein *int* Parameter übergeben, wird die Feldbreite auf den übergebenen Wert gesetzt. Bei leerer Argumentenliste liefert die Funktion die aktuell gesetzte Feldbreite.

Das Füllzeichen einer Ausgabe kann mit der Funktion *fill()* definiert werden. Das standardmässig verwendete Füllzeichen ist ein ' ' (Blank).

Für die Definition der Ausrichtung einer Ausgabe innerhalb einer Feldbreite wird das Bitfeld *adjustfield* benutzt:

Bitfeld	Flag	Bedeutung
ios::adjustfield	ios::left	linksbündig
	ios::right	rechtsbündig
	ios::internal	Vorzeichen linksbündig, Wert rechtsbündig

7.5.6.2 Vorzeichen und Grossbuchstaben

Für die Definition von Vorzeichen und Grossbuchstaben werden folgende Flags verwendet:

Bitfeld	Bedeutung
ios::showpos	positives Vorzeichen mit ausgeben
ios::uppercase	Ausgaben von hex-Values und Buchstaben in numerischen Werten (wissenschaftliche Notationen) ausschliesslich in Grossbuchstaben

Diverse formatierte Ausgaben

```
1  #include <iostream>
2  #include <string>
3  #include <iomanip>
4  using namespace std;
5
6  void Out1( void );
7  void Out2( void );
8
9  // main
10 void main()
11 {
12     cout.fill( '.' );
13     Out1();
14     cout.setf( ios::left );
15     Out2();
16     cout.setf( ios::right );
17     Out1();
18     cout.setf( ios::internal );
19     Out2();
20     cout << endl << string( 50, '-' )
21         << setfill( ' ' ) << endl << endl;
22     cout << "Value: " << 123456789.12345 << endl;
23     cout << "Value: " << showpos << 123456789.12345 << endl;
24     cout << "Value: " << showpos
25         << uppercase << 123456789.12345 << endl;
26 }
27
28 // Ausgabe mit Elementfunktion
29 void Out1( void )
30 {
31     cout.width( 6 );
32     cout << -42 << " ";
33     cout.width( 6 );
34     cout << 0.12 << " ";
35     cout.width( 6 );
36     cout << "Q" << " ";
37     cout.width( 6 );
38     cout << 'Q' << endl;
39 }
40
41 // Ausgabe mit Manipulator
42 void Out2( void )
43 {
44     cout << setw( 6 ) << -42 << " "
45         << setw( 6 ) << 0.12 << " "
46         << setw( 6 ) << "Q" << " "
47         << setw( 6 ) << 'Q' << endl;
48 }
```

liefert folgendes Ergebnis

```
49  ...-42 ..0.12 .....Q Q
50  -42... 0.12.. Q..... Q
51  ...-42 ..0.12 .....Q Q
52  ...-42 ..0.12 .....Q Q
53
54  -----
55
56  Value: 1.23457e+008
57  Value: +1.23457e+008
58  Value: +1.23457E+008
```

7.5.6.3 Zahlensysteme

Bei Integerausgaben kann das Zahlensystem für die Ausgabe mitbestimmt werden. Dabei kann noch unterschieden werden, ob für die verschiedenen Zahlensysteme Prefixangaben bei der Ausgabe erfolgen (Bsp: 0x bei hex):

Bitfeld	Flag	Bedeutung
ios::basefield	ios::oct	Ein- / Ausgabe octal
	ios::dec	Ein- / Ausgabe dezimal (default)
	ios::hex	Ein- / Ausgabe hexadezimal
-	ios::showbase	Zahlensystem kennzeichnen

7.5.6.4 Gleitkommanotation

Für die Definition einer Ausgabe mit Gleitkommawerten wird zunächst spezifiziert, ob die Ausgabe in der Dezimal- oder in der Exponentialdarstellung zu geschehen hat:

Bitfeld	Flag	Bedeutung
ios::floatfield	ios::fixed	Dezimaldarstellung der Werte
	ios::scientific	Exponentialdarstellung der Werte

Anschliessend kann durch die Angabe der Präzision (Anzahl anzuzeigender Ziffern), die Ausgabevorbereitung abgeschlossen werden (optional):

Elementfunktion	Bedeutung
precision()	Genauigkeit setzen oder abfragen

Diverse formatierte Ausgaben

```
1  #include <iostream>
2  #include <string>
3  #include <iomanip>
4  using namespace std;
5
6  void Out( void );
7
8  // main
9  void main()
10 {
11     cout.setf( ios::uppercase );
12     cout << "Value: " << dec << 255 << endl;
13     cout << "Value: " << oct << 255 << endl;
14     cout << "Value: " << hex << 255 << endl << endl;
15     cout.setf( ios::showbase );
16     cout << "Value: " << dec << 255 << endl;
17     cout << "Value: " << oct << 255 << endl;
18     cout << "Value: " << hex << 255 << endl << endl;
19     cout.unsetf( ios::showbase | ios::uppercase );
20     Out();
21     cout.setf( ios::showpoint );
22     Out();
23     cout.unsetf( ios::showpoint );
24     cout.setf( ios::fixed );
25     Out();
26     cout.unsetf( ios::fixed );
27     cout.setf( ios::scientific );
28     Out();
29     cout.unsetf( ios::scientific );
30 }
31
32 void Out( void )
33 {
34     cout.precision( 2 );
35     cout << "Value: "
36         << 412.0 << "      "
37         << 0.0123456789
38         << endl;
39     cout.precision( 6 );
40     cout << "Value: "
41         << 412.0 << "      "
42         << 0.0123456789
43         << endl << endl;
44 }
```

liefert folgendes Ergebnis

```
45 Value: 255
46 Value: 377
47 Value: FF
48
49 Value: 255
50 Value: 0377
51 Value: 0XFF
52
```

```

53 Value: 4.1e+002      0.012
54 Value: 412          0.0123457
55
56 Value: 4.1e+002      0.012
57 Value: 412.000       0.0123457
58
59 Value: 412.00         0.01
60 Value: 412.000000     0.012346
61
62 Value: 4.12e+002      1.23e-002
63 Value: 4.120000e+002  1.234568e-002

```

7.5.6.5 Allgemeine Flags

Neben den obigen Flags existieren noch folgende allgemeingültige Formatflags:

Flag	Bedeutung
<code>ios::skipws</code>	Führende Trennzeichen (Whitespaces) überlesen
<code>ios::unitbuf</code>	Ausgabepuffer nach der Ausgabeoperation leeren
<code>ios::stdio</code>	Ausgabepuffer nach jedem Textzeichen leeren

7.5.6.6 Setzen und Abfragen von Flags

Nebst den Funktionen `setf()` und `unsetf()` ermöglicht die Funktion allgemeine Funktion `flags()` das Auslesen und das Setzen aller Flags.

Abfragen und Setzen aller Flags für die Formatdefinitionen

```

64 ios::fmtflags oldflags;
65
66 ...
67 // aktuelle Flags sichern
68 oldflags = cout.flags()
69
70 // Beeinflussung der Flags
71 cout.setf( ios::hex | ios::uppercase );
72 cout << hex << x << endl;
73 ...
74
75 // Zustand mit alten Formatflags wieder herstellen
76 cout.flags( oldflags );

```

7.5.7 Dateizugriff

7.5.7.1 Klassen und Objekte für Filezugriff

Die "stream-Klassen" können auch für den Zugriff auf Daten der externen Speicher verwendet werden. Zu diesem Zweck existieren in der Klassenhierarchie der IO-Klassen weitere abgeleitete Klassen:

- Die Klasse ifstream besteht für Dateien aus denen gelesen wird.
- Die Klasse ofstream besteht für Dateien in die geschrieben wird.
- Die Klasse fstream besteht für Dateien in die geschrieben und von denen gelesen wird.

7.5.7.2 Dateien öffnen

Eine Datei auf dem externen Speicher wird geöffnet, in dem beim Erzeugen des Streamobjektes als Parameter ein File angegeben wird. Ein Streamobjekt wird dabei ohne Angabe von Flags standardmässig zum Schreiben und zum Lesen geöffnet (In Abhängigkeit der Klasse von welcher das Objekt erzeugt wurde). Nach dem Öffnen des Files sollte immer als erstes der Zustand des Streamobjektes geprüft werden (Fehlerhandling).

Erstellen eines Streamobjektes für den Zugriff auf den externen Speicher (Funktion nur innerhalb eines Funktions-Member zu benutzen)

```
1  ...
2  // Datei "MYFILE.TXT" öffnen
3  ofstream datei ( "MYFILE.TXT" );
4
5  // Fehlerhandling
6  if ( ! datei ) {
7      // Fehlerbehandlung
8      ...
9  }
10 else {
11     // Ausgaben
12     datei << "Schreibe Text in das File" << endl;
13     ...
14 }
15 ...
```

Erstellen eines Streamobjektes für den Zugriff auf den externen Speicher (Funktion über die Grenzen von Funktions-Member hinweg benutzbar). Hier könnte auch ein auto_ptr eine sinnvolle Anwendung ergeben.

```
16 ...
17 // Datei "MYFILE.TXT" öffnen
18 ofstream* datei = new ofstream ( "MYFILE.TXT" );
19 ...
20
```

Für die genauere Spezifikation einer Fileoperation sind untenstehende Flags definiert:

Flag	Bedeutung
ios::in	Lesen (Default bei ifstream)
ios::out	Schreiben (Default bei ofstream)
ios::app	Anhängen

<code>ios::ate</code>	an das Ende positionieren (at end)
<code>ios::trunc</code>	alten Dateiinhalt löschen
<code>ios::binary</code>	Datei im binären Modus öffnen ³

Erstellen eines Streamobjektes für den Zugriff auf den externen Speicher mit Flags

```

21  ...
22  // Ausgaben an file anhängen
23  ofstream datei ( "MYFILE.TXT", ios::app );
24  ...

```

7.5.7.3 Wahlfreie Positionierung in einem File

Für die Positionierung des Filepointers sind folgende Funktionen und Flags in den Klassen definiert:

Funktion	Bedeutung
<code>seekg()</code>	setzt eine Leseposition (get)
<code>seekp()</code>	setzt eine Schreibposition (put)
<code>tellg()</code>	liefert die Leseposition
<code>tellp()</code>	liefert die Schreibposition

Flag	Bedeutung
<code>ios::beg</code>	relativ zum Dateianfang (begin)
<code>ios::cur</code>	relativ zur aktuellen Position (current)
<code>ios::end</code>	relativ zum Dateiende (end)

Positionierungen des Filepointers in einem File

```

25  long pos;
26  ...
27
28  // aktuelle Position merken
29  pos = datei.tellg();
30  ...
31
32  // absolute Positionierung zur gemerkten Stelle
33  datei.seekg( pos );
34  ...
35
36  // relative Positionierung an den Fileanfang
37  datei.seekp( 0, ios::beg );
38  ...
39
40  // relative Positionierung 20 Zeichen ab laufender Position
41  datei.seekp( 20, ios::cur );
42  ...
43
44  // 10 Zeichen vor dem Dateiende positionieren
45  datei.seekp( -10, ios::end );
46  ...

```

³ Ist nicht bei allen Compilern vorhanden (noch nicht genormt)

Wird bis ans Dateiende gelesen, wird das eof-Flag gesetzt. Damit ist der Stream in keinem korrekten Zustand mehr. Um bei einer erneuten Positionierung wieder lesen oder schreiben zu können, muss das Zustandsflag mit `stream.clear()` zurückgesetzt werden.

7.5.7.4 Dateien lesen und schreiben

Für das Lesen und Schreiben von Informationen in resp. aus einer Datei werden die Standard Funktions-Member der Klasse `stream` verwendet.

7.5.7.5 Dateien schliessen

Das explizite schliessen einer Datei erübrigt sich, solange die Dateiobjekte innerhalb eines Funktions-Members erstellt worden sind. Sie werden dann automatisch bei der Destruktorverarbeitung geschlossen.

Erstellen und schliessen eines expliziten Streamobjektes

```
1  ...
2  // Datei "MYFILE.TXT" öffnen
3  ofstream* pdatei = new ofstream ( "MYFILE.TXT" );
4  ...
5
6  // Datei schliessen
7  delete pdatei;
```

7.5.7.6 Umleiten der Standardkanäle

Für die Dateiumlenkung ist es notwendig, dass die Standardstreams in Files umgelegt werden können. Dies geschieht, indem ein bestehendes Streamobjekt dem umzuleitenden Standardstream zugewiesen wird:

Umleiten des Standardstream `cout` in eine Datei

```
8  #include <iostream>
9  #include <fstream>
10 using namespace std;
11
12 // main
13 void main()
14 {
15     cout << "Zeile vor der Umlenkung" << endl;
16
17     // merken des alten streambuffers und anlegen einer datei
18     streambuf *strm_buffer = cout.rdbuf();
19     ofstream umlenkDatei( "Umlenk.txt" );
20
21     // Umlenkung durchführen
22     cout.rdbuf( umlenkDatei.rdbuf() );
23     cout << "Zeile 1 während der Umlenkung" << endl;
24     cout << "Zeile 2 während der Umlenkung" << endl;
25     cout << "Zeile 3 während der Umlenkung" << endl;
26     cout << "Zeile 4 während der Umlenkung" << endl;
27
28     // umlenkung zurückstellen
29     cout.rdbuf( strm_buffer );
30     cout << "Zeile nach der Umlenkung" << endl;
31 }
```

32

liefert folgendes Ergebnis

Auf dem Bildschirm:

33 Zeile vor der Umlenkung
34 Zeile nach der Umlenkung

In der Datei:

35 Zeile 1 während der Umlenkung
36 Zeile 2 während der Umlenkung
37 Zeile 3 während der Umlenkung
38 Zeile 4 während der Umlenkung

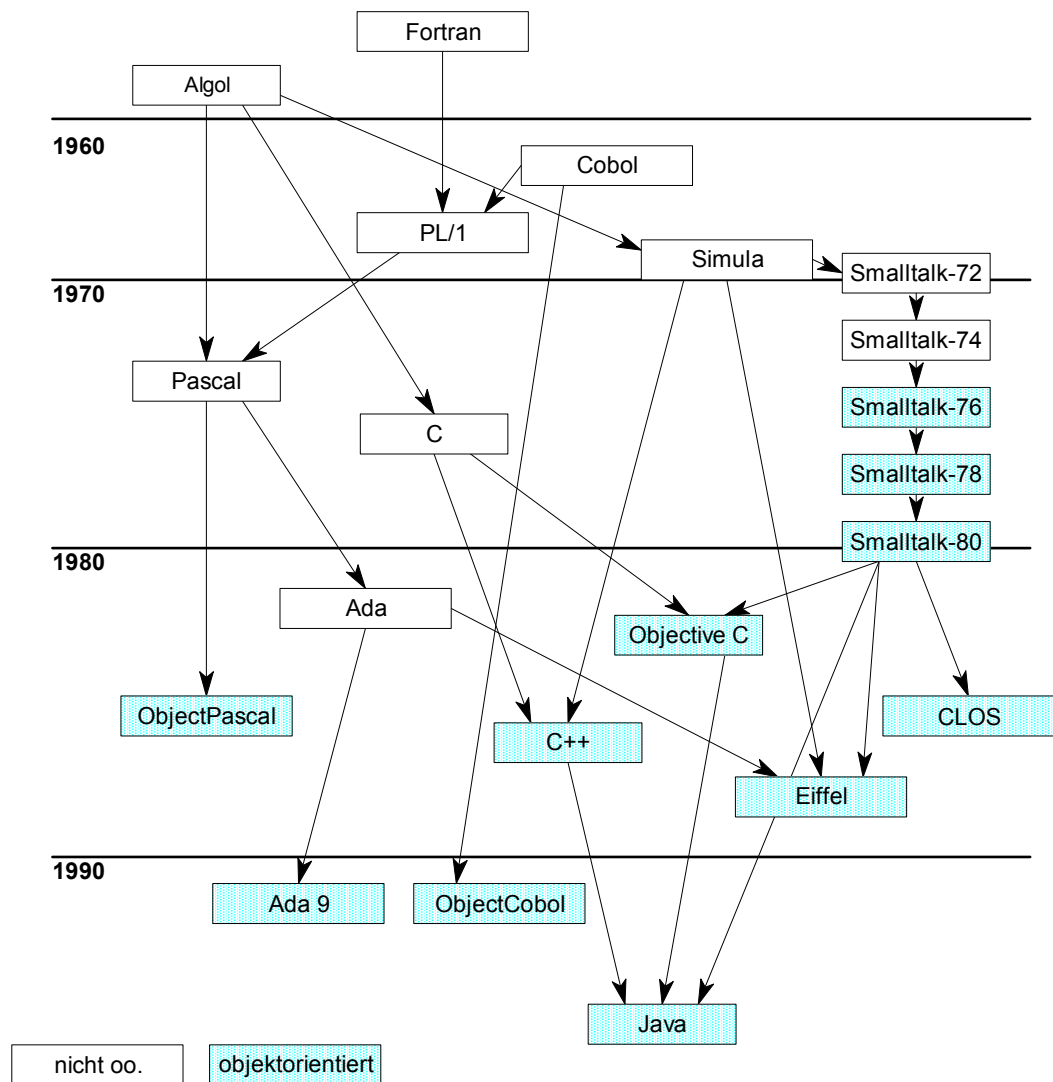
7.6 Weitere Klassen

Neben den in diesem Kapitel beschriebenen Klassen und Methoden definiert die STL noch folgende hier z.Z. noch nicht näher beschriebene Funktionalitäten:

- Ein- und Ausgabe mit String-Stream-Klassen
- Eine numerische Bibliothek für die Bearbeitung von komplexen Zahlen und Valarrays (Vektor und Matrizen-Berechnungen)
- Ein Klassen-Template für die Bearbeitung von Bitsets
- Internationalisierungen

8 Anhang

8.1 Geschichtliche Entwicklung der Sprache C++



8.2 Literaturhinweise

- [1] Sharam Hekmatpour
C++, Der Einstieg für den C-Programmierer
Carl Hanser Verlag München Wien
mit Diskette
- [2] Martin Schader, Stefan Kuhlins
Programmieren in C++, Einführung in den Sprachstandard C++ Version 3.0
Springer Verlag
- [3] Bjarne Stroustrup
Die C++ Programmiersprache, erweitert um Entwürfe zur ANSI-/ISO Standardisierung
2. Auflage 1992, 4. überarbeiteter Nachdruck 1994
Addison-Wesley, ISBN: 3-89319-386-3
- [4] Josuttis, Nicolai
Objektorientiertes Programmieren in C++, von der Klasse zur Klassenbibliothek
1. Auflage 1994
Addison-Wesley
- [5] Stephen Prata
C++, Einführung in die objektorientierte Programmierung
3. Auflage 1993
Waite Group Press, te-wi Verlag
mit Diskette
- [6] Grady Booch
Object-oriented Analysis and Design
2. Auflage 1994
The Benjamin/Cummings Publishing Company, Inc., ISBN: 0-8053-5340-2
- [7] Grady Booch
Objektorientierte Analyse und Design
1. Auflage 1994
Deutsche Übersetzung von [6]
Addison-Wesley, ISBN: 3-89319-673-0
- [8] Nicolai Josuttis
Die C++ Standardbibliothek,
1. Auflage 1996
Addison-Wesley, ISBN: 3-8273-1023-7
- [9] Bernd Oestereich
Objektorientierte Softwareentwicklung mit der Unified Modeling Language,
3. Auflage 1997
R. Oldenbourg Verlag München Wien, ISBN: 3-486-24319-5
- [10] Herbert Schildt
STL Programming from the GROUND UP
1999
Mc GrawHill, ISBN: 0-07-882507-5

*** End of Document ***

8.3 Stichwortverzeichnis

A

Abgeleitete Klasse : siehe auch Klasse
Abstraktion : 1-6
Aggregationen : 1-17
Aggregationsstrukturen : 4-18
argc : 2-36
argv : 2-36
Array : siehe Felder
asm : 2-2
Assoziationen : 4-16
Ausdruck : 2-17
 Auflösung : 2-18
 Prioritäten : 2-18
 Schreibweise : 2-17
Ausnahmebehandlung : Exceptionhandling
auto : 2-2, 2-33
auto_ptr : 7-8
Auto-Pointer : 7-8

B

Basisklasse : siehe auch Klasse
Bezeichner : 2-1
Beziehungsstrukturen : 4-16
bool : 2-2
break : 2-2, 2-26

C

case : 2-2
Casting : 2-20
 Funktionelles : 2-20
 Konventionelles : 2-20
catch : 2-2, 6-2, 6-3
char : 2-2, 2-8
class : 2-2, 2-14, 3-2
Concurrency : siehe Gleichzeitigkeit
const : 2-2
const_cast : 2-2
Containing : 4-18
continue : 2-2, 2-26
Copy-Konstruktor : 3-18
 Schreibweise : 3-19
 Verhalten by ref. : 3-19, 3-22
 Verhalten by value : 3-19
Corba : 1-1

D

Datentypen
 Abgeleitete : 2-9
 Boolscher : 2-9
 Eigene : 2-14

Erweiterter Zeichensatz : 2-9
Felder : 2-9
Ganzzahl : 2-8
Gleitkomma : 2-8
Leerer : 2-9
Referenz : 2-11
Strukturen : 2-14
Zeiger : 2-10
DCOM : 1-1
default : 2-2
Definition : 2-16
Deklaration : 2-16
delete : 2-2, 2-17, 3-14
Deque <> : 7-13
Destruktor : 3-28
 Schreibweise : 3-28
 standard : 3-17
do : 2-2
do while : 2-24
double : 2-2, 2-8
dynamic_cast : 2-2
dynamische Polymorphie : 4-9
Dynamische Speichernutzung : 2-16

E

Einbindung von C Code in C++ : 2-35
Einsprungpunkt : 2-36
else : 2-2
Elternklasse : siehe auch Klasse
Encapsulation : siehe Kapselung
enum : 2-2
env : 2-36
Exceptionhandling : 1-17, 6-1
 Auslösen : 6-3
 Auswerten : 6-3
 Fehlerklassen : 6-2
 Konzept : 6-1
 Schlüsselworte : 6-2
 Schnittstellendefinition : 6-4
 Weitergeben einer Exception : 6-4
exit() : 6-4
explicit : 2-2
Explizite Typenumwandlung : Casting
extern : 2-2, 2-33

F

false : 2-2
Felder : 2-9
float : 2-2, 2-8
for : 2-2, 2-25
friend : 2-2
Friend
 Funktionen : 3-29
 Klasse : 3-32
Funktion
 Standard : 2-32

- Statische : 3-10
- Überladen : 3-16
- Funktionen
 - Rekursive : 2-32
 - Überladung : 2-31
- Funktionen : 2-27
 - Argumente : 2-28
 - Friend : 3-29
 - Rückgabetyt : 2-28
 - Spezifizierer : 2-28
 - Standardargumente : 2-29
 - Syntax : 2-27
 - virtuelle : 4-9
 - Zeiger auf .. : 2-30
- Funktionsüberladung : 2-31

G

- Geltungsbereich : 2-33
- Gleichzeitigkeit : 1-12
- goto : 2-2, 2-26

H

- Header
 - cstdint : 7-6
 - cstdlib : 7-6
 - limits : 7-9
- Headerdatei : 2-34
- Headerdateien : 7-3
- Heap : 3-4
- Hierarchie : 1-9

I

- if : 2-2
- if-else : 2-22
- Implementation : 2-34
- Inheritance : siehe Vererbung
- Initialisiererlisten : 4-13
- inline : 2-2
- Inline : 3-12
- Instanziierung : 3-3
 - dynamische : 3-4
 - statische : 3-3
- int : 2-2, 2-8

K

- Kapselung : 1-7
- Kindklasse : siehe auch Klasse
- Klasse : 1-3
 - Ableitung : 4-2
 - Basisklasse : 4-2
 - Deklaration : 3-1, 4-2
 - Friend : 3-32
 - Implementation : 3-1, 3-3
 - Interface : 3-2
 - Name : 3-2
 - Parametrisierte : 0-1

- Vererbung : 4-2
- Zugriffsrechte : siehe Zugriffsspezifizierer
- Zugriffsspezifizierer : 3-7
- Zugriffsspezifizierer : 4-2
- Klassen : 2-14
 - Abstrakte : 4-14
 - Template : 5-3
- Klassenableitung : 1-16
- Klassenbibliotheken : 2-32
- Klassenbildung : 1-16
- Kommentare : 2-1
- Konstruktor : 3-17
 - copy : siehe Copy-Konstruktor
 - Schreibweise : 3-17
 - standard : 3-17

L

- Lebensdauer : 2-33
- Lexikalische Konventionen : 2-1
- Liste<> : 7-13
- Literale : 2-4
 - Boolean : 2-6
 - Character : 2-5
 - Floating : 2-5
 - Integer : 2-4
 - String : 2-6
- long : 2-2, 2-8

M

- main() : 2-36
- make_pair() : 7-7
- Map<> : 7-16
- max() : 7-5
- Mehrfachvererbung : 4-7
- min() : 7-5
- Modularisierung : 1-8, 2-34
- Modulbildung : 1-16
- Multimap<> : 7-16
- Multiset<> : 7-14
- mutable : 2-2

N

- namespace : 2-3
- Namespace : 7-3
- nested classes : 4-18
- new : 2-3, 2-17, 3-14

O

- Oberklasse : siehe auch Klasse
- Objekt : 1-3
 - Definition : 1-4
 - dynamisches : 3-6
 - Identität : 1-5
 - statisches : 3-5
 - Status : 1-4
 - Verhalten : 1-4

Objektspeicherung : 1-13
operator : 2-3
Operator : 3-13
 new & Delete : 3-14
 Zeichen : 3-13
 Zuweisung : 3-17, 3-25
 Zuweisung Schreibweise : 3-25
Operatoren : 2-6
Operatorprioritäten : 2-18, 2-19
Override : 4-9

P

pair : 7-7
Parametrisierbare Klassen : 1-17
Persistenz : siehe Objektspeicherung
Pointer : siehe Zeiger
Polymorphie
 dynamische : 4-9
 statische : 3-16
Polymorphismus : 1-17
Prioritäten : 2-18
Prioritäten Operatoren : 2-18, 2-19
private : 2-3, 3-7, 4-6
protected : 2-3, 3-7, 4-6
public : 2-3, 3-7, 4-6
pure virtual : 4-11

R

Referenz : 2-11
register : 2-3, 2-33
reinterpret_cast : 2-3
Rekursion : 2-32
return : 2-3, 2-26

S

Schlüsselwörter : 2-2
set terminate() : 6-4
Set<> : 7-14
short : 2-3, 2-8
signed : 2-3, 2-8
sizeof : 2-3
Smart-Pointer : 7-8
Softwareentwicklung : 1-1
Standard Template Library : 7-10
Standardbibliothek : 7-1
 Exceptions : 7-4
 Hilfsfunktionen : 7-5
 Utilities : 7-5
 Vergleichsoperatoren : 7-5
Standardfunktion : 2-32
static : 2-3, 2-33
Static Daten-Member : 3-9
static_cast : 2-3
STL : 7-10
 Algorithmen : 7-20
 Assoziative Container : 7-14
 Container : 7-10
 Dateizugriff : 7-36
 Fileoperation : 7-37

Filepointer : 7-37
I/O
 Dateiumlenkung : 7-38
 Exponentialdarstellung : 7-33
 Feldbreite : 7-31
 Flags : 7-35
 Formatdefinitionen : 7-31
 fstream : 7-36
 Füllzeichen : 7-31
 ifstream : 7-36
 Integerausgaben : 7-33
 Klassen : 7-27
 ofstream : 7-36
 setf() : 7-35
 unsetf() : 7-35
I/O/Standardoperationen : 7-28
Iteratoren : 7-19
Sequentielle Container : 7-12
Standardstream : 7-38
stream : 7-27
 Status : 7-30
 string : 7-23
Strings : Zeichenketten
struct : 2-3, 2-14, 3-2
Strukturen : 2-14
swap() : 7-6
switch : 2-3
switch - case : 2-23
Symbolische Konstanten : 2-16
Syntaktische Beschreibung : 2-1

T

template : 2-3
Template
 Klassen : 5-3
 lokale Variablen : 5-2
 Mehrere Typenvariablen : 5-2
 Typenumwandlung : 5-1
 Überladen von Funktionen : 5-2
Template Funktionen : 5-1
Templates : 1-17, 0-1
terminate() : 6-4
this : 2-3, 3-6
throw : 2-3, 6-2, 6-3, 6-4
Trennzeichen : 2-1
true : 2-3
try : 2-3, 6-2, 6-3
Typ : 1-3
typedef : 2-3, 2-14
typeid : 2-3
Typenprüfung : 1-11
Typenumwandlung
 automatisch bei Templates : 5-1
Typing : siehe Typenprüfung

Ü

Überladen
 Templatefunktionen : 5-2

U

Umwandlung von Standardtypen : 2-19
union : 2-3, 2-14
unsigned : 2-3, 2-8
Unterklasse : siehe auch Klasse
using : 2-3, 7-2
Using : 4-20

V

Vector<> : 7-12
Vererbung : 1-10, 1-16, 4-1
 Zugriffsrechte : 4-4
virtual : 2-3, 4-10
Virtuelle Basisklassen : 4-14

void : 2-3, 2-9
volatile : 2-3

W

wchar_t : 2-3
wchart : 2-9
while : 2-3, 2-24

Z

Zeichenketten : 2-12
Zeiger : 2-10
Zugriffsrechte : 1-16
 Vererbung : 4-4