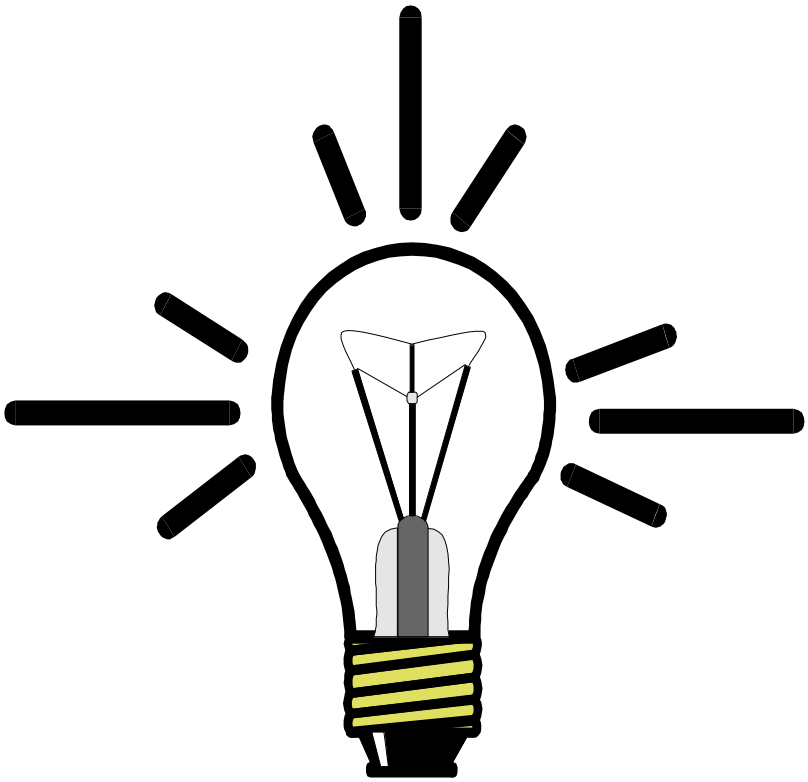
Wer C sagt, muß auch PlusPlus sagen!

Die Sprache
Der Kurs
Die Erleuchtung



Grundkurs C++ Programmierung

STEINBEIS-TRANSFERZENTRUM SOFTWARETECHNIK ESSLINGEN

Inhaltsverzeichnis

Vorwort

Das vorliegende Manuskript soll dem interessierten Leser einen einfachen Einstieg in die Sprache C++ ermöglichen. Daher wurde insbesondere in den einführenden Kapiteln mehr Wert auf anschauliche Erklärungen gelegt als auf die Art von abstrakten Definitionen, die man in Sprachreferenzen findet.

Sollten Sie in der Praxis dennoch einmal in die Situation kommen, etwas ganz genau wissen zu müssen, befragen Sie bitte die Handbücher zu Ihrem Compiler oder benutzen Sie die im Literaturverzeichnis empfohlenen Bücher. Besonders im Hinblick auf Bibliotheksfunktionen, von denen viele noch nicht standardisiert sind, sind ohnehin kaum allgemeingültige Aussagen möglich.

C++ baut auf C auf. Um den Umfang dieses Kurses im Rahmen zu halten, wird die Kenntnis dieser Programmmiersprache vorausgesetzt. Trotzdem werden im Verlaufe des Kurses immer wieder Details der Sprache C - in erster Linie zu Vergleichszwecken - wiederholt.

Um einen möglichst großen Vorteil aus dieser Einführung zu ziehen, sollten Sie folgendes beachten: Halten Sie beim Lesen hin und wieder einmal inne und denken Sie nach. Machen Sie ein paar Experimente mit Ihrem Compiler (es soll ja auch Spaß machen). Und provozieren Sie auch ruhig einmal mit Absicht ein paar Fehler, beobachten Sie die Resultate. Durch die Erfahrungen, die Sie auf diese Weise sammeln, können Sie später Fehler leichter vermeiden bzw finden. Machen Sie sich nicht immer nur klar, wie etwas funktioniert, sondern auch, was dabei schiefgehen kann. Lernen Sie die Formalismen nicht nur auswendig, sondern versuchen Sie, die Hintergründe zu verstehen!

Nutzen Sie die Beispiele! Sie sind sehr einfach und bieten viel Spielraum für Experimente. Hier können Sie viel lernen, wenn Sie sich die Zeit nehmen, um sich mit ihnen zu befassen. Eine Programmiersprache lernt man nur dann, wenn man sie auch "spricht", und sich mit ihr auseinandersetzt. Der Vergleich zu Fremdsprachen ist hier sowohl beabsichtigt als auch berechtigt. Und ganz ähnlich wie der Umgang mit einem dicken Wörterbuch will auch das Auffinden von gesuchten Informationen in Handbüchern gelernt sein. Wer diese Kunst beherrscht und stets weiß, wonach er sucht und wo er es findet, wird immer weniger hilflos, unabhängiger und schneller sein. Natürlich haben Sie all das längst gewußt, aber haben Sie auch danach gehandelt ?

Was ist C++ ?

C++ ist eine Weiterentwicklung der verbreiteten Programmiersprache C und zu dieser weitgehend kompatibel. Bis auf wenige Ausnahmen können vorhandene Bibliothekspakete ohne Änderungen in C++ weiterverwendet werden.

Während zu Beginn von C++ Umsetzer von C++ in C-Code vorherrschten, wobei der C-Code anschließend in Maschinencode übersetzt wurde, gibt es heute echte Compiler auf fast allen Rechnern.

Standen beim Entwurf von C die hardwarenahe Programmierung und Performance-Gesichtspunkte im Vordergrund, so orientieren sich die Erweiterungen in C++ in erster Linie an den Erfordernissen der Programmierpraxis und verfolgen das Ziel, dem Programmierer bei der Erstellung guter Programme zu helfen. Gut in diesem Sinne heißt nicht nur, daß die Programme lauffähig sein sollen! Sie sollen stabil laufen und gut zu lesen und zu pflegen sein. Dieses Ziel wird in erster Line dadurch unterstützt, daß der Compiler dem Programmierer einen ganzen Satz von neuen Möglichkeiten bietet, die letztlich zur Einsparung einer Menge an Schreibarbeit, aber auch zu einem höheren Abstraktionsniveau führen.

C++ Compiler eröffnen nicht unbedingt grundlegend neue Möglichkeiten. Vieles läßt sich auch mit C oder anderen Programmiersprachen realisieren, jedoch mit einem Vielfachen an Verwaltungsaufwand, der bei C++ einfach vom Compiler übernommen wird. Vergessen Sie aber dennoch nicht, daß nach wie vor Sie die Verantwortung für das Programm tragen. Je mehr Dinge vom Compiler im Hintergrund erledigt werden, desto schwerer sind Fehler zu finden, die auf diesen Hintergrundeffekten beruhen. Auf der anderen Seite haben Sie jedoch durch die Arbeitsersparnis mehr Zeit, um über das, was Sie tun, vorher nachzudenken.

Die mit Sicherheit herausragendste Neuerung an C++, von der wohl auch am meisten geredet wird, ist das Klassenkonzept. Dieses Konzept öffnet nun auch für C-Fans das Tor zur objektorientierten Programmierung (OOP). Daneben bietet C++ aber noch eine ganze Anzahl von Leckerbissen, die auch ohne Einsatz von OOP zur Geltung kommen. Diese werden Ihnen im Kapitel „Verbesserungen und Änderungen im prozeduralen Sprachanteil“ vorgestellt.

Um es jedoch bereits hier auf den Punkt zu bringen: Die wesentlichen Änderungen in C++ sind

- eine effiziente Unterstützung der Datenabstraktion
- die Unterstützung objektorientierten Programmierens

Die Unterstützung für Datenabstraktion bedeutet die Einführung von Techniken für die Definition und Anwendung neuer Typen mit einem strengen Typkonzept - vergleichbar mit der Sprache Ada, wobei für die alten Datentypen der Sprache C ein strenges Typkonzept nicht besteht.

Plakativ kann man sagen:

C++ =	C mit deltas + Datenabstraktion + Objektorientierung

Die beabsichtigte Bedeutung des Namens C++ ist heute nicht mehr eindeutig nachvollziehbar. Die geläufigste Version ist, daß der Name aus dem C und dem Inkrementoperator der Sprache C entstand.

Bei C++ besteht mehr als bei allen anderen Sprachen die Gefahr, daß der Programmierer im alten Stil weiter programmiert, so wie er es bei C gelernt hat, da er in C++ alle Sprachelemente, die er bereits von C kennt, nahezu unverändert vorfindet.

Ein Start der objektorientierten Programmierung etwa mit Smalltalk wäre nicht so gefährlich, da man dort alles neu lernen müßte und auf keine eingefahrenen Denkmuster aufsetzen könnte.

Also, auch wenn alle alten C-Tricksereien nach wie vor möglich sind, sind doch viele in C++ einfach nicht mehr angemessen.

Kurzer Überblick über den Sprachumfang

Die folgende Übersicht zeigt, daß C++ weit mächtiger als C ist. C++ gilt heute als komplizierter als Ada! Dies bedeutet natürlich, daß zahlreiche neue Begriffe und neue Sprachmittel vorgestellt werden müssen.

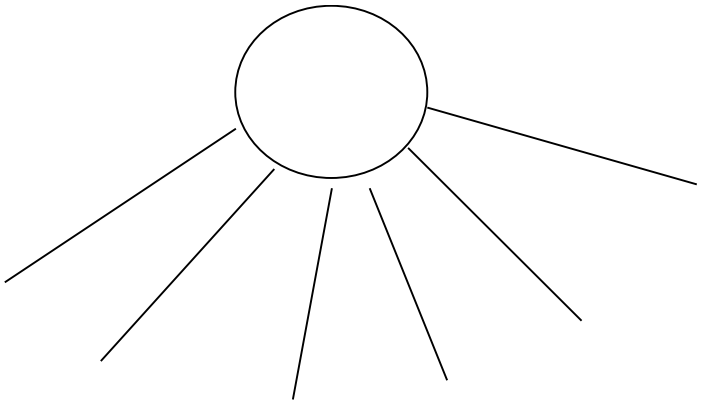
C

Makros
Pointer
Strukturen
Funktionen

C++

Makros
Pointer
Strukturen
Funktionen
Default-Parameter
Überladene Funktionen
Inline-Funktionen
Referenzen
Klassen
private und protected Elemente
Konstruktoren und Destruktoren
benutzerdefinierte Operatoren
Vererbung
Mehrfachvererbung
Templates
Exceptions
etc.

1 Konzepte der objektorientierten Programmierung



C++ ist nichts Neues unter der Sonne!

Dieser Kurs befaßt sich mit C++. Der Autor von C++, Stroustrup, hat versucht, in C++ wichtige Ideen von Ada und von Smalltalk bzw. SIMULA zu übernehmen. Von Ada kam unter anderem das strenge Typkonzept, von Smalltalk bzw. SIMULA das Konzept der Klassen und das Vererbungsprinzip.

```
Vorbilder:

Smalltalk, SIMULA, Ada
```

Smalltalk wird manchmal als der Vater der objektorientierten Programmierung angesehen. Allerdings gibt es einen Vorgänger, SIMULA - eine Programmiersprache zur Simulation - die bereits die Grundlagen der Objektorientierung enthielt. SIMULA diente zur diskreten Simulation auf digitalen Rechneranlagen.

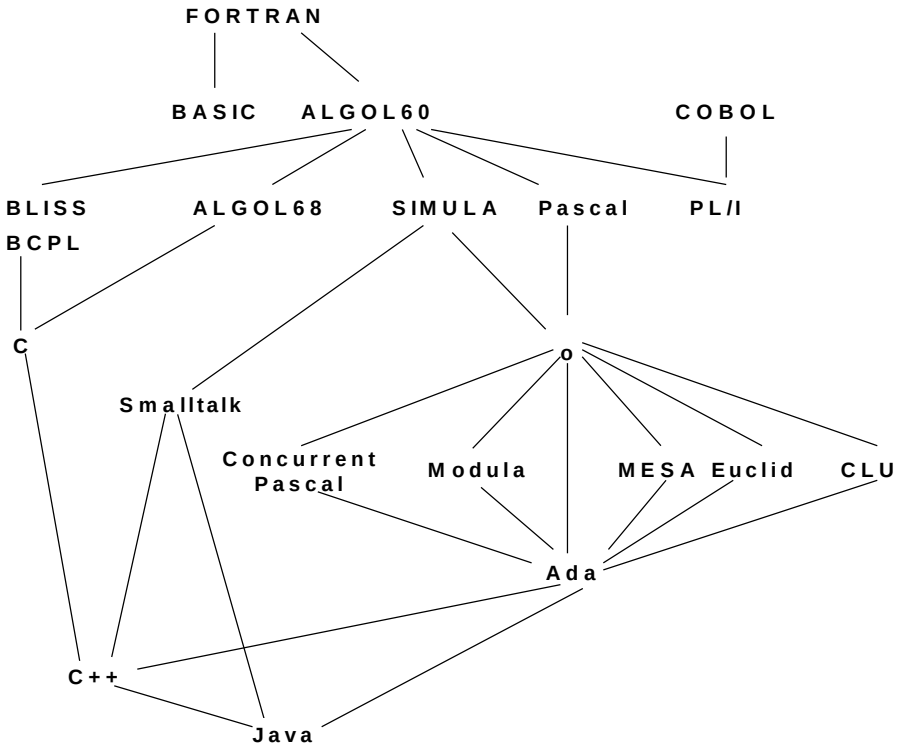


Bild 1 - 1 C++ in der Verwandtschaftstafel einiger höherer Programmiersprachen

1.1 Objekte - die Bausteine des objektorientierten Ansatzes

Wiederverwendbarkeit durch Objekte als Software-IC

Eine Idee des objektorientierten Ansatzes ist es, ein System aus einer Menge unveränderlicher Teile aufzubauen. Kommt eine neue Funktionalität hinzu, so treten neue Teile auf, die alten bleiben aber stabil. Hauptziel ist dabei die Erstellung und Wiederverwendung von Software-Bausteinen (Software-IC's). Ein Beispiel für die Wiederverwendung ist die Einbindung existierender Klassenbibliotheken in das Software-System.

Problembereich und Lösungsbereich

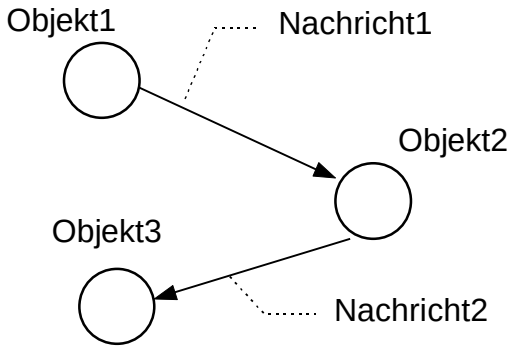
Der Ansatz der Objektorientierung basiert darauf, Objekte der realen Welt zu modellieren und diese Objekte bis zum Systementwurf beizubehalten und zu verfeinern. Ein Objekt entspricht einer Entität der realen Welt. Eine Entität hat im Rahmen des betrachteten Problems eine definierte Bedeutung. Eine **Entität** kann z.B. ein **Gegenstand** wie z.B. ein Auto sein, aber auch ein **abstraktes Konzept** wie z.B. ein Vertrag. Ein Objekt stellt eine Abstraktion einer solchen Entität dar, die als ein nützliches Modell (nichtrelevante Details werden weggelassen) für den betrachteten Ausschnitt der realen Welt dienen kann. Entscheidend für den

objektorientierten Ansatz ist nicht das objektorientierte Programmieren, sondern das Denken in Objekten vom Start des Projektes an.

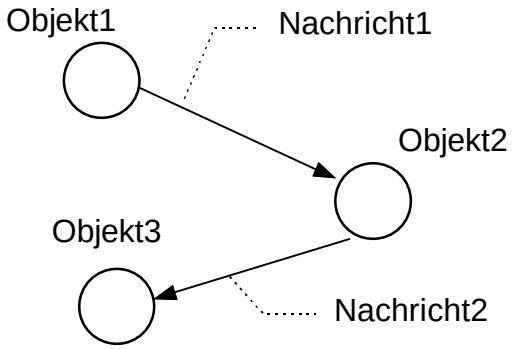
Die Formulierung eines Modells erfolgt bei objektorientierten Techniken also in Konzepten und Begriffen der realen Welt anstelle in computertechnischen Konstrukten. Dies bedeutet, daß die anwendungsorientierte Sicht gegenüber einer computerorientierten Sicht im Vordergrund steht.

Bei der objektorientierten Modellierung versucht man zunächst, die Objekte im **Problembereich** zu erkennen. Der Problembereich ist ein Ausschnitt aus der realen Welt, den man für seine Anwendung betrachtet. Diese Objekte im Problembereich haben miteinander Wechselwirkungen, z.B. gibt ein Benutzer ein Buch an die Bibliothek zurück.

In der Programmierung bildet man die erkannten Objekte und ihre Wechselwirkungen auf Objekte in der Programmiersprache und eine Kommunikation zwischen den Objekten in der Programmiersprache ab.



Objekte des Problembereichs
in der realen Welt



Objekte des Problembereichs
in der Programmiersprache

In der herkömmlichen Software-Entwicklung hingegen steht das Denken in Funktionen, d.h. in Algorithmen, im Vordergrund. Man denkt in Haupt- und Unterprogrammen, die Daten bearbeiten und nicht Gegenstände der realen Welt.

Bei einer objektorientierten Entwicklung denkt man lange Zeit hauptsächlich im **Problem-Bereich (problem domain)**, mit anderen Worten in der Begriffswelt des Kunden, erst später im **Lösungsbereich,** d.h. in Form der technischen Implementierung.

Natürlich kommen in späteren Schritten, nämlich beim Systementwurf, auch Objekte implementierungstechnischer Art dazu, z.B. Objekte, die einen Start-Up des Systems durchführen.

Kapselung

Hinter den Methoden der objektorientierten Programmierung verbirgt sich ein neues Denkmodell, das sich von den bisher gebräuchlichen sehr stark unterscheidet. Es beruht im Kern darauf, daß man Daten und die Funktionen, die mit ihnen umgehen, nicht mehr getrennt, sondern als Einheit betrachtet. Sie verschmelzen zu einem Objekt. Im Idealfall sind alle Daten eines Objekts gekapselt. Auf sie kann nur durch die Methoden des Objektes zugegriffen werden. In der objektorientierten Programmierung wird statt des Begriffes Funktion meist der Begriff Methode verwendet.

Diese Kapselung ist eines der wichtigsten Konzepte der objektorientierten Programmierung. Es besteht in diesem Falle nicht die Trennung zwischen Daten und Funktionen wie in der prozeduralen Programmierung z.B. in C.

Da die Daten einer Kapsel im Idealfall nur durch die Funktionen der Kapsel manipuliert werden können, sind sie nach außen nicht direkt sichtbar. Man spricht dann auch von Information Hiding. Ein solches Objekt tritt mit seiner Umwelt im Idealfall nur über wohldefinierte Schnittstellenfunktionen in Kontakt und unterstützt auf diese Art und Weise das Geheimnisprinzip.

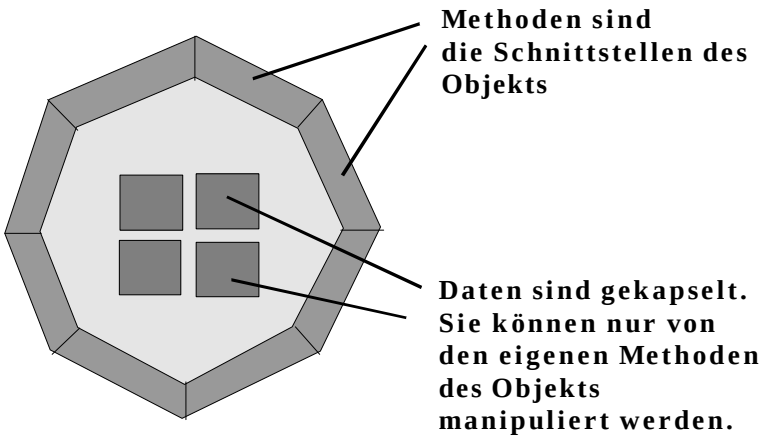


Bild 1 - 2 Daten und Methoden - die Bestandteile von Objekten

Diese **Prinzipien der Kapselung und des Information Hidings** haben einen wichtigen Hintergrund: Die Außenwelt soll möglichst keine Möglichkeit haben, Daten im Inneren des Objekts direkt zu verändern, und so möglicherweise unzulässige Zustände herbeizuführen. Das Verstecken sämtlicher Daten und Routinen in einer "Kapsel", und die Durchführung der Kommunikation mit der Außenwelt durch eigene Schnittstellenfunktionen, bringt dem Programmierer den Vorteil, daß er bei der Implementierung der "Innereien" sehr viele Freiheiten hat. Dem Benutzer bringt dies im Gegenzug den Vorteil, daß er sich a) nicht darum kümmern muß, was genau im Inneren des Objekts wie passiert, und b) daß er immer mit der neuesten Version des Objekts arbeiten kann. Er ist ja nicht vom speziellen inneren Aufbau des Objekts abhängig, und der Programmierer kann diesen immer wieder optimieren, ohne

Komplikationen befürchten zu müssen. Nur die Schnittstellen müssen gleich bleiben. Bereits an dieser Stelle können Sie erkennen, wie wichtig die Schnittstellen sind. Es ist also unbedingt nötig, diese sorgfältig zu entwerfen. Um trotzdem ein Höchstmaß an Flexibilität zu gewährleisten, ist es jedoch immer noch möglich, Teile eines Objekts so zu deklarieren, daß sie ohne weiteres direkt von außen zugänglich sind. Zumindest für die Schnittstellenfunktionen muß diese Eigenschaft in jedem Fall zutreffen.

Einen Großteil des oben Gesagten wie z.B. die Kapselung hätte man auch mit bisherigen Methoden und viel Disziplin beim Programmieren erreichen können, nämlich durch geeigneten Entwurf von Modulen. Aber die Möglichkeiten der OOP gehen noch weiter, da sie das Vererbungsprinzip beinhaltet (siehe unten).

1.2 Grundeigenschaften objektorientierter Sprachen am Beispiel Smalltalk

Das System, das die Objektorientierung vom theoretischen Standpunkt aus wohl am besten realisiert, ist Smalltalk. Auch wenn im folgenden C++ behandelt werden soll, ist es dennoch sehr aufschlußreich, sich zunächst mit dem Konzept von Smalltalk zu befassen. Smalltalk gilt als die Sprache, die die Objektorientierung am klarsten implementiert. Mit ihr lassen sich objektorientierte Ansätze am besten verwirklichen.

Smalltalk ist eine mächtige objektorientierte Programmiersprache mit einer komfortablen Programmierumgebung. Die Sprache Smalltalk umfaßt die Begriffe [1]:

?? Objekt (object)
?? Methode (method)
?? Botschaft (message)
?? Klasse (class)
?? Exemplar (instance)

1.2.1 Objekte

Ein **Objekt** ist eine Gruppierung von Information zusammen mit der Beschreibung, wie diese Information verarbeitet werden soll. Somit enthält ein Objekt Daten und Funktionen, die auf diese Daten angewendet werden können. Mit anderen Worten, ein Objekt vereint in sich sowohl die **Datenstruktur** als auch das **Verhalten**, das durch **Methoden (Funktionen)** beschrieben wird.

Objekte in Smalltalk können beispielsweise sein: Zahlen, Zeichen, Zeichenketten, Grafiken, Warteschlangen etc. Welche Operationen angewendet werden können, hängen von der Art des Objektes ab. Die **Operationen auf den Daten des Objektes** werden als eine **„innere Fähigkeit“** des Objektes betrachtet. So können z.B. dem Objekt „Zahl“ die arithmetischen Operationen, dem Objekt „Zeichenkette“ das

Vergleichen, Verlängern und Ändern zugeordnet werden. Schickt man beispielsweise eine Botschaft an ein Objekt „Zahl“ mit dem Selektor (Methodenname) „add“, so könnte dies bedeuten, daß eine Zahl mit einer anderen Zahl summiert werden soll. Das Senden der gleichen Botschaft an das Objekt „Warteschlange“ könnte hingegen verursachen, daß der Warteschlange ein Element hinzugefügt wird.

1.2.2 Methoden

Eine **Methode** ist die Beschreibung einer einzelnen Funktion bzw. Operation, die für ein Objekt definiert ist. Eine Methode hat mit einer Prozedur bzw. Funktion gemeinsam, daß sie eine Folge von Anweisungen beschreibt, die von einem Prozessor - in der Regel zur Manipulation von Daten - ausgeführt werden können.

Ein Objekt enthält Attribute und Methoden. **Attribute** definieren die **Datenstruktur** der Objekte, die **Methoden** bestimmen das **Verhalten** der Objekte. Der **Zustand** eines Objektes ist festgelegt durch den momentanen Wert seiner Attribute. Der Zustand eines Objektes wird verändert durch die Methoden des Objektes.

Der Begriff des Zustandes eines Objektes wird in zweierlei Bedeutung gebraucht. Einmal in obigem Sinne. Jedes Attribut hat Werte aus seinem Wertebereich. Jede Kombination von Attributwerten stellt einen Zustand dar. Ein solcher Zustand soll hier als **mikroskopischer Zustand** eines Objektes bezeichnet werden. Ein Objekt kann sehr viele mikroskopische Zustände haben. Von Bedeutung bei der Modellierung sind jedoch diejenigen Zustände eines Objektes, die für eine Applikation eine Bedeutung haben. Diese Zustände sollen hier **makroskopische Zustände** genannt werden. Als Beispiel sollen genannt werden die Zustände „Fahren“, „Türen auf“, „Warten auf Knopfdruck“ eines Objektes Fahrstuhl. Solche Zustände sind von Bedeutung, wenn man Zustandsübergänge von Objekten im Rahmen des dynamischen Modells betrachtet - wenn beispielsweise ein Fahrstuhl vom Zustand "Warten auf Knopfdruck" in den Zustand "Fahren" übergeht. Ein solcher Zustand resultiert durch Wechselwirkungen mit der Umgebung z.B. mit der Mechanik bei „Türen auf“ oder mit dem Motor bei „Fahren“ oder durch Warten auf ein Ereignis z.B. das Drücken eines Bedienungsknopfes.

Es gibt jedoch einen wichtigen Unterschied einer Methode in Smalltalk zu den Prozeduren und Funktionen der Sprachen der dritten Generation wie FORTRAN, Pascal oder C: Eine Methode kann nicht eine andere Methode aufrufen und deren Abarbeitung anstoßen. Methoden können nur an Objekte geschickt werden. Es ist hierbei nicht möglich, Methoden von Objekten zu isolieren. Dies gilt in Smalltalk, nicht aber in C++.

1.2.3 Botschaften

In Smalltalk, nicht jedoch in C++, erfolgt der Aufruf einer Methode durch das Senden einer **Botschaft** an das Objekt.

In Smalltalk bilden **Botschaften** die einzige Möglichkeit für eine Kommunikation zwischen Objekten. Eine Botschaft enthält die folgende Information:

- Namen des Empfängers
- Angabe der Methodennamen, die bestimmen, welche Manipulationen an dem Empfänger-Objekt durchzuführen sind. Diese Methodennamen werden auch **Selektoren** genannt.
- eventuell noch Argumente für die Methoden

Damit bittet eine Botschaft um die Ausführung einer Operation, d.h. sie aktiviert die Methode, die Details der Ausführung sind eine „innere Fähigkeit“ des Objektes selbst.

Damit hat man durchaus eine Ähnlichkeit zum Aufruf von Prozeduren in den herkömmlichen Programmiersprachen, denn auch Prozeduren werden mit ihrem Namen aufgerufen. Allerdings gibt es einen wichtigen Unterschied: zu einem Namen gibt es immer nur eine Prozedur. Im Gegensatz dazu kann die Botschaft mit dem gleichen Selektor an verschiedene Objekte gerichtet und von den verschiedenen Empfängern unterschiedlich interpretiert werden. Nicht die Botschaft bestimmt, was abläuft, sondern der Empfänger der Botschaft. Die Botschaft „add“ an eine Warteschlange könnte so aussehen:

```
Warteschlange add ElementNeu
```

Das heißt, daß die Botschaft außer dem Selektor add noch das Objekt ElementNeu, ein sogenanntes Botschafts-Argument enthält.

Durch das Senden einer Botschaft wird die Kontrolle über die durchzuführenden Aktionen an das in der Botschaft bezeichnete Objekt gegeben, welches die Botschaft interpretiert.

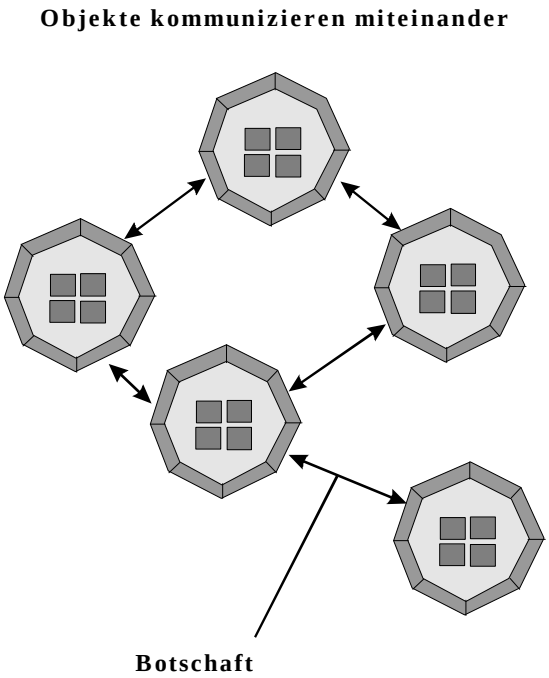


Bild 1 - 3 Informationsaustausch zwischen Objekten als lose gekoppelte Bausteine

Wenigstens in der Logik kommunizieren Objekte über **Nachrichten**. Beim Systementwurf kommt es wieder darauf an, was die Sprache bietet. Während Smalltalk das Nachrichtenkonzept unterstützt, wird es hingegen von C++ nicht unterstützt.

In Smalltalk wird eine Nachricht folgendermaßen an ein Objekt geschickt:

```
Object  message: argument
```

In C++ sieht der Aufruf folgendermaßen aus:

```
Object.message (argument)
```

Wenn ein Objekt eine Nachricht erhält, wird eine seiner Methoden aufgerufen. Dies bedeutet, daß eine Nachricht ein Auftrag an ein Objekt ist, der ein Objekt zum Handeln veranlaßt. Wie das Objekt handelt, ist Sache des Objektes. Die Implementierung der Funktion ist in der Regel nach außen nicht sichtbar.

Dieses nachrichtenbasierte Konzept ist nichts Besonderes, wenn man aus der Prozeßdatenverarbeitung kommt. Hier wird ein System in parallele Prozesse entworfen, die miteinander über Nachrichten (und ggfs. auch über Shared Memory) kommunizieren.

Ein Objekt einer Unterklasse kann dabei auf die Nachricht auch durch Aufruf einer Methode der Vaterklasse antworten, die es von der Vaterklasse geerbt hat.

1.2.4 Klassen

Eine **Klasse** ist die abstrakte Beschreibung einer Gruppe gleich strukturierter Objekte. Diese durch die Klasse beschriebenen Objekte werden Exemplare (der Klasse) genannt (engl. instances).

Zu beachten ist, daß mit der Beschreibung einer Klasse noch keine einzelnen Objekte aus dieser Klasse existieren. Objekte müssen erst erzeugt werden. Der Begriff „Erzeugung“ (eines Exemplars) wird weiter unten näher betrachtet.

Mit der Klassenbeschreibung wird ein Schema zur Bildung von Objekten dieser Klasse vereinbart. Dieses Schema enthält:

- den **Namen** der Klasse
- den internen Aufbau der **Datenelemente** von Objekten dieser Klasse
- alle **Methoden**, die auf Objekte dieser Klassen angewendet werden können.

Ein Beispiel für den Aufbau einer Klassenbeschreibung gibt Bild 1 - 4

Name:	Datum
Datenelemente:	tag <nat. Zahl; 1..31>
	monat <nat. Zahl; 1..12>
	jahr <nat. Zahl>
Methoden:	neues Datum <Datum, Diff>
	(Datum plus/minus Tage)
	TagesDiff <Datum1, Datum2>
	(Anzahl Tage zwischen zwei Daten)
	Wochentag <Datum>
	(Wochentagsbezeichnung eines Datums)
	heute (heutiges Datum)
	.
	.
	etc

Jedes Datum ist ein „Exemplar“ dieser Klasse

Bild 1 - 4 Beschreibung der Klasse „Datum“

Die Möglichkeit, Daten und Funktionen (Methoden) in Typen zusammenzufassen, von denen **Variablen** erzeugt werden können, die im Falle der Objektorientierung **Objekte** genannt werden, ist ein Merkmal objektorientierter Sprachen.

Klasse = Datentyp Objekt = Variable

Eine Klasse implementiert einen abstrakten (benutzerdefinierten) Datentyp. Ein abstrakter Datentyp ist definiert durch die Menge der zulässigen Operationen und durch seinen Wertebereich, wobei streng genommen die Definition der zulässigen Operationen ausreicht.

Diese Möglichkeit ist bei herkömmlichen Sprachen in der Regel nicht gegeben. Sprachen wie Modula-2 und Ada bieten jedoch auch die Möglichkeit der Kapselung von Daten in Modulen (Modula-2) bzw. Paketen (Ada), wobei nur über Schnittstellenprozeduren auf die gekapselten Daten zugegriffen werden kann. Module wie auch Pakete stellen jedoch prozedurale Einheiten dar und keine Datentypen.

1.2.5 Exemplare

Mit der Beschreibung einer Klasse existieren noch keine Objekte dieser Klasse (die sogenannten Exemplare). Das Bilden eines Exemplares kann durch das Senden einer bestimmten Botschaft an eine Klasse erfolgen. Dieser Vorgang wird **„Erzeugung“** eines Objektes genannt. In SMALLTALK selbst wird meist durch das Senden einer Botschaft NEW an eine Klasse ein Exemplar dieser Klasse erzeugt.

Es gibt aber auch Klassen, die nicht nur durch die Aufforderung NEW, sondern als Antwort auf andere Botschaften ein Exemplar erzeugen. Als Beispiel sei die Klasse „Datum“ aus Bild 1 - 4 erwähnt. Die Klasse antwortet auf die Botschaft „heute“ mit der Erzeugung eines Exemplars, welches das heutige Datum repräsentiert.

Die Menge der Botschaften, auf die ein Objekt antworten kann, wird **„Protokoll“** genannt. Nach außen ist von einem Objekt nur sein Protokoll sichtbar. Der interne Aufbau eines Objektes, der nach außen nicht sichtbar ist, besteht aus Daten und Methoden (Prozeduren).

Ein Objekt hat eine Menge von **Variablen**, die zu ihm gehören. Im Beispiel „Datum“ sind es die Variablen `tag`, `monat`, `jahr`. Die Methoden, die das Antwortverhalten auf empfangene Botschaften beschreiben, übernehmen die Rolle der Verarbeitungsprozeduren.

Alle Exemplare einer Klasse haben dieselben Methoden und dieselben Instanzvariablen. Die **Instanzvariablen** der Exemplare sind dieselben wie die der Klasse, allerdings in einer für jedes Exemplar individuellen Ausprägung.

Es gibt jedoch auch Variable, die für alle Exemplare einer Klasse (im Wert) gelten. Diese werden **„Klassenvariable“** genannt.

Unterschiedliche Begriffe in Smalltalk und C++

In Smalltalk umfaßt der Begriff Objekt als Oberbegriff die Begriffe Klasse und Instanz. In C++ werden Instanz und Objekt synonym verwendet.

1.3 Grafische Notation für die objektorientierte Modellierung

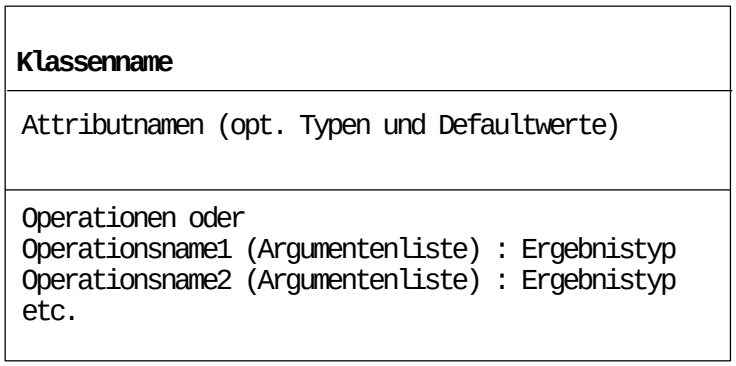
Es gibt keine allgemein akzeptierte grafische Notation für objektorientierte Konstrukte wie Klassen, Beziehungen zwischen Klassen, Objekte etc. Wir verwenden zur Zeit noch die Schreibweise nach der OMT-Methode von Rumbaugh et al. [12], stellen aber in Kürze auf die UML-Methode um. OMT heißt: Object Modelling Technique, UML heißt Unified Modelling Language.

Klasse **Objekt**

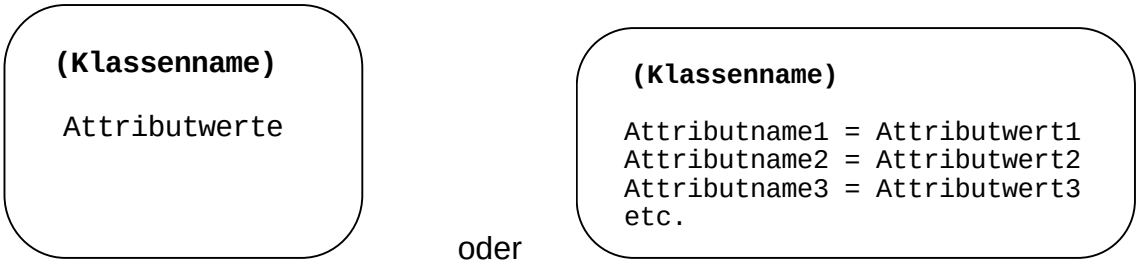
Eine Klasse wird symbolisiert durch ein Rechteck mit dem Klassennamen in fetter Schrift. Ein Objekt ist eine Instanz einer Klasse. Es wird visualisiert durch ein Kasten mit abgerundeten Ecken. Der Klassenname wird angegeben in fetter Schrift in runden Klammern. Die Objektnamen sind in normaler Schrift geschrieben.

Will man die Struktur einer Klasse sichtbar machen, so gibt man zuerst die Attribute und dann die Methoden an. Bei den Objekten stehen statt den Attributen die Zahlenwerte des entsprechenden Objekts, es kann auch geschrieben werden: Attributname = Wert. Die Operationen werden nicht angegeben, sie gelten für die ganze Klasse und müssen nicht mitgeschleppt werden.

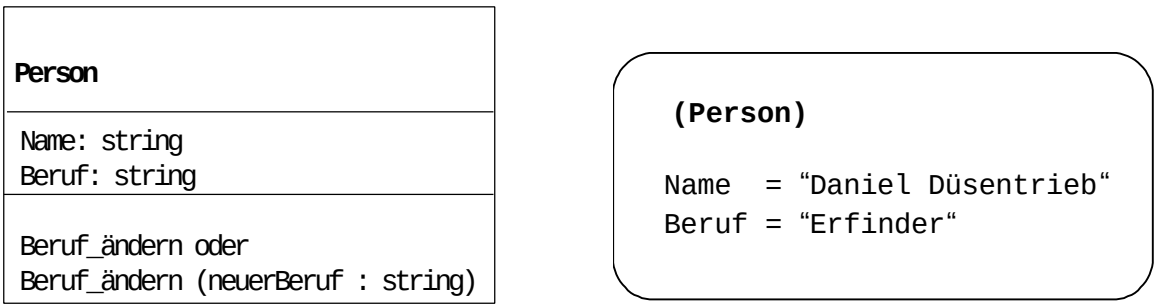
Notation für Klasse:



Notation für Objekt:



Beispiel:



Es muß erwähnt werden, daß die hier aufgeführte Notation bereits überholt ist, da die OMT-Notation weiter entwickelt wurde. Inzwischen werden auch die Objekte rechteckig gezeichnet. Da die neue Notation uns derzeit noch nicht mit Einzelheiten vorliegt, behalten wir vorläufig die in den Büchern veröffentlichte Notation bei.

1.4 Eigenschaften des objektorientierten Ansatzes

1.4.1 Prinzipien des objektorientierten Ansatzes in der Sichtweise von Rumbaugh

Was einen objektorientierten Ansatz ausmacht, ist nicht eindeutig festlegbar. Rumbaugh [3] zählt dazu:

- Klassifikation
- Identität
- Polymorphismus
- Vererbung

Für Booch (siehe unten) sind

- Abstraktion
- Kapselung
- Hierarchisierung
- Modularisierung

wichtig.

1.4.1.1 Klassifikation

Klassifikation bedeutet, daß Objekte derselben Struktur zu Klassen zusammengefaßt werden. Klassen sind die Baupläne der Objekte. Klassen sind Datentypen, Objekte sind Variablen von diesen Typen, auch **„Exemplare“** oder „**Instanzen**“ genannt.

Es gibt aber auch Klassen, zu denen keine Objekte erzeugt werden können. Dies sind die sogenannten **abstrakten Klassen**. Eine abstrakte Klasse wird eingeführt, um eine gemeinsame Vaterklasse für Unterklassen zu haben. Die Implementierung wenigstens einer der Methoden kann dabei nicht in der Vaterklasse erfolgen, sie ist spezifisch für jede Unterklasse und muß in der jeweiligen Unterklasse erfolgen. In der abstrakten Basisklasse (Vaterklasse) ist nur bekannt, daß es eine Methode gibt und wie ihre Schnittstelle aussieht.

1.4.1.2 Identität

Identität bedeutet, daß alle Objekte eine eigene Identität besitzen, also eigene Wesen sind, selbst wenn ihre Datenwerte (Attributwerte) identisch sind.

1.4.1.3 Polymorphismus

Polymorphismus bedeutet, daß gleiche Operationen sich in verschiedenen Klassen verschieden verhalten können. Die Operation **add** ist bei der Klasse „Warteschlange“ eine andere als bei der Klasse „ganze Zahl“.

Eine spezielle Implementation einer Operation in einer Klasse heißt eine Methode.

Eine objektorientierte Operation ist polymorph. Es kann mehr als eine Methode geben, die sie implementiert. Aufgrund des Namens der Operation und der Klasse wird automatisch die richtige Methode gewählt. Polymorphie bedeutet letztendlich, daß eine Operation vom Objekt selbst interpretiert wird - unter Umständen erst zur Laufzeit. Dieser Fall tritt bei Verwendung von Pointern auf Objekte auf, da man hier zur Compilierzeit noch nicht wissen kann, auf was für ein Objekt der Zeiger zur Laufzeit zeigt.

1.4.1.4 Vererbung

Ein Objekt einer Unterklasse erbt alle Attribute und Methoden der Oberklasse. Erbt ein Objekt nur von einer einzigen Oberklasse, so spricht man von **einfacher Vererbung**, erbt ein Objekt von mehreren Oberklassen, so spricht man von **mehrfacher Vererbung**. Es ist eine Frage der Programmiersprache, ob sie eine mehrfache Vererbung unterstützt.

Vererbung bezeichnet also die gemeinsame Verwendung von Attributen und Operationen in Vater-Sohn-Beziehungen. Eine Unterklasse erbt alle Eigenschaften (Attribute, Methoden) einer Oberklasse und fügt ihre eigenen individuellen Eigenschaften hinzu. Die Eigenschaften der Oberklasse müssen nicht in der Unterklasse wiederholt werden.

Mit dem Konzept der Vererbung können Wiederholungen im Entwurf vermieden werden. Gemeinsame Eigenschaften mehrerer Klassen werden in gemeinsamen Oberklassen ausgelagert. Dies führt zu mehr Übersicht und weniger Wiederholung.

1.4.2 Prinzipien des objektorientierten Ansatzes aus der Sichtweise von Booch

Für Grady Booch sind

?? Abstraktion
?? Einkapselung
?? Hierarchisierung
?? Modularisierung

Merkmale des objektorientierten Modells.

1.4.2.1 Abstraktion

Abstraktion bedeutet, sich auf das Wesentliche zu konzentrieren. Um mit der Komplexität der Realität fertig zu werden, braucht der Mensch immer Abstraktionen. Das Abstraktionsniveau hängt dabei stets von der Problemstellung ab.

In der Objektorientierung bedeutet dies, daß man in ein objektorientiertes Modell die für die Problemstellung erforderlichen Aspekte übernehmen muß. Den sinnvollen Abstraktionsgrad zu erkennen ist dabei das zentrale Problem.

1.4.2.2 Kapselung

Abstraktion, Kapselung und Information Hiding sind miteinander eng verwandt.

Die **Abstraktion** konzentriert sich auf das nach außen beobachtbare Verhalten eines Objektes. Der Begriff **Kapselung** konzentriert sich auf die Implementierung dieses Verhaltens. Damit müssen diejenigen Eigenschaften, die nach außen nicht sichtbar sein sollen, verborgen werden. Damit wird bei der Kapselung das Prinzip des Information Hidings - ein altes Prinzip beim Entwurf von Systemen - angewandt.

Das Prinzip des **Information Hiding** bedeutet, daß ein Teilsystem (hier: ein Objekt) nichts von den Implementierungsentscheidungen eines anderen Teilsystems wissen darf. Es darf mit dem anderen Teilsystem nur über wohldefinierte Schnittstellen Informationen austauschen und keine Kenntnisse über den inneren Aufbau seines Partners haben. Damit haben Änderungen im Inneren eines Teilsystems keine Auswirkungen auf andere Teilsysteme, solange die Schnittstellen stabil bleiben.

1.4.2.3 Hierarchie

Abstraktion und Information Hiding sind effiziente Mittel, um mit der Komplexität fertig zu werden. Ein weiteres Mittel ist die Bildung von Hierarchien von Abstraktionen. Bei der Objektorientierung gibt es zwei wichtige Hierarchien:

?? die Vererbungshierarchie („kind of“-Hierarchie, „is a“)
?? und die Zerlegungshierarchie („part of“-Hierarchie).

Während der klassische, strukturierte Ansatz sowohl eine Daten-, als auch eine Funktionshierarchie kennt,

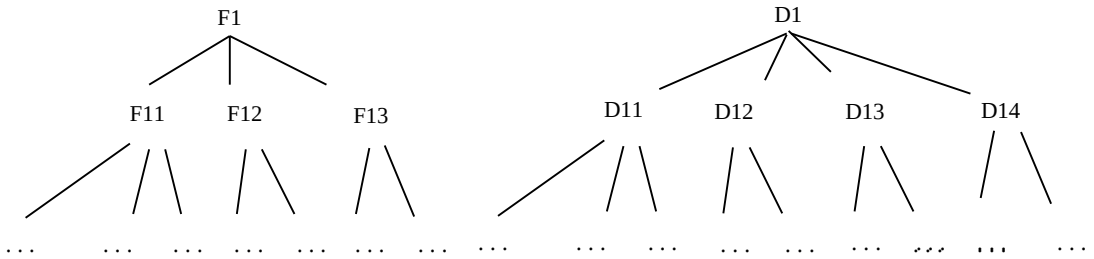


Bild 1-5 Aufrufhierarchie der Funktionen und Zerlegungshierarchie der Daten beim Strukturierten Entwurf

gibt es beim Objektorientierten Ansatz in seiner reinen Form nur noch eine Hierarchie für Objekte (allerdings eine Vererbungshierarchie und eine Zerlegungshierarchie).

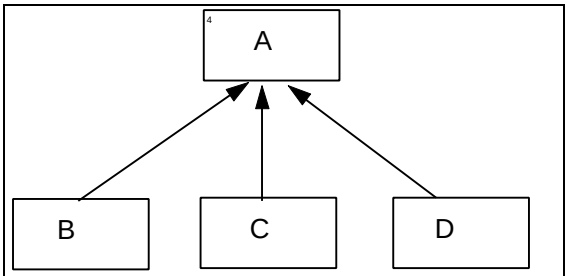


Bild 1-6 Vererbungshierarchie. Die Klassen B, C und D sind von der Vaterklasse A abgeleitet

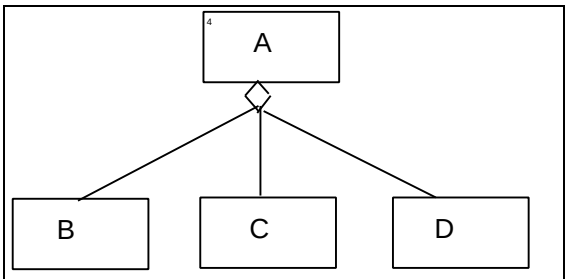


Bild 1-7 Aggregationsshierarchie. Die Klasse A enthält ein Objekt der Klassen B, C und D

Bei der **Vererbungshierarchie** werden die Klassen in Abstraktionsebenen angeordnet. Geht man von unten nach oben, so spricht man von **Generalisierung**, geht man von oben nach unten, so kommt man zu spezielleren Klassen. Man spricht von **Spezialisierung**.

Bei der Zerlegungshierachie hat man auch verschiedene Abstraktionsebenen. Ein Objekt kann aus anderen Objekten als Teilen zusammengesetzt sein (**Aggregation**). Sieht man nur das Ganze, so ist man eine Ebene höher, als wenn man die Teile betrachtet.

1.4.2.4 Modularisierung

Das Prinzip der Modularisierung besteht in der Gliederung eines Programmes in Module. Module sind getrennt compilierbare Einheiten. Wird ein Modul geändert, so muß nur dieser neu kompiliert und anschließend alles gelinkt werden -

vorausgesetzt, es wurde keine Schnittstelle einer Funktion verändert. Wurden Schnittstellen geändert, so müssen alle abhängigen Funktionen neu kompiliert werden.

Im Rahmen von C++ stellen Dateien die Module dar. Sie können getrennt kompiliert werden. In C++ legt man in der Regel für jede Klasse zwei Dateien an:

- den Klassen-Header, der die Definitionen der Klassen enthält (Endung meist .hpp)
- und den Klassenrumpf mit der Implementierung der Methoden (Endung meist .cpp)

Booch führt im Gegensatz zu Rumbaugh in seiner Methode Module ein. Es gibt bei ihm Moduldiagramme und auch Prozeßdiagramme, um parallele Abläufe im System zu beschreiben.

1.5 Industrielles Objektorientiertes Entwerfen und Programmieren

Zu allen Zeiten hat man versucht, bei Neuentwicklungen nicht alles neu zu machen, sondern auf vorhandene Teile in Form von Komponenten oder Standard-Bibliotheks-funktionen zuzugreifen.

Software kann man:

- als Programm komplett selbst entwerfen und schreiben
- in lauffähiger Form übernehmen und ggfs. erweitern (siehe Frameworks)
- oder selbst ausprogrammieren, aber wenigstens das Design von anderer Seite übernehmen. Man beschafft sich also hierbei "Blaupausen" für die Architektur in Form von Entwurfsmustern (Design Patterns) , die noch selbst ausprogrammiert werden müssen.

Die Idee der Wiederwendung findet eine Verstärkung durch den Vererbungs-mechanismus, durch den es möglich ist, für spezielle Branchen-Applikationen ganze Klassenbäume vorzugeben, aus denen dann die für das spezielle Projekt benötigten Klassen durch Spezialisierung abgeleitet werden können. Solche fertig programmierten Klassenbäume werden **Frameworks** genannt.

Für den eigenen Entwurf gibt es ein weiteres Hilfsmittel - die **Design Patterns**. Design Patterns sind nicht ausprogrammiert, sondern stellen **Entwurfsmuster** - also Architekturkonzepte - für verschiedene Problemstellungen dar. Für sein spezielles Programm kann man verschiedene, in der Literatur dokumentierte Entwurfsmuster und ihre Eigenschaften studieren, entscheidet sich dann nach Bewertung der Vor- und Nachteile für ein spezielles Entwurfsmuster und programmiert es dann in der

jeweiligen Programmiersprache selbst aus, es sei denn eine vorhandene Klassenbibliothek unterstützt bereits dieses Entwurfsmuster.

1.6 Vorteile und Nachteile des objektorientierten Ansatzes

Nach Rumbaugh ist „die Essenz der Objektorientierten Entwicklung die Identifikation und Organisation von Konzepten der Anwendungsdomäne, nicht ihre endgültige Repräsentation in einer Programmiersprache, sei sie nun objektorientiert oder nicht“. So führt er in seinem Buch auch verschiedene Beispiele an, die objektorientiert entworfen, aber nicht objektorientiert programmiert, sondern in nicht-objektorientierten Sprachen und relationalen Datenbanken implementiert wurden.

Vorteile des objektorientierten Ansatzes sind

- die Verständlichkeit des Modells für den Kunden, da die Modellierung in den Begriffen der Anwendung erfolgt
- die Stabilität der Objekte bei Änderungen
- die Durchgängigkeit der Notation von der Logik bis zum Systementwurf
- ggfs. Wiederverwendbarkeit

Diese Vorteile sollen hier kurz plausibel gemacht werden.

Verständlichkeit:
Zumindest zu Beginn des Projektes werden Abstraktionen der realen Welt betrachtet und nicht computerspezifische Konstrukte. Dies bedeutet, daß man mit der Sprache des Anwenders spricht.

Stabilität
Objekte werden identifiziert durch Betrachtung der Daten. Die Objekte beinhalten bekanntermaßen außer den Daten auch die Funktionen, die die Daten bearbeiten. Ändern sich die Bearbeitungsalgorithmen in einem klassisch strukturierten System, so kann dies **erhebliche Änderungen des Programmgefüges** nach sich ziehen.

Ändern sich die Bearbeitungsfunktionen bei Objekten, so sind diese Änderungen oftmals weniger aufwendig, da außerhalb der Objekte nur die Schnittstellen der Methoden zu sehen sind und der Rumpf der Methoden im Objekt gekapselt ist. Eine Funktionshierarchie gibt es nicht. Die Methoden eines Objektes stehen gleichberechtigt nebeneinander.

Zur Stabilität der Objekte führt auch die Unterscheidung nach Entity Objekten, Kontroll-Objekten und Interface Objekten, die von Jacobson eingeführt wurden. Hierauf kann an dieser Stelle nicht nähers eingegangen werden.

Durchgängigkeit der Notation
Ein Bruch in der Notation findet beim Objektorientierten Ansatz nicht statt. Es bleibt dem Entwickler also erspart, beim Übergang in eine andere Entwicklungsphase die Notation übersetzen zu müssen. Analyse und Entwurf sind nicht klar voneinander abgegrenzt. Welcher Detaillierungsgrad zur Analyse und welcher Detaillierungsgrad zum Entwurf gehört, ist Sache organisatorischer Festlegungen.

Ein Methodenbruch wie beim Übergang von der Strukturierten Analyse zum Structured Design tritt beim Objektorientierten Ansatz scheinbar nicht auf. Hierauf kann aber an dieser Stelle nicht im Detail eingegangen werden.

Wiederverwendbarkeit
Eine Wiederverwendbarkeit ist nur dann gegeben, wenn man beim Design der Klassen die vorgegebene Anwendung tatsächlich generalisiert, d.h. über den Tellerrand des Projektes hinweg schaut. Dies bedeutet, daß man mehr Zeit aufwenden muß, als für das bearbeitete Projekt nötig wäre. Die Wiederverwendbarkeit kommt also nicht automatisch, sondern muß durch Zusatzaufwand erkauft werden. Es bedarf einiger Sorgfalt beim Entwurf der Hierarchien und der Schnittstellen der Objekte.

Der Aufwand für die Wiederverwertbarkeit muß natürlich geringer sein als der vermutete Aufwand für das Neuschreiben. Dies muß im Einzelfall beurteilt werden. Liegt der Aufwand jeweils in der gleichen Größenordnung, so wird man sich allerdings oft für ein Neuschreiben entscheiden, da Programmierer meist lieber „schreiben“ als „lesen“.

1.7 Einführung in C++

C++ gehört nicht zu denjenigen Programmiersprachen, die wie im Falle Smalltalk oder EIFFEL komplett neu und systematisch zur Umsetzung der objektorientierten Konzepte entworfen wurden. C++ zählt zu den sogenannten **Hybridsprachen**, die eine Erweiterung ursprünglich prozeduraler Sprachen um objektorientierte Sprachelemente darstellen.

Von allen objektorientierten Sprachen dürfte C++ derzeit die größte Verbreitung haben [1].

1.7.1 Entstehung von C++

C++ entstand ab 1983 in einer Forschungsgruppe bei AT&T unter der Leitung von Bjarne Stroustrup. Eine erste Version dieser Sprache wurde 1980 unter „C with classes“ bekannt. 1983 wurde „C with classes“ erweitert und in C++ umbenannt. Die Standardisierung von C++ in einem ANSI-Komitee ist noch nicht abgeschlossen, es existiert jedoch eine brauchbare Draft-Version.

Entwurfsziel bei C++ war, das Schreiben von C-Programmen zu erleichtern, also eine bessere Unterstützung des Programmierers, insbesondere durch eine **stärkere Typenprüfung** und durch **Realisierung von objektorientierten Methodiken**. Aber gleichzeitig sollte sowohl die Effizienz von C nicht wesentlich verschlechtert werden, als auch eine höchstmögliche Kompatibilität zu C erhalten bleiben.

An diesen Zielen muß man gerechterweise die entstandene Sprache C++ messen. Der notwendige Kompromiß zwischen Effizienz, Kompatibilität zu C und objektorientierten Entwurfszielen hat allerdings zu Einschränkungen bei der maximal möglichen Flexibilität, sowie bei der Realisierung des objektorientierten Programmierstills geführt.

1.7.2 Paradigmenwechsel von C zu C++

Jeder Programmiersprache liegt ein bestimmtes Konzept zugrunde, ein bestimmtes Begriffssystem, mit dem die Lösung konzipiert wird. Dieses Konzept nennt man auch ihr **Paradigma**. Ein **Paradigma** ist sozusagen ein **konzeptionelles Muster**. Es umfaßt eine Konstellation von Begriffen und Programmiertechniken.

C ist eine prozedurale Sprache. Die Strategie ist, ein Problem durch einen Satz von Funktionen zu behandeln, die durch ihre Algorithmen die gewünschten Umformungen der Daten bewerkstelligen.

C++ ist mit den Anteilen, die über C hinausgehen, **eine objektorientierte Sprache**. Auch in C++ programmiert man mit Funktionen. Allerdings ist die Strategie eine andere. Es werden Objekte definiert, die ihre Daten und die Funktionen, die auf diesen Daten arbeiten dürfen, in sich enthalten. Mit anderen Worten:

In C++ können neue benutzerdefinierte Datentypen, die sogenannten Klassen, benutzt werden. Dabei sind Daten und Funktionen Komponenten dieser neuen Datentypen.

Ein Objekt ist eine Variable eines solchen neuen benutzerdefinierten Datentyps. Daten und Funktionen stecken damit in einer Kapsel, dem Objekt. Durch Aktivierung der Mitgliedsfunktionen können die Daten verändert werden, ansonsten nicht.

Die Vorgehensweise in C ist anders, man verarbeitet durch festzulegende Funktionen die Daten, die entweder global sind oder durch Funktionsaufrufe übergeben werden. Das bedeutet letztendlich, daß für den Programmablauf eine Aufrufhierarchie von Funktionen erfunden werden muß, die die erforderlichen Datentransformationen durchführt.

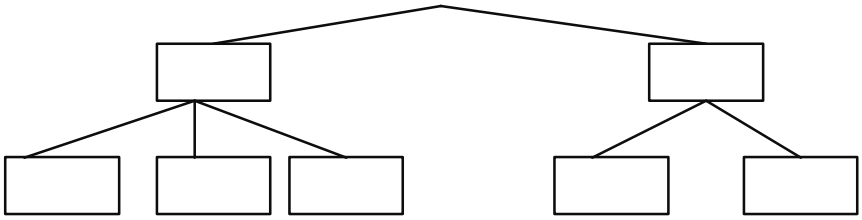


Bild 1 - 8 Die Aufrufhierarchie der Funktionen stellt eine Baumstruktur dar

Dabei sind die Daten ungeschützt. Entweder fließen sie von Routine zu Routine (Übergabeparameter) oder sie sind globale Daten.

Bei der Objektorientierung muß von vornherein festgelegt werden, welche Operationen auf den im Objekt gekapselten und nach außen geschützten Daten möglich sind. Durch Aktivierung der dem Objekt zugehörigen Funktionen bittet man sozusagen das Objekt, seine Daten selbst zu verändern. Damit sind die Daten geschützt. Fehler in den Datenzugriffen sind leichter zu finden, als wenn 1001 Routinen auf globalen Daten herumpflügen. Im Falle des Objektes können nur die im Objekt festgelegten Funktionen die Urheber falscher Berechnungen sein.

Mit dem Übergang von C zu C++ erfolgt also ein Paradigmenwechsel, welcher auch ein Umdenken in der Programmiertechnik zur Folge hat.

Das Vererbungsprinzip der Objektorientierung erlaubt über die Baumstruktur hinaus auch komplexere Beziehungen zwischen Systemteilen als im Rahmen der prozeduralen Programmierung.

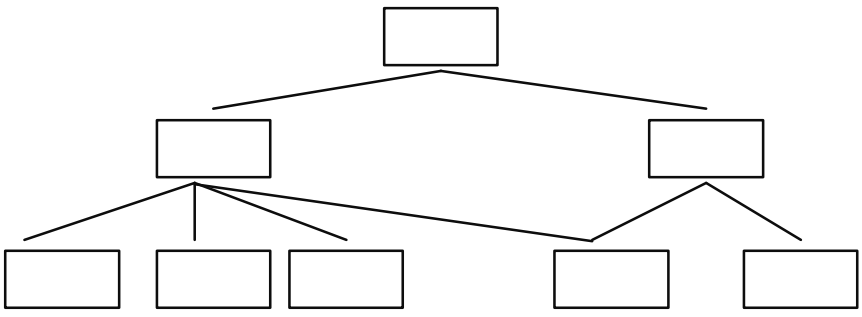


Bild 1 - 9 Komplexer Graph für die Vererbung bei Klassen

Wie aus Bild 1 - 9 ersichtlich ist, kann im Rahmen der Objektorientierung eine Komponente Beziehungen zu anderen Komponenten haben, die allgemeiner sind, als die Topologie eines Baumes.

Ein objektorientierter Entwurf enthält u.a. eine Zusammenstellung von Klassenbeschreibungen. Alle Funktionsabläufe, die beim prozeduralen Paradigma als Teil einer Aufrufhierarchie definiert wurden, stehen beim objektorientierten Entwurf bei den Daten, die sie bearbeiten.

2 Änderungen und Ergänzungen im prozeduralen Teil von C++

Bevor wir uns dem großen Bereich der OOP mit Klassen zuwenden, möchten wir zuerst einige grundlegenden Erweiterungen von C++ besprechen, die auch ohne Einsatz von OOP zur Geltung kommen.

2.1 Zeilenkommentare

Zusätzlich zu den **Blockkommentaren** aus C, die mit `/*` eingeleitet und mit `*/` beendet werden und sich über mehrere Zeilen erstrecken können, besteht in C++ die Möglichkeit, **Zeilenkommentare** zu verwenden. Zeilenkommentare werden durch `//` eingeleitet und kennzeichnen den Rest einer Zeile als Kommentar.

Blockkommentare dürfen **nicht verschachtelt** werden. Da sie Trenner sind, dürfen sie nicht innerhalb von Zeichenkonstanten oder konstanten Zeichenketten auftreten. Durch das Schachtelungsverbot wird das Herauskommentieren größerer Blöcke beim Debuggen oft verhindert. Konsequenter Einsatz von Zeilenkommentaren schafft hier Abhilfe.

Ein Zeilenkommentar bietet sich auch an, um einzelne Zeilen beim Debuggen herauszukommentieren oder Befehle direkt in der gleichen Zeile zu kommentieren. Da mit der nächsten Zeile der Kommentar automatisch beendet wird, besteht bei dieser Form nicht die Gefahr, durch einen vergessenen Kommentarabschluß ganze Blöcke versehentlich auszublenden.

Das folgende Beispiel zeigt zwei Blockkommentare. Jeder Blockkommentar geht vom öffnenden `/*` bis zum schließenden `*/`:

```
/* dies ist ein Kommentar                    */
/* dieser Unfug /* ist auch ein Kommentar    */
```

Die Kommentarzeichen `/*`, `*/` und auch `//` hinter einem Zeilenkommentar werden wie gewöhnliche Zeichen behandelt und haben an dieser Position keinerlei Bedeutung. Dasselbe gilt für die Zeichen `/*` und `//` innerhalb eines Blockkommentars.

2.2 Namen und Gültigkeitsbereiche

Generell gelten in C++ die gleichen Regeln für Namen und deren Gültigkeitsbereich wie in C. Entscheidende Änderungen kommen im Zusammenhang mit dem Klassenkonzept hinzu. Diese werden an entsprechender Stelle besprochen.

Drei kleinere Unterschiede sind unabhängig vom Klassenkonzept zu sehen und werden im folgenden erläutert:

- C++ betrachet Deklarationen als Anweisungen, sie können daher an beliebiger Stelle im Programm stehen.
- Es ist möglich, eine Schleifenvariable im Schleifenkopf zu definieren.
- Mit dem Scope-Operator bietet C++ die Möglichkeit, auf globale Variablen, die an einer Programmstelle durch die Verwendung desselben Namens verdeckt sind, zuzugreifen.

2.2.1 Definitionen und Deklarationen als Anweisungen

Definitionen von Variablen und Deklarationen waren in C nur außerhalb von Funktionen und zu **Beginn** von Blöcken erlaubt. In C++ dürfen Variable an **beliebiger** Stelle des Quelltextes definiert werden. Während in C ein Block wie folgt aufgebaut ist:

```
{
  Definitionen
  Anweisungen
}
```

werden in C++ Definitionen auch als Anweisungen gesehen.

Dabei gilt wie schon in C, daß Variablen, die außerhalb von Funktionen definiert werden, global und Variablen, die innerhalb der Funktionen definiert werden, lokal sind. Andererseits sind Variablen, die außerhalb eines Blockes definiert wurden und innerhalb des Blockes nicht erneut definiert wurden, für diesen Block **global sichtbar**. Sind innerhalb und außerhalb eines Blockes Variablen mit identischen Namen vereinbart, ist im Block nur die innere Variable **sichtbar**.

Bevor weiter darauf eingegangen wird, daß in C++ eine Definition als Anweisung gesehen wird, soll nochmals der Unterschied zwischen Deklaration und Definition, der für C und C++ gleichermaßen gilt, erläutert werden. Eine **Deklaration** macht dem Compiler einen Namen und seinen Typ bekannt. Eine **Definition** legt darüber hinaus einen Speicherplatz an. Wurde bis an eine bestimmte Programmstelle dem Compiler der Namen nicht bekannt gemacht, so führt die Definition gleichzeitig den Namen ein.

In C++ gilt eine Deklaration oder Definition auch als Anweisung. Hierfür wurde von Stroustrup der Begriff Deklarations-Anweisung geprägt.

Daß Deklarationen und insbesondere Definitionen jetzt in C++ auch als Anweisungen gesehen werden, ist auf Performance-Gründe bei großen Objekten zurückzuführen. Der eigentliche Grund kann jedoch erst verstanden werden, wenn Konstruktoren bekannt sind. Die Erklärung folgt im Kapitel „Variablendefinition an beliebiger Stelle des Quelltextes“, Kap. 8.2.

Aus Gründen der Übersichtlichkeit sollte man es vermeiden, wild herumzudefinieren. Die Statements sind schließlich nicht nur für den Compiler, sondern auch für den menschlichen Leser gedacht. Verwenden Sie Definitionen zwischen normalen Anweisungen nur:

- wenn Performance-Gründe bei großen Objekten es erfordern
- und bei der Schaffung dynamischer Variablen mit `new` (siehe Kap. „Neue Operatoren `new` und `delete`“

  So kann die Zeigervariable, die den Rückgabewert des Operators `new` aufnimmt, in derselben Anweisung definiert werden. Dies wird in folgendem Beispiel gezeigt:

```
int * ptr_int_objekt = new int;     //der Rückgabewert von new int
                                    //wird der Zeigervariablen
                                    //ptr_int_objekt zugewiesen, die
                                    //in derselben Anweisung definiert
                                    //wird
```

2.2.2 Definitionen im Schleifenkopf

Während in C im Kopf einer Schleife 3 Ausdrücke jeweils durch ; getrennt stehen wie in folgendem Beispiel:

```
/* Datei deklar2.c */

#include <stdio.h>

int main (void)
{
   int lv;
   for (lv = 0; lv <5; lv++)
     printf ("\n%d-ter Aufruf", lv);
   return 0;
}
```

ist es in C++ möglich, die Laufvariable direkt im Schleifenkopf zu definieren.

```
/* Datei deklar2.cpp */
```

```
#include <stdio.h>

int main (void)
{
   for (int lv = 0; lv <5; lv++)
     printf ("\n%d-ter Aufruf", lv);
   return 0;
}
```

Dabei ersteckt sich die Sichtbarkeit der Schleifenvariablen nur auf die Schleife selbst. In den Anweisungen nach der Schleife ist `lv` nicht mehr sichtbar. Eine Schleife stellt also einen eigenen Block dar.

2.2.3 Der Scope-Operator

Der Gültigkeitsbereich (Sichtbarkeitsbereich) einer Variablen ist jener Bereich des Programmtextes, für den auf diese Variable zugegriffen werden kann.

C kennt zwei Arten von Gültigkeitsbereichen:

- lokaler Gültigkeitsbereich
- globaler Gültigkeitsbereich

In C++ kommt noch ein dritter Gültigkeitsbereich dazu:

- Gültigkeitsbereich in einer Klasse

Scope ist das englische Wort für Gültigkeitsbereich und entsprechend dient der Scope-Operator :: dazu, den Gültigkeitsbereich eines Names bei einem Variablenzugriff oder Funktionsaufruf näher zu spezifizieren. Mit seiner Hilfe ist es zum Beispiel möglich, innerhalb einer Funktion auf eine durch lokale Variable verdeckte globale Variable zuzugreifen. Das folgende Beispiel zeigt die Verwendung des Scope-Operators für lokale und globale Daten:

```
#include <stdio.h>

int x=5; // global für alle Funktionen

void sub(void)
{
 int x=6; // lokal zu sub

 printf("\nLokale Variable (sub)   x=%d", x);
 printf("\nGlobale Variable        x=%d", ::x);
}
```

```
void main(void)
{
 int x=4;  // nicht global, sondern lokal zu main!
 sub();
 printf("\nLokale Variable (main)  x=%d", x);
 printf("\nGlobale Variable        x=%d", ::x);

}
```

Die Ausgabe des Programmes ist:

```
Lokale Variable (sub)   x=6
Globale Variable        x=5
Lokale Variable (main)  x=4
Globale Variable        x=5
```

Im Zusammenhang mit Klassen wird der Scope-Operator :: dazu verwendet, eine Funktionsdefinition eindeutig einer Klassendeklaration zuzuordnen, die die entsprechende Funktionsdeklaration beinhaltet.

Das nächste Beispiel zeigt die Verwendung des Scope-Operators für Klassen.

```
void Klasse_xyz::f1 ()   // f1 aus Klasse Klasse_xyz
{
  ...
}
```

Damit ist klar, daß die Funktion `f1` der Klasse `Klasse_xyz` hier definiert werden soll. Auf diese Verwendung des Scope-Operators werden Sie später oft stoßen.

2.3 Typkonzept

C++ hat im Gegensatz zu C ein **strenges Typkonzept**: es verhindert ungewollte Nebeneffekte bei der impliziten Typkonvertierung, wie es in C möglich ist. In C++ muß bis auf definierte Ausnahmen für die neuen C++ Typen die Typkonvertierung vom Programmierer explizit angegeben werden. Dies trifft auch auf einige Typen zu, die auch schon aus C bekannt sind, wie auf den Typ `void` und auf die Aufzählungstypen (`enum`). Explizite Typkonvertierungen sind daher in C++ Programmen häufiger zu finden. Für sie bietet C++ eine neue, eingängigere Schreibweise an.

2.3.1 Typkonvertierungen

C++ gestattet neben der alten Schreibweise (cast-Notation):

```
(Typname) Ausdruck
```

eine neue Darstellung für eine explizite Typkonvertierung:

```
Typname (Ausdruck).
```

Diese neue Schreibweise erinnert an eine Umwandlungsfunktion, daher liest man auch oft für diese Schreibweise „funktionale Notation“. Im folgenden wird ein Beispiel für eine funktionale Notation gezeigt:

```
float zahl = float (10)   // Konvertierung der Integerzahl 10
                          // in die reelle Zahl 10.0
```

Es ist zu beachten, daß diese Notation nur für Typen verwendet werden kann, die einen einfachen Namen haben.

Will man etwa eine Integerzahl in eine Adresse vom Typ `(unsigned int *)` umwandeln, so liegt ein abgeleiteter Typ vor. Die Wandlung muß in der von C bekannten cast-Notation angegeben werden:

```
unsigned int * p1 = (unsigned int *)0765
```

Definiert man einen einfachen Namen mit Hilfe des `typedef`-Operators, hier z.B.

```
typedef unsigned int * ptrunint;
```

so kann die Wandlung wieder in der funktionalen Notation erfolgen:

```
ptrunint * p2 = ptrunint (0765);
```

Das folgende Beispiel verdeutlicht nochmals diese beiden Schreibweisen in einem lauffähigen Programm.

```
#include <stdio.h>

void main (void)
{
 int i;
 printf ("\n\n%p", &i);

 unsigned int * p1 = (unsigned int *) &i;    // cast-Notation
 printf ("\n%p", p1);

 typedef unsigned int * ptrunint;
 ptrunint * p2 = ptrunint (&i);              // Funkt. Notation
 printf ("\n%p", p2);
}
```

Die Ausgabe des Programmes ist:

```
FFF4
FFF4
FFF4
```

2.3.2 Der Typ void

Die Regeln der Typisierung betreffen auch den Typ `void`. Im Gegensatz zu C kann ein Zeiger auf `void` nicht implizit durch Zuweisung in einen anderen Zeigertyp gewandelt werden. Man kann jedoch auch weiterhin jeden Zeiger an einen Zeiger auf `void` zuweisen, denn damit kann ja auch kein Schaden angerichtet werden.

Beispiel:

```
void main (void)
{
  void *dummy;
  int * ptr1;
  unsigned char *ptr2;
                    // Zeigertyp-Konvertierungen:
   dummy = ptr1;    // in C++  und in C korrekt,
   ptr2  = dummy;   // in C++ falsch, in C jedoch korrekt,
                    // so sieht die Lösung in C++ aus:
   ptr2  = (unsigned char *) dummy;
}
```

In C++ können die Zeigertyp-Konvertierungen auch kürzer geschrieben werden, da `dummy` jetzt überflüssig ist:

```
ptr2 = (unsigned char *) ptr1;
```

> Zeiger vom Typ `void *` können nur explizit in einen anderen Typ konvertiert werden.

2.3.3 Aufzählungstypen

Aufgrund der strengen Typprüfung ist es nicht möglich, einem Objekt eines Aufzählungstyps (`enum`) einen Wert eines anderen Typs zuzuweisen, ohne daß eine explizite Typkonvertierung verwendet wird!

Beispiel:

```
#include <stdio.h>

enum ECOLOR { RED, GREEN, BLUE };

void main (void)
{
  enum ECOLOR x;

  x = GREEN;
  printf ("%d\n", x);
```

```
  x = RED;
  printf ("%d\n", x);
  x = 5;                    // in C++ falsch, in C jedoch korrekt
  printf ("%d\n", x);

}
```

Die Zuweisung `x = 5` ist in C korrekt, nicht jedoch in C++! Die Konstante 5 ist vom Datentyp `int`, der ein anderer Datentyp als `ECOLOR` ist, auch wenn die Konstanten des Aufzählungstyps `int`-Werte haben.

2.4 Referenzkonzept

In C++ gibt es ein Alias-Konzept, d.h. man kann für ein und dasselbe Objekt mehrere Namen einführen. Ein Alias oder eine Referenz auf ein bereits existierendes Objekt wird folgendermaßen eingeführt:

```
T & Bezeichner1 = Bezeichner2;
```

Das Beispiel ist folgendermaßen zu lesen: `Bezeichner1` ist eine Referenz auf das Objekt `Bezeichner2` vom Typ `T`, was bedeutet, daß `Bezeichner1` ein anderer Name (Alias-Name) für das bereits definierte und durch `Bezeichner2` angegebene Objekt vom Typ `T` ist.

Eine Referenz ist ein alternativer Name (Aliasname) für ein Objekt. Referenzen müssen immer bei ihrer Definition initialisiert werden.

Eine Referenz muß nicht initialisiert werden, wenn es sich

?? um eine Extern-Deklaration,
?? um die Deklaration innerhalb einer Klassendeklaration,
?? um die Deklaration eines Parameters oder
?? um den Ergebnistyp einer Funktion

handelt.

Die Initialisierung einer Referenz ist trivial, solange der Initialisierer ein Objekt vom selben Typ darstellt, dessen Adresse ermittelt werden kann. Für den Fall, daß der Initialisierer eine Konstante ist oder nicht im Typ mit der Referenz übereinstimmt, wird auf die weiterführende Literatur oder das Compilerhandbuch verwiesen.

Bei der Benutzung einer Referenz muß man aufpassen: `Bezeichner1++` inkrementiert nicht die Referenz `Bezeichner1`, sondern das Objekt `Bezeichner2` selbst. Der Wert der Referenz kann nach der Initialisierung nicht verändert werden. `Bezeichner1` bezieht sich immer auf das Objekt `Bezeichner2`. Es ist sogar

möglich, über `&Bezeichner1` die Adresse des Objektes `Bezeichner2` zu erhalten. Also:

Jede Operation auf einer Referenz erfolgt auf der referenzierten Variablen

Beispiel:

```
int i = 5;
int & j = i;   // das bedeutet nichts anderes, als daß j ein
               // Alias-Name für i ist. Mit anderen Worten:
               // dasselbe Objekt kann sowohl durch seinen
               // eigentlichen Namen i, als auch durch seinen
               // Alias-Namen angesprochen werden

j = 2;         // damit hat auch i den Wert 2
j++;           // damit hat auch i den Wert 3
```

Wenn die Referenz einmal initialisiert wurde, kann sie nicht mehr zum „Alias“ eines anderen Objektes werden. Mit anderen Worten:

Es ist nicht möglich, eine Referenz auf eine andere Variable umzulenken.

Beispiel:

```
int i = 5;
int k = 2;
int & j = i;

j = k;          // führt nicht dazu, daß nun j eine
                // Referenz auf k ist, vielmehr hat
                // nun die Variable i den Wert
                // von Variable k.
```

Einen Referenztyp kennzeichnet man durch einen **nachgestellten** Adreßoperator (&): `Typ &`, also z.B. `int &`. An dieser Stelle ein Hinweis auf einen häufigen Leichtsinnsfehler: Der Typ heißt zwar `int &` (vgl. `int *`), sollen aber in einer Zeile mehrere Referenzen (oder Pointer) vereinbart werden, dann muß vor **jedem** Variablennamen ein & (*) stehen.

Eine Referenz auf den Typ `void` ist nicht erlaubt. Referenzen auf Referenzen, Vektoren von Referenzen, Zeiger auf Referenzen und Referenzen auf Bitfelder sind ebenfalls nicht zulässig. Dagegen sind Referenzen auf Pointer möglich und ebenso Referenzen auf Funktionen.

Das & darf nur in Deklarationen als Referenzattribut verwendet werden. In Ausdrücken behält es seine Funktion als Adreßoperator! Machen Sie sich die Unterschiede zwischen Pointer, Referenz, Dereferenzierungs- und Adreßoperator klar und kommen Sie nicht durcheinander!

Mit Hilfe der Referenztypen ist es nun in C++ möglich, ein **call by reference** zu realisieren. Sie können sowohl für Parameter als auch für das Funktionsergebnis benutzt werden.

2.5 Funktionen in C++

Einige der Neuerungen von C++ bzgl. Funktionen wurden bereits in ANSI-C übernommen wie z.B. die Prototypform der Deklaration, die Typprüfung der Parameterliste oder die sogenannte Ellipse. Die Ellipse sind das Komma und die drei Punkte dahinter (`,...`) als letzter Parameter in einer Parameterliste. Durch die Ellipse oder auf deutsch „Auslassung“ wird es ermöglicht, daß die zu übergebende Parameterzahl variabel ist, was Ihnen von `scanf` und `printf` bereits bekannt sein sollte. Allerdings muß diese Variabilität vom Programmierer im Funktionsrumpf beherrscht werden! (Hinweis: Zur Unterstützung der Ellipse gibt es Makros in der Datei <`stdarg.h`> wie `va_start, va_arg`.)

Die in ANSI-C noch als Übergangslösung erlaubte sogenannte traditionelle Kernighan&Ritchie-C-Form der Funktionsdeklaration ist in C++ nicht mehr zulässig. Im alten C von Kernighan und Ritchie - also noch vor ANSI-C - wurden bei der **Definition einer Funktion** in der Parameterliste nur die Namen der formalen Parameter angegeben. Die Typen wurden nach der Parameterliste und noch vor der öffnenden geschweiften Klammer des Funktionsrumpfes aufgeführt. Parameter müssen in C++ einzeln aufgeführt werden und können nicht nach dem Typ gelistet werden.

```
// In C++ nicht möglich sind also:

int mult ()        int i, k; {...}
int mult (int i, k)          {...}
```

In C++ gibt es jedoch Erweiterungen, die in ANSI-C nicht zur Verfügung stehen. So ist es in C++ möglich:

?? Default-Werte für formale Parameter zu vergeben,
?? per Referenztyp ein echtes Call-by-Reference zu realisieren und
?? den gleichen Funktionsnamen für unterschiedliche Funktionen zu verwenden.

Diese Erweiterungen werden in den folgenden Abschnitten vorgestellt.

Mit Hilfe einer speziellen Schreibweise für Funktionsnamen ist es möglich, auch Operatoren zu definieren, d.h. die in C++ existierenden Operatorsymbole mit einer neuen Bedeutung zu belegen. Da dies jedoch nur im Zusammenhang mit benutzerdefinierten Klassen erlaubt ist, werden wir darauf erst später eingehen.

2.5.1 Default-Werte für Funktionsparameter

In C++ ist es möglich, Default-Werte für formale Parameter zu vergeben. Die Schreibweise entspricht dabei der einer Initialisierung (die ja dann bei fehlendem aktuellen Parameter auch vorgenommen wird). Man spricht auch von einer Voreinstellung für den Wert eines solchen Parameters.

Damit der Compiler die aktuellen Parameter den formalen eindeutig zuordnen kann, gelten folgende Vorschriften:

?? Ist ein formaler Parameter als Default-Wert deklariert, dann müssen alle folgenden Parameter auch einen Default-Wert besitzen. Default-Parameter stehen also nach allen normalen formalen Parametern ganz rechts in der Parameterliste.

?? Wird bei einem Funktionsaufruf ein Parameter weggelassen, dann müssen alle folgenden Parameter auch weggelassen werden.

Machen Sie sich klar, warum diese Vorschriften nötig sind. Es wird Sie vor Fehlern schützen und davor, sich über diese "blöden Einschränkungen", die eigentlich gar keine sind, aufzuregen. Hierzu ein einfaches Beispiel:

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

struct point {
    float x;
    float y;
};


point makepoint (float u = 0, float v = 0)
{
  point temp;

  temp.x = u;
  temp.y = v;
  return temp;
}

void main (void)
{
  point u;
  u = makepoint ();
  printf("\nDie Koordinaten des Punktes sind: %5.2f %5.2f", u.x, u.y);
  u = makepoint (5);
  printf("\nDie Koordinaten des Punktes sind: %5.2f %5.2f", u.x, u.y);
  u = makepoint (3,4);
  printf("\nDie Koordinaten des Punktes sind: %5.2f %5.2f", u.x, u.y);
}
```

Die Ausgabe des Programmes ist:

```
Die Koordinaten des Punktes sind:  0.00  0.00
Die Koordinaten des Punktes sind:  5.00  0.00
Die Koordinaten des Punktes sind:  3.00  4.00
```

Ein Default-Argument kann in einer späteren Deklaration nicht redefiniert werden (nicht einmal mit demselben Wert). Eine Deklaration kann allerdings Default-Argumente neu hinzufügen, die in einer früheren Deklaration noch fehlten. Dies verdeutlichen die folgenden Beispiele:

Das nächste Beispiel zeigt Default-Werte bei der ersten Deklaration:

```
point makepoint (float = 0, float = 0);


void main (void)

{
 ...
}


point makepoint (float u, float v)   // hier darf jetzt nicht mehr
                                     // initialisiert werden
{
  point temp;

  temp.x = u;
  temp.y = v;
  return temp;
}
```

Würde man in der Funktionsdefinition

```
point makepoint (float u = 0, float v = 0)
```

schreiben, so wäre dies eine Redeklaration der Funktion, die nicht erlaubt ist!

Das folgende Beispiel zeigt Default-Werte bei einer späteren Deklaration:

```
point makepoint (float, float);

point makepoint (float , float = 0);

point makepoint (float = 0, float);

point makepoint (float u, float v)
{
```

```
  point temp;

  temp.x = u;
  temp.y = v;
  return temp;
}
```

Dieses Beispiel ist korrekt, da in jeder Deklaration nur noch zusätzliche Informationen zu den Parametern hinzugefügt werden. Im Interesse der Übersichtlichkeit kann jedoch nur davon abgeraten werden. Damit der Anwender sie effektiv nutzen kann, gehören Default-Werte zur ersten Deklaration im Header-File.

2.5.2 Overloading - Mehrfachbelegung von Funktionsnamen

In der Regel gibt man verschiedenen Funktionen verschiedene Namen. Oftmals verrichten aber verschiedene Funktionen dieselbe Aufgabe, nur für verschiedene Datentypen der Übergabeparameter. Denken Sie z.B. an eine Ausgabe-Funktion, welche die Ausgabe eines Übergabe-Parameters auf den Bildschirm bewerkstelligt. Je nach Datentyp des Arguments braucht man eine andere Funktion. Jede der Funktionen muß dabei etwas anderes tun, um die Ausgabe durchzuführen. Beim **Overloading** können jedoch alle Funktionen denselben Namen tragen. Anhand des Datentyps des Übergabeparameters erkennt der Compiler, welche der Funktionen gemeint ist.

Das Überladen erfolgt ganz einfach durch die Definition verschiedener Funktionen mit gleichem Funktionsnamen, aber verschiedenen Parameterlisten. Der Aufruf der richtigen Funktion ist Aufgabe des Compilers. Man spricht in diesem Zusammenhang vom Überladen eines Funktionsnamens.

Der Nutzen ist, daß man gleichartige Funktionen mit gleichem Namen ansprechen kann. Die Verständlichkeit der Programme kann dadurch erhöht werden.

Hinweise:

?? Zu überladende Funktionen müssen im selben Gültigkeitsbereich deklariert werden. Lokal deklarierte Funktionen überdecken andere evtl. vorhandene Funktionen und überladen sie nicht !

?? Zu beachten ist, daß es nicht möglich ist, zwei Funktionen mit gleichem Funktionsnamen und gleicher Parameterliste, aber verschiedenen Ergebnistypen zu vereinbaren.

?? Die Typen `T` und `T&` werden bezüglich des Überladens von Funktionen nicht unterschieden, da eine Unterscheidung anhand der Typen der aktuellen Parameter nicht möglich ist.

?? Wird keine exakte Übereinstimmung gefunden, wird versucht, eine sinnvolle Typkonvertierung der aktuellen Parameter zu finden (siehe hierzu bei Bedarf die Compilerhandbücher) - besser ist es jedoch stets, selbst für passende aktuelle Parameter zu sorgen, ggfs. durch explizite Typkonvertierung. Dann weiß man auf jeden Fall, was los ist.

Bei gemischter Verwendung von Overloading und Default-Argumenten ist Vorsicht geboten. Die beiden obigen Funktionen unterscheiden sich in der Anzahl ihrer formalen Parameter, jedoch nicht unbedingt in der Anzahl der aktuellen Parameter. Sie können deshalb vom Compiler nur auseinandergehalten werden, wenn der Aufruf mit zwei aktuellen Parametern erfolgt.

```
int testfunk(int x) { ... }                    //Variante 1
int testfunk(int x, int y=23) { ... }          //Variante 2
```

Bei dem Aufruf `testfunk(34)` ist es nicht unbedingt klar für den Compiler, ob Sie die Variante 1 mit `x=34` oder die Variante 2 mit `x=34` und dem Default-Parameter `y=23` aufrufen wollen. Vermeiden Sie solche Mehrdeutigkeiten in Ihrem eigenen Interesse. Manche Compiler weisen Sie bereits bei der Deklaration auf die Gefahren hin.

Der Effekt von Default-Argumenten kann alternativ durch Überladen erzielt werden, wie aus den folgenden zwei Beispielen hervorgeht:

Beispiel mit Default-Argumenten:

```
#include <stdio.h>


void print (int value, int base = 10)
{
if (base == 16) printf ("\n\n\nhex:     %x", value);
 else if (base == 8) printf ("\noktal:   %o", value);
 else if (base == 10) printf("\ndezimal: %d", value);
}

void main (void)
{
 print (10, 16);
 print (10, 8);
 print (10, 10);
 print (10);
}
```

Beispiel mit Overloading:

```
#include <stdio.h>

void print (int value, int base)
{
 if (base == 16) printf ("\n\n\nhex:     %x", value);
 else if (base == 8) printf ("\noktal:   %o", value);
```

```
 else if (base == 10) printf("\ndezimal: %d", value);
}

void print (int value)
{
 printf("\ndezimal: %d", value);
}

void main (void)
{
 print (10, 16);
 print (10, 8);
 print (10, 10);
 print (10);
}
```

Default-Parameter sollten immer dann eingesetzt werden, wenn die gleiche Funktionalität (gleicher Funktionsrumpf) bei unterschiedlicher Parameteranzahl vorliegt. Overloading ist vorzuziehen, wenn bei unterschiedlichen Parametertypen die gleiche Funktionalität implementiert werden soll. Hier muß gleiche Funktionalität nicht gleicher Funktionsrumpf bedeuten.

2.5.3 Call by Reference

Die in C++ möglichen Referenztypen können für Parameter benutzt werden, die per *call by reference* übergeben werden sollen. Die so deklarierten Parameter können also wie *var*-Parameter in Pascal behandelt werden.

In C ist eine *call by reference*-Schnittstelle in der Syntax nicht vorgesehen. De facto kann man das Verhalten einer *call by reference*-Schnittstelle trotz vorliegender *call by value*-Schnittstelle erreichen, indem man einen Zeiger auf den aktuellen Parameter mit *call by value* übergibt. Das klassische Beispiel für einen solchermaßen erzeugten *call by reference* ist `scanf`. Da die lokale Kopie des Zeigers ebenfalls auf die äußere Variable zeigt, kann diese verändert werden. In diese Fußangel ist auch schon mancher Programmierer unbeabsichtigt getreten, der seine Daten in Sicherheit glaubte. Seien Sie also bei der Übergabe von Zeigern stets vorsichtig. Vektoren werden grundsätzlich als Zeiger übergeben, also immer *call by reference*!

Benutzt man die in C++ möglichen Referenztypen für ein *call by reference*, dann braucht man sich um die Problematik der Zeigerlösung nicht zu kümmern. Referenztypen können sowohl für die Parameter als auch für das Funktionsergebnis eingesetzt werden. Das wird an den folgenden Beispielen verdeutlicht.

Referenzen als Übergabeparameter

```
void incr (int & zahl)
{
  zahl++;
}

void main ()
{
  int x = 1;
  printf ("\n\n x = %d", x);
  incr (x);
  printf ("\n x = %d", x);
}
```

Die Ausgabe ist:

```
 x = 1
 x = 2
```

Das & beim Aufruf und der * im Funktionscode wie bei der alten Methode müssen nicht mehr geschrieben werden; der Compiler nimmt uns die Arbeit ab.

Referenzen als Rückgabewert

Auch als Rückgabewert können Referenzen benutzt werden (siehe folgendes Beispiel):

```
unsigned int groessergleich50 = 0;
unsigned int kleiner50        =  0;


unsigned int &  teste (unsigned int wert)
{
  if (wert >= 50)
    return groessergleich50;
  else
    return kleiner50;
}

void main(void)
{
   int i=0;
   randomize(); // Initialisiert den Zufallsgenerator

   while (i < 10000)
   {
     ( teste (random(100)) )++;   // Random(100) generiert
                                  // Zufallszahlen zwischen 0 und 99
     i++;
   }

   printf("Von %d Zahlen zwischen 0 und 99 waren  \n",i);
   printf("%d Zahlen zwischen 0 und 49 \n", kleiner50 );
```

```
   printf("%d Zahlen zwischen 50 und 99\n", groessergleich50 );
}
```

Lvalues

So wie ein dereferenzierter Zeiger auf ein Objekt einen **lvalue** darstellt, stellt auch eine Referenz auf ein Objekt einen lvalue dar. Einer Referenz kann - wie Sie wissen - ein Wert zugewiesen werden. Die Zuweisung erfolgt dabei an das Objekt, auf das die Referenz zeigt.

Damit ist auch klar, daß ein Aufruf einer Funktion, die eine Referenz zurückgibt, ein lvalue ist. Normalerweise ist ein Funktionsaufruf ein rvalue.

Beispiel:

```
#include <stdio.h>
//Datei: lref.cpp

int & test (int & par)
{
  par++;
  return par;
}

void main (void)
{
 int alpha = 3;
 int beta = 0;
 printf ("\n\nalpha hat den Wert: %d", alpha);    //alpha = 3
 beta = test (alpha);
 printf ("\nalpha hat den Wert: %d", alpha);      //alpha = 4
 printf ("\nbeta hat den Wert: %d", beta);        //beta  = 4
 test (beta) = 7;
 printf ("\nbeta hat den Wert: %d", beta);        //beta = 7
}
```

Da dieses Programm nicht selbsterklärend ist, hier eine kommentierte Version:

```
#include <stdio.h>

int & test (int & par)
{
  par++;
  return par;
}

void main (void)
{
 int alpha = 3;
 int beta = 0;
 printf ("\n\nalpha hat den Wert: %d", alpha);    //alpha = 3
```

```
 beta = test (alpha);
```

/* ***Der aktuelle Parameter*** wird dem formalen Parameter zugewiesen:
`int & par = alpha`
`par` ist also eine Referenz auf `alpha` und ist damit ein Aliasnamen für `alpha`.
Die Operation `par++` findet auf der referenzierten Variablen statt. Also wird `alpha` um eins erhöht.

Als **Rückgabewert** wird die Referenz `par zurückgegeben.` `par` selbst ist aber eine Referenz auf `alpha`. Damit wird also von `test (alpha)` eine Referenz auf `alpha` zurückgegeben.
`beta = test (alpha)` -> `beta` wird der Wert der Referenz auf `alpha` zugewiesen. Der Wert der Referenz auf `alpha` ist der Wert des referenzierten Objekts, auf das die Referenz zeigt, und ist damit der Wert von `alpha`.
*/

```
 printf ("\nalpha hat den Wert: %d", alpha);      //alpha = 4
 printf ("\nbeta hat den Wert: %d", beta);        //beta  = 4

 test (beta) = 7;
```

/*Der Rückgabewert von `test (beta)` ist eine Referenz auf `beta`.
Der Referenz auf `beta` wird der Wert 7 zugewiesen. Die Operation findet auf der referenzierten Variablen statt. Also nimmt `beta` den Wert 7 an. */

```
 printf ("\nbeta hat den Wert: %d", beta);        //beta = 7
}
```

Hier noch ein Tip, um Fehler zu vermeiden: Wenn Sie eine Funktion haben, die einen Pointer oder eine Referenz (ja, das geht auch - siehe oben!) zurückliefert, dann stellen Sie sicher, daß diese(r) nicht auf eine lokale (auto) Variable zeigt. Es könnte sonst passieren, daß sich der Inhalt irgendwann auf wundersame Weise ändert. Deklarieren Sie in solchen Fällen entweder die lokale Variable als `static` oder gehen Sie den Weg über die Übergabe eines Zeigers in der Parameterliste. Dieses Problem tritt häufig bei Funktionen auf, die einen `char *` Wert (String) als Return-Wert zurückliefern, um dann den Wert z.B. später durch `printf` auszugeben. Werden zwischenzeitlich andere Ausgaben mit `printf` durchgeführt, so schütten die Parameter für `printf` den Stack mit dem darauf befindlichen Funktionsergebnis zu wie in folgendem Beispiel:

```
#include <stdio.h>

int * f1 (int u)
{
 int x;
  x = u + 10;
  return &x;
}

void f2( int * v )
{
 *v += 10;
}

int * f3 (int u)
{
  static int x;
  x = u + 10;
  return &x;
}

void main( void )
{
 int i = 1;
 int * ptr = &i;
 printf("\n\n*ptr vor f1     : *ptr = %d \n", i);
 // Ausgabe: *ptr = 1

 ptr = f1 ( i );
 printf("*ptr nach f1    : *ptr = %d \n", *ptr);
 // lokale Variablen werden auf dem Stack angelegt
 // Ausgabe: *ptr = 11
 // ptr zeigt auf die lokale Variable x von f1
 // auf dem Stack unmittelbar nach dem Funktionsaufruf
 // ist dies meistens noch korrekt

 printf("*ptr nach printf: *ptr = %d \n", *ptr);
 // Ausgabe: *ptr = -10  (ist hier willkuerlich)
 // Nachdem aber die Funktion printf den Stack mit seinen
 // Parametern und den lokalen Variablen ueberschrieben
 // hat, zeigt ptr auf undefinierte Werte auf dem Stack
```

```
 ptr = &i;
 printf("\n\n*ptr vor f2     : *ptr= %d\n", *ptr);
 // Ausgabe: *ptr = 1

 f2 (ptr);
 printf("*ptr nach f2    : *ptr = %d\n", *ptr);
 // Ausgabe: *ptr = 11

 printf("*ptr nach printf: *ptr = %d \n", *ptr);
 // Ausgabe: *ptr = 11
 // da hier die Variable direkt veraendert wird, kann
 // printf die Variable nicht ueberschreiben


 printf("\n\n*ptr vor f3     : *ptr= %d\n", *ptr);
 // Ausgabe: *ptr = 11
 ptr = f3 ( i );
 printf("*ptr nach f3    : *ptr = %d \n", *ptr);
 // Ausgabe: ptr = 21
 // ptr zeigt auf die lokale statische Variable x von f1

 printf("*ptr nach printf: *ptr = %d \n", *ptr);
 // Ausgabe: *ptr = 21
 // Die lokale statische Variable lebt bis zum Programmende,
 // sie wird nicht auf dem Stack angelegt
}
```

Hier ein Beispiel für **Referenzen auf Funktionen**:

```
#include <stdio.h>

// Definition des Typs ref_auf_fkt_void als Funktion
// void .... (int);
typedef void  ref_auf_fkt_void     (int);
// Definition des Typs ref_auf_fkt_intptr als Funktion
// int * .... (int &);
typedef int * ref_auf_fkt_intptr (int &);

void f1(int i)
{
        printf("Funktion f1: i = %d \n",i);
}

int * f2(int & i)
{
        printf("Funktion f2: i = %d \n",i);
        return &i;
}
```

```
void main (void)
{
        int i = 1;

        ref_auf_fkt_void   & reff1 = f1;
        ref_auf_fkt_intptr & reff2 = f2;

        f1(i);       // Ausgabe: Funktion f1: i = 1
        i++;
        reff1(i);    // Ausgabe: Funktion f1: i = 2
        i++;
        f2(i);       // Ausgabe: Funktion f2: i = 3
        i++;
        reff2(i);    // Ausgabe: Funktion f2: i = 4
}
```

2.6 Ablösung von Makros in C++

Die beiden neuen Schlüsselwörter `const` und `inline` ersetzen in vielen Fällen die Präprozessoranweisung **`#define`**, bei deren Gebrauch es leicht zu heimtückischen Fehlern kommen kann. Um diese Fehler besser zu verstehen, hier zunächst eine kleine Wiederholung der Arbeitsweise des Präprozessors und im Anschluß daran eine genaue Erläuterung des neuen Schlüsselwortes `inline`. Da das Schlüsselwort `const` im Rahmen der ANSI-Standardisierung in C übernommen wurde, wird an dieser Stelle darauf nicht eingegangen. In C++ gibt es hierzu jedoch auch Anwendungen mit Klassen.

Konstante Datenelemente im Objekt werden im Kapitel "Statische Variablen und Statische Funktionen" behandelt. Konstante Memberfunktionen und konstante Objekte im Kapitel "Konstanten im Objekt" des Fortgeschrittenenkurses.

2.6.1 Präprozessor

Der Präprozessor erzeugt aus dem vom Programmierer mit Präprozessor-anweisungen versehenen Quellcode einen neuen, der dann an den Parser (Syntaxprüfer) weitergegeben wird. Diese Neufassung des Quellcodes ist meist nur temporär während der Compilierung vorhanden und ist in aller Regel länger als der ursprüngliche Quellcode. Bei der Fehlersuche ist es oft zweckmäßig, das Löschen dieser Langversion durch einen Compilerschalter zu verhindern, da manche Fehler nur dort zu finden sind.

Im folgenden sollen Makros den in C++ neu eingeführten **Inline-Funktionen** gegenüber gestellt werden.

Der Vorteil von Makros im Vergleich zu den später zu besprechenden Inline-Funktionen liegt darin, daß für die Parameter kein Typ angegeben werden muß. Somit können z.B. Rechenmakros mit einem beliebigen numerischen Typ aufgerufen

werden. Dieser scheinbare Vorteil stellt andererseits auch wieder einen Nachteil dar, da eine Typüberprüfung nicht erfolgt. Es sei an dieser Stelle schon darauf hingewiesen, daß nur mit **Templates** (siehe Fortgeschrittenenkurs) die Flexibilität bezüglich verschiedener Datentypen wie bei Makros und die Sicherheit der Typprüfung von Funktionen erreicht werden kann.

Andererseits gibt es bei der Verwendung von Makros Situationen, die fehlerträchtig sind und bei denen die Verwendung von Inline-Funktionen von Vorteil ist.

Nun zu den Problemen, die bei gewöhnlichen Makros auftreten können. Sie hängen samt und sonders damit zusammen, daß der Präprozessor nur Textersetzung vornimmt. Betrachten wir einmal folgendes Beispiel:

```
//
// Probleme mit Makros: mac1.cpp
//

# include <stdio.h>
# define square(x) x*x
# define double(x) x+x

int quadrat (int n)
{
  return n * n;
}

void main(void)
{
 int z=3;
 printf ("\n\n\nz = %d", z);
 printf ("\nsquare(4) = %d", square(4));     // #1 4*4 =  16
 printf ("\nsquare(4+1) = %d", square(4+1)); // #2 4+1*4+1 = 9 statt 25
 printf ("\nsquare(z) = %d", square(z));     // #3 z*z = 9
 printf ("\nsquare(z++) = %d", square(z++)); // #4 z++*z++ = 12 statt 9
 printf ("\nz = %d",z);                      // #4   z=5 statt z=4
 printf ("\ndouble(4) = %d", double(4));     // #5   4+4 = 8

 printf ("\ndouble(4)*double(5) = %d",double(4)*double(5));
                                            // #6   4+4*5+5 = 29 statt 80
 printf ("\nz * quadrat (z++) = %d", z * quadrat (z++));
                                            // #7   z = 5
                                            //  6 * 25 = 150
// Der ANSI-Standard legt fest, daß alle Nebenwirkungen der Argumente
// stattfinden, bevor eine Funktion aufgerufen wird, dah er auch die 6
// vor quadrat
}
```

Wie Sie leicht sehen, führt die reine Textersetzung zu gewissen Problemen. Wäre `square()` eine Inline-Funktion, dann wären die Ausdrücke zuerst ausgewertet und dann als Parameter übergeben worden. Aber auch bei Verwendung von Makros lassen sich einige der obigen Probleme umgehen. Die Fehler in den Zeilen #2 und

#6 lassen sich vermeiden, indem man sowohl alle Parameter im Makrocode als auch die Makrodefinition selbst klammert. Dies erfolgt im nächsten Beispiel:

```
//
// Probleme mit Makros: mac2.cpp
//

# include <stdio.h>
# define square(x) (x)*(x)
# define double(x) ((x)+(x))

int quadrat (int n)
{
  return n * n;
}

void main(void)
{
  int z=3;
  printf ("\n\n\nz = %d", z);
  printf ("\nsquare(4) = %d", square(4));      // #1   (4)*(4) = 16
  printf ("\nsquare(4+1) = %d", square(4+1));  // #2   (4+1)*(4+1) = 25
  printf ("\nsquare(z) = %d", square(z));      // #3   (z)*(z) = 9
  printf ("\nsquare(z++) = %d", square(z++));  // #4   (z++)*(z++) = 12
  printf ("\nz = %d",z);                       // #4   z=5 statt z=4
  printf ("\ndouble(4) = %d", double(4));      // #5   ((4)+(4)) = 8

  printf ("\ndouble(4)*double(5) = %d",double(4)*double(5));
                                              // #6   ((4)+(4))*((5)+(5)) = 80
  printf ("\nz * quadrat (z++) = %d", z * quadrat (z++));
                                              // #7   z = 5
                                              //  6 * 25 = 150
// Der ANSI-Standard legt fest, daß alle Nebenwirkungen der Argumente
// stattfinden, bevor eine Funktion aufgerufen wird, daher auch die 6
// vor quadrat

}
```

Mit dem Fehler in #4 muß man jedoch (bei Makros) weiterhin leben. Ein anderes Problem kann dann auftreten, wenn ein Makro aus mehreren Befehlen in einer Schleife aufgerufen wird:

```
#define double_increment (a,b) a++; b++;

void main(void)
{
...
for(int x=0; x<10; x++) double_increment (v1, v2);
}
```

Wäre `double_increment` eine Funktion, dann gäbe es überhaupt keine Probleme. So aber wird bei jedem Schleifendurchlauf nur `v1++` ausgeführt. Der Ausdruck

v2++ steht außerhalb der Schleife (Schreiben Sie ein kleines Beispielprogramm und lassen es durch Ihren Präprozessor laufen)! Auch hier ist Abhilfe nur durch Klammerung möglich, diesmal allerdings durch Klammerung des Makrocodes mit geschweiften Klammern. Doch nun zu den beiden neuen Schlüsselwörtern, die durch neue C++-Anweisungen Verbesserungen gegenüber Präprozessor-Anweisungen erbringen.

2.6.2 Neues Schlüsselwort inline

`inline` wird nur im Zusammenhang mit Funktionen verwendet.

Stellt man einem Funktionskopf das Schlüsselwort `inline` voraus, dann wird die Funktion zu einer `inline`-Funktion. Für eine inline-Funktion generiert der Compiler (wenn er der Empfehlung `inline` folgt) keine Aufrufe mehr, sondern fügt jedesmal den gesamten Funktionscode ein.

Man sagt: der Compiler expandiert die Funktion zur Compilierzeit. Die Ersetzung des Programmtextes erfolgt also wie bei Makros zum Übersetzungszeitpunkt.

Das Schlüsselwort **`inline`** stellt für den Compiler nur eine Empfehlung dar, wie auch **`register`** bei den Speicherklassen. So wird ein Compiler es ignorieren, etwa rekursive als `inline` definierte Funktionen zu expandieren.

Unterschied zu normalen Funktionen

Der Vorteil von `inline` deklarierten Funktionen gegenüber normalen Funktionen liegt darin, daß der gesamte Verwaltungsoverhead für die Parameterübergabe auf dem Stack und den Unterprogrammaufruf wegfällt (z.B. Kopieren von Argumenten, Sichern von Maschinenregistern, Programmsprung, Stackverwaltung) und trotzdem die kurze Schreibweise als Funktion beibehalten werden kann. Dafür wird jedoch der Programmcode entsprechend länger.

`Inline` wird in erster Linie bei kurzen Funktionen angewandt. Hier ist das Verhältnis von Verwaltungsoverhead zu Ausführungsdauer besonders ungünstig und die Auswirkungen auf die Länge des Gesamtcodes gering. Bei langen Funktionen dagegen fällt die Zeit für Aufruf und Parameterübergabe gegenüber der Ausführungsdauer nicht ins Gewicht. Hier würde die Verwendung von `inline` in erster Linie den Code aufblasen. Inline-Funktionen lohnen sich bei sehr kleinen Funktionen (Daumenregel: weniger als drei Anweisungen) sowie bei Anwendungen, die absolut zeitkritisch sind.

Unterschied zu Präprozessormakros

Im Unterschied zu Präprozessormakros besitzen Inline-Funktionen einen festgelegten Typ für Parameter und Rückgabewert. Im Gegensatz zu Makros findet für die Übergabeparameter eine Typprüfung statt.

Dafür treten bei ihnen keine der gefährlichen Seiteneffekte mehr auf.

Beispiel:

```
#include <stdio.h>
#include <time.h>
#include <dos.h>


double poly (double x, int y, int z, int w)
{
 return (x*x + y/z  -w);
}

inline double poly2 (double x, int y, int z, int w)
{
 return (x*x + y/z  -w);
}

void main ()
{long lv;
 double diff;
 time_t first, second;

 first = time (NULL);                 // Systemzeit geholt
 for (lv = 1; lv < 3000000; lv++)
   poly (lv/100., 10, 1, 1);
 second = time (NULL);                //Systemzeit geholt

 diff = difftime (second, first);
 printf("\n\nOhne inline gebraucht:%3.0f Sekunden \n",diff );

 first = time (NULL);                 // Systemzeit geholt
 for (lv = 1; lv < 3000000; lv++)
   poly2 (lv/100., 10, 1, 1);
 second = time (NULL);                //Systemzeit geholt

 diff = difftime (second, first);
 printf("\n\nMit inline gebraucht:%3.0f Sekunden \n",diff );
}
```

Inline-Funktionen sind natürlich immer bei Realzeit-Aufgaben von Bedeutung, so kann man beispielsweise in Ada auch Inline-Assembler-Code für besonders zeitkritische Aufgaben einfügen. Auch der Borland C++ Compiler unterstützt Inline-Assembler-Code.

Inline hat bei der objektorientierten Programmierung deshalb eine besondere Bedeutung, da man beim Programmieren mit Klassen typischerweise viele kleine Funktionen verwendet. Wo in traditionell strukturierten Programmen einfach eine Folge von Anweisungen stehen würde, die auf allgemein übliche Weise auf eine Datenstruktur zugreifen würde, stellt man bei diesem Programmierstil bereits Funktionen zur Verfügung, da man von Beginn an die Daten und Funktionen der Objekte im Auge hat. Dies kann zu erheblichen Effizienzeinbußen führen, da Funktionsaufrufe erheblich teurer sind. Auf dieses Problem hin wurde die **`inline`**-Fähigkeit von Funktionen entworfen.

Hinweis für später:

Eine Member-Funktion, die in der Klassendeklaration bereits definiert wird, behandelt der Compiler als **`inline`**-Funktion.

Mit anderen Worten, man kann eine Klasse entwerfen, ohne auch nur ein Minimum an Laufzeit-Einbußen in Kauf nehmen zu müssen. Daher gibt es keine Notwendigkeit für öffentliche Datenelemente. Alle Daten können in Objekten gekapselt werden - die Performance bleibt mit Hilfe der Inline-Funktionen dennoch gewahrt.

2.7 Speicherverwaltung in C++ - die Operatoren new und delete

Zum Anlegen und Freigeben von dynamischen Variablen auf dem Heap gibt es die beiden neuen Operatoren `new` und `delete`. Sie sollen die aus C bekannten (und trotzdem weiterhin verfügbaren) Bibliotheksfunktionen der `malloc/free`-Gruppe ersetzen. Die neuen Operatoren bieten die Möglichkeit, eventuell auftretende Fehler zu behandeln, und können darüberhinaus für selbst definierte Typen (Klassen) angepaßt werden.

Operatoren new und delete

Der unäre (und damit rechtsassoziative) Operator `new` erzeugt ein Objekt vom angegebenen Typ und liefert als Ergebnis einen Zeiger auf das geschaffene Objekt zurück. Damit entfällt auch die Typanpassung des Rückgabewertes, die in C++ bei `malloc` erforderlich wäre, in C jedoch nicht notwendig war (`malloc` liefert einen Zeiger auf `void`, siehe Kap. über `void`). Ein mittels `new` erzeugtes Objekt kann nur mittels `delete` wieder explizit gelöscht werden. Damit wird verhindert, daß der bei der Definition reservierte Speicherplatz nach Verlassen des entsprechenden Blocks freigegeben wird. `delete` muß dabei als Übergabeparameter einen Zeiger auf das Objekt erhalten, welches zuvor mit `new` erzeugt wurde.

Es ist möglich, durch `new` erzeugte Objekte schon beim Aufruf von `new` zu initialisieren, indem eine Initialisierungsliste in Klammern angegeben wird.

Beispiele:

```
int * int_objekt = new int (7);  // Initialisierung mit 7
int * vektor_obj = new int [7];  // Vektor mit 7 Elementen
```

Der Unterschied zwischen runden und eckigen Klammern ist also gravierend! In runden Klammern werden Initialisierungswerte angegeben, in eckigen Klammern wird die Anzahl der für einen Vektor zu beschaffenden Elemente angegeben. Die Angabe einer leeren Klammer bei der Initialisierung ist äquivalent zu der Angabe keiner Klammer: das geschaffene Objekt wird nicht initialisiert. Vektoren können nicht initialisiert werden.

Der ebenfalls unäre Operator `delete` benötigt als Operanden einen Zeiger auf ein Objekt, welches zuvor mittels `new` erzeugt wurde. Die Auswirkung der Anwendung von `delete` auf ein Objekt, das nicht mittels `new` erzeugt wurde, ist undefiniert. `delete` hat keinen Rückgabewert.

Handelt es sich bei dem Operanden um einen Vektor, der freigegeben werden soll, so ist ihm ein eckiges Klammerpaar nebst Blank voranzustellen (im Beispiel:

`delete [] vektor_obj`). Achtung: Ein Aufruf von `delete` ohne die eckigen Klammern gibt nur das erste Element des Vektors frei (ohne Fehlermeldung!). Die restlichen Elemente sind nicht mehr zugreifbar, könnten also noch nicht einmal mehr gelöscht werden (Speicher-Leck). Frühere Versionen von C++ verlangten die Angabe der Dimension des Vektors in den Klammern. Manche Compiler unterstützen dies auch noch heute aus Kompatibilitätsgründen.

Beispiel:

```
int * int_obj = new int;
delete int_obj;

int * vektor_obj = new int [20];
delete [] vektor_obj;
```

Dynamische Variablen werden also per Operator new im Heap angelegt. Sie können nicht über einen Namen angesprochen werden. Der Zeiger, den der Operator new liefert, ist die einzige Möglichkeit, auf sie zuzugreifen. Lebensdauer und Gültigkeit einer dynamischen Variablen unterliegen nicht den Blockgrenzen der Funktion, innerhalb der sie geschaffen wurden. Dynamische Variablen existieren bis zur ihrer expliziten Vernichtung durch den Operator delete bzw. bis zum Programmende.

Der durch new reservierte Speicherbereich wird auch dann nicht freigegeben, wenn er nicht mehr zugreifbar ist, weil kein Zeiger mehr auf ihn zeigt. Es wird also keine sogenannte garbage collection (Müll-Einsammlung) durchgeführt. Man sollte daher Objekte, die nicht mehr benötigt werden, immer durch `delete` löschen, um unnötige Speicherbelegungen zu vermeiden.

Bitte beachten Sie, daß `new`/`delete` inkompatibel ist mit `malloc`/`free`, d.h. Reservierung mit `new` erfordert Freigabe mit `delete` (nicht `free`!) und umgekehrt. Die gleichzeitige Verwendung von beiden Verfahren nebeneinander in einem Programm ist zwar (noch) möglich, aus Gründen der Übersichtlichkeit aber trotzdem unbedingt zu vermeiden.

Ausnahmebehandlung

Wird `new` aufgerufen und steht nicht mehr genügend Speicherplatz zur Verfügung, so wird üblicherweise `NULL` zurückgegeben. Dies läßt sich abändern: bei jedem Fehler in `new` wird ein sogenannter Handler aufgerufen. Im Anwendungsprogramm hat man die Möglichkeit, die Funktion `set_new_handler` aufzurufen und ihr eine Adresse einer selbst geschriebenen Funktion zu übergeben, die dann von der Funktion `set_new_handler` als Handler eingetragen wird. Im Fehlerfall sollte diese Funktion z.B. das Programm mit einer Fehlermeldung abbrechen oder versuchen, durch Freigabe alter Daten Platz für einen neuen Versuch zu schaffen. Wenn Sie sich in diesem Zusammenhang für weitere Details interessieren, lesen Sie bitte in Ihrem Compilerhandbuch nach.

2.8 Strukturen

Der Datentyp einer Struktur besteht in ANSI-C aus dem Schlüsselwort `struct` und dem **structure tag**, einem vom Programmierer festgelegten Namen, der die Struktur identifiziert.

Beispiel:

```
struct point { int x;
               int y;
};
```

Der Datentyp ist `struct point`.

In C++ kann das Schlüsselwort `struct` bei der Typangabe entfallen. Dann ist das structure tag gleichzeitig auch der Typname, d.h. der Datentyp heißt `point`.

In C++ kann man in einer Strukturdefinition neben Daten auch Funktionen definieren.

Wie Sie in Kap. 3.2 sehen werden, ist dies der Ansatzpunkt, um einen Datentyp zu definieren, der sowohl Daten als auch Funktionen als Komponenten hat. Da man auf die Komponenten eines Objektes vom Typ einer Struktur zugreifen kann, muß man dann nur noch einen Weg finden, um die Daten in einer Struktur zu schützen, und schon ist man bei einem Objekt, das für die Daten das Prinzip des Information Hidings realisiert.

Beispiel:

```
//Datei Strukt.cpp
#include <stdio.h>


struct spiel {
   int x;
   int y;
   void f1() {
       x = 2;
       y = 3;
       };
   void f2() {
       printf ("\n\nf2: oh, happy day!");
       };
};


void main ()
{  spiel alpha;
   printf ("\nx=%d y=%d", alpha.x, alpha.y);
   alpha.f1();
   alpha.f2();
   printf ("\nx=%d y=%d", alpha.x, alpha .y);
   alpha.x = 12;
   printf ("\nx=%d y=%d", alpha.x, alpha.y);
   alpha.f2();
 }
```

Hier die Ausgabe des Programms:

```
x=270 y=1478   // zufällige Werte,
               // x und y wurden noch nicht initialisiert
f2: oh, happy day!
x=2 y=3
x=12 y=3

f2: oh, happy day!
```

Der Zugriff auf die Daten wie auch auf die Funktionen erfolgt über ein Objekt der entsprechenden Klasse. Beispiel:

```
alpha.x = 12;
alpha.f1();
```

3 Ein-/Ausgabe in C++

Wie bereits in C, so sind auch die neuen I/O-Routinen von C++ nicht Teil der Sprache, sondern werden in Form von Bibliotheken geliefert. Da C++ außerdem weitgehend zu C kompatibel ist, heißt das, daß die neuen Routinen kein Muß sind, sondern lediglich eine weitere Möglichkeit zur Ein-/Ausgabe darstellen. Die I/O-Funktionen von C aus `stdio.h` und sämtliche anderen alten Headerdateien können auch weiterhin verwendet werden. Dennoch ist es von Vorteil, sich rechtzeitig an die Verwendung der neuen Funktionen zu gewöhnen, um z.B. eigene Typen in das I/O-Modell integrieren zu können.

Bei den I/O-Funktionen gilt ebenso wie bei den Funktionen zur Verwaltung des Heapspeichers die dringende Empfehlung, alte und neue Verfahren tunlichst nicht zu mischen, sondern sich für eines der beiden zu entscheiden - möglichst für die neue Form!

Es sei darauf hingewiesen, daß zum vollen Verständnis der neuen I/O-Funktionen ein vertieftes Wissen aus dem Bereich der OOP erforderlich ist. Aus diesem Grunde werden sie auch erst im Fortgeschrittenenkurs in ausführlicher Form behandelt. Dennoch ist es möglich, sie bereits jetzt zu verwenden, wobei natürlich viele Möglichkeiten noch nicht ausgeschöpft werden.

3.1.1 Das Streamkonzept

Die Sprache C war schon immer sehr stark mit dem Betriebssystem UNIX verknüpft. Unter diesem Betriebssystem ist der Ihnen geläufige Begriff der Datei und insbesondere des Streams (Zeichenstrom) nicht auf physikalische Dateien auf einem Datenträger beschränkt, sondern findet auch für andere Ein-/Ausgabekanäle und Geräte wie den Bildschirm oder die Tastatur Verwendung. Sicherlich sind Ihnen die entsprechenden Dateien `stdout, stdin` und `stderr` in C bekannt. Die neuen Bibliotheksfunktionen bleiben diesem Konzept treu und bauen es unter konsequenter Nutzung der Neuerungen von C++ weiter aus. Die ursprüngliche Version dieser Bibliothek, `stream.h`, wurde inzwischen durch eine neue Version, `iostream.h`, ersetzt.

Die Ein-/Ausgabe in den neuen Bibliotheken von C++ basiert nicht auf Dateien, die über Funktionen mit Daten beschickt werden (z.B. mit `printf` **nach** `stdin`) bzw. aus denen ausgelesen wird (`scanf` aus `stdout`), sondern auf der direkten Übergabe von Daten an Stream-Objekte. Zur Erinnerung: ein Stream ist eine Folge von Zeichen. Die Bedeutung dieser Zeichen (Buchstaben, Zahlen, Bitmuster) spielt dabei für die Verarbeitung keine Rolle. Innerhalb der neuen Headerdateien werden Klassen deklariert, mit deren Instanzen verschiedene Arten von Streams implementiert werden können. Die wichtigsten dieser Klassen sind `istream` zur Realisierung von Streams zur Eingabe (input) und `ostream` zur Realisierung von

Streams zur Ausgabe (output) von Daten. Desweiteren werden noch Klassen für Datenpuffer definiert, die bei der Bearbeitung von Files wichtig sind.

3.1.2 Streams für Bildschirm und Tastatur

Tröstlich ist, daß man Streams verwenden kann, ohne viel über die Hintergründe und Mechanismen ihrer Implementation zu wissen. Das vorliegende Kapitel soll die Fähigkeit vermitteln, Streams für die unformatierte Ein- und Ausgabe an Bildschirm und Tastatur verwenden zu können.

Für die Kommunikation mit der Tastatur und dem Bildschirm werden innerhalb der Headerdateien folgende Stream-Objekte definiert:

`cin`	als Instanz der Klasse `istream`
`cout`	als Instanz der Klasse `ostream`
`cerr`	als Instanz von `ostream`

Die Streams `cin`, `cout` und `cerr` entsprechen den aus C bekannten Datein `stdin, stdout` und `stderr`. Zum Austausch von Daten mit diesen (und anderen) Stream-Objekten wurden die Bitshift-Operatoren << und >> zu Ausgabeoperatoren bzw. Eingabeoperatoren überladen. Sie bleiben selbstverständlich auch als Bitshift-Operatoren verfügbar. Der Compiler erkennt ja anhand der übergebenen Parameter, welche Operator-Funktion er aufrufen muß. Die neuen Operatoren sind aufgrund ihrer Richtung eindeutig hinsichtlich ihrer Bedeutung und für sämtliche Standarddatentypen bereits vordefiniert.

```
cin  >> Eingabevariable;
cout << Ausgabeausdruck;
cerr << Ausgabeausdruck;
```

Das Stream-Objekt `cin` übergibt ein Eingabedatum an die Variable `Eingabevariable`, während das Streamobjekt `cout` ein Ausgabedatum von dem `Ausgabeausdruck` übernimmt. Die neuen Operatoren sind so definiert, daß verkettete Ein-/Ausgaben in einer Zeile möglich sind:

```
cin >> in1 >> in2 >> in3;
```

Diese Eingabe ist äquivalent zu:

```
cin >> in1;
cin >> in2;
cin >> in3;
```

```
cout << "Heute ist der " << tag << ". August " << 1993;
```

Diese Anweisung ist äquivalent zu:

```
cout << "Heute ist der ";
cout << tag;
cout << ". August ";
cout << 1993;
```

Vorsicht bei der verketteten Eingabe über `cin`. Die einzelnen Eingaben müssen durch Whitespace (nicht Kommata o.ä.) getrennt werden, andernfalls sind bizarre Ergebnisse die Folge.

Um die Ausgabe (und die Eingabe) in ähnlich komfortabler Weise wie mit `printf` formatieren zu können, stehen eine ganze Reihe von Manipulatoren zur Verfügung. Darauf kann jedoch an dieser Stelle nicht eingegangen werden. Wir müssen uns hier auf die Ein-/Ausgabe mit Standardformaten beschränken.

Beispiel:

```
#include <iostream.h>

void main (void)

{int in1, in2, in3;

// Verkette Ein/Ausgabe:

 cout << "\nGib einen integer-Wert für den Tag, den Monat und das"
      << "Jahr ein:\n";
 cin >> in1 >> in2 >> in3;
 cout << "Heute ist der " << in1 << "."<<in2<<"."<<in3;

// Äquivalente Folge von Einzelaufrufen:

 cout << "\nGib einen integer-Wert für den Tag, den Monat und das";
 cout << "Jahr ein:\n";
 cin >> in1;
 cin >> in2;
 cin >> in3;
 cout << "Heute ist der ";
 cout << in1;
 cout << "." ;
 cout <<in2;
 cout <<".";
 cout <<in3;
```

4 Klassen

4.1 Einführende Anwendungsbeispiele

Die grundlegenden Begriffe und Vorgehensweisen der objektorientierten Programmierung wurden bereits vorgestellt. Hier ein einführendes Anwendungsbeispiel:

Zum besseren Verständnis nehmen wir als Beispiel für eine Klasse die Klasse Motorboot:

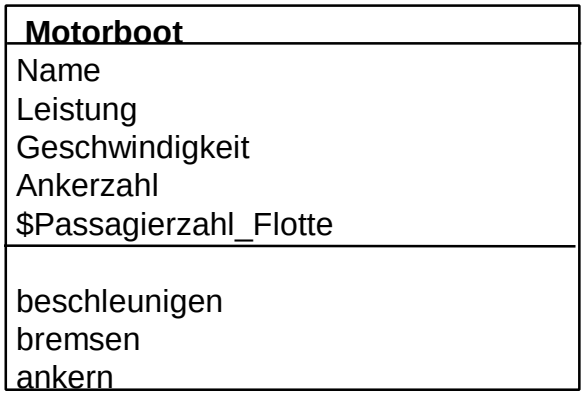


Bild 4 - 1 Klasse Motorboot

Ein konkretes Motorboot, z.B. das Motorboot MS Prinzess ist ein Objekt der Klasse Motorboot. Ein anderes Objekt (hier: der ***Kapitän_MS_Prinzess***) kann nun das Objekt MS_Prinzess (Konkretisierung der Klasse ***Motorboot***) veranlassen, eine Methode (***bremsen***) auszuführen, wobei die Nachricht (ebenfalls ***bremsen***) als Parameter den Wert enthält, den das Attribut Geschwindigkeit nach dem Bremsmanöver besitzen soll (siehe Bild 4 - 2).

Zielobjekt	Methode	Parameter
MS Prinzess	bremsen	10

Bild 4 - 2 Nachrichtenformat

Im Rahmen der OMT-Methode würde man zunächst im Rahmen des Objektmodells notieren, daß zwischen der Klasse Kapitän und der Klasse Motorboot eine Beziehung (Assoziation) herrscht.

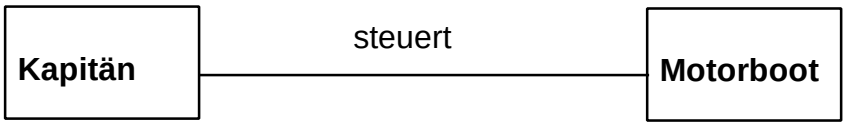


Bild 4 - 3 Objektmodell

Das **Objektmodell** zeigt die Beziehungen zwischen den Klassen.

Assoziationen liest man nach der Regel: von links nach rechts bzw. von oben nach unten. Damit ist klar, daß der Kapitän das Motorboot steuert und nicht das Motorboot den Kapitän.

Um das dynamische Verhalten zu veranschaulichen, gibt es in der OMT-Methode die sogenannten **Event Trace Diagramme**. Sie gehören zu dem sogenannten **Dynamischen Modell** der OMT. Ein solches Schaubild würde in unserem Falle folgendermaßen aussehen:

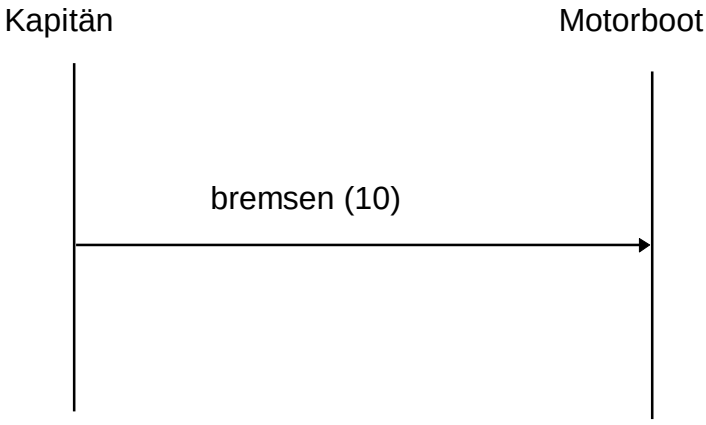


Bild 4 - 4 Event Trace Diagramm

In einem Event Trace muß man sich die Zeitachse vertikal von oben nach unten vorstellen. Man kann mit Hilfe eines Event Trace Diagrammes Szenarien darstellen, welche die Reihenfolge der Ereignisse darstellen.

Durch seine Attribute besitzt ein Objekt immer einen lokalen Zustand, der sich durch Eintreffen einer Nachricht und der darauffolgenden Ausführung einer Methode ändern kann. Diese Eigenschaften (Attribute) dürfen von außen nicht direkt verändert werden, sondern nur indirekt über die Methoden. Nach außen macht ein Objekt oftmals nur seine Methoden sichtbar, d.h. Anzahl und Zustand der Attribute sind nicht direkt erkennbar (wiederum nur über **Methoden**). Ein Objekt wird erst dann aktiv, d.h. es führt eine Methode aus, wenn es durch eine Nachricht dazu aufgefordert wird. Man sagt auch, daß eine **Methode aktiviert** wird. An dieser Stelle soll erneut darauf hingewiesen werden, daß im Rahmen von Smalltalk die Aktivierung mit Hilfe von Nachrichten erfolgt, in C++ hingegen durch Zugriff auf die

entsprechende Methode des Objektes mit Hilfe des Punkt- oder Pfeiloperators, so wie man den Zugriff auf Komponenten von Strukturen durchführt.

Das folgende Bild zeigt eine Reihe von Motorbooten, die zur selben Flotte gehören. Sie werden als Passagierschiffe eingesetzt.

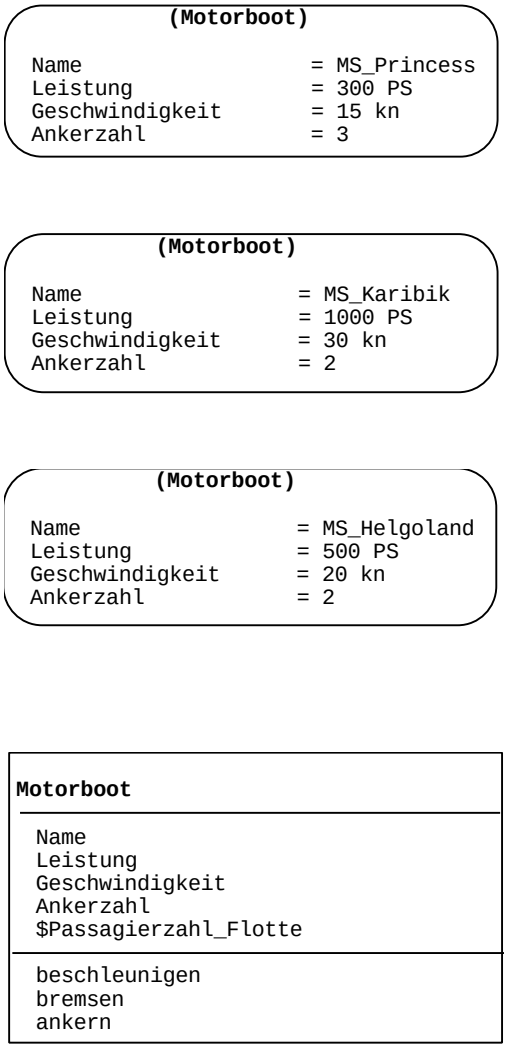


Bild 4 - 5 Objekte der Klasse Motorboot mit gemeinsamer Variable `Passagierzahl_Flotte`

Aufbau von Klassen

Wie man aus dem obigen Beispiel erkennen kann, gibt es für Objekte einer Klasse Daten, die für jedes Objekt spezifisch sein können, und Daten, die für die ganze Klasse identisch sind. Die instanzspezifischen werden **Instanzvariablen** genannt, die für die ganze Klasse identischen heißen **Klassenvariablen.** In C++ werden Klassenvariable als **statische Variable** eingeführt. In obigem Beispiel ist `Passagierzahl_Flotte` eine Klassenvariable.

Methoden, die auf die instanzspezifischen Daten zugreifen, werden auch als **Instanzmethoden** bezeichnet. Daneben gibt es **Klassenmethoden**, die die **Klassenvariablen** bearbeiten. In C++ werden Klassenmethoden durch **statische Funktionen** realisiert. Diese statischen Funktionen bearbeiten die Klassenvariablen, die - wie schon gesagt - in C++ durch **statische Variablen** realiert sind. Die Instanzmethoden werden in C++ auch als nicht statische Funktionen bezeichnet.

Eine Beschreibung der Klasse Motorboot in C++ ähnlicher Notation lautet:

```
class Motorboot
{
 private
    char Name [30]
    int Leistung
    int Geschwindigkeit
    int Ankerzahl
    static int Passagierzahl_Flotte
 public
    void beschleunigen (int delta)
    void bremsen (int delta)
    void ankern ()
}
```

Aus der obigen Definition kann man erkennen, daß Klassen Datentypdefinitionen sind, die die Struktur von Objekten (Attribute) zusammen mit den darauf ausführbaren Funktionen (Methoden) festlegt. Dies befähigt den Programmierer zur Datenabstraktion. Durch die Verwendung von Klassen zur Beschreibung abstrakter Datentypen wird einerseits die Sicherheit erhöht und anderseits die Modifikationsfreundlichkeit verbessert.

Ober- und Unterklassen

Während man gleichartige Objekte zu einer Klasse zusammenfaßt, können ähnliche Objekte Klassen zugeordnet werden, die voneinander abgeleitet werden. Dabei besitzt eine von der Klasse X (*Oberklasse*) abgeleitete Klasse Y (*Unterklasse*) neben den Methoden und Attributen von X noch eigene, zusätzliche Methoden und/oder Attribute.

Als Beispiel wird eine Klasse Motorsegler betrachtet. Ein Motorsegler ist sicherlich einmal ein Motorboot, da es Attribute wie Leistung, Geschwindigkeit, Ankerzahl etc. besitzt und auch die angegebenen Methoden ausführen kann. Zusätzlich besitzt ein Motorsegler noch Methoden wie "Segel setzen" und "Segel bergen" und Attribute wie Segelgröße und Segelanzahl (Bild 4-6).

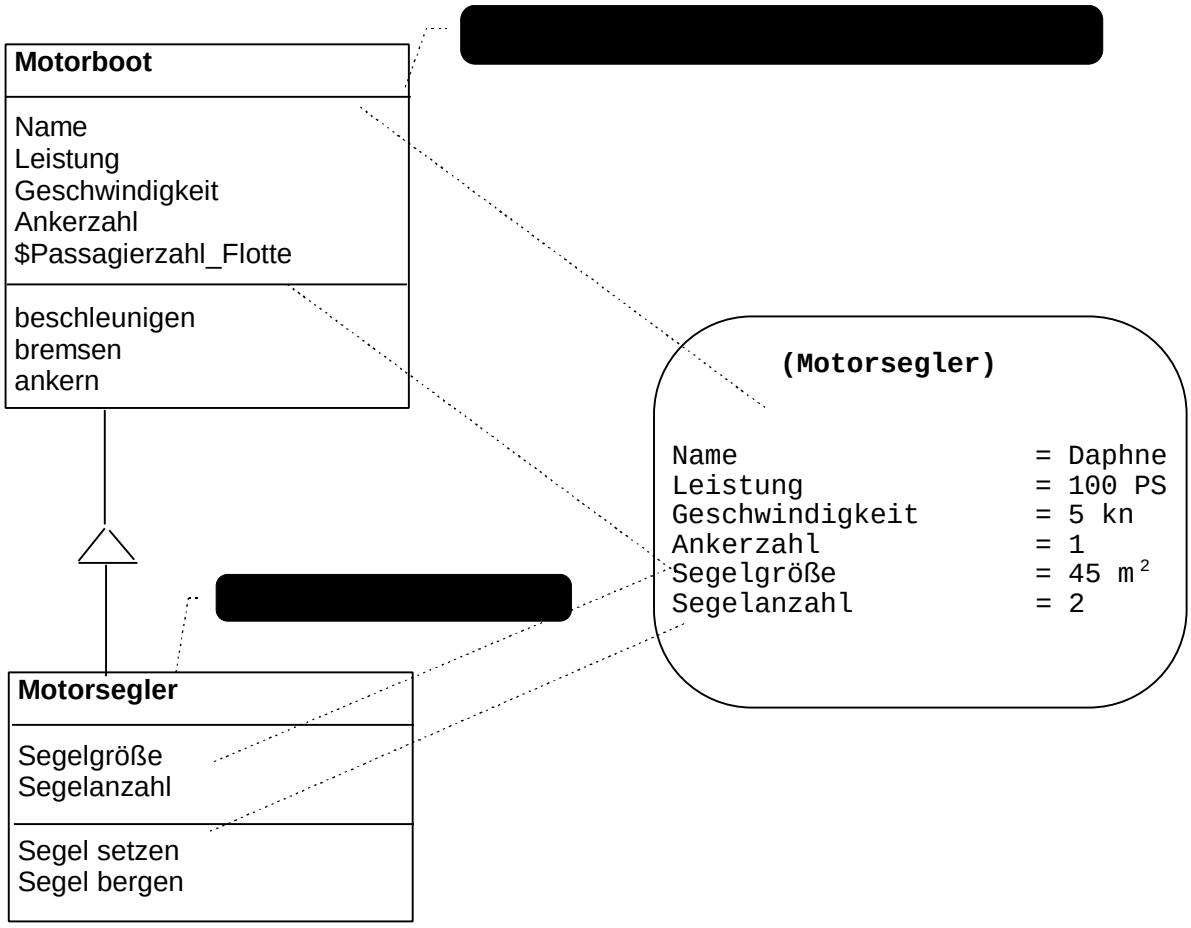


Bild 4 - 6 ***Ableitung der Klasse Motorsegler***

Das Objekt Daphne der Klasse Motorsegler besitzt alle Attribute von Motorboot und Motorsegler und kann auch alle Methoden der beiden Klassen ausführen. D.h. die Klasse Motorsegler erbt Methoden und Attribute der Klasse Motorboot.

Abstrakte Klassen

Einige Klassen einer Klassenhierarchie werden oft als abstrakte Klassen realisiert. Dies sind Klassen, von denen keine Instanzen (Objekte) gebildet werden können. Darin sind Methoden definiert, die keinen bzw. einen leeren Rumpf haben und von denen deshalb auch keine Objekte gebildet werden können. Abstrakte Klassen werden gebildet, um auf oberster Ebene einerseits Attribute festzulegen und anderseits Methoden zu deklarieren, die von den abgeleiteten Klassen dann realisiert werden. Damit wird Overhead vermieden, weil z.B. Attribute nur einmal in einer Oberklasse definiert werden und nicht mehrfach in den Unterklassen.

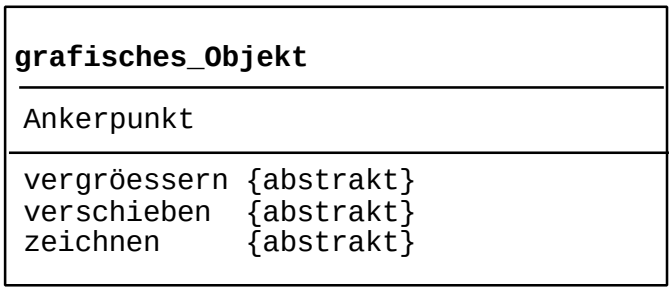
Ein verständliches Beispiel für eine abstrakte Klasse ist die Oberklasse der grafischen Objekte (*graphisches_Objekt*), von der die Unterklassen Punkte, Rechtecke und Kreise abgeleitet werden. In der Oberklasse werden die Attribute Ankerpunkt und Ausdehnung sowie die Methoden `vergrößern, verschieben`,

zeichnen usw. festgelegt (d.h. deklariert). Die Realisierung der Methoden erfolgt aber erst in den einzelnen Unterklassen, da z.B. das Zeichnen eines Kreises eine andere Funktion ist als das Zeichnen eines Rechtecks.

Die **zwei** Klassen *grafisches_Objekt* und *Kreis* könnten folgendermaßen realisiert werden (in C++ ähnlicher Notation):

```
class grafisches_Objekt

{
 private
    coords Ankerpunkt    // mit coords werden Koord. angegeben
 public
    void vergrößern ()   // Methodendeklaration ohne Rumpf
    void verschieben ()
    void zeichnen ()
}
```

Hier die OMT-Notation der Klasse



Optional kann {abstrakt} auch hinter dem Klassennamen stehen.

```
class Kreis : grafisches_Objekt // : heißt "abgeleitet von"
{
 private
 // Ankerpunkt = Mittelpunkt wird aus der Oberklasse geerbt
 public
    void vergrößern ()   // Realisieren der Methode
    void verschieben ()  //      "       "     "
    void zeichnen ()     //      "       "     "
}
```

Vererbung

Vererbung ist **das** Konzept der objektorientierten Programmierung und ist deshalb in allen objektorientierten Sprachen enthalten. Sein Hauptzweck liegt darin, die Ähnlichkeit von neu zu schaffenden Klassen mit bereits vorhandenen Klassen auszunutzen - und zwar entweder im Sinne von Spezialisierung ("von oben nach unten") oder von Abstraktion ("von unten nach oben"). Bei der Vererbung unterscheidet man zwischen zwei Varianten:

-> Einfachvererbung

-> Mehrfachvererbung

Einfache Vererbung

Das Beispiel mit den Klassen ***Motorboot*** und ***Motorsegler*** zeigt eine einfache Vererbung mit nur zwei Hierarchiestufen. Eine Vererbung ist aber prinzipiell über **n** Stufen möglich. Verschiedene Klassen bilden durch ihre Verknüpfung Klassenhierarchien. ***Einfachvererbung*** liegt dann vor, wenn jede Klasse einer Klassenhierarchie nur genau eine direkte Oberklasse (und deren Oberklassen) hat.

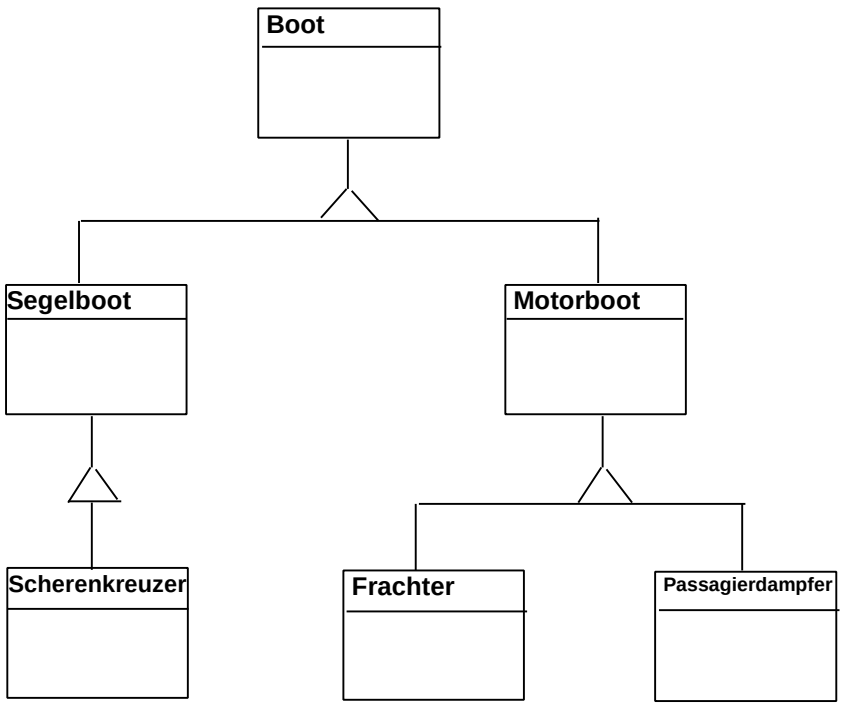


Bild 4-1 Klassenhierarchie mit Einfachvererbung

Ein Objekt der Klasse Frachter erbt also die Methoden und Attribute der Klasse Boot und Motorboot. Soll nun eine Klasse Motorsegler eingefügt werden, die sowohl von der Klasse Segelboot als auch von Motorboot erbt, ist dies bei objektorientierten Sprachen mit Einfachvererbung nicht möglich.

In so einem Fall müßten bei einer Realisierung als Unterklasse von Segelboot die Methoden und Attribute von Motorboot innerhalb der Klasse Motorsegler neu implementiert werden. Dies widerspricht aber der Forderung, eine hohe Wiederverwendbarkeit zu erzielen.

Mehrfache Vererbung

Um dem o.g. Problem mit der erneuten Implementierung von Klassenteilen zu entgehen, sollte man auf jeden Fall eine Programmiersprache verwenden, die Mehrfachvererbung unterstützt. Mehrfachvererbung liegt dann vor, wenn mindestens eine Klasse einer Klassenhierarchie mehr als eine direkte Oberklasse besitzt. Durch diese Eigenschaft ist es möglich, voneinander unabhängige Klassenhierarchien zusammenzuführen, um in einer eigenen Klasse alle Merkmale dieser Hierarchien ausnutzen zu können.

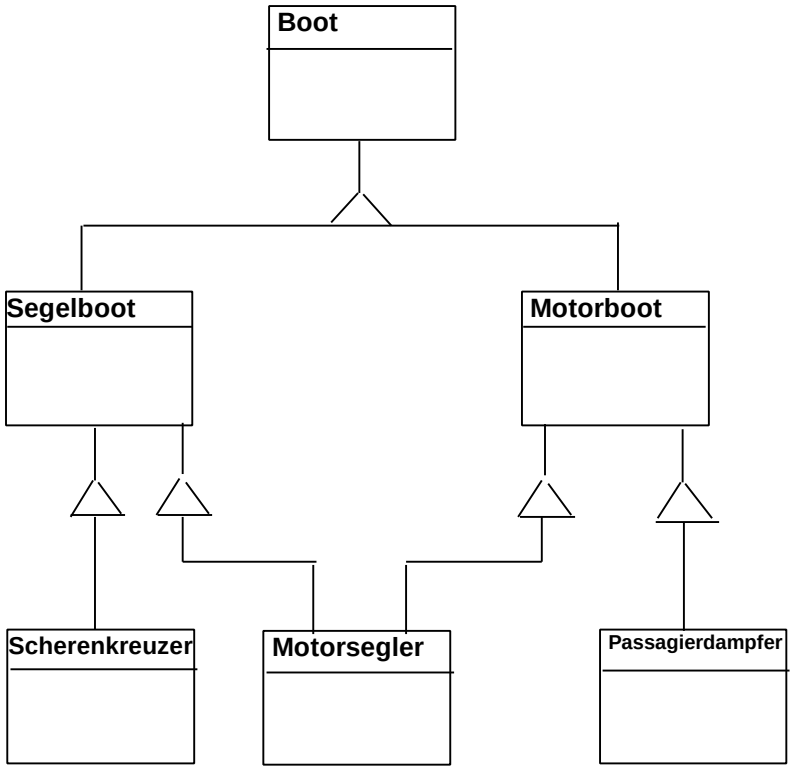


Bild 4-2 Klassenhierarchie mit Mehrfachvererbung

Da eine Klasse, die mehr als eine Oberklasse besitzt, deren Attribute und Methoden erbt, kann es zu Mehrdeutigkeiten kommen. Beispielsweise erbt die Klasse Motorsegler sowohl von Motorboot wie von Segelboot die Methode `ankern`. Wird nun in der Klasse selber die Methode nicht realisiert, so kommt es beim Aufruf der Methode für ein Objekt von Motorsegler zu einem Konflikt. Es ist dann nicht entscheidbar, ob die Methode nun aus Motorboot oder aus Segelboot aufgerufen werden soll. Abhilfe schafft da nur die eigene Realisierung innerhalb von Motorsegler (Bild 4 - 7) oder ein Sprachmittel bzw. eine Angabe an den Übersetzer, nur eine Kopie zu verwenden.

Für die Methoden `bremsen` und `beschleunigen` ist es sinnvoll, beide Methoden zu erben, da der Motorsegler sowohl mit Hilfe der Segel als auch mit Hilfe des Motors beschleunigen und bremsen kann.

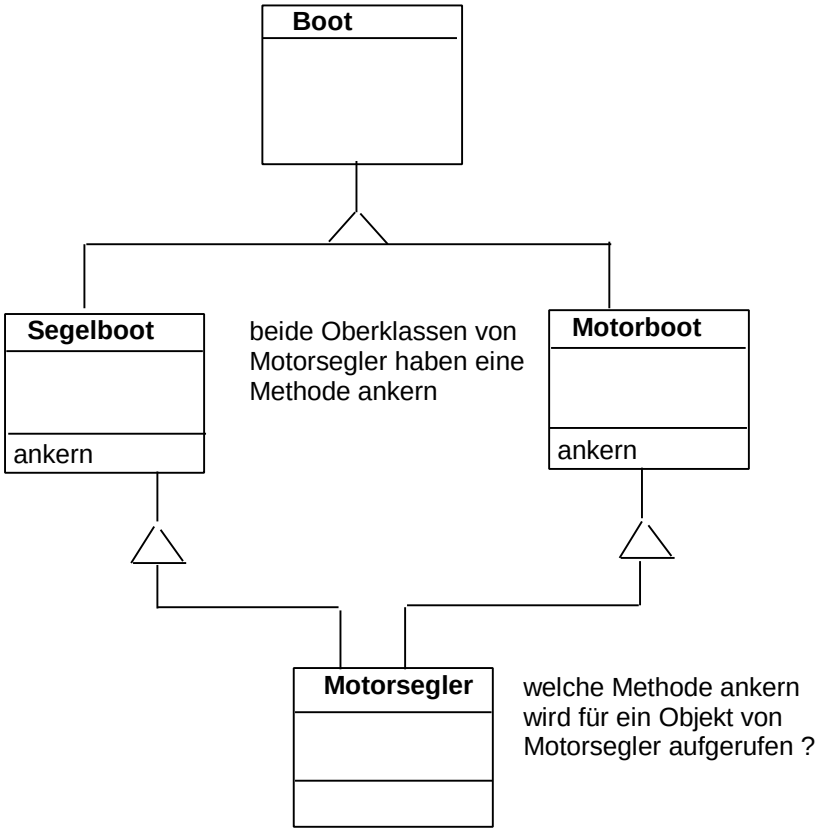


Bild 4 - 7 Probleme bei Mehrfachvererbung

Hier wird man beide Methoden erben wollen. Man muß aber von der Sprache her die Möglichkeit bekommen, gezielt auf die eine oder andere Version zugreifen zu können. In C++ erfolgt dies, indem man beim Motorsegler angibt, auf welche Funktion `ankern` aus welchem Namensraum, d.h. aus welcher Klasse, man zugreifen will.

Durch die mehrfache Vererbung werden die Attribute der Oberklasse Boot zweimal an Motorsegler vererbt. Dies ist nicht immer erwünscht. Daher muß die jeweilige

Programmiersprache die Möglichkeit bieten, daß das Attribut nur einmal angelegt wird.

Wie wir noch sehen werden, bietet C++ hierfür das Sprachmittel, beim Vererben eine Vaterklasse als virtuelle Klasse zu erklären. Dann wird bei einem Enkel oder noch späteren Nachkömmling stets ein Attribut oder eine Methode dieser Virtuellen Vaterklasse nur einfach angelegt, wie oft auch immer vererbt wurde.

Polymorphismus

Unter Polymorphismus im engeren Sinne bzw. dynamischem Binden versteht man die Fähigkeit einer Variablen, auf Objekte unterschiedlicher Klassen einer Klassenhierarchie zu zeigen. Wenn eine Variable ein Zeiger auf die Oberklasse ist, so wird erst zur Laufzeit entschieden, auf welches Objekt einer Unterklasse die Variable tatsächlich zeigt. Damit kann in der Programmierung ein höheres Abstraktionsniveau erreicht werden, denn bei einem Methodenaufruf für die Variable wird zur Laufzeit dynamisch der "richtige" Methodenrumpf (also derjenige für das Objekt) hinzugebunden.

Im Abschnitt "Abstrakte Klassen" in diesem Kapitel wurde als Beispiel die Klassenhierarchie für grafische Objekte aufgeführt. Würde für einen Zeiger auf grafische Objekte die Methode `verschieben` aufgerufen werden, so müßte zur Laufzeit festgestellt werden, auf welches konkrete Objekt der Unterklasse Punkte, Rechtecke oder Kreise gezeigt wird, und welche Realisierung der Methode damit ausgeführt werden muß.

4.2 Definition einer Klasse in C++ anhand eines Beispiels

Eine **Klasse** ist ein **benutzerdefinierter Datentyp (Objekttyp)**. Ein anderes Wort für benutzerdefinierter Datentyp ist **abstrakter Datentyp**. In C++ ist die Klasse eine Erweiterung des `struct`, der Struktur in C. Während ein `struct` in C nur Daten aufnehmen kann, sind in C++ für Klassen Daten und Funktionen erlaubt.

Klassen können als `class, struct` oder `union` definiert werden.

In den Beispielen des Kapitels 3.3 befassen wir uns nur mit Klassen mit dem Schlüsselwort `class`. Auf die Unterschiede zu Klassen mit dem Schlüsselwort `struct` bzw. `union` wird in Kap. 3.4 hingewiesen.

Die Funktionen einer Klasse stellen dabei die Methoden der objektorientierten Programmierung dar. Der besondere Vorteil dabei ist, daß die Daten der Klasse den

Funktionen der Klasse (den **Memberfunktionen**) implizit zur Verfügung stehen. Dies bedeutet, daß die Funktionen einer Klasse alle Daten der Klasse sehen, mit anderen Worten, diese Daten müssen den Funktionen nicht mit Hilfe von Übergabeparametern übergeben werden.

Hinweis:
Der Begriff **member** kommt von der Struktur (`struct`). Die Komponenten eines Structs werden im Englischen als members bezeichnet. Sie stellen in C Daten dar.

Ein **Objekt/Instanz einer Klasse** ist eine physikalisch im Speicher vorliegende Variable. Sie wird in der üblichen Art und Weise deklariert bzw. definiert, und natürlich sind auch Zeiger auf Objekte, Referenzen auf Objekte und konstante Objekte möglich.

Datenelemente sind die bereits von `structs` und `unions` her bekannten Datenfelder innerhalb des Objekts, auf die, sofern sie öffentlich sind, mit der üblichen "." bzw "->" Schreibweise zugegriffen werden kann. Datenelemente werden wir auch als **Attribute** bezeichnen, da diese Bezeichnung bei der Objektorientierten Analyse oft benutzt wird.

Memberfunktionen/Elementfunktionen/Mitgliedfunktionen/Methoden sind die zur Bearbeitung der Daten und zur Realisierung der Schnittstellen im Objekt implementierten Funktionen. Sie werden genau wie die Datenelemente mit derselben Syntax aufgerufen. Sie sind direkt mit dem jeweiligen Objekt verbunden. Eine Methode "weiß", mit welchem Objekt sie gerade arbeitet.

Beispiel:

```
// Datei person.cpp
#include <stdio.h>

class Person {
     // dient zur Speicherung von personenbezogenen Daten
public:
     char * name;
     char * vorname;
     int    alter;
     void   print();
};


void Person::print() {
     printf ("\n\nName:    %s \nVorname: %s \nAlter:   %i\n",
               name, vorname, alter);
}

void main (void)
{
 Person p1, p2;

 p1.name = "Müller";
 p1.vorname = "Fritz";
 p1.alter = 35;

 p2.name = "Meier";
 p2.vorname = "Franz";
 p2.alter = 28;

 p1.print();
 p2.print();
 }
```

Mit

```
class Person {
....
}
```

wird eine Klasse definiert. Diese Klasse kann etwa zur Verwaltung eines Personalstammsatzes in der Buchhaltung verwendet werden. Man sieht sofort die Ähnlichkeit zur gewohnten Strukturdefinition in C. Der Unterschied besteht hauptsächlich in der Deklaration der Funktion `print`, die hier **innerhalb der Klasse deklariert** wird. Die Klasse besteht also aus

?? den **Datenelementen (Mitgliedsvariablen)**: `name`, `vorname` und `alter`
?? und der Funktion (**Mitgliedsfunktion**) `print`

Die Mitgliedsfunktion `print` wird hier **außerhalb der Klasse definiert**. Man sieht, daß im Gegensatz zu gewöhnlichen C-Funktionen bei den Mitgliedsfunktionen

zusätzlich der Klassenname gefolgt von dem Scopeoperator (`::`) angegeben werden muß. An dieser Notation erkennt der Compiler, daß es sich bei `print` um eine Mitgliedsfunktion der Klasse Person handelt. Der Operator `::` hinter dem Klassennamen bewirkt, daß innerhalb der Funktion `print` der Gültigkeitsbereich der Klasse `Person` gilt. Dies bedeutet, daß sich alle Namen in der Funktion auf diese Klasse beziehen.

Andere Klassen können so ebenfalls eine Mitgliedsfunktion `print` definieren, ohne daß es zu Namenskonflikten kommt. Der Compiler weiß also immer genau, zu welcher Klasse eine Mitgliedsfunktion gehört. Wie schon gesagt, kann die `print`-Funktion auf die Mitgliedsvariablen der Klasse zugreifen, sie ist deshalb ohne Übergabeparameter definiert.

Verboten ist es jedoch, zu versuchen, Datenelemente einer Klasse außerhalb der Klasse zu definieren, etwa in der Art

```
float Person::  gehalt;
```

Dies stellt absoluten Unfug dar. Durch ihre Deklaration ist die Klasse mit allen Datenelementen und Mitgliedsfunktionen festgelegt. Bei den Mitgliedsfunktionen reicht es, ihren Aufruf (d.h. die Schnittstelle) zu kennen. Wie in obigem Beispiel gezeigt, kann die Festlegung der Implementation einer Mitgliedsfunktion außerhalb der Klassendeklaration erfolgen.

Wird eine Member-Funktionen vollständig innerhalb der Klassendeklaration definiert, so wird sie automatisch zu einer `inline`-Funktion.

Bei der zweiten, weit gebräuchlicheren Möglichkeit, wird die Funktion innerhalb der Klassendeklaration lediglich deklariert, d.h. ihre Schnittstelle wird innerhalb der Klasse bekannt gemacht, der Funktionsrumpf wird jedoch außerhalb der Klasse definiert. Bei Methoden, die aus mehr als nur einer Handvoll Befehlen bestehen ist dies aus Gründen der Übersichtlichkeit praktisch unabdingbar.

Auch außerhalb definierte Funktionen können `inline` sein, müssen dazu aber mit dem entsprechenden Schlüsselwort gekennzeichnet werden.

Allgemein gilt: Eine Klasse ist in C++ ein eigener Gültigkeitsbereich, alle Mitglieder der Klasse können implizit (d.h. ohne Parameterübergabe) auf alle anderen Klassenmitglieder zugreifen.

Der Gültigkeitsbereich (Bezugsrahmen) einer Deklaration ist jener Bereich des Programmtextes, für den diese Deklaration ihre Bedeutung hat. In diesem Bereich ist das Objekt verfügbar. In C++ kann in drei Bereichen auf ein Objekt Bezug genommen werden:

?? lokal (innerhalb eines Blockes),

?? global (innerhalb einer Datei)

?? oder innerhalb einer Klasse

Bleibt abschließend nur noch anzumerken, daß eine Klassendeklaration wie jede Anweisung durch einen Strichpunkt abgeschlossen werden muß.

Objekte

Analog zu `structs` können von Klassen Variablen gebildet werden. Diese Variablen werden auch **Objekte** oder **Instanzen** genannt, so erzeugt die Anweisung

```
Person p1;
```

ein Objekt der Klasse Person.

Bei der Definition von Objekten einer Klasse werden bei jedem Objekt alle Datenelemente angelegt. Die Funktionen einer Klasse werden nur einmal im Maschinencode angelegt. Sie sind für alle Objekte der Klasse zugänglich.

4.3 Zugriff auf Klassenelemente

Der Zugriff auf Mitgliedsvariablen und Aufrufe von Mitgliedsfunktionen (Member-Funktionen) erfolgt - wie schon gesagt - mit der von den `structs` her bekannten Notation:

```
p1.name = "Müller";
```

bzw. mit

```
p1.print();
```

Die Sprechweise ist:

„Die Mitgliedsfunktion `print` wird für das Objekt `p1` aufgerufen“

oder kürzer:

„für p1 wird `print` aufgerufen“

In C++ werden die Methoden also nicht über Nachrichten aktiviert. Die Aktivierung erfolgt durch Aufruf der Memberfunktion eines Objekts durch Zugriff auf diese Komponente (hier Funktion) des Objekts.

Nicht besprochen ist bis jetzt das Schlüsselwort **`public`**, dies kommt weiter unten. Hier sei jedoch bereits angekündigt, daß das Schlüsselwort `public` die Sichtbarkeit von Klassenmitgliedern für den Rest des Programms regelt.

Man hätte die Funktion `print` allerdings auch gleich in der Klasse definieren können, das hätte dann folgendermaßen ausgesehen:

```
// datei person2.cpp
#include <stdio.h>

class Person {
         // dient zur Speicherung von personenbezogenen Daten
public:
     char * name;
     char * vorname;
     int    alter;
     void   print() {
     printf ("\n\nName:    %s \nVorname: %s \nAlter:   %i\n",
               name, vorname, alter);
     }
};


void main (void)
{
 ...       // genau gleich wie bei Programm person.cpp
}
```

Nächstes Beispiel:

```
// Datei Bruch.cpp

class Bruch {
// diese Klasse dient zur Darstellung eines Bruches

public:
     int zaehler;
     int nenner;

     void print ();
};


#include <stdio.h>

void Bruch::print() {
  printf ("\n der Wert des Quotienten von %i und %i ist %i/%i",
           zaehler, nenner, zaehler, nenner);
}

void main (void)
{

 Bruch b;
 b.zaehler = 1;
 b.nenner = 2;
 b.print();
}
```

Im folgenden Beispiel sind die Klassen `Person` und `Bruch` enthalten:

```
// Datei perbruch.cpp
#include <stdio.h>

class Person {
          // dient zur Speicherung von personenbezogenen Daten
public:
     char * name;
     char * vorname;
     int    alter;
     void   print();
};

void Person::print() {
      printf ("\n\nName:    %s \nVorname: %s \nAlter:   %i\n",
               name, vorname, alter);
}
```

```
class Bruch {
// diese Klasse dient zur Darstellung eines Bruch es

public:
     int zaehler;
     int nenner;

     void print ();
};

void Bruch::print() {
      printf ("\nder Wert des Quotienten von %i und %i ist %i/%i",
          zaehler, nenner,zaehler, nenner);
}

void main (void)
{
 Bruch b;
 b.zaehler = 1;
 b.nenner = 2;

 Person p1, p2;

 p1.name = "Müller";
 p1.vorname = "Fritz";
 p1.alter = 35;

 p2.name = "Meier";
 p2.vorname = "Franz";
 p2.alter = 28;

 p1.print();
 p2.print();
 b.print();
}
```

In diesem Programm sind beide Klassen `Person` und `Bruch` enthalten. An der Klasse der Objekte `p1`, `p2` und `b` erkennt der Compiler, welche Funktion `print()` er aufrufen muß. Diese Entscheidung wird bereits zur Compilierzeit getroffen. Dies ist ein Beispiel für eine sogenannte **statische Bindung**, bei der die Zuordnung eines Funktionsaufrufs zur Compilierzeit erfolgt. Es gibt auch eine sogenannte **dynamische Bindung**, bei der die Zuordnung einer konkreten Funktion zum Funktionsaufruf erst zur Laufzeit des Programms erfolgt.

Obwohl die Funktion `print()` keinen Übergabeparameter hat, erhält sie beim Aufruf über `p1` (`p1.print()`) andere Daten, als beim Aufruf über `p2` (`p2.print()`). Der Compiler realisiert dies intern und greift durch die Nennung des Objektes p bei `p.print` automatisch auf die richtigen Daten zu.

4.4 Der Gültigkeitsbereich einer Klasse

In C++ kann in drei Bereichen auf ein Objekt Bezug genommen werden:
?? lokal (innerhalb eines Blockes),
?? global (innerhalb einer Datei)
?? oder innerhalb einer Klasse

In Kap. 2.7 war ein Beispiel für den Scope-Operator gegeben worden, das den Zugriff auf eine globale Variable bei Vorliegen einer lokalen Variable mit demselben Namen zum Ziele hatte.

Da - wie Sie schon längst wissen - in einer Klassendefinition auch Memberfunktionen definiert werden können, kann sich der Befehlszeiger eines Programms damit auch in einer Klassendefinition befinden.

Sichtbarkeit von Datenobjekten innerhalb einer Memberfunktion:

Alle globalen Variablen sind sichtbar, alle Member-Daten und alle lokale Daten in der Funktion.

Hier nun ein Beispiel für den Zugriff auf ein Klassenelement aus einer Memberfunktion heraus, die eine lokale Variable mit demselben Namen wie das Klassenelement besitzt.

```
// Datei: scopekl.cpp
#include <stdio.h>

class point {
   int x;
   int y;
public:
   point (void);
   void print (void);
   void verschiebe (int delta_x, int delta_y);
};

int x = 333;  //das ist das globale x;

point::point (void) {
  x = 0;
  y = 0;
};

void point::print (void) {
  printf ("\n\nprint: Das Klassenelement x hat den Wert %d", x);
  printf ("\nprint: Das Klassenelement y hat den Wert %d", y);
  printf ("\nprint: Das globale x hat den Wert %d", ::x);
}

void point::verschiebe (int delta_x, int delta_y) {
  int x = 0;
```

```
  point::x = point::x + delta_x;
  y+= delta_y;
  printf ("\nverschiebe: Das Klassenelement x hat  den wert %d", point::x);
  printf ("\nverschiebe: Das Klassenelement y hat den wert %d", y);
  printf ("\nverschiebe: Das lokale x hat den Wert %d", x);
}

void main (void)
{
  point a;
  a.print();
  a.verschiebe (3,4);
  a.print();
}
```

Das Programm gibt aus:

```
print: Das Klassenelement x hat den Wert 0
print: Das Klassenelement y hat den Wert 0
print: Das globale x hat den Wert 333
verschiebe: Das Klassenelement x hat den wert 3
verschiebe: Das Klassenelement y hat den wert 4
verschiebe: Das lokale x hat den Wert 0

print: Das Klassenelement x hat den Wert 3
print: Das Klassenelement y hat den Wert 4
print: Das globale x hat den Wert 333
```

Sie sehen, daß bei einer Memberfunktion durch Angabe der Klasse angegeben wird, daß das entsprechende Klassenelement und nicht die lokale Variable benutzt wird.

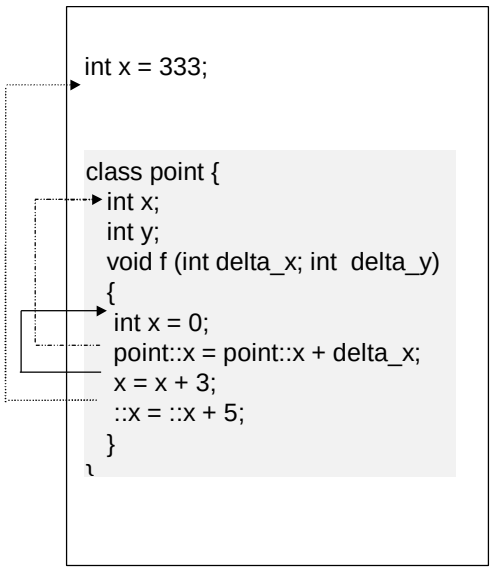


Bild 4 - 8 Zugriff auf lokale Variablen, auf Klassenelemente und globale Variablen

4.5 Kapselung von Klassenelementen

Ein Ziel der objektorientierten Programmierung ist, die Repräsentation der Daten zu verbergen, das bedeutet, daß das **Prinzip des Information Hidings** angewandt wird. Damit kann kein Unbefugter die Daten verändern. Zu den Daten gibt es nur Schnittstellen, die man im Griff hat, nämlich der Zugriff über die Member-Funktionen. Dies bedeutet, daß die obigen Beispiele zwar syntaktisch korrekt sind, jedoch den Zielvorstellungen widersprechen. Es darf also **aus Gründen des Software Engineerings**, die in der Objektorientierung zum Tragen kommen sollen, **keinen Zugriff auf die Daten des Objektes von der Funktion `main` aus** geben. Eine Initialisierung der Member-Daten in `main` ist verpönt.

Wir müssen also die Daten nach außen verbergen. Dies erfolgt mit Hilfe des Schlüsselworts `private`:

```
class Bruch2 {
// diese Klasse dient zur Darstellung  eines Bruches
private:
     int zaehler;
     int nenner;
public:
     void print ();
}
```

Damit hat man von `main` aus keinen Zugriff mehr auf die Daten, sondern nur noch von der Mitgliedsfunktion `print()`.

Dies ist die generelle Vorgehensweise. Man erlaubt fremden Funktionen in der Regel nicht den Zugriff auf die Daten einer Klasse. Dies ist ausschließlich Aufgabe der Member-Funktionen. Damit ist auch bei fehlerhaften Datenbearbeitungen automatisch die Fehlersuche auf die Mitgliedsfunktionen beschränkt.

Da vom Beginn einer Klassendeklaration die Default-Einstellung `private` ist, solange, bis `private` durch `public` aufgehoben wird, muß man `private` nicht extra anschreiben. Möchte der Programmierer anschließend an öffentliche Deklarationen wieder private Deklarationen einfügen, so gibt er im Anschluß an die öffentlichen Deklarationen das Schlüsselwort `private` an. Der Wechsel zwischen `public` und `private` ist beliebig oft möglich.

Die Notation

```
class Bruch2 {
// diese Klasse dient zur Darstellung eines Bruches
     int zaehler;
     int nenner;
public:
     void print ();
}
```

ist äquivalent zu oben.

4.6 Initialisierung von Klassenelementen durch eine spezielle Mitgliedsfunktion

Da wir jedoch die Daten initialisieren wollen, ist nun die Einführung einer **Initialisierungs-Funktion als Mitgliedsfunktion** erforderlich.

Beispiel:

```
class Bruch2 {
// diese Klasse dient zur Darstellung eines Bruches

     int zaehler;
     int nenner;
public:
     void init();
     void print ();
}


#include <stdio.h>

void Bruch2::print() {
     printf ("\n der Wert des Quotienten von %i und %i ist %i/%i",
          zaehler, nenner,zaehler, nenner);
}
void Bruch2::init() {
          zaehler = 0;
          nenner = 1;
}

void main (void)
{

 Bruch2 b;      // bei dieser Definition erhalte n zaehler
                // und Nenner zufällige Werte
 b.init ();     // Initialisierung
 b.print();
}
```

Die Funktion `init` besetzt die Daten des Objektes mit „geeigneten“ Ausgangsdaten. In obigem Beispiel wird ein Objekt der Klasse `Bruch2` nach der Definition initialisiert mit dem Wert 0.

4.7 Unterschiede zwischen Klassen, Strukturen und Unionen

Objekte lassen sich in C++ auf der Basis der strukturierten Typen struct, class und - etwas eingeschränkt -- auch mit union realisieren. In diesem Abschnitt wollen wir uns nun den kleinen aber feinen Unterschieden zwischen den einzelnen Grundformen widmen, um so das Kapitel über OOP abzurunden.

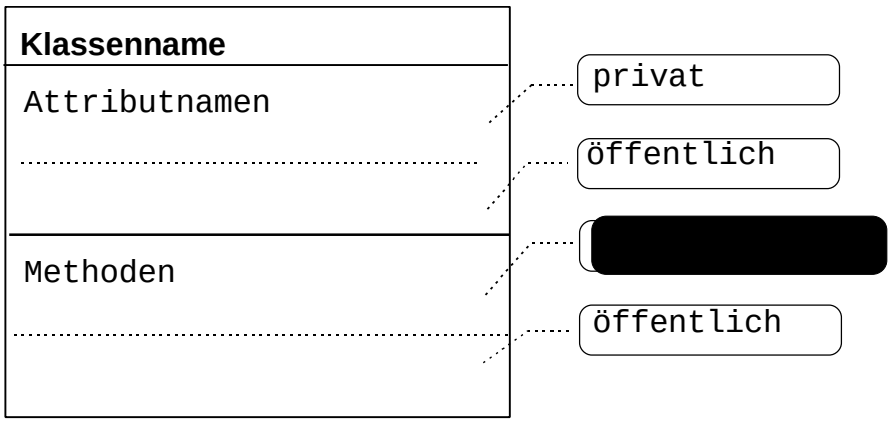


Bild 4 - 9 Darstellung einer Klasse

Klassen können als `class, struct` oder `union` definiert werden.

Bei einer Klasse, die als `struct` oder `union` definiert ist, sind Komponenten ohne Angabe eines Schutzattributes automatisch `public`. Bei `class` ist das voreingestellte Schutzattribut automatisch `private`.

4.7.1 Objekte mit union

Dem Basteln von Objekten mit dem Grundtyp union sind relativ enge Grenzen gesetzt. Dies rührt in erster Linie daher, das unions weder von anderen Objekten abgeleitet werden, noch als Basisklasse (siehe später) dienen dürfen. Damit wird auch die Polymorphie und der Einsatz von virtuellen Funktionen (siehe später) belanglos. Was bleibt, sind Mitgliedsfunktionen und die Möglichkeit, die standardmäßig als public vereinbarten Elemente durch den Einsatz von private oder protected zu schützen. Diese beiden Schutzmechanismen sind bei union-objekten völlig gleichwertig! Da die Datenelemente einer union alle auf dem selben Speicherplatz abgespeichert werden, ist bei der Implementierung von Schutzmechanismen allerdings äußerste Vorsicht geboten. So kann z.B. eine als const deklarierte Variable zwar nicht direkt verändert, jedoch sehr wohl durch eine parallel liegende nicht-const Variable überschrieben werden. Dasselbe gilt auch für anderweitig (scheinbar) geschützte Datenelemente.

```
//
// some experiments with union objects: unionobj.cpp
//
#include <stdio.h>


// Modify this sample object at will.
// Try to inherit sth from another class
union UTestObj
{
  // init for pubdata kills constdata
  UTestObj() : constdata(1000) { }
  UTestObj(int theconst, float theData)
  : constdata(theconst) { pubdata=theData; }

  unsigned getprivdata(void) { return privdata; }
  int getconstdata(void) { return constdata; }

  void setprivdata(unsigned newdata) { privdata=newdata; }
  void constdummy(void) const {}

  float pubdata;

  // same effect as private here
  protected:
  // const is almost useless (just no "direct" modify)
  const  int constdata;
  void proteceddummy(void) {}

  private:              // as usual
  unsigned privdata;
  void privatedummy(void) {}

//  static int counter; // not allowed
  static void staticdummy(void) {};
};

void main(void)
{
  UTestObj obj;

  printf("\n\nThe const is initialized with 1000");
  printf("\nUnion Object:");
  printf("\npublic =%g" , obj.pubdata);
  printf("\nconst  =%d" , obj.getconstdata());
  printf("\nprivate=%#x" , obj.getprivdata());

  obj.pubdata=7;
  printf("\n\nNow setting pubdata to 7");
  printf("\nUnion Object:");
  printf("\npublic =%g" , obj.pubdata);
  printf("\nconst  =%d" , obj.getconstdata());
  printf("\nprivate=%#x" , obj.getprivdata());

  obj.setprivdata(0xffff);
  printf("\n\nNow setting privdata to 0xffff");
```

```
  printf("\nUnion Object:");
  printf("\npublic =%g" , obj.pubdata);
  printf("\nconst  =%d" , obj.getconstdata());
  printf("\nprivate=%#x" , obj.getprivdata());
}
```

4.7.2 Objekte mit struct und class

Für den Grundtyp `struct` steht - genau wie für `class` - die gesamte Bandbreite an Mechanismen zur Verfügung. Vielleicht ist Ihnen bei der Betrachtung der Beispiele ja schon aufgefallen, daß wir uns hier keineswegs auf einen der beiden Typen festlegen wollten. Der einzige Unterschied zwischen den beiden besteht in den standardmäßig gesetzten Defaults für Zugriffsrechte. Während beim Typ `struct` alles standardmäßig auf `public` gesetzt ist, gilt beim Typ class der Modus `private` als Default.

4.8 Schutzmechanismen

Wie bereits erwähnt, besteht die Möglichkeit, Elemente eines Objekts vor einem Zugriff von außen zu schützen. Dies wird durch den Einsatz von drei Schlüsselwörtern erreicht. Mit ihrer Hilfe kann der Zugriffsbereich für alle (bis zum nächsten Schlüsselwort folgenden) Elemente festgelegt werden. Falls kein Schlüsselwort angegeben wird, kommt die Default-Einstellung zum Tragen. Doch nun zu den Schlüsselwörtern selbst:

?? **public (öffentliche Elemente)**
Es bewirkt, daß alle damit gekennzeichneten Elemente für jedermann innerhalb und außerhalb der Klasse zugänglich sind. Dies ist die Default-Vorgabe für `struct` und `union`.

?? **private (private Elemente)**
Dies ist die default-Vorgabe für `class`. Auf so gekennzeichnete Elemente kann nur durch Mitgliedfunktionen der eigenen Klasse zugegriffen werden. Kein Zugriff durch Außenstehende und Nachkommen.

?? **protected (geschützte Elemente)**
Dieses Schlüsselwort ist in seiner Wirkung sehr ähnlich zu `private`. Es verbietet jeglichen Zugriff durch Außenstehende, erlaubt jedoch den Zugriff durch direkte Nachkommen der Klasse.

Innerhalb einer Klassendeklaration sind beliebig viele Bereiche mit jedem der Schlüsselwörter erlaubt. Man wird sich jedoch sinnvollerweise auf einen oder zwei Bereiche pro Schlüsselwort beschränken, je nachdem, ob man die Deklaration

a) nach Zugriffsbereichen und innerhalb derer nach Daten und Methoden oder
b) nach Daten und Methoden und innerhalb derer nach Zugriffsbereichen ordnet.

Dies ist keine Vorschrift des Compilers, sondern eine Empfehlung zur Erhöhung der Übersichtlichkeit.

Eine Struktur sowie eine Union in C ist ein Spezialfall der Klassendefinition, in der alle Member per Definition `public` sind. In C++ können `structs` und `unions` auch die Schlüsselwörter `public` und `private` beinhalten, `protected` ist den Klassen vorbehalten.

Default-Werte sind:	
struct, union:	public
class:	private

4.9 Methoden und this

Methoden sind ein fester Bestandteil ihres Objekts. Innerhalb einer Methode kann direkt auf die Elemente eines Objekts zugegriffen werden. Um dies realisieren zu können, wird jeder Methode bei ihrem Aufruf ein impliziter (unsichtbarer) Parameter übergeben: der **this-Zeiger**, der auf ein Objekt der Klasse zeigt, zu der die Methode gehört.

Beispiel:

```
class Person {
     char name[20];
public:
     void print();
}

void Person::print() {
     printf("Der Name der Person lautet: %s \n",name);
}

void main (void)
{
 Person p;
 p.print();
}
```

Der C++-Compiler wird ein solches Programm umsetzen in:

```
class Person {
     char name[20];
public:
     void print( Person * );
}

void Person::print( Person * this ) {
     printf("Der Name der Person lautet: %s \n", this->name);
}

void main (void)
{
 Person p;
```

```
 Person::print(&p); // Beim Aufruf übergibt der Compiler einen
                    // Zeiger auf das Objekt
}
```

Sie sehen also:

?? jede Memberfunktion wird implizit erweitert um den Parameter **<Klassenname> *** ,
?? jeder Zugriff auf eine Komponente der Klasse wird vom Compiler umgesetzt in einen Zugriff über den `this`-Zeiger (**`this->`**).

Der `this`-Zeiger ist zum normalen Zugriff innerhalb einer Memberfunktion nicht unbedingt nötig, da hier die Angabe der Komponentennamen ausreicht. Daher wird dieser Zeiger meist verwendet, um
?? ein Objekt an eine Funktion zu übergeben, die keine Member-Funktion ist (z.B. eine Freundfunktion (siehe später)) oder
?? um das gesamte Objekt als Return-Wert aus einer Funktion zurückzugeben.

Beispielprogramm:

```
#include <stdio.h>
#include <iostream.h>
#include <string.h>

class Person
{
 private:
       char name[20];
       char vorname[20];
       int gehalt;
 public:
        void print();
        Person( char *, char *, int );
        Person & gehalt_aendern( int );
};


Person::Person( char *n, char *v, int g )
{
        strcpy(name,n);
        strcpy(vorname,v);
        gehalt = g;
}

void Person::print()
{
        // Zugriff implizit ueber this-Zeiger durch Compiler
        printf("\n\nName: %s\nVorname: %s\nGehalt: %6d\n",
               name, vorname, gehalt );

        // Zugriff explizit ueber this-Zeiger
        printf("\n\nName: %s\nVorname: %s\nGehalt: %6d\n",
```

```
                this->name, this->vorname, this->gehalt );
}

Person & Person::gehalt_aendern( int gehalt_neu )
{
        gehalt = gehalt_neu;
        return *this;
}

void main(void)
{
        Person a("Mustermann", "Hans", 2500 );

        a.print();
        // Verkettung von Memberfunktionen
        a.gehalt_aendern( 3000 ).print();
}
```

In diesem Beispiel wird in der Funktion `Person::print()` einerseits direkt auf die Elemente der Klasse zugegriffen, andererseits mit Hilfe des `this`-Zeigers. Wie zu erwarten, ist in beiden Fällen das Ergebnis dasselbe.

Der Aufruf **`a.gehalt_aendern( 3000 ).print();`** ist eine Verkettung von Memberfunktionen. Die Abarbeitung geschieht hierbei von links nach rechts. D.h. zuerst wird die Funktion **`a.gehalt_aendern( 3000 )`** ausgeführt. Mit dem Rückgabewert dieser Funktion wird dann die Funktion **`print()`** aufgerufen.

Sehr sinnvoll ist diese Verkettung hier natürlich nicht. Man hätte genauso gut die Aufrufe **`a.gehalt_aendern(3000 );`** und **`a.print();`** nacheinander schreiben können. Es soll nur an dieser Stelle deutlich machen, welche Möglichkeiten bestehen.

Prinzipiell benötigt man solch ein Vorgehen nur, wenn eine Verkettung von Funktionen auftreten kann wie im folgenden Beispiel:

```
#include <stdio.h>

class Zahl {
     int zahl;
public:
     Zahl( int i ){ zahl = i; };
     Zahl & Inc( );
     Zahl & Dec( );
     int Vergleich( int );
};

Zahl & Zahl::Inc( ){
     zahl++;
     return *this;
}

Zahl & Zahl::Dec( ){
     zahl--;
```

```
     return *this;
}

int Zahl::Vergleich( int i ){
     return i==zahl;
}

void main( void ) {
   const int i = 5;
   Zahl z(i);

// Test der Funktionen Inc und Dec
// wird nacheinander Inc und Dec auf dasselbe Objekt
// aufgerufen, so muß dies wieder die Zahl i ergeben

   if ( z.Inc().Dec().Vergleich( i ) )
     printf("Meine Funktionen sind wohl korrekt \n");
   else
   {
     printf("Es muss wohl noch ein kleiner Fehler ");
     printf("in den Funktionen sein \n");
   }
}
```

4.10 Modularisierung von Programmen

Ein Modul ist eine getrennt compilierbare Einheit - in C eine Datei. Um in Projekten arbeitsteilig arbeiten zu können und damit bei Änderungen von Programmen nur kleine Einheiten abgeändert werden müssen - und die anderen unverändert beim Configuration Manager verbleiben - ist eine Einteilung eines Programmes in Module zwingend notwendig.

4.10.1 Zerlegung in Deklaration und Implementierung von Klassen

Als Beispiel für die Modularisierung wählen wir das zuletzt gezeigte Programm `perbruch.cpp`.

Dieses Programm zerlegen wir in die Module:

```
person.h
bruch.h
person.cpp
bruch.cpp
main.cpp
```

Die einzelnen Module haben die Form:

`person.h`

```
// Datei person.h

class Person {
          // dient zur Speicherung von personenbezogenen Daten
public:
     char * name;
     char * vorname;
     int    alter;
     void   print();
};
```

person.cpp

```
// Datei person.cpp

#include "person.h"
#include <stdio.h>

void Person::print() {
      printf ("\n\nName:    %s \nVorname: %s \nAlter:   %i\n",
               name, vorname, alter);
}
```

bruch.h

```
// Datei bruch.h

class Bruch {
// diese Klasse dient zur Darstellung eines Bruches

public:
     int zaehler;
     int nenner;

     void print ();
};
```

bruch.cpp

```
// Datei bruch.cpp

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

void Bruch::print() {
      printf ("\nder Wert des Quotienten von %i und %i ist %i/%i",
          zaehler, nenner,zaehler, nenn er);
}
```

main.cpp

```
// Datei main.cpp
```

```
# include <stdio.h>
# include "bruch.h"
# include "person.h"

void main (void)
{
 Bruch b;
 b.zaehler = 1;
 b.nenner = 2;

 Person p1, p2;

 p1.name = "Müller";
 p1.vorname = "Fritz";
 p1.alter = 35;

 p2.name = "Meier";
 p2.vorname = "Franz";
 p2.alter = 28;

 p1.print();
 p2.print();
 b.print();
}
```

In den Include-Files mit der Extension .h steht die Deklaration der Klassen. `person.h` enthält die Deklaration der Klasse `Person`, `bruch.h` enthält die Deklaration der Klasse `Bruch`. Der Quellcode der Klassen, d.h. die Datenelemente (Attribute) und die Implementierung der Memberfunktionen, ist in den Dateien `bruch.cpp` und `person.cpp` enthalten. Das Hauptprogramm ist als Applikation in einer separaten Datei `main.cpp` gespeichert. Für jede Klasse, die bei einer Anwendung neu hinzukommt, wird ein Header-File für die Deklaration dieser Klasse und ein Quellfile für die Implementierung ihrer Datenelemente (Attribute) und Memberfunktionen hinzugefügt.

`person.h`	Deklaration der Klasse Person
`bruch.h`	Deklaration der Klasse Bruch
`person.cpp`	Implementierung Memberfunktionen der Klasse Person
`bruch.cpp`	Implementierung Memberfunktionen der Klasse Bruch
`main.cpp`	Anwendungsprogramm

Der Compile-Lauf führt zu den 3 Objektcode-Dateien:

```
person.obj
bruch.obj
main.obj
```

Der Linker bindet daraus unter Einbeziehung der erforderlichen Bibliotheksfunktionen ein ablauffähiges Programm.

Beim Borland-Compiler 3.0 bzw. 3.1 geht im Rahmen der integrierten Entwicklungsumgebung das Erzeugen eines ausführbaren Files über das Anlegen eines sogenannten Projekts. Man klickt „Open Project“ an, gibt einen Projektnamen ein und nimmt dann über „Add Item“ die Quell Files - hier `person.cpp`, `bruch.cpp` und `main.cpp` - in das Projekt auf. Dann compiliert und linkt man mit „Build all“.

4.10.2 Verhindern einer mehrfachen Deklaration einer Klasse beim Arbeiten mit Modulen

Klassen werden in Include-Files vereinbart. Nehmen wir an, die Klasse `alpha` sei in `alpha.h` vereinbart und im Include-File `A.h` und `B.h` würde `alpha.h` mit `#include` aufgenommen. Die Datei `beispiel.cpp` würde beide Include-Files `A.h` und `B.h` mit `#include` einfügen. Damit würde die Klasse `alpha` zweimal vereinbart, was ein Fehler ist (Compilermeldung: Multiple declaration for ‘alpha’).

`alpha.h:`

```
class alpha
{
 . . . .
}
```

`A.h`

```
#include “alpha.h“
. . . .
```

`B.h`

```
#include “alpha.h“
. . . .
```

`beispiel.cpp`

```
#include “A.h“
#include “B.h“
```

Abhilfe schafft die folgende Vorgehensweise. Man formuliert den Include-File `alpha.h` mit Hilfe der Präprozessor-Anweisung `#ifndef:`

`alpha.h`

```
#ifndef alpha_h
#define alpha_h

class alpha {
 . . .
```

```
}

#endif
```

Der Präprozessor bearbeitet die Zeilen von `#define` bis `#endif` nur, wenn die Konstante `alpha_h` noch nicht definiert worden ist. Damit wird eine mehrmalige Deklaration einer Klasse unterbunden.

5 Initialisierung von Instanzen durch Konstruktoren

Die Methode, stets eine eigene Initialisierungsroutine zu schreiben, ist syntaktisch korrekt, aber nicht elegant und fehleranfällig. Das Problem ist, daß in der Praxis oft vergessen wird, die Initialisierungsroutine aufzurufen.

C++ bietet hier die Möglichkeit, eine Initialisierungsroutine automatisch bei der Definition des Objektes ausführen zu lassen. Dazu gibt es in C++ standardmäßige Initialisierungsfunktionen, die **Konstruktoren**. Ist ein solcher Konstruktor vorhanden, so wird dieser sofort nach der Allokation des Speicherplatzes aufgerufen und initialisiert das Objekt. Das bedeutet, daß mit der Definition eines Objektes es vollautomatisch gleichsam zwangsweise initialisiert wird. Hierzu ist es lediglich erforderlich, daß der **Name der Initialisierungsroutine gleich dem Namen der Klasse** ist.

```
class Bruch3 {
// diese Klasse dient zur Darstellung eines Bruches

     int zaehler;
     int nenner;
public:
     Bruch3();
     void print ();
};


#include <stdio.h>

void Bruch3::print() {
      printf ("\n der Wert des Quotienten von %i und %i ist %i/%i",
                zaehler, nenner,zaehler, nenner);
}

Bruch3::Bruch3() {
      zaehler = 0;
      nenner = 1;
}

void main (void)
{
 Bruch3 b;       // automatische Initialisierung
 b.print();
}
```

Die Implementierung der Initialisierungsroutine bleibt also gleich, nur heißt sie nicht mehr `init`, sondern wie die Klasse selbst. Die Anweisung

```
Bruch3 b;
```

bewirkt **nicht nur die Reservierung von Speicher für das Objekt b, sondern gleichzeitig auch den Aufruf der Initialisierungsroutine `Bruch3::Bruch3()`.**

Verfolgen Sie den automatischen Aufruf der Initialisierungsroutine, indem Sie das Programm `Bruch3` im Debugger zeilenweise durchsteppen.

Ein Konstruktor unterscheidet sich von einer normalen Mitgliedsfunktion unter anderem dadurch, daß der **Konstruktor ohne Rückgabeparameter (auch nicht `void`)** deklariert wird.

Ein Konstruktor unterscheidet sich von einer normalen Mitgliedsfunktion auch dadurch, daß er **nicht an eine abgeleitete Klasse vererbt** wird.

Der Compiler erkennt einen **Konstruktor** daran, daß er den **gleichen Namen** trägt **wie die Klasse selbst.**

Bis auf Ausnahmefälle sind **Konstruktoren im allgemeinen `public` definiert**, da sonst keine Objekte der Klasse erzeugt werden können.

Wie Sie bereits wissen, erfolgt die Aktivierung einer Methode, indem für das entsprechende Objekt auf die Memberfunktion zugegriffen wird. Die Aktivierung kann also nur erfolgen, wenn die Methode als `public` zur Verfügung steht. Soll die Methode des Konstruktors aktiviert werden können, muß sie `public` sein.

5.1 Konstruktoren mit Parametern

Wie alle C++-Funktionen können Konstruktoren mit Parametern versehen werden. Auch der Funktionsrumpf ist wie bei einer normalen Funktion aufgebaut. Bei der Erzeugung eines Objekts müssen die Parameter dann angegeben werden.

Beispiel:

```
class Bruch4 {
// diese Klasse dient zur Darstellung eines Bruches

     int zaehler;
     int nenner;
public:

     Bruch4(int, int);
     void print ();
}

#include <stdio.h>
void Bruch4::print() {
     printf ("\n der Wert des Quotienten von %i und %i ist %i/%i",
               zaehler, nenner, zaehler, nenner);
```

```
}


Bruch4::Bruch4(int n_zaehler, int n_nenner) {
     zaehler = n_zaehler;
     nenner = n_nenner;
}

void main (void)
{
 Bruch4 b (7, 3);
 b.print();
}
```

Überzeugen Sie sich, daß jetzt eine Definition

```
Bruch4 d;
```

einen Fehler erzeugt.

Für Konstruktoraufrufe gelten die folgenden Regeln:

Konstruktoren werden zu Beginn der Lebensdauer eines Objektes automatisch aufgerufen. Im einzelnen gilt:

- Konstruktoren für globale Objekte werden vor der Funktion `main` aufgerufen
- Konstruktoren für lokale Objekte einer Funktion werden beim Aufruf dieser Funktion an der Stelle der Definition der lokalen Objekte aufgerufen
- Konstruktoren für dynamische Objekte werden bei der Reservierung des Speicherplatzes durch `new` aufgerufen (siehe später)
- Konstruktoren von Basisklassen werden vor den Konstruktoren ihrer Nachkommen aufgerufen (siehe später)

Nachdem für das Objekt der notwendige Speicherplatz bereit gestellt wurde, wird der Konstruktor ausgeführt. Der Konstruktor dient in der Regel zum Initialisieren des Objektes. Der vom Compiler standardmäßig zur Verfügung gestellte Default-Konstruktor „tut nichts“.

Auf den letzten Punkt werden wir im Kapitel "Ahnengalerie" noch näher eingehen.

Beachten Sie, daß es auch möglich ist, Konstruktoren in Ausdrücken direkt zu verwenden. Es wird dann kein Objekt mit einem Namen angelegt, sondern ein Objekt

ohne Namen, welches nur innerhalb des Ausdrucks, in dem es steht, verwendet werden kann.

Beispiel:

```
ausgabe (Bruch4 (1,8));
```

Hier wird ein Objekt der Klasse `Bruch4` mit einem Zählerwert von 1 und Nennerwert von 8 an die Funktion `ausgabe` übergeben. Nach dem Aufruf der Funktion `ausgabe` wird das Objekt wieder vernichtet.

5.1.1 Variablendefinition an beliebiger Stelle des Quelltextes

Es folgt nun die in Kap. 2.2 vorenthaltene Erklärung, warum in C++ Definitionen zwischen Anweisungen möglich sind.

Nachdem Sie nun das Konzept eines Konstruktors mit Parametern kennengelernt haben ist Ihnen klar, daß man eine Variable nicht zu Programmbeginn definieren und mit Hilfe des Konstruktoraufrufs mit bestimmten Werten belegen kann, wenn die Initialisierungswerte erst ausgerechnet werden müssen. Man müßte also ein Objekt zu Beginn mit „falschen“ Werten belegen und im Programmverlauf die „falschen“ Werte mit den „richtigen“ Werten überschreiben. Dies ist der Grund dafür, warum es möglich ist, an beliebiger Stelle des Quelltextes Variablen zu definieren.

Aus Gründen des Software Engineerings ist davon im allgemeinen abzuraten. Wir führen in der Regel neue Variablen nur dann zwischen den Anweisungen ein, wenn Variablen mit `new` dynamisch erzeugt werden sollen. Bei großen Objekten können Performance-Zwänge dazu führen, daß Ausnahmen gemacht werden müssen.

5.1.2 Wichtige Arten von Konstruktoren

Außer selbst geschriebenen Konstruktoren mit oder ohne Parametern sind in C++

?? Standard-Konstruktoren
?? Kopier-Konstruktoren
?? und Konvertier-Konstruktoren

von Bedeutung.

Diese sollen im folgenden vorgestellt werden.

5.2 Standard-Konstruktor

Der **Standard-Konstruktor** oder **Default-Konstruktor** wird vom Compiler zur Verfügung gestellt. Er ist für jede Klasse automatisch definiert (wird vom Compiler zur Verfügung gestellt), vorausgesetzt, es wird kein Konstruktor selbst definiert. Ein

Standard-Konstruktor einer Klasse benötigt keine Argumente. So war zum Beispiel im Programm `bruch.cpp` ein eigener Konstruktor nicht definiert. Mit

```
Bruch b;
```

was wie eine „normale“ Definition aussieht, wurde der Standardkonstruktor

```
Bruch ()
```

aufgerufen. Der vom Compiler zur Verfügung gestellte Standard-Konstruktor initialisiert nicht.

Der Standard-Konstruktor ist für jede Klasse automatisch definiert, kann jedoch auch explizit definiert werden.

Es ist auch möglich, **selbst einen Standard-Konstruktor zu schreiben**. Dieser muß ohne Parameter aufrufbar sein.

Ein solcher Standard-Konstruktor ist beispielsweise:

```
Bruch::Bruch() {      // kein Übergabeparameter!
   zaehler = 0;
   nenner = 1;
}
```

oder etwa

```
Bruch::Bruch ( int n = 0, int m = 1)
                           // Vorgabewerte für alle Parameter!
   zaehler = n;
   nenner  = m;
}
```

In beiden Fällen erfolgt der Aufruf durch Bruch:

```
Bruch a;
Bruch b;
```

Ein Standard-Konstruktor kann selbst explizit auf zwei verschiedene Weisen definiert werden:

- ohne Übergabeparameter
- mit Default-Werten für alle Parameter,

Der Aufruf

```
Bruch a();
```

würde eine Funktion a ohne Parameter, die ein Objekt vom Typ `Bruch` zurückgibt, deklarieren.

Blättern wir etwas zurück, so sehen wir, daß in Programm `Bruch3.cpp`

```
Bruch3::Bruch3() {
     zaehler = 0;
     nenner = 1;
}
```

ein Standard-Konstruktor war.

Ist ein Konstruktor mit Parametern definiert, so muß jedes Objekt über ihn mit konkreten Werten initialisiert werden, es sei denn, der Konstruktor hat Default-Parameter.

5.3 Kopier-Konstruktor

Ein Konstruktor darf niemals als **Wert**-Parameter (*call by value*) ein Objekt der eigenen Klasse haben. Da ein formaler Parameter ein (lokales) Datenobjekt ist, müßte für die Erstellung des Objekts wieder ein Konstruktoraufruf erfolgen. Dies würde ein rekursives Aufrufen bedeuten, was zu einer unendlichen Rekursion führen würde.

Eine Möglichkeit, einen Konstruktor mit einem Parameter der eigenen Klasse zu erstellen, ist, den Parameter als Referenzparameter zu vereinbaren.

Ein Sonderfall dieser Möglichkeit ist der **Kopierkonstruktor**. Er hat eine Referenz auf ein Objekt der eigenen Klasse als einzigen Parameter. Er dient dazu, ein Objekt einer Klasse zu definieren und mit dem Inhalt eines anderen vorhandenen Objektes derselben Klasse zu initialisieren.

Ein Kopierkonstruktor nimmt als einzigen Parameter eine Referenz auf eine Instanz seiner Klasse entgegen.

Beispiel:

```
Bruch4 b (zaehlerwert, nennerwert);  // Konstruktor
Bruch4 c(b);                         // Kopierkonstruktor: c
                                     // wird mit Daten von b
                                     //  initialisiert
```

Wie schon gesagt, muß ein solcher Kopier-Konstruktor als Parameter eine Referenz auf ein Objekt der Klasse enthalten. Hier die Deklaration eines Kopier-Konstruktors:

```
class Bruch4 {
        .....
public:
        Bruch4 (const Bruch4 & bruchobjekt); // Deklaration
                                             // Kopierkonstruktor
        ....                                 // Deklaration anderer
                                             // Mitgliedsfunktionen
}
```

In der Implementierung müssen die Daten des als Parameter übergebenen Objektes in das eigene Objekt kopiert werden:

```
Bruch4::Bruch4 (const Bruch4 & bruchobjekt) {
     zaehler = bruchobjekt.zaehler;
     nenner = bruchobjekt.nenner;
}
```

Das Schlüsselwort `const` stellt sicher, daß die Kopierfunktion das übergebene Objekt nicht ändern kann. Das Schlüsselwort `const` ist nicht zwingend notwendig, jedoch oft zweckmäßig, um das Referenzobjekt vor Veränderungen zu schützen.

Wird eine Funktion wie z.B.

```
void f (Bruch4 form_param)
```

aufgerufen mit

```
f(akt_param),
```

wobei `akt_param` ein Objekt der Klasse `Bruch4` ist, so wird hierfür der Kopierkonstruktor aufgerufen. Der Kopierkonstruktor kopiert den Wert des aktuellen Parameters `akt_param` in den lokalen Parameter `form_param`..

Der Standard-Kopier-Konstruktor ist für jede Klasse automatisch definiert, kann jedoch auch explizit selbst definiert werden.

Der Standard-Kopier-Konstruktor kopiert elementweise.

Der Standard-Kopier-Konstruktor für `Bruch4` entspricht genau dem oben explizit programmierten Kopier-Konstruktor. Für Klassen wie für `Bruch4` braucht kein Kopier-Konstruktor selbst definiert werden. Es gibt jedoch Fälle, z.B. bei Klassen mit dynamischem Speicher, deren Komplexität das Schreiben von Kopier-Konstruktoren erfordert.

Aufruf des Kopier-Konstruktors:

Es gibt zwei Möglichkeiten, einen Kopierkonstruktor aufzurufen:

```
Bruch4  c (b); // c wird mit den Daten von b initialisiert;
Bruch4  c = b; // c wird mit den Daten von b initialisiert;
```

Beide Schreibweisen sind gleichwertig. Beide bewirken, daß die Funktion des Kopierkonstruktors aufgerufen wird.

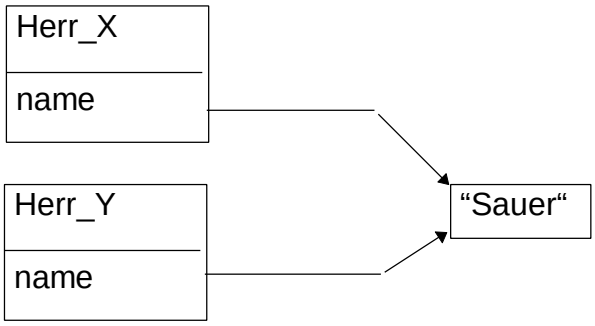
Bei dynamischen Variablen kann es möglicherweise zu Problemen mit dem Kopierkonstruktor kommen.

Beispiel:

```
#include <string.h>
#include <stdio.h>
//Datei: Sauer.cpp

class Person
{
    char * name;
 public:
    Person (char * neuname)
    {
     name = new  char [strlen (neuname)+1];
     strcpy (name,neuname);
    }

    void print (void)
    {
     printf ("\nDer Name ist:%s", name);
    }
```

// bitte beachten Sie den Destruktor jetzt noch nicht, er wird erst später erklärt

```
    ~Person (void)         //dies ist der Destruktor,
    {                      //der "aufräumt"
     delete [] name;
    }
};

void main (void)
{
 printf ("\n");
 Person Herr_X ("Sauer");
 Herr_X.print();
 {
  Person Herr_Y = Herr_X;
  Herr_X.print();
  Herr_Y.print();
 }// Destruktor Herr_Y löscht den Sauer weg.
  Herr_X.print();
}
```



5.4 Konvertierkonstruktor

Ein Konvertier-Konstruktor dient zur Typkonvertierung eines Standard-Typen oder eines selbst definierten Typs zum Typ der Klasse.

Ein Konvertier-Konstruktor ist ein Konstruktor mit einem einzigen Parameter, welcher aber im Gegensatz zum Kopier-Konstruktor von einem anderen Typ ist.

Beispiel:

```
class Bruch{
   int zaehler;
   int nenner;
public:
   Bruch (int par) {  // Konvertierkonstruktor von Standardtyp int
     zaehler = par;
     nenner  = 1;
   };
};
```

Hier wird ein Konvertier-Konstruktor definiert, der den Standard-Typ **int** in den Typ der Klasse **Bruch** wandelt.

Der Konvertierkonstruktor - wenn definiert - wird bei Standardtypen auch implizit, d.h. vom Compiler, angestoßen. D.h. bei der Zuweisung

```
void main (void) {

   Bruch b(3);   // expliziter Aufruf

   b = 4;        // impliziter Aufruf
}
```

wird vom Compiler explizit der Konvertier-Konstruktor b = Bruch(4) benutzt.

Ebenso wird der Konvertier-Konstruktor auch benutzt, wenn man die Konvertierung explizit aufruft:

```
b = (Bruch)4;    // Type-Casting
```

oder

```
b = Bruch( 4 );  // funktioneller Cast
```

5.4.1 Konvertierung von einer Klasse in eine andere

Als abschließendes Beispiel noch eine Konvertierung von einer Klasse in die andere:

```
#include <stdio.h>
#include <string.h>

class Person{
   char name[20];
   char vorname[20];
   int  alter;
public:
   Person ( char * n, char * v, int a ) {
      strcpy( name, n);
      strcpy( vorname, v);
      alter = a;
   };
   char * get_name(void)    { return name; };
   char * get_vorname(void) { return vorname; };
   int    get_alter(void)   { return alter; };
};

class Personal {
   char name[20];
   char vorname[20];
   int  alter;
   int  gehalt;
public:
   Personal ( Person & p) {
      printf("Konvertier-Konstruktor von der Klasse Person ");
      printf("in die Klasse Personal\n");
      strcpy( name,    p.get_name() );
      strcpy( vorname, p.get_vorname() );
      alter  = p.get_alter();
      gehalt = 0;
   };
   void print() {
     printf("%s\t%s\nAlter: %3d\tGehalt: %7d DM\n",
            name, vorname, alter, gehalt );
   };
};

void main( void ) {
   Person p1( "Müller", "Fritz", 31 );
   Personal ps1 = p1;
       // Aufruf des Konvertier-Konstruktors von Personal
```

```
      // Die Gehaltsverhandlungen sind aber, wie man sieht,
      // noch gesondert zu führen
   ps1.print();
}
```

Die Ausgabe ist:

```
Konvertier-Konstruktor von der Klasse Person in die Klasse Personal
Müller  Fritz
Alter:  31      Gehalt:       0 DM
```

5.4.2 Vorwärtsdeklaration

Wenn eine Referenz auf ein Objekt einer Klasse angegeben wird, dann muß die referenzierte Klasse deklariert sein. Wird sie erst später definiert, so resultiert ein Compilerfehler, es sei denn, es wird eine Vorwärtsdeklaration - wie in folgendem Beispiel - verwendet:

```
#include <stdio.h>
#include <string.h>

class Person;  // Vorwärtsdeklaration

class Personal {
   char name[20];
   char vorname[20];
   int  alter;
   int  gehalt;
public:
   Personal ( Person & p);
   void print() {
     printf("%s\t%s\nAlter: %3d\tGehalt: %7d DM\n",
         name, vorname, alter, gehalt );
   };
};


class Person{
   char name[20];
   char vorname[20];
   int  alter;
public:
   Person ( char * n, char * v, int a  ) {
      strcpy( name, n);
      strcpy( vorname, v);
      alter = a;
   };
   char * get_name(void)    { return name; };
   char * get_vorname(void) { return vorname; };
   int    get_alter(void)   { return alter; };
```

```
};


Personal ::Personal ( Person &  p)
{
      printf("Konvertier-Konstruktor von der Klasse Person ");
      printf("in die Klasse Personal\n");
      strcpy( name,    p.get_name() );
      strcpy( vorname, p.get_vorname() );
      alter  = p.get_alter();
      gehalt = 0;
};


void main( void ) {
. . . . .
    // genau wie im letzten Kapitel
. . . . .
}
```

Für das Compilieren von

```
Personal ( Person & p);
```

muß dem Compiler der Typname `Person` bekannt sein. Dies erfolgt durch die Vorwärtsdeklaration. Mehr muß nicht bekannt sein. Wäre der Konvertierkonstruktor von `Personal` jedoch in der Klasse `Personal` definiert worden, dann hätte die Vorwärtsdeklaration überhaupt nichts genützt, da dann auf die bis dahin noch unbekannten Memberfunktionen von p zugegriffen worden wäre.

Die Vorwärtsdeklaration funktioniert natürlich nicht nur bei einem Referenzaufruf, sondern auch bei einem call by value, entscheidend ist, daß dem Compiler der Typname bekannt ist. Aber solange der Typ nicht definiert ist, darf an keiner Stelle auf ein Memberdatum oder eine Memberfunktion zugegriffen werden, noch darf es erforderlich sein, daß der Compiler ein Objekt von diesem Typ anlegt..

5.5 Vom Compiler automatisch zur Verfügung gestellte Konstruktoren und Default-Operatoren

Der Compiler stellt automatisch zur Verfügung:

- einen **Default-Zuweisungsoperator für jede Klasse**. Damit kann ein Objekt an ein Objekt derselben Klasse zugewiesen werden. Das Kopieren erfolgt wie bei Strukturen komponentenweise.
- einen **Default-Konstruktor** (ohne Parameter)
- einen **Default-Destruktor**

?? einen **Default-Kopier-Konstruktor**, der komponentenweise kopiert

Existieren eigene Ausprägungen des Zuweisungsoperators bzw. des Destruktors und dieser Konstruktoren, so können die vom Compiler zur Verfügung gestellten Default-Mechanismen nicht verwendet werden.

5.6 Konstruktoren und dynamische Objekte

Beim dynamischen Anlegen von Variablen mit new wird für Objekte der Konstruktor aufgerufen. Ist ein Konstruktor nicht selbst definiert, wird der Standard-Konstruktor des Compilers benutzt.

5.6.1 new für Objekte

Das folgende Beispiel zeigt die Funktion new einmal für eine dynamische int-Variable und einmal für ein Objekt einer Klasse.

```
// Datei: demonew.cpp

#include <stdio.h>

class demonew
{
   float x;
 public:
   demonew (float);
   void print ();
};

demonew :: demonew (float p1) {
   x = 10* p1;
}

void demonew :: print () {
   printf ("\nWert von demonew: %6.2f",x);
}

void main (void) {
  printf ("\n\n\n");
  int * ptrint  = new int (10);
  printf ("* ptrint hat den Wert %d", *ptrint);
  demonew * ptr =  new demonew (10);
  ptr->print();
  delete ptrint;
  delete ptr;
}
```

Beachten Sie, daß `new int (10)` die dynamisch geschaffene `int`-Variable mit 10 initialisiert. Der Aufruf `new demonew (10)` hingegen ruft den Konstruktor für `demonew` auf und übergibt den Parameter 10. Die Initialisierung erfolgt hier mit dem 10-fachen des übergebenen Wertes.

Hier die Ausgabe des Programmlaufs:

```
* ptrint hat den Wert 10
Wert von demonew: 100.00
```

Der Konstruktor wird aufgerufen, wenn ein Objekt mit `new` auf dem Heap erzeugt wird:

```
Bruch * bp1 = new Bruch (1,5);
```

Beachten Sie bitte, daß die Initialisierung nur bei Verwendung von `new` durchgeführt wird. Wird stattdessen `malloc` verwendet, bleibt das Objekt uninitialisiert:

```
// bei Verwendung von malloc kann das Objekt nicht
// initialisiert werden

Bruch *bp2 = (Bruch*)malloc (sizeof (Bruch));
```

Die Funktion `malloc` beschränkt sich auf die Anforderung eines Speicherbereichs der gewünschten Größe. Der Operator `new` dagegen erzeugt ein Objekt. Dazu gehört aber mehr als nur die Anforderung des benötigten Speichers, nämlich eben auch die Initialisierung des Speicherbereichs mit einem sinnvollen Bitmuster. Dazu kann der Benutzer den Konstruktor angeben.

Der Unterschied zwischen `new` und `malloc` ist symptomatisch zwischen „normaler“ Programmierung in C und objektorientierter Programmierung in C++. Während der Entwickler in C auf die Arbeit mit Speicherbereichen nicht verzichten kann, arbeitet er in C++ eher mit Objekten. Daß Objekte auch Speicher benötigen, ist ein Detail, das der Programmierer dem Compiler überlassen kann.

Steht kein Konstruktor mit Parametern zur Verfügung, so wird der Standard-Konstruktor der Klasse aufgerufen.

Beispiel:

```
Bruch * bp1 = new Bruch;
```

Bei einem Kopierkonstruktor wird die Kopie durch elementweises Kopieren des Originals erstellt. Dies kann bei dynamischen Datenstrukturen, bei denen die einzelnen Elemente einen oder mehrere Zeiger enthalten, zu Problemen führen,

wenn die lokale Kopie wieder freigegeben wird, und der Destruktor die Verkettung der Elemente ändert.

5.6.2 Dynamische Arrays

Die folgende Einschränkung gilt **für statische und für dynamische Arrays von Objekten**:

Eine jede Array-Komponente wird mit dem Standard-Konstruktor initialisiert. Eine Initialisierungsliste wie bei Arrays aus Komponenten der klassischen Datentypen von C (elementare wie auch zusammengesetzte Datentypen) ist nicht möglich. Ein Konstruktor mit Parametern ist ebenfalls nicht möglich. Ist der Standard-Konstruktor nicht selbst geschrieben, so wird der Standard-Konstruktor des Compilers genommen. Eine Initialisierung der Komponenten unterbleibt dann.

Dynamische Arrays

Hier ein Beispiel für ein dynamisches Array aus Objekten:

```
// Datei: feldnew.cpp

#include <stdio.h>

float initwert = 5;

class Arrayelement
{
   float x;
   float y;
 public:
   Arrayelement ();
   void print ();
}

Arrayelement :: Arrayelement () {
   x = initwert++;
   y = x+.5;
}

void Arrayelement :: print () {
   printf ("\nx-Komponente: %6.2f",x);
   printf ("\ny-Komponente: %6.2f",y);
}


void main (void) {
  int lv;
  printf ("\n\n\n");
  Arrayelement * ptr =  new Arrayelement [10];
  for (lv = 0; lv <= 9; lv++){
  printf ("\nDas %d-te Arrayelement hat die Komponenten", lv);
  ptr [lv].print();
  }
```

```
  delete [] ptr;
}
```

Die Ausgabe des Programmes ist:

```
Das 0-te Arrayelement hat die Kompone nten
x-Komponente:   5.00
y-Komponente:   5.50
Das 1-te Arrayelement hat die Komponenten
x-Komponente:   6.00
y-Komponente:   6.50
Das 2-te Arrayelement hat die Komponenten
x-Komponente:   7.00
y-Komponente:   7.50
Das 3-te Arrayelement hat die Komponenten
....
```

5.7 Overloading von Konstruktoren

Wie alle C++-Funktionen können auch Konstruktoren überladen werden, es können also für eine Klasse mehrere Konstruktoren vorhanden sein. Die einzelnen Funktionen müssen sich jedoch in Anzahl und/oder Typ der Parameter unterscheiden. Für unsere Klasse `Bruch` bietet sich ein zweiter Konstruktor an, der nur eine einzige Zahl übernimmt. Möchte der Programmierer ein `Bruch`-Objekt mit einer ganzen Zahl initialisieren, so spart er sich so das explizite Setzen des Nenners auf 1:

```
// Datei: Bruch5a.cpp

class Bruch5 {
// diese Klasse dient zur Darstellung eines Bruches
     int zaehler;
     int nenner;
public:
     // Konstruktor #1
     Bruch5(int, int);
     // Konstruktor #2
     Bruch5(int);
     void print ();
};


#include <stdio.h>

void Bruch5::print() {
      printf ("\nder Wert des Quotienten von %i und %i ist %i/%i",
                zaehler, nenner,zaehler, nenner);
}

 Bruch5::Bruch5(int n_zaehler, int n_nenner) {
      zaehler = n_zaehler;
      nenner = n_nenner;
```

```
}
 Bruch5::Bruch5(int n_zahl) {
      zaehler = n_zahl;
      nenner = 1;
}

void main (void)
{
 Bruch5 a(5);
 Bruch5 b (10,2);
 a.print();
 b.print ();
}

}
```

Denselben Effekt könnte man natürlich auch erreichen, indem man einen Konstruktor mit Default-Wert einführt.

5.8 Destruktoren

Sie sind das Gegenstück zu den Konstruktoren und können gewisse Arbeiten durchführen, wenn ein Objekt sein Leben beendet. Ein Objekt, das lokal in einem Block angelegt wurde, verschwindet am Programmende. Ein globales Objekt verschwindet am Programmende. In beiden Fällen wird sein Speicher automatisch an den Compiler zurückgegeben. Ein dynamisch angelegtes Objekt verschwindet ebenfalls am Programmende und sein Speicher wird dann ebenfalls automatisch an den Compiler zurückgegeben, es sei denn, man braucht das Objekt nicht mehr und möchte den vom Objekt belegten Speicher vorzeitig mit `delete` an den Heap zurückgeben.

Der Compiler stellt einen Standard-Destruktor bereit, der beim Löschen eines Objektes automatisch aufgerufen wird. In speziellen Fällen sollte jedoch beim Löschen eines Objektes noch etwas getan werden, z.B. wenn ein Zähler für die Anzahl der lebenden Objekte existiert. Dann muß beim Tod eines Objekts dieser Zähler verringert werden. Bei dynamischen Datenstrukturen kann es ebenfalls notwendig sein, daß ein Destruktor selbst geschrieben werden muß, z.B. wenn die Verkettung der einzelnen Datenelemente einer verketteten Liste geändert werden muß, da ein Element aus der Liste herausgenommen wird.

Genau wie ein Konstruktor besitzt ein Destruktor keinen Rückgabewert. Außerdem kann er keine Parameter entgegennehmen.

Der Name des Destruktors ist der Name der Klasse mit einem vorangestellten Tilde-Zeichen, z.B: ~ Bruch4()

Hier ein Beispiel aus Kap. 3.5:

Der Destruktor ~Person der Klasse Person:

```
~Person (void)              //dies ist der Destruktor,
    {                       //der "aufräumt"
     delete [] name;
    }
};
```

gibt den mit

```
name = new  char [strlen (neuname)+1];
```

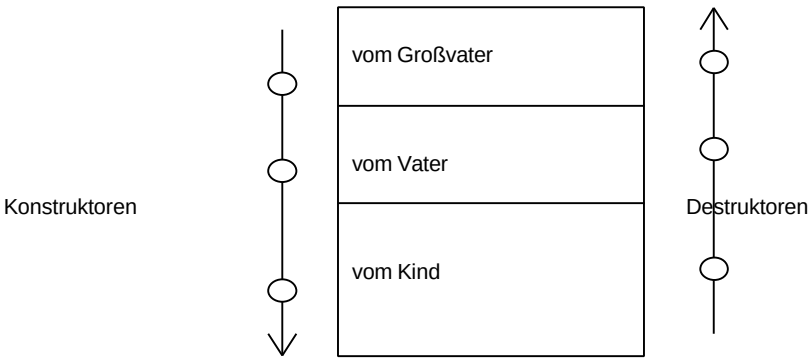
im Heap allokierten Speicherbereich an den Heap zurück.

Da ein Destruktor keine Parameter entgegennehmen kann, kann es für einen Objekttyp auch immer nur genau einen Destruktor geben. Dafür kann ein Destruktor als virtuelle Methode vereinbart werden. Ein Konstruktor kann das nicht.

Destruktoren werden immer dann automatisch aufgerufen, wenn der Gültigkeitsbereich eines Objekts verlassen wird:

- Destruktoren für globale Objekte werden beim Programmende aufgerufen
- Destruktoren für lokale Objekte werden beim Verlassen des Blocks aufgerufen
- Destruktoren für dynamische Objekte werden bei der Freigabe durch `delete` aufgerufen
- Destruktoren von Basisklassen werden nach denen ihrer Nachkommen aufgerufen

Beachten Sie vor allem den letzten Punkt. Während beim Anlegen die Konstruktoren der Ahnen zuerst aufgerufen werden, um den Erbteil zu generieren und zu initialisieren, erfolgt die Freigabe durch die Destruktoren in genau umgekehrter Reihenfolge. Das folgende Bild illustriert die Aufrufreihenfolge für ein Objekt an der Basis einer dreistufigen Hierarchie.

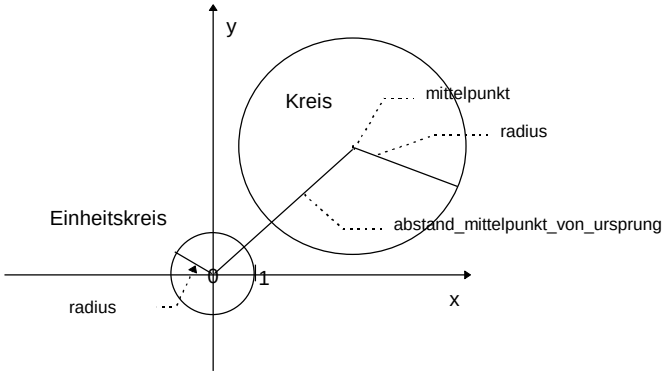


Der Destruktor ist das Gegenstück zum Konstruktor, daher muß es zu jedem Konstruktoraufruf auch einen Destruktoraufruf geben.

Beispiel:

Hier ein Beispiel aus der Geometrie.

Gegeben seien der Einheitskreis mit Mittelpunkt im Ursprung des Koordinatensystems und ein zweiter Kreis. Die Frage ist, ob sich beide Kreise überlappen. Dazu wird der Abstand des Mittelpunktes des zweiten Kreises vom Ursprung berechnet und verglichen, ob dieser Abstand kleiner gleich der Summe der beiden Kreisradien ist oder nicht.




```
// Datei: Kreise.cpp

#include <math.h>
#include <stdio.h>

class Kreis {
        struct
         {float x;
          float y;
         } mittelpunkt;
public:
        float radius;
        Kreis ();
        Kreis (float , float , flo at );
        double abstand_mittelpunkt_von_ursprung ()
        {
          return  sqrt (double (mittelpunkt.x * mittelpunkt.x +
                                mittelpunkt.y * mittelpunkt.y));
        }
}


Kreis::Kreis() {                                    //Standardkonstruktor
      printf ("\n\n\nGib die x-Koordinate für den Kreismittelpunkt ein: ");
      scanf ("%f", &mittelpunkt.x);
      printf ("Gib die y-Koordinate für den Kreismittelpunkt ein: ");
      scanf ("%f", &mittelpunkt.y);
```

```
      printf ("Gib einen Wert für den Radius ein: ");
      scanf ("%f", &radius);
}

Kreis::Kreis(float x, float y, float r) { //Konstruktor mit Parametern
       mittelpunkt.x = x;
       mittelpunkt.y = y;
       radius = r;
}


int  ueberlapp (Kreis & k) {  //Aufruf Kopierkonstruktor für Parameter k
       if (k.abstand_mittelpunkt_von_ursprung() < double (1 + k.radius) )
       return 1;  // ueberlapp prüft, ob übergebener Kreis Überlapp mit Einheitskreis hat
       else return 0;
}                              // Aufruf Destruktor für Parameter k


Kreis globkreis (5,0,4); //Aufruf Konstruktor mit Parametern für globales Objekt

void main (void)
{
 Kreis lokalkreis;      /Aufruf des Standardkonstruktors für lokales Objekt
 if  (ueberlapp(lokalkreis))  printf  ("\nÜberlapp  mit  lokalem  Kreis");
      //Aufruf Kopierkonstruktor
 else printf ("\nkein Überlapp mit lokalem Kreis");
 if  (ueberlapp(globkreis))  printf  ("\nÜberlapp  mit  globalem  Kreis");
      //Aufruf Kopierkonstruktor
 else printf ("\nkein Überlapp mit globalem Kreis");
}     // Aufruf Standard-Destruktor lokale Variable bei Funktionsende
      //Aufruf Standard-Destruktor globale Variable bei Programmende
```

5.9 Übersicht über die Eigenschaften von Konstruktoren und Destruktoren

Übersicht

Konstruktor	**Destruktor**
Initialisierung von Objekten	„Aufräumen“
kein Rückgabewert	kein Rückgabewert
beliebige Parameter	keine Übergabeparameter
Overloading möglich	kein Overloading
Name = Klassenname	Name = ?Klassenname

Beispiel für Aufruf:

```
Bruch b (1,2);
Bruch * pb;
pb = new Bruch (1,5);
```

Beispiel für Aufruf:

```
delete pb;
```

lokale `auto`-Objekte werden am Ende des Blockes, in dem sie gültig sind, automatisch durch Aufruf des Destruktors zerstört. Dynamische Objekte müssen explizit durch delete freigegeben werden. In beiden Fällen wird der Destruktor aufgerufen

6 Besondere Klassenelemente

6.1 Statische Klassenelemente

Wie Ihnen bereits bekannt ist, besteht eine Klasse aus Daten und Funktionen. Die Daten einer Klasse werden vom Compiler für jedes Objekt dieser Klasse angelegt. Die Funktionen werden nur einmal als Maschinencode angelegt und können für die Verarbeitung der Daten eines jeden Objektes aufgerufen werden. Außer diesen normalen Datenelementen und Memberfunktionen gibt es Klassenvariablen und Klassenfunktionen, die in C++ durch statische Variablen und statische Funktionen implementiert werden.

6.1.1 Statische Variablen

Wie Sie ebenfalls schon längst wissen, kann man Klassenvariable definieren, die für jedes Objekt dieser Klasse als globale Daten zur Verfügung stehen, indem man das Schlüsselwort static verwendet.

Beispiel:

```
// Beispiel für statische Klassenelemente: stat1.cpp

#include <stdio.h>

class example
{
    int Nummer_des_Schuelers;
    static int Klassenstaerke; //ist nur Deklaration
 public:
    example();
    void Abzaehlen (void);
};

example::example()
{
  Nummer_des_Schuelers = ++Klassenstaerke;
}

void example::Abzaehlen (void)
{
  printf("\nIch bin die Nummer %d", Nummer_des_Schuelers);
}

int example::Klassenstaerke = 0; // Definition und Initialisierung.
// Ein stat. Member wird extern definiert. Andere Zugriffe
// außer der Initialisierung erfolgen wie bei Klassenmembern
// Die stat. Variable Klassenstaerke ist zugreifbar für Memberfunktionen
```

```
void main(void)
{
  int lv;
  example Schueler [10];
  for (lv = 0; lv <=9; lv++)
  Schueler[lv].Abzaehlen();
}
```

Statische Member sind nicht Teil von Objekten der Klasse, sondern eigenständige Speicherobjekte. Die Initialisierung von statischen Elementen ist ohne die Existenz eines Klassenobjektes möglich.

Wie Sie in diesem Beispiel gesehen haben, ist es also nicht erlaubt, eine statische Variable in der Klassendefinition zu initialisieren. Die Initialisierung hat extern zu erfolgen.

Die Ausgabe des Programmes ist:

```
Ich bin die Nummer 1
Ich bin die Nummer 2
Ich bin die Nummer 3
Ich bin die Nummer 4
Ich bin die Nummer 5
Ich bin die Nummer 6
Ich bin die Nummer 7
Ich bin die Nummer 8
Ich bin die Nummer 9
Ich bin die Nummer 10
```

6.1.2 Statische Funktionen

Erweitern wir nun das vorliegende Programm aus Kap. 3.10.1. Nun soll jeder Schüler noch das Ergebnis des Abzählens bekannt geben. Das bedeutet, daß aus jedem Objekt auf die Klassenvariable zugegriffen werden soll.

```
// Datei: stat2.cpp
// Beispiel für statische Klassenelemente: stat.cpp
//
#include <stdio.h>

class example
{
    int Nummer_des_Schuelers;
    static int Klassenstaerke;
public:
    example();
    void Abzaehlen (void);
    static void Ergebnis (void);
    static void init () {
    Klassenstaerke = 0;
    }
};
```

```
example::example()
{
  Nummer_des_Schuelers = ++Klassenstaerke;
}

void example::Abzaehlen (void)
{
  printf("\nIch bin die Nummer %d", Nummer_des_Schuelers);
}

void example::Ergebnis(void)
{
  printf("\nUnsere Klasse hat %d Schüler", Klassenstaerke);
}
```

int example::Klassenstaerke = 0; // Möglichkeit 1 zur Initialisierung

```
void main(void)
{
  int lv;
  example::Ergebnis(); // Aufruf der static -Funktion, ohne daß ein
                       // Objekt vorhanden ist.
  example Schueler [10];

  for (lv = 0; lv <=9; lv++)
  Schueler [lv].Abzaehlen();
  for (lv = 0; lv <=9; lv++)
  Schueler [lv].Ergebnis();
```

example::init(); // Möglichkeit 2 zur Initialisierung
Schueler [1].init(); // Möglichkeit 3 zur Initialisierung

```
}
```

Die Ausgabe des Programmes ist:

```
Unsere Klasse hat 0 Schüler
Ich bin die Nummer 1
Ich bin die Nummer 2
      . . . . .
Ich bin die Nummer 10
Unsere Klasse hat 10 Schüler
Unsere Klasse hat 10 Schüler
      . . . . .
Unsere Klasse hat 10 Schüler
```

Sie sehen am Beispiel der Initialisierung, daß eine statische Funktion aufgerufen werden kann:

?? durch Angabe des Gültigkeitbereichs der Klasse. *Beispiel:*

```
  example::init();       // Möglichkeit 2 zur Initialisierung
```

durch Zugriff über ein Objekt. *Beispiel:*

```
schueler [1].init();     // Möglichkeit 3 zur Initialisierung
```

Eine nicht statische Funktion wird immer durch Zugriff auf ein Objekt aufgerufen. Diese Funktion erhält dann über den `this`-Zeiger Zugriff auf die Komponenten des Objekts.

Eine statische Funktion hat keinen `this`-Zeiger. Sie kann nur auf die statischen Elemente der Klasse zugreifen. Will man auf nichtstatischen Elementen mit einer statischen Funktion arbeiten, so muß ein Objekt dieser Klasse an die statische Funktion über die Parameterliste übergeben werden.

Auf eine statische Variable kann eine Member-Funktion, eine Freund-Funktion und eine statische Funktion zugreifen. Man braucht dazu kein Objekt der Klasse zu haben.

Zusammenfassung

Auch `static` weicht innerhalb einer Klassendeklaration von seiner üblichen Bedeutung ab. Ein als `static` deklariertes Datenelement wird nur einmal angelegt - egal, wieviele Instanzen der Klasse später definiert werden - ist aber dennoch in jedem Objekt verfügbar. Es gehört also nicht zu einem speziellen Objekt, sondern zur ganzen Klasse. Demzufolge wird es nicht als `"Objekt.data"` oder `"Zeiger_auf_Objekt->data"`, sondern als `"Klasse::data"` angesprochen. Seine Initialisierung muß außerhalb der Klasse vorgenommen werden. Würde dies im Konstruktor geschehen, dann würde das Datenelement ja mit jeder neuen Instanz re-initialisiert. Auch Elementfunktionen können `static` deklariert werden. Sie gehören dann ebenfalls direkt zur Klasse. Da sie beim Aufruf keinen `this`-Pointer erhalten, können sie nur mit statischen Daten arbeiten.

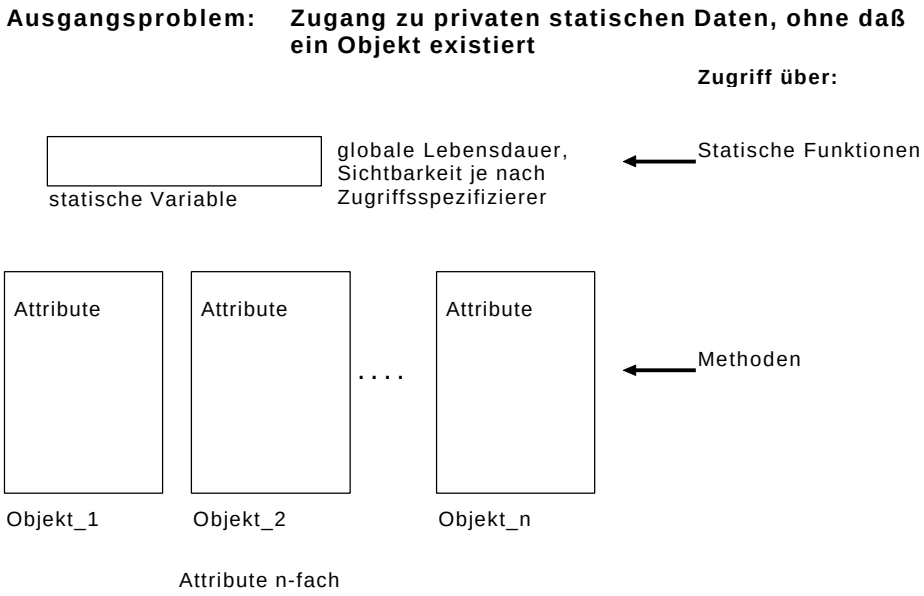


Bild 6 - 1 Statische Funktionen erlauben Zugriffe auf statische Variable, ohne daß ein Objekt vorliegt

6.2 Konstante Datenelemente in Objekten

Im folgenden werden konstante Datenelemente im Objekt behandelt. Konstante Memberfunktionen und konstante Objekte sind Themen des Fortgeschrittenenkurses.

Konstante Datenelemente

Ein Datenelement, welches (mit der üblichen Syntax) als `const` deklariert wurde, wird bei der Generierung des Objekts durch den Konstruktor initialisiert und ist im weiteren Verlauf wie eine normale deklarierte Konstante schreibgeschützt.

Die Initialisierung ist genau wie bei sonstigen Konstanten zwingend erforderlich. Dazu gibt es eine spezielle Syntax, die dem Aufruf des Vorgängerkonstruktors sehr ähnlich sieht und mit dieser nicht verwechselt werden sollte. Diese Initialisierung kann auch für nicht konstante Variablen angewandt werden (statt einer Zuweisung im Rumpf).

Beispiel:

```
class C {
  const int c_i1, c_i2;
  int i3, i4;

public:
  C(int init_int) : c_i1(10),        // So zwingend erforderlich
                    c_i2(init_int),   // (Konstanten)
                    i3(20)           // ist erlaubt (normales Attribut)
  {
    i4 = init_int+30;
  };

  void func(void) {
    i4 = 20;    // OK: i4 ist keine Konstante
    c_i1 = 30;  // Fehler: c_i1 ist eine Konstante
  };
};
```

Wie man sieht, muß der Initialisierungswert des konstanten Datenelementes nicht zur Kompilierzeit feststehen. Zur Initialisierung von konstanten Datenelementen können auch Parameter des Konstruktors verwendet werden (siehe Initialisierung von *c_i2*). Dadurch wird der Mechanismus wesentlich flexibler.

Zu beachten ist bei der Initialisierung mehrerer Konstanten das trennende Komma. Wie schon gesagt, muß die Initialisierung der Konstanten mit dem Konstruktor erfolgen. Der Wert, der einer Konstanten mit Hilfe des Konstruktors zugewiesen wird, kann selbst eine Konstante sein oder aber eine Variable der Parameterliste des Konstruktors.

Bei der Definition einer Instanz der Klasse C wird diese aus ihren Attributen aufgebaut. Dabei wird:

- das Attribut c_i1 angelegt und mit dem Wert 10 initialisiert. Das Attribut bleibt konstant.
- das Attribut c_i2 angelegt und mit dem übergebenen Wert init_int initialisiert. Das Attribut bleibt konstant.
- das Attribut i3 wird mit dem Wert 20 initialisiert
- das Attribut i4 wird zunächst mit undefiniertem Wert angelegt und dann im Rumpf des Konstruktors von C der Wert init_int + 30 zugewiesen.

Für Attribute, die keinen einfachen Datentyp (int, float,...), sondern einen selbstdefinierten Datentyp haben, entspricht die Initialisierung von c_i1, c_i2 bzw. von i3 dem Aufruf eines Konstruktors mit Parametern (1 Aufruf), von i4 dem Aufruf des Standard-Konstruktors mit nachfolgendem Aufruf des Zuweisungsoperators (2 Aufrufe). Machen Sie sich den Unterschied zwischen Initialisierung durch Konstruktor mit Parametern und Zuweisung nach Aufruf mit dem Standardkonstruktor klar und verwenden Sie in Zukunft den Konstruktor mit Parametern.

Beispiel:

```
#include <stdio.h>

class alpha
{
      int i;
      const int ci; //Konstante
 public:
      alpha (int i1, int i2);
      void print (void);
};

alpha::alpha(int i1, int i2) : ci (i2)
{
   i = i1;
};

void alpha::print (void)
{
 printf ("\n\ni: %d", i);
 printf ("\nci: %d", ci);
};

void main (void)
{
 alpha x (3,4);
 x.print();
}
```

Die Ausgabe des Programmes ist:

i: 3
ci: 4

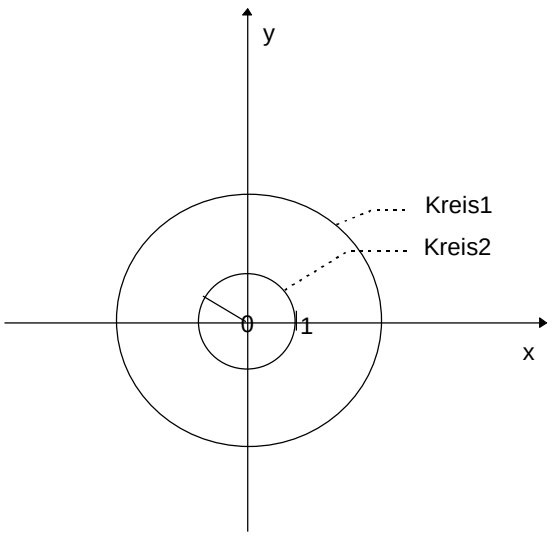
6.3 Objekte als Datenelemente einer Klasse

Wie wir aus der Programmiersprache C kennen ist es möglich, daß Komponenten einer Struktur selber Strukturen sind:

```
struct adresse {
   int hausnummer;
   char * strasse;
   int plz;
   char * stadt;
};


struct person {
   char * name;
   char * vorname;
   struct adresse adr;
};
```

In C++ können dementsprechend Objekte einer Klasse Datenelemente einer anderen Klasse sein. Damit fragen Sie zu recht, wie das denn mit den Konstruktoren und Destruktoren funktionieren soll. Das folgende Beispiel soll weiterhelfen. Dieses Beispiel ist wiederum der Anschaulichkeit wegen der Geometrie entnommen und basiert auf dem Programm `Kreise.cpp` aus Kap. 3.6.

Hier die Ausgangssituation:



Beachten Sie bitte, daß das gesuchte Ergebnis 100x einfacher konventionell berechnet werden kann. Das Beispiel hier wurde gewählt, um zu sehen, wie man mit Objekten in einer Klasse umgeht. Als Ergebnis soll u.a. die Fläche des Rings, der durch die Kreise Kreis1 und Kreis2 begrenzt ist, ausgegeben werden.

Hier das Programm:

```
// Datei: Kreise2.cpp

#include <math.h>
#include <stdio.h>

class Kreis {
     struct{
         float x;
         float y;
      } mittelpunkt;
public:
     float radius;
     Kreis (float , float , float );
     ~Kreis();
     double abstand_mittelpunkt_von_ursprung ()
     {
       return  sqrt (double (mittelpunkt.x * mittelpunkt.x +
               mittelpunkt.y * mittelpunkt.y));
     }
};

class Doppelkreis_um_Ursprung {
     Kreis Kreis1;
     Kreis Kreis2;
     int Farbnummer;
public:
     float Ringflaeche () {
     const float pi = 3.1415;
     return fabs(pi * (Kreis1.radius * Kreis1.radius  -
                    Kreis2.radius * Kreis2.radius));
     };
     void print_farbe ();
     Doppelkreis_um_Ursprung (float, float, float, float,float,
     float, int);
     ~Doppelkreis_um_Ursprung ();
}

Doppelkreis_um_Ursprung::Doppelkreis_um_Ursprung
(float x1, float y1, float r1, float x2, float y2, float r2, int farbe):
        Kreis1 (x1,y1,r1),
        Kreis2 (x2,y2,r2) {
  Farbnummer = farbe;
  printf ("\nKonstruktoraufruf Doppelkreis");
};
```

```
Doppelkreis_um_Ursprung::~Doppelkreis_um_Ursprung () {
  printf ("\nDer Doppelkreis verabschiedet sich");
}

void Doppelkreis_um_Ursprung :: print_farbe () {
  printf ("\nDie Farbnummer ist %d", Farbnummer);
}

Kreis::Kreis(float x, float y, float r) {
       mittelpunkt.x = x;
       mittelpunkt.y = y;
       radius = r;
       printf ("\nKonstruktoraufruf Kreis");
}

Kreis::~Kreis() {
       printf ("\nDer Kreis sagt tschüß");
}

void main (void)
{                                        //Konstruktoraufruf
 Doppelkreis_um_Ursprung testobjekt (0.,0.,5.,0.,0.,4.,7);
 testobjekt.print_farbe ();
 printf ("\nDie Ringflaeche ist %6.2f", testobjekt.Ringflaeche());
}
```

Die Ausgabe des Programms ist:

```
Konstruktoraufruf Kreis
Konstruktoraufruf Kreis
Konstruktoraufruf Doppelkreis
Die Farbnummer ist 7
Die Ringflaeche ist  28.27
Der Doppelkreis verabschiedet sich
Der Kreis sagt tschüß
Der Kreis sagt tschüß
```

Beachten Sie den Konstruktor der Klasse Doppelkreis_um_Ursprung:

```
Doppelkreis_um_Ursprung::Doppelkreis_um_Ursprung (float x1, float y1,
  float r1, float x2, float y2, float r2, int farbe):
  Kreis1 (x1,y1,r1), Kreis2 (x2,y2,r2) {
  Farbnummer = farbe;
};
```

Der Konstruktor hat 7 Parameter, davon werden 6 weitergegeben: 3 an den Konstruktur der Klasse `Kreis` zum Anlegen des Objektes `Kreis1` und 3 an den Konstruktor der Klasse `Kreis` zum Anlegen des Objektes `Kreis2`. Sie wissen ja, daß `Kreis1` und `Kreis2` Elemente der Klasse `Doppelkreis_um_Ursprung` sind. Der übrige Parameter wird zur Initialisierung des Elements `Farbnummer` verwendet. Mit anderen Worten, beim Aufruf des Konstruktors der Klasse `Doppelkreis_um_Ursprung`:

```
// Konstruktoraufruf
Doppelkreis_um_Ursprung testobjekt (0.,0.,5.,0.,0.,4.,7);
```

wird zweimal der Konstruktor für die Klasse `Kreis` aufgerufen. Es wird dabei zuerst der Konstruktor für `Kreis1` und `Kreis2` aufgerufen, bevor die eigentliche Konstruktorfunktion der Klasse `Doppelkreis_um_Ursprung` abgearbeitet wird. Entsprechend wird beim Destruktoraufruf für `testobjekt` der Destruktor für `Kreis1` und `Kreis2` aufgerufen.

Beachten Sie folgendes:

```
class Doppelkreis_um_Ursprung { // Programm nicht lauffähig
  Kreis kreis1;                 // Klasse Doppelkreis_um_Ursprung
  Kreis kreis2;                 // jetzt vor Klasse Kreis
};

class Kreis {
  ...
};
```

In der Klasse Doppelkreis_um_Ursprung kennt der Compiler nicht die member der Objekte `kreis1` und `kreis2`. Da der Compiler aber beim Anlegen einer Klasse die member nach einer bestimmten Anordnung im Speicher ablegt (Alignment), muß der Compiler an dieser Stelle wissen wieviel Speicherplatz er reservieren muß. Da das nicht der Fall ist führt das Programm oben zu einer Compiler-Fehlermeldung. Eine Vorwärtsdeklaration hilft hier nicht. Siehe Kapitel mit Vorwärtsdeklaration.

6.4 Freundfunktionen

Unter einer **Freundfunktion** versteht man eine Funktion, die **keine Elementfunktion der betreffenden Klasse ist** (jedoch Elementfunktion einer anderen Klasse sein darf), und dennoch **über alle Zugriffsrechte einer Elementfunktion verfügt**, d.h. eine Freundfunktion kann auf private Datenelemente zugreifen. **Dazu muß diese Funktion innerhalb der Klasse als `friend`-Funktion bekannt gemacht werden**. Dadurch erhält sie alle Rechte, die auch eine Elementfunktion hat. Freund-Funktionen sind grundsätzlich immer `public`, d.h. das Schlüsselwort `public` muß nicht angegeben werden.

Es ist nicht sinnvoll, eine Freund-Funktion nur für eine einzige Klasse zu deklarieren. Will man sie nur auf einen Objekttyp anwenden, so sollte man sie besser zur Elementfunktion machen. Freund-Funktionen machen dann Sinn, wenn man sie auch außerhalb dieser einen Klasse braucht, z.B. für zwei Klassen oder auch als externe Funktion. Freundfunktionen können dann zum direkten Datenaustausch zwischen Objekten verwendet werden, oder um eine allgemeine Funktion auf Instanzen mehrerer Klassen anwenden zu können

Eine Freundfunktion ist in solchen Fällen kein unmittelbarer Bestandteil einer Klasse. Sie wird nicht in Verbindung mit einem Objekt aufgerufen. **Sie verfügt nicht über einen `this`-Zeiger auf ein Objekt. Man muß deshalb der Friend-Funktion das Objekt als Parameter übergeben**. Aufgerufen wird sie wie eine normale Funktion.

Es ist natürlich auch möglich, daß eine Klasse B eine Elementfunktion einer Klasse A zur Freundfunktion der Klasse B erklärt. Diese Funktion hat einen impliziten `this`-Zeiger auf Objekte der eigenen Klasse A, nicht aber auf Objekte der Klasse B.

Die Freundschaftserklärung erfolgt von innen nach außen. Sie muß innerhalb der Klassendeklaration erfolgen, indem man das Schlüsselwort `friend` zusammen mit einer Deklaration der Freundfunktion einfügt. Das heißt, die Klasse muß ihre Freunde erklären. Eine Funktion kann sich nicht selber zum Freund einer Klasse ernennen.

Es ist auch möglich, **eine ganze Klasse zum Freund einer anderen Klasse zu erklären**, indem man nach `friend` ihren Namen angibt. Damit werden sämtliche Elementfunktionen der anderen Klasse zu Freundfunktionen erklärt.

Beispiel:

```
//Datei: freund3.cpp

#include <stdio.h>

class alpha {
      friend void func (alpha &b);
      private:
          float zahl;
      public:
          void print () {
          printf (" %6.2f", zahl);
          };
          alpha ();           // Konstruktor;
};

alpha::alpha () {
  zahl = 10;
}



void func (alpha &b) {
     b.zahl = b.zahl + 100;
};

void main (void) {
      alpha a1;
      printf("\n\n\n a1.zahl hat den Wert:");
      a1.print();
      func (a1);
      printf("\n Nach Anwendung der Freund-Funktion func hat\
 a1.zahl den Wert:");
      a1.print();
}
```

Das Ergebnis des Programmlaufs ist:

```
 a1.zahl hat den Wert:  10.00
 Nach Anwendung der Freund-Funktion func hat a1.zahl den Wert: 110.00
```

7 Operatoren

Operatoren in C sind üblicherweise auf die verwendeten Standard-Datentypen fixiert, die Bestandteil der Sprache sind. In C++ hingegen hat der Programmierer die Möglichkeit, Operatoren für seine selbstdefinierten Datentypen (bzw. die Objekte seiner Datentypen) einzusetzen und deren Verwendung und Gültigkeitsbereich frei zu definieren.

Für jeden Operator muß festgelegt werden, welchen Typ von Datenobjekten er bearbeiten kann und welche Wirkung dabei erzielt wird.

Ein einfaches Beispiel ist der Operator +. Er ist so definiert, daß er die arithmetische Summe bildet. Die Bedeutung eines Operators in einem Ausdruck hängt dabei von den aktuellen Operanden ab.

Man bezeichnet dies als **Kontextabhängigkeit** oder auch als **Polymorphie**. Sie ist bei vielen Operatoren in C und C++ bereits automatisch vorhanden. Für die einzelnen Grunddatentypen ist die jeweilige Wirkung der Operatoren festgelegt.

An dieser Stelle möchten wir darauf hinweisen, daß alle schon seit Kernighan&Ritchie-C mit überladenen Operatoren arbeiten. Der + Operator z.B. ist für alle numerischen Datentypen von `short` bis long `double` definiert. Dennoch wird mit Sicherheit bei einer `float` Addition eine andere Routine aufgerufen, als bei Addition zweier `int` Werte, da die Repräsentation beider Datentypen ganz verschieden ist. Auch * findet sowohl für die Multiplikation als auch zum Dereferenzieren Verwendung (einmal binär und einmal unär). Neu in C++ ist lediglich, daß der Programmierer nun selbst noch zusätzliche Überladungen vornehmen kann.

Führt man neue Klassen, d.h. neue Datentypen ein, so ist es möglich, die Funktionalität eines Operators für Objekte dieser Klasse neu zu definieren. Es ist also tatsächlich möglich, vordefinierte Operatoren wie +,*, [] zu überladen, um sie für eigene (Objekt)typen einsetzen zu können.

7.1 Überladen von Operatoren

Für eine Klasse können also überladene Operatoren definiert werden, die nur im Zusammenhang mit Objekten dieser Klasse benutzt werden können. Es gelten jedoch einige Einschränkungen beim Überladen von Operatoren:

?? Es können nur die in C++ verfügbaren Operatorsymbole überladen werden. Es ist nicht möglich, neue Operatoren zu definieren.

? Die Bindungsrichtung (Assoziativität) der Operatoren kann nicht geändert werden.

?? Die Auswertungsreihenfolge (Prioritätsregelung) der Operatoren kann nicht verändert werden.

?? Die Wertigkeit der Operatoren kann nicht verändert werden (es kann ein binärer Operator nicht als ternärer Operator neu definiert werden).

?? Die Wirkung der Operatoren auf die Grunddatentypen kann nicht verändert werden.

Folgende Operatoren können nicht überladen werden:

```
.    .*   ::    ?:  sizeof         #
```

Überladen werden können:

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

Wenn Sie sich klarmachen, wieviel Aufwand hinter der richtigen Auswertung von Ausdrücken durch den Compiler steckt, und wie unwahrscheinlich groß die Möglichkeiten wären, die sich ohne diese Einschränkungen ergeben würden (und die der Compiler alle berücksichtigen und verwalten müßte), werden Sie ihre Notwendigkeit sicher einsehen. Im übrigen kommt man in der Regel sehr gut mit den vordefinierten Operatorsymbolen aus.

Die meisten Anwendungen für überladene Operatoren, die Ihnen spontan einfallen, liegen sicherlich im mathematischen Bereich. So wäre es z.B. ohne weiteres möglich, den Operator + so zu überladen, daß er benutzerdefinierte Objekte für komplexe Zahlen oder Matrizen korrekt addiert.

7.2 Definition von Operatoren

Das Überladen von Operatoren in C++ ist deshalb möglich, weil ein Ausdruck mit einem Operator prinzipiell zu einem Funktionsaufruf aufgelöst wird.

Damit wird das Überladen von Operatoren erreicht als Konsequenz der Überladung von Funktionen mit unterschiedlichen Typen von Parametern. In einem Ausdruck wird geprüft, welchen Typ die Operanden haben. Dann sucht der Compiler die entsprechende Operatorfunktion, die genau diese Kombination von Parametern erwartet.

Zur Definition von Operatoren gibt es das Schlüsselwort **`operator`**, dem das entsprechende Operatorsymbol folgen muß.

Es gibt zwei Möglichkeiten, eine neue Operatorfunktion zu implementieren:

?? Die **erste Möglichkeit** wäre als Elementfunktion der Klasse, für deren Objekte sie definiert wurde. Wie bei jeder Elementfunktion muß beim Aufruf ein Datenobjekt dieser Klasse angegeben werden, das innerhalb der Funktion über den `this`-Zeiger erreichbar ist.

Ein Ausdruck mit einem **binären Operator** wird in diesem Falle folgendermaßen aufgelöst:

> **<operand1> <operatorsymbol> <operand2>**
> wird zu
> **<operand1>. operator <operatorsymbol> (<operand2>)**

Beispiel:

```
op1 + op2
```
wird zu:
```
op1.operator+ (op2)
```

`operator` ist das Schlüsselwort für die Operatorfunktionen.

Im Quelltext steht der Operator `<operatorsymbol>` zwischen beiden Operanden. Der Compiler ruft dann für den ersten Operanden `<operand1>` die Funktion `operator <operatorsymbol>`, die für die entsprechende Klasse definiert sein muß, auf und übergibt den zweiten Operanden `<operand2>` als Argument.

Im Sinne der Objektorientierung kann das folgendermaßen interpretiert werden:

An das Objekt `<operand1>` wird die Botschaft `operator <operatorsymbol>` gesandt (Name der Methode) mit dem Botschaftsparameter `<operand2>`. In C++ wird die Methode - wie schon bekannt - nicht über eine Nachricht aktiviert, sondern durch den Funktionsaufruf mit der Punkt- bzw. Pfeil-Notation.

Damit kann der erste Operand eines solchen Operators nur ein Datenobjekt dieser Klasse sein! Klassen aus Bibliotheken können also nur mit der unten beschriebenen 2-ten Möglichkeit um Operatoren ergänzt werden.

Im Falle **unärer Operatoren** gilt analog:

Präfix: <operatorsymbol> <operand1>
wird zu
<operand>. operator <operatorsymbol> ()
Postfix:<operand1> <operatorsymbol>
wird zu
<operand>. operator <operatorsymbol> (int)

Hinweis: Die Erläuterung zu dem dummy-Parameter `(int)` ist bei Stroustrup [9] zu finden

?? Als **zweite Möglichkeit** kann die **Operatorfunktion** auch **außerhalb der Klasse** definiert werden. Sie wird dann in der Regel innerhalb der Klasse als **`friend`** deklariert, damit sie Zugriff auf die Daten der Klasse hat. Alle Operanden eines Ausdrucks werden dann als Argumente der Funktion übergeben:

<operand1> <operatorsymbol> <operand2>
wird zu
operator <operatorsymbol> (<operand1>, <operand2>)

Beispiel:
`op1 + op2` wird zu: `operator+ (op1, op2)`

Im Quelltext steht also **<operand1> <operatorsymbol> <operand2>,** der Compiler macht daraus **operator <operatorsymbol> (<operand1>, <operand2>)**und ruft selbst die neue Operatorfunktion `operator <operatorsymbol>` auf. Dieser Funktion `operator <operatorsymbol>` wird als erster Parameter der Operand `<operand1>` und als zweiter Parameter der Operand `<operand2>` übergeben. Es ist auch möglich, die Operatorfunktionen explizit aufzurufen. Sie werden auch sonst vom Compiler wie normale Funktionen behandelt und anhand der Typen ihrer Argumente unterschieden.

Im Falle **unärer Operatoren** gilt analog:

Präfix: <operatorsymbol> <operand1>
wird zu
operator <operatorsymbol> (<operand1>)
Postfix:<operand1> <operatorsymbol>
wird zu

operator <operatorsymbol> (<operand1>, int)

Beispiel für die erste Möglichkeit:

```
// Datei: op1.cpp
#include <stdio.h>

class obj_type {
   private:
      float summand;
   public:
      obj_type ( float zahl)          // Memberfunktion
        {                             //       +
         summand = zahl;              // Konstruktor
        };                            // Overloading  !!!
      obj_type () {};                 // Konstruktor

      void print () {
       printf ("%6.2f", summand);
      };

      obj_type operator + (obj_type p2) {
       return summand + p2.summand;
      };

      obj_type operator + (float p2) {
           return summand + p2;
      };
};

void main (void) {
    obj_type betrag1 (5.);
    obj_type betrag2 (10.);

    printf ("\n\n\nbetrag1 nach Initialisierung mit  5.: ");
    betrag1.print ();

    printf ("\nbetrag2 nach Initialisierung mit 10.: ");
    betrag2.print ();

    obj_type betrag3;         //betrag3 ist nicht initialisiert!!
    printf ("\nbetrag3 - wurde nicht initialisiert : ");
    betrag3.print ();

    betrag3 = betrag1 + betrag2;
    printf ("\nbetrag3 = betrag1 + betrag2: ");
    betrag3.print();

    betrag3 = betrag3 + 10.;
    printf ("\nbetrag3 nach Addition von 10.: ");
    betrag3.print();

    //betrag3 = 40 + betrag3;
    //printf ("\nbetrag3 nach betrag3 = 40 + betrag3: ");
    //betrag3.print();
```

```
    //geht nicht, da der erste Operand ein Datenobjekt der
    //Klasse obj_type sein muß!!!
}
```

Die Ausgabe ist:

```
betrag1 nach Initialisierung mit  5.:   5.00
betrag2 nach Initialisierung mit 10.:  10.00
betrag3 - wurde nicht initialisiert : 25764397371721250200000000000000.00
betrag3 = betrag1 + betrag2:  15.00
betrag3 nach Addition von 10.:  25.00
```

Beispiel für die zweite Möglichkeit:

```
//Datei Op2.cpp
#include <stdio.h>

class obj_type {
    friend obj_type operator + (obj_type, obj_type);
    friend obj_type operator + (float, obj_type );
    friend obj_type operator + (obj_type, float);
private:
    float summand;
public:
    obj_type ( float zahl)      // Memberfunktion
        {                       //       +
         summand = zahl;        // Konstruktor
        };
    obj_type () {};
    void print ();
};

void obj_type::print () {
       printf ("%6.2f", summand);
    }

obj_type operator + (obj_type p1, obj_type p2) {
    return p1.summand + p2.summand;
}

obj_type operator + (obj_type p1, float p2) {
    return p1.summand + p2;
}

obj_type operator + (float p1, obj_type p2) {
    return p1 + p2.summand;
}

void main (void) {
    obj_type betrag1 (5.);
    obj_type betrag2 (10.);

    printf ("\n\n\nbetrag1 nach Initialisierung mit  5.: ");
    betrag1.print ();

    printf ("\nbetrag2 nach Initialisierung mit 10.: ");
```

```
    betrag2.print ();

    obj_type betrag3;         //betrag3 ist nicht initialisiert!!
    printf ("\nbetrag3 - wurde nicht initialisiert : ");
    betrag3.print ();

    betrag3 = betrag1 + betrag2;
    printf ("\nbetrag3 = betrag1 + betrag2: ");
    betrag3.print();

    betrag3 = betrag3 + 10.;
    printf ("\nbetrag3 nach Addition von 10.: ");
    betrag3.print();

    betrag3 = 40 + betrag3;
    printf ("\nbetrag3 nach betrag3 = 40 + betrag3: ");
    betrag3.print();
}
```

Die Ausgabe des Programms ist:

```
betrag1 nach Initialisierung mit  5.:   5.00
betrag2 nach Initialisierung mit 10.:  10.00
betrag3 - wurde nicht initialisiert : 25764397371721250200000000000000.00
betrag3 = betrag1 + betrag2:  15.00
betrag3 nach Addition von 10.:  25.00
betrag3 nach betrag3 = 40 + betrag3:  65.00
```

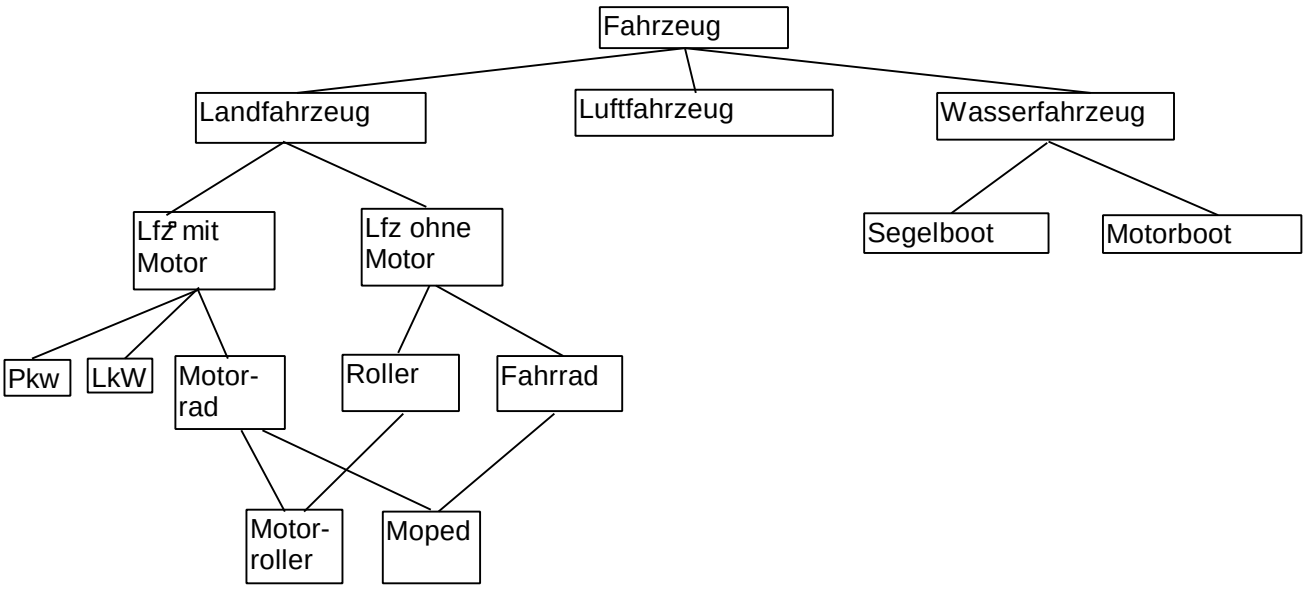
8 Vererbung

Ab diesem Kapitel wollen wir uns mit der Vererbung befassen. Manches ist Ihnen dabei prinzipiell schon aus den vorhergehenden Kapiteln und Beispielen bekannt, und wird hier nur ausführlicher dargestellt.

Das Prinzip der Vererbung

Die Vererbung ist wahrscheinlich das leistungsfähigste Merkmal der objekt-orientierten Programmierung. Die Vererbung ist der Vorgang für die Erstellung neuer Klassen, die **abgeleitete Klassen** genannt werden, von bereits bestehenden Klassen, die manchmal auch **Basisklasse** oder **Elternklasse** genannt werden. Die Mehrzahl in dem letzten Satz ist sehr wohl richtig und nicht ein logischer Fehler, weil es durchaus möglich ist, daß eine abgeleitete Klasse mehrere Eltern hat. Zu mehrfacher Vererbung kommen wir später.

Jede Klasse hat eine Reihe von Fähigkeiten (Elemente). Eine Frage des Entwurfs von Klassen ist, auf welcher Ebene der Klassenhierarchie die verschiedenen Fähigkeiten angeordnet werden sollen. Ein Ihnen schon bekanntes Beispiel soll diese Problemstellung verdeutlichen:



Ein grundlegendes und auch einfaches Prinzip für die Klassenkonstruktion ist, daß eine höhere Klasse immer diejenigen Fähigkeiten enthält, die ihren Kindern gemeinsam ist (Schnittmenge). Dies bedeutet in obigem Bild, daß die Klasse

Wasserfahrzeug alle Fähigkeiten enthalten soll, die Segelbooten und Motorbooten gemeinsam ist.

Die abgeleitete Klasse erbt alle Fähigkeiten (Datenelemente, Methoden) der Basisklasse, kann aber zusätzlich über eigene Verbesserungen und Verfeinerungen verfügen. Dabei bekommt die Basisklasse von der Vererbung nichts mit und bleibt somit unverändert.

Klassenbibliotheken

Ein wichtiger Vorteil der Vererbung bietet die Wiederverwendbarkeit des Codes. Nachdem eine Basisklasse auf Fehler überprüft worden ist, muß sie nicht mehr angerührt werden und kann trotzdem auf verschiedene Situationen angepaßt werden. Dadurch spart man Zeit und Geld und erhöht die Zuverlässigkeit eines Programms.

Eine Folge der Wiederverwendbarkeit sind die im Handel angebotenen Klassenbibliotheken. Ein Programmierer kann eine Klasse verwenden, die von einer anderen Person erstellt wurde und von der er keinen Quellcode hat. Somit hat der Entwickler der Klasse und der Anwender viele Vorteile.

8.1 Vererbung in C++

Eine Klasse wird von einer anderen abgeleitet, indem man nach dem Namen der Klasse - abgetrennt durch einen Doppelpunkt - den Namen der Basisklasse angibt. Damit sind automatisch alle Fähigkeiten (Daten, Methoden) der Basisklasse Bestandteil der abgeleiteten Klasse. **Konstruktoren werden nicht vererbt.**

Beginnen wir mit einem Beispiel:

```
//
// Einfaches Beispiel für Vererbung
// Datei: erb1.cpp

#include <stdio.h>
#include <string.h>

class Eltern                                    // Basisklasse
{
 public:                                        // Akzeptieren Sie hier das Schlüsselwort
                                                // public, auch wenn die Daten eigentlich
                                                // geschützt werden sollten. Den Schutz,
                                                // den protected bietet, werden wir bald
                                                // kennenlernen !
   char Staatsangehoerigkeit_vererbbar [20];
```

```
   Eltern ()
   {
     strcpy (Staatsangehoerigkeit_vererbbar, "tuerkisch");
   };                                          // Standard-Konstruktor


   void Eltern_print ()
   {
    printf ("\n%s", Staatsangehoerigkeit_vererbbar);
   }
};

class Kind :public Eltern        // Abgeleitete Klasse. Nehmen Sie das
                                 // Schlüsselwort public bis auf weiteres
                                 // unbesehen hin!

// Eine Klasse wird von einer anderen abgeleitet, indem man nach dem
// Namen der Klasse - abgetrennt durch einen Doppelpunkt - den Namen der
// Basisklasse angibt. Vergessen Sie public, bis es erklärt wird.

{
  char Staatsangehoerigkeit_Geburtsland [20];
                                 // hat auch noch Staatangehörigkeit
                                 // des Geburtslandes
public:
  Kind (): Eltern ()             // hier wird Konstruktor der Basisklasse
                                 // aufgerufen. Nehmen Sie das bis auf
                                 // weiteres unbesehen hin!
  {
    printf ("\nGib die Staatsangehörigkeit des Geburtslandes ein: ");

    gets (Staatsangehoerigkeit_Geburtsland);
  };

  void Kind_print ()
  {
    printf ("\n%s", Staatsangehoerigkeit_Geburtsland);
  }
};

void main ()
{
  Eltern Papa;
  printf ("\n\nPapa hat die Staatsangehörigkeit: ");
  Papa.Eltern_print();

  printf ("\n\nOskar:");
  Kind Oskar;
  printf ("\nOskar hat die beiden Staatsbürgerschaften:" );
  Oskar.Eltern_print();    // Eltern_print ist Member-Funktion
  Oskar.Kind_print();      // Kind_print ist Member-Funktion

printf ("\nMember-Datum vererbt: %s",Oskar.Staatsangehoerigkeit_vererbbar);
```

```
                                    // Staatsangehoerigkeit_vererbbar
                                    // ist Member-Datum
}
```

Der Dialog des Programmes ist:

```
Papa hat die Staatsangehörigkeit:
tuerkisch

Oskar:
Gib die Staatsangehörigkeit des Geburtslandes ein:
Oskar hat die beiden Staatsbürgerschaften:
tuerkisch
deutsch
Member-Datum vererbt: tuerkisch
```

Eine Basisklasse vererbt ihre Elemente (Daten und Funktionen) an eine abgeleitete Klasse. Die vererbten Datenelemente werden vom Compiler bei den Kindern angelegt, aber bei entsprechendem Zugriffsschutz (siehe später) wird der Zugriff verhindert. Dies bedeutet, daß ein Objekt einer abgeleiteten Klasse zu seinen eigenen Elementen jeweils auch jedes geerbte Element der Basisklasse enthält. Vererbte Bestandteile der Basisklasse können angesprochen werden wie Bestandteile der eigenen Klasse.

Beispiele für das Ansprechen von Bestandteilen der Basisklasse im abgeleiteten Objekt:

```
Hugo.Eltern_print();        // Eltern_print ist Member-Funktion

printf ("\nMember-Datum vererbt: %s",
                Hugo.Staatsangehoerigkeit_vererbbar);
                            // Staatsangehoerigkeit_vererbbar
                            // ist Member-Datum
```

Nächstes Beispiel:

```
// Einfaches Beispiel für Vererbung
// Datei: erb2.cpp

#include <stdio.h>
#include <string.h>

class Klasse1                                   // Basisklasse
{
 public:
   int zahl1;
   Klasse1 ()                                   // Standard-Konstruktor
   {
     printf ("\nKonstruktor 1 aufgerufen");
     zahl1 = 1;
   };
   void Klasse1_print ()
   {
    printf ("\nzahl1 hat den Wert%d",zahl1);
   }
};

class Klasse2 :public Klasse1                   // Abgeleitete Klasse
{
 public:
   int zahl2;
   Klasse2 () : Klasse1()       // Standardkonstruktor für Klasse2 mit Aufruf
   {                            // Konstruktor für Klasse 1
    printf ("\nKonstruktor 2 aufgerufen");
    zahl2 = 2;
   };
   void Klasse2_print ()
   {
    printf ("\nzahl2 hat den Wert%d",zahl2);
   }
};


class Klasse3 :public Klasse2   // Abgeleitete Klasse
{
 public:
   int zahl3;
   Klasse3 () : Klasse2 ()      // Standard-Konstruktor für Klasse 3 mit Aufruf
                                // Konstruktor für Klasse 2
   {
     printf ("\nKonstruktor 3 aufgerufen");
     zahl3 = 3;
   };
   void Klasse3_print ()
   {
    printf ("\nzahl3 hat den Wert%d",zahl3);
   }
};
```

```
void main ()
{
  printf ("\nnun wird Objekt1 definiert");
  Klasse1 objekt1;
  printf ("\nnun wird Objekt2 definiert");
  Klasse2 objekt2;
  printf ("\nnun wird Objekt3 definiert");
  Klasse3 objekt3;

  printf ("\n\nobjekt1:");
  objekt1.Klasse1_print ();

  printf ("\n\nobjekt2:");
  objekt2.Klasse1_print ();
  objekt2.zahl1 = 7;
  objekt2.Klasse1_print ();
  objekt2.Klasse2_print ();

  printf ("\n\nobjekt3:");
  objekt3.Klasse1_print ();
  objekt3.zahl1 = 30;
  objekt3.Klasse1_print ();
  objekt3.Klasse2_print ();
  objekt3.zahl2 = 300;
  objekt3.Klasse2_print ();
  objekt3.Klasse3_print ();
}
```

Die Ausgabe des Programmes ist:

```
nun wird Objekt1 definiert
Konstruktor 1 aufgerufen
nun wird Objekt2 definiert
Konstruktor 1 aufgerufen
Konstruktor 2 aufgerufen
nun wird Objekt3 definiert
Konstruktor 1 aufgerufen
Konstruktor 2 aufgerufen
Konstruktor 3 aufgerufen

objekt1:
zahl1 hat den Wert1

objekt2:
zahl1 hat den Wert1
zahl1 hat den Wert7
zahl2 hat den Wert2

objekt3:
zahl1 hat den Wert1
zahl1 hat den Wert30
zahl2 hat den Wert2
zahl2 hat den Wert300
zahl3 hat den Wert3
```

Sie sehen, daß beispielsweise `zahl1`, `zahl2` und `zahl3` Komponenten von `objekt3` sind. In diesem Fall ist die Vererbung sogar mehrstufig, da erst `Klasse1` an `Klasse2` vererbt wird und dann erst `Klasse2` an `Klasse3`.

8.2 Konstruktoren und Destruktoren bei abgeleiteten Klassen

Beim Konstruktoraufruf für eine abgeleitete Klasse wird der Konstruktor für die Basisklasse aufgerufen, damit ein Objekt einer abgeleiteten Klasse definiert und initialisiert wird. Hierbei sind die **Parameter an den Konstruktor der Basisklasse weiterzugeben** wie in folgendem Beispiel:

```
Klasse2 (int par, int j) : Klasse1 (j)
                    // Konstruktor mit Parametern par und j für Klasse2
                    // und Aufruf des Konstruktors für Klasse1
                    // mit Übergabe des Parameters j
```

In anderen Worten: Wenn der Konstruktor der Basisklasse Parameter erwartet, so müssen diese vom Konstruktor der abgeleiteten Klasse bereitgestellt und mit Hilfe der folgenden Syntax übergeben werden: Zwischen Funktionskopf und der öffnenden geschweiften Klammer bei der Definition des Konstruktors der abgeleiteten Klasse folgt ein Doppelpunkt und dahinter die in Klammern stehende aktuelle Parameterliste für den Konstruktor der Basisklasse mit vorangestelltem Namen der Basisklasse.

Der Konstruktor der Basisklasse wird hierbei vor dem Konstruktor der abgeleiteten Klasse ausgeführt.

Entsprechend wird bei den Destruktoren zuerst der Destruktor der abgeleiteten Klasse und dann der Destruktor der Basisklasse ausgeführt,.

Beim Anlegen eines Objektes einer abgeleiteten Klasse wird erst der Konstruktor der Basisklasse aufgerufen. Mit diesem Konstruktor wird der aus der Basisklasse geerbte Objektteil initialisiert. Erst danach wird der Konstruktor der abgeleiteten Klasse aufgerufen, der zumindest die neu hinzugekommenen Komponenten initialisiert. Es ist aber auch möglich, mit dem Konstruktor der abgeleiteten Klasse die geerbten Komponenten neu zu initialisieren, wenn die Initialisierung durch den Konstruktor der Basisklasse für ein Objekt der abgeleiteten Klasse nicht sinnvoll ist.

Programmbeispiel:

```
//
// Einfaches Beispiel für Vererbung
```

```
// Datei: erb3.cpp

#include <stdio.h>
#include <string.h>

class Klasse1
{
 public:
   int zahl1;
   Klasse1 (int par)
   {
    zahl1 = par;
   };
   void Klasse1_print ()
   {
    printf ("\nzahl1 hat den Wert%d ",zahl1);
   }
};

class Klasse2 :public Klasse1          // Abgeleitete Klasse
{
 public:
   int zahl2;
   Klasse2 (int par, int j) : Klasse1 (j)
                                       // Konstruktor mit Parametern für Klasse2
                                       // und Aufruf des Konstruktors für Klasse1
                                       // mit Übergabe des Parameters
   {
     zahl2 = par;
   };
   void Klasse2_print ()
   {
    printf ("\nzahl2 hat den Wert%d ",zahl2);
   }
};

void main ()
{
  Klasse1 objekt1 (1);
  Klasse2 objekt2 (3,2);

  printf ("\n\nobjekt1:");
  objekt1.Klasse1_print ();

  printf ("\n\nobjekt2:");
  objekt2.Klasse1_print ();
  objekt2.zahl1 = 7;
  objekt2.Klasse1_print ();
  objekt2.Klasse2_print ();

}
```

Die Ausgabe des Programmes ist:

```
objekt1:
zahl1 hat den Wert 1
```

```
objekt2:
zahl1 hat den Wert 2
zahl1 hat den Wert 7
zahl2 hat den Wert 3
```

8.3 Namensgleichheit von Elementen in der Basis- und der abgeleiteten Klasse

Hier eine Variante des letzten Programms:

```
//
// Einfaches Beispiel für Vererbung
// Datei: erb4.cpp

#include <stdio.h>
#include <string.h>

class Klasse1
{
 public:
   int zahl;
   Klasse1 (int par)
   {
    zahl = par;
   };
   void Klasse1_print ()
   {
    printf ("\nzahl von Klasse1 hat den Wert: %d",zahl);
   }
};

class Klasse2 :public Klasse1          // Abgeleitete Klasse
{
 public:
   int zahl;
   Klasse2 (int par, int j) : Klasse1 (j)
                              // Konstruktor mit Parametern par und j für
                              //  Klasse2
                              // und Aufruf des Konstruktors j für Klasse1
                              // mit Übergabe des Parameters
  {
     zahl = par;
   };
   void Klasse2_print ()
   {
    printf ("\nzahl von Klasse 2 hat den Wert: %d",zahl);
   }
};

void main ()
{
```

```
  Klasse1 objekt1 (1);
  Klasse2 objekt2 (3,2);

  printf ("\n\nobjekt1:");
  objekt1.Klasse1_print ();

  printf ("\n\nobjekt2:");
  objekt2.Klasse1_print ();
  objekt2.Klasse2_print ();

  objekt2.Klasse1::zahl = 7;
  objekt2.zahl = 15;
  objekt2.Klasse1_print ();
  objekt2.Klasse2_print ();

}
```

Die Ausgabe des Programms ist:

```
objekt1:
zahl von Klasse1 hat den Wert: 1

objekt2:
zahl von Klasse1 hat den Wert: 2
zahl von Klasse 2 hat den Wert: 3
zahl von Klasse1 hat den Wert: 7
zahl von Klasse 2 hat den Wert: 15
```

Eine Methode der Basisklasse kann nur auf Attribute der Basisklasse zugreifen. Eine Methode der abgeleiteten Klasse kann auf die in der abgeleiteten Klasse definierten Attribute und auf die Attribute der Basisklasse zugreifen - vorausgesetzt, der Zugriffsschutz verhindert dies nicht.

Wird ein von einer Methode einer abgeleiteten Klasse ein Name in der abgeleiteten Klasse nicht gefunden, so wird in der Basisklasse gesucht. Wird also in `Klasse1` und `Klasse2` derselbe Name `zahl` für ein Datenelement verwendet, so wird bei einem Objekt der abgeleiteten Klasse der Name zuerst in der abgeleiteten Klasse gefunden. Der Zugriff auf ein Element der Basisklasse erfolgt mit Hilfe des Scope-Operators wie in folgendem Beispiel:

```
objekt2.Klasse1::zahl = 7;
objekt2.zahl = 15;
```

8.4 Zugriffsschutz bei Vererbung durch Schutzattribute und Zugriffsmodifizierer

Bei der Vererbung werden auch die Schutzattribute von Elementen vererbt. Grundsätzlich kann eine abgeleitete Klasse nicht mehr Zugriffsrechte auf Elemente einer Basisklasse haben als in der Basisklasse selbst.

Zugriffsschutz: Schutzattribute private, protected, public von Klassenelementen

Elemente der Basisklasse, die das Schutzattribut **`private`** haben, sind in der abgeleiteten Klasse nicht zugänglich. Sie sind nur in den Element-Funktionen der Basisklasse erreichbar.

Elemente, die als **`public`** eingestuft sind, sind ungeschützt.

Speziell für abgeleitete Klassen gibt es eine weitere Schutzkategorie: **`protected`**. Solche Daten sind in den abgeleiteten Klassen verfügbar. Alle `protected`-Komponenten einer Klasse haben ohne Vererbung die gleichen Eigenschaften wie `private`-Elemente.

Beispiel:

```
//
// Einfaches Beispiel für Vererbung
// Datei: erb5.cpp

#include <stdio.h>

class Klasse1
{
 protected:
   int zahl1;
 public:
   Klasse1 (int par)
   {
    zahl1 = par;
   };
 protected:
   void Klasse1_print ()
   {
    printf ("\nzahl1 hat den Wert %d",zahl1);
   }
};
```

class Klasse2 :public Klasse1 // Abgeleitete Klasse

```
{
   int zahl2;
 public:
   Klasse2 (int par, int j) : Klasse1 (j)
```

// Konstruktor mit Parametern par und j für Klasse2
// und Aufruf des Konstruktors j für Klasse1
// mit Übergabe des Parameters

```
  {
     zahl2 = par;
     zahl1 = 50;
```

//zugänglich

```
   };
   void Klasse2_print ()
```

```
   {
    printf ("\nzahl2 hat den Wert %d",zahl2);
    printf ("\nzahl1 hat den Wert %d",zahl1);
   }
};

void main ()
{
  Klasse1 objekt1 (1);
  Klasse2 objekt2 (3,2);

  printf ("\n\nobjekt1:");
  //objekt1.Klasse1_print ();              //nicht zugänglich
  printf ("\n\nobjekt2:");
  //objekt2.Klasse1_print ();              //nicht zugänglich
  //objekt2.zahl1 = 7;                     //nicht zugänglich
  objekt2.Klasse2_print ();
}
```

Die Ausgabe des Programmes ist:

```
objekt1:

objekt2:
zahl2 hat den Wert 3
zahl1 hat den Wert 50
```

Zugriffsmodifizierer der Basisklasse private, protected und public

Gehen wir von obigem Beispiel aus:

```
class Klasse2 :public Klasse1       // Abgeleitete Klasse
```

Zugriffsmodifizierer der Basisklasse

Der **Zugriffsmodifizierer `public`** bewirkt, daß alle Schutzattribute aus der Basisklasse unverändert übernommen werden. Der **Zugriffsmodifizierer `private`** bewirkt, daß in der abgeleiteten Klasse alle geerbten Elemente der Basisklasse `private` sind. Das hat in der abgeleiteten Klasse noch keine Konsequenzen. Die Auswirkungen werden erst sichtbar, wenn Objekte einer solchen Klasse definiert werden oder wenn diese Klasse an eine andere Klasse weiter vererben soll.

Wird mit dem Zugriffsmodifizierer **`private`** vererbt, so sind beispielsweise Elemente, die in der Basisklasse `public` sind, in der abgeleiteten Klasse `private`. Dies hat die Konsequenz, daß man aus globalen Funktionen nicht mehr auf diese Komponenten zugreifen kann. Komponenten, die `private` geerbt wurden, sind beim Weitervererben dann in den nachfolgenden Klassen nicht mehr greifbar.

Geben Sie den Zugriffsmodifizierer **`protected`** an, so werden `public` und `protected` Komponenten `protected`, `private` Komponenten bleiben `private`.

Geben Sie keinen Zugriffsmodifizierer an, so wird der Default-Wert genommen. Der Default-Wert beim Borland C++ Compiler ist `private`.

Tabellarische Übersicht:
Sichtbarkeit von Member-Funktionen und Member-Daten durch Kombination von Zugriffsmodifizierern und Schutzattributen der Klassenelemente

Zugriffs-modifizierer	*Schutzattribute der Elemente (Member-Fkt./Member-Daten)*			
	public	*protected*	*private*	
public	public	protected	private	
protected	protected	protected	private	
private	private	private	private	

9 Mehrfache Vererbung

Mehrfache Vererbung bedeutet, daß eine Klasse mehrere Basisklassen haben kann und von diesen Basisklassen Elemente erbt.

Eine Problemstellung für den Compiler bei der mehrfachen Vererbung ist die Frage nach dem Suchmechanismus für Namen, wenn ein Name in einer Klasse nicht gefunden wird und in den Vaterklassen gesucht werden muß. Was passiert, wenn in zwei Basisklassen derselbe Name vorkommt? Welcher ist der richtige? Soll das zuerst gefundene Element genommen werden?

C++ läßt hier nichts anbrennen und setzt nicht auf automatische komplexe Suchstrategien - die der Programmierer selbstverständlich auch beherrschen muß - sondern sagt, daß der Programmierer durch Angabe des Scope-Operators und der Klasse angeben muß, welches Element er meint.

Hier wieder ein Beispiel, damit es nicht zu trocken wird:

```
// Beispiel für mehrfache Vererbung
// Datei: erb7.cpp
#include <stdio.h>
#include <string.h>

class Vater                                    // #1 Basisklasse
{
 protected:
   char Staatsangehoerigkeit_vererbbar [20];
 public:
   Vater ()
   {
     strcpy (Staatsangehoerigkeit_vererbbar, "englisch");
   };                                          // Standard-Konstruktor
   void Vater_print ()
   {
    printf ("\n%s", Staatsangehoerigkeit_vererbbar);
   }
};

class Mutter                                   // #2 Basisklasse
{
 public:
   char Staatsangehoerigkeit_vererbbar [20];
   Mutter ()
   {
     strcpy (Staatsangehoerigkeit_vere rbbar, "französisch");
   };                                          // Standard-Konstruktor
   void Mutter_print ()
   {
    printf ("\n%s", Staatsangehoerigkeit_vererbbar);
   }
```

```
};

class Kind :public Vater, public Mutter    // #3 Abgeleitete Klasse
{
  char Staatsangehoerigkeit_Geburtslan d [20];
          // hat auch noch Staatsangehörigkeit des Geburtslandes
public:
 Kind (): Vater (), Mutter ()              // #4 hier werden die Konstruktoren
                                           // der Basisklasse aufgerufen
  {
    printf ("\nGib die Staatsangehörigkeit des Geburtslandes ein: ");
    gets (Staatsangehoerigkeit_Geburtsland);
  };
  void Kind_print ()
  {
    printf ("\n%s", Staatsangehoerigkeit_Geburtsland);
  }
};

void main ()
{
  Vater Papa;
  printf ("\n\nPapa hat die Staatsangehörigkeit: ");
  Papa.Vater_print();

  Mutter Mama;
  printf ("\n\nMama hat die Staatsangehörigkeit: ");
  Mama.Mutter_print();


  printf ("\n\nOskar:");
  Kind Oskar;                              // #5 Definition eines Objektes
                                           // der abgeleiteten Klasse
  printf ("\nOskar hat die beiden Staatsbürgerschaften:");
  Oskar.Kind_print();
  Oskar.Vater_print ();
  // printf ("\nMember-Datum vererbt: %s", // #6 vererbtes Element
  //       Oskar.Vater::Staatsangehoerigkeit_vererbbar);
  // nicht zugänglich wegen protected in Klasse Vater
  printf ("\n\nHugo:");
  Kind Hugo;                               // #7 Definition eines Objektes
                                           // der abgeleiteten Klasse
  printf ("\nHugo hat die beiden Staatsbürgerschaften:");
  Hugo.Kind_print();
  Hugo.Mutter_print();
  printf ("\nMember-Datum vererbt: %s",    // #8 vererbtes Element
       Hugo.Mutter::Staatsangehoerigkeit_vererbbar);
}
```

Der folgende Dialog wurde mit dem Programm geführt:

```
Papa hat die Staatsangehörigkeit:
englisch
```

```
Mama hat die Staatsangehörigkeit:
französisch

Oskar:
Gib die Staatsangehörigkeit des Geburtslandes ein: deutsch
Oskar hat die beiden Staatsbürgerschaften:
deutsch
englisch

Hugo:
Gib die Staatsangehörigkeit des Geburtslandes ein: amerikanisch
Hugo hat die beiden Staatsbürgerschaften:
amerikanisch
französisch
Member-Datum vererbt: französisch
```

Sie sehen (siehe #3 in obigem Programm), wie es in C++ möglich ist, eine Klasse von mehreren Basisklassen - hier von der Klasse Vater (#1) und Mutter (#2) - abzuleiten. Dazu gibt man die Basisklassen durch Kommata getrennt nach dem Doppelpunkt an.

```
class Kind :public Vater, public Mutter      // #3 Abgeleitete Klasse
```

Für jede Basisklasse kann der Zugriffsmodifizierer separat vergeben werden. In unserem Beispiel wurde zweimal der Modifizierer public gewählt.

Unter #4 sehen Sie die Konstruktoraufrufe für die Basisklassen bei der Definition des Konstruktors für die abgeleitete Klasse:

```
Kind (): Vater (), Mutter ()        // #4 hier werden die
                                    // Konstruktoren
                                    // der Basisklassen
                                    // aufgerufen
```

Wird ein Objektes der abgeleiteten Klasse (siehe #5 und #7) wie z.B.

```
Kind Oskar;                         // #5 Definition eines Objektes
                                    // der abgeleiteten Klasse
```

definiert, so werden für alle geerbten Elemente Speicherobjekte angelegt.

An Ihrer Nasenspitze sehe ich, daß Sie grübeln. Daher erweitern wir das obige Programm noch etwas. Wir setzen in der Vaterklasse das Schutzattribut von `Staatsangehoerigkeit_vererbbar` auf `public` und fügen in `main` den folgenden Teil an:

```
printf ("\n\nFelicitas:");
  Kind Felicitas;
  printf ("\nFelicitas hat die drei Staatsbürgerschaften:");
  Felicitas.Kind_print();
```

```
  strcpy (Felicitas.Mutter::Staatsangehoerigkeit_vererbbar,
        "spanisch");
  strcpy (Felicitas.Vater::Staatsangehoerigkeit_vererbbar,
        "italienisch");
  printf ("\nMember-Datum vererbt: Mutter %s",
      Felicitas.Mutter::Staatsangehoerig keit_vererbbar);
  printf ("\nMember-Datum vererbt Vater: %s",
      Felicitas.Vater::Staatsangehoerigkeit_vererbbar);

  printf ("\n\nVater");
  printf ("\n---------");
  Papa.Vater_print();

  printf ("\n\nMutter");
  printf ("\n---------");
  Mama.Mutter_print();

  printf ("\n\nFelicitas");
  printf ("\n---------");
  Felicitas.Mutter_print ();
  Felicitas.Vater_print ();
```

Der zusätzliche Dialog war::

```
Felicitas:
Gib die Staatsangehörigkeit des Geburtslandes ein:  dänisch
Felicitas hat die drei Staatsbür gerschaften:
dänisch
Member-Datum vererbt: Mutter spanisch
Member-Datum vererbt Vater: italienisch

Vater
-----
Vater_print: englisch

Mutter
------
Mutter_print: französisch

Felicitas
---------
Mutter_print: spanisch
Vater_print: italienisch
```

Dieses Beispiel zeigt, daß Felicitas drei Staatsangehörigkeiten besitzt:

```
char Staatsangehoerigkeit_Geburtsland [20];
Mutter::Staatsangehoerigkeit_vererbbar
Vater::Staatsangehoerigkeit_vererbbar
```

Die geerbten Staatsangehörigkeiten wurden durch

```
strcpy (Felicitas.Mutter::Staatsangehoerigkeit_vererbbar,
 "spanisch");
strcpy (Felicitas.Vater::Staatsangehoerigkeit_vererbbar,
 "italienisch");
```

einfach verändert. Betrug! Betrug! Man kann doch Staatsbürgerschaften nicht einfach wie ein Hemd wechseln!

Da

```
Felicitas.Mutter::Staatsangehoerigkeit_vererbbar
Felicitas. Vater::Staatsangehoerigkeit_vererbbar
```

lokale Elemente im Objekt Felicitas sind, steht im Paß des Vaters nach wie vor:

```
englisch
```

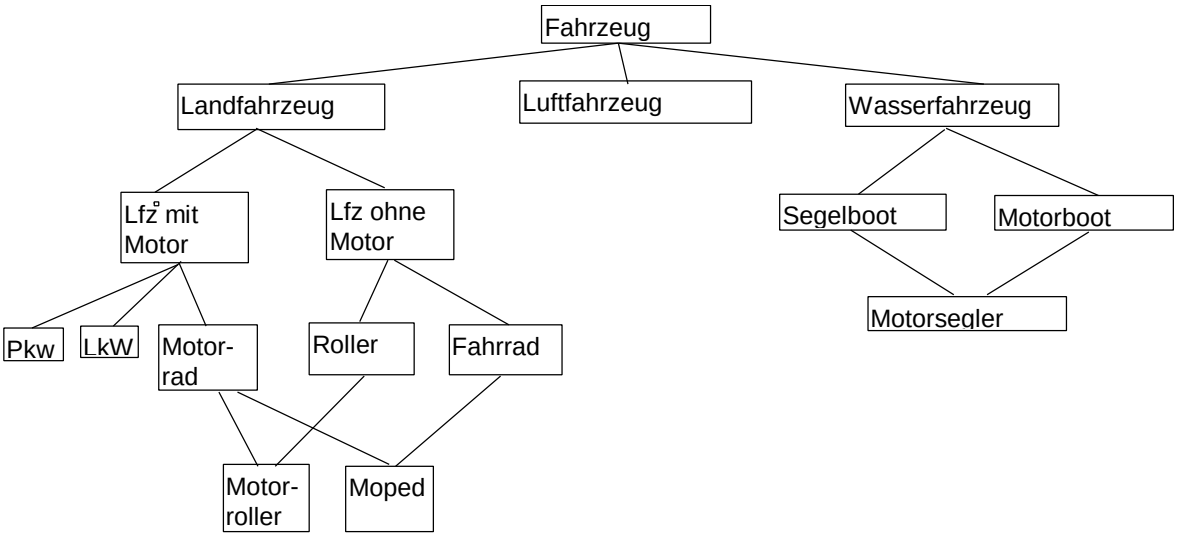
und im Paß der Mutter steht:

```
französisch
```

Die Urkundenfälschung von Felicitas ist auf Felicitas beschränkt!

9.1 Virtuelle Basisklassen

Betrachten wir wieder das Beispiel Motorsegler:



Die Klasse Motorsegler würde alle Eigenschaften (Elemente) der Klasse Wasserfahrzeug doppelt erben, einmal über die Klasse Segelboot und zum anderen über die Klasse Motorboot. Dies ist nicht gewollt, da die beiden Datenstrukturen - sieht man von der Vergeudung des Speicherplatzes ab - widersprüchliche Einträge enthalten könnten. Also muß man dafür sorgen, die Elemente von Wasserfahrzeug nur einfach zu vererben. Der Compiler tut dies, wenn man Wasserfahrzeug zur **virtuellen Basisklasse** macht. Dann sorgt der Compiler dafür, daß bei allen Vererbungsprozessen ein Element dieser Basisklasse maximal einmal in einer abgeleiteten Klasse vorkommt.

Beachten Sie bitte, daß Wasserfahrzeug eine ganz normale Klasse ist. Sie könnte beispielsweise ihre Elemente ganz normal an eine Klasse U-Boot vererben. Eine **Basisklasse** kann man jedoch auch **als virtuelle Basisklasse weitervererben**, wie hier Wasserfahrzeug nach Segelboot und Motorboot. Dabei hat die **Vererbung** von Wasserfahrzeug **als virtuelle Basisklasse** für Segelboot und Motorboot keine Konsequenzen. Erst bei Kindern, die sowohl von Segelboot, als auch von Motorboot erben - also bei den **Enkeln** von Wasserfahrzeug - zeigt die virtuelle Basisklasse Konsequenzen. Diese Enkel von Wasserfahrzeug können von der virtuellen Basisklasse jedes vererbte Element nur in einfacher Form enthalten.

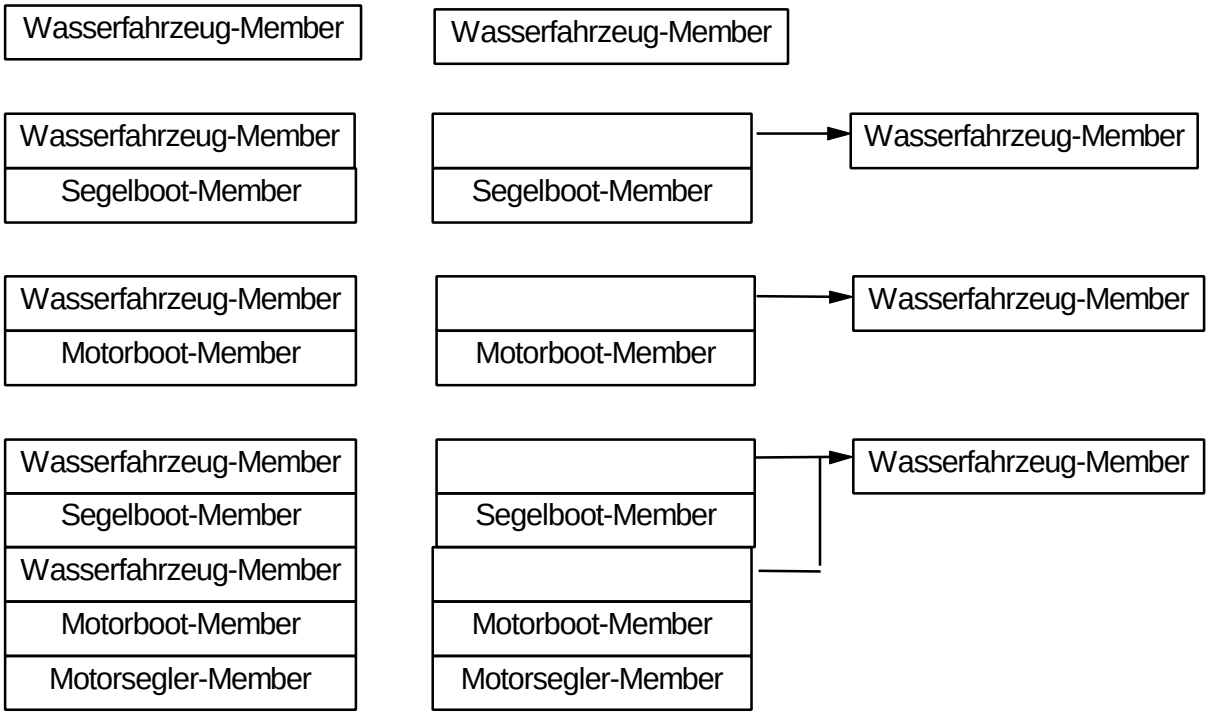
Beim Vererbungsvorgang muß man die Basisklasse wasserfahrzeug also virtuell an die Klasse Segelboot und Motorboot vererben:

```
class Segelboot : virtual wasserfahrzeug
{
 ...
};

class Motorboot : virtual wasserfahrzeug
{
 ...
};
```

Die Auswirkung der virtuellen Vererbung zeigt sich noch nicht in den Klassen `Motorboot` und `Segelboot`. Sie zeigt sich erst eine Ebene tiefer bei der Klasse `Motorsegler`, da jedes Objekt dieser Klasse jeweils nur ein Element der Klasse `Wasserfahrzeug` enthält.

Datenrepräsentation

a) ohne virtuelle Basisklasse **b) mit virtueller Basisklasse**

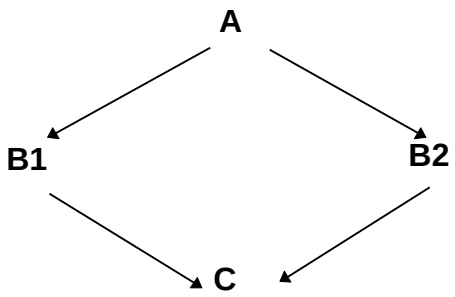


ohne virtuelle Basisklasse ist in der Klasse Motorsegler Wasserfahrzeug mehrdeutig, da doppelt vorhanden

Ohne virtuelle Basisklassen werden die geerbten Memberdaten und -funktionen alle im erbenden Objekt angelegt. Eine Implememtierungsmöglichkeit für den Compiler für virtuelle Basisklassen ist, im Falle von Mehrdeutigkeiten im Objekt nur Zeiger auf die mehrfach geerbten Member anzulegen.

Kurze Zusammenfassung:

Das Schlüsselwort `virtual` verhindert den doppelten Einbau von Erbgut in einer abgeleiteten Klasse. Zwar darf eine Basisklasse nicht direkt mehrmals an eine abgeleitete Klasse vererbt werden (class X : Y,Y { ... } ist verboten), aber indirekt ist dies sehr wohl (und auch ungewollt) möglich. Dazu ein einfaches Beispiel:



Das Erbgut von A gelangt auf zwei Wegen über B1 und B2 zu C und wird dort zweimal angelegt.

```
//
// Problem der Mehrfachvererbung
//

class A
{
  protected:
  int basedata;
};

class B1 : public A
  {  };

class B2 : public A
  {  };

class C : public B1, public B2
{
  public:
  int getdata ()
       { return basedata; }   // FEHLER: zweideutig
};
```

Der Übersetzer meldet hier einen Fehler. Warum? Sowohl die Klasse B1 als auch die Klasse B2 haben ein Datenelement `basedata`, weil sie beide von der Klasse A abstammen. Da die Klasse C sowohl von der Klasse B1 als auch von der Klasse B2

abgeleitet wurde, weiß der Übersetzer nicht, welches basedata er in der Klasse C verwenden soll.

Abhilfe schafft hier das Schlüsselwort virtual. B1 und B2 haben jeweils eine eigene Kopie von basedata. Aber beim Weitervererben an C wird basedata nur einmal angelegt.

```
//
// Lösung durch das Schlüsselwort virtual
//

class A
{
  protected:
  int basedata;
};

class B1 : virtual public A
  {  };

class B2 : virtual public A
  {  };

class C : public B1, public B2
{
  public:
  int getdata ()
       { return basedata; }   // Jetzt ist es eindeutig
};
```

Bis jetzt waren das nur Klassendefinitionen, die mit dem Compiler geprüft werden konnten. Jetzt soll Sie ein lauffähiges Programm vergnügen:

```
// Datei erb9.cpp
//
// Problem der Mehrfachvererbung
//
#include <stdio.h>

class A
{
  protected:
     int basedata;
  public:
     A () {
     basedata = 1;
     }
     void print ()
     {
     printf ("\nbasedata hat den Wert %d", basedata);
```

```
    }
};

class B1 : virtual public A
{
  public:
    B1 () : A()
    {
      basedata =  10;
    }
};

class B2 : virtual public A
{
  public:
    B2 () : A ()
    {
     basedata = 20;
    }
};

class C : public B1, public B2
{
  public:
    C () : B1 (), B2 ()
    {
     basedata = 100;
    }
};

void main ()
{
  printf ("\n\n");
  A alpha;
  B1 beta1;
  B2 beta2;
  C gamma;
  printf ("\n\nalpha: ");
  alpha.print();
  printf ("\n\nbeta1: ");
  beta1.print();
  printf ("\n\nbeta2: ");
  beta2.print();
  printf ("\n\ngamma: ");
  gamma.print();
}
```

Die Ausgabe des Programmes ist:

```
alpha:
basedata hat den Wert 1

beta1:
```

```
basedata hat den Wert 10

beta2:
basedata hat den Wert 20

gamma:
basedata hat den Wert 100
```

Ein paar Worte zum Mechanismus. Wie Sie bereits gehört haben, kann eine Klasse (A) nicht direkt als virtuell gekennzeichnet werden. Dies ist durchaus sinnvoll, denn so kann diese Klasse (A) in verschiedenen Zweigen der Hierarchie sowohl virtuell als auch in nicht virtueller Form auftreten. Der Mechanismus ist also flexibel. Die virtuelle Eigenschaft erhält die Klasse erst mit der Vererbung an andere Klassen (B1, B2) durch das vorangestellte Schlüsselwort. Werden diese Klassen dann abermals weitervererbt (an C), dann "weiß" der Compiler, daß er bei den virtuell geerbten Klassen eine Prüfung auf Mehrfachvererbung durchführen muß. Dabei spielt es keine Rolle, ob noch Zwischenstufen vorhanden sind. Zum Beispiel hätte B1 noch an eine Klasse C1 und B2 an eine Klasse C2 vererbt werden können. Bei der Vererbung von C1 und C2 an eine Klasse D würden dennoch die Anteile, die von A stammen, nur einmal weitergegeben.

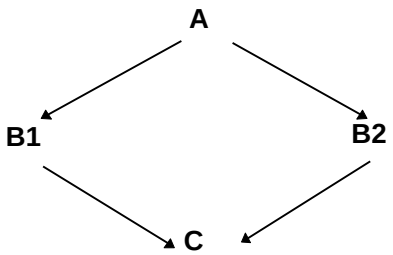
Konstruktoren bei virtuellen Basisklassen

Wenn Basisklassen als virtuelle Basisklassen vererbt werden, gelten für die Konstruktoren der abgeleiteten Klassen die folgenden Regeln:

?? Beim Anlegen eines Objektes einer abgeleiteten Klasse werden zuerst die Konstruktoren der virtuellen Basisklassen aufgerufen und erst dann die Konstruktoren von nichtvirtuellen Basisklassen.

?? Hierbei wird immer der Standard-Konstruktor der virtuellen Basisklasse aufgerufen. Der Aufruf dieses Standard-Konstruktors erfolgt stets von der zuletzt abgeleiteten Klasse. Alle anderen Aufrufe des Konstruktors der virtuellen Basisklasse werden nicht ausgeführt, d.h. ignoriert.

Beispiel:




```
// Datei erb9.cpp
//
```

```
// Problem der Mehrfachvererbung
//
....
class A
{
 ...
     A () {                                  // Default-Konstruktor
     basedata = 1;
     }
 ...
};

class B1 : virtual public A
{
  public:
     B1 () : A()
...
}

class B2 : virtual public A
{
  public:
     B2 () : A ()
 ...
};

class C : public B1, public B2
{
  public:
     C () : B1 (), B2 ()

 ...
};


void main ()
{
  A alpha;          // Aufruf des Standard-Konstruktors für A

  B1 beta1;         // Aufruf des Standard-Konstruktors von A und
                    // dann des Konstruktors von B1

  B2 beta2;         // Aufruf des Standard-Konstruktors von A und
                    // dann des Konstruktors von B2

  C gamma;          // Aufruf des Standard-Konstruktors von A,
                    // dann des Konstruktors von B1 und B2,
                    // dann der Aufruf des Konstruktors von C.
                    // Der Aufruf des Standard-Konstruktors A von
                    // B1 bzw. B2 wird ignoriert
```

```
...
}
```

9.2 Virtuelle Methoden

Virtuelle Methoden sind von entscheidender Bedeutung, wenn es darum geht, die Eigenschaft der Polymorphie zu realisieren. Wie Sie inzwischen wissen, kann eine Klasse Eigenschaften von einer Basisklasse erben. Die abgeleitete Klasse hat die Möglichkeit, eine geerbte Elementfunktion zu ändern, indem sie sie mit einer eigenen Elementfunktion gleichen Namens und dem gleichen Rückgabewert und der gleichen Parameterliste überlädt.

Erfolgt der Aufruf einer Elementfunktion mit einem Datenobjekt, so ist aus dem Typ des Datenobjekts ersichtlich, welche Funktion aufgerufen werden soll. Der Compiler stellt dies bereits beim Compilieren fest. Dieser Vorgang wird als **statische Bindung** bezeichnet.

Ein Objekt einer abgeleiteten Klasse darf jederzeit als Objekt der Basisklasse verwendet werden. Dies gilt auch, wenn es über Zeiger oder Referenzen auf Objekte der Basisklasse manipuliert wird. Der Compiler geht aber dann davon aus, daß es sich um Objekte der Basisklasse handelt und ruft die Methoden der Basisklasse auf, auch wenn sie in der abgeleiteten Klasse neu implementiert wurden. Dies liegt an der statischen Bindung, da mit einem Zeiger (einer Referenz) auf die Basisklasse die Methoden der Basisklasse als Code generiert werden.

Hier zunächst ein Erproben der Aussage, daß ein Objekt einer abgeleiteten Klasse jederzeit als Objekt der Basisklasse verwendet werden kann:

```
#include <stdio.h>
//Datei: zuobj.cpp

class alpha
{
  public:
    int zahl;
    alpha () {
    zahl = 6;
    }
};

class beta : public alpha
{
  public:
    char zeichen;
    beta () : alpha () {
    zeichen = 'o';
    zahl = 12;
    };

};

void main (void)
{
```

```
  alpha * zeiger_alpha;
  beta  * zeiger_beta  = new beta;
  alpha c;
  printf ("\n\nc.zahl hat den Wert %d", c.zahl);

  zeiger_alpha = zeiger_beta;   // ein Objekt der abgeleiteten
                                // Klasse darf jederzeit
                                // als Objekt der Basisklasse
  c = *zeiger_beta;             // verwendet werden


  printf ("\nc.zahl hat den Wert %d", c.zahl);

  delete zeiger_beta;
}
```

Hier die Ausgabe des Programms:

```
c.zahl hat den Wert 6
c.zahl hat den Wert 12
```

Damit die Information, daß Objekte einer abgeleiteten Klasse manipuliert werden, bei Funktionsaufrufen nicht verloren geht, muß zur Laufzeit, in Abhängigkeit vom tatsächlichen Typ des Objektes, die richtige Funktion aufgerufen werden. Dies wird mit **dynamischem Binden** oder **spätem Binden** bezeichnet.

In C++ muß hierfür das Schlüsselwort **virtual** bei der Deklaration einer Funktion in der Basisklasse angegeben werden, wenn die Funktion in einer abgeleiteten Klasse eventuell überschrieben (redefiniert) werden soll. Nichtvirtuelle Funktionen können nicht überschrieben, sondern nur überladen werden. Bei der Redefinition in der abgeleiteten Klasse muß die Methode nicht erneut durch **virtual** gekennzeichnet werden. Es wird aber aus Gründen der besseren Lesbarkeit der Programme empfohlen.

Wenn eine Memberfunktion als virtuelle Funktion deklariert wird, wird bei Verwendung von Zeigern und Referenzen erst zur Laufzeit entschieden, welche Funktion aufgerufen wird. In Abhängigkeit von der Klasse des Objektes wird die dazugehörige Funktion aufgerufen.

Unter **Polymorphismus im engeren Sinne** versteht man die Fähigkeit, den Typ eines Objektes zur Laufzeit zu ermitteln. Dies bedeutet, daß zur Laufzeit die entsprechenden redefinierten Methoden aufgerufen werden müssen. Polymorphismus im engeren Sinne existiert damit nur für redefinierte Methoden bei einem indirekten Zugriff über Pointer oder Referenzen.

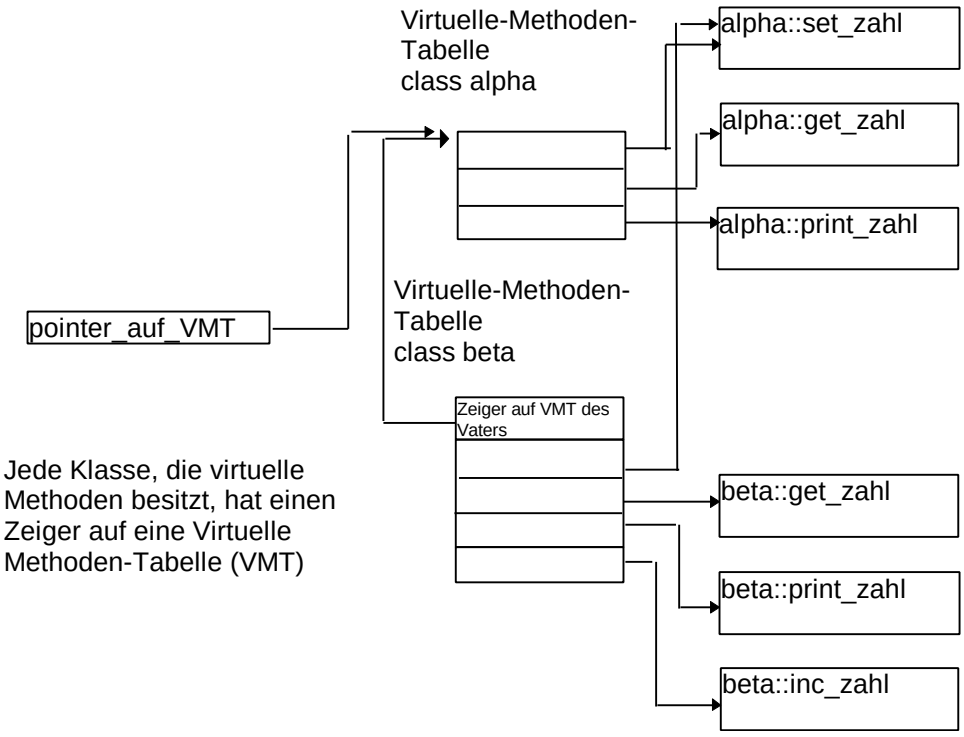
Beachten Sie aber, daß ein Funktionsaufruf einer virtuellen Funkion Zeit kostet, da erst zur Laufzeit festgestellt werden muß, welche Funktion tatsächlich aufgerufen werden muß.

Realisiert wird dies vom Compiler mit Hilfe von Virtuellen-Methoden-Tabellen je Klasse.

Hier ein Beispiel für die Verwendung des Schlüsselwortes `virtual`:

```
class alpha {
       int zahl;
public:
       virtual int get_zahl () {
       ...};
       virtual void set_zahl (int){
       ...};
       virtual void print_zahl () {
       ...};
}


class beta : public alpha {
public:
       int get_zahl () {
       ...};
       void print_zahl () {
       ...};
       virtual int inc_zahl (){
       ...};
}
```

Als triviale Anwendung hierfür kann man sich etwa vorstellen, daß die Ein/Ausgabe in Klasse `alpha` dezimal, in Klasse `beta` hexadezimal erfolgen soll.



Hier nun konkrete Programme:

```
//
// Datei virt1.cpp
//
#include <stdio.h>
#include <stdlib.h>

class Vater1
{
  protected:
    int zahl;
  public:
    Vater1 () {
       zahl = 1;
    };
    void show (void) {
    printf ("\nIch bin vom Typ Vater1");
    printf ("\nZahl hat den Wert: %d", zahl);
    };
};

class Sohn1 : public Vater1
{
 public:
    Sohn1 () : Vater1 () {
    };
    void show (void) {
       printf ("\nIch bin vom Typ Sohn1");
       printf ("\nZahl hat den Wert: %d", zahl);
    };
};
```

```
class Vater2
{
 protected:
    int nummer;
 public:
    Vater2 () {
       nummer = 1;
    };
    virtual void show (void) {
       printf ("\n\nIch bin vom Typ Vater2");
       printf ("\nnummer hat den Wert: %d", nummer);
    };
};

struct Sohn2 : public Vater2
{
 public:
    Sohn2 () : Vater2 () {
  };
    virtual void show (void) {
       printf ("\nIch bin vom Typ Sohn2");
       printf ("\nnummer hat den Wert: %d", nummer);
    };
};

void main (void)
{
  Sohn1 * zeiger1 = new Sohn1;
  Sohn2 * zeiger2 = new Sohn2;
  printf ("\n\n");
  zeiger1->show ();
  zeiger2->show ();

  Vater1 * zeiger3 = new Sohn1;
  Vater2 * zeiger4 = new Sohn2;
  printf ("\n\n");
  zeiger3->show ();       //#1 welche Funktion wird aufgerufen ???
  zeiger4->show ();       //#2 welche Funktion wird aufgerufen ???

  delete zeiger1;
  delete zeiger2;
}
```

Das Programm gibt aus:

```
Ich bin vom Typ Sohn1
Zahl hat den Wert: 1
Ich bin vom Typ Sohn2
nummer hat den Wert: 1


Ich bin vom Typ Vater1
Zahl hat den Wert: 1
Ich bin vom Typ Sohn2
nummer hat den Wert: 1
```

Wie an der Stelle #1 erkennbar wird, wird bei Vater1 und Sohn1 statisch gebunden, da das Schlüsselwort virtual nicht verwendet wird. An der Stelle #2 wird ersichtlich, daß bei der Verwendung von virtual in Abhängigkeit vom Typ des Objektes die richtige Methode verwendet wird. Die Auswahl der Methode erfolgt zur Laufzeit über die Virtuelle-Methoden-Tabelle.

Im nächsten Beispiel wird alles noch plastischer. Aus dem Programmcode können Sie nicht erkennen, worauf der Zeiger `ptr` überhaupt zeigt. Dies wird erst zur Laufzeit durch den Bediener des Programms entschieden.

Hier das Beispiel:

```
#include <stdio.h>
#include <string.h>
// Datei: virt2.cpp

class alpha
{
   int zahl;
 public:
   alpha (int par) {
   zahl = par;
   };
   alpha () {
   zahl = 0;
   };
   virtual void print (void) {
   printf ("\n\n Die Zahl ist: %d", zahl);
   };
};

class beta : public alpha
{
   char zeichen;
 public:
   beta (char par1, int par2):alpha(par2) {
   zeichen = par1;
   };
   virtual void print (void) {
   printf ("\n Das Zeichen ist: %c", zeichen);
   };
};

void main (void)
{ char buffer [10];
  alpha * ptr;
  alpha * ptralpha = new alpha (3);
  beta * ptrbeta = new beta ('3',10);
  printf ("\nGib ein, welches Objekt ausgedruckt werden soll: \
  [alpha, beta]: ");
  gets (buffer);

  if ((strcmp (buffer,"alpha")== 0))
```

```
      ptr = ptralpha;
  else if ((strcmp (buffer,"beta")== 0))
      ptr = ptrbeta;
  ptr -> print ();
  delete ptralpha;
  delete ptrbeta;
}
```

Die folgenden beiden Dialoge wurden geführt:

Dialog 1:

```
Gib ein, welches Objekt ausgedruckt werden soll:   [alpha, beta]:  alpha

 Die Zahl ist: 3
```

Dialog 2:

```
Gib ein, welches Objekt ausgedruckt werden soll:   [alpha, beta]:  beta

 Das Zeichen ist: 3
```

Zeiger auf Objekte vom Typ einer Basisklasse können auch auf Objekte aller davon abgeleiteten Klassen zeigen. Wird also mit Zeigern und Referenzen auf Klassen gearbeitet, so kann unter Umständen beim Kompilieren noch nicht feststehen, auf welches Objekt (der Basisklasse oder der abgeleiteten Klasse) der Zeiger zur Laufzeit zeigt. Wie schon gesagt, daß Schlüsselwort `virtual` sorgt für das dynamische oder späte Binden zur Laufzeit.

Folgende Spielregeln sind im Umgang mit virtuellen Elementfunktionen zu beachten:

- Alle Ausprägungen einer virtuellen Methode in den verschiedenen Klassen müssen in Anzahl und Typen (und Reihenfolge) ihrer formalen Parameter übereinstimmen. Dies ist notwendig, da alle Aufrufe aus einem einzigen Aufruf im Quelltext generiert werden.

- Die Nachkommen müssen eine virtuelle Methode nicht überladen. Sie können ohne weiteres die Methode aus der Basisklasse übernehmen, falls diese ihren Zweck erfüllt. **Bis auf den Konstruktor** kann jede Elementfunktion als virtuell deklariert werden; insbesondere auch (interne) Operatorfunktionen und Destruktoren.

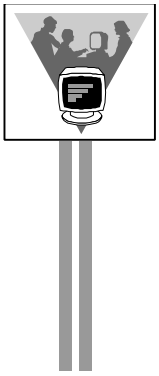
Zum Abschluß dieses Themas noch ein etwas komplexeres Beispiel:

Nehmen wir an, daß Sie verschiedene Objekte in einer linearen Liste (oder einer anderen dynamische Struktur) verwalten. Der Zeiger auf die einzelnen Objekte innerhalb der Liste sei ein Zeiger auf den Urahnen all Ihrer Objekte, der in der Hierarchie ganz oben steht. Damit kann er auf alle seine Nachkommen zeigen, die einzige Möglichkeit, eine solche Struktur aufzubauen ! Sollte das jetzt etwas verworren klingen, betrachten Sie bitte kurz das untenstehende Beispiel, bevor Sie weiterlesen. Dort ist eine sehr einfache Hierarchie mit drei Generationen realisiert. Nun gehen Sie die Liste vom Anfang bis zum Ende durch, und rufen für jedes eingehängte Objekt die Funktion `show( )` auf -- was geschieht ?

```
//
// Beispiel über virtuelle Funktionen
//
#include <stdio.h>
#include <stdlib.h>

class Eins
{
 public:
   int zahl;
   virtual void show(void);
};

class Zwei : public Eins
{
 public:
   virtual void show(void);
};

class Drei  : public Zwei
{
 public:
   virtual void show(void);
};

void Eins::show(void)
{
  printf ("\nIch bin von der Klasse Eins. Zahl hat den Wert %d", zahl);
}

void Zwei::show(void)
{
  printf("\nIch bin von der Klasse Zwei. Zahl hat den Wert %d", zahl);
}

void Drei::show(void)
{
  printf("\nIch bin von der Klasse Drei. Zahl hat den Wert %d", zahl) ;
}

class Listenelement
{
 public:
   Eins * zeiger;
   Listenelement * next;
```

```
};

#define link(typ)              \
runner=new Listenelement;      \
runner->zeiger=new typ;        \
runner->zeiger->zahl=fill;     \
runner->next=root;             \
root=runner;                   \
break

void main(void)
{
  Listenelement *root= NULL, * runner = NULL;

  int dice;
  printf("\n\nVirtual Member Functions: Generating List...");
  for (int fill=0; fill<10; fill++)
  {
     dice=rand() & 0x03;
     switch (dice)
     {
     case 1: link(Eins);
     case 2: link(Zwei);
     case 3: link(Drei);
     }
  }

  printf("done\n");
  runner=root;
  while (runner)
  {
    runner->zeiger->show();
    runner=runner->next;
  }
}
```

Die Ausgabe des Programmes ist:



```
Virtual Member Functions: Generating List...done

Ich bin von der Klasse Zwei. Zahl hat den Wert 9
Ich bin von der Klasse Drei. Zahl hat den Wert 7
Ich bin von der Klasse Drei. Zahl hat den Wert 6
Ich bin von der Klasse Eins. Zahl hat den Wert 5
Ich bin von der Klasse Zwei. Zahl hat den Wert 3
Ich bin von der Klasse Zwei. Zahl hat den Wert 2
Ich bin von der Klasse Zwei. Zahl hat den Wert 1
Ich bin von der Klasse Zwei. Zahl hat den Wert 0
```

Das Beispiel realisiert eine triviale Hierarchie aus 3 Objekttypen, die sich lediglich in der Art der Anzeige des Index durch die virtuelle Funktion `show()` unterscheiden.

Das Datenelement `zahl` wird von der Basisklasse `Eins` an die beiden folgenden Klassen vererbt. Beachten Sie ganz unten die Klasse `Listenelement` zur Realisierung einer einfach-rückwärtsverketteten Liste und das Makro zum Erzeugen und Einhängen eines neuen dynamischen Objektws. Zu Beginn des Kapitels über OOP hatten wir bemerkt, daß man für Definitionen von Methoden außerhalb der Klassendeklaration den Scope-Operator benutzen sollte, um die Methode eindeutig der Klasse zuzuordnen. Was am Anfang wie eine Empfehlung aussah, wird spätestens mit dem Einsatz von virtuellen Methoden zum Muß. Sie tragen ja alle denselben Namen und haben dieselbe Parameterliste. Vergleichen Sie dazu die externen Definitionen der `show()` Funktionen.

Das Hauptprogramm deklariert zwei Zeiger zur Behandlung der linearen Liste, generiert anschließend in einer Schleife 10 Objekte und hängt diese in die Liste ein. Der Typ dieser Objekte wird zufällig bestimmt. Das Makro dient dabei lediglich zur Einsparung von Schreibarbeit bei der Realisierung des immer gleichen Einhängvorgangs. Anschließend wird mit dem Zeiger "runner" die Liste einmal von hinten nach vorne durchgearbeitet, und die `show()` Elementfunktion der eingehängten Objekte aufgerufen.

Testen Sie das Programm zuerst in der vorliegenden Form, und führen Sie dann einen erneuten Probelauf durch, nachdem Sie das Schlüsselwort `virtual` aus der Klassendeklaration von `Eins` entfernt haben. Vergleichen Sie die Ergebnisse. Versuchen Sie auch ruhig einmal, eine der obenstehenden Spielregeln zu verletzen und machen Sie sich ihren Hintergrund klar.

9.3 Abstrakte Basisklassen

In der Tat ein abstrakter Begriff. Unter einer abstrakten Basisklasse versteht man eine Klasse, die lediglich dazu dient, gemeinsame Eigenschaften anderer Klassen zum Zwecke der Vererbung in sich zu vereinigen. Von ihr werden dann die eigentlich interessanten Klassen abgeleitet.

Von einer abstrakten Basisklasse selbst können keine Instanzen (d.h. Objekte) gebildet werden.

Eine abstrakte Basisklasse enthält mindestens eine abstrakte Funktion (pure virtual function), deren Funktionsrumpf leer ist.

Man kann also von jeder Klasse Objekte erzeugen, solange eine solche Klasse nicht mindestens eine rein virtuelle (abstrakte) Funktion enthält.

Solche abstrakten Funktionen dienen nur dazu, die Form der entsprechenden Funktion bei den Nachkommen festzuschreiben.

Eine abstrakte Funktion (pure virtual function) wird gekennzeichnet durch den Zusatz "= 0" (siehe folgendes Beispiel):

```
/* ************************************************************** */
/* Definition der abstrakten Basisklasse GraphObj                 */
/* ************************************************************** */

class GraphObj {

  protected:

    int x;               // x Koordinate (genaue Bedeutung siehe spaeter)
    int y;               // y Koordinate (genaue Bedeutung siehe spaeter)
    int farbe;           // Farbe
    char name [20]  ;    // Name
    BOOLEAN sichtbar;    // ist TRUE, wenn das Objekt sichbar ist

  public:

    GraphObj();              // Standard -Konstruktor
    GraphObj(int,
             int,
             int,
             const char *);  // Konstruktor mit 4 Parametern
    virtual ~GraphObj() {}   // virtueller Destruktor

    virtual void setPosition(int n_x, int n_y);
    virtual void setFarbe(int n_farbe);

    virtual void zeichne(void) = 0;    // rein virtuelle Methoden
    virtual void loesche(void) = 0;    // (abstrakte Methoden)
                                // fuer die nur der Kopf  (Schnittstelle)
                                // aber kein Code innerhalb der Klasse
                                // angegeben werden kann.
                                // Da die Klasse noch keine volle
                                // Funktionalitae t zu Verfuegung stellt,
                                // und somit keine Objekte (Instanzen)
                                // dieser Klasse erzeugt werden koennen,
                                // wird sie als abstrakte Basisklasse
                                 // bezeichnet.
    void verschiebe(int, int);
};
```

Von der Klasse `GraphObj` können keine Objekte erzeugt werden. Die Funktion `zeichne` ist eine abstrakte Funktion. In den abgeleiteten Klassen wird diese Funktion definiert. Die Schnittstelle ist jedoch bereits festgelegt.

Kleines Glossar

Abgeleitete Klasse: *engl. derived class*

Eine abgeleitete Klasse wird aus einer Basisklasse abgeleitet. Eine abgeleitete Klasse hat Zugriff auf alle Elemente (Daten und Funktionen) der Basisklasse, die in der Basisklasse nicht als private definiert wurden.

Abstrakter Datentyp: *engl. abstract data type*

Ein abstrakter Datentyp ist ein Softwaremodell, das einen Wertebereich und die auf dem Wertebereich zulässigen Operationen festlegt. Streng genommen ist die Definition der zulässigen Operationen ausreichend für die Definition des Datentyps. Hinter dem Konzept des abstrakten Datentyps steckt das Konzept der Trennung der Einzelheiten der Repräsentation der Daten von den abstrakten Operationen, die auf den Daten definiert sind.

In C++ können Klassen für die Realisierung abstrakter Datentypen verwendet werden.

Abstrakte Basisklasse: *engl. abstract base class*

Eine abstrakte Basisklasse in C++ dient nur als Ausgangspunkt einer Ableitung. Es kann keine Objekte dieser abstrakten Basisklasse geben. Abstrakte Basisklassen in C++ müssen mindestens eine rein virtuelle Funktion enthalten.

Abstrakte Funktion (rein virtuelle Funktion): *engl. pure virtual function*

Eine als virtual markierte Funktion einer abstrakten Basisklasse. Der Funktionskörper einer abstrakten Funktion ist leer. Abstrakte Funktionen können nicht ausgeführt werden, da es keine Objekte zu abstrakten Basisklassen gibt. Abstrakte Basisklassen dienen nur zur logischen Strukturierung, um einen gemeinsamen Vater für konkrete Klassen zu haben.

Basisklasse: *engl. base class*

Die Elternklasse (Vaterklasse, Mutterklasse), aus der Eigenschaften (Daten, Funktionen) in die abgeleiteten Klassen (Unterklassen) vererbt werden. Eine Klasse kann in C++ mehrere Basisklassen haben (Mehrfachvererbung).

Bezugsrahmen (Gültigkeitsbereich): *engl. scope*

Der Gültigkeitsbereich (Bezugsrahmen) einer Deklaration ist jener Bereich des Programmtextes, für den diese Deklaration ihre Bedeutung hat. In diesem Bereich ist das Objekt verfügbar. In C++ kann in drei Bereichen auf ein Objekt Bezug genommen werden: lokal (innerhalb eines Blockes), global (innerhalb einer Datei) oder innerhalb einer Klasse.

Bindung: *engl. binding*

Darunter versteht man den Zeitpunkt, zu dem Botschaften und Funktionen einem ausführenden Programmteil zugeordnet werden. Die Bindung eines Funktionsnamens an eine Funktionsadresse kann entweder zur Compile-Zeit (*early binding*, *statische Bindung, frühe Bindung*) oder zur Laufzeit (*late binding*, *dynamische Bindung, späte Bindung*) erfolgen. Die Bindung zur Compile-Zeit wird in C++ dadurch errreicht, daß der Aufruf einer Elementfunktion immer im Zusammenhang mit einem Datenobjekt einer Klasse erfolgt. Wenn eine Memberfunktion als virtuelle Funktion deklariert wird, wird bei Verwendung von Zeigern und Referenzen erst zur Laufzeit entschieden (Bindung zur Laufzeit), welche Funktion aufgerufen wird. In Abhängigkeit von der Klasse des Objektes wird die dazugehörige Funktion aufgerufen.

call by reference

Eine der Methoden, um Argumente an Funktionen zu übergeben. Hier wird die Adresse des Arguments an die Funktion übergeben. Der Vorteil dieses Übergabemechanismus liegt darin, daß die aufgerufene Funktion den Argumentenwert manipulieren kann.

call by value

Ebenfalls eine Methode, um Argumente an Funktionen zu übergeben. Hier wird allerdings eine Kopie des Argumentenwertes an die Funktion übergeben. Die aufrufende Funktion kann somit den Argumentenwert nicht manipulieren.

Default-Konstruktor
gleichbedeutend mit Standard-Konstruktor (siehe dort)

Definition

Deklarationen sind zugleich auch Definitionen, wenn bei der Deklaration direkt auch der entsprechende Speicherplatz für das bezeichnete Objekt reserviert wird.

Deklaration

Eine Deklaration deklariert einen Namen in einem Bezugsrahmen (Scope). Bei der Deklaration muß für den Namen ein Typ angegeben werden.

Beispiel: Extern-Deklaration einer Variablen
Soll eine externe Variable verwendet werden, bevor sie definiert wird, oder wird sie in einer anderen Quelldatei definiert, als in der, in der sie benutzt wird, dann ist eine extern-Deklaration erforderlich. Mit der Deklaration werden die Eigenschaften einer Variablen (insbesondere der Datentyp) festgelegt. In **genau einem** Programm-Modul

muß eine solche Variable dann **ohne extern** auf der obersten Ebene, d.h. außerhalb aller Funktionen, definiert sein. Die Definition sorgt über diese Eigenschaften hinaus für den Speicherplatz.

Eine Deklaration ist keine Definition, wenn:

?? sie das Schlüsselwort extern enthält (eine Initialisierung einer solchen Variablen ist nicht zulässig, eine so deklarierte Funktion enthält keinen Funktionsrumpf)

?? nur eine Funktion deklariert wird, ohne den Funktionsrumpf anzugeben (Funktionsprototyp)

?? es die typedef-Deklaration ist, bei der vom Programmierer ein eigener Name für einen Datentyp festgelegt wird

?? es sich um die Deklaration einer Klasse handelt

?? es sich um die Deklaration eines statischen Members in einer Klassendeklaration handelt.

Destruktor
Ein Destruktor (Zerstörer) ist eine spezielle Methode (Funktion), die dafür sorgt, daß die von den Konstruktoren belegten Speicherbereiche wieder freigegeben werden. Ein Destruktor wird automatisch aufgerufen, wenn der Gültigkeitsbereich eines Klassenobjekts verlassen wird. Auch bei Verwendung des Operators delete wird ein Destruktor aufgerufen. Der Name eines Destruktors ist in C++ gleich dem Klassennamen mit vorgesetzter Tilde ~.

Freund: *engl. friend*

Dem Prinzip des Information Hiding entsprechend ist der Zugriff auf einzelne Datenelemente eines Klassen-Objektes streng geregelt und wird in manchen Fällen bewußt verhindert. Bei eng verwandten Klassen, die teils miteinander agieren müssen, kann es nützlich sein, einen Zugriff auf die privaten Datenelemente zu erlauben. Solche zugriffsberechtigten Funktionen können im Rahmen der Klassendefinition als Freund (friend) vereinbart werden und dürfen demnach auf die Datenelemente der Klasse zugreifen. Die Eigenschaften *friend* muß von der Klasse verliehen werden, auf deren Mitglieddaten zugegriffen werden soll.

Geheimnisprinzip: *engl. information hiding*

Das Geheimprinzip sorgt dafür, daß die internen (privaten) Strukturen eines Objekts einer Klasse nach außen hin unzugänglich sind. Nur der Implementierer einer Klasse kennt normalerweise die internen Strukturen eines Objekts. Implementierung und Schnittstellen werden voneinander getrennt. Das Geheimnisprinzip sichert somit die Integrität einer Klasse und ihrer Objekte, da wichtige Daten von außen nicht mehr zugänglich ind. In C++ kann das Geheimnisprinzip durch abgestufte Sichtbarkeitsbereiche (*private, protected, public*) gelockert werden.

Gültigkeitsbereich (Bezugsrahmen): *engl. scope*

Der Gültigkeitsbereich (Bezugsrahmen) einer Deklaration ist jener Bereich des Programmtextes, für den diese Deklaration ihre Bedeutung hat. In diesem Bereich ist das Objekt verfügbar. In C++ kann in drei Bereichen auf ein Objekt Bezug genommen werden: lokal (innerhalb eines Blockes), global (innerhalb einer Datei) oder innerhalb einer Klasse.

Information Hiding

siehe Geheimnisprinzip

Instanz: *engl. instance*

Eine Instanz ist eine konkrete Ausprägung einer Klasse, also ein Objekt. Das Verhältnis von Instanz zu Klasse entspricht dem von Variable zu Datentyp.

Kapselung: *engl. encapsulation*

Eines der wichtigsten Konzepte der objektorientierten Programmierung. Darunter versteht man die sinnvolle Zusammenfassung von Daten und Funktionen in einer gemeinsamen Kapsel (der Klasse). Es besteht in diesem Falle nicht die Trennung zwischen Daten und Funktionen wie in der prozeduralen Programmierung z.B. in C.

Die Daten einer Kapsel können im Idealfall nur durch die Funktionen der Kapsel manipuliert werden. In diesem Fall schließt Kapselung das Geheimnisprinzip ein, da die Daten nicht frei gelegt werden, sondern nur durch die Funktionen der Kapsel verändert werden können. Die Struktur der Daten ist dadurch nach außen nicht sichtbar.

Klasse: *engl. class*

Eine Klasse ist die abstrakte Beschreibung einer Menge gleich strukturierter Datenobjekte. Die Klassendefinition enthält die Beschreibung der Datenelemente (Mitglieddaten, engl. *member data*) , die jedes Objekt dieser Klasse enthält, und der Mitgliedfunktionen (Methoden, engl. *member functions*), die auf jedes Objekt dieser Klasse anwendbar sind, in einer gemeinsamen Kapsel. Herausgehobene Mitgliedfunktionen sind die Konstruktoren und Dekonstruktoren einer Klasse. Abstrakte Datentypen können durch Klassen realisiert werden.

Konstruktor
Ein Konstruktor (Aufbauer) ist eine Routine, die automatisch aufgerufen wird, wenn der Gültigkeitsbereich eines Klassenobjekts betreten wird. In C++ ist der Name eines Konstruktors gleich dem Klassennamen.

Kopierkonstruktor: *engl. copy constructor*

Besondere Form eines Konstruktors, die besonders bei der Argumentübergabe *by value* von Objekten benötigt wird.

Mehrfachvererbung: *engl. multiple inheritance*

Eine abgeleitete Klasse kann einen direkten Vorgänger (Basisklasse) haben oder mehrere direkte Vorgänger. Bei Mehrfachvererbung wird im abgeleiteten Objekt für jede Basisklasse ein Basisobjekt als Kopie abgelegt.

Member-Funktion

Anderes Wort für Mitgliedfunktion

Methode

Anderes Wort für Mitgliedfunktion.

Mitglied-Funktion

Operation, die auf ein Objekt angewandt werden darf. Sie ist in der dem Objekt zugehörigen Klasse festgelegt.

Objekt

Eine Instanz (Ausprägung) einer Klasse. Das Verhältnis von Objekt zu Klasse entspricht dem von Variable zu Datentyp.

OOD (Object Oriented Design)

Objektorientiertes Design. Innerhalb des Software Engineering die Disziplin, die sich mit dem Entwurf objektorientierter Programme beschäftigt.

OOP (Object Oriented Programming)

Objektorientierte Programmierung, die Programmierung in einer OOPL unter Berücksichtigung objektorientierter Methoden. (Die bloße Verwendung von C++ reicht nicht aus!)

OOPL: *engl. object oriented programming language*

Objektorientierte Programmiersprache. Was eine OOPL auszeichnet, ist nicht unumstritten. So wird z.B. die Meinung vertreten, eine OOPL verdiene diese Bezeichnung, wenn sie vier Eigenschaften aufweise: Abstraktion, Kapselung, Vererbungskonzept, Polymorphismus. Modula-2 beispielsweise ist nach dieser Klassifikation keine OOPL.

OOPS: *engl. object oriented programming system*

Ein objektorientiertes Entwicklungssystem bietet eine integrierte Entwicklungsumgebung, die meist aus Compiler, Editor, Debugger, Objektdatenbank und weiteren Werkzeugen (zum Beispiel für *rapid prototyping*) besteht. Die derzeitigen OOPS sind meist grafik- und fensterorientiert. Typische OOPS sind Smalltalk und TRELLIS (Digital Equipment). Ziel einer solchen Umgebung ist die Beschleunigung des Programmier- und Designvorgangs. Dazu werden unter Umständen Hunderte von Musterklassen bereitgestellt.

Polymorphismus: *engl. polymorphism*

Unter Polymorphismus (Vielgestaltigkeit) versteht man die Eigenschaft, verschiedene Zwecke trotz Verwendung gleicher Namen zu erfüllen. In C++ wird Polymorphismus durch mehrere Verfahren erreicht: Funktionen können überladen werden, Operatoren können überladen werden, virtuelle Funktionen können eingesetzt werden. Eine engere Definition verwendet den Begriff Polymorphismus nur im Zusammenhang mit virtuellen Funktionen.

Rein virtuelle Funktion (abstrakte Funktion): *engl. pure virtual function*
siehe abstrakte Funktion

Standard-Konstruktor

Wird automatisch immer dann benutzt, wenn ein Konstruktor nicht explizit definiert wurde. Explizit definiert werden kann auch der Standard-Konstruktor selbst. Die Konstruktion muß so erfolgen, daß der Standard-Konstruktor ohne Parameter aufrufbar ist.

Stream (*engl.)*

C++ verwendet zur Ein- und Ausgabe das Stream-Konzept. Streams sind Klassen, die in *stream.h* (ab Version 2.0 *iostream.h*) definiert werden. Es gibt in C++ nur wenige vordefinierte Klassen, darunter *istream* und *ostream*. Die Standardströme *cin*, *cout* und *cerr* sind vordefinierte Objekte dieser Klasse. Für die Eingabe oder Ausgabe über diese Ströme sind in *stream.h* und *iostream.h* die Operatoren << und >> überladen worden. Das Stromkonzept funktioniert gleichermaßen für Bildschirm und Tastatur wie für Dateien. Neben dem Stromkonzept können auch die Ein/Ausgabe-Möglichkeiten von C weiter verwendet werden, dies sollte jedoch vermieden werden.

Stroustrup

Bjarne Stroustrup, Entwickler bei AT&T, gilt als Erfinder von C++. Sein Buch *The C++ Programming Language* hat für C++ einen ähnlichen Stellenwert wie Kernighan & Ritchie für C.

Überladen: *engl. overload*

In C++ ist das Überladen von Funktionen und Operatoren möglich. Das Konzept des Überladens ist eine der Vorausetzungen für Polymorphismus. Das Überladen von Operatoren wird vor allem dazu verwendet, bekannte Operatoren auf neue Datentypen (realisiert durch Klassen) anzuwenden. Die Semantik (Bedeutung) überladener Operatoren wird durch benutzereigene Operatorfunktionen definiert.

Vererbung: *engl. inheritance*

Das Vererbungskonzept gilt für die Beziehung zwischen Basisklasse und abgeleiteten Klassen. In abgeleiteten Klassen werden die Mitglieddaten der Basisklasse als Kopie bereitgestellt (vererbt) - was aber nicht heißt, daß unkontrolliert darauf zugegriffen werden darf. Die ererbten Mitgliedfunktionen können ebenfalls kontrolliert (Schlüsselwort *public* oder *protected*) mitverwendet werden.

Virtuelle Basisklasse: *engl. virtual base class*

Virtuelle Basisklassen garantieren bei Mehrfachvererbung, daß Basisobjekte, die über mehrere Ableitungspfade geerbt wurden, nur einmal im abgeleiteten Objekt erscheinen.

Virtuelle Funktionen: *engl. virtual function*

Eine virtuelle Funktion erlaubt in C++ die Bindung von Funktionsname an eine Funktionsadresse zur Laufzeit. Virtuelle Funktionen sind in C++ an das Konzept der abgeleiteten Klassen gebunden. Dabei definiert eine Basisklasse die virtuellen Funktionen, während die aktuellen (tatsächlich aufgerufenen) Funktionen durch die abgeleiteten Klassen bereitgestellt werden.

Erfolgt der Aufruf einer virtuellen Funktion mit einem Datenobjekt, so ist aus dem Typ des Datenobjekts ersichtlich, welche Funktion aufgerufen werden soll. Der Compiler stellt dies bereits beim Compilieren fest (statische Bindung).

Wenn eine Memberfunktion als virtuelle Funktion deklariert wird, wird bei Verwendung von Zeigern und Referenzen erst zur Laufzeit entschieden, welche Funktion aufgerufen wird. In Abhängigkeit von der Klasse des Objektes wird die dazugehörige Funktion aufgerufen.

Wiederverwendbarkeit: *engl. reusability*

Die Wiederverwendbarkeit von Software ist eines der Hauptziele der OOP. Sie wird erreicht durch verschiedene Maßnahmen wie die Verwendung von inhaltlich klar getrennten Quellcodemodulen, durch die Verwendung von Musterklassen, durch die Trennung von Implementation und Schnittstellen.

Literatur

Folgende Zeitschriften und Bücher wurden bei der Erstellung dieses Kurses verwendet:

1) Wilhelm Bolkart, Programmiersprachen der vierten und fünften Generation

2) Werner Achtert, Das große Buch zu C++, Data Becker

3) Martin Aupperle: Borland C++ Einsteigerkurs, Jetzt objektorientiert entwickeln, Borland Magazin

4) Falko Bause, Wolfgang Tölle, Einführung in die Programmiersprache C++, Vieweg

5) Falko Bause, Wolfgang Tölle: DAS VIEWEG-BUCH ZU C++ Version 3, Vieweg

6) Michael Götz, Diplomarbeit FHTE SS 93

7) Matthias Haun, Profi-Workshop C++, IWT

8) Edgar Huckert: Programmieren in C++, Markt & Technik

9) Bjarne Stroustrup: The C++ Programming Language, Addison Wesley

10)Ellis, B. Stroustrup, The Annotated C++ Reference Manual', Addison Wesley

11)Scott Meyers, Effektiv C++ programmieren

12)Rumbaugh, M. Blaha, W. Prmerlani, F. Eddy, W. Lorensen, Objektorientiertes Modellieren und Entwerfen, Hanser

13)Josuttis, Objektorientiertes Programmieren in C++, Addison-Wesley