Vergleich von Delphi u. VC++

http://kickme.to/tiger/

Vergleich von Delphi und Visual C++ unter den Aspekten der objektorientierten Spracheigenschaften und der Klassenbibliotheken

Die folgende Arbeit entstand im Rahmen meiner Diplomarbeit an der Fachhochschule für Technik Esslingen im Sommer 1996. Die Arbeit betreuten Prof. W. Walser und Prof. Dr. H. Kull. Für zahlreiche Hinweise und Ratschläge bedanke ich mich besonders bei Prof. Werner Walser. Die Veröffentlichung im Internet erfolgt mit seiner Zustimmung.

Jan - Michael Strube

Kornwestheim, den 1.1.1997

Um das ganze Dokument in gepackter Form herunter zu laden, bitte hier klicken.

Inhaltsverzeichnis

1. Das Betriebssystem Windows mit seinem "Win32 - API"

2. Syntax und Eigenschaften der Sprachen Visual C++ und Object Pascal

2.1 Grundideen objektorientierter Entwicklung, der Analyse und des Designs

2.2 Umsetzung des OOP-Gedankens in C und Pascal

2.3 Sprachumfang beider Systeme im Vergleich

2.3.1 Lexikalische Elemente und Operatoren

2.3.2 Standardtypen

2.3.3 Variablen und Konstanten

2.3.4 Anweisungen

2.3.5 Programmstruktur

2.3.6 Klassen und Objekte

2.3.6.1 Struktur und Realisierung

2.3.6.2 Vererbung (Inheritance)

2.3.6.3 Zuweisungskompatibilität

2.3.6.4 Methoden und spezielle Felder

2.3.6.5 Konstruktoren und Destruktoren

2.3.6.6 Metaklassen

2.3.6.7 Friends

2.3.6.8 Überladen von Operatoren

2.3.6.9 Klassen-Schablonen

2.3.6.10 Eigenschaften - Properties

2.3.6.11 Laufzeit-Typinformationen

3. Übersicht über die bei Visual C++ und Delphi mitgelieferten Bibliotheken

3.1 Laufzeitbibliotheken - Run-Time Libraries

3.2 Klassenbibliotheken "Microsoft Foundation Classes" (MFC) und "Visual Component Library" (VCL)

3.2.1 Erweiterte Stringtypen

3.2.2 Architektur der Klassenbibliotheken

3.2.3 Fensterbasierende Entwicklung

4. Zusammenfassung

Diese Seite wurde bisher  mal besucht.

 Zurück zur Homepage

1. Das Betriebssystem Windows mit seinem "Win32 - API"

Windows stellte in früheren Versionen nur einen Betriebssystem-Aufsatz dar, der dem Anwender IBM-kompatibler Personal-Computer eine graphische Benutzeroberfläche zur Verfügung stellte. Entwickelt von der Firma Microsoft, wies Windows schon damals entscheidende Merkmale wie standardisierte, konsistente Programmoberfläche, die Möglichkeit zur gleichzeitigen Ausführung mehrerer Anwendungen und die Bereitstellung geräteunabhängiger Schnittstellen auf. Eine für Programmierer frei zugängliche Schnittstelle, das Windows-API (Application Program Interface), stellt sich aus der Sicht eines Entwicklers als eine sehr umfangreiche Funktions-Bibliothek dar. Erst die Möglichkeit des Zugriffs auf eine derartige Bibliothek schafft die Voraussetzung dafür, daß Windows-Programme ein einheitliches Aussehen und prinzipiell gleichartiges Verhalten aufweisen.

Das ursprünglich auf 16-bit-Technologie beruhende API wurde von Microsoft weiterentwickelt und unter der Kurzbezeichnung "Win32" in verschiedenen Windows-Versionen integriert. In einem von dieser Firma neu entwickelten Betriebssystem, "Windows NT" genannt, wurde es am überzeugendsten implementiert.

Jedes Win32-Programm läuft in einem separaten 32-bit Adreßraum ab. Die Adreßräume einzelner Programme sind vollständig voneinander getrennt, so daß gewollte oder ungewollte Zugriffe auf Adreßbereiche fremder Programme strikt unterbunden werden. Das Win32-API trifft auf große Akzeptanz in der Industrie; selbst zu Windows NT konkurrierende Betriebssysteme, wie etwa IBMs OS/2, integrieren mittlerweile das Win32-API bei sich (bei OS/2 unter dem Namen "Open32").

Das Win32-API setzt sich aus verschiedenen Funktionsgruppen zusammen:

- Fensterverwaltung (User)
- Grafische Geräteschnittstelle (GDI)
- System-Funktionen (Kernel)
- Multimedia-Erweiterungen
- Netzwerk-Erweiterungen
- sonstige Erweiterungen (Open GL, Telefon-API, ...)

Die **Fensterverwaltung** ist für die Erzeugung und Verwaltung von Fenstern zuständig und weist eine ereignisgesteuerte Architektur auf. Eingaben eines Benutzers lösen Ereignisse aus. Das System sendet einer Anwendung die sie betreffenden Ereignisse in Nachrichten-Form. Jede 32-bit Anwendung besitzt eine private Warteschlange (Message-Queue), in der das System alle Nachrichten ablegt (asynchronous input model). Anwendungen lesen diese Warteschlange kontinuierlich aus und können selbst entscheiden, wie sie, bzw. ob sie überhaupt, auf die so zugestellten Nachrichten reagieren wollen.

Windows identifiziert alle im System vorhandenen Fenster durch sogenannte Window-Handles (Typ Hwnd). Generell stellen Handles nur Kennungen dar (32-bit Zahlen), die das System bei der Erzeugung von Windows-Objekten zurückliefert. Handles repräsentieren die ihnen zugrunde liegenden Windows-Objekte.

Aussehen und Verhalten von Standardfenstern kann vom Programmierer verändert werden. Generell veränderte Fenster können im System als neue Fensterklasse angemeldet werden. Das Win32-API stellt bereits standardmäßig eine ganze Anzahl registrierter Fensterklassen zur Verfügung: Schalter (Buttons), Eingabe-Felder (Edit-Fields), ComboBoxen, ListBoxen, ScrollBars, Static-Fenster (zum Anzeigen von Text und Graphik).

Durch das **GDI** (Graphics Device Interface) werden Funktionen zur Verfügung gestellt, die der Grafikausgabe auf Bildschirm, Drucker oder Metadatei dienen. Grafikausgaben erfolgen stets auf einem Geräte-Kontext (Device Context = DC). Ausgaben in Geräte-Kontexte von Fenstern sollten im allgemeinen nur als Reaktion auf das Auftreten ganz bestimmter Nachrichten (wie *WM_PAINT* oder *WM_NCPAINT*) erfolgen.

Die **System-Funktionen** gestatten Anwendungen den Zugriff auf Ressourcen, die in engem Zusammenhang mit dem Betriebssystem stehen. Dazu zählen der Zugriff auf Speicher, Dateisystem und Prozesse. Jedes laufende Programm stellt einen Prozeß dar. Beim Starten eines Prozesses wird vom System ein Primär-Thread erzeugt, der seinerseits eigene Threads erzeugen kann. Windows NT teilt die zur Verfügung stehende Rechenzeit zwischen allen im System angemeldeten Threads auf. Im Multiprozessor-Betrieb ist echte Parallelverarbeitung möglich; im Einprozessor-Betrieb findet eine Quasi-Parallelabarbeitung statt. Jeder Prozeß besitzt einen 4 GByte großen, virtuellen Adreßraum. Alle Threads eines Prozesses besitzen einen eigenen Stack innerhalb dieses Adreßraums. Die Threads eines Prozesses können gemeinsam und gleichzeitig auf globale und static-Variable des Programms zugreifen. Damit der Zugriff auf nur begrenzt vorhandene Ressourcen (wie z.B. globale Variablen) aus den Threads heraus kontrolliert erfolgt, bietet das Win32-API mehrere Möglichkeiten zur Synchronisation an: Kritische Bereiche, Mutex-Objekte, Semaphoren und Ereignisse.

Thread-lokale Variable werden im lokalen Thread-Stack angelegt und sind dadurch vor unerwünschten Zugriffen geschützt.

Es ist in Windows möglich, **dynamisch ladbare Bibliotheken** (DLL = Dynamic Link Libraries) zu entwickeln. Mit ihrer Hilfe lassen sich eigene APIs erstellen, die anderen Programmen zur Verfügung gestellt werden können. DLLs beinhalten hauptsächlich Funktionen und Windows-Ressourcen (Strings, Menüs, Dialoge, Bitmaps, Icons usw.) und können von beliebig vielen Prozessen gleichzeitig benutzt werden. Sobald eine DLL in den Adreßraum eines Prozesses eingeblendet ist, stehen die Funktionen und Ressourcen sämtlichen Threads dieses Prozesses zur Verfügung.

Das Entwickeln graphisch orientierter Windows-Programme ist bei der Verwendung herkömmlicher Sprachen (z.B. von C) sehr aufwendig. Bereits beim Schreiben weniger komplexer Anwendungen entstehen relativ lange Quelltexte. Durch die langen Code-Sequenzen wird der Programmtext schnell unübersichtlich, so daß sich leicht Fehler einschleichen können. Der Einsatz objektorientierter Sprachen und geeigneter Klassenbibliotheken soll dabei helfen, die Komplexität des Window-APIs zu verbergen, ohne daß man jedoch auf dessen große Funktionalität verzichten müßte. Man hofft, durch den Einsatz von Klassenbibliotheken, Entwickler von immer wiederkehrenden Routine-Aufgaben (wie z.B. dem Subclassing einzelner Fenster-Elemente) befreien zu können. Übersichtlicherer, klar strukturierter Code soll zur Senkung der Fehlerrate beitragen.

Anmerkung: Das Win32-API beruht intern auf ANSI-C und ist nicht objektorientiert realisiert worden. Vererbungsmechanismen und virtuelle Funktionen werden von ihm nicht unterstützt. Die in dem Zusammenhang auftretenden Begriffe Klasse (z.B. bei Fenster-Klasse) und Objekt (z.B. bei Zeichenobjekt) haben nichts mit den bei der objektorientierten Programmierung verwendeten Begriffen zu tun.

Zurück zum Inhaltsverzeichnis

Weiter in Kapitel 2

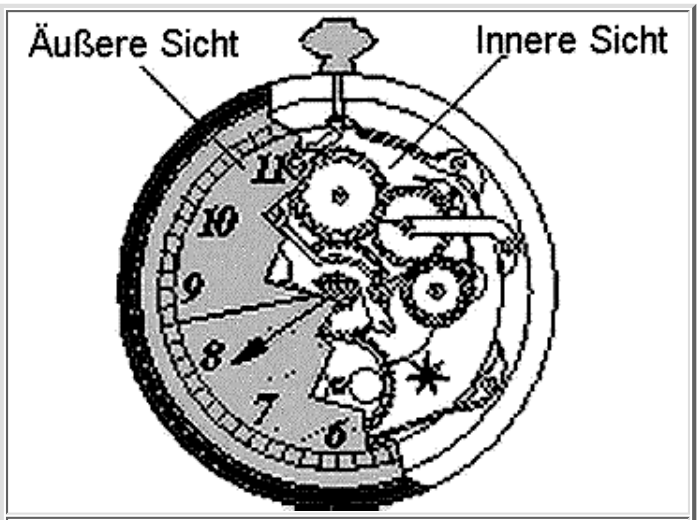
2. Syntax und Eigenschaften der Sprachen Visual C++ und Object Pascal

2.1 Grundideen objektorientierter Entwicklung, der Analyse und des Designs

Während Softwareentwicklung in den 70er Jahren maßgeblich durch die Anwendung des Prinzips der strukturierten Programmierung geprägt war, wurde in den folgenden Jahren immer deutlicher, daß eine Steigerung hinsichtlich Quantität und Qualität nur unter Zuhilfenahme eines ganz neuen Konzepts möglich sein würde, der objektorientierten Programmierung (OOP). Beim bis dahin angewandten strukturierten Ansatz wird eine Aufgabe in immer kleinere Verarbeitungsschritte aufgespalten. Diese Vorgehensweise wird deshalb auch Top-Down-Ansatz genannt. Sie zeichnet sich dadurch aus, daß Daten im Vergleich zu Funktionen eine untergeordnete Rolle spielen. Im Gegensatz dazu entsteht bei der objektorientierten Entwicklung ein Programm nicht um eine Anzahl von Funktionen herum, sondern wird durch die Verwendung von Objekten geprägt. Objekte spiegeln dabei Elemente des Anwendungsbereichs wider. Sie sind Datenstrukturen (genauer gesagt Instanzen derselben), die ein bestimmtes Verhalten aufweisen, welches durch Funktionen innerhalb der Objekte festgelegt wird.

Die Deklaration des Objekts, einschließlich der Festlegung von Datenstruktur, den Funktionen und deren Implementierung wird als **Klasse** bezeichnet. So wie beispielsweise eine Variable vom Typ Integer eine Instanzierung dieses Integertyps darstellt, so ist auch ein Objekt eine Instanz eines bestimmten Klassentyps.

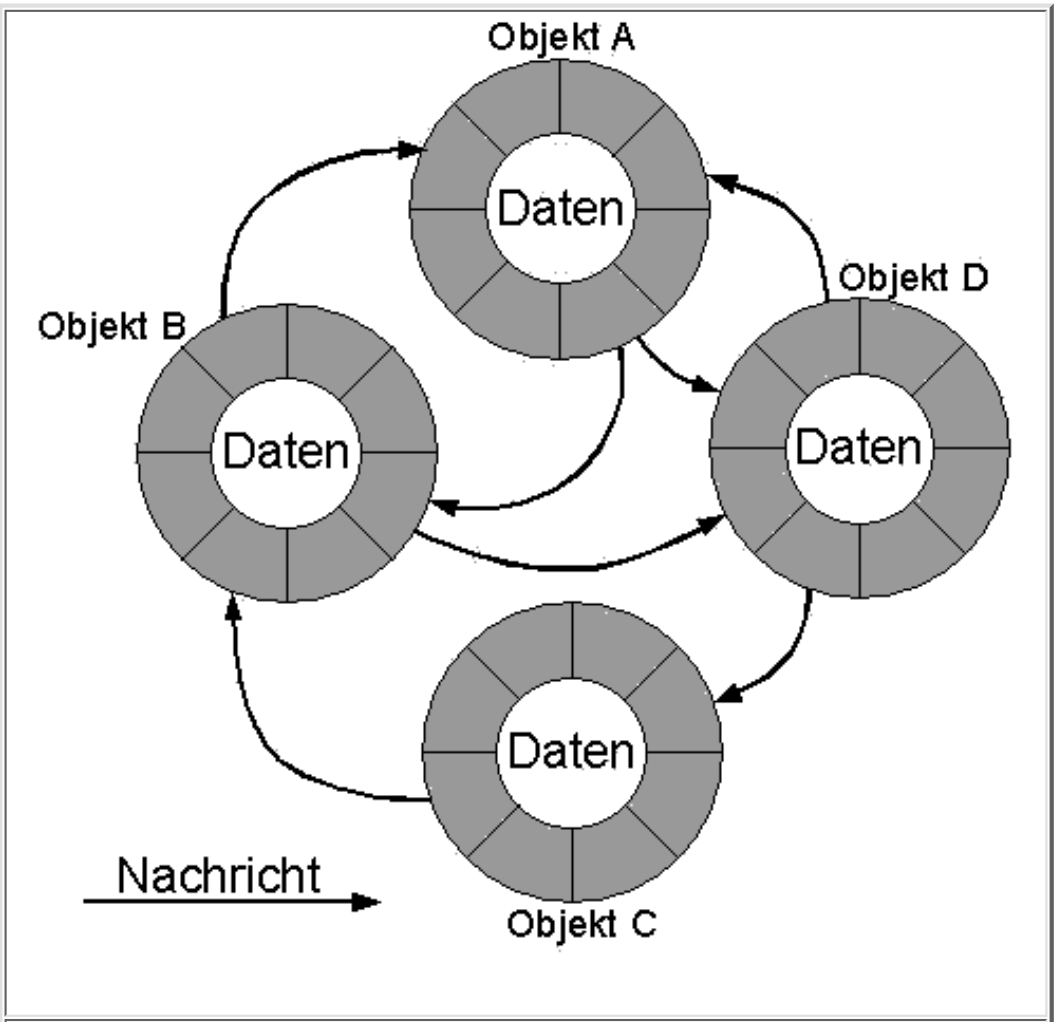
Funktionen in Objekten werden auch als Methoden bezeichnet. Sie verleihen dem Objekt bestimmte Verhaltensweisen und werden oft dazu benutzt, Zugriff auf die Daten im Objekt zu ermöglichen. Auf diese Weise können einerseits die Daten im Objekt vor äußeren Zugriffen geschützt werden und es existiert doch andererseits eine wohldefinierte Schnittstelle nach außen. Diese Zugriffskontrolle, welche eine Beschränkung des Zugriffs auf interne Details darstellt, ist ein wesentliches Merkmal objektorientierter Programmiersprachen und wird unter dem Begriff der **Kapselung** geführt.



In Objekten werden die Details implementiert. Sie liegen geschützt und versteckt im Inneren. Objekte besitzen eine Schnittstelle nach außen. Man unterscheidet äußere Sicht - **Interface** - und innere Sicht - **Implementation**.

Wenn man Objekte und Klassen von innen betrachtet, wird man damit konfrontiert, wie sie zusammengestellt sind, wie sie arbeiten. Betrachtet man sie von außen, wie es ein Anwender tut, interessiert nur ihr Zweck und ihre Leistungsfähigkeit. Man klärt wofür sie da sind und was sie können. Die Kapselung von Variablen in Objekten wird häufig auch "Datenkapselung" und "Information-Hiding"

genannt. Kapselung ist weitgehend auch bei der herkömmlichen, strukturierten Programmierung möglich, wird jedoch bei der objektorientierten Programmierung unter anderem durch die Einführung differenzierter Schutzbereiche besonders gut unterstützt.



In den meisten Programmen wird zur Lösung eines bestimmten Problems nicht nur eins, sondern eine ganze Anzahl von Objekten existieren.

Den Objekten werden bestimmte Verantwortlichkeiten zugewiesen, die sie durch ihre Funktionen erfüllen müssen. Durch Aufgabenverteilung und eine enge Zusammenarbeit wird der Auftrag des Systems ausgeführt. Neben dem Datenzugriff über die öffentlichen Methodenschnittstellen kommunizieren sie miteinander, indem sie Botschaften versenden und empfangen. So beschreibt es Dan Ingalls, einer der Entwickler der OO-Sprache Smalltalk, wie folgt: "Statt eines bitfressenden Prozessors, der Datenstrukturen hin- und herschaufelt, haben wir nun ein Universum von wohlerzogenen Objekten, die sich höflich gegenseitig bitten, zur Erfüllung ihrer jeweiligen Anliegen beizutragen." [1].

Methoden sind also ein Teil eines Objekts und "umlagern" dessen Daten. Allerdings gruppiert man bei der realen, technischen Umsetzung in Rechnern die Methoden nicht tatsächlich mit den Instanzvariablen eines jeden neuen Objekts. Solch ein Vorgehen würde objektorientierte Programme sehr vergrößern und Ressourcen verschwenden. Speicher wird deshalb nur für die Daten, die Variablen eines jeden Objekts, belegt. Es besteht kein Grund, Speicher für Methoden zu allozieren. Das Objekt braucht nur die Zugriffsmöglichkeit zu seinen Methoden, so daß alle Instanzen derselben Klasse auf denselben Funktionen-Pool zugreifen, ihn sich teilen können. Es gibt nur *eine* Kopie der Methoden im Speicher, egal wie viele Instanzen einer Klasse erzeugt werden.

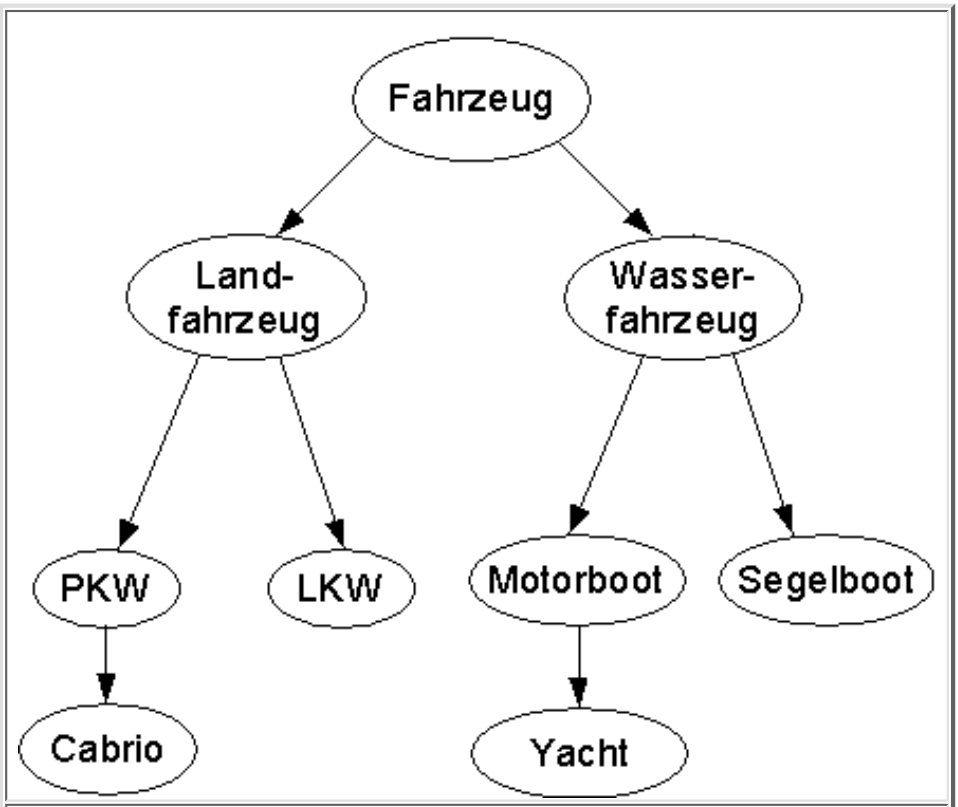
Damit bei Funktionsaufrufen trotzdem eine Beziehung zwischen aufgerufener Methode und aufrufendem Objekt hergestellt werden kann, wird bei jedem Methodenaufruf ein impliziter Parameter an die aufgerufene Funktion übergeben. Dieser Parameter ist ein Zeiger auf die Objektinstanz. Aufgrund der großen Bedeutung dieses Zeigers wird ihm in objektorientierten Sprachen ein fester symbolischer Name zugewiesen. In der Sprache C++ nennt man ihn *this* und in Object Pascal *self*.

Objekte wurden so entworfen, daß sie in erster Linie als Datenbehälter dienen, weil man erkannt hat, daß Datenstrukturen, im Gegensatz zur Funktionsweise einer Anwendung, oft das einzig Verläßliche und Dauerhafte darstellen. Herkömmliche, nicht objektorientierte Programme, müssen bei Änderungen der Anforderungen oft komplett neu- bzw. große Teile umgeschrieben werden. Das zentrale Gerüst von Objekten in einer objektorientierten Anwendung, die Klassenstruktur, kann dagegen in solch einem Fall bestehen bleiben, wenn sie richtig und weitsichtig entworfen wurde. Objektorientierte Programme sind deswegen nicht nur flexibler, sondern auch noch wesentlich besser wartbar. Um einmal erstellte Software-Komponenten in demselben oder anderen Projekten erneut verwenden zu können, muß die Möglichkeit bestehen, die Dienste zu variieren, die ein Modul seinem Klienten zur Verfügung stellt. Diese, unter dem Begriff der **Wiederverwendbarkeit** bekannte, herausragende Eigenschaft objektorientierter Komponenten resultiert aus den Kennzeichen Kapselung, Vererbung, Polymorphismus, Überladen und dynamischen Eigenschaften.

Ein generelles Ziel objektorientierter Programmierung ist es, den Code so zu schreiben, daß er möglichst oft wiederverwendet werden kann. Die Wiederverwendbarkeit wird, ebenso wie bei strukturierter Programmierung, von mehreren Faktoren beeinflußt:

- Zuverlässigkeit und Fehlerfreiheit des Codes
- Vorhandensein einer umfassenden Dokumentation
- Vorhandensein klarer und einfacher Schnittstellen
- Effizienz des Codes
- Umfang der abgedeckten Funktionalität.

Durch **Vererbung** gibt ein Klasse seine Eigenschaften an eine neue Klasse weiter. Neue Klassen können auf Existierenden aufbauen, um bereits vorhandene Eigenschaften in modifizierter oder erweiterter Form zu übernehmen. Man erstellt zunächst weniger spezialisierte, elementare Grundtypen und erzeugt dann darauf aufbauend vererbte Klassen, die neben den grundlegenden noch zusätzliche, besondere Eigenschaften und Verhaltensweisen besitzen.



Beispiel einer Klassenhierarchie mit Superklasse Fahrzeug und den abstrakten Basisklassen Landfahrzeug und Wasserfahrzeug

Vererbte oder auch **abgeleitete** Klassen können drei verschiedene Veränderungen erfahren:

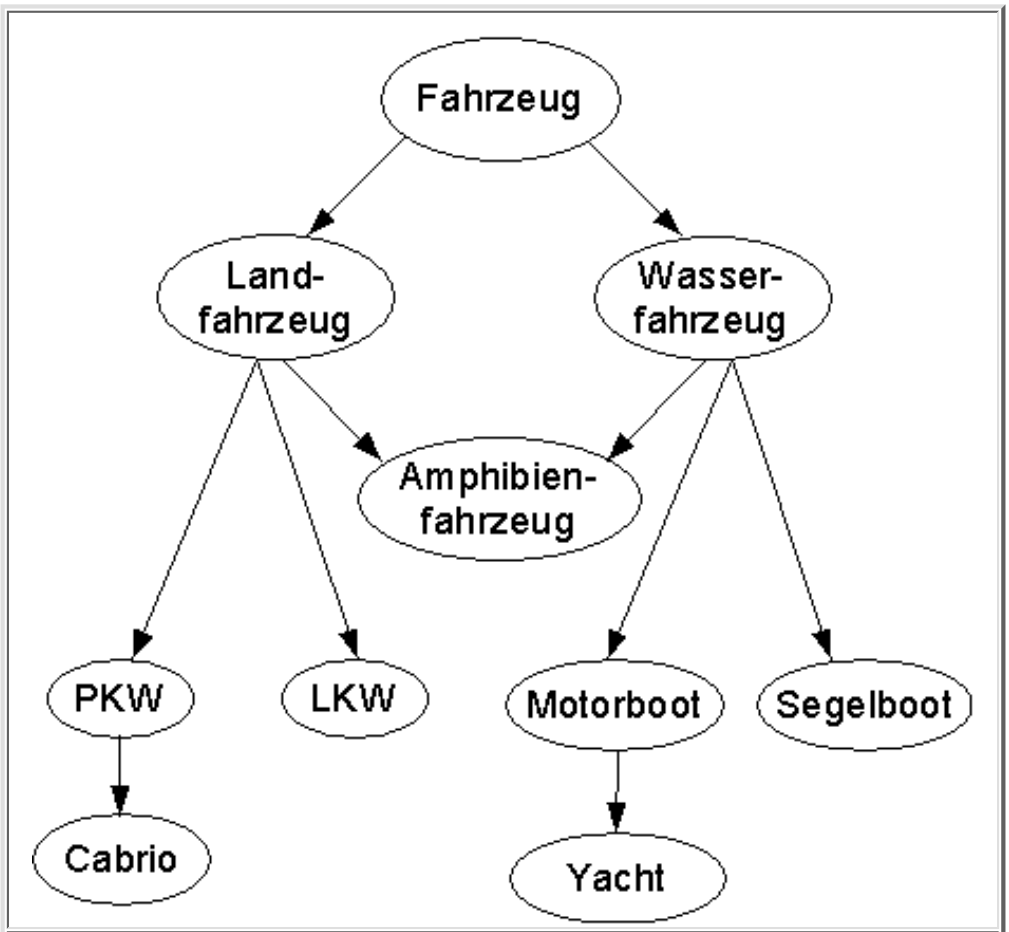
1. Die Klassendeklaration kann durch das Hinzufügen neuer Methoden und Variablen erweitert werden.

2. Eine bestehende Methode kann durch eine Neue ersetzt, überschrieben werden. Man erreicht das durch das Einfügen einer neuen Methode, die denselben Namen wie die Methode des Vorgängers besitzt.
3. Eine bestehende Methode kann durch eine Neue erweitert werden. Die neue Funktion kann dazu eine der vorhergehenden Funktionen aus einer der Basisklassen aufrufen. Beim Aufruf der Funktion muß angegeben werden, aus welcher Vorgängerklasse (Basisklasse) die aufzurufende Funktion entstammen soll.

Einmal vorhandene Methoden können von den Erben nie wieder entfernt werden. Auch im Basisobjekt vorhandene Variablen können von Nachfolgern nicht entfernt oder überschrieben (im Sinne einer Neu- oder Umdefinition) werden.

Durch Vererbung kann eine ganze Familie von Klassen entstehen, welche man Klassenhierarchie nennt.

In einigen objektorientierten Sprachen besteht daneben auch die Möglichkeit der Mehrfachvererbung. Ein Objekt erbt dabei nicht nur die Eigenschaften eines, sondern mehrerer Basisobjekte.



Objekt Amphibienfahrzeug entsteht durch Mehrfachvererbung aus Landfahrzeug und Wasserfahrzeug

Durch den Vererbungsmechanismus müssen gemeinsame Eigenschaften nur einmal festgelegt werden. Vom "Code-Recycling" durch Vererbung verspricht man sich zum einen eine **Produktivitätssteigerung** bei der Softwareentwicklung, muß doch so nicht immer wieder "das Rad aufs neue erfunden" werden. Andererseits gelangt man zu einer erhöhten **Zuverlässigkeit** der Software, da weniger Code neu erstellt und getestet werden muß. Bereits als zuverlässig bekannter und getesteter Standard-Code aus umfangreichen Bibliotheken steht zur Verfügung.

Überladen und Polymorphismus stehen in engem Zusammenhang. Polymorphie entstammt dem Griechischen und steht für das Wort "Vielgestaltigkeit" bzw. die Möglichkeit, sich von mehreren Seiten zu zeigen. Operationen, die Werte beliebiger Typen übernehmen können, nennt man polymorph, während man unterschiedliche Operationen, die denselben Namen benutzen, überladen nennt.

Eine Funktion, die in zwei verschiedenen Klassen deklariert ist, kann dank des Polymorphismus unterschiedliche Aktionen ausführen. So könnte zum Beispiel das Objekt Fahrzeug aus der oben aufgeführten Klassenhierarchie eine Funktion "FAHR_LOS" definieren, die jedoch nur das Starten des Antriebswerks bewirkt. Die vererbten Klassen überladen "FAHR_LOS" so, daß Klasse Landfahrzeug die Antriebsenergie an vier Räder überträgt, während dessen Klasse Wasserfahrzeug diese an eine Schiffsschraube weiterleitet. Allgemeiner ausgedrückt heißt das, daß die endgültige Implementierung polymorpher Funktionen erst in den Unterklassen im Sinne der Verantwortlichkeiten der Klassen und Objekte implementiert werden. Unterschiedliche Objekte können beim Aufruf der gleichen Methoden oder beim Versenden derselben Nachrichten an sie auf ihre eigene, spezifische Art und Weise reagieren.

Diese Flexibilität können Objekte nur deswegen aufweisen, weil sie mehrere dynamische Konzepte kennen und anwenden:

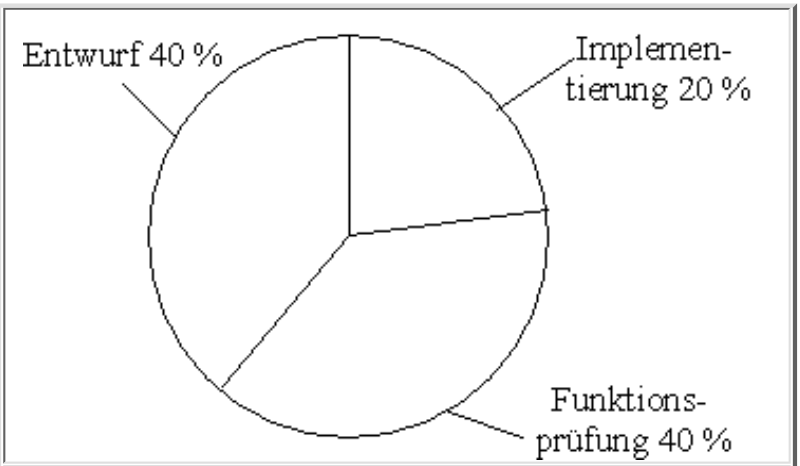
- dynamische Typprüfung
- dynamisches Binden (spätes Binden)
- dynamisches Laden

Statische Typprüfungen während des Übersetzens von Programmen sind sehr nützliche Eigenschaften von Compilern und helfen, Fehler zu vermeiden. Die polymorphen Eigenschaften der objektorientierten Sprachen machen es aber manchmal notwendig, Typen erst dynamisch zur Laufzeit zu prüfen und auszuwerten. Zum Beispiel könnte eine Methode ein Objekt als Parameter mitliefern, ohne daß beim Übersetzen der genaue Typ dieses Objekts bekannt ist. Der Typ wird erst zur Laufzeit determiniert, ist möglicherweise auch erst jetzt bekannt.

Dynamisches Binden, manchmal auch spätes Binden genannt, verschiebt beim Übersetzen die exakte Entscheidung, welche Methode aufzurufen ist. Vielmehr wird dies erst zur Laufzeit des Programms genau entschieden. Die Sprachen C++ und Object Pascal verlangen stets statische, eindeutige Typangaben im Quelltext (*strong compile-time type checking*). Objekttypen sind direkt zum Typ der eigenen Basisklasse oder zu einem beliebigen anderen, vorhergehenden Klassentyp des Vererbungszweiges zuweisungskompatibel. D.h., spezialisiertere Objekte können immer so benutzt werden, als ob sie weniger spezialisierte Objekte wären. Durch statische Typkonvertierung oder dynamische Typprüfung mit anschließender Typkonvertierung kann aber auch umgekehrt ein Objekt einer weniger spezialisierten Klasse in ein Objekt einer höher spezialisierten Klasse umdefiniert werden. Insbesondere bei dynamischer Typprüfung *kann* der Compiler nicht wissen, welche Klassenmethoden er aufrufen muß. In C++ und Object Pascal wird die Entscheidung, welche genaue Methode aufzurufen ist, zur Laufzeit getroffen, wenn die Methode mit dem Schlüsselwort **virtual** deklariert worden ist. So deklarierte Methoden nennt man **virtuelle Methoden**. Erst zur Laufzeit des Programms wird festgelegt, welcher Funktionsrumpf an einen konkreten Funktionsaufruf gebunden wird.

Neben dynamischer Typprüfung und dem dynamischen Binden unterstützen manche objektorientierte Programmiersprachen zusätzlich das Konzept des dynamischen Ladens. Programmteile werden dabei erst dann in den Arbeitsspeicher geladen, wenn sie tatsächlich benötigt werden. C++ und Object Pascal unterstützen dynamisches Laden nur in soweit, wie es durch die Betriebssysteme mit dem Windows 32 - API ermöglicht wird: nämlich dynamisch ladbare Bibliotheken (DLL's) und systemweit einsetzbare Objekte nach dem OLE2 (Object Linking and Embedding Version 2) Verfahren.

Es ist einleuchtend, daß durch das Ausnutzen der Prinzipien von Kapselung, Polymorphie, Überladen und dynamischer Eigenschaften Anwenderprogramme einfacher und besser lesbar werden und so letztlich auch leichter erweitert werden können. Ein Anwender kümmert sich nicht darum, wie eine Aufgabe ausgeführt wird, sondern nur darum, welche Aufgaben überhaupt erfüllt werden können, währenddessen die Objekte "für sich selbst verantwortlich" sind. So können unter anderem auch viel leichter ungültige Zustände von Attributen vermieden werden. Hierarchien bieten die Möglichkeit, ein besseres Verständnis des Anwendungsbereichs zu erlangen und sorgen für eine gewisse Ordnung. Jetzt ist es möglich, lokale Veränderungen in großen Systemen vorzunehmen, ohne daß globale Modifikationen nötig werden. Besonderes Gewicht erfahren diese Aussagen vor dem Hintergrund, daß 80% Zeit der Softwareentwicklung nicht mit dem Neu-Schreiben, sondern mit Testen, Wartung, Erweiterung und Fehlerbereinigung bestehender Systeme verbracht werden [4].



Durchschnittlicher Zeitbedarf bei der Softwareerstellung [4]

Negativ allerdings stehen den vielen Vorteilen und der großen Flexibilität objektorientierter Systeme ein prinzipiell erhöhter Ressourcen- und Rechenbedarf gegenüber. Ein System / Programm wird entsprechend [4] durch verschiedene Qualitätsmaßstäbe bewertet:

Korrektheit	fordert exakte Erfüllung der Aufgabe
Adaptierbarkeit	Anpassung an ähnliche Probleme
Portabilität	Anpassung an andere Betriebssysteme
Kompatibilität	Kombinierbarkeit von Teilsystemen
Zuverlässigkeit	Wahrscheinlichkeit für befriedigende Ausführung
Robustheit	Verkraften von Fehlbedienung, u. ä.
Verfügbarkeit	Ausfall- bzw. Standzeiten
Benutzerfreundlichkeit	bewertet Schnittstelle Benutzer <=> Programm
Wartbarkeit	Fehlerbeseitigung und funktionale Erweiterung/ Anpassung
Effizienz und Leistung	bewertet Nutzung aller Betriebsmittel (wie Speicher und Rechenzeit)

Da diese Forderungen teilweise im Widerspruch zueinander stehen und sich gegenseitig ausschließen, kann kein Programm alle Kriterien gleichzeitig, geschweige denn in idealer Weise erfüllen. Effizienz stellt also nur eine unter vielen Forderungen dar. Generell kann aber gesagt werden, daß der Einsatz objektorientierter Systeme die Erfüllung vieler der Kriterien begünstigt.

In vielen der heute verbreiteten objektorientierten Sprachen ist der eigentliche Compiler nur ein Teil des Entwicklungssystems. Er wird in einer integrierten Entwicklungsumgebung (IDE) von leistungsfähigen Editoren, ausgefeilten kontextsensitiven Hilfesystemen, einer Projektverwaltung, Klassenbrowsern, Debuggern, erweiterbaren Komponentenbibliotheken und weiteren Tools umgeben. Mehr und mehr gewinnt auch das "visuelle Programmieren" an Bedeutung. Die Bedienelemente einer zu erstellenden Anwendung werden vom Entwickler graphisch plaziert und angeordnet. Codefragmente können dann in einem weiteren Schritt den Bedienelementen und bestimmten, auswählbaren Ereignissen dieser Bedienelemente zugeordnet werden. Selbst Datenbankanbindungen- und verknüpfungen können interaktiv und auf visuelle Art erfolgen. Die benötigten Entwicklungszeiten lassen sich durch diese komfortablen Helfer ganz beträchtlich senken, während dessen die Fehlerrate sinkt. Der Entwickler wird von stupiden, immer gleichbleibenden Arbeitsschritten befreit und kann sich um so mehr auf die eigentliche Lösung einer bestimmten Aufgabe konzentrieren.

Alle genannten Vorteile OO-basierter Sprachen und ihrer Entwicklungssysteme erfüllen sich allerdings nicht von alleine, sondern nur dann, wenn der Phase der Softwareerstellung eine sorgfältige Analyse und ein wohlüberlegtes Klassendesign vorausgehen. Ziel der

Analysephase (OOA) ist es, eine möglichst dauerhafte Klassenstruktur zu erstellen. Die Designphase (OOD) soll danach klären, wie das Programm selbst ablaufen soll und auf welche Weise die Aufgaben erfüllt werden. Man bemüht sich darum, die Klassenstruktur zu verfeinern, die Architektur des Programms zu erstellen, was auch konkrete Entscheidungen hinsichtlich Datenhaltung, Speicherverwaltung, Fehlerbehandlung u.s.w. beinhaltet, so daß dann anschließend eine reibungslose Implementierung erfolgen kann. Um dem Prozeß des Entwerfens objektorientierter Systeme eine klare Struktur zu geben, wurden mehrere Methoden entworfen, unter anderem die von G. Booch, dargestellt in [1]. Der grundsätzliche Ablauf in diesen beiden Phasen wird von ihm so dargestellt:

1. Identifizierung der Klassen und Objekte, die im System angewendet werden sollen. Das ist der schwierigste Teil beim Systementwurf. Man muß sich zu allererst intensiv mit dem Anwendungsbereich auseinandersetzen, um ein tieferes Verständnis zu erlangen. Das dynamische Verhalten von Systemen im Anwendungsbereich kann z. B. als Quelle für Klassen und Objekte dienen. Interessante Szenarien spielt der Entwickler durch und erkennt so, welche Objekte ein ähnliches Verhalten aufweisen. Sie geben Aufschluß über sinnvolle Klassen.
2. Zuweisung der Attribute und Funktionen zu den Klassen und Objekten. Man bestimmt die Verantwortlichkeit des jeweiligen abstrakten Objekts und sucht dann Verhaltensmuster, die mehreren Klassen und Objekten gemeinsam sind. Mit der Zeit findet man neue Klassen und durchdringt das Anwendungsgebiet dabei immer besser.
3. Festlegung der Beziehungen von Klassen und Objekten untereinander. Häufig werden dabei Klassen, die in einem früheren Stadium erstellt wurden umgeändert, fallen ganz weg, werden nachträglich in gemeinsamen Oberklassen zusammengefaßt oder auch aufgeteilt, falls sie zu komplex geworden sind.
4. Implementierung des unter 1. - 3. erstellten Entwurfs in der Zielsprache (z. Bsp. C++ oder Object Pascal)

Booch teilt darüber hinaus den Entwurfsprozeß in Mikroprozesse und den Makroprozeß (entsprechend 1. bis 4.) ein. Mikroprozesse betreffen die tägliche Arbeit des Entwicklers und schließen das Finden von Klassen und Objekten, ihrer Attribute, Methoden und Beziehungen untereinander ein. Das Entwurfsverfahren läuft im Kern so ab: Zuerst werden sehr abstrakte Klassen des Anwendungsbereichs gesucht, die anschließend im Mikroprozeß bearbeitet werden. Dabei entdeckt man Klassen auf anderen Abstraktionsniveaus, wie Oberklassen. Einige Klassen werden sich als unnötig erweisen und werden somit wieder verworfen. Jede der neuen Klassen wird wieder in den Mikroprozeß eingeschleust und man hangelt sich so mit jeder neuen Runde des Mikroprozesses auf eine niedrigere Abstraktionsstufe herunter. Das Verfahren wird durch eine Reihe, in enger Beziehung zu einander stehender Diagramme unterstützt: Klassen-, Zustands-, Objekt-, Interaktions-, Modul- und Prozeßdiagramme. Es existiert eine stattliche Anzahl von Programmen, die bei der Analyse und dem Design objektorientierter Systeme behilflich sind und den Entwickler unterstützen.

2.2 Umsetzung des OOP-Gedankens in C und Pascal

Im letzten Schritt des Makroprozesses wird das gewonnene Objektmodell in eine Programmiersprache umgesetzt, die natürlich objektorientiertes Programmieren erlauben muß, besser jedoch aktiv fördern sollte. Neben Sprachen wie Simula und Smalltalk, die das Erstellen eleganten OO-Codes ermöglichen, wurden im Laufe der Zeit sehr vielen Sprachen, die zunächst nur die prozedurale Strukturierung und Modularisierung zuließen, objektorientierte Spracherweiterungen hinzugefügt. Typische Vertreter sind die Sprachen C und Pascal. Sie besitzen unter anderem Recordtypen und Zeiger und beherrschen beschränkte Formen der Typerweiterung, sowie der Kapselung. Während bei Sprachen wie PL/1, Fortran, Cobol und Ada mit ihren objektorientierten Erweiterungen eine nicht-triviale Umsetzung bestehenden Codes von der jeweiligen Vorgängersprache notwendig ist, besitzt C++ die Sprache C und Object Pascal die Sprache Pascal als praktisch vollständig implementierte Teilsprachen. C++ und Object Pascal fanden sicher auch wegen der leichten Wiederverwendbarkeit bestehenden Codes Akzeptanz in der Industrie. Aus diesem Grund sollen im folgenden diese beiden Sprachen und ihre derzeitige Umsetzung in zwei kommerziell vertriebenen Entwicklungsumgebungen miteinander verglichen werden.

Die **Sprache C++** wurde Anfang der 80er Jahre maßgeblich von Bjarne Stroustrup entwickelt [6]. Sein Wunsch war es, eine Programmiersprache zu schaffen, die das objektorientierte System der Sprache SIMULA mit dem sehr leistungsfähigen Compiler C verbindet und das "Konzept der Klassen" effektiv umsetzt. So entstand zunächst eine Spracherweiterung, die Stroustrup "C mit Klassen" nannte. Diese wurde in den folgenden Jahren durch ihn mit dem Ziel erweitert, die Sprache einem möglichst großen Kreis von Programmierern nahezubringen. Sie wurde in der ersten Zeit zunächst noch als Präprozessor in C realisiert und bekam bald den Namen C++. Um dem Ziel einer großen Verbreitung nahe zu kommen, wurde besonders darauf Wert gelegt, daß diese neue Sprache

abwärtskompatibel zum bisherigen C sein sollte. Der neue, elementare Sprachkonstrukt "class" ist eine Erweiterung des C-Schlüsselwortes "struct" und war somit den C Programmierern vertraut. So sagt Stroustrup: "Mit dem Entwurf von C++ wollte ich ein Werkzeug zur Lösung von Problemen schaffen und nicht eine bestimmte Behauptung beweisen; die anschließende Entwicklung dieser Sprache war auf die Bedürfnisse der Programmierer ausgerichtet." [6]

Eine allgemeingültige Sprachdefinition wurde durch ihn im Buch "The C++ Programming Language" gegeben und wird mittlerweile durch ein internationales Komitee standardisiert (ANSI / ISO) und beständig weiter entwickelt. Die derzeit aktuell gültige Version trägt die Nummer 3.0. Eine Vielzahl bestehender C++ Compiler halten sich, ebenso wie Visual C++ der Firma Microsoft, relativ streng an diese Vorgaben, bieten darüber hinaus aber noch zusätzliche Erweiterungen an.

PASCAL wurde 1970 durch Prof. Niklaus Wirth mit dem Ziel entwickelt, eine kleine Lehrsprache zu Unterrichtszwecken zu erhalten, anhand derer gute Programmiertechniken demonstriert werden konnten. Der Name Pascal wurde zu Ehren des französischen Mathematikers und Philosophen Blaise Pascal (1623 - 1662) gewählt. Pascal zeichnet sich durch starke Typbindung und eine Reihe vorgegebener Typen aus, bietet aber auch die Möglichkeit, benutzerdefinierte Typen zu erstellen. Auch diese Sprache fand viele Anhänger und verbreitete sich insbesondere deswegen rasch, weil bald für alle gängigen Computersysteme Compiler zu Verfügung standen.

Auch Standard-Pascal wurde vor einiger Zeit normiert, bekannt unter dem Namen ISO-Pascal. 1989 wurde eine erweiterte Normierung unter dem Namen "Extended Pascal" veröffentlicht (ISO 10206). Man hat seit dem jedoch versäumt, Anpassungen vorzunehmen, so daß bis heute standardisierte Vorgaben zu objektorientierten Erweiterungen fehlen. Einzig ein unverbindlicher Bericht unter dem Namen "Technical Report on Object-Oriented Extensions to Pascal" wurde bisher veröffentlicht. [8]

Trotzdem hat sich Pascal ständig weiter entwickelt und besitzt heute ebenfalls objektorientierte Spracherweiterungen. Erweiterungen haben die Firmen vorgenommen, die kommerziell Compiler für Pascal entwickelt haben. Dabei hat sich der Softwarehersteller Borland hervorgetan, der in Pascal frühzeitig objektorientierte Erweiterungen implementierte. Der ihrer aktuellen Entwicklungsumgebung "Delphi" zugrunde liegenden Sprache haben sie den Namen "Object Pascal" verliehen.

C++ und Object Pascal besitzen sehr viele Gemeinsamkeiten. Neben den nicht zu übersehenden Unterschieden im Sprachumfang finden sich aber auch häufig Unterschiede im Detail, die zunächst leicht übersehen werden können. Deswegen soll im folgenden der Sprachumfang beider Systeme detailliert betrachtet und verglichen werden. Alle Angaben zu C++ beziehen sich auf das Entwicklungssystem "Microsoft Visual C++ Version 4.00 Professionell Edition" für INTEL-basierte Systeme. Ausführungen zu Object Pascal beziehen sich auf die konkrete Realisierung in Borlands Compiler "Delphi Version 2.0 Client/Server". Beide Systeme laufen nur auf Windows 32 basierenden Betriebssystemen und können auch nur für diese Betriebssysteme Maschinencode erzeugen.

2.3 Sprachumfang beider Systeme

Einen ersten Eindruck von der Komplexität der beiden Sprachen bietet folgende Tabelle (erweitert nach [5]):

Sprache	Schlüsselworte	Anweisungen	Operatoren	Seiten Dokum.
Pascal	35	9	16	28
Modula-2	40	10	19	25
Object Pascal (Delphi v2.0)	*92*	*24*	*23*	*198*
C	29	13	44	40
C++ v1.0	42	14	47	66
C++ v3.0	48	14	52	155
Visual C++ v4.0	*83*	*22*	*61*	*291*
Ada	63	17	21	241

Schlüsselworte (Keywords) haben in einer Programmiersprache eine feste Bedeutung und können nicht umdefiniert werden. Beispiele sind: class, if, for, goto und register.

Anweisungen (Statements) beschreiben algorithmische Aktionen, die ein Programm ausführen kann. Man unterscheidet zwischen *einfachen* Anweisungen, wie z. B. Zuweisungsanweisung (:=), Prozeduranweisung, Sprunganweisungen und *strukturierten* Anweisungen. Zu ihnen zählen z. B. Blockanweisungen, Schleifen und bedingte Anweisungen.

Streng betrachtet besitzt Object Pascal nur 67 Schlüsselworte. 25 weitere Worte nennt Borland "Standard-Anweisungen" mit vordefinierter Bedeutung [9]. Zu ihnen zählen Begriffe wie assembler, message und virtual. Ein Neudefinieren dieser Wörter ist theoretisch möglich, wovon aber strikt abgeraten wird. Man muß diese Standard-Anweisungen deswegen als Quasi-Schlüsselworte ansehen.

Das bloße Zählen der Anzahl der Schlüsselwörter, Anweisungen und Operatoren kann leicht ein verzerrtes Bild von der Komplexität verschiedener Sprachen liefern. So weist C++ einige Schlüsselwörter auf, die trotz gleichen Namens je nach Anwendungsbereich eine komplett andere Bedeutung haben. Als ein extremes Beispiel kann hier das Wort "static" angeführt werden. Es gibt erstens die lokale static Variable, die ihren Wert über das Blockende hinaus behält. Es existieren zweitens mit static definierte dateibezogene Bezeichner, deren Gültigkeit / Sichtbarkeit auf diese Datei begrenzt sind. Es gibt drittens Klassenvariable, die static sind und deren Existenz dadurch unabhängig von der Klasse ist. Und es gibt viertens static Member- (Klassen-)Funktionen. Hier steht der static-Bezeichner dafür, daß diesen Funktionen der "this"-Zeiger nicht als Argument übergeben wird. Die Existenz von Schlüsselkonzepten mit Mehrfachbedeutungen führt leicht zu Verwirrungen. In der oben aufgeführten Tabelle kommen solche komplexen Sachverhalte nicht zum Ausdruck.

Trotzdem macht allein die Anzahl der fest vordefinierten Schlüsselworte, Standard-Anweisungen und Operatoren deutlich, daß C++ und Object Pascal im Vergleich zu ihren prozeduralen Vorfahren durch die objektorientierten Erweiterungen wesentlich komplexer und umfangreicher geworden sind. Sie reflektieren offensichtlich die Tatsache, daß viele Veränderungen nötig waren, um den neuen Anforderungen gerecht zu werden. Sie zeigen aber auch, daß das Erlernen und Meistern dieser neuen Sprachen schwieriger geworden ist. Erschwerend kommt beim objektorientierten Programmieren gegenüber dem in C und Pascal hinzu, daß mindestens ein extra Schritt beim Entwurf des OO-Designs nötig wird, bevor mit dem eigentlichen Schreiben der Software begonnen werden kann.

2.3.1 Lexikalische Elemente und Operatoren

Programme bestehen aus Elementen, die man Tokens nennt, und stellen eine Sammlung von Zeichen des Compiler-Vokabulars dar. Sie schließen in beiden Sprachen die Zeichen des einfachen (7 Bit) ASCII-Zeichensatzes ein und lassen keine deutschen Umlaute zu. Eine Trennung der Tokens wird durch eingefügte "Whitespaces" wie Leerzeichen, Tabulatoren, Zeilenumbrüche und Kommentare definiert. Bei der lexikalischen Analyse produziert der Compiler interne Tokens, indem er jeweils die längste Zeichenkette auswählt, die einem Token entspricht. Es gibt fünf Token-Kategorien :

- Schlüsselworte (reservierte Wörter / Keywords)
- Bezeichner (Identifiers)
- Konstanten (inklusive String-Konstanten)
- Operatoren
- Trennzeichen (Punctuators)

Bezeichner werden zur Kennzeichnung von Objekten, Variablen, Klassen, Strukturen, Funktionen, Prozeduren, Labels und Makros verwendet. Namen von Bezeichnern bilden eine Folge von Buchstaben, Ziffern und dem Unterstrichzeichen "_". Bezeichner dürfen nicht mit einer Ziffer beginnen und keine Leerzeichen enthalten. VC++ verwendet maximal 247 Zeichen eines Bezeichners, während

in Object Pascal nur die ersten 63 Zeichen signifikant sind. C++ unterscheidet zwischen Groß- und Kleinschreibung von Bezeichnern, ebenso bei Schlüsselwörtern. Es ist aber auch beim Schreiben von Pascal-Programmen empfehlenswert, wiederholt auftretende Bezeichner in gleicher Weise zu schreiben. Die Lesbarkeit wird verbessert und die Wiedererkennung erleichtert.

Kommentare werden beim Übersetzungsvorgang nicht ausgewertet, sondern einfach ignoriert. C++ und Object Pascal leiten einzeilige Kommentare durch die Zeichen // ein. Sie reichen immer bis zum Ende der Zeile. Mehrzeilige Kommentare werden in C++ durch /* eingeleitet und mit */ beendet und in Object Pascal mit { oder (* eingeleitet und entsprechend mit } bzw. *) beendet.

C++	Object Pascal
`//einzeiliger C++ Kommentar`	`//einzeiliger Pascal Kommentar`
`/* das ist ein mehrzeiliger` ` Kommentar in C++` `*/`	`{das ist ein mehrzeiliger` `Kommentar in Pascal}` `(* das ebenso *)`

Innerhalb von Kommentaren dürfen alle Zeichen des erweiterten ASCII-Zeichensatzes verwendet werden. Die einzige Ausnahme betrifft Object Pascal und verbietet die Verwendung des Zeichens $ bei mehrzeiligen Kommentaren. Tokens der Form {$...} werden als Compiler-Befehl interpretiert. C++ benutzt einen sogenannten **Präprozessor**, um derartige Compiler-Befehle noch vor der eigentlichen lexikalischen Analyse auszuwerten. Dieser konvertiert den Quellcode von seiner Präprozessor-Form in reine C++ Syntax um. Solche Direktiven werden hier immer durch das Symbol # eingeleitet.

Bedingte Übersetzung beherrschen beide Compiler. Als gleichwertig kann betrachtet werden:

	VC++	Delphi
Bedingtes Symbol mit dem angegebenen Namen definieren und das Symbol auf TRUE setzen. Bedingte Symbole gleichen booleschen Konstanten, da sie entweder den Wert TRUE oder FALSE besitzen.	`#define` *`identifier`*	`{$DEFINE` *`identifier`*`}`
Definition eines zuvor definierten bedingten Symbols *identifier* aufheben.	`#undef` *`identifier`*	`{$UNDEF` *`identifier`*`}`
Der darauf folgende Quelltext wird nur dann übersetzt, wenn die angegebene Bedingung zutrifft (TRUE liefert), also *identifier* bei ifdef definiert bzw. bei ifndef nicht definiert ist.	`#ifdef` *`identifier`* `#if defined` *`identifier`* `#ifndef` *`identifier`* `#if !defined` *`identifier`*	`{$IFDEF` *`identifier`*`}` `{$IFDEF` *`identifier`*`}` `{$IFNDEF` *`identifier`*`}` `{$IFNDEF` *`identifier`*`}`
Steuern den Kontrollfluß bzw. setzen das Ende der bedingten Anweisungen.	`#else` `#elif` `#endif`	`{$ELSE}` `keine Entsprechung` `{$ENDIF}`

VC++	Delphi

Bsp.:

```
#if !defined( EXAMPLE_H )
#define EXAMPLE_H

class Example
{
...
};

#endif
```

Bsp.:

```
{$IFDEF _DEBUG}
{ShowMessage('Debug'+StrText)}
{$ELSE}
{ShowMessage(StrText)}
{$ENDIF}
```

Einige Symbole werden durch die Compiler bzw. Bibliotheken bereits vorbelegt, wie z. B. _WIN32 in VC++ und Win32 in Object Pascal. Sie erleichtern das Erstellen von portierbarem Code. Neben den einfachen Mechanismen zur bedingten Compilierung existieren in C++ weitere Möglichkeiten, vor dem eigentlichen Übersetzungsvorgang Manipulationen am Quelltext vorzunehmen.

Der Inhalt einer ganzen Datei kann in C++ durch die Präprozessor-Anweisung #include *Datei*, in Pascal durch die Compiler-Anweisung {$I *Datei*} oder {$INCLUDE *Datei*} im Quelltext eingefügt werden. Das Einfügen erfolgt an der Stelle, an der die Compileranweisung auftritt. In C++ wird die include-Anweisung intensiv dazu benutzt, um die, in getrennten Dateien geführten, Schnittstellendefinitionen (Header-Dateien) und Implementierungen (cpp-Dateien) wieder zusammen zuführen. Weitere Erläuterungen folgen im Abschnitt "Programmstruktur".

Beide Compiler können Projekteinstellungen, global bzw. lokal auf einen bestimmten Text-abschnitt bezogen, im Quelltext umdefinieren. C++ bemüht dazu wiederum seinen Präprozessor und verwendet sogenannte "Pragma - Direktiven" als Compiler-Anweisungen in der Form #pragma CompilerInstruction. Sie stellen maschinenspezifische Anweisungen dar. Delphi nutzt statt dessen wieder seine Art der Compiler-Anweisungen in der Form {$...}. Um beim Übersetzen die Ausgabe von Warnungen zu vermeiden / die Optimierung zeitweise auszuschalten, könnte folgendes geschrieben werden:

VC++

```
// Warnung Nr. 34 wird
// unterdrueckt

#pragma warning(disable: 34)
void func()
{
a;
}
#pragma warning(default: 34)
```

Delphi

```
// alle Warnungen werden
// unterdrueckt

{$WARNINGS OFF}
procedure func;
begin
a;
end;
{$WARNINGS ON}
```

```
// jegliche Optimierung aus

#pragma optimize("", off)
void func()
{
b;
}
#pragma optimize("", on)
```

```
// jegliche Optimierung aus

{$OPTIMIZATION OFF}
procedure func;
begin
b;
end;
{$OPTIMIZATION ON}
```

Pragma-Anweisungen sind, ebenso wie die Anweisungen zum bedingten Compilieren, auf Grund der Präprozessor-Nutzung unter C++ wesentlich flexibler einsetzbar, als dies mit den Compileranweisungen von Delphi der Fall ist. Der Einsatz des Präprozessors ermöglicht überdies noch die Definition von Makros; eine Spracherweiterung, die Delphi fehlt (zu Makros mit Parametern siehe auch Abschnitt "2.3.5 Programmstruktur").

Beide Sprachen bieten eine Vielzahl von **Operatoren** an. Einige der Operatoren finden in der jeweils anderen Sprache gleichartige Gegenstücke, sind aber laut Sprachdefinition keine Operatoren, sondern Ausdrücke, Funktionen o. ä. Die folgende Tabelle listet alle Operatoren beider Sprachen und gleichartige Gegenstücke auf.

Operator	VC++	Object Pascal
kleiner als	**<** `boolean IsSmaller;` `IsSmaller= 4 < 8; // TRUE`	**<** `var IsSmaller: Boolean;` `IsSmaller:= 4 < 8; // TRUE`
kleiner gleich	**<=**	**<=**
größer als	**>** `boolean IsLarger;` `IsLarger= 4 > 8; // FALSE`	**>** `var IsLarger: Boolean;` `IsLarger:= 4 > 8; // FALSE`
größer gleich	**>=**	**>=**
enthalten in	**kein solcher Operator definiert**	**in** `if 'L' in ['A'..'Z'] then` `writeln('Buchstabe L' +` `gehört dazu');`
Ungleich-heit	**!=** `boolean IsDifferent;` `IsDifferent= 4 != 8;//TRUE`	**<>** `var IsDifferent: Boolean;` `IsDifferent:=4 <> 8;//TRUE`

Gleichheit	**==** `boolean IsEqual;` `IsEqual= 4 == 8; // FALSE`	**=** `var IsEqual: Boolean;` `IsEqual:= 4 = 8; // FALSE`
Zuweisungs-operator	**=** `unsigned short i;` `i = 4;`	**:=** `var i: Word;` `i := 4;`
Anmerkung:	Der Zuweisungsoperator und der Operator zur Prüfung auf Gleichheit wurden bereits in der Sprache C vielfach kritisiert. Allzuleicht wird statt einer gewünschten Gleichheits-prüfung unbeabsichtigt eine Zuweisung durchgeführt: `void main()` `{` `int a = 3; // a ist 3` `if (a = 4)` `  printf("a ist 4");` `else` `  printf("a nicht 4");` `}` Ausgabe: a ist 4 Der else-Zweig wird nie durchlaufen werden. Bei if (a = 4) wird nicht a geprüft, sondern a wird der Wert 4 zugewiesen und der Erfolg dieser Zuweisung geprüft. Der verläuft natürlich immer erfolgreich und liefert also TRUE. Der Compiler kann solch einen Fehler nicht aufdecken, weil es kein syntaktischer oder semantischer, sondern ein logischer Fehler ist.	
Erweiterte Zu-weisungs-operatoren	**<Bez1><Operator>=<Bez2>** entspricht <Bez1>=<Bez1> <Operator><Bez2> sofern keine Seiteneffekte auftreten, wie z.B. beiv[i++] *= 5;	**keine erweiterten Zuweisungsoperatoren** Statt dessen können die, sowieso besser lesbaren, normalen Operatoren angewendet werden.

	`int a, b;` `a += b;` `a -= b;` `a *= b;` `a /= b;` `a += b;` `a %= b;` `a >>= b;` `a <<= b;` `a &= b;` `a ^= b;` `a |= b;`	`var a, b: Integer;` `a := a + b;` `a := a - b;` `a := a * b;` `a := a / b;` `a := a + b;` `a := a mod b;` `a := a shr b;` `a := a shl b;` `a := a and b;` `a := a xor b;` `a := a or b;`
Addition	**+**	**+**
Subtraktion	**-**	**-**
Multiplikation	*****	*****
Division	**/**	**/**
Ganzzahl-Division	**div()** `#include <stdlib.h>` `int i;` `i= div(10, 3).quot;// i=3` `// Komma-Anteil wird` `// abgeschnitten` Alternativ: implizite Typkonvertierung, die aber fehleranfälliger ist: `i = 10 / 3;`	**div** `var i: Integer;` `i:= 10 div 3;  // i=3` `// Komma-Anteil wird` `// abgeschnitten` Alternativ: explizite Typkonvertierung: `i:= Trunc(10 / 3);`
Modulo-operator	**%** `int k;` `k = 32 % 3;  // k = 2`	**mod** `var k: Integer;` `k:= 32 mod 3;  // k = 2`

boolscher NICHT Operator	**!** `#include <wtypes.h>` `boolean bekannt;` `bekannt = ! TRUE;` `// bekannt = FALSE`	**not** `var bekannt: Boolean;` `...` `bekannt:= not TRUE;` `// bekannt = FALSE`
boolscher UND Operator	**&&** `#include <wtypes.h>` `boolean Inside;` `int i = 5;` `Inside = (i>2) && (i<9);` `// Inside = TRUE`	**and** `var Inside: Boolean;` `  i: Integer;` `i:= 5;` `Inside:= (i>2) and (i<9);` `// Inside = TRUE`
boolscher ODER Operator	**||** `#include <wtypes.h>` `boolean Outside;` `int i = 5;` `Outside = (i<2) || (i>9);` `// Outside = FALSE`	**or** `var Outside: Boolean;` `  i: Integer;` `i:= 5;` `Outside:= (i<2) or (i>9);` `// Outside = FALSE`
bitweises 1-er Komple-ment	**~** `int k, l;` `k = 6;  // 0...00110` `l = ~k; // 1...11001`	**not** `var k, l: Integer;` `...` `k:= 6;     // 0...00110` `l:= not k; // 1...11001`
bitweises Schieben nach links (shift left)	**<<** `int i, j;` `i = 4;      // 0...000100` `j = i << 2; // 0...010000`	**shl** `var i, j: Integer;` `i:= 4;      // 0...000100` `j:= i shl 2;// 0...010000`

bitweises Schieben nach rechts (shift right)	**>>** `int i, j;` `i = 16;  // 0...010000` `j = i >> 2; // 0...000100`	**shr** `var i, j: Integer;` `i:= 16;  // 0...010000` `j:= i shr 2;// 0...000100`
bitweiser UND Operator	**&** `int x, m, y;` `x = 21;  // 0...010101` `m = 19;  // 0...010011` `y = x & m;  // 0...010001`	**and** `var x, m, y: Integer;` `x:= 21;  // 0...010101` `m:= 19;  // 0...010011` `y:= x and m;// 0...010001`
bitweiser ODER Operator	**|** `int x, m, y;` `x = 21;  // 0...010101` `m = 19;  // 0...010011` `y = x | m;  // 0...010111`	**or** `var x, m, y: Integer;` `x:= 21;  // 0...010101` `m:= 19;  // 0...010011` `y:= x or m; // 0...010111`
bitweiser EXKLUSIV-ODER Operator	**^** `int x, m, y;` `x = 21;  // 0...010101` `m = 19;  // 0...010011` `y = x ^ m;  // 0...000110`	**xor** `var x, m, y: Integer;` `x:= 21;  // 0...010101` `m:= 19;  // 0...010011` `y:= x xor m;// 0...000110`
Vektor-operator	**[]** `//Array Variable` `int v[10]`	**Array[]** `var` `v: Array[0..9] of Integer;`
Funktions-aufruf	**()** `ShowWindow(Para1, Para2);`	**()** `ShowWindow(Para1, Para2);`

Bereichs-zugriffs-(Scope-) Operator	**::** `boolean CHelloApp::Init()` `{` `//Def d. Klassen-Methode` `}`	**.** `THelloApp.Init: Boolean;` `begin` `//Def d. Klassen-Methode` `end;`
Inkrement-operator	**++** `als Präfix-Operator:` `++i;` `int x, y;` `x = 3;` `y = ++x;` `// y = 4 und x = 4` `als Postfix-Operator:` `i++;` `int x, y;` `x = 3;` `y = x++;` `// y = 3 und x = 4`	**inc()** `inc(i); oder  i:= i + 1;` `var x, y: Integer;` `x:= 3;` `inc(x);` `y:= x;` `// y = 4 und x = 4` `inc(i); oder i:= i + 1;` `var x, y: Integer;` `x:= 3;` `y:= x;` `inc(x);` `// y = 3 und x = 4`
Dekrement-operator	**--** `als Präfix-Operator:` `--i;` `als Postfix-Operator:` `i--;`	**dec()** `dec(i); oder  i:= i - 1;` `dec(i); oder  i:= i - 1;`
Anmerkung:	Die C++ Ausdrücke, die gleichzeitig eine Wertzuweisung und eine Inkrementierung / Dekrementierung ihres Wertes ausführen, sind nicht exakt zu dem angegebenen Pascal-Code äquivalent.	Die C++ Ausdrücke ermitteln ihren rechtsseitigen Ausdruck nur einmal. Bei komplizierten Ausdrücken mit Seiteneffekten können deswegen beide Formen unterschiedliche Ergebnisse liefern.
SizeOf-Operator	**sizeof()** `sizeof(int)  // = 4`	**SizeOf()** `SizeOf(Integer) // = 4`

Punkt-operator	**.** `struct Adresse` `{` `char *Strasse;` `int  Nummer;` `};` `...` `Adresse adr1;` `adr1.Strasse = "XWeg";` `adr1.Nummer = 43;`	**.** `type` `Adresse = record` `  Strasse: PChar;` `  Nummer: Integer;` `end;` `...` `var adr1: Adresse;` `...` `adr1.Strasse:= 'XWeg';` `adr1.Nummer:= 43;`
Pfeil-operator	**->** `Adresse *adr2; // Zeiger` `...` `adr2 = new Adresse;` `adr2 -> Strasse = "XWeg";` `adr2 -> Nummer = 43;` ` äquivalent dazu ist` `(*adr2).Strasse = "XWeg";` `(*adr2).Nummer = 43;`	**^** `var` `adr2: ^Adresse; //Zeiger` `...` `New(adr2);` `adr2^.Strasse:= 'XWeg';` `adr2^.Nummer:= 43;` ` oder` `adr2.Strasse:= 'XWeg';` `adr2.Nummer:= 43;`
Dereferen-zierungs-operator	***** `int i = 100;` `int *z; // Zeiger auf int` `z = &i;  // z zeigt auf i` `*z = *z + 1; // i = i + 1` `// i ist jetzt 101`	**^** `var i: Integer;` `  z: PInteger;  //Zeiger` `...` `i:= 100;` `z:= @i;   // z zeigt auf i` `z^:= z^ + 1;  // i:= i + 1` `// i ist jetzt 101`
Anmerkung:	****** *ist Präfix-Operator.*	**^** *ist Postfix-Operator.*
Adreß-operator	**&** `int i;` `printf("Adresse von \` `      i = %u", &i);`	**@** `var i: Integer;` `...` `writeln('Adresse von` `i=', LongInt(@i));`

Referenz	**&** `int &r = i;` `/* r ist Referenz von i,` `also nur ein anderer` `Bezeichner (Alias) für` `i, keine neue Variable` `*/`	**Die Definition von Referenzen ist in Object Pascal nicht möglich**
Anmerkung:		Das Wort Referenz wird auch in der Dokumentation zu Object Pascal verwendet, steht hier aber als Synonym für Zeiger (Objektreferenz) und Typ-referenz für Klassen (Klassenreferenz).
New - Operator	für einfache Typen `::opt. new Typ InitListeopt.` `int *int_dyn; // Zeiger` `int_dyn = new int;` oder `int_dyn = new int(7);`	für einfache Typen `New(var P: Pointer)` `var int_dyn: PInteger;` `...` `New(int_dyn);` oder `New(int_dyn);` `int_dyn^:= 7;`
New - Operator	für Vektor (Array) Typen `::opt. new Typ InitListeopt.` `char *str_dyn; // Zeiger` `str_dyn = new char[20];`	*für Vektor (Array) Typen* `GetMem(var P: Pointer,` `Size: Integer)` `var str_dyn: PChar;` `GetMem(str_dyn, 20);` oder alternativ mit "New": `type` `PMyStr20 = ^MyStr20;` `MyStr20 =` `Array[0..19] of Char;` `...` `var str_dyn: PMyStr20;` `...` `New(str_dyn);`
Anmerkung:	Der new Operator wird in C++ auch dazu verwendet, dynamische Objekte anzulegen.	In Object Pascal dagegen werden dynamische Objekte ausschließlich mit dem Constructor Create instanziert!

Delete-Operator	für einfache Typen ::opt. **delete** Pointer **delete** int_dyn;	für einfache Typen **Dispose(**var P: Pointer**)** **Dispose(**int_dyn**)**;
Delete-Operator	für Vektor (Array) Typen ::opt. **delete** [] Pointer **delete** [] str_dyn;	für Vektor (Array) Typen **FreeMem(**var P: Pointer, [Size: Integer]**)** **FreeMem(**str_dyn, 20**)**;
Anmerkung:	Der Scope-Operator :: muß benutzt werden, wenn der new / delete- Operator überladen wurde und der globale new / delete Operator aufgerufen werden soll.	
Konditionaler Operator	*e1* **?** *e2* **:** *e3* Wirkung: if (*e1*) e2; else e3; z = (a > b) **?** a **:** b; ist äquivalent zu if (a > b) z = a; else z = b;	**kein solcher Operator** Gleichwertiger Ersatz ist durch Nutzung des if / else Ausdrucks möglich. if (a > b) then z:= a else z:= b;
Komma-operator	**Ausdruck1, Ausdruck2, ..., AusdruckN** Der Gesamt-Ausdruck wird von links nach rechts ausgewertet, und der Wert des gesamten Ausdrucks entspricht dem Wert von **AusdruckN**. int a, b, c; a = 0; b = 0; c = (a++, b++, b++, b++); // a = 1, b = 3, c= 2	**kein solcher Operator** Einen gleichwertigen Ersatz erhält man durch das sequentielle Aufrufen der N Ausdrücke. var a, b, c: Integer; a:= 0; b:= 0; inc(a); inc(b); inc(b); c:= b; inc(b); // a = 1, b = 3, c= 2

Klassen-element-zeiger	.* (bei statischen Objekten) ->* (bei dynamischen Objekten) class Test{ public: int i; }; void main() { // Klassenzeiger für // Klasse Test int Test :: *DatZeiger; // soll auf i zeigen. // Ist in allen Ob- // jekten der Klasse // Test nutzbar. DatZeiger = &Test :: i; // statisches Objekt a // erzeugen Test a; // dynamisches Objekt b // erzeugen Test* b = new Test; // dynamisches Objekt c // erzeugen Test* c = new Test; // Datenbelegung a.i = 11; b->i = 26; c->i = 45; printf("a.i = %i", a.*DatZeiger); printf("b->i = %i", b->*DatZeiger); printf("c->i = %i", c->*DatZeiger); } Ausgabe: a.i = 11 b->i = 26 c->i = 45	**Klassenelementzeiger auf Daten nicht definiert.** Klassenelementzeiger auf Methoden erfolgt durch Methodenzeiger-Typ und den Punktoperator. Siehe dazu Beispiel Methodenzeiger im Abschnitt "2.3.2 Standardtypen".
Anmerkung:	Klassenelementzeiger (class member pointer) können nicht nur auf Klassen-Daten, sondern auch auf Klassen- Methoden zeigen.	

	typeid()	**wird nicht benötigt,**
Objekt-Typ-Informations-Operator	typeid() liefert als Ergebnis eine Klasse "type_info", die die Run-Time-Type-Informationen (RTTI) enthält. `#include <typeinfo.h>` `class Test{` `// leere Klasse` `};` `void main()` `{` `Test* a = new Test;` `printf("Klasse heißt %s",` `  **typeid(**a**)**.name());` `}` `Bildschirm-Ausgabe:` `Klasse heißt class Test *`	weil alle Objekte in Object Pascal immer alle Typ-Informations-Methoden besitzen. `type` `Test = class` `  // leere Klasse` `end;` `var a: Test;` `begin` `a:= Test.Create;` `writeln('Klasse heißt ',` `  a.ClassName);` `end.` `Bildschirm-Ausgabe:` `Klasse heißt Test`
dynamische Objekt-Typ-prüfung	**über Vergleich der RTTI** `Test* a = new Test;` `if (**typeid(**a**) ==**` `  **typeid(**Test***))**` `printf("selbe Klasse");` `else` `printf("andere Klasse");`	***object* is *class*** `procedure Pruef(Sender:` `  TObject);` `begin` `if **Sender is Test** then` `  writeln('selbe Kl.')` `else` `  writeln('andere Kl.');` `end;` `...` `var a: Test;` `begin` `a:= Test.Create;` `Pruef(a);` `end.`

dynamische Objekt-Typ-konver-tierung (zur Lauf-zeit)	**dynamic_cast <*Typ*> (*Bez.*)** `class Tst1{};` `class Tst2: public Tst1{};` `...` `Tst1* a;` `Tst2* b = new Tst2;` `a= dynamic_cast<Tst1*>(b);`	***object* as *class*** `type` `Tst1 = class end;` `Tst2 = class(Tst1) end;` `...` `var a: Tst1;` `  b: Tst2;` `...` `b:= Tst2.Create;` `a:= b as Tst1;`
statische Typ-konver-tierung (zur Über-setzungszeit)	***typ*()** `int i = 12;` `char c = char(i);` `oder äquivalent` `char c = (char)i;` Anmerkung: Statt des Cast-Operators () sollten die neueren Operatoren static_cast, reinterpret_ cast und const_cast verwendet werden. [6, Seite 429]	***typ*()** `var i: Integer;` `  c: Char;` `...` `c:= Char(i);` Anmerkung: Der Cast-Operator () in Object Pascal entspricht in seiner Wirkungsweise dem Opera-tor static_cast in C++.
statische Typ-konver-tierung (zur Über-setzungszeit)	**static_cast <*Typ*> (*Bez.*)** `int a;` `char b[10];` `a = static_cast<int>(b);` `/* wird durch Compiler abgewiesen, da es gänz-lich verschiedene Typen sind */`	***typ*()** `var` `a: Integer;` `b: Array[0..9] of Char;` `a:= PChar(b);` `{ wird durch Compiler abgewiesen, da es gänzlich verschiedene Typen sind }`
statische Typkonver-tierung (zur Über-setzungszeit)	**reinterpret_cast <*Typ*> (*Bz*)** `int a;` `char b[10];` `a = reinterpret_cast<int>(b);` `/* wird vom Compiler ak-zeptiert (um-interpre-tiert) */` `const_cast <Typ> (Bez.)`	Operatoren, die "reinterpret_cast" und "const_cast" entsprechen würden, sind in Object Pascal nicht definiert, da deren Wirkungsweise dem typensicheren Konzept von Pascal diametral entgegen stehen. Um trotzdem nicht zu unflexibel zu werden, definiert Object Pascal den Typ *Variant*. Variablen dieses Typs können ein Wert darstellen, der seinen Typ dynamisch während der Laufzeit ändern kann.

	//nur wegen Abwärtskompa- //tibilität zu C //definiert	

In beiden Sprachen besitzen Funktionsaufrufe nach Scope- und Vektoroperator die höchste Priorität. Um logisch falsche Ausdrücke zu vermeiden, sollten logisch zusammengehörende Ausdrücke in Klammern () zusammengefaßt werden.

C++ gestattet das sogenannte Überladen von Operatoren innerhalb von Klassen. Eine Erläuterung folgt im Abschnitt "2.3.6 Klassen und Objekte".

Zurück zum Inhaltsverzeichnis
Weiter in Kapitel 2.3.2

2.3.2 Standardtypen

Praktisch jeder Bezeichner, so z. B. jede Variable, muß in beiden Sprachen mit einem Typ assoziiert werden. Datentypen können durch die Attribute Name, Gültigkeits- und Sichtbarkeitsbereich, Konstruktionsregel, Wertebereich und zulässige Operationen charakterisiert werden.

Elementare Datentypen (simple types) bilden die Grundlage für die Erzeugung weiterer, benutzerdefinierter, abgeleiteter Typen. Integer, Real, Char und Boolean sind Beispiele für elementare Typen. Die wichtigsten in Visual C++ und Object Pascal sind:

Ganzzahl-Typen:

VC++	Object Pascal	Länge in Win32 (Intel)	darstellbarer Bereich
char	Char = AnsiChar	1 Byte	ASCII-Code 0 . . 255
wchar_t	WideChar	2 Byte	Uni-Code 0 . . 65535
signed char	ShortInt	1 Byte	-128 . . 127
unsigned char	Byte	1 Byte	0 . . 255
short	SmallInt	2 Byte	- 32768 . . 32767
unsigned short	Word	2 Byte	0 . . 65535
int = long	Integer = LongInt	4 Byte	-2147483648 . . 2147483647
-	Cardinal	4 Byte	0 . . 2147483647
unsigned int = unsigned long	-	4 Byte	0 . . 4294967296
__int64	Comp	8 Byte	$-2^{63} + 1 \,..\, 2^{63} - 1$
CURRENCY	Currency	8 Byte	$-9{,}22337 \cdot 10^{14} \,..\, 9{,}22337 \cdot 10^{14}$

SizeOf(Cardinal) liefert 4 Byte als Ergebnis. Dieser Typ benutzt aber eigentlich nur 31 Bit; das Vorzeichen Bit wird von Delphi 2 einfach ignoriert.

Currency stellt Dezimalzahlen immer mit 4 Dezimalstellen dar. Er wird intern als Integertyp realisiert, so daß das Intervall zwischen zwei darstellbaren Zahlen im gesamten darstellbaren Zahlenbereich konstant 0,0001 beträgt. Er eignet sich gut für die Repräsentation von Geldbeträgen.

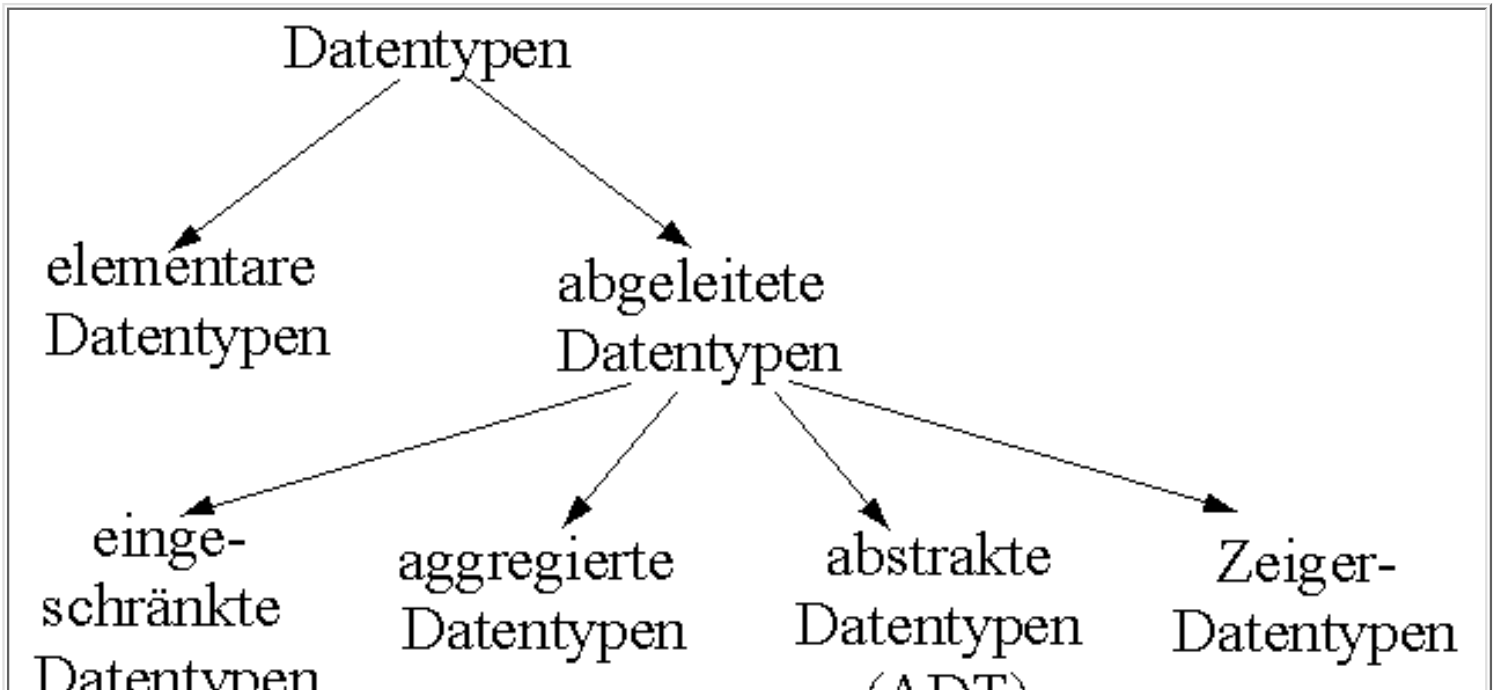
Dezimalzahl-Typen:

VC++	Object Pascal	Länge in Win32 (Intel)
float	Single	4 Byte
double	Double	8 Byte
long double	Extended	10 Byte

Boolsche Typen:

VC++	Object Pascal	Länge in Win32 (Intel)
boolean	Booleans	1 Byte
BOOL	BOOL = LongBool	4 Byte

Alle anderen, abgeleiteten Datentypen werden unter Verwendung der elementaren Datentypen gebildet.



Mögliche Einteilung der Datentypen nach [11]

Eingeschränkte Datentypen sind dadurch gekennzeichnet, daß der Wertebereich eines vorhandenen Basisdatentyps eingeschränkt wird. Dies kann durch die Angabe eines Intervalls (Teilbereichstyp) oder die explizite Aufzählung der zulässigen Werte (Aufzählungstyp) geschehen.

Ein eingeschränkter Datentyp in Objekt Pascal ist der ***Teilbereichsdatentyp*** (subrange type). Die Definition eines Teilbereichstyps spezifiziert den kleinsten und den größten Wert im entsprechenden Teilbereich: type_name = wert_min .. wert_max;

```
type
  DoppelZiffern = 10..99;

var Zahl: DoppelZiffern;
Zahl:= 55;
Zahl:= 123;  // Fehler, wird vom Compiler abgewiesen
```

Die Nutzung des Teilbereichsdatentyps erspart Prüfungen der Art:

```
var Zahl1: Integer;
    Zahl2: Integer;
    ...
if (Zahl2 >= 10) and (Zahl2 <= 99) then
  Zahl1:= Zahl2
else
  Fehlerbehandlung;
```

Falls einer Teilbereichstyp-Variablen zur Laufzeit ein Wert außerhalb des gültigen Bereichs zugewiesen wird, wird eine Exception der Art *ERangeError* ausgelöst, sofern das Programm mit aktivierter Bereichsüberprüfung übersetzt wurde. Exceptions werden im Abschnitt "2.3.4 Anweisungen" besprochen.

Den Teilbereichstypen wird ungerechtfertigter Weise allgemein sehr wenig Beachtung geschenkt. Immerhin tragen sie durch ihre aktive Bereichsüberprüfung während der Übersetzungs- und der Laufzeit dazu bei, mögliche programminterne Inkonsistenzen zu vermeiden bzw. helfen dabei, solche aufzudecken. Visual C++ besitzt keinen Teilbereichsdatentyp.

Dagegen kennen beide Sprachen ***Aufzählungen*** (enumeration type). Aufzählungen werden durch eine Reihe symbolischer Bezeichner gebildet, die alle Integerkonstanten repräsentieren. Der erste Bezeichner bekommt automatisch den Wert 0, der zweite den Wert 1 usw. Durch Angabe konstanter Ausdrücke und Zuweisungen an die Bezeichner können in C++ auch andere Werte für die Integer-Konstanten definiert werden.

C++	Pascal
`enum Farbe {Kreuz, Karo,` `Herz, Pik};` `...` `Farbe Karte;` `Karte = Herz;` `printf("%d", Karte);` Ausgabe: 2	`type` `Farbe = (Kreuz, Karo,` `Herz, Pik);` `...` `var Karte: Farbe;` `Karte:= Herz;` `writeln(Karte);` Ausgabe: 2
`enum Antwort` `{` `Nein,` `Ja,` `KannSein = -1` `};`	

Die Werte **aggregierter Datentypen** (komplexer Datentypen, composite types) bestehen meist aus verschiedenen Komponenten, evtl. mit jeweils eigenem Datentyp. Die Komponenten dieses Typs sind sichtbar, alle zulässigen Operationen sind uneingeschränkt auf sie anwendbar. Zu dieser Art Typen zählen Vektoren, Records, das Set und Varianten.

Vektoren, häufig auch Arrays oder Felder genannt, besitzen eine bestimmte Anzahl von Elementen eines bestimmten Typs. Sie können ein- oder mehrdimensional sein. Ihre Elemente müssen in C++ mit den Werten 0 bis (Anzahl-1) indiziert sein. In Pascal hingegen können obere und untere Indexgrenze frei gewählt werden.

Visual C++ und Object Pascal verwenden den Begriff "Feld" auch im Zusammenhang mit Objekt-Membern. Um Verwechslungen vorzubeugen, sollten hier bevorzugt die Begriffe Vektor oder Array verwandt werden.

	C++	Pascal
ein-dimensional	`float v[3];` `// Vektor von 3 reellen` `// Zahlen v[0], v[1], v[2]`	`var v: Array[6..8]of Single;` `// Vektor von 3 reellen` `// Zahlen v[6], v[7], v[8]`
mehr-dimensional	`int a[2][3];` `// belegter Speicherplatz` `// ist 2*3*sizeof(int) Byte`	`var a: Array[0..1, 0..2]` `of Integer;` `// belegter Speicherplatz` `// ist 2*3*SizeOf(Integer)` oder alternativ `var a: Array[0..1] of` `Array[0..2]` `of Integer;`

Die Elemente der Arrays können in C++ problemlos auch über Zeiger angesprochen werden. Object Pascal unterstützt diese Art des Zugriffs nur bei Arrays vom Typ Char. C++ gestattet die Initialisierung mehrerer Elemente des Arrays in *einem* Ausdruck durch eine kommaseparierte Liste:

```
int z[4] = {9, 8, 7}  // z[0]=9  z[1]=8  z[2]=7
```

Elemente, die in der Initialisierungsliste keinen Anfangswert zugewiesen bekommen (hier z[4]), werden vom Compiler zu Null initialisiert.

Direkt-Zuweisungen der Inhalte von Array-Variablen gleicher Struktur sind in Pascal möglich, in C++ durch das Umkopieren des Array-Speichers realisierbar:

C++	Pascal

```
#include <memory.h>

int x[8], y[8];

x[0] = 8;
x[1] = 14;
x[2] = 3;

y = x; // Fehler
memcpy(y, x, sizeof(x));
```

```
var x, y: Array[1..8] of Integer;

x[1]:= 8;
x[2]:= 14;
x[3]:= 3;

y:= x;
```

Ein Problem stellen Überschreitungen der Indexgrenzen in C++ dar. In einem 4 Elemente umfassenden Array wird ein Zugriff auf ein fünftes, nicht gültiges Element, in der Art

```
z[4]:= 10;
```

anstandslos und ohne Warnhinweise von C++ übersetzt, obwohl ein Zugriff auf einen Speicherbereich außerhalb des Arrays erfolgt! Das ist ein Programmierfehler, der einem Programmierer leicht unterläuft, der aber schwerwiegende Folgen nach sich ziehen kann, die im ungünstigsten Fall zunächst nicht einmal bemerkt werden. Die Fehlersuche gestaltet sich entsprechend schwierig. Object Pascal unterbindet die Überschreitung von Array-Grenzen zur Übersetzungs- und zur Laufzeit. Letzteres jedoch nur dann, wenn das Programm mit gesetzter Projektoption "Bereichsüberprüfung" compiliert wurde. Wird zur Laufzeit die Array-Grenze überschritten, so wird wiederum eine Exception der Art *ERangeError* ausgelöst.

Am 2. April 1981 äußerte Brian W. Kernighan, einer der Entwickler von C, seine Meinung über Standard-Pascal in einem Aufsatz mit dem Titel "*Warum Pascal nicht meine bevorzugte Programmiersprache ist"* [12]. Der gravierendste sprachliche Mangel von Pascal ist seiner Meinung nach die Tatsache, daß Funktionen und Prozeduren als Parameter nur Arrays mit fester Größe akzeptieren. Durch diese Einschränkung ist es in Standard-Pascal nicht möglich Funktionen zu schreiben, die verschieden große Arrays als Parameter akzeptieren. Object Pascal erweitert Standard-Pascal deshalb an dieser Stelle und führt "Offene Array-Parameter" ein (Open Array). C++ kennt derartige Typen unter dem Begriff der "Incomplete Types".

VC++	Object Pascal

```
void DoIt(int Arr[])
{
  printf("%d\n", Arr[1]);
}



void main()
{
  int z2[2] = {1, 2};
  int z2[3] = {1, 2, 3};
  DoIt(z2);
  DoIt(z3);
}
```

```
procedure DoIt(Arr: Array of
                        Integer);
begin
  writeln(Arr[1]);
end;

var z2: Array[0..1] of Integer;
    z3: Array[0..2] of Integer;
begin
  z2[0]:= 1; z2[1]:= 2;
  z3[0]:= 1; z3[1]:= 2; z3[2]:= 3;
  DoIt(z2);
  DoIt(z3);
end.
```

Die Anzahl der Elemente eines offenen Arrays können in Object Pascal über die Funktion High(OpenArray) abgefragt werden.

Eine Eigenheit von C++ bei der Parameterübergabe an Funktionen wird deutlich, wenn man die Ausgabe der folgenden Programmstücke vergleicht:

VC++

```
void DoIt(int a, int Arr[])
{
  printf("%d %d\n", a, Arr[0]);
  a = 0;
  Arr[0] = 0;
}


void main()
{
  int z = 2;
```

Object Pascal

```
procedure DoIt(a: Integer; Arr:
             Array of Integer);
begin
  writeln(a, Arr[0]);
  a:= 0;
  Arr[0]:= 0;
end;

var z: Integer;
   zArr: Array[0..1] of Integer;
begin
```

```
  int zArr[2] = {2};

  printf("%d %d\n",z, zArr[0]);
  DoIt(z, zArr);
  printf("%d %d\n",z, zArr[0]);
}
```

Ausgabe:

2 2

2 2

2 0

```
  z:= 2;
  zArr[0]:= 2;
  writeln(z, zArr[0]);
  DoIt(z, zArr);
  writeln(z, zArr[0]);
end;
```

Ausgabe:

2 2

2 2

2 2

Der Wert des Arrays wurde in C++ durch den Aufruf der Prozedur DoIt global verändert, obwohl der Funktionsparameter syntaktisch als Call-By-Value - Parameter angegeben wurde. C++ reserviert generell bei Funktionsaufrufen für alle Arrays keinen Stack-Speicher, sondern übergibt nur einen Zeiger an die Funktion. Object Pascal verhält sich entsprechend, wenn der Funktionsparameter als VAR-Parameter deklariert wird: **var** Arr: Array of Integer. Der Array-Parameter wird in diesem Fall auch als Call-By-Reference Wert übergeben; eine lokale Kopie auf dem Stack wird dann nicht angelegt, sondern ein Zeiger verweist auf den Speicher an der Aufrufstelle.

In Pascal wurde ein weiterer aggregierter Datentyp implementiert, das ***Set***. Es definiert eine Sammlung von Objekten desselben Ordinaltyps mit maximal 256 möglichen Werten. Die eckigen Klammern bezeichnen in Pascal den Mengenkonstruktor und haben nichts mit dem Vektoroperator in C++ zu tun.

```
var
  KleinbuchstabenMenge: Set of 'a'..'z';
begin
  KleinbuchstabenMenge:= ['b', 'm', 'l'];
  // jetzt wird 'm' aus der Menge entfernt
  KleinbuchstabenMenge:= KleinbuchstabenMenge - ['m'];
  // die entstandene Menge ist ['b', 'l']
  KleinbuchstabenMenge:= [];   //leere Menge
end;
```

Der Operator in gestattet Tests der Art

```
if (c = Blank) or (c = Tab) or (c = NewLine) then ...
```

in der leichter lesbaren Form

```
if c in [Blank, Tab, NewLine] then ...
```

zu schreiben. Ein vordefinierter Typ, der dem Set entspricht, existiert in C++ nicht.

Ein anderer aggregierter Datentyp ist der Verbundtyp. In C++ wird er ***Struct*** und in Pascal ***Record*** genannt. Er stellt ein Aggregat von Elementen beliebigen Typs dar und gruppiert sie unter einem einzigen Namen. Die Elemente werden in C++ "Member", in Pascal "Felder" genannt.

C++	Pascal
`struct WohntIn` `{` `  int PLZ;` `  char* Stadt;` `};` `WohntIn HeimatOrt;` `...`	`type` `  WohntIn = record` `  PLZ: Integer;` `  Stadt: PChar;` `  end;` `var HeimatOrt: WohntIn;` `...`

Der Zugriff auf die Member / Felder erfolgt bei statischen Variablen über den Punktoperator, bei dynamischen Variablen über den Pfeiloperator (siehe Beispiel in der Tabelle der Operatoren weiter vorn). Verkettete Listen (Linked Lists) werden häufig mit Hilfe dynamischer Variablen dieses Datentyps realisiert.

Visual C++ und Delphi vergrößern den Verbunddatentyp beim Übersetzen standardmäßig so, daß den Membern des Structs / den Feldern des Records solche Adressen zugewiesen werden können, die "natürlich" für die Member- / Feldtypen sind und die das beste Laufzeitverhalten bei modernen Prozessoren bewirken. Bekannt unter dem Begriff des "Structure Alignment" wird so z. B. ein Struct / Record, der eigentlich nur 5 Byte belegen würde,

auf 8 Byte vergrößert. Beide Sprachen bieten jedoch die Möglichkeit, dieses Verhalten zu unterbinden.

VC++	Object Pascal
`struct WohntIn` `{` `  int PLZ;  // 4 Byte` `  char Landkennung; // 1 Byte` `};` `...` `printf("%d", sizeof(WohntIn));` `// **8 Byte**`	`type` `  WohntIn =  record` `  PLZ: Integer;  // 4 Byte` `  Landkennung: Char; // 1 Byte` `  end;` `  ...` `writeln(sizeof(WohntIn));` `// **8 Byte**`
`#pragma pack(1)` `struct WohntIn` `{` `  int PLZ;  // 4 Byte` `  char Landkennung; // 1 Byte` `};` `#pragma pack(8)` `...` `printf("%d", sizeof(WohntIn));` `// **5 Byte**`	`type` `  WohntIn = **packed** record` `  PLZ: Integer;  // 4 Byte` `  Landkennung: Char; // 1 Byte` `  end;` `  ...` `writeln(sizeof(WohntIn));` `// **5 Byte**`

Structs können in C++ neben Daten auch Funktionen aufnehmen. Man zählt sie dann zu abstrakten Datentypen (ADT).

Ein Spezialfall des Struct- / Record- Typs stellt der Typ ***Union*** / ***varianter Record*** dar. Ein Union deklariert in C++ einen struct, in dem jedes Member dieselbe Adresse besitzt. Man kann sagen: ein Union gestattet die Aufnahme verschiedener, vordefinierter Typen. Ein varianter Record deklariert in Pascal einen Record, in dem unterschiedliche Typen in unterschiedlicher Anzahl gespeichert werden können. Anders gesagt: variante Records gestatten die Aufnahme verschiedener, vordefinierter Typen und neu zu bildender Record-Typen. Auch hier besitzt jedes Feld bzw. jeder Subrecord dieselbe Adresse.

Der Union- / Record- Typ ist so groß, daß er gerade das größte Member / die größte Felderstruktur aufnehmen kann.

C++	Pascal
union FreiZahl { int i; double d; char* s; }; FreiZahl z; // z wird als Integer- // Typ behandelt z.i = 5; // z wird als Double- // Typ behandelt z.d = 5.0; // z wird als String- // Typ behandelt z.s = '5,0'; printf("%s", z.s); // Ausgabe: 5,0 printf("%d", z.i); // Fehl-Ausgabe // zur Laufzeit,weil z // inzwischen nicht mehr // als Integer behandelt // wird	type FreiZahl = **record** **Case** Integer Of 0: **(**i: Integer**)**; 1: **(**d: Double**)**; 2: **(**s: PChar**)**; end; var z: FreiZahl; // z wird als Integer- // Typ behandelt z.i:= 5; // z wird als Double- // Typ behandelt z.d:= 5.0; // z wird als String- // Typ behandelt z.s:= '5,0'; writeln(z.s); // Ausgabe: 5,0 writeln(z.i); // Fehl-Ausgabe // zur Laufzeit,weil z // inzwischen nicht mehr // als Integer behandelt // wird

Pascal gestattet die Aufnahme varianter und nichtvarianter Teile in *einem* Record. Einen entsprechenden Typ erhält man in C++, indem man unbenannte (anonyme) Unions innerhalb von Structs deklariert.

C++	Pascal

```
struct Person
{
  unsigned short Nummer;
  union
  {
    struct
    { char Geburtsort[41];
      int Geburtsjahr;};
    struct
    { char Land[26];
      char Stadt[31];
      int PLZ;};
  };
};
```

```
Person = record
  Nummer: Word;
  Case Integer Of
    0: (Geburtsort: String[40];
        Geburtsjahr: Integer);
    1: (Land: String[25];
        Stadt: String[30];
        PLZ: Integer);
end;
```

Der Ausdruck **Case Integer Of** in Pascals varianten Records kann leicht mißverstanden werden. Man kann ihn sich zum Verständnis des varianten Records aber einfach wegdenken. Er stellt nur einen syntaktischen Ausdruck dar, der es dem Compiler erlaubt zu verstehen, was hier deklariert wird: bestimmte Felder (oder Feldergruppen) im Record, die denselben Speicherplatz belegen. Der Compiler versucht niemals herauszufinden, welcher Case-Zweig benutzt werden soll. Das liegt vollkommen in der Hand des Entwicklers. Wenn man in einer Variablen vom Typ Person dem Feld Geburtsort einen Wert zugewiesen hat, dann ist man im Programm *selbst* dafür verantwortlich, dieses Feld nicht als Land zu interpretieren.

Aus diesem Grund müssen Unions und variante Records stets mit Vorsicht eingesetzt werden. Eine Typprüfung durch den Compiler ist nicht möglich.

Unions und variante Records bilden die Grundlage für den Typ Variant. Mit dem Typ Variant können Werte dargestellt werden, die ihren Typ dynamisch ändern. Varianten werden auch dann verwendet, wenn der tatsächliche Typ, mit dem gearbeitet wird, zur Übersetzungszeit unbekannt ist. Man bezeichnet Varianten auch als sichere "untypisierte" Typen (safe "non-typed" types) oder als "spät gebundene Typen" (late-bound types).

Intern bestehen sie aus einem Datenfeld für den zu speichernden Wert und einem weiteren Feld, das den Typ angibt, der im Datenfeld enthalten ist. Varianten können ganze oder reelle Zahlen, Stringwerte, boolesche Werte, Datum-Uhrzeit-Werte sowie OLE-Automatisierungsobjekte enthalten. Object Pascal unterstützt explizit Varianten, deren Werte Arrays aufnehmen können, deren Größe und Dimensionen wechseln und die Elemente eines der oben genannten Typen enthalten können. Mit dem speziellen Variantenwert VT_EMPTY in Visual C++ und varEmpty in Object Pascal wird angezeigt, daß einer Variante bislang noch keine Daten zugewiesen wurden. Der grundlegende Varianten-Typ wird in Visual C++ nur sehr schlecht unterstützt. Der Entwickler ist für Initialisierungen und die Zuweisung an das Feld, das die Typinformation angibt, selbst verantwortlich. Eine spezielle Klasse ColeVariant kapselt den Variant-Typ ein und bietet eine bessere Unterstützung für Varianten (speziell für OLE-Verfahren).

VC++	Object Pascal

```
#include <oaidl.h>

VARIANT v1, v2;

VariantInit(&v1);
VariantInit(&v2);

// Typ short setzen
v1.vt = VT_I2;
v1.iVal = 23;

// Typ Double setzen
v1.vt = VT_R8;
v1.dblVal = 23.75;

VariantCopy(&v2, &v1);
// v2 = v1
printf("Wert=%g  Typ=%d\n",
       v2.dblVal, v2.vt);
```

```
var v1, v2: Variant;

v1:= 23;
// varInteger
writeln(VarType(v1));

v1:= 23.75;
// varCurrency
writeln(VarType(v1));

v2:= v1;

writeln(v2 + ' ' + IntToStr(
        VarType(v2)));
```

Variablen vom Typ Variant belegen in beiden Sprachen 16 Byte; die Strukturen sind identisch.

Um die Flexibilität dieses Typs darzustellen, sei ein weiteres Beispiel angeführt. Eine Funktion Addiere übernimmt die (unbestimmten) Werte eines Varianten-Arrays, addiert sie und liefert die Summe als Ergebnis im Integer-Format.

```
function Addiere(Zahlen: Array of Variant): Integer;
var
  i, Sum: Integer;
begin
  Sum:= 0;
  for i:= 0 to High(Zahlen) do Sum:= Sum + Zahlen[i];
  Addiere:= Sum;
end;
```

Ein Aufruf könnte dann z.B. in der Form erfolgen:

```
writeln(Addiere([2,  8.00, ' 5,00 DM']));
```

Hier wird eine Integer-Zahl, eine Zahl vom Typ Double und ein String miteinander addiert. Die Ausgabe lautet 15.

Operationen auf Varianten laufen langsamer ab als Operationen auf statisch typisierten Werten. Um eine Vergleichsmöglichkeit zu schaffen, wurde eine zweite Funktion "AddiereStatisch" geschrieben, deren Parameter als statische Typen deklariert wurden. Eigene Messungen haben gezeigt, daß ein Aufruf der Funktion AddiereStatisch (mit statischen Parametern) 0,66 µs und ein Aufruf von Addiere (mit varianten Parametern) 40,65 µs benötigt, letzterer also ungefähr 60 mal langsamer ist. Die Messungen erfolgten auf einem Rechner mit Pentium-Prozessor (150 MHz Taktrate), 8 kByte Primär-Cache und 512 kByte Sekundär-Cache. Die Funktionen wurden jeweils 1 Mio. mal aufgerufen und die Zeiten gemessen. Es kann davon ausgegangen werden, daß die betreffenden Programmteile ausschließlich im schnellen Cache-RAM abgelaufen sind.

Prozedurale Datentypen ermöglichen es, Prozeduren und Funktionen als Einheiten zu behandeln, sie Variablen zuzuweisen und als Parameter übergeben zu können. Man unterscheidet zwei Kategorien von prozeduralen Typen: Zeiger auf globale Prozeduren und Methodenzeiger.

Zeiger auf globale Prozeduren sind Zeiger auf Prozeduren und Funktionen außerhalb von Klassen. Der Zeiger speichert die Adresse der Funktion oder Prozedur. Hier ein Beispiel, bei dem ein prozeduraler Typ als Parameter für die Funktion SetWindowsHook deklariert wird:

```
typedef
  LRESULT(CALLBACK* HookProc)(int code, WPARAM wParam,
                              LPARAM lParam);


int SetWindowsHook(int aFilterType, HookProc aFilterProc);
{...
/* bei einem späteren Aufruf von SetWindowsHook erwartet der Compiler beim Parameter aFilterProc die Adresse einer Funktion, die genauso deklariert wurde wie HookProc. Die Adresse einer Funktion erhält man durch den Adressoperator &
*/

}
```

Obj. Pascal

```
type
  HookProc = function(code: Integer; wparam: WPARAM;
                      lparam: LPARAM): LRESULT;

function SetWindowsHook(aFilterType: Integer;
                        aFilterProc: HookProc): Integer;
begin
  ...
{bei einem späteren Aufruf von SetWindowsHook erwartet der Compiler beim Parameter aFilterProc die Adresse einer Funktion, die genauso deklariert wurde wie HookProc. Die Adresse einer Funktion erhält man durch den Adressoperator @}

end;
```

So deklarierte Callback- (Rückruf-) Funktionen werden in Windows gewöhnlich nicht vom *eigenen* Programm, sondern von Windows *selber* aufgerufen (etliche Funktionen im Windows-API funktionieren nur in Zusammenarbeit mit vom Entwickler zu erstellenden Callback-Funktionen).

Eine Variable vom Typ Methodenzeiger kann auf eine Prozedur oder Funktionsmethode eines Objekts oder einer Klasse zeigen. In C++ stellen Methodenzeiger allerdings in Wahrheit keine Zeiger dar, sondern sind ein Adreß-Offset bezüglich eines Objektes auf ein Element dieses Objektes. Die Bezeichnung "Methodenzeiger" wurde dem eigentlich treffenderen Begriff "Methoden-Offset" vorgezogen, um die syntaktische Parallele zur Zeigersyntax und zum globalen Funktionszeiger zu unterstreichen [6, S. 387]. In Object Pascal dagegen stellen Methodenzeiger wirkliche Zeiger dar. Sie werden in der Form von zwei Zeigern realisiert: der erste Zeiger speichert die Adresse einer Methode, der zweite (versteckte Zeiger) speichert eine Referenz auf das Objekt, zu dem die Methode gehört. Prozedurale Typen werden in Object Pascal durch die Anweisung of object deklariert.

Syntax und Funktionsweise von Methodenzeigern sollen an einem Beispiel gezeigt werden. Der Methodenzeiger heißt Aufruf und soll auf sehr einfache Funktionen zeigen können, die keinen Wert als Parameter erwarten und auch keinen Wert als Ergebnis liefern. Im Konstruktor der Klasse CMyObject / TMyObject wird dem Methodenzeiger Methode1 zugewiesen, d.h. er wird mit Methode1 in der Klasse "verbunden". Ein späterer Aufruf des Methodenzeigers wird den Aufruf der Methode1 zur Folge haben. Die Implementierung der Methode1 setzt ihrerseits den Methodenzeiger auf eine andere Methode2. Ein später folgender Aufruf des Methodenzeigers wird nun den Aufruf von Methode2 zur Folge haben. Methode2 setzt den Methodenzeiger wieder auf Methode1 zurück.

VC++	Object Pascal
class **CMyObject** { private: void Methode1(void); void Methode2(void); public: // Aufruf ist eine // Methodenzeiger- // Variable **void(CMyObject::*Aufruf)** (void); CMyObject(); }; CMyObject::CMyObject() { // zuerst soll Aufruf auf // Methode1 zeigen Aufruf = Methode1; // identisch könnte man // auch schreiben // Aufruf = &CMyObject::Methode1; }; void CMyObject::Methode1(void) { printf("Methode 1 aufgerufen"); // DYNAMISCHER Methoden- // Wechsel Aufruf = Methode2; }; void CMyObject::Methode2(void) { printf("Methode 2 aufgerufen"); // DYNAMISCHER Methoden- // Wechsel Aufruf = Methode1; };	type **TMyObject** = class private procedure Methode1; procedure Methode2; public // Aufruf ist eine // Variable vom Typ // Methodenzeiger **Aufruf: procedure of object;** constructor Create; end; Constructor TMyObject.Create; begin // zuerst soll Aufruf auf // Methode1 zeigen Aufruf:= Methode1; // identisch könnte man auch // schreiben // Aufruf:= Self.Methode1; end; procedure TMyObject.Methode1; begin writeln('Methode 1 aufgerufen'); // DYNAMISCHER Methoden- // Wechsel Aufruf:= Methode2; end; procedure TMyObject.Methode2; begin writeln('Methode 2 aufgerufen'); // DYNAMISCHER Methoden- // Wechsel Aufruf:= Methode1;

```
void main()
{
  CMyObject* myObj;

  myObj = new CMyObject;
  // Operator Klassen-
  // Member-Zeiger  ->*
  (myObj->*myObj->Aufruf)();
  (myObj->*myObj->Aufruf)();
  (myObj->*myObj->Aufruf)();
};
```

Ausgabe:

Methode 1 aufgerufen

Methode 2 aufgerufen

Methode 1 aufgerufen

```
end;

var
  myObj: TMyObject;

begin
  myObj:= TMyObject.Create;
  // Aufruf, als ob "Aufruf"
  // eine Methode wäre
  myObj.Aufruf;
  myObj.Aufruf;
  myObj.Aufruf;
end.
```

Ausgabe:

Methode 1 aufgerufen

Methode 2 aufgerufen

Methode 1 aufgerufen

Durch Methodenzeiger ist es möglich, zur Laufzeit bestimmte Methoden von Objektinstanzen aufzurufen.

Durch den zweiten, versteckten Zeiger bei Object Pascals Methodenzeigern wird immer auf die Methode eines ganz konkreten Objekts verwiesen. Methodenzeiger ermöglichen dadurch das Erweitern eines Objekts durch Delegieren eines bestimmten Verhaltens an ein anderes Objekt. Das Ableiten eines neuen Objekts und Überschreiben von Methoden kann so manchmal umgangen werden.

Delphi verwendet Methodenzeiger, um Ereignisse zu implementieren. Alle Ereignismethoden, die den Objekten in der Anwendung beim visuellen Programmieren mit Hilfe des Objektinspektors zugewiesen werden, beruhen auf Methodenzeigern. Alle Dialogelemente erben beispielsweise eine dynamische Methode namens *Click* zur Behandlung von Mausklickereignissen. Die Implementierung von *Click* ruft, sofern vorhanden, die Mausklick-Ereignisbehandlungsroutine des Anwenders auf. Der Aufruf erfolgt mit Hilfe des Methodenzeigers *OnClick*. Ob der Anwender eine eigene Mausklick-Ereignisbehandlungsroutine definiert hat, wird dadurch erkannt, daß der Methodenzeiger auf einen gültigen Wert verweist, also nicht den Wert nil besitzt.

```
procedure TControl.Click;
begin
```

```
  if OnClick <> nil then OnClick(Self);
end;
```

Einmal erstellte Ereignisbehandlungsroutinen können einfach durch weitere Objekte benutzt werden. Diese Objekte lassen den entsprechenden Methodenzeiger (z.B. *OnClick*) dazu einfach auf dieselbe Ereignisbehandlungsroutine zeigen.

C++ kennt neben Methodenzeigern auch Zeiger auf Daten in Klassen und Objekten. Die Syntax ähnelt der von Methodenzeigern. Ein Beispiel wurde in der Tabelle der Operatoren angegeben.

Wie bei aggregierten Datentypen bestehen die Werte **abstrakter Datentypen** (abstract data type, **ADT**) meist aus verschiedenen Komponenten mit jeweils eigenem Datentyp. Der Zugriff auf die Komponenten ist über Operationen (Methoden) möglich, die Bestandteil der Datentypdeklaration sind.

C++ gestattet die Deklaration von Structs, die Inline-Funktionen und geschützte Bereiche zulassen. Durch die Schlüsselwörter *private* und *public* können die Zugriffsmöglichkeiten von außen auf die (inneren) Daten und Funktionen kontrolliert gesteuert werden.

```
struct WohntIn
{
  public:
    void reset() {PLZ = 0; Landkennung = ' '; used = FALSE;}
    boolean isUsed() {return used;}
    int PLZ;
    char Landkennung;
  private:
    boolean used;
};

    ...

WohntIn Ort;
Ort.reset();
if (!Ort.isUsed()) Ort.PLZ = 70806;
Ort.used = TRUE; // Fehler, kein Zugriff auf private Member
```

Member-Daten und Funktionen sind standardmäßig (wenn keine Schutzart angegeben wurde) public, also frei zugänglich.

Der wichtigste abstrakte Datentyp in der objektorientierten Programmierung ist die ***Klasse***. In C++ wie in Object Pascal werden Klassen durch das Wort class deklariert. Die C++ Structs mit Member-Funktionen ähneln den Klassen. Klassen-Member in Object Pascal sind standardmäßig public deklariert, in C++ dagegen private. Aufgrund der Bedeutung und Komplexität der Klassen wird dieser Typ in einem eigenen Abschnitt "Klassen und Objekte" besprochen.

Die letzte Gruppe von Typen bilden die **Zeiger-Datentypen**. Werte von Zeiger-Datentypen sind Adressen (von Variablen, Unterprogrammen, usw.). Die Variablen eines Zeiger-Datentyps heißen Zeiger (Pointer). Prinzipiell können Zeiger eingeteilt werden in:

	VC++	Object Pascal	Länge in Win32 (Intel)
untypisierter Zeiger	void *	Pointer	4 Byte
typisierter Zeiger	typ *	^Typ = PTyp	4 Byte

Generell sind typisierte und untypisierte Zeiger insofern identisch, als sie beide auf eine bestimmte Adresse zeigen. Bei typisierten Zeigern weiß der Compiler aber zusätzlich noch, daß die Adresse, auf die der Zeiger zeigt, der Adress-Anfang einer Variablen eines ihm bekannten Typs ist. Somit sind Prüfungen beim Übersetzen möglich. Untypisierte Zeiger können (aufgrund der Typunkenntnis) nicht dereferenziert werden.

Die drei wichtigen Zeiger-Operatoren Dereferenz-, Pfeil- und Adreßoperator werden in der Tabelle der Operatoren aufgeführt. Ein Zeiger kann auch auf "Nichts" zeigen; er enthält keine relevante Adresse. Um einen Zeiger auf "Nichts" zeigen zu lassen, weist man ihm in C++ den Wert *NULL* und in Pascal den Wert *nil* zu. Eine Unterteilung der Zeiger in verschiedene Größen, wie in 16-bit Betriebssystemen üblich (near, far, huge), ist in 32-bit Umgebungen überflüssig geworden. Alle Zeiger in Win32 sind 32 bit groß.

Ein sehr wichtiger, weil viel verwandter Zeigertyp ist der Zeiger auf Zeichenketten. Pascal folgt hier der Vorgehensweise von C++ und behandelt den Typ Array of Char und Zeiger darauf genauso.

VC++	Object Pascal

```
#include <string.h>


char KurzText[100];
char* PKurzText; // Zeiger

strcpy(KurzText, "Hallo Welt");
printf("%s\n", KurzText);
PKurzText = KurzText;
PKurzText = PKurzText + 3;
printf("%s\n", PKurzText);
*PKurzText = 'b';
PKurzText++;
*PKurzText = 'e';
printf("%s\n", KurzText);
KurzText[8] = '\0';
printf("%s\n", PKurzText-1);
```

Ausgabe:

Hallo Welt

lo Welt

Halbe Welt

be We

```
Uses SysUtils;

var
 KurzText: Array[0..99] of Char;
 PKurzText: PChar; // Zeiger

StrCopy(KurzText, 'Hallo Welt');
writeln(KurzText);
PKurzText:= KurzText;
PKurzText:= PKurzText + 3;
writeln(PKurzText);
PKurzText^:= 'b';
inc(PKurzText);
PKurzText^:= 'e';
writeln(KurzText);
KurzText[8]:= #0;
writeln(PKurzText-1);
```

Ausgabe:

Hallo Welt

lo Welt

Halbe Welt

be We

Als Alternativen zu den Char-Arrays und ihrer Zeiger-Arithmetik bieten beide Sprachen Ersatztypen an:

	VC++	Object Pascal	Länge in Win32 (Intel)
String-Klasse	CString	-	4 Byte
Stringzeiger	-	String = AnsiString	4 Byte
String, kurz	-	ShortString[xxx] mit 1 <= xxx <= 255	xxx Byte

Visual C++ und Object Pascal preisen ihren jeweiligen neuen Stringtyp als vollwertigen Ersatz des Char-Array-Typs an, der bei gleicher Flexibilität doch leichter handhabbar sei. Ein Vergleich folgt im Abschnitt "Erweiterte Stringtypen".

Neben den aufgeführten Typen gibt es in beiden Sprachen noch einige weitere vordefinierte Typen, wie Bitfelder in C++ und Dateitypen in Pascal, die hier aber nicht besprochen werden sollen.

Um Programme gut lesbar zu gestalten, ist es sinnvoll, abgeleitete Typen zu benennen. C++ bietet mit dem Schlüsselwort *typedef*, Pascal mit dem Schlüsselwort *type* diese Möglichkeit. Schreibt man ein Programm, in dem häufig 8x8 Integer-Matrizen vorkommen, kann man in

VC++	Object Pascal
statt immer wieder `int matrix_a[8][8];` zu schreiben, einmalig einen Typ definieren: `typedef int Matrix[8][8];` und diesen dann stets bei der Definition von Variablen verwenden: `Matrix a, b;`	statt immer wieder `var matrix_a:` `  Array[0..7,0..7] of Integer;` zu schreiben, einmalig einen Typ definieren: `type Matrix =` `  Array[0..7,0..7] of Integer;` und diesen dann stets bei der Definition von Variablen verwenden: `var a, b: Matrix;`

Auch um Arrays fest vorgegebener Länge in Object Pascal als Funktionsparameter übergeben zu können, ist eine vorherige, benutzerdefinierte Typ-Definition nötig. Statt

```
MeineProzedur(x: Array[0..7,0..7] of Integer);
begin
  ...
end;
```

muß geschrieben werden:

```
MeineProzedur(x: Matrix);
begin
  ...
end;.
```

Das Schlüsselwort type hat in Pascal eine größere Bedeutung als typedef in C++, weil Pascal-Typen keine "Tag"-Namen kennen. Auch Klassen-Definitionen (zur Festlegung der Klassen-Schnittstellen) müssen deswegen z.B. in einem type-Abschnitt aufgeführt werden:

VC++	Object Pascal
`class Lanfahrzeug` `{` `  ...` `};`	`type` `  Landfahrzeug = class` `  ...` `  end;`

2.3.3. Variablen und Konstanten

Variablen stellen Instanzen von Typen dar. Der Typ einer Variablen definiert die Menge der Werte, die sie annehmen kann, sowie die Operationen, die mit ihr durchgeführt werden dürfen. Eine Variablen-Definition schreibt neben der Typangabe, die Angabe eines Bezeichners, einer Speicherkategorie und eines lexikalischen Gültigkeitsbereichs vor. Durch ihre Vereinbarung im Programm erfolgt implizit eine statische Speicherreservierung durch den Compiler. Die gleichzeitige Initialisierung einer Variablen bei ihrer Definition ist in C++ möglich und in Pascal nicht möglich. Die Verwendung nicht initialisierter Variablen stellt einen schweren Fehler dar, der oft ein sporadisches Fehlverhalten des betreffenden Programms verursacht. Die Compiler beider Entwicklungssysteme erkennen Lesezugriffe auf nicht initialisierte Variablen und geben beim Übersetzen Warnungen aus: Visual C++ meldet

"variable ... used without having been initialized" und Delphi teil dem Entwickler mit, daß "die Variable ... wahrscheinlich nicht initialisiert wurde".

Mögliche Geltungsbereiche von Variablen sind globale und lokale (Block-) Gültigkeit. Blöcke bestehen aus Deklarationen und Anweisungen. Der globale Gültigkeitsbereich wird in C++ durch den sogenannten externen Gültigkeitsbereich (external scope) gebildet, der alle Funktionen in einem Programm umfaßt. In Pascal wird der globale Gültigkeitsbereich durch den globalen Block gebildet.

Lokale Blöcke werden z.B. durch Prozedur-, Funktions- und Methodendeklaration gebildet. Variablen, die innerhalb solcher Blöcke definiert werden, heißen lokale Variablen und werden im Stack einer Anwendung abgelegt. Der Stack wird mitunter auch Stapel genannt. Er bezeichnet einen Speicherbereich, der für die Speicherung lokaler Variablen reserviert ist. Der Gesamtbedarf an Speicherplatz im Stack ergibt sich aus der Summe des Einzelbedarfs sämtlicher Funktionen und Prozeduren, die zu einem gegebenen Zeitpunkt gleichzeitig aktiv sein können. Der Bedarf an Stack-Speicher ist bei rekursiven Funktionsaufrufen besonders groß; die Gefahr eines Stack-Überlaufs besteht, wenn der Stack nicht genügend groß dimensioniert wurde.

Der Gültigkeitsbereich einer Variablen wird dadurch bestimmt, in welchem Block sie definiert wurde. Variablen sind im aktuellen Block und in allen, diesem Block untergeordneten Blöcken gültig, solange im untergeordneten Block nicht eine Variable mit gleichem Namen definiert wurde. In dem Fall wird sie von der neuen, gleichnamigen Variablen "verdrängt" und ist bis zum Blockende nicht mehr sichtbar.

Gemeinsam ist beiden Sprachen, daß Variablen definiert werden müssen, bevor sie benutzt werden können (*declaration-before-use*). Von dieser Einschränkung abgesehen, gestattet C++ die Variablen-Definition an beliebiger Stelle im Quelltext (*on-the-spot variables*), während Pascal zwingend vorschreibt, daß Variable für jeden Gültigkeitsbereich in einem gesonderten Abschnitt, der mit dem Schlüsselwort var eingeleitet wird, definiert werden müssen. Der Entwickler wird dadurch zu einer gewissen Disziplin gezwungen. Dagegen können die Variablen in C++ beliebig verstreut im Text auftauchen. Andererseits kann es bei größeren Blöcken vorteilhaft sein, wenn die Variablen erst dort definiert werden, wo sie tatsächlich gebraucht werden. Ihr Zweck ist dadurch in C++ sofort erkennbar und muß in Pascal durch geschickte, selbsterklärende Namenswahl verdeutlicht werden. Die Gültigkeit einer Variablen endet in beiden Sprachen mit dem Blockende, in dem sie definiert wurde.

VC++	Object Pascal
`int x;  // globale Variable` `void Test()` `{` `  int i;  // lokale Variable` `  i = 3;` `  for (int m=1; m <= x; m++)...` `  // lokale Variable m wird` `  // erst innerhalb der` `  // for-Schleife definiert` `}` `void main(){` `  x = 8;`	`var x: Integer;  // globale Var.` `Procedure Test;` `var` `  i, m: Integer;  // lokale Var.` `begin` `  i:= 3;` `  for m:= 1 to x do ...` `end;` `begin`

```
  i = 7;
  // Fehler, da i nur lokale
  // Blockgültigkeit innerhalb
  // der Funktion besitzt
}
```

```
  x:= 8;
  i:= 7;
  // Fehler, da i nur lokale
  // Blockgültigkeit innerhalb
  // der Prozedur besitzt
end.
```

Variable können folgenden Speicherkategorien (auch Speicherklassen genannt) angehören:

	VC++	Object Pascal
auto	Ja	Ja
extern	Ja	Nein
register explizit	Ja	Nein
register automatisch	Ja	Ja
static	Ja	Nein

Auch in Object Pascal kann "register" explizit verwendet werden, aber nur bei Funktionsdeklarationen. Die ersten drei Parameter solcherart deklarierter Funktionen übergeben ihre Werte von der Aufrufstelle zur aufgerufenen Funktion nicht über den Stack-Speicher, sondern über die CPU-Register EAX, EDX und ECX (in dieser Reihenfolge).

Alle Variablen in Pascal und alle Variablen in C++, bei denen keine explizite Angabe der Speicherkategorie erfolgte, zählen bei deaktivierter Code-Optimierung zur Kategorie auto. Wenn der Gültigkeitsbereich einer Variablen verlassen wird, wird der für sie reservierte Speicher *auto*matisch frei gegeben. Beim Verlassen einer Funktion werden deswegen auch solche lokalen Variablen automatisch wieder gelöscht.

Variable, die mit extern definiert sind, weisen den Compiler an, die Variablen irgendwo anders zu suchen, in der gleichen oder in einer anderen Datei. Solche Variablen werden niemals ungültig.

Mit "register" definierte Variablen weisen den Compiler an, diese, sofern möglich, in schnellen Prozessor-Registern zu halten. Solche Variablen können z.B. als schnelle Schleifen-Zähler sinnvoll verwendet werden. Bei eingeschalteter Optimierung werden explizite register-Anweisungen durch Visual C++ ignoriert. Der Compiler erzeugt dann, ebenso wie Delphi, selbständig Register-optimierten Code für oft benutzte Variable und Ausdrücke. D.h. Variable sind oftmals Register-Variable, ohne daß man es explizit angegeben hat. Man wird darüber auch nicht informiert, kann es höchstens durch eine Analyse des erzeugten Assembler-Codes erkennen. Delphi geht hier sogar soweit, eine Lebenszeit-Analyse ("lifetime analysis”) für Variable vorzunehmen. Wenn eine bestimmte Variable I exklusiv in einem Codeabschnitt und eine andere Variable J in einem späteren Abschnitt exklusiv verwendet wird, verwendet der Compiler *ein einziges* Register für I und J.

Eine lokal definierte Variable, die mit static gekennzeichnet ist, behält ihren Wert über das Blockende hinaus. Wenn der Block erneut betreten wird, ist der alte Wert noch immer verfügbar. Extern- und Static- Variable, die nicht explizit initialisiert wurden, werden durch das System mit 0 initialisiert. Das trifft auch auf Variable strukturierter Typen zu: alle Elemente solcher Variablen werden zu 0 initialisiert. Variable der Kategorien auto und register werden nicht automatisch initialisiert.

Konstanten bekommen bei ihrer Definition einen unabänderlichen, festen Wert zugewiesen. Die Konstante ist schreibgeschützt, ist eine Nur-Lese-Variable. Es gibt typisierte und untypisierte Konstanten.

Bei der Definition typisierter Konstanten werden oft elementare Datentypen benutzt:

VC++	Object Pascal
`const int Minimum = 144;` `  ...` `Minimum = 25;` `// Fehler, Zuweisung` `// nicht erlaubt`	`const Minimum: Integer = 144;` `  ...` `Minimum:= 25;` `// Fehler, Zuweisung` `// nicht erlaubt`

Auch aggregierte und Zeiger - Datentypen können als typisierte Konstanten definiert werden:

VC++	Object Pascal
`const int FestWerte[3] =` `  {1, 2, 24};`	`const FestWerte:` `  Array[0..2] of Integer =` `  (1, 2, 24);`

Eine Besonderheit in C++ stellt die Tatsache dar, daß mit const definierte Ausdrücke nur innerhalb der Datei Gültigkeit besitzen. Sie können nicht direkt in einer anderen Datei importiert werden. Wenn eine Konstante aber trotzdem auch in einer anderen Datei gültig sein soll, so muß explizit eine Definition mit dem Schlüsselwort extern vorgenommen werden.

Neben den typisierten Konstanten existieren in beiden Sprachen untypisierte Konstanten. Sie können mitten im Quelltext als feststehende Zeichen und Zeichenketten auftreten oder als konstanter Ausdruck explizit definiert worden sein. Sie werden in C++, anders als in Pascal, nicht durch das Schlüsselwort const, sondern durch einen Ausdruck der Art #define Konstante (ohne abschließendes Semikolon) angegeben und durch den Präprozessor vor dem eigentlichen Übersetzungslauf an den entsprechenden Stellen im Quelltext ersetzt.

	C++	Pascal
Numerische Konstante im Text	`double Betrag;` `  ...` `Betrag = 35.20;` `// hexadezimal` `Betrag = 0x3F20;`	`var Betrag: Double;` `  ...` `Betrag:= 35.20;` `// hexadezimal` `Betrag:= $3F20;`
String-Konstante im Text	`printf("Das ist mehr"\` `  "zeiliger Text\n");`	`writeln('Das ist mehr' +` `  'zeiliger Text'#13);`
durch explizite Definition	`#define Laenge = 13` `  ...` `printf("%d", Laenge);`	`const Laenge = 13;` `  ...` `writeln(Laenge);`

String-Konstanten werden auch String-Literale genannt. Im Text auftretende ASCII-Zeichen wie '\0' oder #0 nennt man Zeichen-Literale. Beachtet werden muß, daß const-Ausdrücke Blockgültigkeit besitzen, während der Gültigkeitsbereich der #define-Anweisung erst durch eine zugehörige #undefine-Anweisung aufgehoben wird.

Konstanten werden von den Compilern im Code-Abschnitt des zu erzeugenden Programms eingefügt. Da konstante Ausdrücke schreibgeschützt sind, kann sich der Codeinhalt zur Laufzeit nicht ändern. Sich selbst modifizierender Code (veränderte Konstanten-Werte) würde die Optimierungsstrategien moderner Prozessoren, wie die des Pentium Pro von Intel, unterlaufen. Die hier angewandten Verfahren der überlappten, parallelen und vorausschauenden Befehlsausführung können nur dann beschleunigend wirken, wenn einmal geladener Code nicht verändert wird. Delphi gestattet jedoch aus Gründen der Abwärtskompatibilität, mit der Compiler-Anweisung {$WRITEABLECONST ON} Schreibzugriffe auf typisierte Konstanten

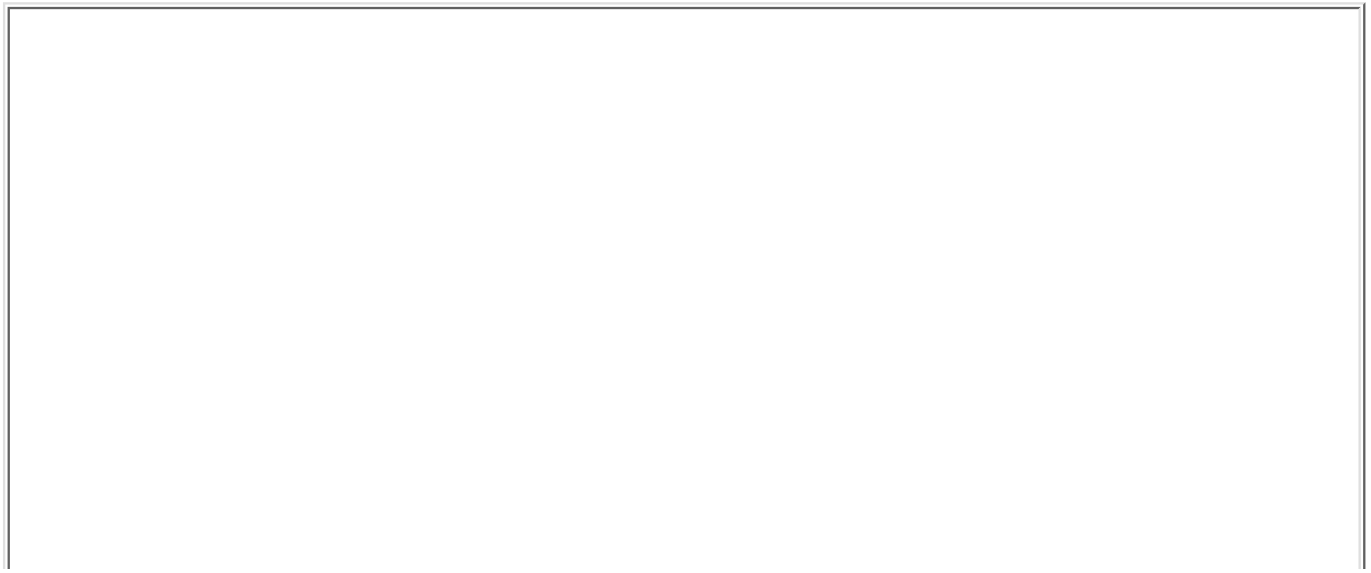
zuzulassen.

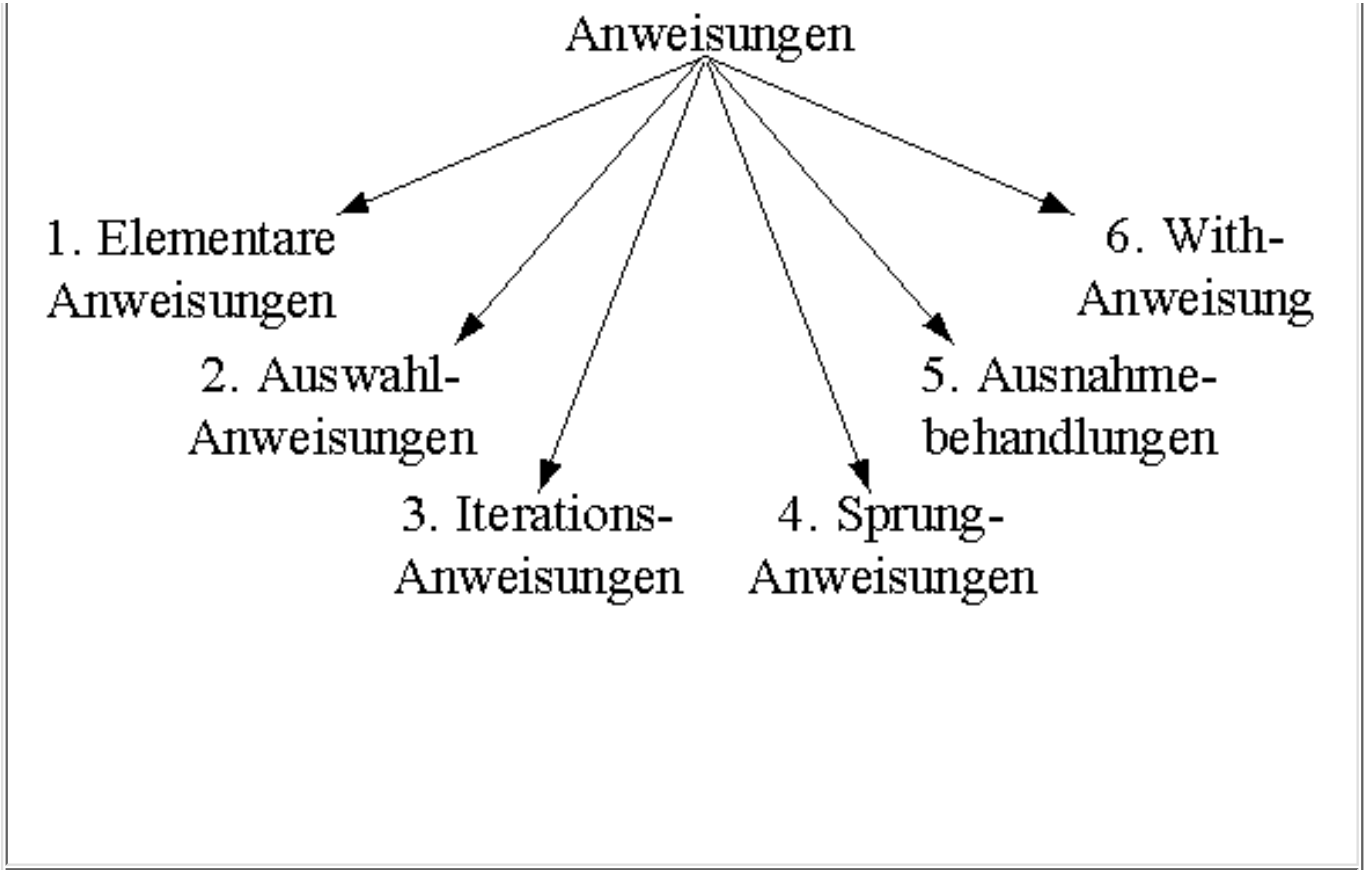
Das Wort const kann außerdem in C++ und Object Pascal vor Funktionsparametern auftreten. Bei so deklarierten Parametern werden vom Compiler Schreibzugriffe auf diese unterbunden:

VC++	Object Pascal
`void MyProc(char const x)` `{` `  x = 'm';  // Fehler` `}`	`procedure MyProc(const x: Char);` `begin` `  x:= 'm'; // Fehler` `end;`

2.3.4 Anweisungen

Anweisungen (statements) beschreiben algorithmische Vorgänge, die vom Computer ausgeführt werden können. Sie steuern oft den Kontrollfluß von Programmen. Jeder Anweisung kann in Visual C++ und Object Pascal ein Label vorangestellt werden, auf das eine Bezugnahme mit der Sprunganweisung "goto" möglich ist.





Mögliche Einteilung der Anweisungen

	Anweisung	VC++	Object Pascal
1	Leere Anweisung	`;`	`;`
1	Zuweisung	`i = m;`	`i := m;`
1	Verbund-Anweisung	`{` `  ...` `}`	`begin` `  ...` `end`

2	If - Else-Anweisung	`if (i == m)` `  printf("gleich");` `else` `  printf("ungleich");`	`if i = m then` `  writeln('gleich')` `else` `  writeln('ungleich');`
2	Case- Anweisung	`char Zeichen = 'x';` `switch (Zeichen)` `{` `  case 'ä': printf("ae");` `  break;` `  case 'ö': printf("oe");` `  break;` `  case 'ü': printf("ue");` `  break;` `  case 'ß': printf("ss");` `  break;` `  default:` `  printf("%c",Zeichen);` `};`	`var Zeichen: Char;` `Zeichen:= 'x';` `Case Zeichen Of` `  'ä': write('ae');` `  'ö': write('oe');` `  'ü': write('ue');` `  'ß': write('ss');` `Else` `  write(Zeichen);` `End;`
3	While- Anweisung	`while (x != y) ...` `while (TRUE) {};` `//Endlosschleife!`	`while x <> y do ...` `while True do;` `//Endlosschleife!`
3	For- Anweisung	`for (i = 0; i < n; i++)` `  a[i] = b[i+2];`	`for i:= 0 to n-1 do` `  a[i]:= b[i+2];`
3	Repeat- Anweisung	`do` `  x = x + 1;` `while (x <= 100);` `// x ist jetzt 101`	`repeat` `  x:= x + 1;` `until x > 100;` `// x ist jetzt 101`

	Anmerkung:	Die Repeat-Anweisung garantiert die mindestens einmalige Ausführung der Anweisung in der Schleife.	
4	Goto- Anweisung	`void main()` `{` `  Label1: printf("L 1");` `  ...` `  goto Label1;` `}`	`Label Label1;` `begin` `  Label1: write('L 1');` `  ...` `  goto Label1;` `end.`
4	Break- Anweisung	`for (i=0; i < n; i++)` `{` `  if (a[i] < 0)` `  {` `  printf("Fehler");` `  break;` `  }` `  ...` `}`	`for i:= 1 to n do` `begin` `  if a[i] < 0 then` `  begin` `  writeln("Fehler");` `  Break;` `  end;` `  ...` `end;`
	Anmerkung:	Die Break-Anweisung bewirkt das sofortige Beenden der While-, For- und Repeat- Schleifen. In C++ wird sie auch zum Sprung aus der Case-Anweisung benötigt.	
4	Continue-Anweisung	`for (i=0; i < n; i++)` `{` `  if (a[i] < 0)` `  {` `  continue;` `  // neagtive Elemente` `  // werden ignoriert` `  }` `  ...` `}`	`for i:= 1 to n do` `begin` `  if a[i] < 0 then` `  begin` `  continue;` `  // neagtive Elemente` `  // werden ignoriert` `  end;` `  ...` `end;`

	Anmerkung:	Die Continue-Anweisung bewirkt den sofortigen Beginn der nächsten Schleifeniteration (d.h. bewirkt den Sprung an das Ende der Schleife).	
4	Return- Anweisung	**// bei Prozeduren** void Hallo(int a) { printf("Hallo "); if (a == 0) **return**; printf("Welt\n"); } **// bei Funktionen** int Kubik(int m) { **return** m * m * m; }	**// bei Prozeduren** procedure Hallo(a: Integer); begin write("Hallo "); if a = 0 then **exit**; writeln("Welt"); end; **// bei Funktionen** function Kubik (m: Integer): Integer; begin **Kubik :=** m * m * m; // exit-Aufruf ist // hier überflüssig end; // statt // Kubik:= m*m*m; // wäre auch möglich // Result:= m*m*m;
5	Ausnahme-Behandlung (exception handling)	**try** **{** <zu schützender Block>; **}** **catch(<Typ1>)** **{** BehandlungsBlock1; **}** **catch(<Typ2>)** **{** BehandlungsBlock2; **}**	**Uses SysUtils;** **try** <zu schützender Block>; **except** **on** <class1> **do** BehandlungsBlock1; **on** <class2> **do** BehandlungsBlock2; **end;**

5	Ausnahme-auslöse-Anweisung (raise exception)	`void IsInside(int Wert,` `  int Min,` `  int Max)` `{` `  if ((Wert < Min) ||` `  (Wert > Max))` `  throw "Wert außerhalb";` `}` `...` `int i;` `i = 15;` `try` `{` `  IsInside(i, 1, 10);` `}` `catch(char* str)` `{` `  printf("Exception %s` `  aufgetreten", str);` `}`	`procedure IsInside(Wert,` `  Min,` `  Max: Integer);` `begin` `  if (Wert < Min) or` `  (Wert > Max) then` `  raise ERangeError.` `  Create('Wert` `  außerhalb');` `end;` `...` `var i: Integer;` `i:= 15;` `try` `  IsInside(i, 1, 10);` `except` `  on E: ERangeError do` `  write('Exception '+` `  E.Message +` `  'aufgetreten');` `end;`
5	Ausnahme- Ende-Behandlung (termination handling)	`__try` `{` `  <zu schützender Block>;` `}` `__finally` `{` `  <Termination-Block>;` `}`	`try` `  <zu schützender Block>;` `finally` `  <Termination-Block>;` `end;`

<table>
<tr><td></td><td></td><td>

```
char* str;

// Ressource belegen
str = new char[50];
__try
{
  // Laufzeitfehler
  // provozieren!
  strcpy(str, NULL);
}
__finally
{
  printf("Termination");
  // Ressource sicher
  // freigeben
  delete [] str;
}
printf("Good Bye!\n");
```

</td><td>

```
var str: PChar;

// Ressource belegen
GetMem(str, 50);
try
  // Laufzeitfehler
  // provozieren!
  StrCopy(str, nil);
finally
  writeln('Termination');
  // Ressource sicher
  // freigeben
  FreeMem(str, 50);
end;
writeln('Good Bye!');
```

</td></tr>
<tr><td></td><td>Anmerkung:</td><td colspan="2">In beiden Programmstücken wird niemals die Ausgabe "Good Bye!" erscheinen. Nur der finally-Block kann trotz Laufzeitfehler noch sicher erreicht werden. Die Freigabe der belegten Ressourcen (hier des Speichers) kann so auf alle Fälle ordnungsgemäß ausgeführt werden.</td></tr>
<tr><td>6</td><td>With- Anweisung</td><td>Anweisung nicht definiert</td><td>

```
TDatum = record
  Tag: Integer;
  Monat: Integer;
  Jahr: Integer;
end;
var BestellDatum: TDatum;

with BestellDatum do
  if Monat = 12 then
  begin
    Monat:= 1;
    Jahr:= Jahr + 1;
  end else
    Monat:= Monat + 1;
```

</td></tr>
</table>

	Anmerkung:		Die With-Anweisung vereinfacht den Zugriff auf Felder von Records und Objekten.

Anmerkungen zur For - Schleife:

Die For Schleife hat in C++ folgende Form:

```
for (expression1; expression2; expression3)
  statement
next statement
```

Sie ist vollkommen äquivalent zur While Anweisung:

```
expression1;
while (expression2)
{
  statement;
  expression3;
}
next statement ,
```

wenn kein continue Ausdruck innerhalb der Schleife vorkommt. Zunächst wird expression1 ausgewertet, wobei typischerweise eine Laufvariable initialisiert wird. Dann wird expression2 ausgewertet. Wenn das Ergebnis TRUE ist, wird statement ausgeführt, expression3 ausgewertet und wieder an den Anfang der For-Schleife gesprungen. Allerdings wird nun die Auswertung von expression1 übersprungen. Diese Iteration wird so lange fortgesetzt, bis expression2 FALSE liefert. Es wird mit der Ausführung von next statement fortgesetzt.

In Object Pascal besitzt die For Schleife folgendes Aussehen:

```
for Laufvariable:= Anfangswert <to> | <downto> Endwert do
  statement;
next statement
```

Anders als in C++ entspricht die For Schleife in Object Pascal nicht der While Schleife, wenngleich ihre Wirkung ähnlich ist. In Object Pascal wird, je nachdem, ob das Schlüsselwort **to** oder **downto** benutzt wird, die Laufvariable jeweils um eins erhöht oder vermindert, währenddessen man in C++ im Ausdruck expression3 frei festlegen kann, was bei jedem Schleifendurchgang passieren soll. Die Ende- oder Abbruchbedingung der Schleife wird, anders als in C++, nur einmal ausgewertet. Folgendes ist zwar syntaktisch nicht falsch, aber unsinnig:

```
j:= 100;
for i:= 0 to j do dec(j);
```

Eine weitere Besonderheit bezüglich der Object Pascal Implementierung der For Schleife besteht darin, daß die verwendete Laufvariable eine lokal definierte Variable sein muß. Die Laufvariable darf innerhalb der Schleife nicht verändert werden.

```
var
  i: Integer;
begin
  for i:= 1 to 20 do inc(i);
end;
```

Delphi weist diesen Code bereits beim Compilieren ab und meldet, daß der Laufvariablen innerhalb der Schleife kein Wert zugewiesen werden darf. Allerdings ist es auch in C++ kein besonders guter Progammierstil, die Abbruchbedingung durch Setzen der Laufvariablen innerhalb der Schleife zu erwirken und ergibt undurchschaubaren Code.

Nach dem Ende der For Schleife ist der Wert der Laufvariablen in C++ definiert und in Pascal undefiniert. Das oft benutzte Beispiel für StringCopy darf in Pascal so nicht umgesetzt werden:

```
for (i = 0; i < 4; s[i++] = 'x');
s[i] = \0;
```

Die Variable i hat in C++ in der letzten Zeile den Wert 5. Falls i in Pascal nach der For Schleife ebenfalls den Wert 5 haben sollte, so ist das reiner Zufall. In der Praxis ist es aber so, daß die Laufvariable nach der Schleife tatsächlich meist den richtigen Wert besitzt und deswegen leicht unvorhergesehene Fehler im Programm auftreten können. Folgendes Programmstück

```
var
  i, j: Integer;
begin
  for i:= 1 to 4 do
    j:= i;
  writeln('Variable i = ', i);
  writeln('Variable j = ', j);
end;
```

liefert folgende Ausgabe:

Variable i = 5

Variable j = 4

Delphi erzeugt dabei (bei eingeschalteter Optimierung) für die Schleife folgenden Assembler- Code:

```
:D5D    mov    ebx,00000001   // for i:= 1 to 4 do
:D62    mov    esi,ebx        // j:= i;
:D64    inc    ebx            // entspricht i:= i + 1
:D65    cmp    ebx,00000005   // prüfen ist i = 5 ?
:D68    jne    (D62)          // wenn nicht gleich, dann
                              // springe zu Adresse (:D62)
:D6A    mov    edx,DDC        // writeln('Variable i = ', i);
:D6F    mov    eax,1FC
:D74    call    @Write0LString
:D79    mov    edx,ebx
:D7B    call    @Write0Long
:D80    call    @WriteLn
:D8A    mov    edx,DF4        // writeln('Variable j = ', j);
:D8F    mov    eax,1FC
:D74    call    @Write0LString
:D79    mov    edx,ebx
:D7B    call    @Write0Long
:D80    call    @WriteLn
```

Man erkennt hier, daß Delphi für die Laufvariable i zur Optimierung des Laufzeitverhaltens Register verwendet (hier Register ebx). Falls nach der Schleife zwischenzeitlich zufällig Register ebx überschrieben worden wäre, dann würde writeln(i) einen "falschen" Wert ausgeben. Genau das Verhalten kann man beobachten, wenn man folgenden Code ausführt:

```
var
  i, j, k, l: Integer;
begin
  for i:= 1 to 4 do j:= i;
  for k:= 8 to 10 do j:= k;
  for l:= 12 to 16 do j:= l;
  writeln('Variable i = ', i);
  writeln('Variable k = ', k);
  writeln('Variable l = ', k);
  writeln('Variable j = ', j);
  readln;
end;
```

Ausgabe:

Variable i = 5

Variable k = 11

Variable l = 11

Variable j = 16

Anders als man erwarten könnte ist Variable l nun nicht 17 sondern 11.

Anmerkungen zu Ausnahmebehandlungen (Exceptions):

Exceptions sind ein Hilfsmittel zur Behandlung von Ausnahmesituationen in Programmen, insbesondere zur Fehlerbehandlung. Wenn eine Exception ausgelöst wird, verzweigt das Programm sofort zu einem Exception-Handler, der auf den Fehler reagieren kann. Durch die Benutzung von Exceptions wird der normale Kontrollfluß des Programms vom Kontrollfluß für Fehlerfälle getrennt, wodurch das Programm besser strukturiert wird und leichter zu warten ist. Der Einsatz von Exceptions garantiert, daß Fehler im ganzen Programm in einer konsistenten Weise behandelt und angezeigt werden. Belegte Systemressourcen können mit Sicherheit frei gegeben werden und Laufzeitfehler müssen ein Programm nun nicht mehr zwangsweise beenden.

Die Exceptions in Visual C++ und Object Pascal haben viele Gemeinsamkeiten; insbesondere sind sie in beiden Sprachen Typ-basierend. Beim Erzeugen einer Exception wird der Typ der Exception festgelegt, indem beim throw bzw. raise Aufruf ein Argument übergeben wird. Die Exception ist dann vom selben Typ wie der Typ dieses Arguments. Exceptions können in C++ von beliebigem Typ sein. In Object Pascal dagegen sind Exceptions nur vom Typ Objekt (bevorzugt abgeleitet von der Klasse *Exception*) zulässig.

Mehrere Exception-Handler können, einem try-Block nachfolgend, angelegt werden. Sie werden durch das Wort catch in C++ und except in Object Pascal eingeleitet. Der zuständige Exception-Handler, der für die Auswertung der aufgetretenen Exceptions verantwortlich ist, wird dadurch ausgewählt, indem geprüft wird, von welchem Typ die Exception ist und welcher Handler diese Exception verarbeiten kann.

Eine einmal erzeugte Exception bleibt so lange bestehen, bis sie behandelt oder die Anwendung beendet wird. Deshalb müssen auftretende Exceptions in irgendeiner Weise behandelt werden. Visual C++ und Object Pascal bieten vordefinierte Standardbehandlungsfunktionen (Default-Handler) für Exceptions an. Deswegen braucht man nicht alle möglicherweise auftretenden Exceptions in jedem try...catch / try...except Block zu behandeln.

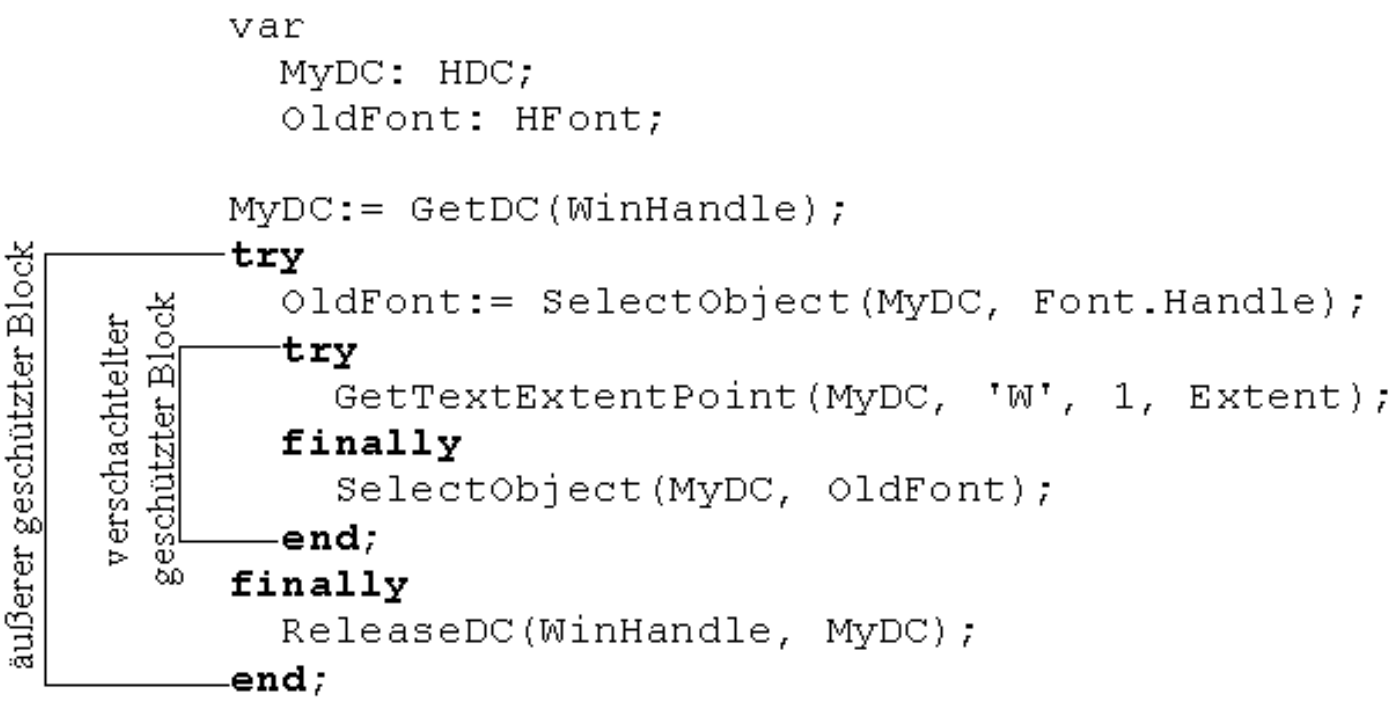
Die Klassenbibliothek von Visual C++ (MFC) bietet, ebenso wie die Klassenbibliothek von Delphi (VCL), eine Vielzahl bereits vordefinierter Exception-Klassen an. Ein entscheidender Unterschied zwischen beiden Sprachen ist die Tatsache, daß eine Exception-Klasse in Visual C++ innerhalb des Exception-Handlers gelöscht werden muß und in Object Pascal nicht gelöscht werden darf.

```
catch(CException* e)
{
  // Behandlung der Exception hier.
  // "e" beinhaltet Informationen über die Exception.
  // Zuletzt ist das Löschen der Exception nötig!
  e->Delete();
}
```

Dagegen führt in Object Pascal die Behandlung einer Exception dazu, daß das Exception-Objekt automatisch entfernt wird.

Visual C++ unterstützt zwei verschiedene Exception-Modelle: "C++ Exception Handling" und "Strukturiertes Exception Handling". Microsoft empfiehlt das erste, ANSI-normierte Modell zu nutzen. Leider bietet dieses Exception-Handling keine Unterstützung für try/ finally Ressourcen-Schutzblöcke. Wenn man diese Schutzblöcke trotzdem verwenden möchte, ist man deswegen gezwungen, das Modell des "Strukturierten Exception Handlings" zu verwenden. Allerdings ergibt sich dabei das Problem, daß nicht beide Exception-Modelle in einer Funktion gleichzeitig verwendet werden dürfen.

Die Verwendung von try / finally Schutzblöcken ist bei der Windows-Programmierung besonders sinnvoll, weil hier verwendete Ressourcen-Objekte (wie Stifte, Pinsel und Zeichenflächen usw.) immer explizit vom Programm freigegeben werden müssen. Eine Verschachtelung der try / finally Anweisungen ist hier notwendig. Die Freigabe belegter Ressourcen erfolgt meist umgekehrt zur Belegungsreihenfolge.




```
var
  MyDC: HDC;
  OldFont: HFont;

MyDC:= GetDC(WinHandle);
try
  OldFont:= SelectObject(MyDC, Font.Handle);
  try
    GetTextExtentPoint(MyDC, 'W', 1, Extent);
  finally
    SelectObject(MyDC, OldFont);
  end;
finally
  ReleaseDC(WinHandle, MyDC);
end;
```

Auch das beliebige Verschachteln von try / except - Blöcken und try / finally - Blöcken ist möglich.

Damit Standard-Exceptions in Object Pascal funktionieren muß die Unit SysUtils in's Programm eingebunden werden. Die Basisklasse aller Exceptions mit dem Namen *Exception* ist in dieser Unit definiert.

2.3.5 Programmstruktur

Programme sind eine Sammlung von Funktionen, Prozeduren und Deklarationen. C++ und Pascal sind block-strukturiert. Funktionen und Prozeduren können in Pascal beliebig tief verschachtelt sein, d.h. jede Funktion kann weitere Unterfunktionen definieren. C++ erlaubt für Funktionen nur eine Schachtelungstiefe von 1, d.h. Funktionen können keine weiteren Unterfunktionen definieren. Beide Sprachen erlauben rekursive Aufrufe.

	VC++	Object Pascal

function (call by value)	`double Quadrat(int x)` `{` `  return x * x;` `}`	`function Quadrat(x: Integer):` `  Double;` `begin` `  Result:= x * x;` `end;`
procedure (call by value)	`void Quadrat(int x)` `{` `  printf("%e", x * x);` `}`	`procedure Quadrat(x: Integer);` `begin` `  writeln(x * x);` `end;`
leere Argumentenliste	`void Test()` `{` `  printf("Hallo");` `}` `Aufruf: Test();` `// mit Klammer`	`procedure Test;` `begin` `  writeln("Hallo");` `end;` `Aufruf:  Test;` `// ohne Klammer`
Variablen Argument (call by reference)	`void Tausch` `  (int& i, int& j);` `{` `  int t = i;` `  i = j;` `  j = t;` `}`	`procedure Tausch` `  (var i, j: Integer);` `var t: Integer;` `begin` `  t:= i;` `  i:= j;` `  j:= t;` `end;`

Standard- Argumente	`int Potenz(int a,` `  int n = 2)` `{` `  int Result = a;` `  int i;` `  for (i=1; i<n; i++)` `  Result = Result*a;` `  return Result;` `};` `mögliche Aufrufe:` `  Potenz(3);` `  Potenz(5,2);` `  Potenz(3,4);`	Object Pascal unterstützt keine Standard-Funktions-Argumente

Die Funktion *Potenz* zur Lösung der Aufgabe a^n besitzt das Standard-Argument n = 2. Die Funktion kann deswegen wahlweise auch mit nur einem Parameter aufgerufen werden. Als Ergebnis liefert *Potenz* in diesem Fall das Quadrat der Zahl *a*. Standard-Argumente müssen keine Konstanten sein; auch globale Variablen oder Funktionsausdrücke sind zulässig. Die Standard-Argumente müssen immer am Ende der Parameterliste stehen. Durch die Verwendung von Standard-Argumenten lassen sich keine Effizienz-Vorteile erzielen, da bei einem Funktionsaufruf immer alle Parameter übergeben werden (gegebenenfalls der Standardwert). Überdies bergen Standard-Argumente Risiken, da versehentlich nicht übergebene Argumente durch den Compiler nicht erkannt werden können.

C++ gestattet das **Überladen** von Funktionen (*function overloading*). Hinweis: Die Dokumentationen zu Visual C++ und Object Pascal benutzen den Begriff "Überladen" mit ganz verschiedener Bedeutung: Visual C++: Funktion mit veränderter Parameterliste deklarieren oder für einen Operator eine neue Bedeutung definieren. Obj. Pascal: als Synonym für das Überschreiben von Klassen-Methoden

Beim Überladen werden einer Funktion mehrere Bedeutungen zugeordnet. Mehrere Funktionen werden dazu mit gleichem Namen, aber unterschiedlichen Parametertypen und / oder Parameteranzahl definiert. Beim Aufruf einer so überladenen Funktion wird beim Übersetzen vom Compilier durch Typvergleich des aktuellen Parameters mit dem formalen Parameter die entsprechende (zutreffende) Funktion ausgewählt. Sind alle Typen der formalen Parameter ungleich dem Typ des aktuellen Parameters, so wird versucht, durch Typkonvertierung die zugehörige Funktion zu bestimmen. Zu überladende Funktionen müssen alle im selben Gültigkeitsbereich deklariert sein. Weiterhin muß der Ergebnistyp bei allen überladenen Funktionen gleich sein.

Object Pascal gestattet das Überladen von Funktionen nicht. Die einzige Möglichkeit, das Verhalten von C++ nachzuahmen besteht darin, Funktionen mit sehr ähnlichem Namen zu deklarieren. So könnten sich z.B. die Namen mehrerer Funktionen nur im letzten Buchstaben unterscheiden, der damit angibt, mit welchem Parameter-Typ die Funktion aufzurufen ist.

VC++	Object Pascal
`void Drucke(int i)` `{` `  printf("%d", i);` `}` `void Drucke(char* s)` `{` `  printf("%s\n", s);` `}` `void Drucke(double k, int i)` `{` `  printf("%e %d", k, i);` `}` `...` `Drucke(16);` `Drucke(23.8, 12);` `Drucke("Hallo Welt");`	`procedure DruckeI(i: Integer);` `begin` `  writeln(i);` `end;` `procedure DruckeS(s: String);` `begin` `  writeln(s);` `end;` `procedure DruckeDI(k: Double;` `  i: Integer);` `begin` `  writeln(k,' ',i);` `end;` `...` `DruckeI(16);` `DruckeDI(23.8, 12);` `DruckeS('Hallo Welt');`

Das Überladen von Funktionen wird mitunter auch als "ad-hoc-Polymorphismus" bezeichnet. Während man bei dem auf Klassen basierenden (echten) Polymorphismus (*pure polymorphism*) eine Klasse verändert, um ein anderes Verhalten zu bewirken, so ändert man bei dem auf Funktionen basierenden ad-hoc-Polymorphismus einen oder mehrere Funktions-Parameter, um ein anderes Verhalten zu erzielen.

Wenn in C++ bei überladenen Funktionen neue Argumente als Standard-Argumente deklariert werden, kann ein späterer Funktionsaufruf unter Umständen nicht eindeutig einer konkreten Funktion zugeordnet werden.

```
void Drucke(int i)
{
  printf("%d", i);
}

// überladene Funktion mit Standard-Argument
void Drucke(int i, int j=5)
```

```
{
  printf("%d %d", i, j);
}

void main()
{
  Drucke(3);
  // Fehler; welche Drucke-Funktion ist gemeint?
}
```

Der Compiler bricht die Übersetzung mit einer Fehlermeldung der Art "ambiguous call to overloaded function" ab. Der Funktionsaufruf ist mehrdeutig.

Aufrufkonventionen für Funktionen und Prozeduren:

Reihenfolge der Argumente	VC++	Object Pascal
von rechts nach links	__cdecl	cdecl
die ersten zwei in Registern, folgende von rechts nach links	__fastcall	-
von links nach rechts	WINAPI	pascal
die ersten drei in Registern, folgende von links nach rechts	-	register
von rechts nach links	__stdcall	stdcall

Bsp.:

```
void __stdcall Tausch(int& i, int& j)
{
  ...
}

procedure Tausch(var i, j: Integer); stdcall;
begin
  ...
end;
```

Wenn keine expliziten Angaben bei der Funktionsdeklaration erfolgen, wird in Visual C++ und Object Pascal die Standard-Aufrufkonvention benutzt. Standard-Aufrufkonvention ist in:

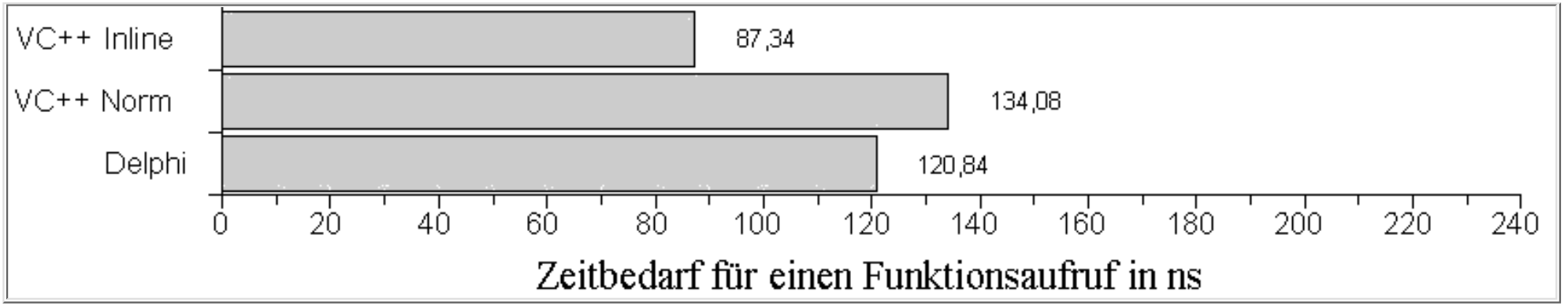
Visual C++:	__cdecl
Object Pascal:	register
Win32 - API:	stdcall

Funktionen können in C++ mit dem Vorsatz "Inline" versehen werden.

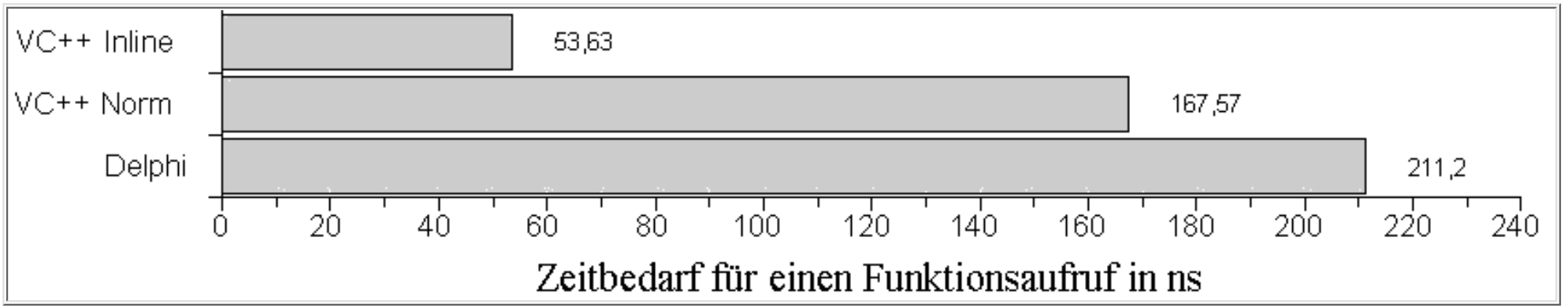
```
inline int Quadrat(int m)
{
  return m * m;
}
```

An jeder Stelle im Programm, an dem der Compiler Quadrat vorfindet, ersetzt dieser den Funktionsaufruf durch den Anweisungsteil des Funktionsrumpfes. Es liegt somit kein Funktionsaufruf mehr vor. Die Inline-Deklaration von Funktionen ist nur sinnvoll, wenn der Funktionsrumpf sehr klein ist.

Um den Geschwindigkeitsvorteil von Inline-Funktionen gegenüber normalen Funktionen in Visual C++ und Object Pascal einschätzen zu können, wurden eigene Messungen angestellt. Die oben angegebene Funktion Quadrat wurde dazu 90 Millionen mal aufgerufen und die benötigten Zeiten gemessen.



Weitere Messungen sollten das Zeitverhalten klären, wenn statt der Integer-Ganzzahlen Fließkomma-Zahlen vom Typ Double als Parameter und Rückgabewert der Funktion Quadrat eingesetzt werden.



Im Extremfall können also Inline-Funktionen bis zu 3-4 mal schneller als normale Funktionen sein.

Eine weitere Ersatz-Form für sehr kleine Funktionen stellen in C++ Makros dar. Diese werden durch den Präprozessor vor dem eigentlichen Übersetzen im Quelltext expandiert.

```
#define Quadrat(m) ((m) * (m))
#define Kubik(m)   (Quadrat(m) * (m))
   ...
printf("%d", Kubik(2));
```

```
// wird expandiert zu printf("%d", 2 * 2 * 2);
```

Bei Makros ist auf sorgsame Klammersetzung zu achten, um Expansions-Fehler, wie im folgenden Beispiel, zu vermeiden:

```
#define Quadrat(m) m * m
   ...
printf("%d", Quadrat(3+2));
// wird expandiert zu printf("%d", 3+2 * 3+2);
// man erhält als Ergebnis nicht 25 sondern 11
```

Makros bieten keine Typsicherheit. Inline-Funktionen und Makros müssen in Object Pascal durch normale Funktionen realisiert werden.

Eine wesentliche Erweiterung der Sprache C++ stellen Schablonen (Templates) dar. Sie sind in Object Pascal nicht implementiert.
Funktions-Schablonen gestatten mit Hilfe des Schlüsselworts "template" die Definition einer Familie von Funktionen, die mit verschiedenen Typen arbeiten können. Sie stellen sogenannten "parametrischen Polymorphismus" (parametric polymorphism) zur Verfügung, der es erlaubt, den selben Code mit verschiedenen Typen zu nutzen, wobei der Typ als Parameter im Code angegeben wird.

C++: ohne Funktions-Schablone	C++: mit Funktions-Schablone
`// Minimum - Funktion für` `// Integer-Zahlen` `int Min(int a, int b)` `{` `  if (a < b) return a;` `  else return b;` `};` `// Minimum - Funktion für` `// Double-Zahlen` `double Min(double a, double b)` `{` `  if (a < b) return a;` `  else return b;` `}; `	`template <class Typ>` `Typ Min(Typ a, Typ b)` `{` `  if (a < b) return a;` `  else return b;` `};` `void main()` `{` `  int i = Min(8, 3);` `  double d = Min(8.2, 3.1);` `  char c = Min('Ü', 'Ö');` `}`

```
// Minimum - Funktion für
// Char-Zeichens
char Min(char a, char b)
{
  if (a < b) return a;
  else return b;
};
```

Durch die Angabe von "class" im Ausdruck ***template <class Typ>*** wird in diesem Zusammenhang nur angegeben, daß *Typ* für einen noch zu spezifizierenden Typ steht. Durch die Definition der Schablone wird noch keine Funktion *Min* definiert. Eine Definition mit konkreter Typangabe wird vom Compiler erst dann angelegt, wenn im Programm ein Funktionsaufruf erfolgt (im Beispiel Min(8, 3)). Der Compiler ersetzt dann den abstrakten *Typ* durch einen konkreten Typ; er "instanziert" eine spezielle Version der Schablone.

Durch den Einsatz von Schablonen gelangt man zu großer Flexibilität, kann Redundanzen im Code vermeiden und muß doch trotzdem nicht auf die strikte Typprüfung durch den Compiler verzichten. Ihr Einsatz kann zudem eine Verkleinerung des vom Compiler erzeugten Programms bewirken. Die Beschreibung von Schablonen ist nicht nur für Funktionen sondern auch für Klassen möglich (vgl. "2.3.6.9 Klassen-Schablonen").

Die Funktion main() oder wmain() wird in Visual C++ als Startpunkt für die Programmausführung benutzt. Programme, die wmain() benutzten, werden Programmstart-Argumente im 16 bit breiten UNI-Zeichencode übergeben. Jedes VC++ Programm muß eine main() oder wmain() Funktion besitzen. Der Startpunkt aller Pascal-Programme wird durch einen globalen Funktionsblock gebildet, der mit begin eingeleitet und mit end. beendet wird. Ebenso wie jedes Programm, muß auch jede Unit mit dem Schlüsselwort end, gefolgt von einem Punkt enden: end.

Größere Programme sollte man nicht monolithisch in einem Block erstellen, sondern über mehrere Dateien verteilen. Um die einzelnen Programme zusammenzuführen oder auch bereits existierende Bibliotheken einem Programmtext hinzuzufügen, verwenden beide Compiler unterschiedliche Konzepte. In C++ werden Programmteile in mehreren Dateien mit der Endung CPP (= C Plus Plus) implementiert. In getrennten Header Dateien (*.H) werden die Schnittstellen der CPP-Dateien veröffentlicht. Durch die Verwendung der "#include"- Präprozessor-Anweisung können einzelne Header-Dateien in einem CPP-Modul eingefügt werden. Dem Compiler sind dadurch externe Deklarationen mit exakten Typangaben bekannt, so daß er alle Programm-Module erfolgreich übersetzen kann. Nach dem Übersetzen werden die nun übersetzten Module (sogenannte Objekt-Dateien mit der Endung OBJ) durch den Linker zusammengebunden. Es entsteht das ausführbare Programm (Standard-Endung EXE) oder eine Bibliothek (Standard-Endung DLL).

Objekt Pascal verwendet das sogenannte Konzept der Units. Eine Hauptdatei mit der Dateiendung DPR (= Delphi PRojekt) oder PAS (= PAScal) wird durch einen Ausdruck program <ProgrammName> oder library <LibraryName> eingeleitet. Alle weiteren Programm-Module werden in Units (Dateiendung PAS) untergebracht. Sie bestehen aus zwei Abschnitten: einem mit dem Schlüsselwort Interface eingeleiteten, öffentlichen und einem mit dem Schlüsselwort Implementation eingeleiteten, privaten Implementierungsteil. Im Interface-Abschnitt von Pascal-Units werden wie bei Visual C++ die Schnittstelleninformationen in den Header-Dateien veröffentlicht. Der Implementation-Teil beinhaltet die tatsächliche Realisierung einzelner Programmteile und entspricht dem Inhalt einzelner CPP-Module. Übersetzte Units erhalten die Endung DCU (= Delphi Compiled Unit).

Durch das Schlüsselwort "uses" können einem Pascal-Programm oder einer Unit andere Pascal-Units bekannt gemacht werden. Zu einzelnen Units

können sogenannte Formulare gehören, die das visuelle Gestalten von Programmoberflächen erlauben. Sie haben stets denselben Namen wie die zugehörige Unit, besitzen aber die Dateiendung DFM (=Delphi ForMular).

<table>
<tr><th>C++</th><th>Object Pascal</th></tr>
<tr>
<td>Syntax:

#include "<i>Datei</i>"
#include <<i>Datei</i>>

<i>Datei</i> darf Pfadangaben enthalten. Bei der ersten Version wird <i>Datei</i> zunächst im aktuellen Verzeichnis gesucht. Nur wenn sie dort nicht gefunden wurde, wird sie im Pfad gesucht. Pro Zeile darf nur eine include Anweisung erscheinen.

Wirkung:

Der ganze Inhalt von <i>Datei</i> wird durch den Präprozessor an der Stelle im Quelltext eingefügt, an dem die #include Anweisung auftritt.</td>
<td>Syntax 1:

program | library Name;
uses Unit1 [in 'Datei1'],
... ,
UnitN [in 'DateiN'];

Syntax 2:

Unit Name;
interface
uses Unit1, ... , UnitN;
...
implementation
uses UnitN+1, ... , UnitM;
...

initialization
...
finalization
...
end.

Bei Programmen und Bibliotheken können Datei- und Pfadangaben dem Schlüsselwort in folgen.

Das Schlüsselwort uses darf in Programmen und Bibliotheken nur ein einziges Mal, in Units nur einmal pro Abschnitt auftreten. Alle einzubindenden Units müssen in einer kommaseparierten Liste aufgeführt werden.</td>
</tr>
</table>

Bsp.:

```
// Test Programm,
// das nichts tut

#include <windows.h>
#include "c:\bwcc32.h"
#include <string.h>

void main()
{
}
```

Bsp.:

```
program Test;
{$APPTYPE CONSOLE}

uses
  Windows,
  BWCC32 in 'C:\BWCC32.PAS',
  SysUtils;

{$R Test.res} // Einbinden von
              // Ressourcen
begin
end.
```

Auch in Delphi kann der Inhalt ganzer Dateien durch die Compiler-Anweisung {$I *Datei*} oder {$INCLUDE *Datei*} in Quelltexte eingefügt werden.

Am letzten Beispiel erkennt man einen weiteren Unterschied: anders als in Visual C++ werden in Delphi zusätzliche Dateien, die zum endgültigen Programm dazugelinkt werden sollen, direkt im Programmtext über Compiler-Anweisungen eingebunden. Ressource-Dateien werden mittels {$R *Datei.RES*} oder {$RESOURCE *Datei.RES*}, Objektdateien aus anderen Programmiersprachen im INTEL-Object-Format durch {$L *Datei.OBJ*} oder {$LINK *Datei.OBJ*} hinzugefügt. In Delphi arbeiten, durch das Konzept der Units bedingt, Übersetzer und Linker wesentlich enger zusammen, als dies in Visual C++ der Fall ist. Nachdem eine Anwendung einmal übersetzt wurde, werden Programmteile, die sich nicht verändert haben, nicht neu übersetzt. Sie können im Speicher gehalten und direkt vom Speicher in die zu bildende EXE-Datei eingebunden werden. Nicht benutzte Funktionen und Variable werden nicht mit in das zu bildende Programm aufgenommen (smart linking).

Aber auch Visual C++ bietet ein "inkrementelles Linken", bei dem nur veränderte Programmteile in der entstehenden Anwendung neu eingebunden werden. Kürzere Linkzeiten sind die Folge. Die Turn-Around Zeiten in Visual C++ sind aber im Vergleich zu Delphis Compilier- und Link- Zeitbedarf spürbar länger.

Die zentrale Verwaltung eines jeden VC-Programms stellt die Projektdatei dar (Projektname.MDP), die man bei Microsoft "Workspace" nennt. In dieser werden, neben den Einstellungen auch für mehrere Erstellungsoptionen, intern alle zum Projekt gehörenden Dateien vermerkt. Beim Übersetzen erstellt Visual C++ eine Make-Datei, mit Hilfe derer beim Übersetzen zwei voneinander getrennte Vorgänge gesteuert werden: erstens der Compile- und im Anschluß daran der Link - Vorgang. Delphi kann dagegen nur eine Erstellungsoption pro Projekt verwalten und speichert diese in einer Datei mit dem Namen Projektname.DOF ab.

Bei der Verwendung von Bibliotheken fremder Hersteller kann es leicht passieren, daß in einer Header-Datei oder einer Delphi-Unit ein Bezeichner (Identifier) definiert wurde, der bereits in einer anderen Header-Datei bzw. Delphi-Unit verwendet wird. Natürlich kann für unterschiedliche Aufgaben nicht zweimal derselbe Name verwendet werden; der Compiler wird die Übersetzung an der entsprechenden Stelle mit einer Fehlermeldung abbrechen.

VC++	Object Pascal
`// Lib1.h` `char Drucke(char c);` `...` `// Lib2.h` `void Drucke(int Len;` `char* Text);` `...`	`Unit Lib1;` `interface` `function Drucke(c: char);` `...` `Unit Lib2;` `interface` `function Drucke(Len: Integer;` `Text: PChar);` `...`

Zur Lösung dieses Problems können in Visual C++ "Namensbereiche" (Namespaces) definiert werden. Alle in einem Namensbereich deklarierten Bezeichner bekommen einen frei definierbaren Zusatzbezeichner hinzugefügt, der es weniger wahrscheinlich macht, daß Mehrdeutigkeiten auftreten können.

Alle Bezeichner in einer Object Pascal-Unit besitzen implizit so einen Zusatz-Bezeichner, der den Namen der Unit trägt.

VC++	Object Pascal

```
// Lib1.h
namespace Lib1
{
  char Drucke(char c);
  ...
}

// Lib2.h
namespace Lib2
{
  void Drucke(int Len,
              char* Text);
  ...
}
```

```
Unit Lib1;
interface
  function Drucke(c: char);
  ...



Unit Lib2;
interface
  function Drucke(Len: Integer;
                  Text: PChar);
  ...
```

Ein Aufruf erfolgt dann in der Art:

VC++	Object Pascal
`#include "Lib1.h"` `#include "Lib2.h"` `Lib1::Drucke('a');` `Lib2::Drucke(10, "Hallo Welt");`	`Uses` `  Lib1, Lib2;` `Lib1.Drucke('a');` `Lib2.Drucke(10, 'Hallo Welt');`

Selbst einige Bezeichner in der Klassenbibliothek Delphis führen zu Namenskonflikten mit Funktionen des Windows-API, so daß sogar hier manchmal die Unit-Namen mit angegeben werden müssen (z.B. VCL-Klasse Graphics.TBitmap und Record Windows.TBitmap oder VCL-Methode OleAuto.RegisterClass(...) und Funktion Windows.RegisterClass(...)).

Zurück zum Inhaltsverzeichnis

Weiter in Kapitel 2.3.6

2.3.6 Klassen und Objekte

2.3.6.1 Struktur und Realisierung

Ebenso wie viele andere Sprachelemente ähneln sich auch Klassen und Objekte in C++ und Object Pascal. Sie bilden die Basis für die sehr umfangreichen Klassenbibliotheken Microsoft Foundation Classes (MFC) und Visual Component Library (VCL). Diese wiederum bilden die Grundlagen und Voraussetzungen für die Entwicklungsumgebungen, die ein visuelles Programmieren ermöglichen.

In Klassen und Objekten können Variable, Funktionen und Eigenschaften (Properties) deklariert werden. Variable in Klassen und Objekten werden Datenelemente, Member-Daten und Felder genannt. Klassen-Funktionen werden als Member-Funktionen und Methoden bezeichnet. Im folgenden werden die Bezeichnungen **Feld**, **Methode** und **Property** verwandt. Felder, Methoden und Properties sind Member, die **Elemente**, der Klassen und Objekte.

Eine Tabelle soll zunächst einen Überblick darüber verschaffen, in welchem Umfang Visual C++ und Object Pascal den Typ Klasse unterstützen:

	VC++	Object Pascal
differenzierte Zugriffsrechte	Ja	Ja
Einfachvererbung (single inheritance)	Ja	Ja
Mehrfachvererbung (multiple inheritance)	Ja	Nein
Zugriff auf Vorfahr mit "inherited"	Bedingt	Ja
statische Methoden	Ja	Ja
virtuelle Methoden (Polymorphismus)	Ja	Ja
dynamische Methoden	Nein	Ja
Botschaftsbehandlungs-Methoden	Nein	Ja
Botschaftszuteilung	Nein	Ja
static-Felder	Ja	Nein
static-Methoden = Klassen-Methoden	Ja	Ja
const-Felder und Methoden	Ja	Nein
inline-Methoden	Ja	Nein
Objekt-Variable in Klasse (Vorwärts-Dekl.)	Ja	Ja
verschachtelte Klassen	Ja	Nein
Konstruktor	Ja	Ja
KopieKonstruktor	Ja	Bedingt
virtueller Konstruktor	Nein	Ja
Destruktor	Ja	Ja
Metaklassen	Nein	Ja
Klassen-Schablonen (class Templates)	Ja	Nein
Überladen von Methoden und Operatoren	Ja	Nein
Friends	Ja	Bedingt
Feld-Zeiger für Klassen	Ja	Nein
Methoden-Zeiger für Klassen	Ja	Ja
abstrakte Basisklassen / -methoden	Ja	Ja
Properties (Eigenschaften)	Nein	Ja

Run-Time-Type-Information (RTTI)	Ja	Ja
Feld- / Methoden-Name nachschlagen	Nein	Ja

Eine **Klassen-Definition** besteht in C++ und Object Pascal aus einem Klassenkopf, zu dem das Schlüsselwort "class" und der Name der Klasse gehören, und einem Klassenrumpf. In C++ kann und in Object Pascal muß die Implementierung der Methoden abgesetzt von der Klassen-Definition erfolgen. C++ erlaubt es, die Methoden-Implementierung auch direkt innerhalb des Klassenrumpfes, im Anschluß an die Methoden-Deklaration, vorzunehmen. Eine Klasse Komplex, welche die in der Mathematik verwendeten komplexen Zahlen auf dem Computer abbilden soll, soll als Beispiel dienen:

VC++

```
class Komplex   // KlassenKopf
{               // KlassenRumpf
  public:
    void Zuweisung(double r,
                   double i)
         {Real = r; Imag = i;}

    void Ausgabe()
         {printf("%g + i*%g",
                 Real, Imag);}
  private:
    double Real;
    double Imag;
};
```

Object Pascal

Methoden-Implementierung erfolgt in Object Pascal immer abgesetzt vom Klassenrumpf

```
class Komplex   // KlassenKopf
{               // KlassenRumpf
  public:
    void Zuweisung(double r,
                   double i);
    void Ausgabe();
  private:
    double Real;
    double Imag;
};

// abgesetzte
// Methodenimplementierung
void Komplex::Zuweisung(
            double r, double i)
{
  Real = r;
  Imag = i;
}

void Komplex::Ausgabe()
{
  printf("%g+i*%g",Real,Imag);
}
```

```
type
  Komplex = class // KlassenKopf
  public          // KlassenRumpf
    procedure Zuweisung(r, i:
                        Double);
    procedure Ausgabe;
  private
    Real: Double;
    Imag: Double;
  end;

// abgesetzte
// Methodenimplementierung
procedure Komplex.Zuweisung(
                  r, i: Double);
begin
  Real:= r;
  Imag:= i;
end;

procedure Komplex.Ausgabe;
begin
  writeln(Real, '+i*', Imag);
end;
```

Auch in C++ werden für gewöhnlich Klassenrumpf und Methodenimplementierung voneinander getrennt. Der Klassenrumpf, die Klassen-Schnittstelle, wird von der Klassen-Implementierung entkoppelt. Die Methoden-Deklarationen stellen dann eine Art Forward-Deklaration dar. Der Klassenrumpf wird in Header-Dateien, die Methodenimplementierung in CPP-Moduldateien definiert. Entsprechend wird in Object Pascal der Klassenrumpf im Interface-Abschnitt und die Methoden-Implementierung im Implementation-Abschnitt einer Unit eingefügt. Über Include- bzw. Uses-Anweisungen können in anderen Modulen bestehende Klassen in beiden Sprachen leicht bekannt gemacht werden.

Klassen belegen keinen Speicher, da sie nur eine spezielle Typ-Deklaration darstellen. Eine Speicherbelegung erfolgt erst beim Instanzieren von Klassen. Eine Klasseninstanz heißt Objekt. C++ gestattet das Anlegen statischer und dynamischer Instanzen.

In Object Pascal sind dagegen, genauso wie z.B. in der "Internet-Sprache" Java der Firma Sun Microsystems, ausschließlich dynamische Objekt-Instanzen zulässig. Der Vorschlag des ANSI-ISO-Komitees für ein Objekt-Modell in Pascal [8, S. 12 / S. 44], genannt **Referenzmodell**, wurde fast vollständig in Object Pascal realisiert. Eine Variable vom Typ einer Klasse enthält kein Objekt, sondern einen Zeiger. Dieser Zeiger kann zur Laufzeit ein Objekt referenzieren oder den Wert nil enthalten um anzudeuten, daß er gerade kein gültiges Objekt referenziert. Beim Erzeugen eines neuen Objekts wird dem Zeiger die Adresse des Speicherbereiches zugewiesen, der für das Objekt reserviert wurde. D.h., alle Objekte werden auf dem Heap angelegt und belegen immer nur 4 Byte im Stack-Speicher. Der Heap bezeichnet den Speicher, den eine Anwendung belegt, um z.B. dynamische Variable erstellen zu können.

Felder und Methoden dynamisch angelegter Objekte werden normalerweise wie bei gewöhnlichen dynamischen Variablen angesprochen: Zeiger->Feld in C++ und Zeiger^.Feld in Object Pascal. Da nun aber in Object Pascal sowieso *alle* Objekte dynamisch angelegt werden und Zeiger darstellen, kann der Compiler beim Zugriff auf die Elemente eines Objekts eine Dereferenzierung automatisch durchführen. Es ist weder notwendig noch möglich, den Zeiger explizit zu dereferenzieren. Ein Objekt stellt sich trotz dynamischer Allokation wie eine statische Variable dar. Ein Zugriff erfolgt, ebenso wie in Java, in der Form ObjektZeiger.Feld bzw. ObjektZeiger.Methode. Beim Zugriff auf das gesamte Objekt wird der Zeiger nicht dereferenziert. Statt dessen wird mit dem Wert des Zeigers, also der Adresse der Instanz gearbeitet. Das wirkt sich insbesondere bei Zuweisungen aus.

Das Object Pascal Referenzmodell wird aufgrund seiner Eigenschaft des "automatischen Dereferenzierens" mitunter "**zeigerloses Konzept**" genannt. Zeigerlos deshalb, weil vor dem Programmierer die Tatsache verborgen wird, daß er in Wahrheit ständig mit Zeigern hantiert. Tatsächlich scheint es für Programmier-Anfänger hilfreich zu sein, zunächst von Zeigern und der Zeiger-Syntax verschont zu bleiben. Das Referenz-Modell wird vielfach als leichter verständlich und einfacher lehrbar empfunden.

Instanzierung, Nutzung und Freigabe des Objekts Komplex kann in beiden Sprachen so erfolgen:

VC++	Object Pascal
statisch `Komplex a;` `a.Zuweisung(10, 5);` `a.Ausgabe();`	keine statische Instanzierung möglich
dynamisch `Komplex* b;` `b = new Komplex;` `b->Zuweisung(10, 5);` `b->Ausgabe();` `delete b;`	dynamisch `var b: Komplex;` `b:= Komplex.Create;` `b.Zuweisung(10, 5);` `b.Ausgabe;` `b.Free;`

Statische Objekte werden, wie alle anderen statischen Variablen auch, automatisch freigegeben, sobald ihr Gültigkeitsbereich verlassen wird.

Beachtet werden muß jedoch, daß dynamisch erzeugte Objekte vom Programmierer selbst wieder freigegeben werden müssen, da sonst mit der Zeit Speicher-Löcher (=unbenutzter, nicht freigegebener Speicher) entstehen. Eine Einbettung in einem try/finally-Ressourcenschutzblock des Codes, der auf dynamisch erzeugte Objekte zugreift, ist ratsam, um eine sichere Freigabe des belegten Speichers zu erwirken.

VC++	Object Pascal
`Komplex* b;` `b = new Komplex;` `__try` `{` `  b->Zuweisung(10, 5);` `  b->Ausgabe();` `}` `__finally` `{` `  delete b;` `}`	`var b: Komplex;` `b:= Komplex.Create;` `try` `  b.Zuweisung(10, 5);` `  b.Ausgabe;` `finally` `  b.Free;` `end;`

Die wichtigen Ziele objektorientierter Programmierung, Abstraktion und Verkapselung, werden durch die gezielte Vergabe von Zugriffsrechten an Felder und Methoden in Klassen erzielt. Folgende **Zugriffsrechte**, auch Sichtbarkeitsbereiche genannt, stehen zur Verfügung:

	VC++	Object Pascal
private	Nur klasseneigene Methoden und deren *Friends* haben Zugriffsrecht. `class Any` `{` `  private: int x;` `};` `Any* MyObjekt = new Any;` `MyObjekt->x = 15;` `// Fehler, da kein` `// Zugriffsrecht`	Unbeschränkte Zugriffsrechte innerhalb *des* Moduls (z.B. Unit), in dem die Klasse definiert ist. Außerhalb des Moduls besteht keinerlei Zugriffsrecht. `type` `  Any = class` `  private x: Integer;` `  end;` `MyObjekt := Any.Create;` `MyObjekt.x := 15;` `// kein Fehler, da Zugriff` `// im selben Modul ge-` `// stattet ist`
protected	Zugriffsrechte wie bei private. Außerdem haben aber auch abgeleitete Klassen Zugriffsrecht.	Zugriffsrechte wie bei private. Außerhalb des Moduls, in dem die Klasse definiert ist, haben aber außerdem auch abgeleitete Klassen Zugriffsrecht.
public	Alle haben uneingeschränkte Zugriffsrechte.	Alle haben uneingeschränkte Zugriffsrechte.
published	-	Dient der Veröffentlichung von Properties im Objektinspektor. Uneingeschränkte Zugriffsrechte (wie public)

automated	-	Dient der Veröffentlichung von OLE- Automations - Typ - Informationen. Uneingeschränkte Zugriffsrechte (wie public)

C++ und Object Pascal können innerhalb von Objekten weitere Objekte über Zeigervariable aufnehmen. Um wechselseitig voneinander abhängige Klassen deklarieren zu können, werden **Vorwärtsdeklarationen** eingesetzt.

VC++	Object Pascal
class B; // forward-Deklar. **class A**{ public: B* ObjB; }; **class B**{ public: A* ObjA; };	type **B = class; // forward-Deklar.** **A = class** public ObjB: B; end; **B = class** public ObjA: A; end;

Auch in Object Pascal wird also bei der Vorwärtsdeklaration einer Klasse, anders als bei Vorwärtsdeklarationen von Funktionen, die Anweisung "forward" nicht benötigt.

Während Object Pascal verschachtelte Funtkionsdefinitionen, jedoch keine verschachtelten Klassendefinitionen zuläßt, drehen sich die Verhältnisse in C++ paradoxerweise genau um. C++ gestattet die Definition **verschachtelter Klassen** (nested classes).

```
class A{                // äußere Klassen-Definition
public:
  class B{              // innere Klassen-Definition
  public:
    void DoIt_B(void);
  private:
    char c;             // A::B::c
  };
  void DoIt_A(void);
private:
  char c;               // A::c
};

void A::B::DoIt_B()
{
  c = 'b';
  printf("B:  %c\n", c);
};

void A::DoIt_A()
```

```
{
  B ObjB;

  c = 'a';
  ObjB.DoIt_B();
  printf("A:  %c\n", c);
};

void main()
{
  A ObjA;
  A::B ObjB;

  ObjA.DoIt_A();
  ObjB.DoIt_B();
}
```

Ausgabe:

B: b

A: a

B: b

Die innere Klasse (hier B) ist für die äußere, sie umschließende Klasse (hier A), lokal. Verschachtelte Klassen werden manchmal auch Member-Klassen genannt.

Das ANSI-ISO-Komitee hat verschachtelte Klassen in OO-Pascal abgelehnt, weil "keine guten Gründe gefunden werden konnten, Verschachtelung zuzulassen und die Sprachdefinition, -implementation und Nutzung unnötig verkompliziert wird". [8, S. 41/42]

Klassen lassen sich in C++ auch lokal innerhalb von Funktionen definieren.

```
void MyProc(void)  // Prozedur
{
  class B{         // lokale Klasse
    int i;
    ...
  };

  B Obj;
   ...
};
```

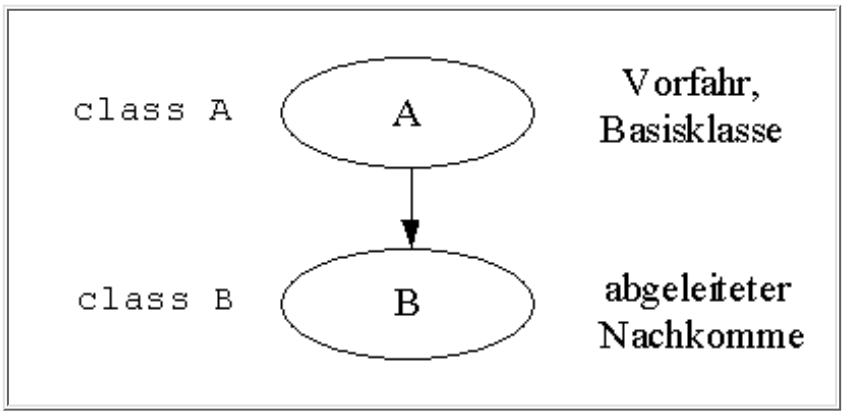
Visual C++ gestattet keine Definition von Methoden in lokalen Klassen. Lokale Klassen dürfen außerdem nicht von anderen Klassen erben. Aufgrund dieser starken Einschränkungen sind sie praktisch nicht interessant. Christian merkt dazu in [3] an: "Mit den verschachtelten und den lokalen Klassen hat man versucht, eine Nische zu füllen. Allerdings hat man sich damit weit von der eigentlichen Philosophie von C++ und der objektorientierten Programmierung entfernt." Die beide Spracheigenschaften *verschachtelte* und *lokale Klassen* stellen Beispiele für das unnötige Überfrachten einer Sprache dar.

2.3.6.2 Vererbung (Inheritance)

Bei der Definition einer neuen Klasse kann eine bereits bestehende Klasse als *Vorfahr* (ancestor) angegeben werden. Die neue Klasse, auch

Nachkomme genannt, kann dadurch automatisch über sämtliche Felder, Methoden und Properties des Vorfahren verfügen. Sie *erbt* diese Elemente von ihrer *Basisklasse* (base class). Die neue Klasse ist von ihrem Vorfahr *abgeleitet* (derived).

In Object Pascal ist die Klasse TObject der Urahn aller Klassentypen. Alle Klassen, die nicht explizit von einer anderen Klasse abgeleitet werden, werden automatisch von TObject abgeleitet. In Visual C++ haben nicht-abgeleitete Klassen dagegen keinen generellen Vorfahren. Programme, die die MFC benutzen, leiten ihre Klassen oft von CObject oder von einem Nachkommen CObject's ab.



VC++	Object Pascal
class A **{** public: int Feld1; int Methode1(); **};** **class B : public A** **{** // erbt alles von A **}** int **A::**Methode1() { printf("A: Methode 1\n"); return 1; }; ... A* ObjA; B* ObjB; ObjA = new A; ObjB = new B; **ObjA**->Methode1(); **ObjB**->Methode1(); delete ObjA; delete ObjB; Ausgabe: A: Methode 1 A: Methode 1	type **A = class** public Feld1: Integer; function Methode1: Integer; property Property1: Integer read Feld1 write Feld1; **end;** **B = class(A)** // erbt alles von A **end;** function **A.**Methode1: Integer; begin writeln('A: Methode 1'); Result:= 1; end; ... var ObjA: A; ObjB: B; ObjA:= A.Create; ObjB:= B.Create; **ObjA**.Methode1; **ObjB**.Methode1; ObjA.Free; ObjB.Free; Ausgabe: A: Methode 1 A: Methode 1

Zusätzlich können neue Elemente deklariert oder vom Vorfahren geerbte Elemente überschrieben werden. **Überschreiben** bedeutet, es wird für ein Element der neuen Klasse ein Bezeichner gewählt, der bereits im Vorfahren verwendet wird. Dadurch wird die überschriebene Komponente innerhalb der neuen Klasse verdeckt.

VC++

```
class A
{
  public:
    int Feld1;
    int Methode1();
};


class B : public A
{
  public:
  int Feld1;
  // überschriebenes Feld1
  int Methode1();
  // überschrieben
};


int A::Methode1()
{
  printf("A: Methode 1\n");
  return 1;
};

int B::Methode1()
{
  printf("B: Methode 1\n");
  return 1;
};

...

A* ObjA;
B* ObjB;

ObjA = new A;
ObjB = new B;

ObjA->Methode1();
ObjB->Methode1();

delete ObjA;
delete ObjB;
```

Ausgabe:

A: Methode 1

B: Methode 1

Object Pascal

```
type
  A = class
  public
    Feld1: Integer;
    function Methode1: Integer;
    property Property1: Integer
        read Feld1 write Feld1;
  end;

  B = class(A)
  public
    Feld1: Integer;
    // alle überschrieben
    function Methode1: Integer;
    property Property1: Integer
        read Feld1 write Feld1;
  end;


function A.Methode1: Integer;
begin
  writeln('A: Methode 1');
  Result:= 1;
end;

function B.Methode1: Integer;
begin
  writeln('B: Methode 1');
  Result:= 1;
end;

...

var ObjA: A;
    ObjB: B;

ObjA:= A.Create;
ObjB:= B.Create;

ObjA.Methode1;
ObjB.Methode1;

ObjA.Free;
ObjB.Free;
```

Ausgabe:

A: Methode 1

B: Methode 1

Das Überschreiben hat aber keinen Einfluß auf den Vorfahren selbst. Man beachte, daß Objekt B zwei Felder und zwei Methoden (und in Pascal zwei Properties) besitzt. Da aber jeweils die alten und die neuen Elemente denselben Namen besitzen, werden die alten Elemente *verborgen bzw. überschrieben, jedoch nicht ersetzt.* Der explizite Zugriff auf Elemente vorhergehender Klassen erfolgt in C++ mit Hilfe des Scope-Operators über die genaue Angabe der (Vorfahr-)Klasse und in Object Pascal mit Hilfe des Schlüsselwortes inherited. Mit inherited wird immer auf die vorhergehende Klasse zugegriffen, ohne daß man deren genauen Namen angeben muß.

Methode 1 in Klasse B und das Hauptprogramm werden im Beispiel abgeändert:

VC++	Object Pascal
`int B::Methode1()` `{` `  A::Methode1();` `  printf("B: Methode 1\n");` `  return 1;` `};` `...` `B* ObjB;` `ObjB = new B;` `ObjB->Methode1();` `delete ObjB;` Ausgabe: A: Methode 1 B: Methode 1	`function B.Methode1: Integer;` `begin` `  inherited Methode1;` `  writeln('B: Methode 1');` `  Result:= 1;` `end;` `...` `var ObjB: B;` `ObjB:= B.Create;` `ObjB.Methode1;` `ObjB.Free;` Ausgabe: A: Methode 1 B: Methode 1

Object Pascals *Inherited* darf nur in der Methodenimplementierung benutzt werden, während C++ die explizite Klassenangabe auch außerhalb der Methodenimplementierung (zum Beispiel im Hauptprogramm) gestattet, wenn die Klassen-Ableitung wie im Beispiel öffentlich (public) erfolgte:

```
B* ObjB;

ObjB = new B;
ObjB->Methode1();
ObjB->A::Methode1();
ObjB->A::Feld1 = 5;
ObjB->Feld1 = 6;
delete ObjB;
```

Durch die explizite und exakte Angabe der (Vorfahr-)Klasse in C++ werden, anders als in Object Pascal, statische Klassenbindungen erwirkt und polymorphes Verhalten dadurch unterlaufen. Einen zu Object Pascal kompatiblen Zugriff auf die Eltern-Klasse kann aber nach [6] mit relativ wenig Aufwand auch in C++ realisiert werden, indem in jeder Klasse durch eine typedef-Deklaration unter dem Bezeichner "inherited" die jeweilige Eltern-Klasse bekannt gemacht wird:

```
class A
{
  ...
```

```
};

class B : public A
{
  typedef A inherited;
  void virtual print();
  ...
};

class C : public B
{
  typedef B inherited;
  void virtual print();
  ...
};


void C::print()
{
  inherited::print();
  ...
};
```

Die jeweilige typedef-Deklaration stellt eine lokale Typ-Deklaration dar. Der Gültigkeitsbereich eines klassen-lokalen Typs ist auf die Klasse und deren Klassen-Methoden beschränkt, in dem er deklariert wurde. Durch den begrenzten Gültigkeitsbereich ist sichergestellt, daß inherited für jede Klasse einmalig ist und keine Konflikte mit Basis- oder abgeleiteten Klassen auftreten können, die einen eigenen Typ unter dem selben Namen deklariert haben.

Object Pascal gestattet keine lokalen Typ-Deklarationen.

Bei der Vererbung kann in C++ optional angegeben werden, ob die Vererbung *private* oder *public* erfolgen soll (voreingestellt ist *private*).

```
class B : public A
{
   ...
};
```

```
class B : private A
{
   ...
};
```

Protected- und *public*- Elemente der Basisklasse werden bei einer *private-Ableitung zu private*-Elementen. Ableitungen in Object Pascal entsprechen der public Ableitung in C++: die Zugriffsrechte der Basisklasse bleiben in der abgeleiteten Klasse erhalten.

Neben der Einfachvererbung (single inheritance) gestattet C++ zusätzlich **Mehrfachvererbung** (multiple inheritance), bei der eine abgeleitete Klasse aus mehreren Basisklassen gebildet wird. In Anlehnung an das vierte Bild in Kapitel 2.1 kann Klasse "Amphibienfahrzeug" durch Mehrfachvererbung aus Land- und Wasserfahrzeug gebildet werden:

```
class AmphibienFahrzeug : public LandFahrzeug,
                          private WasserFahrzeug
{
  ...
};
```

Probleme können Mehrdeutigkeiten bereiten, wenn Elemente mit gleichem Namen (aber unterschiedlicher Funktionalität) in mehreren Basisklassen existieren.

Bei der Entwicklung der Klassenbibliothek von Visual C++ wurde auf die Verwendung von Mehrfachvererbungen verzichtet. Die Online-Dokumentation merkt dazu an: "Wir haben herausgefunden, daß die Mehrfachvererbung weder bei der Entwicklung einer Klassenbibliothek, noch beim Schreiben ernsthafter Anwendungen benötigt wird." [13]

2.3.6.3 Zuweisungskompatibilität

In C++ können nur Klassen, die *public* abgeleitet wurden, vom Compiler implizit in ihre Basisklassen konvertiert werden. Solche C++ Klassen und alle Klassen in Object Pascal sind zuweisungskompatibel mit sich selbst und mit ihren Vorfahren.

VC++	Object Pascal
`class Figur{` `  ...` `};` `class Rechteck : public Figur{` `  ...` `};` `class RundRechteck : public` `  Rechteck{` `  ...` `};` `class Ellipse : public Figur{` `  ...` `};`	`type` `  Figur = class` `  ...` `  end;` `  Rechteck = class(Figur)` `  ...` `  end;` `  RundRechteck = class(Rechteck)` `  ...` `  end;` `  Ellipse = class(Figur)` `  ...` `  end;`
`Figur* F;` `Rechteck* R;` `RundRechteck* RR;` `Ellipse* E;` `  ...` `F = F;` `F = R;` `F = RR;` `F = E;` `R = RR;` `R = F;` `// Fehler, inkompatible Typen` `R = E;` `// Fehler, inkompatible Typen` `RR = R;` `// Fehler, inkompatible Typen`	`var F:  Figur;` `  R:  Rechteck;` `  RR: RundRechteck;` `  E:  Ellipse;` `  ...` `F := F;` `F := R;` `F := RR;` `F := E;` `R := RR;` `R := F;` `// Fehler, inkompatible Typen` `R := E;` `// Fehler, inkompatible Typen` `RR := R;` `// Fehler, inkompatible Typen`

Wie man sieht, gilt die Zuweisungskompatibilität umgekehrt nicht. Eine Klasse ist mit keinem ihrer Nachfahren direkt zuweisungskompatibel (R := F). Zwischen gänzlich unverwandten Klassen, die völlig verschiedenen Ästen der Objekthierarchie

entstammen, besteht ebenfalls keinerlei Zuweisungskompatibilität (R := E).

Durch statische und dynamische Typkonvertierung sind aber trotzdem auch solche Zuweisungen möglich:

VC++	Object Pascal
dynamische Objekttyp-Konvertierung: `RR = new RundRechteck;` `F = RR;` `...` `R = dynamic_cast<Rechteck*>(F);`	dynamische Objekttyp-Konvertierung: `RR := RundRechteck.Create;` `F := RR;` `...` `R := F as Rechteck;`
statische Objekttyp-Konvertierung: `F = new Figur;` `...` `R = static_cast<Rechteck*>(F);`	statische Objekttyp-Konvertierung: `F := Figur.Create;` `...` `R := Rechteck(F);`

Solche Typwandlungen funktionieren nur dann, wenn beide Objekttypen Zeiger sind (ist in Object Pascal immer der Fall).

Bei **dynamischer Typ-Konvertierung** verläuft die Zuweisung zur Laufzeit nur dann erfolgreich, wenn der zuzuweisende und zu konvertierende Objektzeiger auf ein Objekt zeigt, das zuweisungskompatibel zur Ziel-Variablen ist. Diese Bedingung ist im Beispiel oben erfüllt: Zunächst wurde F ein gültiges Objekt vom Typ *RundRechteck* zugewiesen. Durch Typkonvertierung wird das Objekt, auf das F zeigt (ein *RundRechteck*), in ein *Rechteck* konvertiert. *RundRechtecke* sind, wie weiter oben gesehen, zu *Rechtecken* zuweisungskompatibel, weil *RundRechteck* ein Nachfahre von *Rechteck* ist (also ist R = RR erlaubt).

Eine Besonderheit bei der dynamischen Typkonvertierung stellt in Object Pascal die Tatsache dar, daß bei der Verwendung des "as" - Operators in dem Ausdruck

```
MyObject as MyClass
```

"MyClass" nicht unmittelbar den Namen einer Klasse angeben muß, sondern auch eine Variable vom Typ einer "Metaklasse" sein kann. Die tatsächliche Klasse dieser Variablen wird dann erst zur Laufzeit entnommen; d.h. der Compiler weiß in diesem Fall zur Übersetzungszeit weder, *welcher* Objekttyp zu konvertieren ist, noch *in welchen* Objekttyp diese Unbekannte zu konvertieren ist. Alle Entscheidungen werden auf die Laufzeit verschoben.

Beim Fehlschlag einer dynamischen Konvertierung wird in Visual C++ eine Exception der Klasse "bad_cast" und in Object Pascal eine Exception der Klasse "EInvalidCast" ausgelöst. Dynamische Typ-Konvertierungen beruhen auf den Run-Time-Typ-Informationen (RTTI) von Klassen.

Bei **statischer Typ-Konvertierung** existiert *keine* Einschränkung der Art, daß der zu konvertierende Objektzeiger auf ein Objekt zeigen muß, das zuweisungskompatibel zum Zieloperanden ist. Dafür gehen aber bei der Nutzung der statischen Konvertierung polymorphe Eigenschaften des zu konvertierenden Objekts verloren. Die Methodenadressen virtueller Methoden werden nicht zur Laufzeit aus der Methodentabelle (VMT), sondern bei der Übersetzung statisch ermittelt.

2.3.6.4 Methoden und spezielle Felder

Methoden sind Prozeduren oder Funktionen, deren Funktionalität fest mit einer Klasse verknüpft ist. Methoden werden üblicherweise in Verbindung mit einer Instanz der Klasse aufgerufen. Bei Klassenmethoden ist auch ein Aufruf in Verbindung mit einer Klasse möglich.

Alle Methoden, die in einer Klasse deklariert werden, sind standardmäßig **statisch** (nicht zu verwechseln mit static!). Die Adresse der aufzurufenden Methode steht bereits zur Compilierzeit fest und kann vom Linker fest in das Programm eingetragen werden. Statische Methoden entsprechen in dieser Hinsicht gewöhnlichen Funktionen und Prozeduren außerhalb von Klassen. In C++ sind alle Konstruktoren statische Methoden.

Virtuelle Methoden werden in beiden Sprachen mit Hilfe des Wortes "virtual" deklariert. Die Adresse des Funktions-Aufrufes steht bei ihnen nicht bereits zur Zeit der Compilierung fest. Statt dessen wird die exakte Methoden-Adresse erst zur Laufzeit des Programms ermittelt.

Wird in C++ eine virtuelle Funktion in einer abgeleiteten Klasse neu definiert, dann ist auch diese Version der Funktion automatisch virtuell. Eine weitere Angabe von virtual ist erlaubt, aber nicht unbedingt nötig. In Object Pascal wird dagegen nur bei der Erstdeklaration einer virtuellen Funktion der Methodenzusatz virtual angegeben. Wird eine virtuelle Funktion in einer abgeleiteten Klasse neu definiert, dann muß hier durch Angabe des Wortes "override" kenntlich gemacht werden, daß diese Funktion eine neue Implementierung der virtuellen Basisfunktion darstellt. Andernfalls wird die neue Funktion als statische Funktion interpretiert und überschreibt (versteckt) die virtuelle Funktion. Wird in einer abgeleiteten Klassen eine bisher existierende, virtuelle Funktion erneut deklariert und mit dem Bezeichner virtual versehen, so geht in Object Pascal die Beziehung zur bisherigen virtuellen Funktionen verloren. Es wird vielmehr eine neue virtuelle Funktion erzeugt, welche die bisherige überschreibt (versteckt).

Die einmal festgelegte Liste formaler Parameter muß beim Überschreiben einer virtuellen Methode beibehalten bleiben. In C++ können zwar alle Methoden überladen werden, also mit gleichem Namen, aber veränderten Parametern neu definiert werden. Allerdings geht beim Überladen virtueller Methoden die polymorphe Beziehung zu möglichen Vorfahr-Methoden verloren. Es können also beim Überladen höchstens neue virtuelle Funktionen erstellt werden.

VC++	Object Pascal
`class Figur{` `public:` `  void virtual print();` `};` `class Rechteck : public Figur{` `  void virtual print();` `};` `class RundRechteck : public Rechteck{` `  void virtual print();` `};` `void Figur::print()` `{` `  printf("Ich bin die Figur");` `};` `void Rechteck::print()` `{` `  printf("Ich bin ein` `  Rechteck");` `};` `void RundRechteck::print()`	`type` `  Figur = class` `  procedure print; virtual;` `  end;` `  Rechteck = class(Figur)` `  procedure print; override;` `  end;` `  RundRechteck = class(Rechteck)` `  procedure print; override;` `  end;` `procedure Figur.Print;` `begin` `  writeln('Ich bin die Figur');` `end;` `procedure Rechteck.Print;` `begin` `  writeln('Ich bin ein` `  Rechteck');` `end;`

```
{
  printf("Ich bin ein
          RundRechteck");
};
```

```
procedure RundRechteck.Print;
begin
  writeln('Ich bin ein
           RundRechteck');
end;
```

```
Figur* F;
RundRechteck* RR;

RR = new RundRechteck;
F = RR;
F->print();
// print() von Figur
// aufgerufen!
delete RR;
```

Ausgabe:

Ich bin ein RundRechteck

```
var F:  Figur;
    RR: RundRechteck;

RR:= RundRechteck.Create;
F:= RR;
F.Print;
// Print von Figur
// aufgerufen!
RR.Free;
```

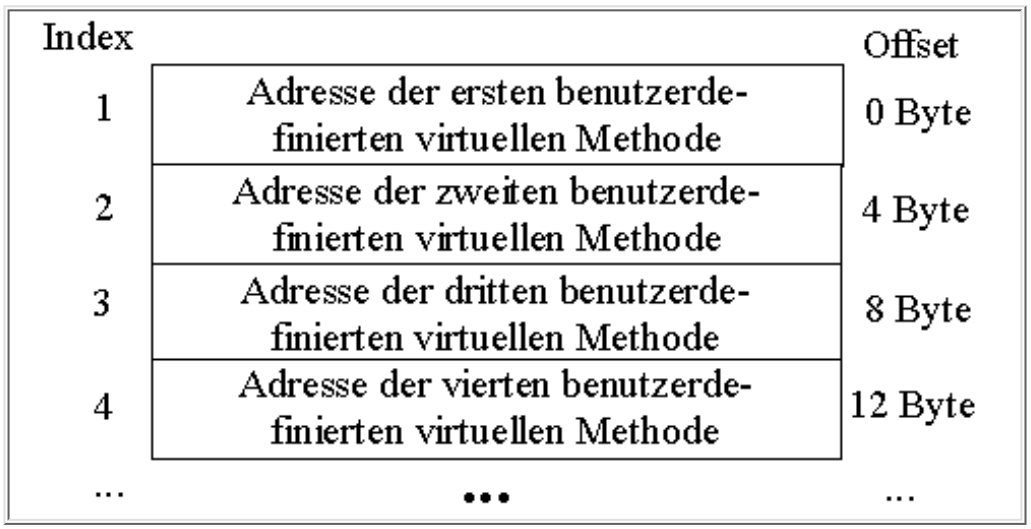
Ausgabe:

Ich bin ein RundRechteck

Wenn die Methoden print nicht virtuell deklariert worden wären, hätte man statt dessen die Ausgabe "Ich bin die Figur" erhalten.

Dynamische Methoden sind nur in Object Pascal vertreten und werden durch die Angabe von "dynamic" anstelle von "virtual" deklariert. Sie verhalten sich exakt gleich wie virtuelle Funktionen, unterscheiden sich jedoch in der (internen) Implementierung. Um den Sinn dynamischer Methoden zu verstehen, muß man zunächst untersuchen, wie virtuelle Methoden durch die Compiler praktisch realisiert werden.

In identischer Weise legen Visual C++ und Object Pascal eine sogenannte **virtuelle Methodentabelle**, abgekürzt VMT, für jede Klasse (nicht für jedes Objekt!) an. Die VMT wird in C++ manchmal auch V-Tabelle (abgekürzt vtbl) genannt. Sie stellt ein Array von 32 bit langen Pointern dar, die auf die virtuellen Methoden zeigen. Jede Methode, die einmal virtuell deklariert wurde, erhält hierin einen Eintrag:



Die Anordnung der Einträge erfolgt in derselben Reihenfolge wie bei der Deklarierung. Der Compiler kodiert einen Methodenaufruf einer virtuellen Funktion als indirekten Aufruf über dieses Array anhand des Indexes der virtuellen Methode. Dieser Index ist zur Zeit der Compilierung bekannt und für alle Methoden in einer virtual/override-Sequenz einer Klassenhierarchie gleich. Jede Klasse bekommt eine vollständige Kopie der VMT der Elternklasse, die nach Bedarf vergrößert wird, wenn die Klasse neue virtuelle Methoden deklariert. Bei virtuellen Methoden, die in Nachfahren neu definiert werden (override), ersetzt der Compiler einfach die Adresse im entsprechenden Eintrag der VMT durch die der neuen Methode.

Den Aufruf einer virtuellen Methode per Methoden-Index kann man sich zur Laufzeit grob so vorstellen:

1. Aktuelles Objekt und Klassentyp dieses Objekts ermitteln.
2. VMT-Array für diesen Klassentyp aufsuchen.
3. Element VMT_ARRAY[MethodenIndex] auslesen. Man erhält die Einsprung-Adresse, an der die Funktion beginnt, die für das aktuelle Objekt aufgerufen werden soll.
4. Zu dieser Adresse springen und Abarbeitung beginnen. Als Parameter wird der this- / Self - Parameter (implizit) an die Methode übergeben.

Der Aufruf virtueller Methoden läuft, trotz des großen Aufwands, recht schnell ab, benötigt aber natürlich minimal mehr Zeit als der Aufruf statischer Methoden.

Zur Realisierung dynamischer Methoden legt Object Pascal neben der VMT noch eine zweite, **dynamische Methodentabelle** (DMT) für jede Klasse an. Auch diese Tabelle stellt ein Array von Pointern auf Methoden dar, ist jedoch anders aufgebaut als die VMT. Zu jedem Pointer (=Value) existiert ein numerischer Schlüssel (=Key). Beim Aufruf einer *dynamic*- Methode wird nicht einfach ein Index zum Zugriff verwendet, sondern das Array anhand des Keys (der die Rolle des Indexes übernimmt) durchsucht. Anders als bei den VMTs hat jede Klasse nur *die* Methoden-Einträge in der DMT, die mit *dynamic* neu deklariert bzw. die mit *override* als polymorphe Nachfahren einer dynamischen Methode deklariert wurden. Dafür besitzen DMTs einen Zeiger, der auf die DMT der Elternklasse verweist. Wenn die Suche nach dem Key in der DMT der Klasse erfolglos blieb, werden die DMTs der Elternklasse(n) durchsucht, notfalls bis zurück zur DMT der Klasse TObject.

Dynamische Methoden bieten nur dann Vorteile, wenn eine Basisklasse sehr viele virtuelle Methoden und eine Anwendung zusätzlich sehr viele Nachkommen-Klassen deklariert, und zwar überwiegend ohne *override*. Der Mechanismus der dynamischen Methoden ist hauptsächlich für die Implementierung von Botschaftsbehandlungsmethoden gedacht. Gegenüber "normalen" virtuellen Methoden ist der Zugriff langsamer, dafür wird durch eine DMT aber weniger Speicherplatz verbraucht als für eine entsprechende VMT. Eine virtuelle Methodentabelle für eine bestimmte Klasse ist immer mindestens so groß, wie die VMT der Elternklasse.

Botschaftsbehandlungs-Methoden sind ebenfalls nur in Object Pascal realisiert. Sie dienen dazu, benutzerdefinierte Antworten auf dynamisch zugeteilte Botschaften zu implementieren. Sie sind intern als dynamische Methoden realisiert.

Windows-Programme werden oft durch einen Strom von Ereignissen aktiviert, die durch äußere Einflüsse hervorgerufen wurden. Anwendungen, die eine Benutzerschnittstelle aufweisen, werden durch ereignis-auslösende Eingabegeräte wie Tastatur und Maus bedient. Jedes Drücken einer Taste, jeder Klick mit der Maus erzeugt Botschaften, die die Anwendung empfängt. Es ist die Aufgabe der Anwendung, auf diese Botschaften sinnvoll im Sinne der Anwendung zu reagieren. Botschaftsbehandlungs-Methoden eignen sich hervorragend, um benutzerdefiniert auf Windows-Botschaften reagieren zu können. Folgerichtig setzt Delphis Klassenbibliothek Botschaftsbehandlungs-Methoden dazu ein, um die Behandlung von Windows-Botschaften zu implementieren. Andererseits sind aber die Botschaftsbehandlungs-Methoden vollkommen unabhängig vom Nachrichtensystem Windows.

Eine Methode wird zu einer botschaftsverarbeitenden Methode, indem an ihre Deklaration das Schlüsselwort "message", gefolgt von einer ganzzahligen, positiven Konstanten angehängt wird. Diese Konstante ist der Botschaftsbezeichner.

```
type
  TMyButton = class(TButton)
    procedure WMButton1Down(var Message); message WM_LBUTTONDOWN;
    ...
  end;
```

Zusätzlich muß die Methode eine Prozedurmethode sein, die über genau einen var-Parameter beliebigen Typs verfügt. Auch ein untypisierter var-Parameter ist erlaubt, so daß die Übergabe beliebiger Typen möglich wird. Wichtig für das Funktionieren des Botschaftsmechanismus ist nur, daß die ersten vier Byte des var-Parameters die Nummer der Botschaft enthalten. Die darauf folgenden Daten können vom Programmierer frei bestimmt werden.

```
type
  MeineMsg = record
    Msg: Cardinal;   // die ersten 4 Byte für Botschafts-Nummer
    Text: String;    // Beispiel für einen Botschafts-Inhalt
  end;


A = class
  public
    procedure Botschaftbehandlung1(var M: MeineMsg); message 1;
  end;

procedure A.Botschaftbehandlung1(var M: MeineMsg);
begin
  writeln('A: Botschaft ', M.Text, ' erhalten.');
end;

var
  ObjA: A;
  Nachricht: MeineMsg;
begin
  ObjA:= A.Create;

  Nachricht.Msg:= 1;            // NachrichtenNummer eintragen
  Nachricht.Text:= 'Guten Tag'; // NachrichtenInhalt eintragen
  ObjA.Dispatch(Nachricht);     // Nachricht verschicken

  ObjA.Free;
end.
```

Programm-Ausgabe: A: Botschaft Guten Tag erhalten.

In diesem Beispiel schickt Objekt A eine Botschaft an sich selbst. Das ist nicht besonders sinnvoll, demonstriert aber den Mechanismus des Versendens und Empfangens von Botschaften. Durch den Aufruf von Dispatch wird eine Botschaft verschickt. Der Empfänger der Botschaft ist klar bestimmt. Es ist aber für den Sender unerheblich, von welchem Klassen-Typ der Empfänger ist; der Empfänger kann irgendein beliebiges Objekt sein.

Die Prozedur Dispatch, die in allen Objekten bekannt ist, erledigt auch das Auffinden der zur Botschaft passenden Methode und ruft diese dann auf. Dazu sucht Dispatch zuerst in der DMT der tatsächlichen Klasse der Instanz nach einer Methode mit der in der Botschaft enthaltenen Nummer. Ist dort keine solche Methode enthalten, wird, wie bei allen dynamischen Methoden, die Tabelle des Vorfahren der Klasse durchsucht, dann die von dessen Vorfahren, und so weiter. In welchem Schutzabschnitt einer Klasse (*private, protected, public*) eine Botschaftsbehandlungs-Methode definiert wurde, ist für den Empfang von Botschaften unerheblich.

Es ist ein entscheidendes Merkmal der Botschaftsbehandlungs-Methoden, daß für den Aufruf einer botschaftsverarbeitenden Methode über Dispatch nur deren Nummer entscheidend ist. Der Name der Methode ist völlig irrelevant (im Beispiel lautet er Botschaftbehandlung1). Es ist insbesondere möglich, eine botschaftsverarbeitende Methode durch eine Methode völlig anderen Namens zu überschreiben, solange nur die Nummern übereinstimmen.

Enthält beim Aufruf von Dispatch der gesamte in Frage kommende Zweig der Klassenhierarchie keine Methode mit der angegebenen Nummer, dann wird eine virtuelle Methode *DefaultHandler* aufgerufen, die in der Basisklasse TObject wie folgt deklariert ist:

```
procedure DefaultHandler(var Message); virtual;
```

Ein Aufruf dieser Methode bewirkt nichts, weil der Funktionsrumpf der Methode leer ist. Es ist aber möglich, *DefaultHandler* zu überschreiben, um ein anderes Standardverhalten bei unbehandelten Nachrichten zu implementieren, zum Beispiel die Ausgabe einer Fehlermeldung. Innerhalb der Klassenbibliothek VCL ruft die Methode *DefaultHandler* für Fenster-Klassen die Windows-API-Funktion

DefWindowProc auf, die der Standardbehandlung von Windows-Botschaften dient.

Man kann für das Versenden von Botschaften eine alternative Routine schreiben, die sich mehr an die Syntax der Windows-Funktion SendMessage anlehnt:

```
function SendMsg(ZielObject: TObject; var Message): Boolean;
begin
  if ZielObject <> nil then begin
    ZielObject.Dispatch(Message);
    SendMsg:= True;
  end else
    SendMsg:= False;
end;
```

Im obigen Beispiel-Programm könnte dann das Versenden einer Botschaft statt durch

```
   ObjA.Dispatch(Nachricht);
```

durch einen Aufruf

```
   SendMsg(ObjA, Nachricht);
```

erfolgen.

Auch innerhalb von Botschaftsbehandlungs-Methoden kann die ererbte Botschaftsbehandlungs-Methode der Vorfahr-Klasse mit Hilfe einer inherited-Anweisung aufgerufen werden. Der var-Parameter wird beim inherited-Aufruf automatisch als Parameter übergeben, ohne daß er explizit angegeben werden müßte. Botschaftsbehandlungs-Methoden unterstützen auf diese Weise noch besser dynamische Konzepte. Ein Objekt sendet eine Botschaft und der Empfänger entscheidet selbst, was er mit der eingehenden Nachricht macht: ob er sie überhaupt auswertet, ob er die Nachricht anderen mitteilt, ob er dem Sender darauf antworten will usw. Ob eine Botschaft behandelt werden soll (vom Objekt "verstanden wird"), wird allein dadurch bestimmt, ob eine Botschaftsbehandlungs-Methode mit der Botschaftsnummer definiert wurde. In jedem Fall steht immer eine ererbte Implementierung zur Verfügung.

Die durch Dispatch aufgerufene Methode ähnelt im weitesten Sinne einer Software-Interrupt-Behandlungsroutine. Der "Interrupt" wird durch das Verschicken einer Nachricht mittels Dispatch ausgelöst. Es findet also insbesondere *kein* ressource-fressendes Polling statt, bei dem ein Objekt in einer Endlosschleife prüfen würde, ob irgendwelche Botschaften eingegangen sind, um dann in einer bestimmten Weise darauf zu reagieren.

Das Schlüsselwort "**static**" hat in C++ auch im Zusammenhang mit Klassen-Feldern und Methoden besondere Bedeutung. Ein Feld, das in einer Klasse mit static deklariert wurde, wird von allen Variablen dieser Klasse (von allen Objekten) gemeinsam genutzt. Es wird nur einmal gespeichert und ist deswegen nicht von einer bestimmten Instanz abhängig. Static-Felder müssen explizit deklariert und im globalen (dateiweiten) Block definiert werden. Sie wirken im Kontext der zugeordneten Klasse wie globale Variable.

```
class A{
public:
  static int x;
  ...
};

int A::x = 8;               // explizite Definition nötig

void main()
```

```
{
  A ObjA1;
  A ObjA2;

  ObjA1.x = 4;
  printf("%d\n", A::x);     // Ausgabe 4
  printf("%d\n", ObjA2.x);  // Ausgabe 4
}
```

Object Pascal kennt keine static-Felder. Die einzige (nicht sehr elegante) Möglichkeit, static-Felder nachzuahmen besteht darin, diese durch globale Variable zu ersetzen. Nach Möglichkeit sollten sie im Implementation-Abschnitt einer Unit definiert werden, so daß sie nicht im gesamten Programm sichtbar sind.

Beide Sprachen können Methoden definieren, die bereits vor einer Klassen-Instanzierung aufgerufen werden können. C++ nennt diese **static-Methoden** und Object Pascal bezeichnet sie als **Klassenmethoden** (gekennzeichnet durch das Wort class). Innerhalb solcher Methoden kann nicht auf Felder, Properties und normale Methoden zugegriffen werden.

VC++	Object Pascal
`class A{` `public:` `  static void Print(void);` `};` `void A::Print(void)` `{` `  printf("Print!\n");` `};` `void main()` `{` `  A::Print();` `  // Aufruf als Klassenmethode` `  A* ObjA = new A;` `  ObjA->Print();` `  // Aufruf als Objektmeth.` `}`	`type` `  A = class` `  public` `  class procedure Print;` `  end;` `class procedure A.Print;` `begin` `  writeln('Print!');` `end;` `var ObjA: A;` `begin` `  A.Print;` `  // Aufruf als Klassenmethode` `  ObjA:= A.Create;` `  ObjA.Print;` `  // Aufruf als Objektmeth.` `end.`

Syntaktischer Unterschied: Einer static-Methode darf in C++ bei der Methodenimplementierung nicht ein zweites Mal das Schlüsselwort static vorangestellt werden. In Object Pascal dagegen muß das Schlüsselwort class auch vor der Methodenimplementierung ein zweites Mal erscheinen. Static-/Klassenmethoden werden auch in den Klassenbibliotheken eingesetzt, um z.B. neue Objektinstanzen anzulegen:

VC++	Object Pascal
Im Makro DECLARE_DYNCREATE: `static CObject* CreateObject();`	In der Klassendefinition zu TObject: `class function NewInstance:` `  TObject; virtual;`

Neben static-Feldern und Methoden kennt C++ noch **const-Felder und Methoden**. Bei const-Methoden wird das Schlüsselwort const zwischen Parameterliste und Funktionsrumpf plaziert und gibt damit an, daß durch diese Methode keine Felderinhalte des Objekts modifiziert werden.

```
class A{
public:
  int i;
  void Print(void) const;
};

void A::Print(void) const
{
  printf("Print!\n");
  i = 8;
  // Fehler, da Felder nicht verändert werden dürfen
};
```

Der Compiler prüft allerdings nicht wirklich, ob Felder unverändert bleiben. Er meldet beim Übersetzen einen Fehler, wenn er einen Ausdruck erkennt, der die Möglichkeit zur Veränderung eines Feldes bietet. So dürfen Felder z.B. generell nicht als left-value Ausdrücke auftreten. Der Compiler bemängelt deswegen eine Zuweisung der Art

```
i = i;
```

die eigentlich gar keine Feldänderung zur Folge hat.

Const-Felder stellen Felder in Objekten dar, die bei der Objekt-Initialisierung einen festen Wert zugewiesen bekommen und dann nicht mehr verändert werden dürfen. Die Initialisierung muß in einer kommaseparierten Konstruktor-Initialisierungsliste erfolgen. Auch Referenzen müssen über eine Konstruktor-Initialisierungsliste initialisiert werden. Referenz steht hier wieder im Zusammenhang mit dem C++ Operator, durch den ein Alias für eine Variable erzeugt wird. Diese C++ Referenz hat nichts direkt mit dem "Referenz-Modell" von Object Pascal zu tun.

Initialisierung eines const-Feldes in einer C++ Klasse	Initialisierung eines Referenz-Feldes in einer C++ Klasse
`class A{` `public:` `  const int i;` `  A(void);  // Konstruktor` `  void Print(void);` `};` `A::A(void) : i(3)  // i = 3` `{` `};` `void A::Print(void)` `{` `  printf("%d", i);` `};` `void main()` `{`	`class A{` `public:` `  int i;` `  int& k;  // Referenz` `  A(void);  // Konstruktor` `  void Print(void);` `};` `A::A(void) : k(i)  // k = i` `{` `  i = 6;` `};` `void A::Print(void)` `{` `  printf("%d", k);` `};` `void main()` `{`

```
  A ObjA;
  ObjA.Print();
}
```

Ausgabe: 3

```
  A ObjA;
  ObjA.Print();
}
```

Ausgabe: 6

Einfache Funktionen können in C++ mit dem Zusatz **inline** deklariert werden (vgl. auch Kapitel 2.3.5). Das gilt auch für statische und virtuelle Methoden in Klassen. Steht der genaue Typ eines Objekts bereits zur Compilierzeit fest, so wird ein virtueller Funktionsaufruf nicht über die VMT codiert, sondern vom Compiler als inline-Funktion expandiert.

```
class A{
public:
  virtual void Print(void);
};

class B : public A{
public:
  virtual void Print(void);
};

inline void A::Print(void)
{
  printf("Print von A");
};

inline void B::Print(void)
{
  printf("Print von B");
};

void main()
{
  B Obj;
  Obj.Print();
}
```

Der tatsächliche Objekttyp steht aber, zumindest aus der Sicht des Compilers, bei dynamisch instanzierten Objekten niemals genau fest. Virtuelle inline-Methoden müssen deswegen meist doch über die VMT codiert werden, so daß ein Geschwindigkeitsvorteil nur bei Aufrufen (kleiner) statischer inline-Funktionen als sicher gelten kann.

Abstrakte Methoden werden in Basisklassen als virtuelle oder dynamische Methoden definiert. In C++ werden sie auch "rein virtuelle Funktionen" (pure virtual functions) genannt. Abstrakte Methoden werden erst in abgeleiteten Klassen implementiert, sind aber bereits in der Basisklasse bekannt und können so von anderen Methoden aufgerufen werden. Für abstrakte Methoden wird keine Adresse in die VMT eingetragen; der VMT-Eintrag erhält den Wert NULL bzw. nil.

VC++	Object Pascal

```
class A{
public:
  // abstrakte Methode
  virtual void Print(void) = 0;
};



class B : public A{
public:
  virtual void Print(void);
};

void B::Print(void)
{
  printf("Born in the USA");
};
```

```
type
  A = class
  public
    // abstrakte Methode
    procedure Print;
              virtual; abstract;
  end;

  B = class(A)
  public
    procedure Print; override;
  end;

procedure B.Print;
begin
  writeln('Born in the USA');
end;
```

Eine Klasse, die mindestens eine abstrakte Methode besitzt, wird abstrakte Basisklasse genannt. In Visual C++ kann eine Klasse erst dann instanziert werden, wenn alle abstrakten Methoden überschrieben und implementiert wurden. Object Pascal läßt die Instanzierung von Klassen zu, in denen abstrakte Methoden noch nicht implementiert wurden, gibt aber beim Übersetzen eine entsprechende Warnmeldung aus. Wird eine abstrakte Methode eines Objekts aufgerufen, bei dem die Methode nicht überschreiben wurde, so wird zur Laufzeit eine Exception ausgelöst.

Auf Felder und Methoden einer Klasse kann auch mittels eines Zeigers verwiesen werden (Klassenelement-Zeiger). Zeiger auf Felder (Datenelement-Zeiger) wurden bereits in der Tabelle der Operatoren eingeführt und Zeiger auf Methoden im Abschnitt "Prozedurale Datentypen" besprochen.

2.3.6.5 Konstruktoren und Destruktoren

Konstruktoren sind spezielle Methoden, die das Erzeugen und Initialisieren von Objekten steuern. In C++ müssen Konstruktoren denselben Namen tragen, den die Klasse besitzt. In Object Pascal dagegen können die Namen für Konstruktoren beliebig gewählt werden. Daß eine Methode ein Konstruktor ist, wird hier durch das Schlüsselwort "constructor" kenntlich gemacht, der den Vorsatz "procedure" ersetzt.

Destruktoren bilden das Gegenstück zu den Konstruktoren und definieren die Aktionen, die mit dem Entfernen eines Objekts verbunden sind. In C++ muß ein Destruktor denselben Namen wie die Klasse, jedoch mit vorgesetzter Tilde ~ , tragen. Destruktoren in Object Pascal werden durch das Schlüsselwort "destructor" kenntlich gemacht und unterliegen keinen Namensbeschränkungen. Pro Klasse ist in C++ die Definition eines Destruktors, in Object Pascal die Definition beliebig vieler Destruktoren gestattet. Destruktor-Methodenaufrufe können in beiden Sprachen keinen Wert als Ergebnis zurückliefern; Funktions-Parameter sind in Object Pascal zulässig und in C++ nicht zulässig. Anders als in C++ muß in Object Pascal bei vererbten Klassen der Destruktor der Vorgänger-Klasse explizit aufgerufen werden (inherited), sofern ein solcher Aufruf gewünscht wird.

VC++	Object Pascal

```
class A{
private:
  int i;
public:
  A(int x);  // Konstruktor
  ~A();      // Destruktor
  void Print(void);
};

A::A(int x)  // Konstruktor
{
  i = x;
  // möglich wäre auch
  // this->i = x;
};

A::~A(void)  // Destruktor
{
  printf("Tschuess\n");
};

void A::Print(void)
{
  printf("%d\n", i);
};

void main()
{
  A* ObjA;
  ObjA = new A(5);
  ObjA->Print();
  delete ObjA;
}
```

```
type A = class
private
  i: Integer;
public
  constructor Start(x: Integer);
  destructor Fertig;
  procedure Print;
end;

constructor A.Start(x:Integer);
begin
  i:= x;
  // möglich wäre auch
  // Self.i:= x;
end;

destructor A.Fertig;
begin
  writeln('Tschuess');
end;

procedure A.Print;
begin
  writeln(i);
end;

var
  ObjA: A;
begin
  ObjA:= A.Start(5);
  ObjA.Print;
  ObjA.Fertig;
end.
```

Bei einer statischen Objekt-Variablen werden in C++ Konstruktor und Destruktor automatisch aufgerufen, wenn der Gültigkeitsbereich der Variablen betreten bzw. verlassen wird. Am Beispiel oben wird ein Unterschied bei der dynamischen Objektinstanzierung zwischen beiden Sprachen deutlich. Ein Objekt wird in C++ durch die Operatoren new und delete erzeugt, ohne daß die Konstruktor- / Destruktor-Methoden explizit aufgerufen werden. In Object Pascal dagegen werden Konstruktor und Destruktor namentlich aufgerufen. Der Konstruktor führt hier als erstes die Speicherreservierung und der Destruktor nach Abarbeitung der benutzerdefinierten Aktionen die abschließende Speicherfreigabe durch. Als Ergebnis eines Konstruktor-Aufrufs wird in Object Pascal (im Erfolgsfall) eine Zeiger-Referenz auf das neu angelegte Objekt zurückgeliefert.

Bei einem Konstruktor-Aufruf werden in Object Pascal nach der Speicherreservierung für das Objekt alle Felder mit 0 und alle Zeiger mit nil initialisiert. C++ führt keine automatische Felderinitialisierung durch. Nur Felder, die explizit in der Initialisierungsliste des Konstruktors aufgeführt werden, bekommen den dort angegebenen Wert zugewiesen (wie im Beispiel mit const-Objektfeld weiter oben).

Wird in C++ bei einer Klassendefinition kein Konstruktor angegeben, so legt der Compiler implizit einen Default-Konstruktor an. In Object Pascal stehen allen Klassen der statische Default-Konstruktor *Create* und der virtuelle Default-Destruktor *Destroy* (bzw. eine statische Methode *Free*, die ihrerseits *Destroy* aufruft) zur Verfügung.

Konstruktoren können in C++, wie das bei allen normalen Funktionen und Methoden möglich ist, überladen werden, d.h., es können innerhalb eines Objekts weitere Konstruktoren mit veränderter Parameterliste definiert werden.

Ein besonderer Konstruktor ist jener, dessen Parameter-Typ eine Referenz auf ein Objekt der eigenen Klasse darstellt. Er wird Kopier-Konstruktor (Copy-Constructor) genannt und in folgenden Fällen aufgerufen:

1. Bei der Initialisierung eines Klassenobjekts durch ein anderes Objekt
2. Beim Aufruf einer Funktion und der Übergabe eines Objekts als aktuellem Parameter
3. Bei der Ergebnisrückgabe einer Funktion (Objekt ist Rückgabewert)

Wenn kein Kopie-Konstruktor definiert wird, so wird in den genannten drei Fällen in der Objekt-Kopie jedem Feld der Wert des entsprechenden Quell-Objekt-Feldes zugewiesen (memberweises Kopieren/Initialisieren). Probleme können aber dann auftreten, wenn einige Objekt-Felder Zeiger sind. In diesem Fall wird nur der Zeiger (=die Adresse) selbst kopiert, nicht jedoch der Inhalt, auf den er verweist. Das Problem soll an einem Beispiel verdeutlicht werden. Ohne Kopie-Konstruktor (linke Spalte) verweist die Zeigervariable *Text* im kopierten Objekt auf den (dynamischen) Speicherbereich des Quellobjekts. Durch den Kopie-Konstruktor (rechte Spalte) wird für das kopierte Objekt ein neuer dynamischer Speicherbereich alloziert. Anschließend wird durch einen StringCopy-Aufruf explizit der Text aus dem dynamischen Speicherbereich des Quell-Objekts in den neuen dynamischen Speicher des Ziel-Objekts kopiert.

C++: Ohne Kopie-Konstruktor

```
class A{
public:
  // Standard-Konstruktor
  A();


  void Print(void);
  char* Text;
};


// Standard-Konstruktor
A::A()
{
  printf("Std.-Konstruktor");
  Text = new char[50];
  strcpy(Text, "Hallo Welt");
};









void A::Print(void)
{
  printf("%s\n", Text);
};

void main()
{
  A Obj1;
  Obj1.Print();

  A Obj2 = Obj1;      // !!
  Obj1.Text[3] = '\0';
```

C++: Mit Kopie-Konstruktor

```
class A{
public:
  // Standard-Konstruktor
  A();
  // Kopie-Konstruktor
  A(A& QuellObj);
  void Print(void);
  char* Text;
};

// Standard-Konstruktor
A::A()
{
  printf("Std.-Konstruktor");
  Text = new char[50];
  strcpy(Text, "Hallo Welt");
};

// Kopie-Konstruktor
A::A(A& QuellObj)
{
  printf("Kopie-Konstruktor");
  Text = new char[50];
  strcpy(Text, QuellObj.Text);
};

void A::Print(void)
{
  printf("%s\n", Text);
};

void main()
{
  A Obj1;
  Obj1.Print();

  A Obj2 = Obj1;        // !!
  Obj1.Text[3] = '\0';

  Obj2.Print();
}
```

```
  Obj2.Print();
}
```

Ausgabe:

Std.-Konstruktor

Hallo Welt

Hal

Ausgabe:

Std.-Konstruktor

Hallo Welt

Kopie-Konstruktor

Hallo Welt

Object Pascal kennt keinen Kopie-Konstruktor. Als Ersatz wurde in Delphis Klassenbibliothek VCL eine Methode "Assign" deklariert, die durchgängig in fast allen Klassen existiert:

```
procedure Assign(Source: TPersistent); virtual;
```

Sie weist ein Objekt einem anderen zu. Der allgemeine Aufruf lautet:

```
Destination.Assign(Source);
```

Dem Objekt *Destination* wird dadurch der Inhalt von Source zugewiesen. Das klassenspezifische Überschreiben der Methode Assign (bzw. der mit Assign verbundenen Methode AssignTo) entspricht somit der Definition eines klassenspezifischen Kopie-Konstruktors in C++. Allerdings wird eine Assign-Methode, anders als ein Kopie-Konstruktor, niemals automatisch aufgerufen.

Destruktoren können in C++ und Object Pascal virtuell sein. Ein polymorphes Überschreiben wird notwendig, wenn klassenspezifische Freigabeaktionen ausgeführt werden müssen. Virtuelle Konstruktoren sind nur in Object Pascal zulässig und werden im Zusammenhang mit Metaklassen verwendet.

2.3.6.6 Metaklassen

Metaklassen sind in C++ nicht definiert; neben Object Pascal unterstützen z.B. die Sprachen Smalltalk und CLOS (Common Lisp Object System) Metaklassen. Eine Metaklasse ist ein Datentyp, dessen Werte Klassen sind. Während eine Klasse einen Typ definiert, der Objektreferenzen aufnehmen kann, kann eine Variable vom Typ einer Metaklasse Referenzen auf Klassen enthalten. Deshalb werden Metaklassen auch "Klassenreferenztypen" genannt.

VC++	Object Pascal
Klasse => Objekt zu lesen als: Instanzen von Klassen sind Objekte	Metaklasse => Klasse => Objekt zu lesen als: Instanzen von Metaklassen sind Klassen Instanzen von Klassen sind Objekte

Der Wertebereich einer Metaklasse umfaßt die Klasse, für die sie deklariert wurde und alle deren Nachfahren, insbesondere auch jene, die

erst zu einem späteren Zeitpunkt deklariert werden. Metaklassen können überall dort im Programm eingesetzt werden, wo auch direkt mit Klassen operiert wird:

- Aufruf von Klassenmethoden
- Prüfen und Wandeln von Typen mit den Operatoren "is" und "as"
- polymorphes Konstruieren von Objekten (mit Hilfe eines virtuellen Konstruktors)

In allen Fällen muß die tatsächliche Klasse nicht bereits zur Übersetzungszeit feststehen, sondern es wird zur Laufzeit die Klasse benutzt, die als Wert der Variablen vorgefunden wird.

```
type
  TFigur = class
    constructor Create; virtual; // virtueller Konstruktor
     ...
  end;

  TRechteck = class(TFigur)
    constructor Create; override;
     ...
  end;

  TKreis = class(TFigur)
    constructor Create; override;
    ...
  end;

  TFigurClass = class of Tfigur; // MetaKlasse

var
  // Klassen-Referenz Variable
  ClassRef: TFigurClass;
  // Objekt-Instanz Variable
  Figur: TFigur;
begin
  // Klassen-Referenz zeigt auf Klasse Kreis
  ClassRef:= TKreis;
  // polymorphes Erstellen eines Objekts vom Typ Kreis
  Figur:= ClassRef.Create;
    ...
  // Objekt wieder freigeben
  Figur.Free;

  // Klassen-Referenz zeigt jetzt auf  Klasse Rechteck
  ClassRef:= TRechteck;
  // polymorphes Erstellen eines Rechteck - Objekts
  Figur:= ClassRef.Create;
    ...
  // Objekt wieder freigeben
  Figur.Free;
end.
```

Man muß beim polymorphen Konstruieren von Objekten beachten, daß die Konstruktoren der zu benutzenden Objekte als virtuell gekennzeichnet sind. Wenn das nicht der Fall ist, findet die Bindung bereits zur Übersetzungszeit statt. Es wird dann stets eine Instanz der Klasse erzeugt, zu der die Metaklasse der Variablen deklariert wurde (im Beispiel wäre das TFigur).

Stroustrup bewertet in [6] Metaklassen als "sicher recht leistungsfähig", möchte sie aber andererseits nicht in C++ implementieren: "Ich verwehrte mich damals nur gegen den Gedanken, jeden C++ Programmierer mit diesen Mechanismen zu belasten zu müssen ...".

2.3.6.7 Friends

Durch "Friend"-Deklarationen lassen sich in C++ Zugriffseinschränkungen einer Klasse für ausgewählte Funktionen oder Klassen deaktivieren. In Object Pascal existiert ein implizites Friend-Modell, bei dem automatisch alle Klassen "Freunde" sind, die innerhalb einer Unit implementiert sind.

Durch die expliziten und impliziten Freundschaften wird die Klassen-Kapselung aufgehoben und somit bewußt ein elementarer Grundsatz objektorientierter Entwicklung unterlaufen. Andererseits ermöglichen es Freundschaften oftmals, daß Klassen klein und einfach gehalten werden können.

VC++	Object Pascal
`class A{` `friend class B;` `// Klasse B ist` `// Freund von A` `private:` `  int TopSecret;` `};` `class B{` `public:` `  void WeiseZu(A* ObjA);` `};` `void B::WeiseZu(A* ObjA)` `{` `   ObjA->TopSecret = 7;` `   // B hat Zugriff auf` `   // private Daten von A` `};` `void main()` `{` `  A* Obj1 = new A;` `  B* Obj2 = new B;` `  Obj2->WeiseZu(Obj1);` `  delete Obj1;` `  delete Obj2;` `}`	`type` `A = class` `private` `  TopSecret: Integer;` `end;` `B = class` `public` `  procedure WeiseZu(ObjA: A);` `end;` `procedure B.WeiseZu(ObjA: A);` `begin` `  ObjA.TopSecret := 7;` `   // B hat Zugriff auf` `   // private Daten von A` `end;` `var Obj1: A; Obj2: B;` `begin` `  Obj1:= A.Create;` `  Obj2:= B.Create;` `  Obj2.WeiseZu(Obj1);` `  Obj1.Free;` `  Obj2.Free;` `end.`

```
class A{
friend void GlobalProc(A* ObjA);
private:
  int TopSecret;
};

// globale Funktion
void GlobalProc(A* ObjA)
{
  ObjA->TopSecret = 5;
};

void main()
{
  A Obj;
  GlobalProc(&Obj);
}
```

```
keine Freundschaften
    für einzelne
     Funktionen
    und Methoden
```

Im letzten Beispiel erhält eine globale Funktion *GlobalProc* die vollständige Zugriffsberechtigung auf alle geschützten Elemente in der Klasse *A* (hier auf die private Variable TopSecret).

Um zu verhindern, daß unbeabsichtigt auf geschützte Klassenelemente anderer Klassen zugegriffen wird, müssen in Object Pascal einzelne Klassen in separaten Units untergebracht werden.

2.3.6.8 Überladen von Operatoren

Ein Spezialfall des Überladens von Funktionen in C++ stellt das Überladen von Operatoren (operator overloading) dar. Vordefinierte Operatoren erlangen, angewandt auf Klassen, per Definition eine neue Bedeutung. Unter Verwendung des Schlüsselworts "operator" lautet die generelle Syntax:

```
Ergebnistyp operator Operator-Zeichen ( ... , ... );
```

Priorität und Assoziativität des überladenen Operators bleiben erhalten. Die zu Beginn des Kapitels eingeführte Klasse *Komplex* wird im Beispiel um den Operator "+" erweitert:

```
class Komplex
{
  public:
    void Zuweisung(double r,
                   double i);
    void Ausgabe();
    Komplex operator+(Komplex z);
  private:
    double Real;
    double Imag;
};

  ...

Komplex Komplex::operator+(Komplex z)
{
```

```
  z.Zuweisung(Real + z.Real, Imag + z.Imag);
  return z;
};

void main()
{
  Komplex a, b, c;

  a.Zuweisung(3, 2);
  a.Ausgabe();       // 3+i*2
  b.Zuweisung(10, 5);
  b.Ausgabe();       // 10+i*5

  c = a + b;         // überladener Operator + wird benutzt
  c.Ausgabe();       // 13+i*7
}
```

Durch das Konzept der überladenen Operatoren wird einem Programmierer die Möglichkeit gegeben, eine gewohnte, konventionelle Notation zur Manipulation von Klassenobjekten anzubieten.

Da Object Pascal diese Technik nicht beherrscht, müssen neu zu definierende Funktionen die Aufgabe der überladenen Operatoren übernehmen:

```
Komplex = class
public
   ...
  procedure Addiere(z: Komplex);
  procedure Assign(z: Komplex);
   ...
end;
...

begin
  ...
  a.Addiere(b);
  c.Assign(a);
end;
```

2.3.6.9 Klassen-Schablonen

Wie bereits im Kapitel "2.3.5 Programmstruktur" erwähnt, stellen Schablonen eine wesentliche Erweiterung der Sprache C++ dar. In Object Pascal existiert kein entsprechendes Gegenstück zu dieser Spracherweiterung.

Analog zu Funktions-Klassen werden Klassen-Schablonen zur Beschreibung von Schablonen für Klassen erstellt. Mit Hilfe des Schlüsselworts "template" wird eine Familie von Klassen definiert, die mit verschiedenen Typen arbeiten kann. Im Beispiel wird eine Template-Klasse definiert, die ein gewöhnliches Array mit 20 Elementen kapselt und Grenzüberprüfung beim Schreiben der Elemente durchführt. Der exakte Typ, den die 20 Elemente des Arrays annehmen sollen, wird bei der Definition der Template-Klasse jedoch offen gelassen. Er wird nicht näher spezifiziert und allgemein "**Typ**" genannt.

```
template <class Typ>
class TemplateKlasse{
public:
  void SetArray(Typ a, int b);
```

```
private:
  Typ EinArray[20];
};

template <class Typ>
void TemplateKlasse<Typ>::SetArray(Typ a, int b)
{
  if(( b >= 0 ) && (b <= 19))
    EinArray[b] = a;
}


void main()
{
  TemplateKlasse<double>* MeinArray1;
  MeinArray1 = new TemplateKlasse<double>;
  // erstes Element belegen
  MeinArray1->SetArray(14.783 , 0);
  // zweites Element belegen
  MeinArray1->SetArray( -3.66 , 1);
  delete MeinArray1;

  TemplateKlasse<char*>* MeinArray2;
  MeinArray2 = new TemplateKlasse<char*>;
  // erstes Element belegen
  MeinArray2->SetArray("Hallo Welt" , 0);
  // zweites Element belegen
  MeinArray2->SetArray("Aha" , 1);
  delete MeinArray2;
}
```

Zunächst wird ein Objekt *MeinArray1* angelegt, dessen 20 Elemente vom Typ Double sind. Im Anschluß daran wird ein Objekt *MeinArray2* angelegt, dessen 20 Elemente Zeiger auf Strings sind (Typ = char*). Bei der Definition und der Instanzierung eines Objekts, das durch eine Klassen-Schablone beschrieben wird, wird dazu explizit der konkrete Typ in spitzen Klammern angegeben. Erst wenn der Compiler auf die Bezeichnung einer Klassen-Schablone in Verbindung mit einem genauen Typ trifft, legt er die entsprechende Klassendefinition im globalen Gültigkeitsbereich des Programms an. Dabei wird der Parameter-Typ textuell durch den konkreten Typ-Namen (hier double bzw. char*) ersetzt.

Außer Typ-Parametern können bei der Schablonen-Definition auch Argumente angegeben werden, die keinen Typ repräsentieren:

```
template <class Typ, int i>
```

Im Beispiel oben könnte man dadurch z.B. die starre Fixierung auf 20 Elemente aufheben. Die exakte Array-Größe würde dann erst bei der Schablonen-Instanzierung durch den Parameter **i** festgelegt werden.

Schablonen eignen sich besonders zur Erstellung leistungsfähiger und kompakter Containerklassen. Durch den Einsatz von Schablonen kann oftmals auf die Nutzung undurchschaubarer und fehleranfälliger Präprozessor -"#define"- Sequenzen verzichtet werden. Die Klassenbibliothek von Visual C++ (MFC) nutzt intensiv sowohl Funktions- als auch Klassen-Schablonen.

2.3.6.10 Eigenschaften - Properties

Eigenschaften, im folgenden Properties genannt, stellen benannte Attribute eines Objekts dar. Properties sind nur in Object Pascal verfügbar.

Wirkung nach außen:

Properties sehen für den Endanwender eines Objekts wie Felder aus, da sie über einen Typ verfügen, wie Felder gelesen werden können und ihr Wert in Form einer Zuweisung verändert werden kann. Properties erweitern aber gewöhnliche Felder, da sie intern Methoden verkapseln, die den Wert des Feldes lesen oder schreiben. Properties können behilflich sein, die Komplexität eines Objekts vor dem Endanwender zu verbergen. Sie gestatten es, den Objektstatus durch Seiteneffekte zu verändern.

Um die Wirkungsweise und Nutzung zu verdeutlichen, soll ein Beispiel angeführt werden. Jedes Fenster-Objekt in Delphis Klassenbibliothek (VCL) besitzt ein Property mit dem Namen "Top". Dies ist ein Lese- und Schreib-Property, mit dem einerseits die aktuelle y-Koordinate der Fensterposition ermittelt werden kann, mit dem andererseits beim Zuweisen eines Wertes an das Property das Fenster an die angegebene y-Koordinate bewegt wird. Ein Objekt-Anwender muß zum Verschieben des Fensters keine Funktionen aufrufen.

```
// Lesen und Ausgabe des Propertys
ShowMessage(IntToStr(MeinFenster.Top));

// Schreiben des Propertys
MeinFenster.Top := 20;
// das Fenster wird zur Position 20 verschoben
```

Ein weiteres Beispiel stellt das Property "Caption" dar. Durch das Lesen der Property erhält man den aktuellen Fenster-Text (Überschrift) und durch das Zuweisen an das Property wird der Fenster-Text gesetzt.

```
ShowMessage(MeinFenster.Caption);

MeinFenster.Caption := 'Hallo Welt';
```

Der Programmierer muß sich weder damit beschäftigen, wie die Fensterposition / der Fenster-Text ermittelt werden kann, noch muß er wissen, wie man in Windows ein Fenster verschiebt oder wie die Überschrift eines Fensters gesetzt wird.

Realisierung im Inneren:

Es ist möglich, sowohl den lesenden als auch den schreibenden Zugriff auf ein Property einzeln zu gestatten bzw. zu unterbinden. Drei verschiedene Properties erhalten im Beispiel unterschiedliche Zugriffsrechte:

```
MeineKlasse = class
  FText: String; // normales Feld in d. Klasse

  // Lese- und Schreib-Zugriff
  property Text: String read FText write FText;
  // nur Lese-Zugriff
  property Lies_Text: String read FText;
  // nur Schreib-Zugriff
  property Schreib_Text: String write FText;
end;

  ...
```

```
// Property Text gestattet Schreib- und
Text := 'Hallo';
// Lese-Zugriff
ShowMessage(Text);

Schreib_Text := 'Welt';
ShowMessage(Lies_Text);
// Fehler, da nur Lese-Zugriffe gestattet sind
Lies_Text := 'Guten Tag';
// Fehler, da nur Schreib-Zugriffe gestattet sind
ShowMessage(Schreib_Text);
```

Allen drei Properties liegt ein Feld (FText) in der Klasse zugrunde, aus dem gelesen bzw. in das geschrieben wird. Wenn einem Property zum Lesen und Schreiben dasselbe Feld zugrunde liegt (wie bei Text), dann ist dieses Property eigentlich überflüssig, weil es nur den Direktzugriff auf das Feld in der Klasse abfängt.

Sinnvollere Properties erhält man, wenn einer oder beiden Zugriffsarten (**read** / **write**) eine Methode zugrunde liegt. In dem Fall wird beim Zugriff auf das Property automatisch die angegebene Methode aufgerufen, so daß durch das Auslesen bzw. das Zuweisen beliebig komplexe Operationen in Gang gesetzt werden können.

```
MeineKlasse = class
  // normales Feld in d. Klasse
  FText: String;
  // Methode
  procedure SetText(Value: String);
  property Text: String read FText write SetText;
end;

procedure MeineKlasse.SetText(Value: String);
begin
  if Value <> FText then begin
    // Wandlung in Großbuchstaben
    FText:= AnsiUpperCase(Value);
    // Fenstertext mit Windows-API - Funktion setzen
    SetWindowText(MyWin.Handle, PChar(FText));
  end;
end;
  ...

var MeinObj: MeineKlasse;
begin
  MeinObj := MeineKlasse.Create;
  MeinObj.Text := 'Hallo Welt';
end;
```

Als Ergebnis der Zuweisung an Property Text wird implizit die Methode *SetText* aufgerufen, die ihrerseits dem Feld FText den Wert "HALLO WELT" zuweist. Derselbe Text erscheint zudem auch in der Überschrift-Zeile eines Fensters.

Die einem Property zugrundeliegenden Methoden müssen immer dieselbe Syntax aufweisen:

Lese-Methode	`function MethodenName: PropertyTyp;`
Schreib-Methode	`procedure MethodenName(Value: PropertyTyp);`

Borland empfiehlt für Lese- und Schreib-Methoden eine einheitliche Namensgebung zu gebrauchen. Der Name einer Lese-Methode sollte demnach stets mit dem Vorsatz "Get" beginnen, dem der Property-Name folgt (im Beispiel wäre das *GetText*). Der Name einer Schreib-Methode sollte stets mit dem Vorsatz "Set" beginnen, dem der Property-Name folgt (wie im Beispiel zu sehen: *SetText*). Das Feld, auf dem ein Property beruht, sollte immer mit dem Buchstaben "F" beginnen, dem der Property-Name folgt (im Beispiel: *FText*).

Beim Übersetzen ersetzt der Compiler jede Auslese- und jede Zuweisungs-Anweisung eines Propertys durch das zugeordnete Feld bzw. durch einen Aufruf der zugeordneten Methode. Bei Methodenaufrufen, die Schreibzugriffe ersetzen, setzt er dabei den Wert des Propertys als aktuellen Parameter der Set-Methode ein.

Die Implementierungs-Methoden von Properties können virtuell sein. In abgeleiteten Objekten können die Methoden polymorph überschrieben werden. Das Lesen und Zuweisen der gleichen Properties kann dann im Basis-Objekt ganz andere Aktionen als im abgeleiteten Objekt zur Folge haben.

Neben einfachen Properties existieren noch indizierte Properties, die sich dem Benutzer wie ein Array darstellen. Schreib- und Lese-Methoden für indizierte Arrays benötigen einen weiteren Parameter, der den Index für den Zugriff festlegt.

Für alle Properties, die im published-Abschnitt einer Klasse deklariert sind, erzeugt der Compiler spezielle Informationen, die für die Zusammenarbeit mit Delphis "Objektinspektor" von Bedeutung sind. Diese veröffentlichten Properties bilden die Basis für das "visuelle Programmieren". Beim visuellen Programmieren und Gestalten von Programmen können Objekte in einer Design-Phase auf sogenannten Formularen, den Fenstern der Anwendung, plaziert werden. Alle veröffentlichten Properties werden zur Design-Zeit von Delphi ausgewertet und im Objektinspektor zur Ansicht gebracht. Die Werte der Properties können im Objektinspektor manuell verändert werden, wobei Änderungen zur Design-Zeit genau dieselben automatischen Methodenaufrufe wie im endgültig übersetzten Programm auslösen. Durch dieses Verhalten kann der Programmierer sofort, noch bevor er das Programm neu übersetzt und gestartet hat erkennen, wie sich Eigenschaftsänderungen auswirken. Die Gestaltung der Programmoberfläche kann so aufgrund der Properties regelbasiert nach dem beim Menschen sehr beliebten Schema "Trial and Error" (Versuch und Irrtum) erfolgen. Der Entwickler setzt die Hintergrundfarbe eines Fensters auf rot, indem Property Color im Objektinspektor entsprechend geändert wird. Der implizite Aufruf der Funktion SetColor im betreffenden Fenster-Objekt stößt daraufhin selbständig ein Neuzeichnen des eigenen Fensterbereichs an; das Fenster wird rot dargestellt. Der Entwickler erkennt, daß ein roter Fensterhintergrund doch weniger gut anzusehen ist und kann sofort wieder Property Color auf Grau zurücksetzen.

Die Bezeichnung "visuelles Programmieren" geht auf diese visuellen Gestaltungsabläufe zurück. Durch einfache Maus-Klicks und das Ändern einiger Property-Werte können so erstaunlich schnell komplexe Anwendungen entstehen. Selbst das Erstellen von Datenbankprogrammen mit relationalen Daten-Beziehungen, Abfragen (Queries), Filtern usw. wird auf diese Weise zum Kinderspiel.

Allerdings muß beim Erstellen eigener Objekte, hier auch Komponenten genannt, beachtet werden, daß auf die Änderung aller veröffentlichten Property-Werte richtig bzw. überhaupt reagiert wird. Das kann beim Neu-Erstellen von Komponenten zunächst einigen Mehraufwand bedeuten, der sich aber schnell bezahlt macht. Jeder Endanwender von Objekten erwartet wie selbstverständlich, daß vorgenommene Änderungen sofort entsprechende, sichtbare Auswirkungen zur Folge haben.

Um auf komfortable Weise auch die Werte sehr komplexer, strukturierter Properties im Objektinspektor verändern zu können gestattet es Delphi, für solche Typen eigene "Property-Editoren" zu schreiben. Sie müssen bei der Programmierumgebung (IDE) registriert und angemeldet werden. Die Property-Editoren stellen teilweise komplette kleine Anwendungen, manchmal mit mehreren Dialogfenstern, dar. Sie werden vom Objektinspektor aufgerufen und dienen einzig und allein dem "visuellen Programmieren" und werden (bzw. sollten) nicht mit in das endgültig erstellte Programm aufgenommen (werden).

Die zunächst weniger spektakulär anmutende Spracherweiterung der Properties in Object Pascal kann die hinter diesem Konzept stehende Leistungsfähigkeit erst im Zusammenspiel mit der geschickt gestalteten Programmierumgebung Delphis zum Tragen bringen. Es stellt eine bedeutende Neuerung dar, daß bei der Objekt- und Komponenten-Entwicklung ein Großteil des erstellten Codes mehr oder weniger ausschließlich dafür geschrieben wird, die Gestaltung und Entwicklung von Programmen zu vereinfachen und zu beschleunigen. Properties bilden also nicht nur die Grundlage für "visuelles Programmieren" sondern sind auch der ausschlaggebende Grund für deutlich kürzere Entwicklungszeiten in Delphi gegenüber Visual C++ bei der Programmierung fensterbasierender, GUI-orientierter Anwendungen. GUI = Graphical User Interface (graphische Benutzeroberfläche)

2.3.6.11 Laufzeit-Typinformationen

Durch die Laufzeit-Typinformation (Run-Time Typ Information = RTTI) kann man zur Laufzeit eines Programms die Zugehörigkeit eines

Objekts zu einer bestimmten Klasse sicher ermitteln. Damit die Klassen-Zugehörigkeit bestimmt werden kann, muß der Compiler zusätzlichen Code im zu erstellenden Programm einfügen. In Visual C++ werden RTTI erst dann eingefügt, wenn das in den Projektoptionen explizit angegeben wurde. In Object Pascal stehen RTTI immer für alle Objekte zur Verfügung, da bereits die Basisklasse "TObject", von der alle anderen Objekte abgeleitet sind, Typinformationen zur Verfügung stellt.

In Visual C++ und Object Pascal stehen Operatoren und Funktionen zur Ermittlung und Auswertung der RTTI zur Verfügung. In der Tabelle der Operatoren wurden bereits der Objekttyp-Informationsoperator, der dynamische Konvertierungsoperator und die Möglichkeiten zur Objekttyp-Prüfung aufgeführt. C++ Programme müssen die Datei *TYPEINFO.H* einbinden, um die RTTI ermitteln und auswerten zu können. Der typeid()-Operator liefert als Ergebnis ein Objekt, das die Informationen über ein anderes Objekt enthält. Dieses Informations-Objekt ist in der genannten Header-Datei definiert. Im wesentlichen liefert das Informations-Objekt den Klassennamen und eine Methode "before", die das Informations-Objekt eines benachbarten Objekts liefert. Es besteht aber keine Beziehung zwischen dem so ermittelten Nachbar-Informations-Objekt und Vererbungsbeziehungen zwischen den Klassen.

Derartige Informationen kann Object Pascal liefern. Durch eine Methode "InheritsFrom" kann geprüft werden, ob ein Objekt von einer bestimmten Klasse abstammt. Die Ermittlung solcher Informationen wird ermöglicht, da Object Pascal die RTTI in einer Tabelle verwaltet. Diese RTTI-Tabelle enthält Informationen über die Klasse, deren Vorfahr-Klasse und alle veröffentlichten Felder und Methoden. Durch die Klassen-Methode "MethodAddress" und die Methode "FieldAddress" kann man prüfen, ob eine bestimmte Methode bzw. ein bestimmtes Feld in einem Objekt vorhanden ist. Man kann also zur Laufzeit Feld- und Methoden-Name nachschlagen (field name lookup / method name lookup). Die Lookup-Methoden liefern die Adresse der gefundenen Methode bzw. des gefundenen Felds zurück. Falls keine Methode / kein Feld unter dem genannten Namen existiert, liefern die Lookup-Methoden "nil" zurück.

Dynamische Typkonvertierungen und Metaklassen basieren auf den Run-Time Typ Informationen.

Zurück zum Inhaltsverzeichnis
Weiter in Kapitel 3

3. Übersicht über die bei Visual C++ und Delphi mitgelieferten Bibliotheken

3.1 Laufzeitbibliotheken - Run-Time Libraries

Bei der Programmierung in Visual C++ und Delphi kann man auf alle Funktionen und Strukturen des Window-APIs zugreifen. Dazu muß man in Visual C++ Programmen die Header-Datei *windows.h* und in Delphi-Programmen die Unit *Windows* einbinden. Um zusätzliche, erweiterte APIs nutzen zu können, ist manchmal die Einbindung weiterer Header-Dateien bzw. Units nötig (z.B. commctrl.h / Unit CommCtrl).

Eine andere wichtige Funktionen-Sammlung stellen die Run-Time Libraries (abgekürzt RTL) dar. In beiden Sprachen werden durch diese Bibliotheken oft benötigte Routinen zur Verfügung gestellt. Zu ihnen zählen z.B. Ein-/Ausgabe-Routinen, Routinen zur String-Bearbeitung, arithmetische Routinen, Routinen zur Typkonvertierung und Speicherverwaltung. Um die RTL-Routinen nutzen zu können, müssen ebenfalls die zugehörigen Header-Dateien bzw. Units eingebunden werden. Einzige Ausnahme stellt die Unit *System* in Object Pascal dar, die automatisch in allen Delphi-Programmen eingebunden wird.

Visual C++ stellt bei Routinen, die Strings als Funktions-Parameter benötigen, jeweils zwei funktional identische Versionen bereit: eine kann String-Parameter im ANSI-Stringformat entgegennehmen, die andere Version (mit einem "w" im Funktionsnamen) akzeptiert String-Parameter nur im Format des erweiterten Win32-Zeichensatzes "UNI-Code". Die meisten RTL-Routinen Delphis unterstützen nur ANSI-String-Parameter.

In der folgenden Tabelle werden einige öfter verwendete RTL-Routinen aufgeführt.

	Visual C++	Delphi
Ein-/Ausgabe-Routinen:		
öffne Datei	fopen	Reset, Rewrite
schließe Datei	fclose	CloseFile
zu bestimmter Position in Datei bewegen	fseek	Seek
prüfe, ob am Ende der Datei	feof	Eof
aktuelle Position in Datei ermitteln	ftell	FilePos
schreibe Variable in Datei	fwrite	Write
Ausgabe auf Standard-Ausgabe	printf	Write / Writeln
Einlesen von Standard-Eingabe	scanf	Read / Readln
Routinen zur Bearbeitung null-terminierter Zeichenketten:		
schreibe formatierten String in Variable	sprintf	StrFmt
kopiere String in einen anderen String	strcpy, strncpy	StrCopy, StrLCopy
vergleiche zwei Strings	strcmp, stricmp, strncmp	StrComp, StrICmp, StrLComp
hänge String an das Ende eines anderen an	strcat, strncat	StrCat, StrLCat
ermittle die Anzahl der Zeichen im String	strlen	StrLen
fülle String mit einem Zeichen	_strset, _strnset	FillChar
suche ein Zeichen im String (von links)	strchr	StrScan
suche ein Zeichen im String (von rechts)	strrchr	StrRScan
suche Zeichenkette im String	strcspn	StrPos
arithmetische Routinen:		
absoluter Wert (Betrag)	abs	Abs

Sinus, Cosinus, Tangens	sin, cos, tan	Sin, Cos, Tan
Sinus-, Cosinus-, Tangens- Hyperbolicus	sinh, cosh, tanh	Sinh, Cosh, Tanh
Arcus- Sinus, -Cosinus, -Tangens	asin, acos, atan	ArcSin, ArcCos, ArcTan
Exponentialfunktion e^x	exp	Exp
Potenz y^x	pow	Power
natürlicher und dekadischer Logarithmus	log, log10	Ln, Log10
Wurzel	sqrt	Sqrt
ganzzahliger Anteil	modf	Int
Komma-Anteil	modf	Frac
Typkonvertierung:		
Real => Integer (durch Abschneiden der Nachkommastellen)	int()	Trunc
Gleitkommawert => String	_ecvt, _fcvt, _gcvt	FloatToStr
Gleitkommawert <= String	strtod	StrToFloat
Integer => String	_itoa, _ltoa	IntToStr, Str
Integer <= String	strtol, atoi, atol	StrToInt, Val
Zeit => String	ctime	TimeToStr
Zeit <= String	-	StrToTime
Datum => String	strftime	DateToStr
Datum <= String	-	StrToDate
UNI-Code String => Ansi-Code String	wcstombs	WideCharToString
UNI-Code String <= Ansi-Code String	mbstowcs	StringToWideChar
Speicherverwaltung:		
reserviere Speicher auf dem Heap	malloc	AllocMem, GetMem
ändere Speichergröße	realloc	ReallocMem
Speicher freigeben	free	FreeMem
kopiere Bytes von der Quelle zum Ziel	memcpy, memmove	Move
fülle Speicher mit angegebenem Zeichen	memset	FillChar
Datei- und Verzeichnisverwaltung:		
Datei umbenennen	rename	RenameFile
lösche angegebene Datei	remove	DeleteFile
Verzeichnis erstellen	_mkdir	CreateDir
lösche Verzeichnis	_rmdir	RemoveDir
wechsle Arbeits-Verzeichnis	_chdir	SetCurrentDir

3.2 Klassenbibliotheken "Microsoft Foundation Classes" (MFC) und "Visual Component Library" (VCL)

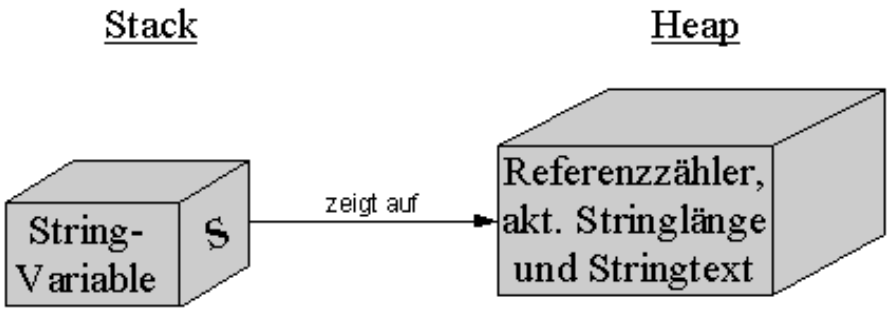
3.2.1 Erweiterte Stringtypen

Einige Klassen der MFC gehören nicht zum Hierarchiebaum dieser Klassenbibliothek. Sie sind nicht von der Klasse CObject abgeleitet und dienen allgemeinen Zwecken, wie der Serialisierung (Klasse CArchive), der Verwaltung von Strings, Daten und Listen. In Object Pascal werden dafür vielfach neue, elementare Typen eingeführt.

MFC	Object Pascal / VCL
Klasse CArchive	-
Klasse CString	Typ String (AnsiString)
Klasse CTime	Typ TDateTime
Klasse CTimeSpan	Typ TDateTime
Klasse COleDateTime	Typ TDateTime
Klasse COleCurrency	Typ Currency
Klasse COleVariant	Typ Variant

Naturgemäß werden in den meisten Programmen String-Typen besonders oft verwendet. Die Klasse CString in Visual C++ und der Typ String in Object Pascal stellen eine Alternative zu den umständlich zu nutzenden Char-Arrays und den Standard-Pascal-Strings (Typ ShortString), die maximal 255 Zeichen aufnehmen können, dar.

Alle Methoden der Klasse CString sind nicht-virtuelle Methoden. Die Klasse besitzt nur ein einziges Feld (=Variable), das einen Zeiger auf eine Struktur darstellt. Auch der String in Object Pascal stellt in Wahrheit nur einen Zeiger auf eine ganz ähnliche Struktur dar. Selbst wenn eine Variable vom Typ CString bzw. String statisch deklariert wurde, wird der ihr zugewiesene Text niemals auf dem Stack, sondern immer auf dem Heap abgelegt. Neben dem eigentlichen Text werden auf dem Heap noch einige Zusatzinformationen (wie z.B. die aktuelle Länge des Stringtexts) gespeichert.



CString- bzw. String-Variable besitzen keine deklarierte maximale Länge und können bei Bedarf Text bis zu einer Länge von 2 GByte aufnehmen. SizeOf(String) liefert aber, unabhängig von der Textlänge, immer 4 Byte als Ergebnis.

VC++	Object Pascal
`#include <afx.h>` `CString S = "Das ist ein Text";` `printf("%d Byte", sizeof(S));` `Ausgabe:  4 Byte`	`var S: String;` `S:= 'Das ist ein Text';` `writeln(SizeOf(S), ' Byte');` `Ausgabe:  4 Byte`

Beim Auslesen/Zuweisen einer CString- bzw. String-Variablen wird automatisch eine Dereferenzierung des Zeigerwertes vorgenommen. Man braucht nicht zu beachten, daß man es in Wahrheit ständig mit einer Zeiger-Variablen zu tun hat. Auch die Verwaltung des dynamisch reservierten Speichers für CString- bzw. String-Variable erfolgt automatisch und erfordert keinen zusätzlichen, vom Entwickler zu schreibenden Code. Insbesondere kann der gespeicherte Text länger werden, ohne daß man sich um die Bereitstellung zusätzlichen Speichers kümmern muß. Diese Eigenschaft der CString- und String-Variablen vereinfacht die oft benötigte Verkettungs-Operation. Das Verketten ist mit dem Plus-Operator möglich.

VC++	Object Pascal
CString S1, S2, S3; S1 = "Das ist ein zusam"; S2 = "men-gesetzter Text"; **S3 = S1 + S2;** printf("%s", S3);	**var S1, S2, S3: String;** S1:= 'Das ist ein zusam'; S2:= 'men-gesetzter Text'; **S3:= S1 + S2;** writeln(S3);

Der Zugriff auf einzelne Zeichen-Elemente kann wie bei Array-Typen erfolgen. Bei CString wird auf das erste Element mit der Index-Nummer 0, bei Object Pascal Strings mit der Index-Nummer 1 zugegriffen. Die Ausgabe des zweiten Zeichens erfolgt somit durch:

VC++	Object Pascal
printf("%c", **S3[1]**); Ausgabe: a	writeln(**S3[2]**); Ausgabe: a

Der schreibende Zugriff auf einzelne Zeichen muß bei CStrings mit der Funktion *SetAt(Index, Zeichen)* erfolgen.

Eine direkte Typwandlung in einen Zeigertyp, der auf ein Array von Zeichen zeigt, das mit einem Null-Zeichen endet, ist problemlos möglich. Funktionen, die Zeiger erwarten (wie z.B. viele Windows API-Funktionen), können so auch mit den neuen Stringtypen genutzt werden.

VC++	Object Pascal
MessageBox(0, **LPCSTR(S3)**,"", 0); oder durch MessageBox(0, **S3.GetBuffer(0)**,"", 0);	MessageBox(0, **PChar(S3)**,'',0);

Auch umgekehrt ist die direkte Zuweisung einer Zeigertyp-Variablen an eine CString- bzw. String-Variable möglich. Der Stringtext wird in dem Fall kopiert.

VC++	Object Pascal

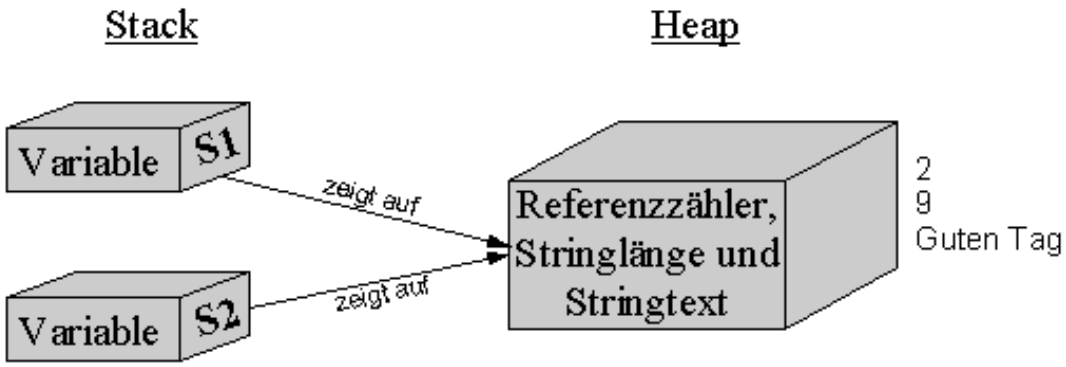
```
CString Str;
char* Txt;

Txt = new char[50];
strcpy(Txt, "Guten Tag");
Str = Txt;
delete Txt;

printf("%s", Str);
```

```
var Str: String;
    Txt: PChar;

GetMem(Txt, 50);
StrCopy(Txt, 'Guten Tag');
Str := Txt;
FreeMem(Txt, 50);

writeln(Str);
```

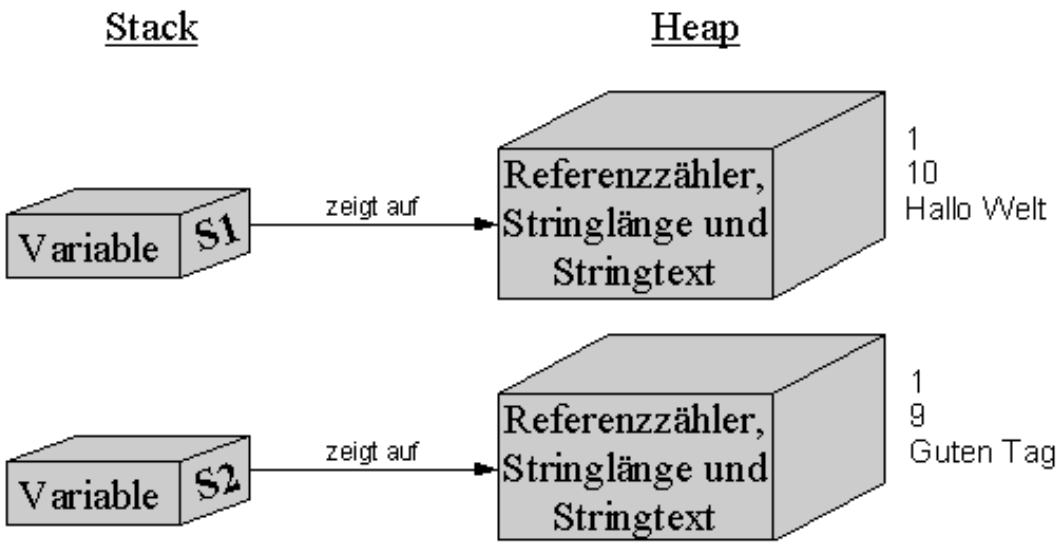
Eine der Zusatzinformationen, die bei CString und String neben dem eigentlichen Stringtext auf dem Heap gespeichert wird, ist ein Referenzzähler (reference counter). Bei der Zuweisung eines Strings zu einem anderen String, wird der möglicherweise recht lange Stringtext zunächst nicht umkopiert.

VC++	Object Pascal
`S1 = "Guten Tag";` `S2 = S1;`	`S1:= 'Guten Tag';` `S2:= S1;`

Das einzige, was kopiert wird, ist der 4 Byte lange Zeiger auf den Stringtext; d.h., im angegebenen Beispiel weist String-Variable S2 nach der Zuweisung auf den selben Stringtext wie S1.



Zusätzlich wird der Wert des Referenzzählers um eins erhöht. Man zählt somit die Anzahl der Verweise; der Referenzzähler hat jetzt den Wert 2. Wenn einer CString- bzw. String-Variablen ein *neuer*, anderer Wert zugewiesen wird, wird der Referenzzähler des vorherigen Wertes vermindert und der Referenzzähler des neuen Wertes erhöht.



VC++	Object Pascal	Wert des Referenzzählers, auf den S1 zeigt	Wert des Referenzzählers, auf den S2 zeigt
`CString S1;` `CString S2;` `S1 = "Guten Tag";` `S2 = S1;` **`S1 = "Hallo Welt";`**	`var S1: String;` `S2: String;` `S1 := 'Guten Tag';` `S2 := S1;` **`S1 := 'Hallo Welt';`**	- - 1 (neu) 2 (=1+1) 1 (neu)	- - - 2 1 (=2-1)

Man denke sich, daß beim Object Pascal-Beispiel beide Zuweisungen an die Variable S1 dynamisch zur Laufzeit erfolgen (z.B. durch Einlesen von Text über Readln(S1)). Bei Stringkonstanten und -literalen erzeugt Delphi einen Speicherblock mit derselben Belegung wie bei einem dynamisch reservierten String, setzt den Referenzzähler aber in Wahrheit auf den Wert -1. So können Stringroutinen erkennen, daß Blöcke, die einen Referenzzählerwert -1 besitzen, nicht verändert werden dürfen.

Wenn der Referenzzähler den Wert Null erreicht, wird der dynamisch belegte Speicherplatz, der den Stringtext und die Zusatzinformationen enthält, freigegeben.

Durch das Zählverfahren werden Zuweisungs-Anweisungen sehr schnell und speichersparend abgearbeitet. Erst wenn der Inhalt einer CString / String-Variablen verändert wird und zudem der Referenzzähler einen Wert größer 1 besitzt, muß eine String-Kopie angelegt werden. Im folgenden Beispiel wird eine Änderung am String durch die Änderung eines einzelnen Zeichens im String bewirkt.

VC++	Object Pascal

```
CString S1, S2;
S1 = "Guten Tag";
S2 = S1;
S1.SetAt(4, 'R');
// separate Kopie für S1
// wird automat. angelegt

printf("%s\n", S1);
printf("%s\n", S2);
```

Ausgabe:

GuteR Tag

Guten Tag

```
var S1, S2: String;
S1:= 'Guten Tag';
S2:= S1;
S1[5]:= 'R';
// separate Kopie für S1
// wird autom. angelegt

writeln(S1);
writeln(S2);
```

Ausgabe:

GuteR Tag

Guten Tag

Durch die angelegte Stringkopie wird sichergestellt, daß durch eine Modifikation nicht auch andere String-Variablen, die auf den selben Stringwert verweisen, verändert werden. Diese Vorgehensweise ist unter dem Begriff der "Copy-on-write-Semantik" bekannt.

Wenn in Object Pascal der Gültigkeitsbereich einer String-Variablen verlassen wird (wie z.B. beim Verlassen einer Funktion mit lokalen String-Variablen), wird ihr Referenzzähler automatisch um eins vermindert. Falls der Zähler dann Null ist, wird wiederum der dynamisch belegte Speicherplatz für den Stringtext freigegeben. String-Variable verhalten sich somit wie ganz gewöhnliche Variable; der dynamische Anhang wird geschickt vor dem Programmierer verborgen.

In Visual C++ werden statisch instanzierte Objekte automatisch beim Verlassen des Gültigkeitsbereichs freigegeben. Der Destruktor der Klasse CString sorgt auch hier für die Freigabe des dynamisch belegten Speichers, wenn nach dem Dekrementieren des Referenzzählers dessen Wert Null ist. CString-Variable können neben gewöhnlichen Strings auch solche im erweiterten UNI-Code Format aufnehmen.

Viele Operatoren und Funktionen sind in der CString-Klasse und in Object Pascal innerhalb der Laufzeitbibliothek in den Units *System* und *SysUtils* definiert. Das im Kapitel "2.3.2 Standardtypen" aufgeführte Beispiel zur Stringbearbeitung mit Hilfe von Zeigern auf Zeichenketten könnte mit CString bzw. String so realisiert werden:

VC++

```
#include <afx.h>

CString KurzText, KurzText2;

KurzText = "Hallo Welt";
printf("%s\n", KurzText);
KurzText2 = KurzText;
KurzText2 = KurzText2.Mid(3);
printf("%s\n", KurzText2);
KurzText.SetAt(3, 'b');
KurzText.SetAt(4, 'e');
printf("%s\n", KurzText);
KurzText = KurzText.Left(8);
printf("%s\n", KurzText.Right(5));
```

Ausgabe:

Hallo Welt

lo Welt

Object Pascal

```
{$LONGSTRINGS ON} //ist Standard

var KurzText, KurzText2: String;

KurzText:= 'Hallo Welt';
writeln(KurzText);
KurzText2:= KurzText;
Delete(KurzText2, 1, 3);
writeln(KurzText2);
KurzText[4]:= 'b';
KurzText[5]:= 'e';
writeln(KurzText);
Delete(KurzText, 1, 3);
writeln(Copy(KurzText, 1, 5));
```

Ausgabe:

Hallo Welt

lo Welt

Halbe Welt be We	Halbe Welt be We

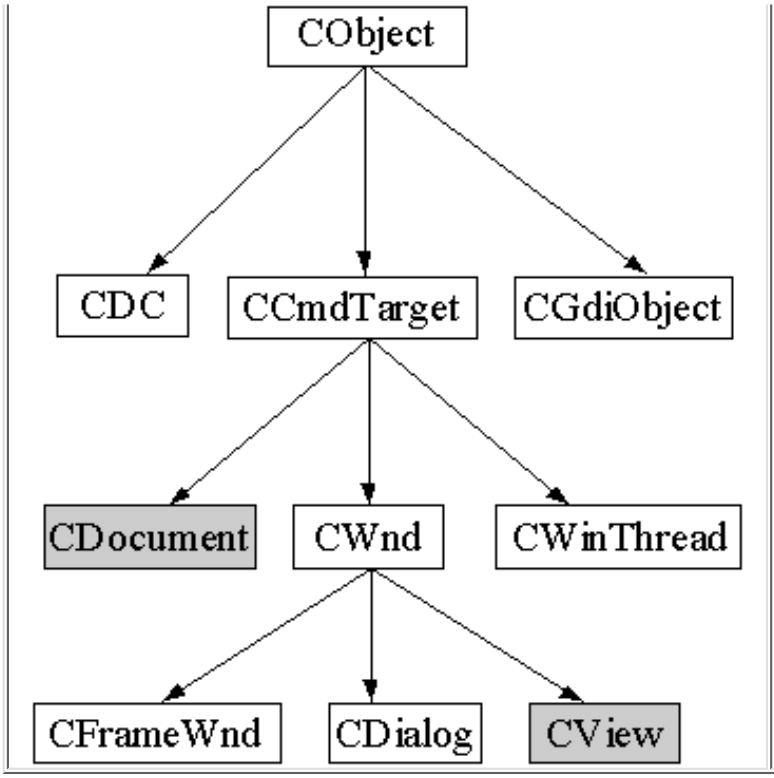
3.2.2 Architektur der Klassenbibliotheken

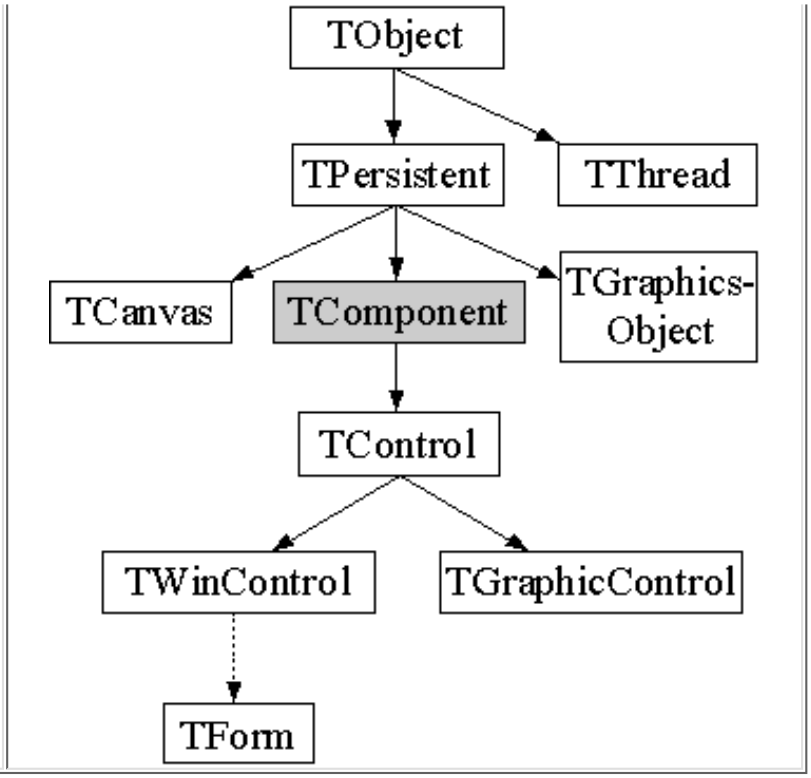
Die MFC und die VCL sind objektorientierte Klassenbibliotheken, die mit Visual C++ bzw. Delphi ausgeliefert werden. Sie sind in der jeweiligen Zielsprache implementiert worden (MFC in C++ und VCL in Object Pascal) und liegen fast vollständig im Quellcode vor. Ihr Studium bringt dem Leser die jeweilige Sprache nahe und kann zum Verständnis der Bibliotheken beitragen. Die in der MFC und VCL definierten Klassen können als Rahmen bei der Erstellung von eigenen Anwendungen dienen (Application Framework). Durch sie kann einfach ein Programmgerüst gebildet werden, das elementare Funktionalitäten aufweist und das mit anwendungsspezifischem Code zu füllen ist. MFC und VCL sind besonders gut auf die zahlreichen Aufgaben spezialisiert, die bei der Erstellung von Windows-Programmen anfallen und können dem Entwickler einen Großteil der Arbeit abnehmen, die bei herkömmlicher, prozeduraler Entwicklung entsteht. Sie kapseln durch eine Vielzahl von Klassen mehr oder weniger vollständig das Windows-API ab, verbauen dem Programmierer aber trotzdem nicht den direkten Zugriff darauf. Aufgrund des durch sie abgedeckten großen Funktionsumfangs sind beide Bibliotheken sehr komplex und ihre Handhabung entsprechend schwierig zu erlernen. Vollständige, fehlerfreie, aktuelle und sorgfältig aufbereitete Dokumentationen der Klassen-Bibliotheken, zum Beispiel in Form von elektronischer Online-Hilfe, stellen eine wichtige Voraussetzung dar, sie erfolgreich meistern zu können. Visual C++ und Delphi stellen sehr gute und umfangreiche Online-Hilfen bereit, die aber trotzdem noch an vielen Stellen Korrekturen und Verbesserungen vertragen können.

Die Entwicklungsumgebung von Visual C++ wurde mit Hilfe der MFC, die von Delphi mit Hilfe der VCL entwickelt. Beide Programme belegen überzeugend, daß mit den jeweiligen Klassenbibliotheken auch komplexe und anspruchsvolle Windows-Anwendungen erstellt werden können.

Die Namen aller MFC-Klassen beginnen mit dem Buchstaben "C", alle Klassennamen der VCL mit einem "T" (mit Ausnahme der Exception-Klassen, die alle mit "E" beginnen). Die Strukturansichten der Klassenbibliotheken widerspiegeln die teilweise unterschiedlichen Konzepte, die MFC und VCL zugrunde liegen.

zentrale Klassen in der MFC	zentrale Klassen in der VCL





Die Klasse **TObject** ist die Basisklasse aller Klassen innerhalb der VCL, gehört aber selbst nicht zur VCL, sondern ist Bestandteil der Sprachdefinition von Object Pascal und in der Unit *System.pas* implementiert. Auf den ersten Blick scheint diese Tatsache ein eher nebensächliches Detail zu sein. Bei genauerer Betrachtung der Klassenmodelle von C++ und Object Pascal erkennt man jedoch, daß durch die in der Basissprache von Object Pascal bereitgestellten Möglichkeiten zum Versenden und Empfangen von Botschaften, die Existenz erweiterter Laufzeit-Typinformationen, Metaklassen und Properties der Grundstein für die elegante Gestaltung einer Windows-Klassenbibliothek in Delphi gelegt wurde.

Dem Objektmodell von C++ fehlen drei der genannten objektorientierten Erweiterungen von Object Pascal. Auf die Nachbildung von Properties verzichtet die Klassenbibliothek von Visual C++. Die Klasse CCmdTarget und sogenannte Botschaftslisten (message maps) gestatten den Botschaftsaustausch zwischen wichtigen Objekten der MFC. Ein erweitertes, nicht ANSI-normiertes Laufzeit-Typinformations-Modell wird mit der Klasse CObject eingeführt.

CObject stellt eine Methode *IsKindOf()* bereit, die die erweiterten Laufzeit-Typinformationen auswertet. Die Funktion ermittelt, genauso wie TObjects Methode *InheritsFrom()*, ob ein Objekt mit irgendeiner anderen Klasse verwandt ist. Die Laufzeit-Typinformationen der Klasse CObject werden auch ausgewertet, um zur Laufzeit die Instanzierung beliebiger Klassen zuzulassen, die möglicherweise zur Übersetzungszeit noch nicht genau bekannt oder bestimmt sind.

```
class CMyClass : public CObject
{
  ...
};

CRuntimeClass* pRuntimeClass = RUNTIME_CLASS(CMyClass);
CObject* pObject = pRuntimeClass->CreateObject();
```

Im Beispiel wird dynamisch eine Instanz der Klasse CMyClass angelegt und pObject zugewiesen. Das ist möglich, obwohl CMyClass keinen virtuellen Konstruktor besitzt. Die dynamische Erzeugung von Objekten wird in Object Pascal mit Hilfe von Metaklassen und virtuellen Konstruktoren realisiert.

CObject bietet außerdem Unterstützung beim Lesen von Objekten aus dauerhaftem Speicher und beim Schreiben von Objekten in dauerhaften Speicher. Das Lesen und Schreiben wird zusammengefaßt auch "Serialisierung" genannt. Um Objekte dauerhaft, "persistent" abzulegen, werden oftmals Dateien auf beständigen Datenträgern, wie z.B. Festplatten angelegt. Bei der Serialisierung

legt ein Objekt seinen Status, der hauptsächlich durch den Inhalt seiner Felder bestimmt wird, auf dem permanenten Datenträger ab. Später kann das Objekt erneut erzeugt werden, indem der Objekt-Status wieder eingelesen wird. Das Objekt ist selbst für das Lesen und Schreiben seiner Daten verantwortlich. Deswegen müssen Klassen, die von CObject abgeleitet sind, die Methode "Serialize" in geeigneter Weise überschreiben.

```
class CMyClass : public CObject
{
public:
  DECLARE_SERIAL(CMyClass)
  CMyClass(){}; // Standard-Konstruktor muß definiert werden
  void Serialize(CArchive& archive);
  ...
  int Feld1;
  long Feld2;
};


void CMyClass::Serialize(CArchive& archive)
{
  CObject::Serialize(archive);

  // klassen-spezifische Schreib-/Lese- Aktionen
  if(archive.IsStoring())
    archive << Feld1 << Feld2;
  else
    archive >> Feld1 >> Feld2;
};
```

Um die "C++ Spracherweiterungen" der Klasse CObject nutzen zu können, müssen bei der Erstellung neuer Klassen spezielle Makros in der Klassendefinition aufgerufen werden. Im obigen Beispiel wird mit DECLARE_SERIAL(CMyClass) bekannt gemacht, daß die Klasse CMyClass die Funktionen zur Serialisierung benutzen wird. Bei der Implementierung der Klasse (in der CPP-Datei) muß ein weiteres Mal ein Makro aufgerufen werden: IMPLEMENT_SERIAL(CMyClass, CObject). Weitere Makros DECLARE_DYNAMIC / DECLARE_DYNCREATE und zugehörige IMPLEMENT-Makros können angewandt werden, wenn nur ein Teil der "Spracherweiterungen" benutzt werden soll.

In Delphis Klassenbibliothek VCL existiert keine direkte Entsprechung zu den Serialisierungs-Möglichkeiten fast aller MFC-Objekte. Der Klassenname "**TPersistent**" einer direkt von TObject abgeleiteten Klasse legt die Vermutung nahe, daß hier gleichwertige Methoden zu finden sein könnten; sie bestätigt sich aber nicht. Diese Klasse stellt vielmehr eine abstrakte Basisklasse dar, deren Methoden vom Benutzer zu überschreiben sind. Mit TStream, TMemoryStream, TBlobStream und TFileStream stellt aber auch die VCL Klassen bereit, die das flexible Lesen und Schreiben binärer Daten ermöglichen. Eine bessere, direkt nutzbare Unterstützung für die Serialisierung steht ab der Klasse TComponent zur Verfügung. Intern verwenden die Klasse TComponent und alle von ihr abgeleiteten Klassen ausgiebig Serialisierungs-Methoden, um den Status von Objekten in Ressourcen-Dateien abzulegen. Auf diese Weise werden unter anderem automatisch die Properties aller zur Design-Zeit instanzierten Komponenten in den Formular-Dateien (*.DFM) festgehalten. Methoden, die die explizite Serialisierung zusätzlicher Daten in den DFM-Dateien ermöglichen, stehen ebenfalls zur Verfügung. Die Serialisierung betreffende Themen sind in der Dokumentation Delphis ausgesprochen schlecht dokumentiert.

In der MFC ist die Klasse **CCmdTarget** ein direkter Nachfahre der Klasse CObject und stellt die "Botschaftslisten-Architektur" allen Nachfahr-Klassen zur Verfügung. Die Methode *OnCmdMsg* erlaubt das Versenden von Botschaften. Damit eine Klasse Botschaften empfangen kann, muß eine "Botschaftsliste" (message map) angelegt werden. Sie wird mitunter auch "Nachrichtentabelle" oder "Empfängerliste" genannt. Bei der Deklaration einer Klasse wird mit dem Makroaufruf DECLARE_MESSAGE_MAP() eine solche Botschaftsliste angelegt. Bei der Klassen-Implementierung (in der CPP-Datei) wird die Botschaftsliste *ausserhalb* der Klasse angelegt. In der Botschaftsliste werden für alle Botschaften spezielle Makros eingetragen, auf die in spezieller Weise reagiert werden soll. Für jede Botschaft ist eindeutig bestimmt, welche Methode beim Eintreffen dieser Botschaft aufgerufen werden soll. Die Botschaftsliste wird durch das Makro BEGIN_MESSAGE_MAP eingeleitet und durch das Makro END_MESSAGE_MAP beendet.

```
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
   //{{AFX_MSG_MAP(CMainFrame)
   ON_WM_KEYDOWN()
   ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
   ON_COMMAND(ID_FILE_NEW, OnFileNew)
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

Ähnlich wie bei den Botschaftsbehandlungs-Methoden von Object Pascal, die meist im *private*-Abschnitt einer Klasse deklariert werden, erfolgt hier innerhalb der Botschaftsliste die Verbindung zwischen der Botschaft, repräsentiert durch eine eindeutige Ganzzahl, und der aufzurufenden "Handler-Methode". Bei jeder empfangenen Botschaft werden die Botschafts-Nummern aller Einträge in der Botschaftsliste mit der eingegangenen Botschaft verglichen. Wird für eine Botschaft keine Botschaftsnummer und zugehörige Behandlungsroutine gefunden, wird automatisch in der Botschaftsliste der Basisklasse nach einer Behandlungsroutine gesucht. Der Name der Basisklasse muß deswegen beim Makroaufruf durch BEGIN_MESSAGE_MAP gesondert angegeben werden. Ebenso wie in Object Pascal wird die Suche solange fortgesetzt, bis entweder eine Behandlungsroutine aufgerufen oder die leere Botschaftsliste der Basisklasse CCmdTarget erreicht worden ist. Bei allen fensterbasierten Klassen (verwandt mit CWnd) wird dann die Fenster-Standardfunktion *DefWindowProc* der aktuellen Fensterklasse aufgerufen.

Im angegebenen Beispiel wird beim Auftreten der Botschaft ON_WM_KEYDOWN die Methode CMainFrame::OnKeyDown(...) aufgerufen; beim Auftreten der Botschaften ID_APP_ABOUT und ID_FILE_NEW werden die Methoden CMainFrame::OnAppAbout()bzw. CMainFrame::OnFileNew() aufgerufen.

Eine Besonderheit stellen die Kommentare

```
//{{AFX_MSG_MAP(CMainFrame)  und   //}}AFX_MSG_MAP
```

dar. Sie wurden nicht vom Programmierer geschrieben, sondern werden von Visual C++ angelegt und dienen dem "ClassWizard" als Orientierungshilfe beim Analysieren des Quelltextes. Sie werden auch Makro-Kommentare genannt und dürfen vom Programmierer nicht aus dem Quelltext entfernt werden, da der ClassWizard sonst nicht mehr benutzt werden kann.

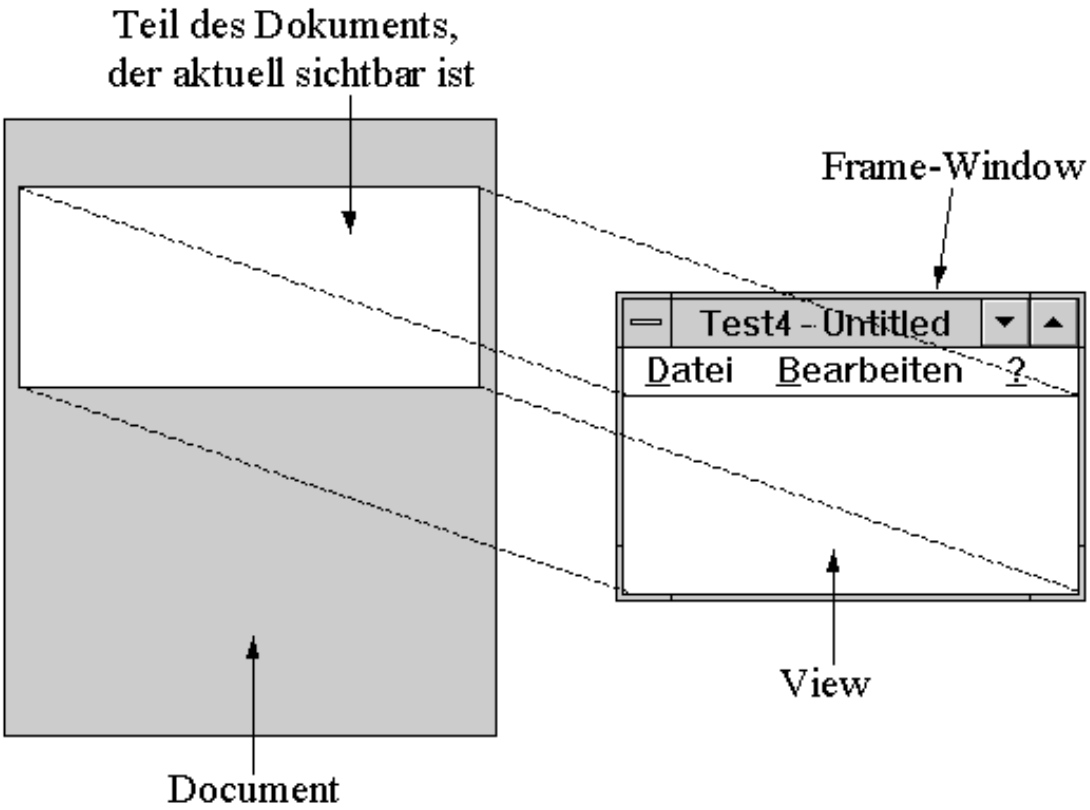
Delphi verzichtet generell auf das Anlegen derartiger Kommentare und wertet statt dessen sehr schnell ("on the fly") den Quelltext aus.

MFC und VCL verfolgen das Ziel, die Windows-Programmierung zu vereinfachen, so daß es kaum verwundert, daß die Klassen beider Klassenbibliotheken eine Reihe von Gemeinsamkeiten aufweisen. Trotzdem basieren MFC und VCL auf grundlegend unterschiedlichen Konzepten:

Konzept der MFC	Konzept der VCL
Document/View-Architektur	Komponenten-Architektur

In den weiter vorn gezeigten Klassenbäumen sind die Klassen, denen bei der Umsetzung der jeweiligen Architektur eine besondere Bedeutung zukommt, grau unterlegt.

Die **Document/View-Architektur** ist dadurch gekennzeichnet, daß das "Document" als Daten-Quelle bzw. Daten-Server fungiert und durch ein "View" zur Ansicht gebracht wird.



Der Anwender erstellt neue Dokumente über den Menüpunkt "Neu" oder lädt bestehende Dokumente über den Menüpunkt "Öffnen". Er arbeitet mit dem Dokument, indem er über das View auf das Dokument einwirkt. Dokumente werden typischerweise in Dateien gespeichert.

Eine anwendungsspezifische Klasse, die von der Klasse **CDocument** abgeleitet wird, fungiert als Daten-Objekt. Da ein Dokument selbst kein Fenster besitzt, wurde CDocument nicht von der "Fenster"-Klasse CWnd abgeleitet. Die Ableitung von CCmdTarget stellt andererseits sicher, daß auch mit Dokument-Klassen ein Botschafts-Austausch möglich ist. In der View-Klasse - **CView** - wird festgelegt, wie dem Endanwender das Dokument dargestellt wird und wie er mit ihm arbeiten kann. Verschiedene View-Klassen der MFC können als Basis für die Erstellung anwendungsbezogener Views dienen: CScrollView, CFormView, CEditView, usw. Gemeinsam ist allen Views, daß sie innerhalb von "Dokument-Rahmen-Fenstern" angezeigt werden. Wenn ein Programm in der Gestalt einer SDI-Windows-Anwendung (Single Document Interface) erscheinen soll, dann dient die Klasse **CFrameWnd** als Rahmen-Fenster. Soll das Programm dagegen in Gestalt einer MDI-Windows-Anwendung (Multiple Document Interface) erscheinen, so dienen die Klassen CMDIFrameWnd und CMDIChildWnd als Rahmen-Fenster. Ein "Application"-Objekt, das von der Klasse CWinApp abstammt, kontrolliert alle zuvor aufgeführten Objekte und ist zudem für die Initialisierung und korrekte Beendigung der Anwendung verantwortlich.

Auch Programme, die mit Delphis VCL erstellt werden, besitzen ein einziges "Application"-Objekt, das hier von der Klasse TApplication abstammt und ebenso wie CWinApp in der MFC als Programm-Kapsel fungiert. Die VCL weist eine **Komponenten-Architektur** auf, die sehr eng mit der Entwicklungsumgebung Delphis verknüpft ist. Alle Objekte, die man bei der Programmentwicklung öfter benötigt, werden in einer strukturierten "Komponenten-Palette" plaziert.

Die Komponenten sind die Bausteine eines Delphi-Programms, wobei jede Komponente ein einzelnes Anwendungselement, wie z.B. ein Dialogelement, eine Datenbank oder eine Systemfunktion darstellt. Alle Komponenten sind von der Klasse **TComponent** abgeleitet, die somit das grundlegende Verhalten aller Komponenten bestimmt. Delphis Entwicklungsumgebung und die VCL unterscheiden zwischen einer Designphase und einer Laufzeitphase. Eine Komponente kann prüfen, ob sie sich aktuell in der Designphase (im Formulardesigner) befindet:

```
if csDesigning in ComponentState then ...    // im Designer
```

Komponenten können durch derartige Prüfungen ganz bewußt ein verändertes Verhalten zur Designzeit aufweisen. Normalerweise

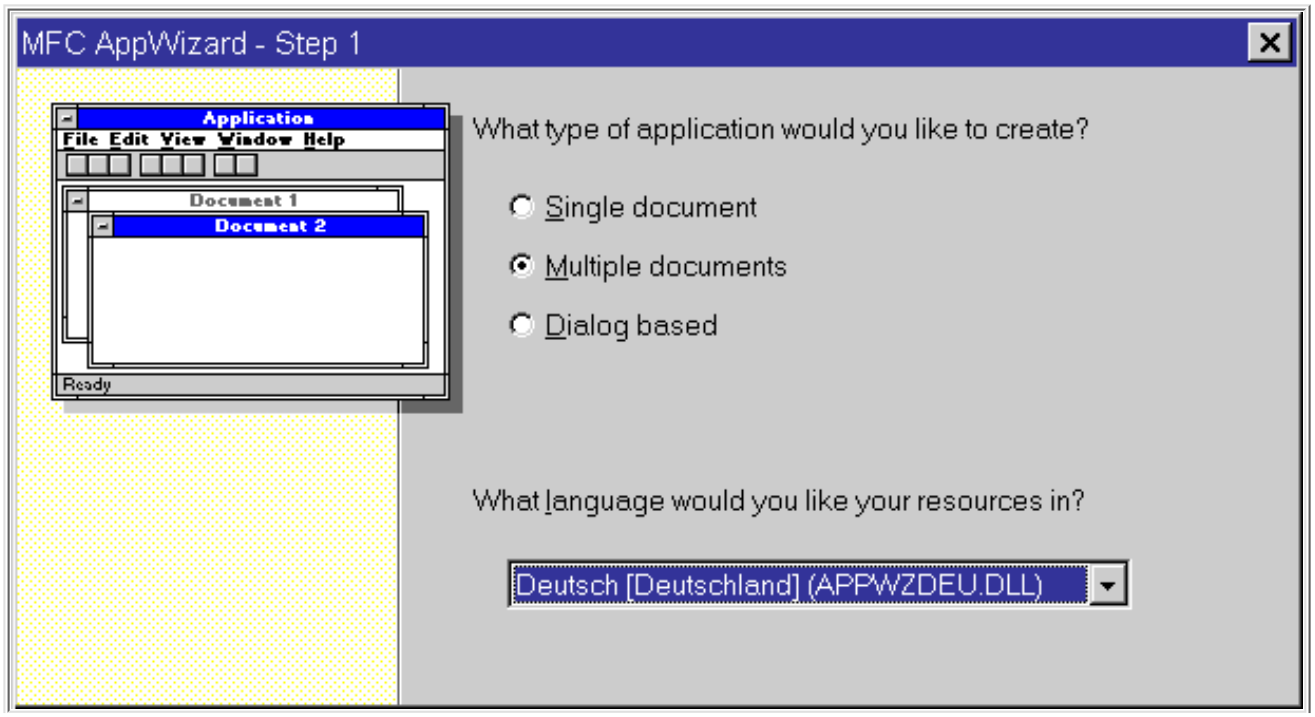
aber verhalten sich Komponenten während der Designzeit genauso wie zur Laufzeit des endgültig übersetzten Programms. Die Komponentenpalette kann beliebig erweitert werden, indem neue Klassen in der Entwicklungsumgebung angemeldet werden. Damit sich auch neu hinzugefügte Komponenten zur Designzeit genauso wie zur Laufzeit verhalten, werden alle Komponenten, die sich bei Delphi anmelden, übersetzt und in eine DLL-Bibliothek aufgenommen (Standard-Name Cmplib32.DCL). Während der Designzeit können Komponenten aus der Komponentenpalette ausgewählt werden und auf "Formularen", den Hauptfenstern der zukünftigen Anwendung, plaziert werden. Beim "Plazieren" einer Komponente erzeugt Delphi eine dynamische Objektinstanz von dieser Klasse. Einige Formulare erscheinen im endgültigen Programm allerdings nicht als Fenster. Solche Formulare existieren nur zur Designzeit und dienen z.B. als sogenannte Daten-Module.

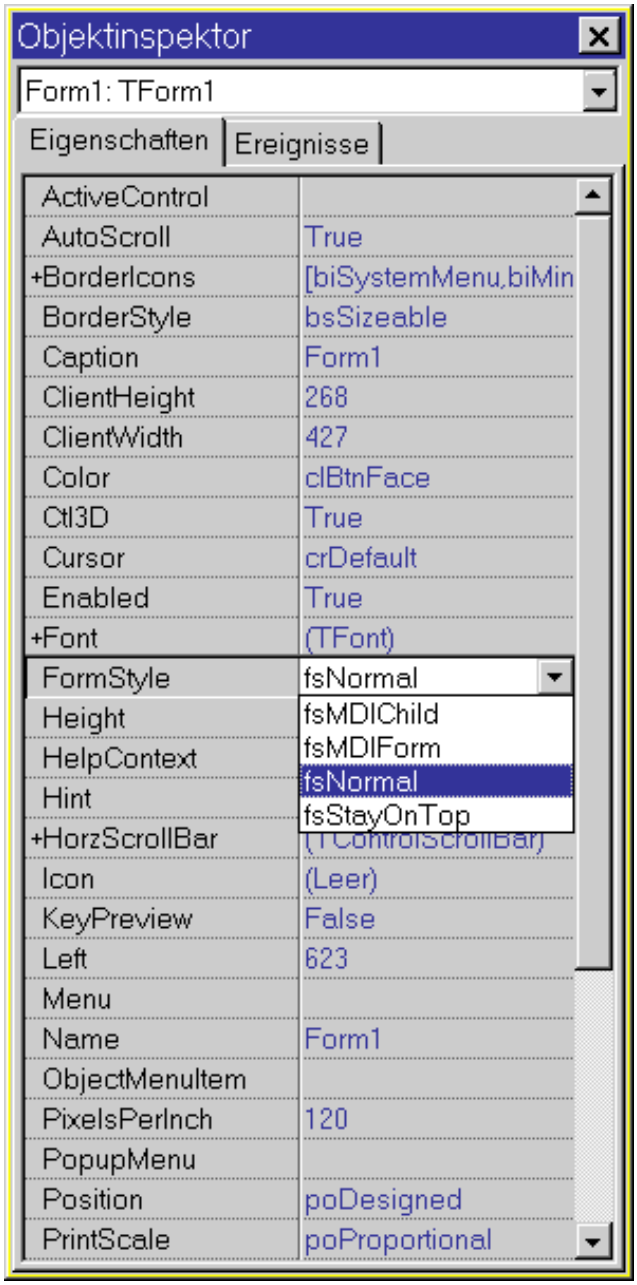
Neben sichtbaren Komponenten, wie Edit-Feldern, Schaltern, Listboxen usw. können in der Komponentenpalette auch solche Komponenten aufgenommen werden, die zur Laufzeit eines Programms unsichtbar sind. In der Standard-Version der VCL zählen dazu z.B. Timer oder Datenquell-Objekte für den Zugriff auf Datenbanken. Alle zur Laufzeit sichtbaren Komponenten müssen von der Klasse **TControl** abgeleitet werden. Sichtbare Komponenten werden in der VCL in jene unterteilt, die mit ihrer Komponente ein eigenes Fenster darstellen (und somit ein Handle von Windows beanspruchen) und in jene, die kein eigenes Window-Handle benötigen. Alle Komponenten, die von der Klasse **TGraphicControl** abgeleitet werden, verzichten auf den eigenen Fenster-Status und schonen damit die Windows-Ressourcen. Solche Komponenten können zur Laufzeit nicht fokussiert werden und erhalten keine Windows-Botschaften. Sie erhalten somit auch keine WM_PAINT Botschaft und werden deswegen von dem Formular dargestellt, auf dem sie plaziert wurden. Alle Komponenten, die von der Klasse **TWinControl** abgeleitet werden, besitzen automatisch ein eigenes Fenster-Handle. Die Klasse TWinControl ähnelt der Klasse CWnd in der MFC, bei der ebenfalls erstmals ein Fenster-Handle zur Klassendefinition dazukommt. Eine Entsprechung zur Klasse TGraphicControl existiert in der MFC nicht.

Während der Designphase ist ein Objektinspektor das Bindeglied zwischen dem äußeren Erscheinungsbild der Anwendung und dem Quelltext, der der Anwendung zugrundeliegt. Alle Properties einer Klasse, die im *published*-Abschnitt aufgeführt wurden, erscheinen als Einträge im Objektinspektor. Mit seiner Hilfe können die Property-Werte aller Komponenten und Formulare einer Anwendung festgelegt und jederzeit verändert werden.

3.2.3 Fensterbasierende Entwicklung

Der Verwaltung von und dem Umgang mit Fenstern fällt bei der Programmentwicklung unter Windows eine zentrale Rolle zu. In der MFC existieren für die verschiedenen grundlegenden Fenster eigene Klassen: CFrameWnd für SDI-Fenster, CMDIFrameWnd für MDI-Fenster und CDialog für Dialogboxen. Mit einem in Visual C++ integrierten Tool, dem AppWizard, kann bei der Neuerstellung von Programmen angegeben werden, zu welcher Kategorie das zukünftige Programm gehören soll. Der AppWizard generiert aus den gesammelten Informationen ein Programm-Grundgerüst, das als Basis für die weitere Programmierung dient.





In der VCL existiert dagegen nur eine grundlegende Hauptfenster-Klasse vom Typ TForm. Ob ein Hauptfenster wie ein SDI-, MDI- oder Dialogfenster erscheinen soll, wird durch ein Property des Formulars bestimmt, das mit dem Objektinspektor festgelegt werden kann. Anders als bei dem Programm-Generator AppWizard von Visual C++ können in Delphi einmal getroffene Entscheidungen jederzeit auf einfache Weise rückgängig gemacht werden. Wenn man z.B. ein Programm zunächst im MDI-Stil entwickelt hat, kann man es später mit sehr wenig Aufwand in ein Programm im SDI-Stil umwandeln, ohne eine Zeile Programm-Code ändern zu müssen.

Formulare können sich auch wie Dialoge verhalten. Die VCL verzichtet auf die Nutzung von Dialogboxen, wie sie durch das Windows-API zur Verfügung gestellt werden und emuliert sie durch gewöhnliche Fenster. Für einen Endanwender sind keine Unterschiede zu "normalen" Dialogboxen erkennbar. Für den Programmierer ergibt sich aber der Vorteil, daß ein Dialog ohne Aufwand z.B. in ein MDI-Fenster verwandelt werden kann.

Für normale Fenster eines Programms stehen in Visual C++ keine visuellen Editoren zur Verfügung. Nur Dialoge, die auf den "echten" Dialogboxen des Window-APIs basieren, können in Visual C++ mit Hilfe eines Ressourcen-Editors graphisch gestaltet werden. Der Ressourcen-Editor ist in der Entwicklungsumgebung eingebettet. Das Design von mit ihm erstellten Dialogen wird in einer separaten Datei gespeichert (*.RC). Dialoge besitzen zunächst keine Beziehungen zu Klassen. Über ein Tool mit dem Namen ClassWizard kann jedoch für jeden Dialog eine passende Klasse generiert werden. ClassWizard erzeugt für die so generierte Klasse eine CPP- und eine Header-Datei, die wiederum eine Vielzahl von Makro-Kommentaren aufweist. Die auf dem Dialog plazierten Dialog-Elemente repräsentieren, anders als die Komponenten in Delphi, keine Objekte. Sie erscheinen in Visual C++ deswegen logischerweise auch nicht in der Klassen-Definition. Die Dialog-Elemente werden statt dessen über Dialog-ID-Nummern angesprochen. Diese Nummern erhalten symbolische Bezeichner, die in den Header-, CPP- und RC-Dateien erscheinen. Eine Umbenennung eines solchen symbolischen Bezeichners führt, anders als bei Delphi, nicht zu einer automatischen Anpassung in den Quelltext-Dateien. Die Änderungen müssen per Hand durchgeführt werden. Für jedes Dialog-Element werden in der Ressourcen-Datei nur 5 Eigenschaften gespeichert: Windows-Fensterklasse, Fenstertext, Fensterstil, Position und Ausrichtung. Auch relativ einfache Änderungen, wie eine veränderte Farbdarstellung eines Dialog-Elements, können nicht im Ressourcen-Editor vorgenommen werden. Für eine Farbänderung muß z.B. Code geschrieben werden, der in geeigneter Weise auf eine Windows-Botschaft vom Typ WM_GETDLGCOLOR reagiert. Die Änderung anderer Eigenschaften erfordert oftmals ein Subclassing des Dialog-Elements zur Laufzeit. Alle Komponenten in Delphi weisen wesentlich mehr änderbare Eigenschaften (Properties) auf. Erst durch die Nutzung von OCX-Dialog-Elementen können auch im Ressourcen-Editor von Visual C++ weitere Attribute verändert werden. OCX-Dialog-Elemente befinden sich in separaten DLL's. Sie müssen, ähnlich wie Komponenten in Delphi, beim System angemeldet werden, bevor sie im Ressourcen-Editor benutzt werden können. Allerdings müssen bei Programmen, die solche OCX-Komponenten benutzen, die betreffenden DLL's zusammen mit dem endgültigen Programm ausgeliefert werden. Die Komponenten Delphis fügen sich dagegen nahtlos in die VCL ein und werden in die entstandene EXE-Datei mit aufgenommen. Um zusätzliche DLL's braucht man sich nicht zu kümmern.

Auch die Behandlung spezieller Ereignisse von Dialog-Elementen erfordert in Visual C++ ein Subclassing des Elements - eine Ereignis-Handler-Funktion muß manuell geschrieben werden. Alle Komponenten der VCL weisen alle wichtigen Ereignisfunktionen auf; ein Subclassing ist nicht nötig.

Da Visual C++ und die MFC nur Dialog-Elemente unterstützen, die fenster-basiert sind (also ein Window-Handle besitzen), stellen selbst "Static-Text"-Elemente ein Fenster dar. Dies stellt eine Ressourcen-Verschwendung dar, da solche Dialog-Elemente niemals fokussiert werden können und deswegen auch kein Window-Handle benötigen. "Static-Text"-Elemente stellen in der VCL

Ableitungen der Klasse TGraphicControl dar und belegen kein Window-Handle.

Da die Dialog-Elemente in Visual C++ keine Objekte darstellen, muß die MFC und der Programmierer einigen Mehraufwand betreiben, um die vom Endanwender eingegebenen Daten zu verifizieren und in das eigentlich objektorientierte Programm zu transferieren. Die MFC führt dazu ein Dialog-Verfifizierungsmodell (DDV = Dialog Data Validation) und ein Dialog-Datenaustauschmodell (DDX = Dialog Data eXchange) ein, das einige Arbeit abnehmen kann. Bei Delphis VCL findet der Datentransfer zwischen Windows-Element und einem Objektfeld innerhalb der Komponenten statt. Der Entwickler kann Validierungen vornehmen, indem er vordefinierte Ereignisse überschreibt und braucht sich um Datentransfers nicht zu kümmern.

 Zurück zum Inhaltsverzeichnis

 Weiter in Kapitel 4

4. Zusammenfassung

Die Entwicklungssysteme Visual C++ und Delphi werden mit integrierter Entwicklungsumgebung, eingebettetem C++ - bzw. Object Pascal - Compiler, Zusatzbibliotheken inklusive Klassenbibliothek und zusätzlichen Tools ausgeliefert. Bei der Bewertung der Systeme muß insbesondere zwischen den Compilern, den Spracheigenschaften und den Klassenbibliotheken unterschieden werden.

Die Visual C++ und Delphi zugrundeliegenden Sprachen sind sehr leistungsfähig, aber auch sehr komplex. Im Gegensatz zu interpretierenden Sprachen wie z.B. Visual Basic, Java oder REXX, erzeugen die Compiler von Visual C++ und Delphi Programme, die selbständig vom Prozessor ausgeführt werden können. Bei der Erzeugung des Maschinen-Codes führen sie lexikalische, syntaktische und semantische Prüfungen durch, die viele Programmier-Fehler bereits vor der Ausführung des Programms aufdecken. Derartige Prüfungen sind interpretierenden Sprachen fremd, weswegen bei ihnen selbst primitive syntaktische Fehler häufig erst beim Ausführen der Programme auftreten und somit oft zu spät erkannt werden.

Die Übersetzer der Sprachen "Delphi" und "Borland C++" der Firma Borland basieren auf demselben Compiler-"Back-End". Möglicherweise ist das einer der Gründe, warum sich die Compiler von Visual C++ und Delphi oft sehr ähnlich verhalten. Gemeinsame Eigenschaften stellen z.B. dar:

- Code-Generierung nur für 32-bit Systeme,
- wieder-aufsetzende Fehlerprüfungen,
- Ausgabe von Warnungen und Hinweisen bei möglichen Programmierfehlern,
- umfangreiche Optimierungen bei der Code-Erzeugung,
- Inline-Assembler,
- Einbettung in leistungsfähige Entwicklungsumgebungen (IDE's) und
- integrierte, leistungsfähige Debugger zur Fehlersuche.

Die Übersetzungsgeschwindigkeit differiert dagegen stark. Delphi übersetzt Programme deutlich schneller als Visual C++. Großen Gewinn kann Delphi dabei aus vorübersetzten Unit-Dateien ziehen, die komplette Symboltabellen von Programmteilen darstellen. Die Qualität des von beiden Compilern generierten Codes ist vergleichbar gut. Die Compiler führen selbsttätig eine Reihe von Optimierungen aus. Das Vorhalten von Variablen in CPU-Registern, um Zugriffe auf den langsameren Speicher zu reduzieren, stellt ein Beispiel dafür dar.

Die grundlegenden Sprachregeln in Visual C++ und Object Pascal sind sehr ähnlich; die generelle Programmstruktur ist identisch. Praktisch jeder Algorithmus, der in einer der beiden Sprachen geschrieben wurde, kann deswegen durch rein mechanische Umsetzung in die jeweils andere Sprache

transformiert werden. Beide Sprachen zeichnen eine starke Typprüfung und die Möglichkeit aus, neue Typen aus der Palette bereits bestehender Elementartypen bilden zu können. Visual C++ und Object Pascal kennen auch moderne Sprachkonstrukte zur Behandlung von Ausnahme-Situationen (Exception-Handling) und try/finally-Ressourcen-Schutzblöcke. Ein Präprozessor gestattet in C++ die Expansion zuvor deklarierter Makros; die Makro-Technik ist Object Pascal nicht bekannt.

Größere Unterschiede weisen dagegen die Teile der Sprachdefinitionen auf, die objektorientierte Erweiterungen festlegen. Gemeinsam sind Visual C++ und Object Pascal die Eigenschaften, daß durch Klassen und Objekte Felder (=Variable) und Methoden (=Funktionen) verkapselt werden, differenzierte Zugriffsrechte auf diese bestehen und eine Vererbung bestehender Klassen an neue, abgeleitete Klassentypen möglich ist. Beide Sprachen unterstützen statische und virtuelle Methoden, die mit ihnen verbundenen polymorphen Objekt-Eigenschaften und Laufzeit-Typ-Informationen.

Darüber hinaus weist das Klassen-Modell von VC++ folgende entscheidenden Erweiterungen gegenüber dem von Object Pascal auf:

- Schablonen (Templates)
- überladbare Methoden und Operatoren
- Mehrfachvererbung
- static-Felder
- const-Felder und Methoden

Erweiterungen im Klassen-Modell von Object Pascal unterstützen in besonderem Maße dynamische Spracheigenschaften:

- Referenz-Modell, bei dem alle Objekte Referenz-Zeiger darstellen
- Properties (Eigenschaften)
- dynamische und Botschaftsbehandlungs-Methoden
- Botschaftszuteilung
- Metaklassen und virtuelle Konstruktoren

Das Fehlen einiger Spracheigenschaften in jeweils einer der beiden Sprachen muß nicht zwangsläufig als Nachteil gewertet werden. Die Sprachdefinitionen haben nach 25-jähriger Entwicklung einen derart großen Umfang angenommen, daß es schon heute schwer fällt, sie zu erlernen und vollständig zu verstehen. Die beste Sprache nützt aber dann nichts, wenn sie aufgrund ihrer Komplexität niemand mehr erlernen kann oder will. Es gilt auch hier: Weniger ist manchmal Mehr.

Das in Visual C++ integrierte C++ und das Object Pascal von Delphi stellen moderne und flexible Sprachen dar, die fast nur die guten Eigenschaften mit ihren Vorgängern C und Standard-Pascal gemein haben. Beide Sprachen besitzen spezifische Stärken und Schwächen. Ein gewichtiger Vorteil von C++ besteht darin, daß durch das ANSI/ISO-Komitee eine normierte Sprachdefinition erstellt wurde. Die Normierung stellt sicher, daß Quelltexte auch mit den C++ Compilern anderer Hersteller übersetzt werden können. Eine vergleichbare Normierung existiert für Object Pascal nicht.

Object Pascal hat eine ganze Reihe von Konzepten aus C++ übernommen, diese aber mitunter in eine einfachere, klarere Syntax umgesetzt. Als Beispiel können hier der dynamische Typkonvertierungsoperator "as" und der gegenüber C++ erweiterte Methodenzeiger angeführt werden.

Den beiden Entwicklungssystemen liegen Zusatzbibliotheken bei. Besondere Bedeutung kommt dabei den Klassenbibliotheken zu; genannt MFC in Visual C++ und VCL in Delphi. Sie wurden mit dem Ziel entwickelt, dem Entwickler einen Großteil der Arbeit abzunehmen, die beim Erstellen von Programmen für die Windows-Plattformen anfällt. Sie verkapseln das Win32-API in einer Vielzahl verschiedener Klassen, so daß auf die direkte Verwendung der Windows-Funktionen verzichtet werden kann.

Die Erweiterungen im Objektmodell der Sprache Object Pascal haben es den Entwicklern der VCL gestattet, die Klassenbibliothek elegant zu implementieren. Die MFC führt in der Basisklasse "CObject" vergleichbare Erweiterungen ein: Botschaftsbehandlungs-Methoden, Möglichkeiten zur Botschaftszuteilung und zum dynamischen Instanzieren von Klassen. Da diese Erweiterungen nicht Bestandteil der normierten Sprachdefinition von C++ sind, mußten die Entwickler der MFC diese Sprachzusätze durch Makros "hinein-mogeln". Sie erfüllen in der Praxis ihren Zweck, wollen aber trotzdem nicht so recht in das Erscheinungsbild einer modernen, objektorientierten Sprache passen.

Die Klassen der MFC lehnen sich sehr nah an das Windows-API an und bilden eine relativ dünne Schicht um es herum. Die Klassen der VCL abstrahieren das Windows-API auf einer höheren Stufe. Alle Windows-Kontrollelemente wie Schalter, Edit-Felder, Scrollbars usw., aber auch nicht sichtbare Systemfunktionen werden hier in sogenannten Komponenten-Klassen verkapselt, die auf Formularen, den Fenstern und Dialogen der Anwendung plaziert werden. Den Zustand dieser Elemente bestimmen Properties, denen man in einem, Objektinspektor genannten Fenster, aber auch durch Programmcode zur Laufzeit, Werte zuweisen kann. Die VCL führt ein erweitertes Ressourcen-Format ein, um zusätzliche Eigenschaften der Komponenten speichern zu können. Darin können z.B. hierarchische Beziehungen zwischen den Fenstern und Kontrollelementen festgehalten werden. Der Entwickler "belebt" die Elemente eines Formulars durch anwendungsspezifischen Code, der vorwiegend als Reaktion auf Ereignisse wie OnClick, OnChange, OnShow ausgeführt wird.

Die MFC nutzt das Standard-Ressourcenformat von Windows, um visuell gestaltete Dialoge zu speichern. Die Windows-Kontroll-Elemente auf Dialogen stellen keine Objekte dar, nur für die Dialoge selbst können nachträglich in der Anwendung Klassen angelegt werden. Ein ClassWizard kann einen Teil der dabei anfallenden Arbeit abnehmen. Alle Nicht-Dialog-Fenster einer MFC-Anwendung können nicht graphisch-visuell gestaltet werden. Diese Fenster und die auf ihnen erscheinenden Windows-Kontrollelemente müssen manuell zur Laufzeit erzeugt erzeugt werden, was einen erheblichen Mehraufwand darstellt. Die MFC unterstützt durch einige Klassen eine sogenannte Document/View-Architektur. Diesem Konzept liegt die Feststellung zugrunde, daß sich viele Programme insofern ähneln, als sie ihre Daten in Dokumenten/Dateien speichern und dem Anwender innerhalb von Fenstern, den Views, zur Ansicht bringen. Anwender hantieren - durch die Views - mit den Daten und speichern sie wieder in den Dokumenten. Programme, die nach diesem Schema arbeiten, erhalten durch die Document- und View-Klassen der MFC aktive Unterstützung. Es findet eine Entkopplung zwischen der Datenhaltung und der Datendarstellung statt.

Die Klassenbibliotheken MFC und VCL unterstützen die Anwendungsentwicklung mit einer Vielzahl weiterer Klassen. Zu ihnen zählen z.B. erweiterte Strings, Grafikklassen, dynamische Arrays, Listen, Container und Klassen zur Datenbankanbindung.

Der Vergleich von MFC und VCL hat deutliche Unterschiede zwischen den Klassenbibliotheken aufgezeigt. Bei der Entscheidungsfindung, welche Bibliothek - und damit verbunden- welches Entwicklungssystem verwendet werden soll, spielen eine Vielzahl von Faktoren eine Rolle. In jedem Fall sollte aber nicht das Entwicklungssystem die Lösung diktieren, sondern die angestrebte Lösung das Entwicklungssystem bestimmen. Wenn eine Programmaufgabe durch die Anwendung der Document/View-Architektur gelöst werden kann, ist dem Einsatz der MFC und Visual C++ der Vorzug zu geben. Falls zur Lösung einer Programmaufgabe eher viele Fenster, Dialoge und Windows-Kontrollelemente eingesetzt werden sollen, scheint die Verwendung der VCL und Delphis ratsam.

Die Klassenbibliotheken beider Sprachen stellen proprietäre, herstellerspezifische Lösungen dar und sind nicht standardisiert. Programme, die mit ihnen erstellt wurden, können praktisch kaum auf andere Computer- oder Betriebssysteme portiert werden. Nur die MFC bietet die Möglichkeit an, Programme zumindest für ein anderes Betriebssystem (Apple Macintosh-System) zu generieren.

Die Geschwindigkeit der mit Visual C++ und Delphi erzeugten Programme ist ähnlich. Rein subjektiv betrachtet, scheinen aber Programme, die mit der MFC erstellt wurden, ein bißchen schneller abzulaufen. Andererseits trägt der Einsatz der VCL mit der guten API-Kapselung und Delphis visueller Entwicklungsumgebung wesentlich zur Produktivitätssteigerung bei der Programmierung bei. Man kann sich in Delphi mehr darauf konzentrieren, *was* man erstellen möchte und muß seltener überlegen *wie* man ein bestimmtes Problem löst.

Zurück zum Inhaltsverzeichnis

Literaturverzeichnis