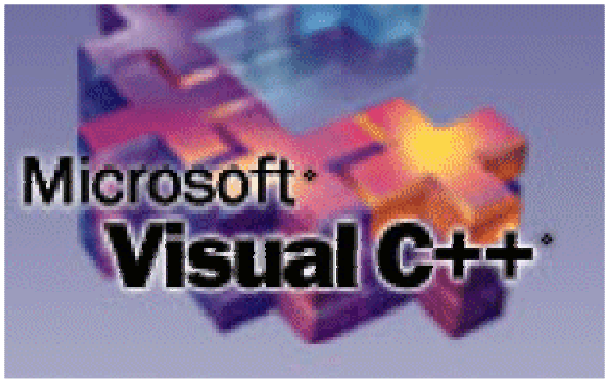
Seminarunterlagen
Einsatz von Microsoft Visual C++ 5.0

Norbert Schlia

Teil 1: Einführung in Microsoft Visual C++ 5.0
Teil 2: Programmierung in C/C++
Teil 3: Programmierung mit der Windows C API (Windows SDK)
Teil 4: Programmierung mit Microsoft Foundation Class (MFC) 4.2



1. Auflage 11/1997
2. erweiterte Auflage 04/1998
3. überarbeitete Auflage 09/1998
4. überarbeitete Auflage 10/1998
5. überarbeitete Auflage 12/1998

Vorwort

Dieses Script ist im Verlaufe der C++-Seminare entstanden, die ich für die Firma Makro Factory im Auftrag der Fraunhofer Gesellschaft halte. Mit jedem Seminar flossen neue Ideen in das Script ein, aber es wurden auch viele Fehler und Mängel aufgedeckt. Daher möchte ich den zahlreichen Seminarteilnehmern danken, die mit ihren Tips und Hinweisen zum Reifeprozess dieser Unterlagen beigetragen haben.

Ebenso möchte ich den Geschäftsführern der Macro Factory GdbR mbH, Torsten Materna und Peter Dittmar, danken. Durch ihr Vertrauen war es möglich, das Seminar mit einem eigenen Skript zu beginnen. Natürlich haben alle deutschen Verlage mit dazu beigetragen, da es nirgendwo ein passendes Script gab.

Vielen Dank auch an meine Mutter Brigitte Schlia für Korrekturlesen und das Überarbeiten des Scriptes.

Anregungen und Hinweise können jederzeit an eine meiner Email-Adressen geschickt werden.

Viel Erfolg mit dem hier Erlernten.

Norbert Schlia

Email: norbert@norbert.de
www: http://www.norbert.de/oblivion/
Compuserve: 101756,573

Inhaltsverzeichnis

VORWORT **5**

INHALTSVERZEICHNIS **7**

EINLEITUNG **11**

Über diese Trainingsunterlage 11
Inhalte der einzelnen Teile 11
Inhalt Teil 1: Einführung in Microsoft Visual C++ 5.0 ..... *11*
Inhalt Teil 2: Programmierung in C++ *12*
Inhalt Teil 3: Programmierung mit der Windows C API (Windows SDK) *12*
Inhalt Teil 4: Programmierung mit Microsoft Foundation Class 4.2 ..... *12*

ÜBERBLICK UND EINFÜHRUNG **13**

Ziele des Kapitels: 13
Hinweise zur Installation 13
Die Komponenten von Visual C++ 15

ARCHITEKTUR **19**

Terminologie der Microsoft Entwicklungsumgebungen 20
Das Developer Studio und seine Einstellungsoptionen 20
Neuerungen in Version 5.0 21
Automatisierung und Makros *22*
ClassView *22*
Benutzerdefinierte Symbolleisten und Menüs *22*
Debugger *22*
Direkthilfe *22*
Verbindung zum Internet *22*
Projektarbeitsbereiche und Projektdateien *22*
Ressourcen Editoren *23*
Texteditor *23*
Assistentenleiste *24*
Assistenten *24*
C++-Compiler, Linker- und Ressourcencompiler 24
Codeview-Debugger 24

APPWIZARD **27**

Ziele des Kapitels: 27
Einrichten und Verwalten von Arbeitsbereiche und Projekten 27
Automatisches Erstellen von Applikationen 29
Bereitstellen der Header-, Ressource- und sonstige Dateien 34
Löschen von Dateien aus dem Projekt 35
Die Popup-Menüs des Arbeitsbereichs 35
Compilierung des neuen Programmes 36
Verwendung des generierten Minimalprogrammes 37

CLASSWIZARD **40**

Nachrichten und Klassen miteinander verbinden 40

DER INTEGRIERTE DEBUGGER VON VISUAL C++ 5.0 ..... **44**

Ziele des Kapitels 44
Der Debugger 44
Befehle im Untermenü "Debug starten" des Menüs "Erstellen" 45
Befehle im Menü "Debug" zur Steuerung der Ausführung des Programms 47
Die Schnellüberwachung 48

PROGRAMMIERUNG IN C++ **50**

VORBEMERKUNGEN ... 50
Dieses Kapitel ... *50*
Aufbau des C++ Kurses ... *50*
DIE URSPRÜNGE VON C++ ... 51

DAS KONZEPT VON C++ ... 53

POLYMORPHISMUS ... 53
DATEINAMEN ... 53
AUCH DIE KOMMENTARE SIND BESSER ... 54
ERWEITERTER SYNTAX BEI FUNKTIONEN ... 54
Funktionsüberladung ... *54*
Defaultwerte bei Argumenten ... *55*

KLASSEN ... 57

BESTANDTEILE EINER KLASSE ... 58
INSTANZIERUNG VON KLASSEN ... 59
KONSTRUKTOREN UND DESTRUKTOREN ... 60
Standardkonstruktor und einfache Konstruktoren ... *60*
Minimalkonstruktor ... *60*
Konstruktoren mit Parametern ... *60*
Initialisierung von Konstanten mit Initialisierungslisten ... *60*
Kopierkonstruktor ... *61*
Destruktoren ... *61*
KLASSENFUNKTIONEN ... 63
Deklaration von Klassenfunktionen ... *63*
Aufruf von Klassenfunktionen ... *63*
ÜBERLADUNG ... 64
Überladene Klassenfunktionen ... *64*
Überladene Konstruktoren ... *64*
FUNKTIONEN MIT VORGABEWERTEN BEI ARGUMENTEN ... 65
INLINE FUNKTIONEN ... 66
FELDER VON OBJEKTEN ... 67
Initialisierung statischer Felder von Objekten ... *67*
Initialisierung dynamischer Felder von Objekten ... *67*
ZUGRIFF AUF FELDER ... 68

ABGELEITETE KLASSEN ... 69

ANPASSUNG VON KLASSENVERERBUNG ... 70
VIRTUELLE FUNKTIONEN ... 71
KONSTRUKTOREN UND DESTRUKTOREN ABGELEITETER KLASSEN ... 72
Initialisierungslisten ... *72*
Aufrufreihenfolge ... *72*
Virtuelle Destruktoren ... *73*
MEHRFACHVERERBUNG ... 74
Syntax der mehrfachen Vererbung ... *74*
KONSTRUKTOREN BEI MEHRFACHVERERBUNG ... 75
Initialisierungslisten ... *75*
Aufrufreihenfolge ... *75*
Namenskonflikte ... *75*

CONTAINER-KLASSEN UND UMFASSENDE KLASSEN ... 77

UMFASSENDE KLASSEN UND MEMBER-OBJEKTE ... 78
INITIALISIERUNGSLISTEN ... 79
AUSFÜHRUNGSREIHENFOLGE ... 80

ABSTRAKTE KLASSEN ... 81

WEITERE SPRACHELEMENTE ... 83

C++ EINGABE/AUSGABE ... 83
REFERENZVARIABLEN ... 84

Operator Überladung ... 85
Überladung von Operatoren ... 85
Liste überladbarer Operatoren ... 86
Erweiterungen der Initialisierungslisten ... 87
Initialisierbare Klassenelemente ... 87
Der Bereichsoperator "::" ... 88
Zeiger auf Objekte ... 89
Zeiger auf Basisklasse ... 89
Dynamischer und statischer Zeigertyp ... 89
This-Zeiger ... 90
Der Friend Operator ... 91
Friend-Klasse ... 91
Friend-Funktionen ... 91
Statische Klassenelemente ... 93
Statische Klassenvariable ... 94
Statische Klassenfunktion ... 96

NEUE SPRACHELEMENTE ... 97

Funktions- und Klassen-Templates (Schablonen) ... 97
Funktions-Templates ... 97
Klassen-Templates ... 97

EINFÜHRUNG IN DIE WINDOWS API UND DAS WINDOWS SDK ... 99

Die Windows API und ihre Tücken ... 99
Übungen ... 103
Übung 1 "HELLOWIN" ... 103
Übung 2 "SYSMETS" ... 108
Übung 3 "HEXCALC" ... 111
Übung 4 "SPINCUBE" ... 112
Übung 5 "DYNDLG" ... 112
Übung 6 "DEVCAPS" ... 112
Übung 7 "COLORS" ... 112

PROGRAMMIERUNG MIT MFC ... 113

Das erste MFC Programm ... 113
Application Architecture Class ... 114
CObject ... 114
CWinApp ... 114
Window Support ... 114
CWnd ... 115
CFrameWnd ... 115
Sonstige Klassen ... 115
Eine einfache MFC Anwendung ... 117
Ein Wörterbuch auf SQL Basis ... 119
Die Bestandteile der generierten Minimalanwendung ... 125
Datenbankanbindung automatisieren ... 125
Anzeigen der Daten in gebundenen Kontrollelementen (Bound Controls) ... 130
Verbinden der Controls mit der Datenbank ... 132
Fremdobjekte ... 132
DDX für Fremdobjekte ... 132
Der Klassen-Assistent und Fremdobjekte ... 133
Zuordnung von Datensatzansicht-Steuerelementen zu einer Datensatzgruppe ... 134
Setzen von Parametern und Sortierungsfolge der Datenbank ... 136
Füllen des Kombinationsfeldes ... 138
Suchfunktion realisieren ... 142
"Sicherheitsmaßnahmen" in der Entwicklungsversion ... 143
Die Übersetzung ... 144
Anzeige der Übersetzungen ... 146
Aufräumen und Schönheitskorrekturen ... 157

ANHANG 159

ABBILDUNGSVERZEICHNIS 159
ABKÜRZUNGEN UND FACHBEGRIFFE 161
DIE UNGARISCHE NOTATION 163
DIE STRUKTUR DER SQL DATENBANK 165
LITERATURHINWEISE 167

Einleitung

Über diese Trainingsunterlage

Dieses Skript dient als Leitfaden durch das **Seminar 3.10** "**Einsatz von Visual C++**". Voraussetzung sind Kenntnisse der Programmiersprache C und in objektorientierter Programmierung (3.01/3.02 Einführung in die Programmiersprache C, 3.07 Objektorientierte Programmierung - Konzepte, 3.08 objektorientierte Programmierung). Es wird die Bedienung von Windows 95/Windows NT sowie von Windowsprogrammen als bekannt angenommen.

Das Skript führt die Teilnehmer gezielt auf die für die im Alltag wichtigen Grundlagen hin und sie erlernen schnell und effektiv die Bedienung von **Visual C++ 5.0**. Auf eine vollständige Auflistung der Leistungsmerkmale wird absichtlich verzichtet, diese finden sich in den aktuellen Handbüchern von **Visual C++ 5.0**.

Für den jeweiligen Abschnitt relevante Worte sind fett gedruckt:

Um im entsprechenden Abschnitt **wichtige Begriffe** herauszuheben, sind diese dort **fett** gedruckt. Dies hebt die **wichtigen Informationen** hervor und erleichtert das Erfassen der **entscheidenden Details**.

Alle Beispiele sind folgendermaßen aufgebaut:

Ziel	→	erläutert die Problemstellung und formuliert die konkrete Aufgabe
Lösungsweg	→	führt Sie schrittweise zum gewünschten Ergebnis
Ergebnis	→	dokumentiert das richtige Ergebnis, z.B. einer Bildschirmausgabe
Anmerkung	→	weist auf ergänzende Details und wissenswerte Zusatzinformationen hin

Wichtige Punkte sind durch einen grauen Kasten hervorgehoben:

- Wichtige, prägnante Informationen können so schnell erfaßt werden.
- Das schnelle Nachschlagen und Wiederauffrischen von Informationen wird begünstigt.

Ziel:

Die Teilnehmer lernen ein modernes Entwicklungswerkzeug kennen und werden zielgerichtet in die einzelnen Komponenten von **Visual C++** eingeführt. Das Seminar vermittelt einen praxisorientierten Überblick über die Arbeitsweise mit **Visual C++**.

Inhalte der einzelnen Teile

Die einzelnen Teile sind thematisch abgeschlossenen Module, die auch einzeln behandelt werden können. Sie sollten jedoch in der vorgegebenen Reihenfolge abgearbeitet werden, da sie aufeinander aufbauen. Jeder Teil kann aber individuell je nach kollektivem Wissensstand mehr oder weniger vertieft werden.

Inhalt Teil 1: Einführung in Microsoft Visual C++ 5.0

- Hilfestellung zur Installation
- Beschreibung der verschiedenen Auswahlmöglichkeiten
- kurze Übersicht über die Zusatzkomponenten von **Visual C++**
- typische Einsatzgebiete der Komponenten
- Beschreibung der Einrichtung und Wartung von Arbeitsbereichen und Projekten
- Bereitstellung zusätzlicher Header- und Ressourcedateien
- Erstellen und Compilierung eines Minimalprogrammes mit dem **AppWizard**
- Erste Schritte in der manuellen Programmierung und Anpassung des Minimalprogrammes
- Beschreibung der Komponenten des Debuggers
- Handhabung des Debuggers

- Setzen und Anwenden von Haltepunkten
- Kontrolle von Variablen und Zuständen

Inhalt Teil 2: Programmierung in C++

- Das Tutorial gibt nur eine Übersicht über die Elemente und die Handhabung von C++.
- Die wichtigsten Teile von C++ werden angesprochen und erklärt.

Inhalt Teil 3: Programmierung mit der Windows C API (Windows SDK)

- Erlernen der Handhabung der Oberfläche von **Visual C++ 5.0**
- Umgang mit C, CPP, RC und anderen Dateien unter **Visual C++ 5.0**

Inhalt Teil 4: Programmierung mit Microsoft Foundation Class 4.2

- Einführung in die Grundlagen der Windows-Programmierung
- Sinn und Zweck von Rückruf- (Callback-) Funktionen
- Gerätekontexte und Handles
- Erlernen der Konzepte von **MFC**
- Umgang mit den Werkzeugen
- Kennenlernen der wichtigsten Klassen (Anwendung, Fenster, Datenbankzugriffe, Kontrollelemente).
- Erstellung von Online-Hilfen.

Überblick und Einführung

Ziele des Kapitels:

- Hilfestellung zur Installation
- Beschreibung der verschiedenen Auswahlmöglichkeiten
- kurze Übersicht über die Zusatzkomponenten von **Visual C++**
- typische Einsatzgebiete der Komponenten

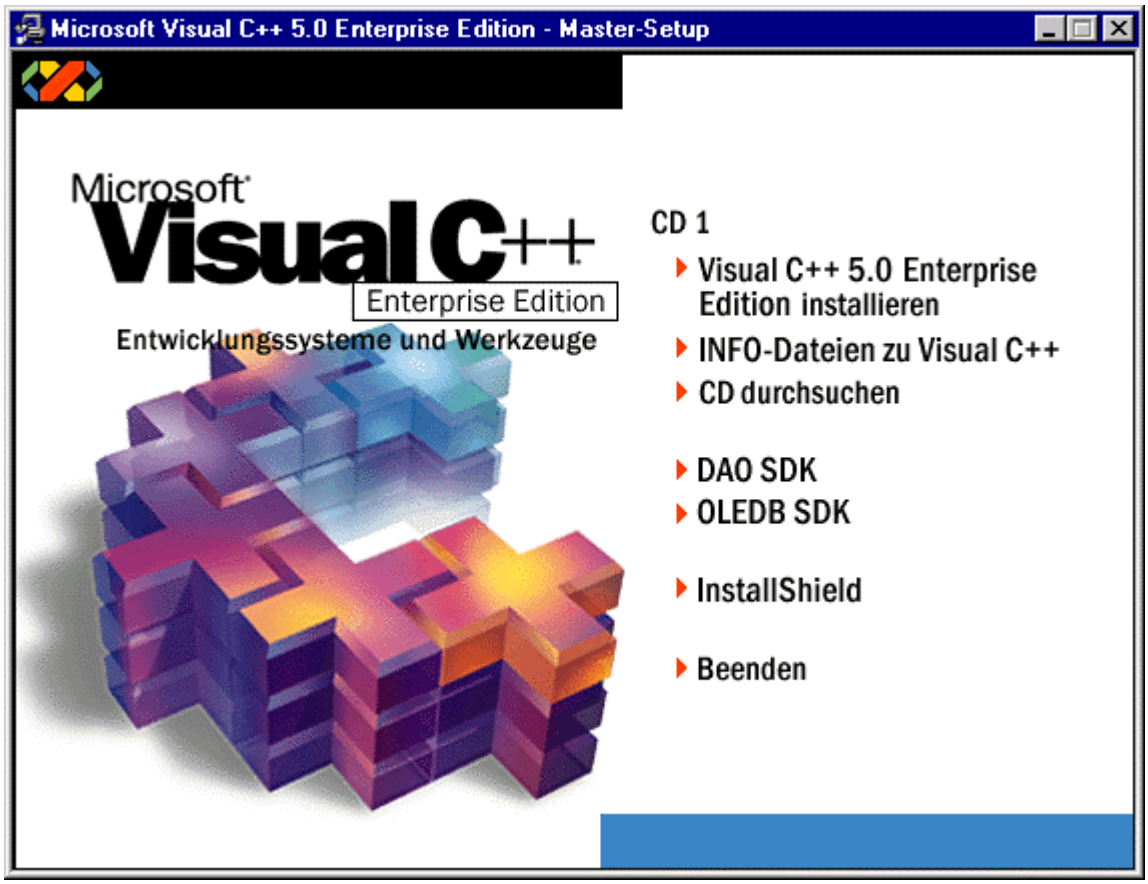


Abbildung 1: MS Visual C++ Master Setup

Hinweise zur Installation

Nach dem Einlegen der Installations-CD wird automatisch das Setup gestartet. Nach der Anwahl von "**Visual C++ 5.0 Enterprise Edition installieren**" beginnt die eigentliche Installationsroutine. Mit dem Master Setup können außerdem das **DAO (Database Access Objects) SDK** und das **OLE DB (Object Linking and Embedding Database) SDK** installiert werden. Ebenso ist eine auf **Visual C++** angepaßte Version von **InstallShield** vorhanden, mit dem professionelle Windows- Installationsprogramme erstellt werden können.

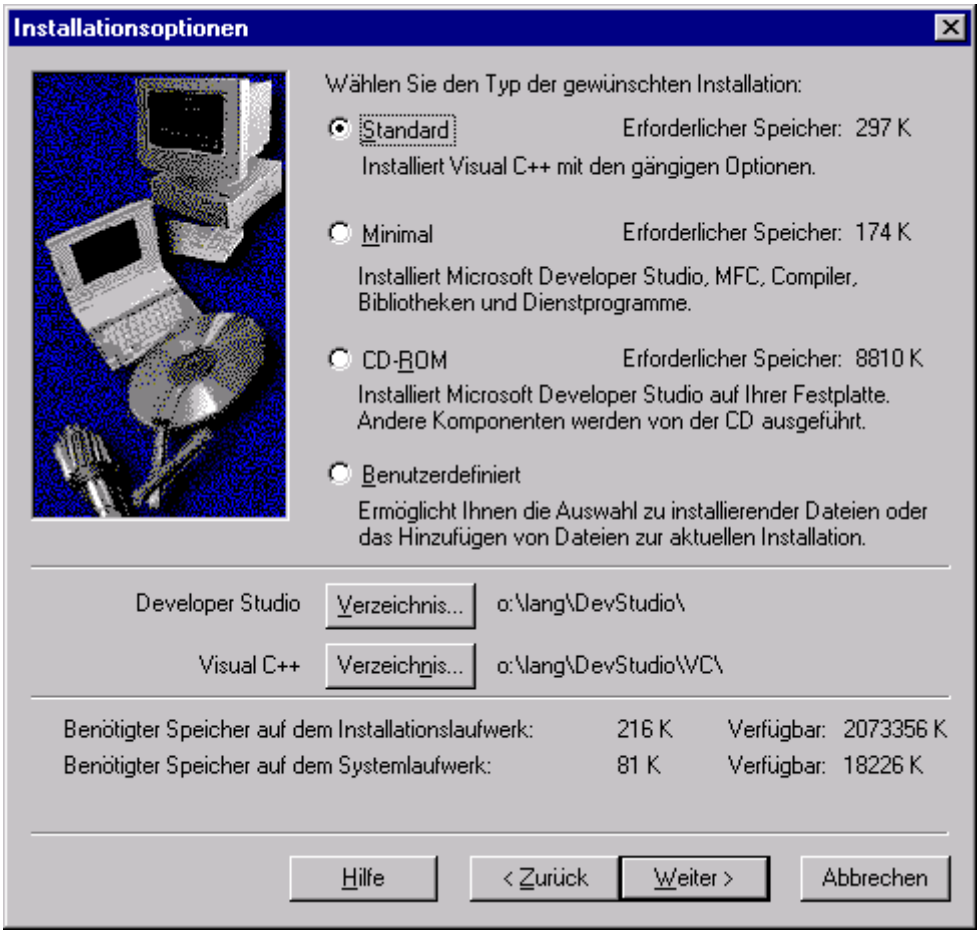
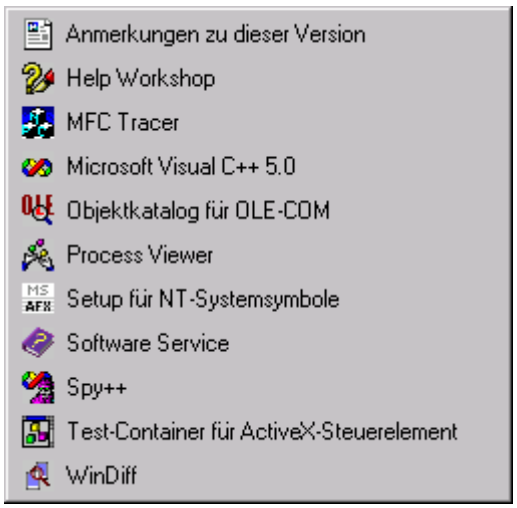


Abbildung 2: Visual C++ Setup Installationsoptionen

In der Regel genügt es, die Standardinstallation ablaufen zu lassen. Sie installiert alle notwendigen Komponenten, inklusive **DAO** und **Windows SDK**. Platzsparender ist die Minimalinstallation, fehlende Komponenten können jedoch jederzeit nachträglich installiert werden. Ebenfalls platzsparend ist die Option "CD-ROM". In diesem Fall aber muß die CD immer im Laufwerk sein, wenn man **Visual C++** verwendet. Unter der benutzerdefinierten Installation können zusätzliche Komponenten ausgewählt werden. Um trotzdem ein schnelles Arbeiten zu gewährleisten, ist dazu allerdings ein schnelles CD-ROM-Laufwerk notwendig.



Die Komponenten von Visual C++

Abbildung 3: Visual C++ Programmgruppe

Help Workshop

Der **Help Workshop** ermöglicht Erstellung und Wartung von **Windows Online-Hilfen**. Er erleichtert die ansonsten ziemlich umständliche Umwandlung von **RTF** (**Rich Text Format**) Dateien in das Windows HLP-Format.

OLE/COM Object Viewer

Verwalten von **OLE-** und **COM-**Objekten, Übersicht über installierte **OCX/ActiveX Komponenten** und deren Eigenschaften.

Process Viewer

Übersicht über laufende Prozesse und Threads auf dem eigenen und anderen Computern im Netzwerk. Anzeige des Speicherbedarfs, der Speicherbelegung, der Rechenzeiten und der Thread-Prioritiäten. Es ist auch möglich einige Änderungen vorzunehmen:

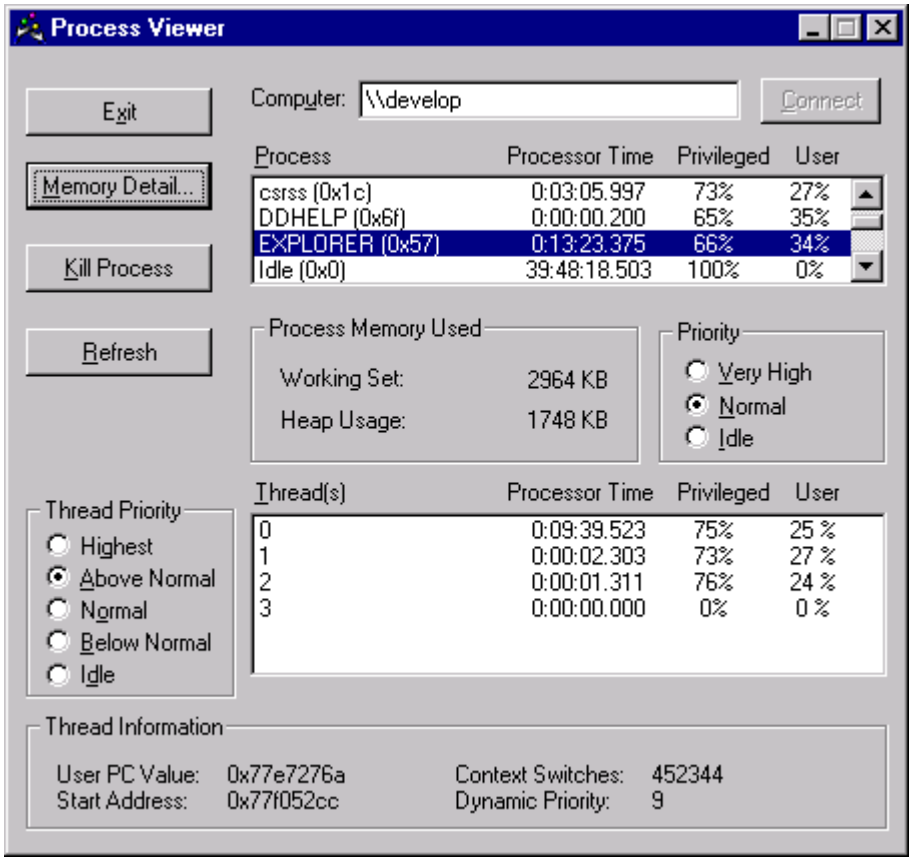


Abbildung 4: Visual C++ Process Viewer

Setup für NT-Systemsymbole

Normalerweise kann man eigene Programme nur soweit debuggen, bis sie NT-Systemfunktionen aufrufen. Dort fehlen dann natürlich die notwendigen symbolischen Informationen, die dem Debugger Aufschluß über Variablen- und Funktionsnamen geben. Mit der Installation der Systemsymbole bekommt er die notwendigen Zusatzinformationen, so daß das Debuggen sehr erleichtert wird. Der Vorteil wird mit dem Preis zusätzlichen Plattenplatzes erkauft, daher sind die Symboldateien (*.dbg) bei der "normalen" Installation von NT nicht dabei.

Spy++

Spy++ (SPYXX.EXE) ist ein Win32-basiertes Programm, das Ihnen eine Grafikansicht der Prozesse, Threads, Fenster und Fensternachrichten des Systems ermöglicht. **Spy++** kann:

- eine Grafikstruktur der Beziehungen zwischen den einzelnen Systemobjekten anzeigen. Dazu gehören: Prozesse, Threads und Fenster.
- nach bestimmten Fenstern, Threads, Prozessen oder Nachrichten suchen.
- die Eigenschaften ausgewählter Fenster, Threads, Prozesse oder Nachrichten anzeigen.
- direkt in der Ansicht ein Fenster, einen Thread, einen Prozeß oder eine Nachricht auswählen.
- das Suchwerkzeug zur Wahl eines Fensters durch Mauspositionierung verwenden.
- Nachrichtenoptionen durch Verwenden umfassender Parameter zur Nachrichtenprotokollierung festlegen.

WinDiff

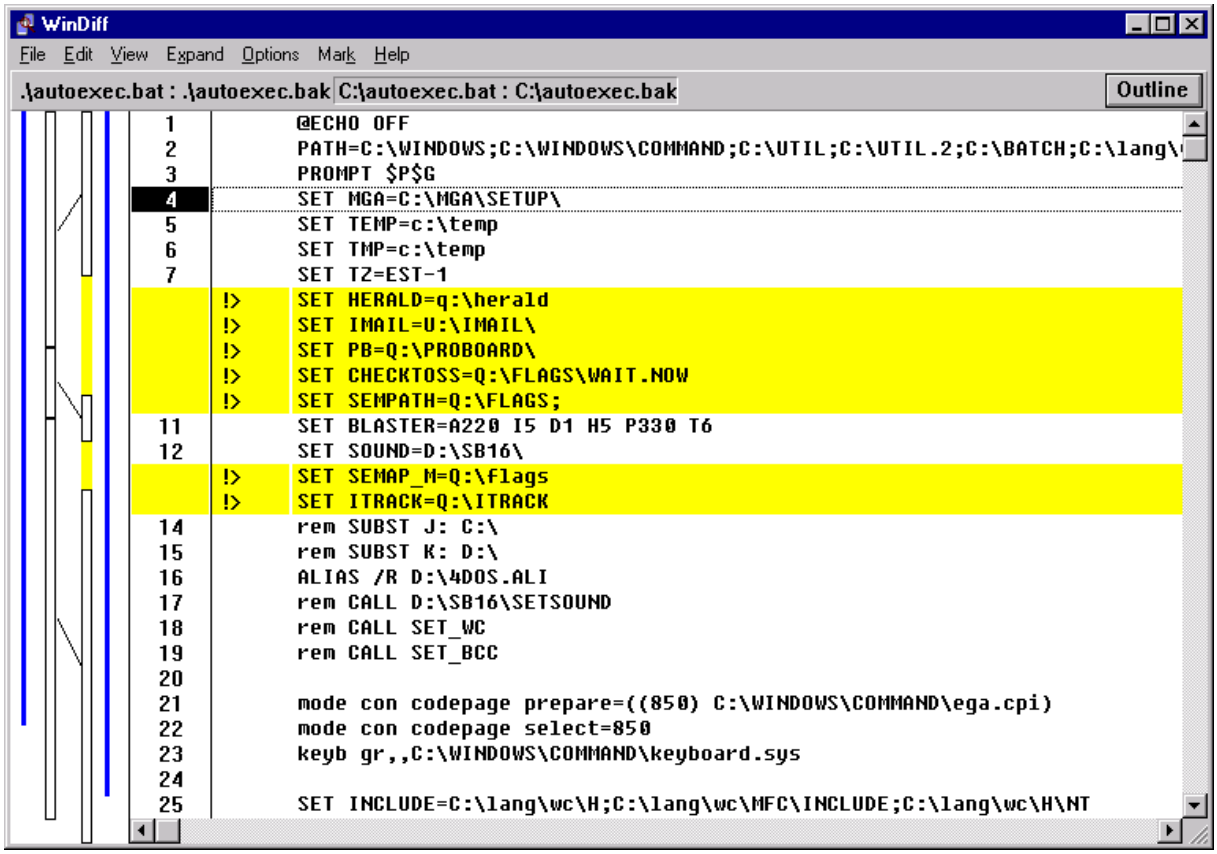


Abbildung 5: WinDiff

WinDiff vergleicht zwei Dateien und stellt die Abweichungen graphisch dar. Dies ist hilfreich beim Vergleichen von Sourcecodes oder Texten.

ActiveX-Container

Der Testcontainer stellt einen **ActiveX-Steuerelement-Container** bereit, in dem Sie Ihr Steuerelement testen können. Nachdem das Steuerelement erfolgreich erstellt worden ist, können Sie seine Leistungsfähigkeit und den Funktionsumfang testen, indem Sie die Eigenschaften verändern, die Methoden aufrufen und die Ereignisse auslösen. Der **Testcontainer** kann Ereignisprotokolle und Benachrichtigungen zur Datenbindung anzeigen. Außerdem bietet er Möglichkeiten zum Testen der Robustheit der Funktionen eines **ActiveX-Steuerelements**: Sie können Eigenschaften als Stream oder im Substorage speichern, sie erneut laden und die gespeicherten Stream-Daten überprüfen.

Architektur

Abbildung 6: MS Visual C++ Logo

Ziele des Kapitels:

- Beschreibung des **Developer Studios**
- Erklärung der Komponenten des **Developer Studios**
- Einrichten eines Arbeitsbereiches und eines Projektes

Die Entwicklungsoberfläche von **Visual C++** wurde von Microsoft vollständig neu konzipiert. Alle Programmiersprachen von Microsoft besitzen jetzt dieselbe Oberfläche: **Visual C++ 5.0**, **Visual BASIC 5.0**, **Visual Java 1.1**. Damit hat der Entwickler den Vorteil, daß er sich nicht mit diversen Oberflächen und Terminologien herumplagen muß. Alle installierten Programmiersprachen finden sich "unter einem Dach", dem gemeinsamen "**Developer Studio**".

Microsoft Visual C++, Version 5.0, ist in drei Editionen erhältlich:

Einsteiger Edition

Erlaubt das Erlernen der Programmiersprache C++ und das Arbeiten mit den Profi-Tools von **Visual C++**. Die Einsteiger-Edition schließt alle Funktionsmerkmale der **Professional-Edition** ein, mit Ausnahme der Möglichkeiten zur Codeoptimierung, des Profilers und der statischen Verknüpfung mit der **MFC-Bibliothek**

Professional Edition

Entwickeln von Anwendungen, Diensten und Steuerelementen für die Win32-Plattformen, einschließlich Windows 95 und Windows NT. Sie können die Benutzeroberfläche des Betriebssystems oder Konsolen-APIs als Ziel wählen.

Enterprise Edition

Entwickeln und Testen von Client-/Server-Anwendungen für den Einsatz im Internet oder Intranet. Die Enterprise-Edition enthält zusätzlich zu den Funktionen der Professional Edition Dienstprogramme für die Arbeit mit SQL-Datenbanken und zum Testen von gespeicherten SQL-Prozeduren. Das Quellcode-Verwaltungssystem **Visual SourceSafe** vereinfacht die Entwicklung von Anwendungen im Team.

Terminologie der Microsoft Entwicklungsumgebungen

Die natürlichen Gegebenheiten sind eigentlich immer dieselben, daher unterscheiden sich die Oberflächen integrierter Entwicklungsoberflächen verschiedener Hersteller wie Watcom, Computer Associates oder Microsoft hauptsächlich durch ihre Terminologie. Um Mißverständnisse zu vermeiden und einen Vergleich zu den vorherigen Versionen der VC-Oberflächen zu bieten, hier eine Beschreibung der Komponenten.

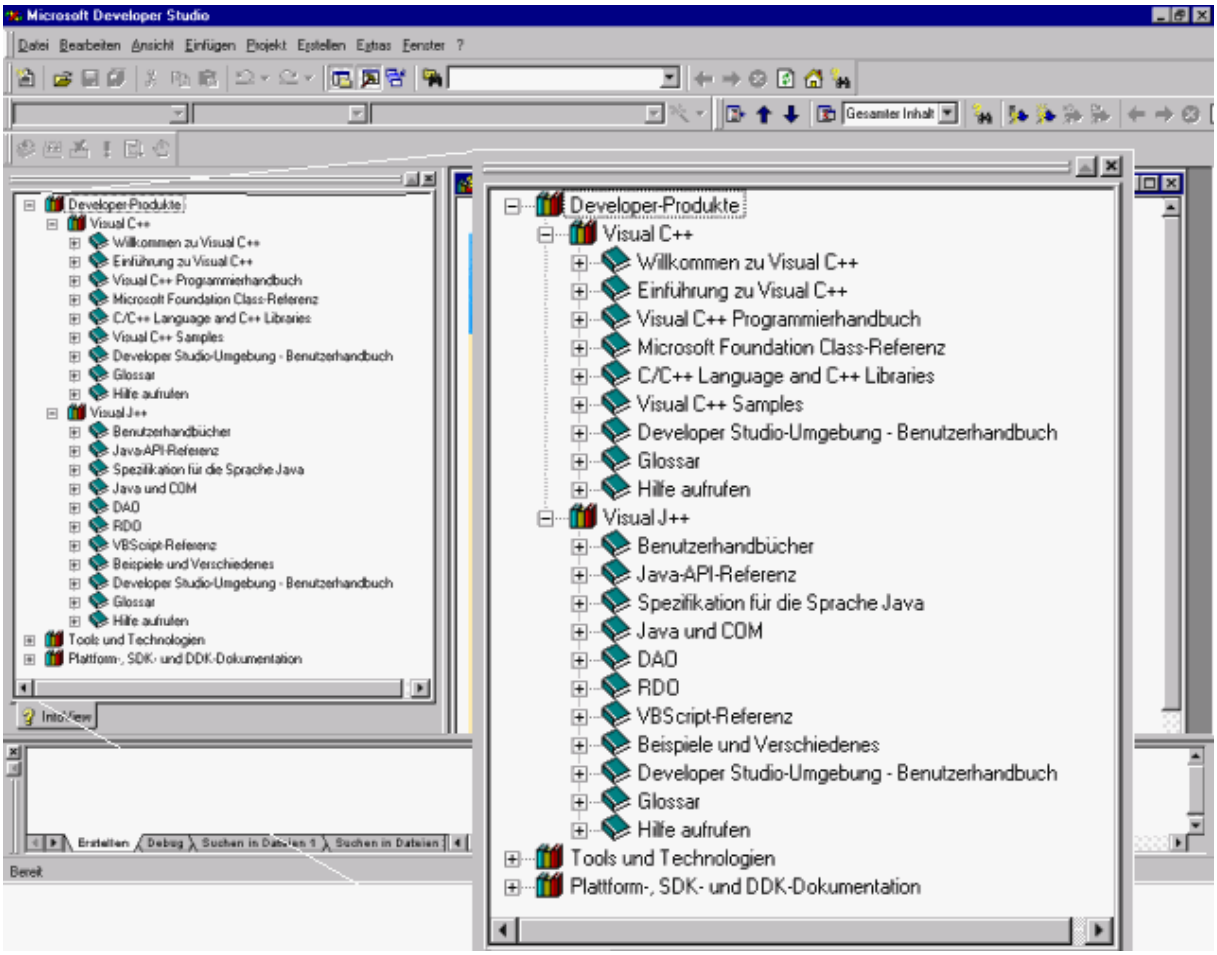


Abbildung 7: Developer Studio mit zwei installierten Programmiersprachen

- Die "Visual Workbench" wurde durch das "**Developer Studio**" ersetzt. Es integriert alle installierten MS-Programmiersprachen unter einer Oberfläche (Abbildung 7: Developer Studio mit zwei installierten Programmiersprachen).
- Ein "Arbeitsbereich" umfaßt mehrere einzelne "Projekte". Ein Projekt ist das kleinste vollständige Individuum, es kann sich dabei z.B. um eine EXE, eine DLL, ein ActiveX-Control oder auch ein Java-Applet handeln. Der Arbeitsbereich faßt zusammengehörige Teile einer Aufgabe zusammen.
- Die kleinsten Einzelteile jedes Projektes in einem Arbeitsbereich sind schließlich die Files mit den Quellcodes (*.cpp, *.c, *.java etc.), die Ressourcen und alle anderen notwendigen Dateien.
- Die bewährte Strategie, alle für die Erzeugung eines Programmes notwendigen Elemente in einem Projekt zusammenzufassen und somit dem Anwender die Arbeit abzunehmen, diese zu erstellen und letztlich korrekt zusammenzufügen, wird konsequent weitergeführt.

Das Developer Studio und seine Einstellungsoptionen

Das **Developer Studio** besitzt kein Menü, in dem Einstellungen gemacht werden können. Die Anpassung an die Wünsche des Anwenders geschieht vielmehr durch einfache Drag & Drop Operationen. Beim ersten Start präsentiert sich die Oberfläche so wie in Abbildung 7: Developer Studio mit zwei installierten Programmiersprachen.

Unterhalb befindet sich das "**Ausgabe**"**-Fenster**. Es ist durch die Tastenkombination **ALT-2** aktivierbar. C++ und Ressourcen Compiler, Linker und viel andere Tools benutzen dieses Fenster, um kurze Nachrichten über den Verlauf des aktuellen Vorganges anzuzeigen.

Abbildung 8: Das Ausgabe-Fenster

Auf der linken Seite erscheint der aktuell **aktive Arbeitsbereich**, der über **ALT-0** angesprochen werden kann. Auf dem Karteireiter findet man zunächst die Handbücher von **Visual C++**, seinen Komponenten sowie den anderen installierten Programmiersprachen und sämtliche Beispiele. Dieses System ist sehr hilfreich beim Auffinden von Informationen und Tips, wie sie während der Entwicklung immer wieder notwendig sind.

Bei **aktivem Arbeitsbereich** wird das Tab um weitere Reiter erweitert, die Übersichten über die Klassen, die Ressourcen und die einzelnen Files bieten.

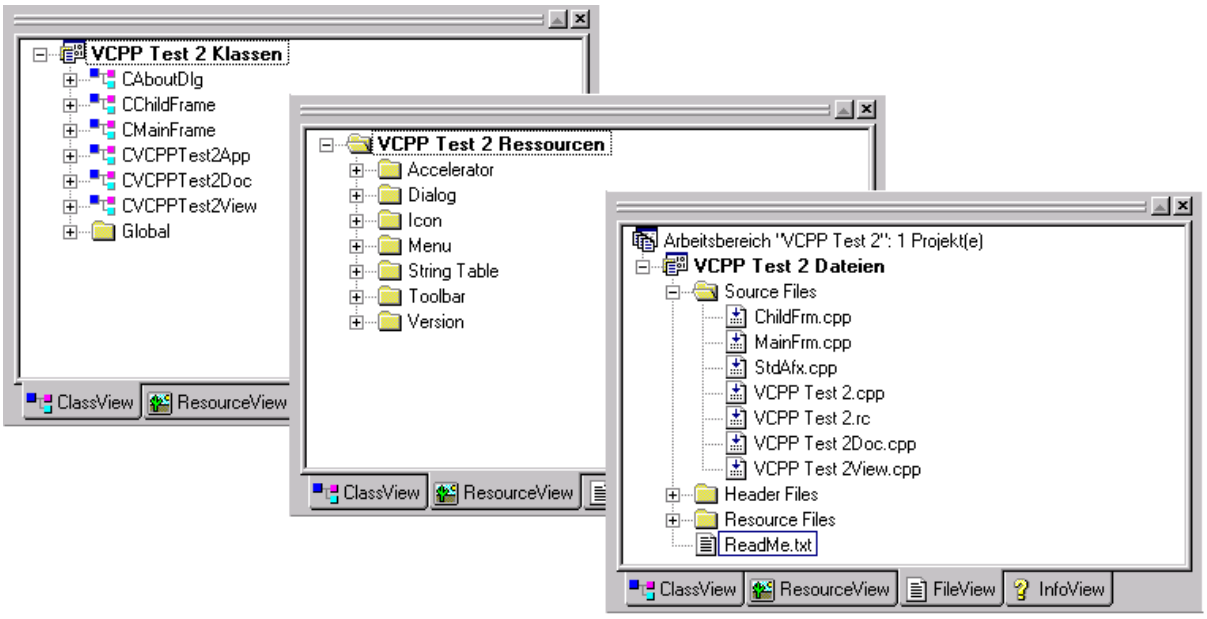


Abbildung 9: Der Arbeitsbereich

Rechts neben dem **Arbeitsbereich** werden weitere Fenster positioniert, je nachdem, was im Arbeitsbereich angewählt wurde: Der **Infoview** mit Informationen aus den Handbüchern, Beispielen und Hilfetexten. Weitere Fenster werden bei Bedarf angezeigt: Dialog-Editor, Debugger etc. Alle Fenster können nach persönlichen Wünschen frei positioniert, vergrößert und verkleinert werden.

Neuerungen in Version 5.0

Developer Studio wurde durch einige neue Merkmale ergänzt und bereits vorhandene Merkmale verbessert. Damit können Sie einfacher als jemals zuvor Anwendungen entwickeln, die höchsten Ansprüchen genügen. **Developer Studio** umfaßt nun auch **Visual J++ 1.1** und **Visual InterDev** sowie **Visual C++** und **MSDN**.

Developer Studio enthält folgende neue Leistungsmerkmale:

Automatisierung und Makros

Verwenden Sie **Visual Basic Skript**, um Routinevorgänge oder sich wiederholende Vorgänge zu automatisieren. Das Aufzeichnen von Makros ermöglicht ein schnelles Schreiben von Skripts. Sie können **Visual Studio** und seine Komponenten als Objekte handhaben. Dies bedeutet, daß Sie Vorgänge, die das Öffnen, Bearbeiten und Schließen von Dokumenten oder die Größenanpassung von Fenstern umfassen, automatisieren können. Mit Hilfe des Objektmodells von **Developer Studio** können Sie außerdem integrierte Add-Ins erstellen.

ClassView

ClassView arbeitet nun sowohl mit Java- als auch mit C++-Klassen. Sie können neue Klassen erstellen, indem Sie **MFC**, **ATL** oder eigene Klassen verwenden. Des weiteren können Sie Ordner verwenden, um Klassen nach Ihren persönlichen Wünschen in **ClassView** zu gruppieren.

ClassView bietet nun außerdem die Möglichkeit des Anzeigens und Bearbeitens von Schnittstellen für **COM-Objekte**, die in **MFC** oder **ATL** implementiert sind.

Benutzerdefinierte Symbolleisten und Menüs

Sie können Symbolleisten und Menüs mit noch mehr Flexibilität als zuvor an Ihre persönliche Arbeitsweise anpassen. So können Sie z.B. folgendes tun:

- eine neue Symbolleiste oder ein neues Menü erstellen.
- eine vorhandene Symbolleiste oder ein vorhandenes Menü individuell anpassen.
- Menübefehle oder Symbolleistenschaltflächen hinzufügen oder entfernen.
- einer Symbolleiste ein Menü hinzufügen.
- einen Menübefehl so verschieben, daß aus ihm eine Symbolleistenschaltfläche entsteht.
- eine Symbolleistenschaltfläche so verschieben, daß aus ihr ein Menübefehl entsteht.
- ein Menü oder eine Symbolleistenschaltfläche auf mehrere Symbolleisten kopieren, so daß das Menü bzw. die Schaltfläche auch noch verfügbar ist, wenn einige Symbolleisten ausgeblendet sind.

Debugger

Sie können eine Verbindung zu laufenden Anwendungen herstellen und diese mit dem Debugger auf Fehler untersuchen. Außerdem können Sie die Makro-Sprachunterstützung (macro language support) zur Automatisierung des Debuggers verwenden.

Direkthilfe

Die Direkthilfe ist eine spezielle Kontexthilfe für Steuerelemente in Dialogfeldern und Fenstern. Zum Aufrufen dieser Hilfe klicken Sie auf die Schaltfläche ? (Symbol für die Direkthilfe) in der oberen rechten Ecke eines Dialogfeldes oder eines Fensters. Der Cursor verwandelt sich in ein Fragezeichen. Anschließend können Sie im Dialogfeld oder Fenster auf ein beliebiges Steuerelement klicken, um ein kurzes Hilfethema zu diesem Steuerelement aufzurufen.

Verbindung zum Internet

Mit **Developer Studio** können Sie sich Seiten im World Wide Web ansehen. Sie können entweder den brandneuen InfoViewer oder Ihren eigenen registrierten Web-Browser verwenden, um sich Microsoft im Web anzusehen. Wenn eine Webadresse (URL) in einem Fenster erscheint, können Sie auf diese Adresse klicken und sich so die entsprechende Webseite anzeigen lassen. Dank dieses Merkmals ist gewährleistet, daß Sie als Benutzerin oder Benutzer von **Visual Studio** immer die neuesten Nachrichten, Dokumentationen, Berichtigungen und/oder Updates erhalten, sobald diese zur Verfügung stehen.

Projektarbeitsbereiche und Projektdateien

Dank eines neuen flexiblen Projektsystems können Sie einen Arbeitsbereich mit unterschiedlichen Projekttypen einrichten. Sie können z.B. einen Arbeitsbereich erstellen, der

ein Projekt aus Visual InterDev sowie ein in J++ erstelltes Applet enthält.

Arbeitsbereichsdateien tragen nun die Erweiterung .DSW (vorher .MDP), Projektdateien die Erweiterung .DSP (vorher .MAK).

Build-Dateien wurden in zwei Typen aufgeteilt: interne Build-Dateien (Erweiterung .DSP) und externe Build-Dateien (Erweiterung .MAK). Interne Build-Dateien werden erstellt, wenn Sie innerhalb von **Developer Studio** ein neues Projekt anlegen oder wenn Sie ein Projekt von einer vorherigen Version konvertieren. Interne Build-Dateien sind nicht kompatibel mit NMAKE. Sie können eine externe, mit NMAKE kompatible Build-Datei erstellen, indem Sie im Menü Projekt auf Makefile exportieren klicken.

Das Erstellen von neuen Arbeitsbereichen und Dateien ist einfacher. Sie können das gleiche Dialogfeld zum Erstellen eines Arbeitsbereichs, eines Projekts oder einer Datei verwenden, und Sie können ein neues Element erstellen und es gleichzeitig in ein Projekt oder einen Arbeitsbereich einfügen. In Visual-InterDev können Sie außerdem Dateiassistenten verwenden, um Gruppen von zusammengehörenden Dateien anzulegen.

Ihre Projekte können aktive Dokumente, wie z.B. Kalkulationstabellen oder Word-Dokument-Dateien enthalten, die Sie bearbeiten können, ohne die integrierte Entwicklungsumgebung von **Visual Studio** verlassen zu müssen. Weitere Informationen über das Hinzufügen von Dokumenten anderen Typs zu Ihrem Projekt finden Sie unter Hinzufügen von Dokumenten zu einem Projekt.

Beim Erstellen eines Arbeitsbereichs legt **Developer Studio** die Datei Arbeitsbereichsname.dsw an, die eine neue Erweiterung trägt. Die Arbeitsbereichsdatei enthält nun nicht mehr Daten, die speziell Ihren lokalen Rechner betreffen. Sie können jetzt folgendes tun:

- die Arbeitsbereichsdatei einem Quellcodeverwaltungsprojekt hinzufügen.
- einen Arbeitsbereich von einem anderen Rechner oder einem Netzwerkverzeichnis kopieren und die Kopie des Arbeitsbereichs direkt öffnen, ohne eine neue Arbeitsbereichsdatei für Ihren lokalen Rechner anlegen zu müssen.

Ressourcen Editoren

Verwenden Sie in **Visual C++** die Assistentenleiste mit den Dialogfeldern, um den Grafikelementen Ihres Programms Quellcode zuzuordnen.

Der Zugriffstasteneditor, der Binäreditor, der Dialogeditor und der Zeichenfolgeneditor unterstützen die Suche nach Zugriffstasten, ASCII-Zeichenfolgen, hexadezimalen Bytes, Steuerelement-IDs oder Bezeichnungen sowie nach speziellen Zeichenfolgen mit Hilfe des Befehls Suchen.

Falls Sie an Zugriffstasten, Dialogfeldern, Menüs oder Zeichenfolgen arbeiten und an mehreren Elementen die gleiche Änderung vornehmen möchten, können Sie dies tun, indem Sie alle zu ändernden Elemente markieren und dann im Menü Ansicht auf Eigenschaften klicken. Falls Sie mehrere Elemente markiert haben, erscheint nach Eingabe des Befehls Eigenschaften eine Eigenschaftenseite namens "**Mehrfachauswahl Eigenschaften**". Die Änderungen, die Sie auf dieser Seite vornehmen, gelten für jedes Element, und wenn Sie eine Änderung rückgängig machen möchten, müssen Sie den Befehl Rückgängig nur einmal wählen.

Texteditor

Der Texteditor kann Headerdateien ohne Erweiterung mit der entsprechenden Syntaxeinfärbung versehen. Weitere Informationen hierzu finden Sie unter Einstellen der Syntaxeinfärbung.

Wenn Sie klar zwischen dem Steuerelement- und dem Textbereich eines Quellcodefensters unterscheiden möchten, sollten Sie die Farbe für die Markierungsleiste individuell anpassen. Dies ist insbesondere dann nützlich, wenn Sie Quellcodedateien mit geringer vertikaler

Ausrichtung der Elemente, wie z. B. HTML-Dateien, bearbeiten möchten.

Der Befehl Suchen in Dateien unterstützt zwei separate Ausgabebereiche. Dieses Merkmal ermöglicht es Ihnen, die Ausgabe einer vorherigen Suche beizubehalten.

Assistentenleiste

Die Assistentenleiste wurde stark erweitert und arbeitet nun mit Visual J++. Jedes Sprachpaket kann die Assistentenleiste zur Überwachung seiner Klassen oder sonstiger Objekte verwenden, die dem Parser der Assistentenleiste angezeigt werden.

Assistenten

Microsoft hat eine ganze Reihe neuer Assistenten für Sie entworfen, darunter Assistenten für die neuen integrierten Pakete Visual J++ und Visual-InterDev (verfügbar, falls Sie diese Pakete installiert haben). Mit Hilfe dieser Assistenten können Sie Dateien, Steuerelemente und neue Projekttypen erstellen.

C++-Compiler, Linker- und Ressourcencompiler

C++-Compiler, Linker-und Ressourcencompiler sind in der Oberfläche so integriert, daß ihre Aufrufe für den Anwender transparent erfolgen. Die Entwicklungsoberfläche ordnet den jeweiligen Files den jeweils passenden Compiler zu und linkt das Projekt schließlich zu einer lauffähigen EXE, einer DLL oder einem ActiveX.

Codeview-Debugger

Der Debugger ist eines der wichtigsten Tools der Entwicklungswerkzeuge. Bekanntermaßen verbringt der Entwickler einen Großteil seiner Zeit damit, Code zu debuggen. Daher ist ein leistungsfähiges und einfach zu handhabendes Werkzeug unabdingbar. Auch der Debugger ist fest in das **Developer Studio** integriert.

Auf einem Rechner, auf dem **Visual C++** installiert ist, besteht mit ihm die Möglichkeit, jede andere Applikation zu debuggen. Tritt in einer Anwendung ein Fehler auf, kann diese den Debugger starten und ermöglicht so eine Fehlersuche. Vorbedingung, damit nicht nur disassemblierter Maschinencode sichtbar wird, ist die Verfügbarkeit von Debugging-Symbolen. Diese können mittels des Setup für NT- Systemsymbole zumindest für das Betriebssystem und einige MS-Programme installiert werden (vergl.: Die Komponenten von Visual C++).

AppWizard

Ziele des Kapitels:

- Beschreibung der Einrichtung und Wartung von Arbeitsbereichen und Projekten
- Bereitstellung zusätzlicher Header- und Ressourcedateien
- Erstellen und Compilierung eines Minimalprogrammes mit dem **AppWizard**
- erste Schritte in der manuellen Programmierung und Anpassung des Minimalprogrammes

Einrichten und Verwalten von Arbeitsbereiche und Projekten

Über Datei/Neu können Arbeitsbereiche eingerichtet und dann Projekte hinzugefügt werden.

Folgende Abbildung zeigt den Karteireiter, der Ihnen beim Erstellen neuer Arbeitsbereiche assistiert.

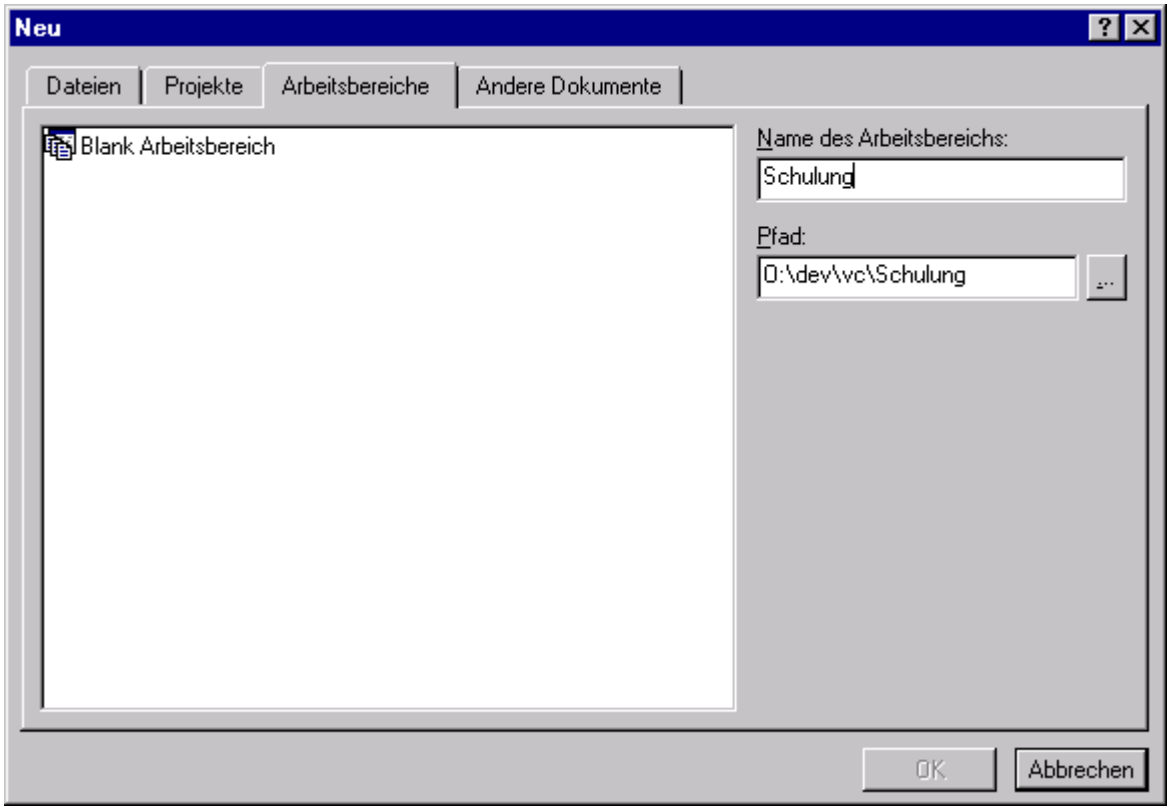


Abbildung 10: Neuen Arbeitsbereich einrichten

In einem Arbeitsbereich faßt man zusammengehörige Projekte zusammen. Ein größere Programmieraufgabe könnte z.B. aus mehreren Hauptprogrammen bestehen, die ihrerseits einige DLLs verwenden (dies unter Umständen auch gemeinsam). Dabei können sogar **C++-**, **Java-** und **Visual BASIC-**Programme miteinander "vermischt" werden. Wenn Sie die Programmgruppe von **Visual C++** betrachten, sehen Sie auch mehrere Programme, die getrennt erstellt wurden, jedoch zusammen zum Arbeitsbereich "**Visual C++ 5.0**" gehören.

Eine saubere Aufteilung der Aufgaben in verschiedene Arbeitsbereiche sorgt für einen aufgeräumten elektronischen Arbeitsplatz und erleichtert die Erfüllung der Programmieraufgaben. Die physikalische Ablage sämtlicher Dateien in genau definierten Unterverzeichnissen erlaubt außerdem sichere Backups und die Möglichkeit, die Sourcecodes vollständig weiterzugeben.

Aufgabe 1: Erstellen Sie den Arbeitsbereich "Schulung"

*Erstellen Sie einen neuen Arbeitsbereich "**Schulung**" in Ihrem Verzeichnis. Wir werden diesen Arbeitsbereich im Verlauf des Seminars für weitere Aufgaben benötigen.*

Vorgehensweise

Klicken Sie auf Datei/Neu. Der Karteireiter "**Neu**" erscheint (siehe Abbildung 10: Neuen Arbeitsbereich einrichten). Aktivieren Sie das Tab "**Arbeitsbereiche**". Geben Sie als Pfad Ihr persönliches Arbeitsverzeichnis an. Falls Sie den vollständigen Pfad nicht kennen, können Sie über das Button rechts ein Suchfenster aktivieren. Geben Sie nun als Arbeitsbereich "**Schulung**" an und aktivieren Sie "**Blank Arbeitsbereich**" (ja, das heißt wirklich so...). Das "OK"-Button wird nun freigegeben. Bestätigen Sie die Angaben durch Klicken auf "OK". Der neue Arbeitsbereich ist jetzt angelegt und aktiv.

Um den Arbeitsbereich zu speichern...
...klicken Sie auf "Datei/Arbeitsbereich" speichern

Um den Arbeitsbereich zu schließen...
...klicken Sie auf "Datei/Arbeitsbereich" schließen

Um den Arbeitsbereich später wieder zu aktivieren...
...klicken Sie auf "Datei/Arbeitsbereich" öffnen.
...klicken Sie auf "Datei/Zuletzt geöffnete Arbeitsbereiche" und wählen Sie den gewünschten Bereich aus.

- ***Achtung:** Verwenden Sie im Dateinamen (und am besten auch Anwendungsnamen) keine Umlaute und Sonderzeichen! **Visual C++** findet sonst Teile des Projektes nicht.*

Automatisches Erstellen von Applikationen

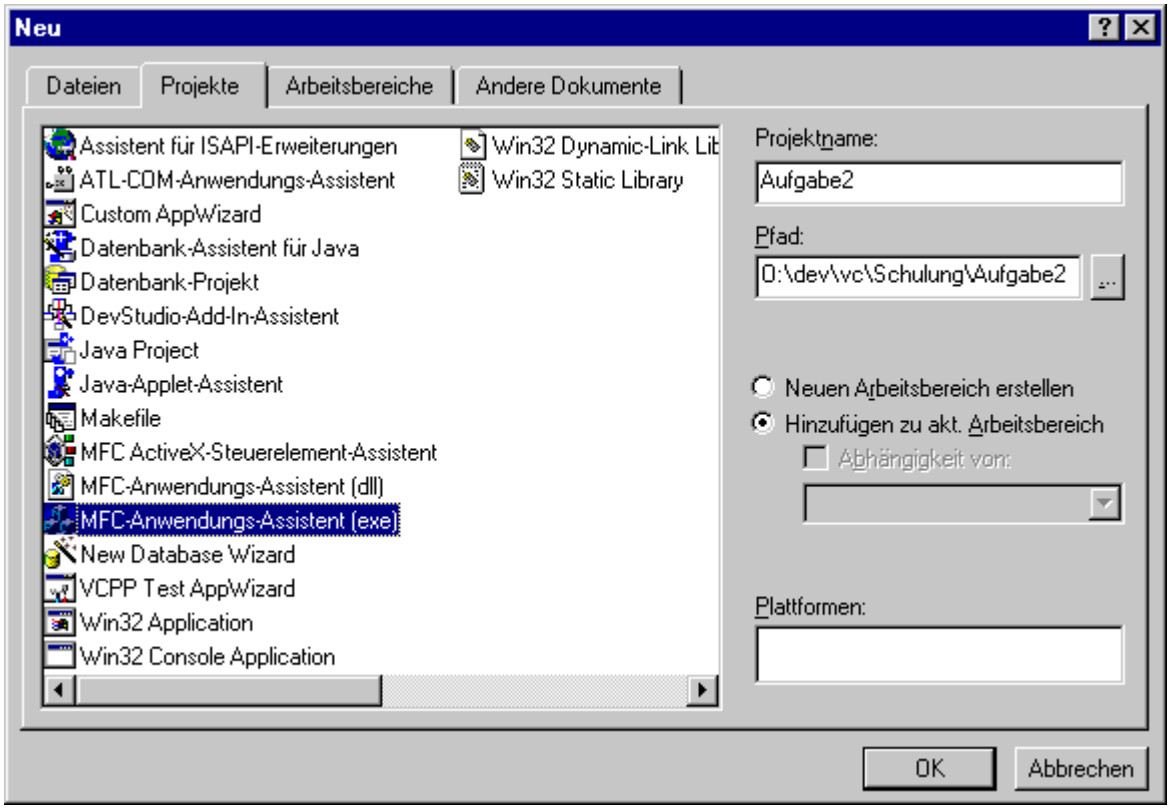


Abbildung 11: Neues Projekt einrichten

Mit dem oben abgebildeten Tabellenreiter ist es möglich, verschiedenste Projekte zu erstellen. Interessant für dieses Seminar sind speziell die Assistenten für **MFC-**Anwendungen als EXE und DLL.

Aufgabe 2: Legen Sie das Projekt "Aufgabe2" an

*Erstellen Sie ein neues Projekt "**AppWizard-Übung**" im Arbeitsbereich "**Schulung**" mit Hilfe des **MFC-**Anwendungsassistenten für EXE-Dateien.*

Vorgehensweise

Klicken Sie auf "**Datei/Neu**". Der Karteireiter "**Neu**" erscheint (siehe Abbildung 10: Neuen Arbeitsbereich einrichten). Aktivieren Sie das Tab "**Projekte**". Geben Sie als Pfad Ihr persönliches Arbeitsverzeichnis an. Falls Sie den vollständigen Pfad nicht kennen, können Sie über das Button rechts ein Suchfenster aktivieren. Geben Sie nun als Projektname "**AppWizard-Übung**" an und aktivieren Sie "MFC-Anwendungsassistent (EXE)".

Nun erscheint der Assistent, der Sie in mehreren Schritten zu Ihrer Minimalanwendung führt.

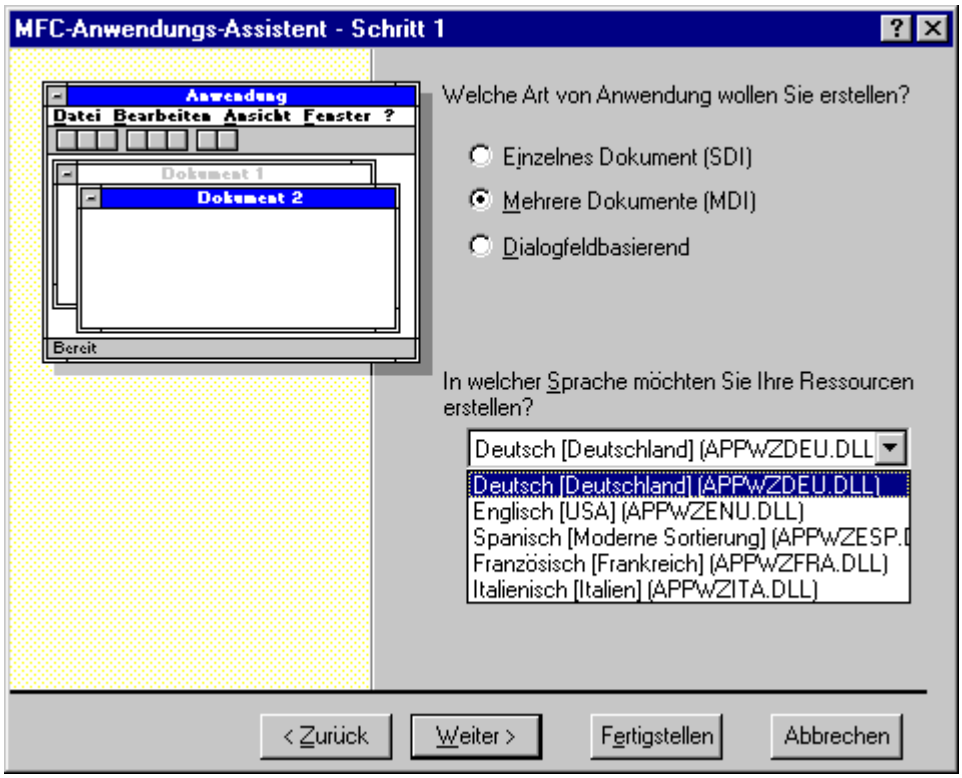


Abbildung 12: MFC-Anwendungsassistent

Der Assistent hilft ihnen, Ihre Anwendung zu generieren. Bitte prüfen Sie alle Änderungen, die Sie auf den sechs Schritten durchführen, sorgfältig. Sie können mit "Zurück" und "Weiter" jederzeit vorher gemachte Angaben revidieren, ist die Anwendung jedoch einmal generiert, sind die meisten Vorgaben bindend und können nicht mehr verändert werden.

Schritt 1

Wählen Sie zunächst die Art der Anwendung: **SDI**, **MDI** oder **dialogbasierend**.

Eine **SDI** (**Single Document Interface**) Anwendung erlaubt immer nur ein einziges offenes Fenster neben dem Rahmenfenster der Anwendung selbst. Dies vereinfacht die programmiertechnische Handhabung, da grundsätzlich nur ein Objekt offen ist und behandelt werden will.

Eine **MDI-** (**Multi Document Interface**) Anwendung erlaubt die parallele Existenz mehrerer Fenster auf dem Anwendungsrahmen. Microsoft Word ist ein solches Programm, es ist möglich, mehrere Texte gleichzeitig zu laden und zu bearbeiten. Das **Developer Studio** ist ebenfalls eine **MDI-** Anwendung. Vom programmtechnischen Standpunkt her ist eine **MDI-** Anwendung etwas schwieriger zu handhaben, jedoch helfen objektorientierte Programmierung (**OOP**) und die Assistenten mit **MFC** (der **Microsoft Foundation Class**) über die übelsten Probleme hinweg.

Eine dialogbasierende Anwendung ist die einfachste Version eines Windows-Programmes. Hier existiert immer nur ein einziges Fenster, das Rahmenfenster selbst. Auf ihm spielen sich alle Operationen ab (ein Beispiel: das Notepad). Für einfache Anwendungen ist dies die beste Lösung, da sie den Benutzer direkt in das relevante Fenster führt.

Wählen Sie außerdem die gewünschte Sprache aus. Standardmäßig stehen Deutsch, Englisch, Spanisch, Französisch und Italienisch zur Verfügung. Diese Einstellung beeinflußt die vom Programm automatisch generierten Infos (Fehlermeldungen, Bestätigungen) und die verwendeten Zeichensätze.

Schritt 2

Im zweiten Schritt kann die Datenbankunterstützung gewählt werden. Wir benötigen für diese Anwendung keine, daher lassen wir die Einstellungen wie sie sind.

Die möglichen Optionen zur Datenbankunterstützung sind:

- Keine Unterstützung.
- Nur die Header einbinden: Nur die zu Datenbankzugriffen notwendigen Header werden eingebunden, alle anderen Aufgaben bleiben dem Programmierer überlassen.
- Datenbankanwendung ohne Dateiunterstützung: Die für Datenzugriffe notwendigen Header und Klassen werden eingebunden. Das Programm als **SDI**-**Anwendung** ohne Datei Laden/Speichern- Funktionen generiert.
- Datenbankanwendung mit Dateiunterstützung: Die für Datenzugriffe notwendigen Header und Klassen werden eingebunden. Das Programm als **SDI**- oder **MDI**-**Anwendung** mit Datei Laden/Speichern-Funktionen generiert.

Wenn Sie Datenbankunterstützung anwählen, können Sie ferner gleich eine Datenquelle und die zugehörige Tabelle definieren. Sie haben dabei die Auswahl unter einer **ODBC Datenquelle** (mittels Treibern) oder direkt über **DAO** (**Data Access Objects**), wobei dann nur Microsoft Access kompatible Datenbanken unterstützt werden.

Schritt 3

Auch im dritten Schritt gibt es keine Option, die wir benötigen. Bitte deaktivieren Sie nur "**ActiveX Steuerelemente**", denn wir werden die **ActiveX-**Unterstützung nicht benötigen.

Die möglichen Optionen sind hier:

- **Keine ActiveX Unterstützung:** Wählen Sie diese Option, wenn das Programm keine **ActiveX-** Unterstützung (früher OLE) haben soll. Standardmäßig erstellt der Anwendungsassistent Programme ohne **ActiveX-**Unterstützung.
- **Container:** Wählen Sie diese Option, wenn das Programm verknüpfte und eingebettete Objekte enthalten soll.
- **Miniserver:** Wählen Sie diese Option, wenn das Programm die Funktion zur Erstellung und Verwaltung von Verbunddokument-Objekten erhalten soll. Beachten Sie, daß **Miniserver** nicht eigenständig laufen können und nur eingebettete Elemente unterstützen.
- **Vollserver:** Wählen Sie diese Option, wenn das Programm die Funktion zur Erstellung und Verwaltung von Verbunddokument-Objekten erhalten soll. **Vollserver** können eigenständig laufen und unterstützen sowohl verknüpfte als auch eingebettete Elemente.
- **Container und Server:** Wählen Sie diese Option, wenn das Programm sowohl **Container** als auch **Server** sein soll. Ein **Container** ist ein Programm, das eingebettete oder verknüpfte Elemente in seine eigenen Dokumente übernehmen kann. Ein **Server** ist ein Programm, das Automatisierungselemente für Containerprogramme erstellen kann.

Für die Optionen 2 - 5 steht zusätzlich die Möglichkeit zur Verfügung, die in **ActiveX Controls** erstellten Dokumente über die Serialisierung von **MFC** zu speichern. Dazu muß "**ActiveX Dokumentserver**" aktiv sein.

- **Unterstützung für Verbunddateien:** Wählen Sie diese Option, um die Dokumente des Container-Programms serienmäßig mit dem Verbunddateiformat zu erstellen. Im Verbunddateiformat wird ein Dokument, das ein oder mehrere Automatisierungsobjekte enthält, in eine Datei gespeichert, wobei der Zugriff auf die Dateien für die einzelnen Objekte nach wie vor möglich ist. Mit dieser Option können auf Anforderung die ursprünglichen Daten des Objekts geladen und inkrementelle Speicherungen in die ursprünglichen Daten vorgenommen werden.
- **Unterstützung für Automatisierung:** Durch die Aktivierung von **Automatisierung** können Sie mit dem Programm Objekte bearbeiten, die in einem anderen Programm implementiert

sind oder das Programm **Automatisierungsclients** zur Verfügung stellen.

- **ActiveX Steuerelemente:** Wählen Sie diese Option, wenn das Programm **ActiveX Steuerelemente** verwenden soll. Das Programm unterstützt standardmäßig **ActiveX Steuerelemente**. Wenn Sie diese Option nicht wählen, jedoch zu einem späteren Zeitpunkt **ActiveX Steuerelemente** in das Projekt einfügen möchten, müssen Sie der **Member-Funktion InitInstance** des Programms einen Aufruf zu **AfxEnableControlContainer** hinzufügen.

Schritt 4

Auf diesem Schritt aktivieren Sie bitte "**Kontextabhängige Hilfe**", damit automatisch eine Hilfedatei generiert wird. Alle anderen Optionen bleiben unverändert.

- **Andockbare Symbolleiste:** Wählen Sie diese Option, um dem Programm eine Symbolleiste hinzuzufügen. Die Symbolleiste enthält Schaltflächen zum Erstellen eines neuen Dokuments, zum Öffnen und Speichern von Dokumentdateien, zum Ausschneiden, Kopieren, Einfügen und Drucken von Text, zum Anzeigen des Dialogfelds Info und zum Eingeben des Hilfemodus. Durch Aktivieren dieser Option werden auch Menübefehle hinzugefügt, mit denen die Symbolleiste angezeigt oder ausgeblendet werden kann. Standardmäßig verfügen Programme über eine andockbare Symbolleiste.
- **Statusleiste zu Beginn:** Wählen Sie diese Option, um dem Programm eine Statusleiste hinzuzufügen. Die Statusleiste zeigt automatisch den Status der **Feststell-**, der **NUM**- und der **Rollen-Taste** sowie eine Nachrichtenzeile an, in der Hilfetext für Menübefehle und Symbolleistenschaltflächen angezeigt wird. Durch Aktivieren dieser Option werden auch Menübefehle hinzugefügt, mit denen die Statusleiste angezeigt oder ausgeblendet werden kann. Standardmäßig verfügen Programme über eine Statusleiste.
- **Drucken und Seitenansicht:** Wählen Sie diese Option, wenn der Anwendungsassistent den Code für die Behandlung der Befehle zum Drucken, für die Druckereinrichtung und die Seitenansicht generieren soll, indem er Member-Funktionen in der Klasse **CView** der **MFC**-Bibliothek aufruft. Der Anwendungsassistent fügt dem Menü des Programms auch Befehle für diese Funktionen hinzu. Standardmäßig unterstützen die Programme die Druckfunktion.
- **Kontextabhängige Hilfe:** Wählen Sie diese Option, wenn der Anwendungsassistent einen Satz von Hilfedateien für kontextabhängige Hilfe generieren soll. Für die Unterstützung der Hilfe ist der Hilfecompiler erforderlich. Sie können den Hilfecompiler nachträglich installieren, indem Sie Setup erneut ausführen.
- **3D Steuerelemente:** Wählen Sie diese Option, wenn die Oberfläche des Programms eine dreidimensionale Schattierung erhalten soll. Die Programme haben standardmäßig eine dreidimensionale Schattierung.
- **MAPI Unterstützung:** Wählen Sie diese Option, wenn das Programm Mail-Nachrichten erstellen, bearbeiten, übertragen und speichern soll.
- **Windows Sockets:** Wählen Sie diese Option, wenn das Programm Windows Sockets unterstützen soll. Windows Sockets ermöglichen es Ihnen, Programme zu schreiben, die über **TCP/IP**-Netzwerke kommunizieren.
- **Wie viele Dateien sollen in der Liste der zuletzt verwendeten Dateien angezeigt werden:** Gibt die Anzahl der Dateien an, die in der Liste der zuletzt verwendeten Dateien angezeigt werden sollen. Standardmäßig beträgt die Anzahl 4.
- **Weitere Optionen:** Erlaubt die Definition der standardmäßigen Dateiendung bei Speicheroperationen, den Typ des Dokuments (Beispiel bei Notepad: Erweiterung .txt, Typ "Text") sowie Zeichenfolgen für die Beschriftung des Hauptfensters, Namen des Dokumenttyps (wird beim Speichern und Laden eingeblendet). Außerdem "Datei Neu" Name (dieser Wert wird verwendet, wenn neue Dateien angelegt werden).

Schließlich können auch die Fensterstile definiert werden.

Schritt 5

Hier kann angegeben werden, ob Kommentare im Programm erzeugt werden. Dies ist natürlich sehr sinnvoll. Ferner kann festgelegt werden, ob die **MFC-**Bibliothek statisch oder dynamisch eingebunden werden soll. Damit ist gemeint, daß die Laufzeitbibliothek entweder fest an unsere Anwendung angelinkt wird oder als DLL zur Laufzeit eingebunden. Der Weg über die DLL hat den Vorteil, daß die Laufzeitbibliotheken nur einmal auf der Platte "herumliegen" müssen, da sie von allen Anwendungen, die sie benötigen, gemeinsam verwendet werden.

Schritt 6

Visual C++ zeigt nun die Klassen und zugehörigen Dateien an, die generiert werden sollen. Es ist zwar möglich, diese jetzt zu ändern, aber wir verzichten darauf, um Verwirrungen zu vermeiden. Hier etwas zu ändern ist nur dann ratsam, wenn man genau weiß, was man vorhat.
Mit dem Klick auf "Fertigstellen" sollte in etwa folgendes Fenster erscheinen:

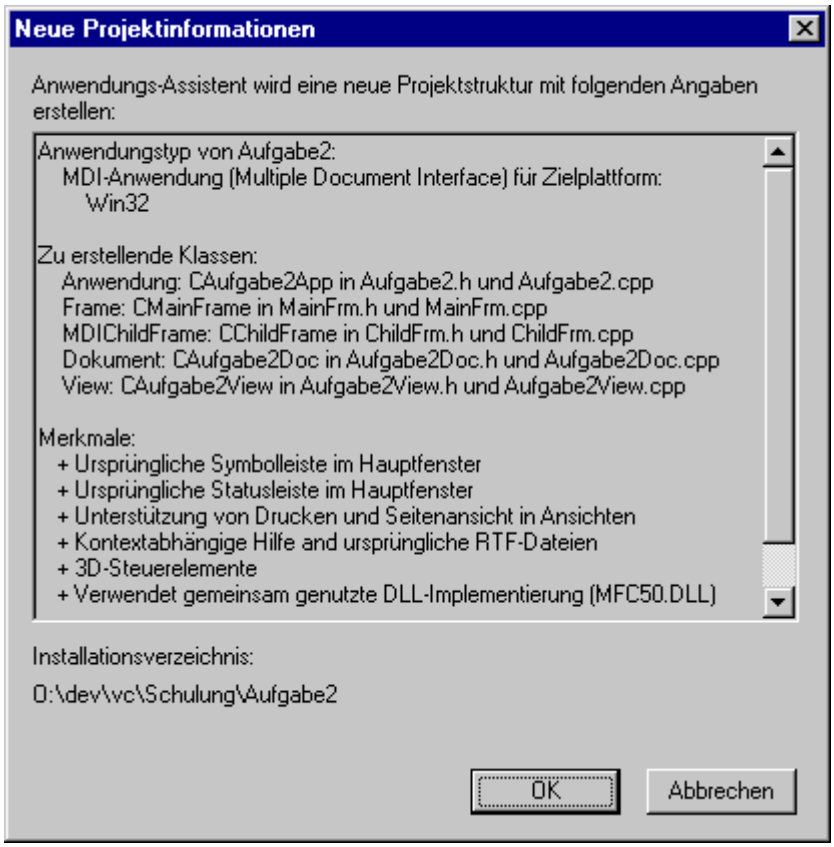


Abbildung 13: Neue Projektinformationen

Klicken Sie auf "OK", und ihre Anwendung wird generiert.

Bereitstellen der Header-, Ressource- und sonstige Dateien

Im vorhergehenden Abschnitt wurde beschrieben, wie eine Minimalanwendung mit Hilfe des **AppWizard** erstellt werden kann. Dieser erzeugt alle notwendigen Quellfiles: Header, Ressourcen, Online-Hilfen. Natürlich ist es damit nicht getan. Um zusätzliche Ressourcen einzubinden - eigene Icons, Bitmaps etc. oder APIs und Bibliotheken von Fremdanbietern zu verwenden, müssen diese ins Projekt eingefügt werden.

Hinzufügen neuer Dateien

Um neue Dateien hinzuzufügen, aktivieren Sie "Projekt/Dem Projekt hinzufügen/Neu". Es erscheint der bereits bekannte Tabellenreiter. Wählen Sie den Typ der Datei aus, die sie erstellen wollen (Bitmap, C/C++ Header, C++ Quellcode etc.). Die Datei wird automatisch im richtigen Ordner erstellt und in den integrierten Editor geladen.

Hinzufügen bereits vorhandener Dateien

Über "Projekt/Dem Projekt hinzufügen/Dateien" aktivieren Sie ein Suchfenster, in dem Sie eine oder mehrere Dateien auswählen können. Diese Dateien werden automatisch im richtigen Ordner erstellt und in den integrierten Editor geladen.

Die zweite Möglichkeit ist, das Kontextmenü des Arbeitsbereichs zu verwenden:

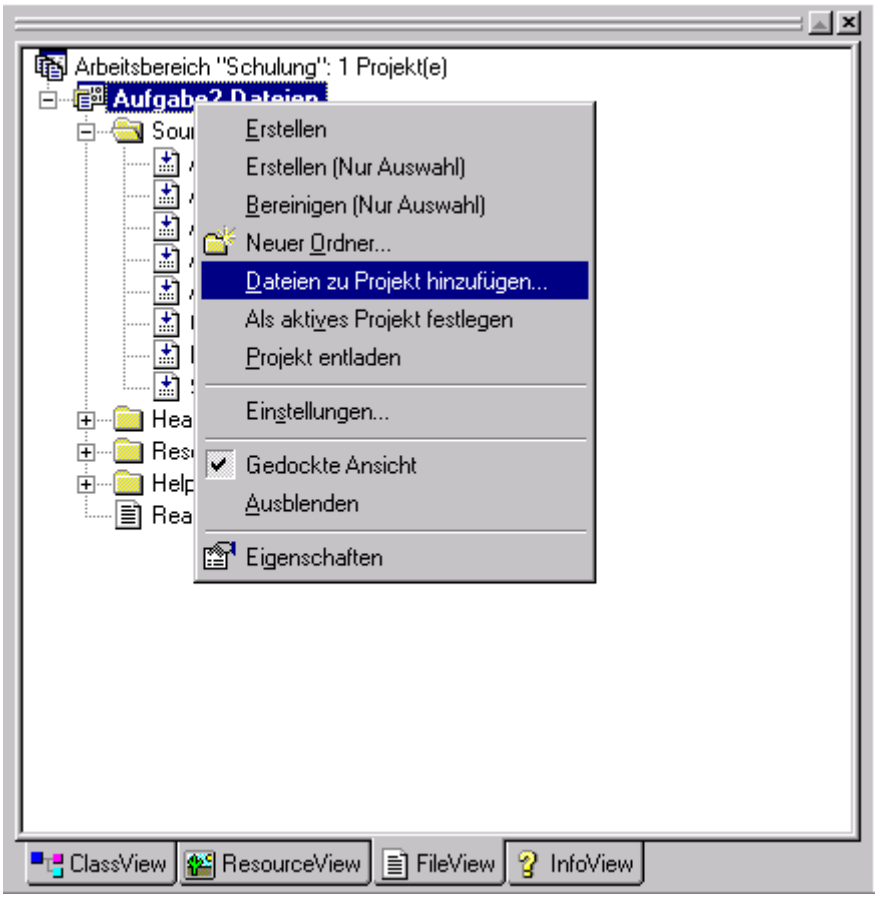


Abbildung 14: Kontextmenü des Arbeitsbereichs

Sie erhalten dabei dasselbe Ergebnis wie über die Menüleiste.

Klicken Sie mit der rechten Maustaste auf einen der Ordner, erscheint ein etwas kürzeres Menü, mit dem Sie Dateien gezielt einem bestimmten Ordner hinzufügen können:

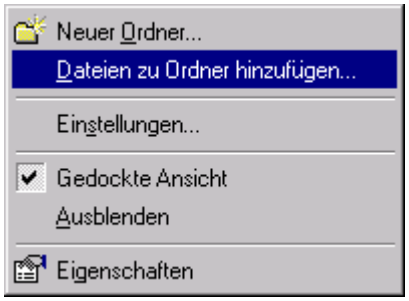


Abbildung 15: Popup Menü für Ordner

In keinem Fall werden die Dateien bewegt oder kopiert, sie bleiben also an dem Ort, von dem aus sie eingefügt wurden. Dies ist immer dann sinnvoll, wenn sie gemeinsam genutzt werden sollen und somit die Gefahr verringert wird, daß bei Wartungsarbeiten und Änderungen Kopien von ihnen übersehen werden.

Sollten Sie es wünschen, eine private Kopie der hinzuzufügenden Files zu verwenden, so kopieren Sie sie zunächst in das Verzeichnis des Arbeitsbereiches, bevor Sie sie hinzufügen.

Löschen von Dateien aus dem Projekt

Nicht mehr benötigte Dateien und Ordner können einfach markiert und mit **ENTF** aus dem Projekt gelöscht werden. *Dabei wird die Datei nur aus dem Projekt entfernt, aber nicht physikalisch gelöscht!* Sollten Sie die Datei ganz von Ihrer Platte löschen wollen, müssen Sie dies nach dem Entfernen "von Hand" mittels Explorer oder vom DOS-Prompt aus machen.

Die Popup-Menüs des Arbeitsbereichs

Der Arbeitsbereich besitzt zwei Popup-Menüs, die wichtige Funktionen bereitstellen. Deshalb werden diese jetzt hier nochmals im Detail besprochen. Wird auf einem Projektnamen die rechte Maustaste betätigt, dann erscheint ein Popup-Menü mit folgenden Optionen:

- **Erstellen:** Erstellt das aktuelle Projekt.
- **Erstellen (nur Auswahl):** Erstellt den ausgewählten Ordner bzw. die ausgewählten Files des aktuellen Projekts.
- **Neuer Ordner:** Erzeugt einen neuen Ordner.
- **Dateien zu Projekt hinzufügen:** Neue Dateien hinzufügen.
- **Als aktives Projekt festlegen:** Legt (bei mehreren Projekten im Arbeitsbereich) das aktuelle als aktiv fest.
- **Projekt entladen:** Entfernt das aktuelle Projekt aus dem Arbeitsbereich.
- Wird auf einem Ordner die rechte Maustaste betätigt, dann erscheint ein kürzeres Popup-Menü mit folgenden Optionen:
- **Neuer Ordner:** Erzeugt einen neuen Ordner.
- **Dateien zu Projekt hinzufügen:** Neue Dateien hinzufügen (siehe „Bereitstellen der Header-, Ressource- und sonstige Dateien“),

Compilierung des neuen Programmes

Es ist an der Zeit, das Programm zu compilieren. Es gibt mehrere Möglichkeiten, diesen Prozess anzustoßen. Zum einen kann man den Menüpunkt "**Erstellen**" anwählen und dort "**Aufgabe2.exe neu erstellen**" bzw. "**Alles neu erstellen**" anwählen. Im ersten Fall wird nur das aktive Projekt neu generiert. Im anderen Fall werden alle im Arbeitsbereich befindlichen Projekte in der nach ihren Abhängigkeiten richtigen Reihenfolge erstellt. Ist ein Quellfile aktiviert (im Falle Abbildung 16: Das Menü "Erstellen" ist es Aufgabe2.hpj), so kann man auch nur dieses eine File compilieren lassen.

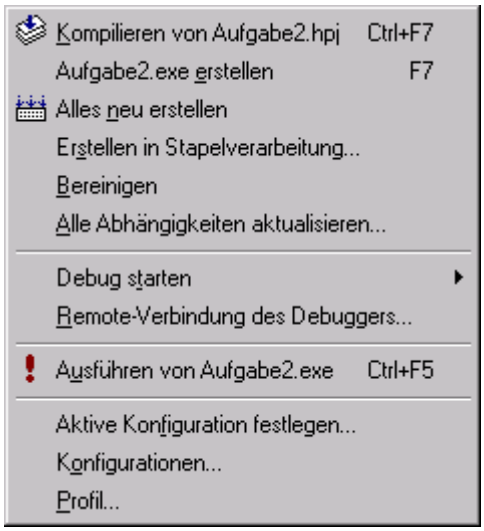


Abbildung 16: Das Menü "Erstellen"

Alternativ kann man auch die Shortcuts Ctrl-F7 oder F7 respektive verwenden.
Mit der Maus am schnellsten ist man jedoch über den Toolbar:

Abbildung 17: Menüleiste Arbeitsbereich

Diese Buttons lösen (von links nach rechts) folgende Vorgänge aus:

- Compilieren des aktiven Files
- Erstellen des Projektes
- Abbruch des Compiliervorganges (nachdem er gestartet wurde)
- Programm ausführen
- Programm im Debugger ausführen oder fortsetzen
- Haltepunkt einfügen/entfernen

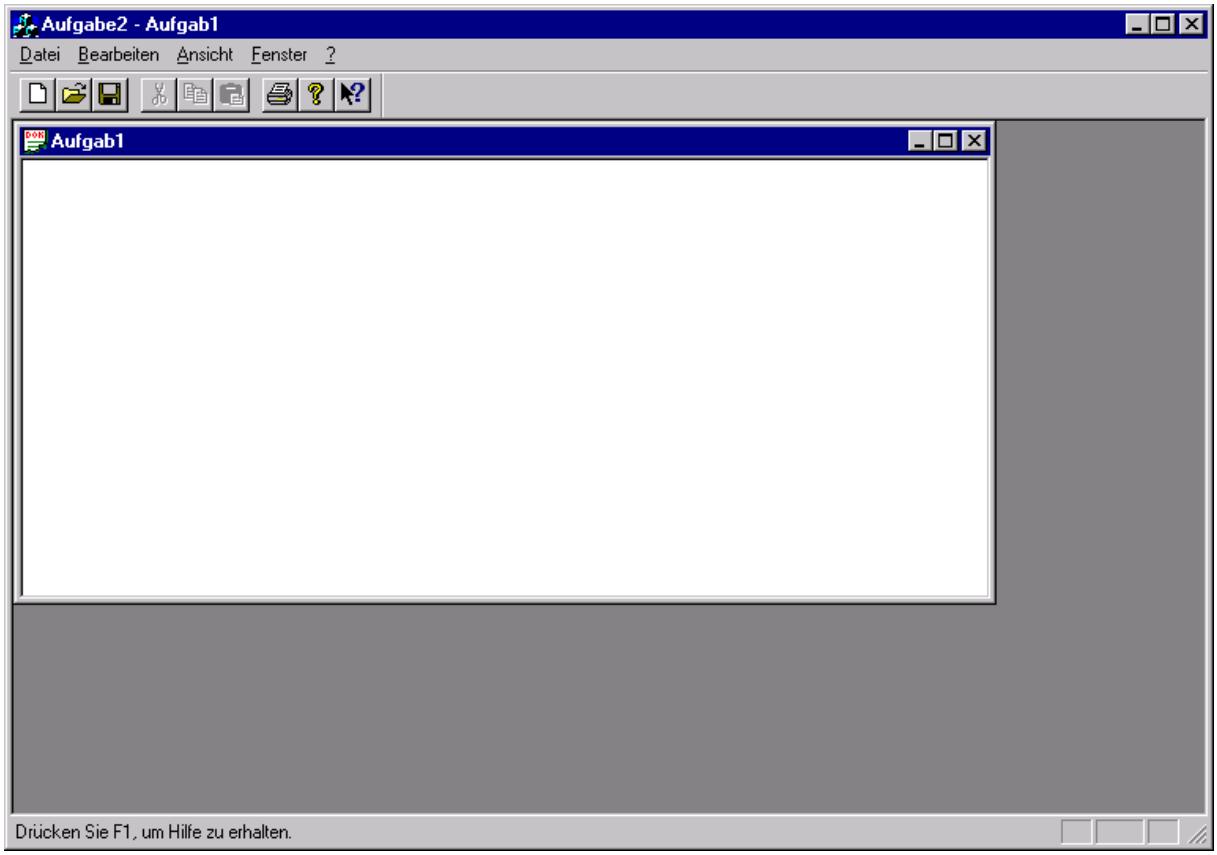
Die Bedeutung von Punkt 6 wird später mit dem Debugger besprochen.

Aufgabe 3: Generieren Sie die ausführbare Datei zum Projekt "Aufgabe 2"

Generieren Sie die ausführbare Datei zum Projekt Aufgabe 2.

Vorgehensweise

Klicken Sie auf Button 2 "Erstellen des Projektes" oder drücken Sie F7. Microsoft **Visual C++** erstellt Ihre Anwendung.



Verwendung des generierten Minimalprogrammes

Abbildung 18: Das generierte Minimalprogramm

Nun haben wir es also geschafft, ein Programm zu generieren. **Visual C++** erstellt für uns alle notwendigen Klassen, Ressourcen und sonstigen Dateien. Was wir danach haben ist ein reines Rohgerüst ohne eigenes Leben, es ist nur sichergestellt, daß alle notwendigen Nachrichten abgearbeitet werden, um Programm und Fenster anzuzeigen.

Aufgabe 4: Starten Sie "Aufgabe2.exe"

Starten Sie das ausführbare Programm, die in Aufgabe 2 erstellt wurde.

Vorgehensweise

Klicken Sie auf Button 4 "Programm ausführen". Microsoft **Visual C++** startet Ihre Anwendung. Sie sehen ein einfaches Programm wie oben, das jedoch auf alle Tastatur- und Mausinteraktionen bereits reagiert.

Nun haben wir ein hübsches Windowsprogramm, das man starten und beenden, aber mit dem man sonst nichts Rechtes anfangen kann. Geben wir ihm also ein wenig Sinn und erstellen wir ein kleines Fenster, das ein paar wenn auch nicht sehr sinnvolle doch wenigstens von uns programmierte Dinge tut.

Jetzt hat unser **AppWizard** eine ganze Menge Code erzeugt, den wir gar nicht brauchen. Deshalb werden wir uns zunächst von diesem entledigen:

Aufgabe 5: Entfernen Sie die überflüssigen Klassen

*Entfernen Sie die überflüssigen Klassen aus dem Projekt. Überflüssig sind **CAufgabe2Doc**, **CAufgabe2View** und **CChildFrame**, die nur benötigt werden, um das in unserem Fall nutzlose weiße **Childfenster** zu erzeugen.*

Vorgehensweise

Markieren Sie im Arbeitsbereich die Header Files "**Aufgabe2Doc.h, Aufgabe2View.h und ChildFrm.h**", und drücken Sie **ENTF**. Im **ClassView** sind die drei Klassen jetzt verschwunden. Danach markieren Sie die Source Files "**Aufgabe2Doc.cpp, Aufgabe2View.cpp** und **ChildFrm.cpp**." und drücken ebenfalls **ENTF**. Schließlich markieren Sie noch unter Ressource Files "**Aufgabe2Doc.ico**" und löschen dieses auch.

Starten Sie die Compilierung - sie werden schnell feststellen, daß sich das Projekt jetzt nicht mehr compilieren läßt. Der Grund dafür ist, daß nach unserer Löschoperation natürlich einige Klassen verschwunden sind.

Die genaue Bedeutung dieser Klassen werden wir später eingehender besprechen.

Aufgabe 6: Räumen Sie das Projekt auf

Löschen Sie die überflüssigen Header-, Source- und Ressourcefiles aus dem Arbeitsverzeichnis.

Vorgehensweise

Wir verwenden den Windows Explorer, um die unnötigen Dateien zu entfernen. Wechseln Sie in das Verzeichnis ...\Schulung\Aufgabe2 und löschen Sie die Dateien, die wir oben aus dem Projekt entfernt hatten, auch physikalisch. Hinweis: "**Aufgabe2Doc.ico**" befindet sich im \res\ Verzeichnis.

Wählen Sie über das Popup-Menü den Arbeitsbereich "Bereinigen", um die jetzt nicht mehr aktuellen Zwischendateien zu löschen.

Nun können auch die Referenzen auf die gelöschten Objekte entfernt werden:
Laden Sie "**Aufgabe2.rc**" und löschen Sie Zeile 70:

```
IDR_AUFGABTYPE ICON DISCARDABLE "res\\Aufgabe2Doc.ico"
```

Laden Sie "Aufgabe2.cpp" und löschen Sie Zeile 8, 9, 10:

```
#include "ChildFrm.h"
#include "Aufgabe2Doc.h"
#include "Aufgabe2View.h"
```

Ebenso Zeile 68 - 79:

```
// Dokumentvorlagen der Anwendung registrieren. Dokumentvorlagen
// dienen als Verbindung zwischen Dokumenten, Rahmenfenstern und Ansichten.
CMultiDocTemplate* pDocTemplate;
pDocTemplate = new CmultiDocTemplate(
   IDR_AUFGABTYPE,
   RUNTIME_CLASS(CAufgabe2Doc),
   RUNTIME_CLASS(CchildFrame),     // Benutzerspezifischer MDI-Child-Rahmen
   RUNTIME_CLASS(CAufgabe2View));

AddDocTemplate(pDocTemplate);
```

Nach diesen Operationen läßt sich das Programm wieder compilieren und starten. Beim "Spielen" mit ihm wird man aber schnell feststellen, daß doch noch etwas faul ist. Der Versuch, ein File zu öffnen endet mit einem Absturz. Das liegt daran, daß jetzt auf das nicht mehr existente weiße Fensterchen zugegriffen wird. Dies gibt uns zwei gute Gelegenheiten, uns

zunächst dem **ClassWizard** zu widmen und schließlich, einmal den **Debugger** auszuprobieren.

ClassWizard

Der **ClassWizard** oder auch Klassenassistent wird mittels **Ctrl-W** oder "**Ansicht/Klassenassistent**" aktiviert. Er steht ihnen bei Änderungen an allen Klassen zur Seite, erlaubt das Verbinden von Windows-Nachrichten mit bestimmten Codefragmenten und ermöglicht somit die Reaktion auf Ereignisse.

Nachrichten und Klassen miteinander verbinden

Am Beginn des Files "Aufgabe2.cpp" befindet sich folgender Code:

```
/////////////////////////////////////////////////////////////////////////////
//
// Caufgabe2App
BEGIN_MESSAGE_MAP(CAufgabe2App, CWinApp)
    //{{AFX_MSG_MAP(CAufgabe2App)
    ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
    // HINWEIS - Hier werden Mapping-Makros vom
    // Klassen-Assistenten eingefügt und entfernt.
    // Innerhalb dieser generierten Quelltextabschnitte NICHTS VERÄNDERN!
    //}}AFX_MSG_MAP
    // Dateibasierte Standard-Dokumentbefehle
    ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
    ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
    // Standard-Druckbefehl "Seite einrichten"
    ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()
```

Die vorletzten beiden ON_COMMAND-Zeilen verbinden die Nachricht, daß die Menüpunkte "Neue Datei" oder "Datei öffnen" angewählt wurden. Sie rufen hier noch direkt die Standardmethoden der Basisklasse auf. Das führt zu Verwicklungen, da wir auf die automatisch generierte Unterstützung verzichtet haben. Hier ist unsere Intervention gefragt: Wir müssen eigenen, angepaßten Code einfügen.

Entfernen Sie zunächst diese drei Zeilen:

```
    // Dateibasierte Standard-Dokumentbefehle
    ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
    ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
```

Das ist erlaubt, da sie nicht zwischen den beiden MSG_MAP Zeilen stehen. Wer hier etwas verändert, erlebt sein blaues Wunder: die Zeilen werden immer wieder neu generiert und jede Änderung gnadenlos überschrieben.

- **Visual C++ 5.0** generiert Code, den es zwischen speziellen Kommentarzeilen plaziert. Diese dürfen nie verändert werden, da sie immer wieder überschrieben werden.

Aufgabe 7: Reparieren Sie das angeschlagene Programm mit dem ClassWizard

Ersetzen Sie die fehlerhaften Aufrufe OnFileNew und OnFileOpen durch eigene.

Vorgehensweise

Starten Sie den **ClassWizard**. Jetzt kann es passieren, daß er erst einmal kräftig losmeckert:

Abbildung 19: ClassWizard Fehlermeldung

Die Meldung ist logisch, denn wir haben die Klasse schließlich inklusive ihren Files gelöscht. Also bringen wir das gleich in Ordnung, indem wir die Referenz aus dem **ClassWizard** löschen:

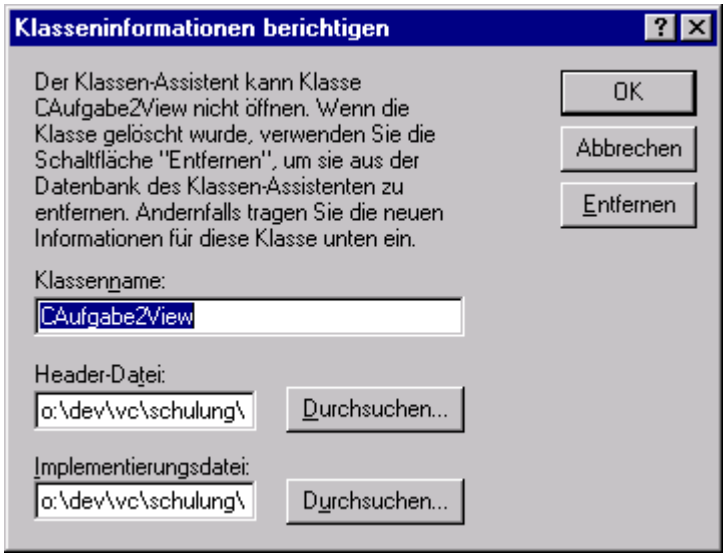


Abbildung 20: Klasseninformationen bereinigen

Wie empfohlen, löschen wir die Klasse einfach mittels "Entfernen". Jetzt bekommen wir den **ClassWizard** erstmals zu Gesicht:

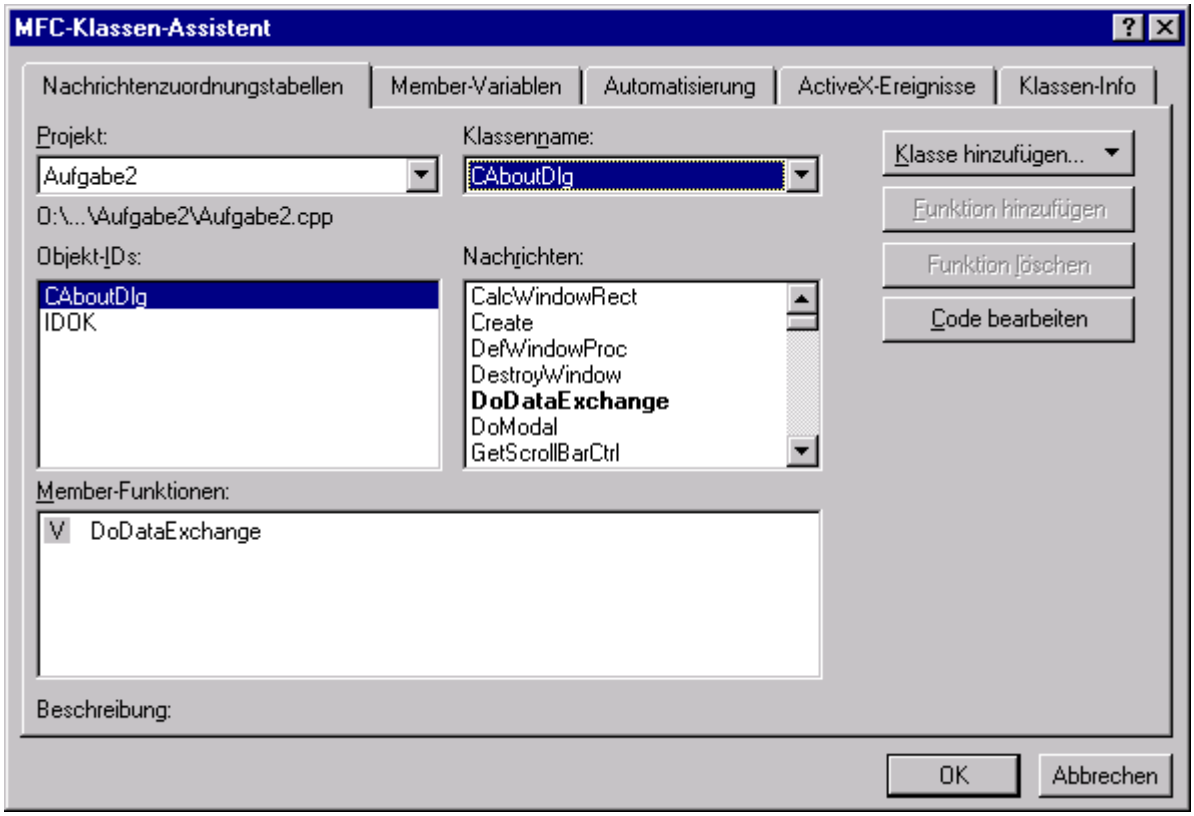


Abbildung 21: Der ClassWizard

Mit dem **ClassWizard** können verschiedene Operationen an den Klassen vorgenommen werden. So können Variablen hinzugefügt werden, ebenso wie Nachrichtenbehandlungen. Über "Klassenname" kann eine andere Klasse ausgewählt werden. Sie werden feststellen, daß die übrigen verflixten Klassen ebenfalls noch herumgeistern. Also wählen wir diese an und löschen sie nach dem bekannten Muster: "**CAufgabe2View**", "**CAufgabeDoc**", "**CChildFrame**". Jetzt haben wir's endlich geschafft, uns aller Leichen zu entledigen.

Wählen Sie nun bitte die Klasse "**Aufgabe2App**" an. Suchen Sie unter Objekt-IDs **ID_FILE_NEW** heraus und wählen Sie unter "Nachrichten" **COMMAND**. Das Button "Funktion hinzufügen" wird jetzt aktiviert. Durch einen Klick darauf erscheint ein kleines Fenster, das nach dem Funktionsnamen fragt. Wir ändern den vorgegebenen Wert "**OnFileNew**" in "**OnMyFileNew**" und klicken auf "**OK**". Diese Änderung ist nicht unbedingt notwendig, aber da es "oben" in der Klasse bereits eine virtuelle Funktion "**OnFileNew**" gibt, mit der wir aber nichts zu tun haben wollen, ist eine Andersbenennung sinnvoll.

Jetzt wiederholen wir das Ganze für **ID_FILE_OPEN**, benennen hier die Funktion in "**OnMyFileOpen**" um und klicken schließlich im **ClassWizard** auf **OK**.

Folgendes Ergebnis haben wir im **ClassWizard** erreicht:

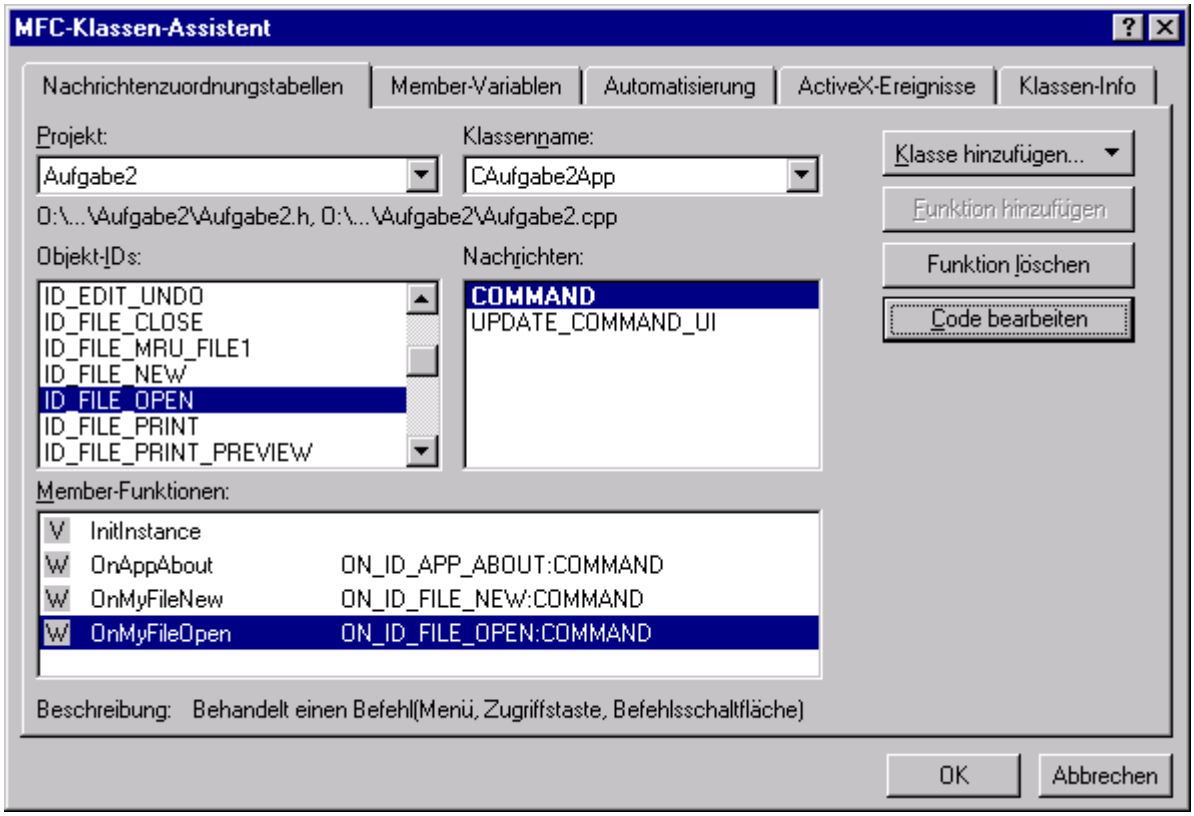


Abbildung 22: ClassWizard mit zwei neuen Nachrichten

An die beiden Nachrichten der Menüpunkte "**Neue Datei**" und "**Datei öffnen**" sind hiermit zwei neue Funktionen gebunden. Ein Blick in das Source File "**Aufgabe2.cpp**" zeigt am Ende der Datei die beiden neuen Funktionen:

```
void CAufgabe2App::OnMyFileNew()
{
     // TODO: Code für Befehlsbehandlungsroutine hier einfügen
}
void CAufgabe2App::OnMyFileOpen()
{
     // TODO: Code für Befehlsbehandlungsroutine hier einfügen
}
```

Jetzt können wir das Projekt neu compilieren, was anstandslos gelingt. Ein Aufruf der beiden Sorgenkinder "Neue Datei" und "Datei öffnen" zeigt - es passiert nichts. Vollkommen logisch, denn die beiden Funktionen tun ja - noch - nichts.

Der integrierte Debugger von Visual C++ 5.0

Ziele des Kapitels

- Beschreibung der Komponenten des Debuggers
- Handhabung des Debuggers
- Setzen und Anwenden von Haltepunkten
- Kontrolle von Variablen und Zuständen

Der Debugger

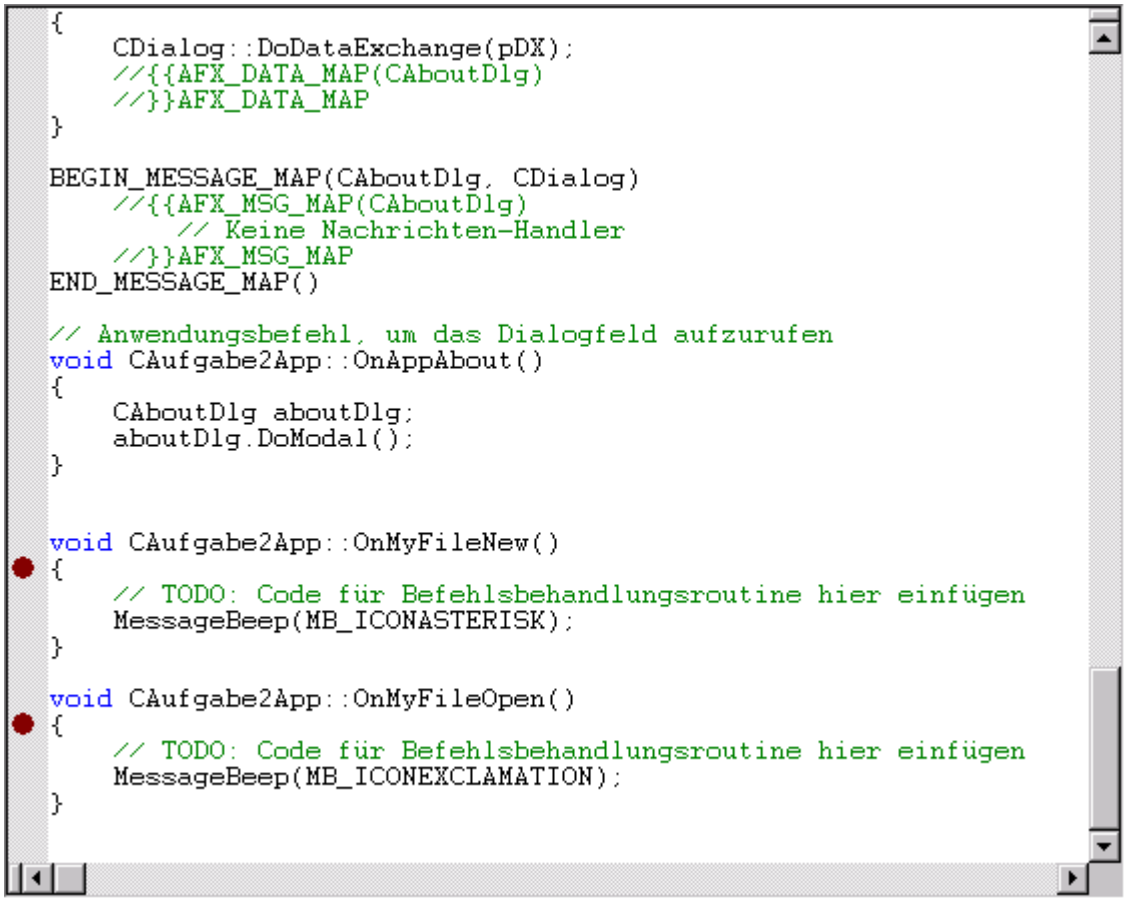
Wenn beim Programmieren etwas reibungslos klappt, erweckt das immer Argwohn. Schauen wir doch mal nach, ob die beiden neu generierten Funktionen überhaupt aufgerufen werden.

Aufgabe 8: Setzen Sie einen Haltepunkt auf die beiden neuen Funktionen

Setzen Sie Haltepunkte auf die beiden neuen Funktionen und prüfen Sie, ob diese bei Aktivierung der beiden Menüpunkte korrekt aktiviert werden.

Vorgehensweise

Laden Sie mittels des FileViews des Arbeitsbereichs die Datei "Aufgabe2.cpp" in den Editor. Gehen Sie zunächst an das Ende des Files zu den beiden Funktionen. Setzen Sie den Cursor auf die Zeile nach **`void CAufgabe2App::OnMyFileNew()`** und fügen Sie einen Haltepunkt ein. Dies geschieht mittels der Menüleiste des Arbeitsbereichs (siehe Abbildung 17: Menüleiste Arbeitsbereich) oder durch den Shortcut F9. Wiederholen Sie den Vorgang für die zweite Funktion **`void CAufgabe2App::OnMyFileOpen()`**:



```
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CAboutDlg)
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
    //{{AFX_MSG_MAP(CAboutDlg)
        // Keine Nachrichten-Handler
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

// Anwendungsbefehl, um das Dialogfeld aufzurufen
void CAufgabe2App::OnAppAbout()
{
    CAboutDlg aboutDlg;
    aboutDlg.DoModal();
}


void CAufgabe2App::OnMyFileNew()
{
    // TODO: Code für Befehlsbehandlungsroutine hier einfügen
    MessageBeep(MB_ICONASTERISK);
}

void CAufgabe2App::OnMyFileOpen()
{
    // TODO: Code für Befehlsbehandlungsroutine hier einfügen
    MessageBeep(MB_ICONEXCLAMATION);
}
```

Abbildung 23: Der Editor mit den neuen Haltepunkten

Um den **Debugger** zu starten, wählen Sie im Menü "**Erstellen**" im Untermenü "**Debug starten**" den Befehl "**Ausführen**", "**In Aufruf springen**", "**Ausführen bis Cursor**" oder "**Verbinden mit Prozess**". Die folgende Tabelle zeigt die Befehle, die im Untermenü "**Debug starten**" des Menüs "**Erstellen**" erscheinen können und die zugehörigen Aktionen.

Wenn Sie möchten, können Sie auch die **API-**Funktion **`MessageBeep(MB_ICONEXCLAMATION)`** in die beiden Funktionen einsetzen. Das hat zur Folge, daß immer bei Aufruf ein kurzer "Piep" erzeugt wird.

Befehle im Untermenü "Debug starten" des Menüs "Erstellen"

Menübefehl	Aktion
Ausführen	Führt den Code von der aktuellen Anweisung bis zu einem Haltepunkt oder zum Programmende aus (gleichbedeutend mit der Symbolleistenschaltfläche Ausführen).
In Aufruf springen	Führt die Anweisungen des Programms nacheinander aus und springt jeden Funktionsaufruf an, der vorkommt.
Ausführen bis Cursor	Führt das Programm bis zu der Zeile aus, in der die Einfügemarke steht. Dies ist gleichbedeutend mit dem Setzen eines vorübergehenden Haltepunktes an der Position der Einfügemarke.
Verbinden mit Prozess	Verbindet den Debugger mit einem laufenden Prozess. Dann können Sie den Prozess anhalten und die Debugging-Schritte wie gewohnt ausführen.

Achtung! Verwenden Sie nicht CTRL-F5 bzw. "Programm ausführen", denn dadurch wird nur die Anwendung gestartet, nicht aber der Debugger!

Ihre Anwendung wird nun gestartet und sofort in **OnMyFileNew()** gestoppt, da diese Nachricht beim Start generiert wird. Setzen Sie die Ausführung zunächst mittels F5 fort. Das

Programm läuft jetzt "ganz normal".

Aktivieren Sie "Datei neu" oder "Datei öffnen". Der Debugger stoppt am definierten Haltepunkt. Jetzt haben Sie die Möglichkeit, Ihr Programm in einzelnen Schritten oder bis zum Ende der aktuellen Funktion auszuführen. Dies ermöglicht eine präzise Fehlersuche im Code:

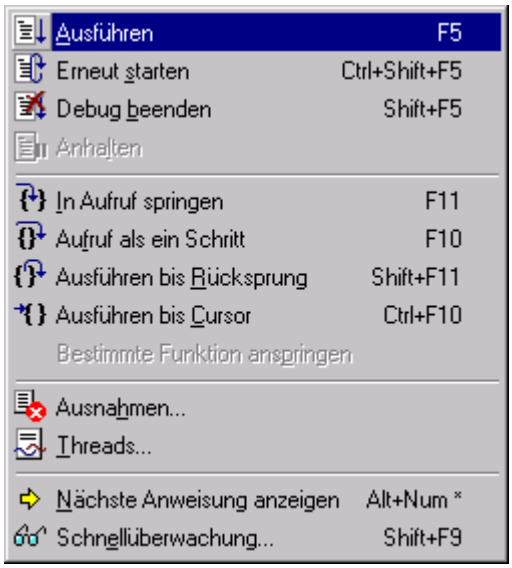


Abbildung 24: Befehle des Debuggers

Wenn Sie den Debug starten, wird anstelle des Menüs "Erstellen" das Menü "Debug" in der Menüleiste angezeigt. Dann können Sie die Ausführung des Programms mit Hilfe der Befehle steuern, die in der folgenden Tabelle genannt werden.

Befehle im Menü "Debug" zur Steuerung der Ausführung des Programms

Menübefehl	Aktion
Ausführen	Führt den Code von der aktuellen Anweisung bis zu einem Haltepunkt oder zum Programmende aus (gleichbedeutend mit der Symbolleistenschaltfläche Ausführen). Ist das Menü Debug nicht verfügbar, so können Sie im Menü Erstellen im Untermenü Debug starten den Befehl Ausführen wählen.
Erneut starten	Setzt die Ausführung auf die erste Programmzeile zurück. Durch diesen Befehl wird das Programm erneut in den Speicher geladen, die aktuellen Werte aller Variablen werden verworfen (Haltepunkte und Überwachungsausdrücke bleiben gültig). Die Funktionen main() oder WinMain() werden automatisch angehalten.
Debug beenden	Beendet den Debug. Sie kehren zur normalen Bearbeitung zurück.
Anhalten	Hält das Programm an der aktuellen Position an.
In Aufruf springen	Führt die Anweisungen des Programms nacheinander aus und springt jeden Funktionsaufruf an, der vorkommt. Falls das Menü Debug nicht verfügbar ist, können Sie im Menü "Erstellen" im Untermenü "Debug starten" den Befehl "In Aufruf "springen wählen.
Aufruf als ein Schritt	Führt die Anweisungen des Programms nacheinander aus. Wird bei Verwendung dieses Befehls ein Funktionsaufruf erreicht, so wird die Funktion ausgeführt, ohne daß die einzelnen Anweisungen der Funktion durchlaufen werden.
Ausführen bis Rücksprung	Führt das Programm aus einem Funktionsaufruf heraus aus und hält das Programm bei der Anweisung an, die unmittelbar auf den Aufruf für die Funktion folgt. Mit diesem Befehl können Sie die Ausführung der aktuellen Funktion rasch beenden, wenn Sie festgestellt haben, daß sich in der Funktion kein Programmfehler befindet.
Ausführen bis Cursor	Führt das Programm bis zu der Zeile aus, in der sich die Einfügemarke befindet. Dies ist gleichbedeutend mit dem Setzen eines vorübergehenden Haltepunktes an der Position der Einfügemarke. Falls das Menü Debug nicht verfügbar ist, können Sie im Menü Erstellen im Untermenü Debug starten den Befehl Ausführen bis Cursor wählen.
Bestimmte Funktion anspringen	Die Befehle des Programmes werden in Einzelschritten abgearbeitet und die angegebenen Funktionsaufrufe eingegeben. Dies funktioniert für jede beliebige Schachtelungstiefe der Funktionen.
Ausnahmen	Listet die registrierten Ausnahmebedingungen und die Reaktionen auf sie auf.
Threads	Listet die aktiven Threads des Programmes.
Nächste Anweisung anzeigen	Setzt den Cursor auf die nächste auszuführende Anweisung.
Schnellüberwachung	Aktiviert das Schnellüberwachungsfenster

Die Schnellüberwachung

Die Schnellüberwachung ist eine sehr wichtige Funktion. Sie erlaubt das Prüfen von Variablen und Überwachen deren Inhalts. Sie ist ein wichtiges Werkzeug bei der Suche nach Fehlern.

Das Dialogfeld "Schnellüberwachung" enthält ein Textfeld, in dem Sie den Namen eines Ausdrucks oder einer Variablen eingeben können, und ein Tabellenblattfeld, das den aktuellen Wert des Ausdrucks bzw. der Variablen anzeigt, den bzw. die Sie eingegeben haben.

Mit Schnellüberwachung können Sie rasch den Wert einer Variablen oder eines Ausdrucks untersuchen. Sie können mit der Schnellüberwachung auch den Wert einer Variablen ändern und eine Variable oder einen Ausdruck zum Fenster „Überwachung" hinzufügen.

Das Tabellenblattfeld "Aktueller Wert" zeigt immer nur eine Variable bzw. einen Ausdruck an. Wenn Sie eine neue Variable bzw. einen neuen Ausdruck in das Textfeld eingeben und die **EINGABETASTE** drücken, wird die vorherige Variable bzw. der vorherige Ausdruck im Feld "Aktueller Wert" durch die neue Eingabe ersetzt.

Anzeigen einer Skalarvariablen bzw. eines Skalarausdrucks

Wenn Sie im Textfeld eine Skalarvariable oder einen Skalarausdruck eingeben, zeigt die Schnellüberwachung das Ergebnis in der ersten Zeile des Tabellenblatts an. Wenn Sie jedoch eine Array-, eine Objekt- oder eine Strukturvariable eingeben, zeigt die Schnellüberwachung in dem Tabellenblatt zusätzliche Details an, was Sie an den Feldern mit Pluszeichen "+" und Minuszeichen "-" erkennen. Klicken Sie auf diese Felder, um die Ansicht der Variablen ein- oder auszublenden.

Anzeigen eines Objekts, einer Referenz oder eines C++-Zeigers auf ein Objekt

Wenn die Variable ein Objekt, eine Referenz oder ein C++-Zeiger auf ein Objekt ist, blendet die Schnellüberwachung die Variable automatisch ein, so daß die wichtigsten Daten in der obersten Ebene angezeigt werden. Beim folgenden C++-Objekt z.B.:

```
CString String = {...}
char * m_pchData =0x7ffdf000 "abc"
int m_nDataLength=4
int m_nAllocLength=1244628
```

zeigt die Schnellüberwachung folgendes an:

CString String = {"abc"}

Anzeigen eines Objekts

Ist die Variable ein C++-Zeiger oder eine Referenz auf ein Objekt, so zeigt die Schnellüberwachung automatisch die Referenz oder den Zeiger an. Die Schnellüberwachung fügt ein weiteres Element zu dem eingeblendeten Objekt hinzu. Dieses zusätzliche Element, das wie eine weitere Basisklasse aussieht, kennzeichnet die abgeleitete Unterklasse. Wenn z.B. eine als C++-Zeiger auf ein CObject deklarierte Variable tatsächlich auf CComboBox zeigt, erkennt die Schnellüberwachung diese Tatsache und fügt ein zusätzliches Element hinzu, so dass Sie auf die CComboBox-Elemente zugreifen können.

Die Schnellüberwachung zeigt Werte in ihrem Standardformat an. Sie können das Anzeigeformat mit Hilfe von Formatierungssymbolen ändern, z. B. um Unicode-Zeichen anzuzeigen.

Programmierung in C++

Abbildung 25: C++ Logo

Vorbemerkungen

Dieses Kapitel

Dieses Kapitel gibt eine Übersicht über die Elemente und die Handhabung von C++. Die wichtigsten Teile von C++ werden angesprochen und erklärt. Voraussetzung für dieses Seminar ist eine gewisse Vertrautheit mit C++, jedoch reicht für das Verständnis auch die Fähigkeit aus, C++ Code zu lesen und zu verstehen.

C++ ist eine Obermenge von C, ein gutes C++ Programm nutzt das beste der beiden Welten - von C und C++. Es gibt einige Konstrukte in C++, die sehr gefährlich sein können und den Entwickler in beliebig große Probleme stürzen kann. Auch diese Dinge finden sich in diesem Tutorial als Randbemerkung, ebenso wie erst vor kurzem von der ANSI-Kommission neu hinzugefügte Sprachelemente (z.B. Funktions- und Klassen-Templates) mit einer kurzen Bewertung über Nutzen und Gefahren im Programmierералltag.

Aufbau des C++ Kurses

Dieser Kurs lehrt C++ auf eine Weise, in der auch eine natürliche Sprache erlernt wird. Wie beim Studium einer Fremdsprache werden zunächst die wichtigsten Elemente erklärt, ohne den Lernenden mit unnötigen Details zu belasten. Danach wird das Erlernte schrittweise vertieft und Spezialfälle behandelt. Ziel ist es, einen schnellen Zugang zu C++ ohne Ballast zu ermöglichen.

Es wird zunächst oft darauf verzichtet, das genaue "Warum" zu erklären. Sollten Sie genauere Informationen zu einem Thema benötigen, finden Sie jeweils Verweise auf die Vertiefung des entsprechenden Themas. So können Sie selber bestimmen, wie weit ins Detail sie gehen möchten und können. Es empfiehlt sich also, das Tutorial in der vorgegebenen Reihenfolge durchzuarbeiten, jedoch bei Bedarf einfach weiterzublättern und das Drumherum nachzulesen.

Im Gegensatz zum üblichen Ansatz sind zusammengehörende Themenbereiche deshalb nicht zusammengefaßt, sondern kehren nach und nach ausführlicher wieder. So wird z.B. zunächst allgemein die Technik der Überladung von Funktionen erklärt. Später finden Sie diese Technik dann bei Konstruktoren wieder, und erst fast am Ende des Kapitels wird die Überladung von Klassenfunktionen Punkt für Punkt dargelegt. Überladung ist eines der wichtigsten Merkmale von C++, daher wird sie auch sehr früh erwähnt. Doch es nutzt nichts, wenn man noch nicht einmal richtig die Verwendung von Klassenfunktionen begriffen hat, aber bereits mit Überladung hin und her bombardiert wird.

Das Tutorial zielt also stark auf den "Aha-, das habe ich doch schon mal gehört!" Effekt ab und regt dabei das aktive Nachschlagen der interessanten Themen an.

Die Ursprünge von C++

Die Sprache C wurde ursprünglich von AT&T entwickelt, um das Betriebssystem UNIX damit zu erstellen. Das Hauptziel war dabei die Erzeugung von effektivem, portierbarem Code. UNIX sollte auf den verschiedensten Rechnerarchitekturen implementiert werden. Da es komplett in C programmiert war, ließ sich UNIX leicht auf neue Plattformen portieren.

Der erste Standard basiert auf den Arbeiten von Brian W. Kernighan und Dennis M. Ritchie und wird heute meist mit K&R bezeichnet. Später wurde die Standardisierung vom ANSI-Komitee übernommen. ANSI-C ist der heute allgemein verwendete Standard, der K&R als Teilmenge enthält (die meisten Compiler generieren Warnungen, setzen den Code aber trotzdem um).

Bjarne Stroustrup, ein Mitarbeiter von AT&T, entwickelte in den 80er Jahren C++, um eine objektorientierte Erweiterung der Sprache C zu schaffen. Die erste Implementierung von C++ existierte als Präprozessor für C-Compiler, war also quasi nur ein "Aufsatz" auf den normalen C-Compiler.

Da Objektorientierung zu diesem Zeitpunkt neu war und alle existierenden Implementierungen von objektorientierten Sprachen sehr langsam und wenig effektiv waren, war das Hauptziel von C++, die Effektivität von C zu bewahren.

Zunächst wurde C++ euphorisch begrüßt. Da der C++-Compilerbau jedoch sehr kompliziert ist, endete die Euphorie schnell in Ernüchterung, scheiterten Projekte denn oft an Defiziten der verfügbaren Compiler und letztlich auch der noch jungen Sprache. Als dann die ersten 4GL und RAD-Sprachen herauskamen (also Sprachen der vierten Generation sowie RAD = Rapid Development Tools), wurde C++ schnell der frühe Tod bescheinigt.

Seitdem graphische Betriebssysteme wie Windows NT und Windows 95 sich immer stärker durchsetzen, deren Benutzeroberflächen durch ihre Beschaffenheit geradezu nach objektorientierten Sprachen schreien, und letztlich auch wirklich effektive RAD-Tools für C++ auf den Markt gekommen sind, hat C++ eine wahre Renaissance erfahren.

Die Möglichkeit, z.B. eine Klasse mit allen notwendigen Eigenschaften eines Windows-Fensters zu definieren, schließt Fehlerquellen weitgehend aus und verringert den Zeitaufwand gegenüber einer C- Lösung immens. Im Verlauf des Seminars werden wir zunächst auf den "klassischen" Ansatz in C eingehen und dann diese Dinge in C++ mittels **MFC** wiederholen, um die Vorteile herauszuheben.

Das Konzept von C++

C++ kann als prozedurale Sprache mit zusätzlichen Konstrukten betrachtet werden, einige davon wurden für die objektorientierte Programmierung hinzugefügt und einige für einen verbesserten prozeduralen Syntax. Ein gut geschriebenes C++-Programm zeigt Elemente sowohl von objektorientiertem Programmierstil und klassischem prozeduralem Syntax. C++ ist außerdem eine erweiterbare Sprache, da man neue Typen definieren kann, so daß sie sich genauso wie die vordefinierten Typen verhalten, die Teil der Sprache C oder C++ sind. C++ wurde für die Erstellung großer Anwendungen entwickelt.

C++ ist eine objektorientierte Erweiterung von C. C++ kann verwendet werden, um C-Programme zu compilieren. Alles, was man in C lösen kann, kann genauso unter C++ gemacht werden. C++- Programme und ihre Erweiterungen können jedoch nicht mit einem C-Compiler umgesetzt werden. C ist also als Teilmenge vollständig in C++ enthalten.

Polymorphismus

Der Begriff "Polymorphismus" stammt aus dem Griechischen und bedeutet auf deutsch schlicht "Vielgestaltigkeit". Bei Sprachen bezeichnet dies die Fähigkeit, je nach Situation verschiedene Formen anzunehmen. C++ ist in der Lage, auf bestimmte Ereignisse objektspezifisch zu reagieren. Dies wird realisiert, indem die notwendigen Behandlungsroutinen in den einzelnen Klassen an die veränderten Gegebenheiten angepaßt werden können (siehe "Abgeleitete Klassen").

Dateinamen

Als Konvention hat sich für C++ mittlerweile eingebürgert, C++-Quelldateien mit der Endung .CPP zu versehen (manche Compiler verwendeten zeitweise auch .CXX). Header bzw. Include-Dateien erhalten die Endung .H (in der Anfangszeit von C++ war es eher üblich, die Endung .HPP zu verwenden).

Die meisten Compiler schalten sich automatisch in C++-Modus, wenn sie .CPP-Dateien umsetzen und in C-Modus, wenn sie .C-Dateien bearbeiten. Da es für Header-Dateien keine solche Konvention gibt, kann man sich beliebig Probleme einhandeln, wenn man versehentlich Header, die C++-Definitionen enthalten, in "normale" C-Files einbindet.

Man sollte daher diesen feinen Unterschied berücksichtigen und keine "Mischfiles" erzeugen, sondern alles fein sortiert in getrennten Headern aufheben. Selbstverständlich wird der C++-Compiler C- Header bis auf wenige Ausnahmen klaglos umsetzen.

Lassen sich Mischfiles nicht vermeiden, dann kann man auch mit Defines arbeiten. Die meisten Compiler definieren **__CPLUSPLUS__**, wenn sie im C++-Modus arbeiten.

```
#if __CPLUSPLUS__
class CFILE
{
        private:
              long ptr;                    // Dateizeiger
              char Name[FNAME_LEN];        // Dateiname
              short state;                 // Dateistatus
              short mode;                  // Dateimodus
        public:
              CFILE(void);                 // Konstruktor
              ~CFILE(void);                // Destruktor
              short Open(char *,short);    // "Öffnen" Funktion
              short Close(void);           // "Schließen" Funktion
              short Read(char *,short);    // "Lesen" Funktion
              short Write(char *,short);   // "Schreiben" Funktion
};
#endif
```

Auch die Kommentare sind besser...

Waren bei C-Kommentare noch grundsätzlich zwischen den Zeichenfolgen "/*" und "*/" einzuschließen, erlaubt C++ die Kennzeichnung mittels eines doppelten Schrägstrichs "//". Er markiert den Beginn eines Kommentars, der bis zum Ende der Zeile geht:

```
/*
    Der C Kommentar kann sich durchaus über mehrere Zeilen erstrecken.
    Er geht von der Startmarkierung bis zur Endmarkierung.
*/

//  Der neue doppelte Schrägstrich ignoriert alles bis zum Zeilenende.
```

Die Lesbarkeit des Codes verbessert sich deutlich, wenn man bei kurzen Kommentaren, speziell am Zeilenende, von diesem neuen Syntax Gebrauch macht. Die meisten C++-Compiler akzeptieren "//" auch im C-Modus (wenn auch eigentlich illegalerweise).

Erweiterter Syntax bei Funktionen

Funktionsüberladung

Ein sehr leistungsfähiger Mechanismus von C++ ist die Möglichkeit, Funktionen mit verschiedenen Parametern mehrfach zu definieren. Der Compiler sucht dann die passende Funktion aus und ruft sie auf.

Will man z.B. eine Funktion zur Berechnung von x^2 definieren, dann würde man z.B. je eine Funktion für ganze Zahlen und reelle Zahlen definieren:

```
unsigned long ulong_square(unsigned long uVal)
{
      return (uVal * uVal);
}
float float_square(float fVal)
{
      return (fVal * fVal);
}
```

Diese Vorgehensweise hat natürlich ihre Tücken. Vor allen Dingen muß man immer darauf achten, daß man die richtige Funktion aufruft. In C++ kann man die beiden obigen Funktionen auch folgendermaßen definieren:

```
unsigned long square(unsigned long uVal)
{
      return (uVal * uVal);
}
float square(float fVal)
{
      return (fVal * fVal);
}
```

In C gäbe das eine Fehlermeldung, in C++ führt es dazu, daß der Compiler automatisch die richtige Funktion auswählt. Dies geschieht anhand der Argumente. Wird **square()** ein **float** übergeben, so wird die Gleitkommaversion verwendet, bei **unsigned long** jedoch die andere. Natürlich muß dann für jede Gelegenheit die passende Funktion definiert sein, ansonsten erhält man eine Fehlermeldung des Compilers.

Die korrekte Definition der Prototypen im Header ist damit ein muß! Konnte darauf bei C-Compilern durchaus noch verzichtet werden – außer einer Warnung passierte dort nichts – müssen im C++- Modus alle Funktionen deklariert sein, bevor sie benutzt werden können. Das liegt daran, daß der C- Compiler sonst nicht in der Lage ist, die richtige Funktion aus der Sammlung der überladenen herauszusuchen. Würde z.B. die obige Funktion mit einem

„double"-Parameter aufgerufen, so kann der C++-Compiler trotzdem die „float"-Version verwenden. Ohne Prototyp würde er jedoch nach einer „double"-Version suchen und diese nicht finden.

Aber Vorsicht!

Macht man von dieser Technik exzessiv Gebrauch, so weiß irgendwann nur noch der Compiler, welche Funktion er nehmen wird - und das Ergebnis mag dann nicht mehr sonderlich befriedigend sein.

Defaultwerte bei Argumenten

Mußten in C noch alle Parameter angegeben werden, so können bei C++ Defaultwerte definiert werden:

```
void info(char *psInfo, char *psHeader = "Fehler")
{
      cout << psHeader << ": " << psInfo << endl;
}

Int main(void)
{
      info("Hallo Welt");
      info("Noch ein Test", "Hallochen");

      return 0;
}
```

Die Bildschirmausgabe dieses Programmes würde dann folgendermaßen aussehen:

```
Fehler: Hallo Welt
Hallochen: Noch ein Test
```

Der fehlende Parameter wird durch den vorgegebenen Wert ersetzt. Beachten Sie bitte, daß sie Defaultwerte nur für die rechten Parameter angeben können. Es ist nicht möglich, für Parameter 2 keine Angaben zu machen, für Parameter 3 einer Funktion aber schon. Ein Aufruf der Art:

```
func("Parm1",, "Parm3");
```

erzeugt eine Fehlermeldung.

Klassen

Das Klassenkonzept ist der Grundstein für alle objektorientierten Programmiersysteme (OOPS). Eine Klasse besteht aus einem Objekt (oder einer Gruppe von Objekten) und den Funktionen, die die Objekte betreiben. In C++ wird eine Klasse auf dieselbe Weise definiert wie eine Struktur, es wird lediglich anstatt **„struct“** das neue Schlüsselwort **„class“** verwendet:

```
class CFILE;                                    // Instanz einer Klasse
```

C++ erweitert zum einen die **„struct“** Anweisung und bietet zusätzlich die neue Anweisung **„class“** an.

Objekte innerhalb von Software entstanden aus der Notwendigkeit heraus, die reale Welt im Computer zu simulieren. Nur die für die Aufgabe wichtigen Merkmale und Verhaltensweisen der realen Welt werden modelliert und finden sich in Form von Objekten wieder.

Mittels Vererbung (vgl. "Abgeleitete Klassen") können Eigenschaften von Klassen ganz oder teilweise an andere weitergegeben werden. Diese Technik erhöht die Wiederverwendbarkeit von Code und verringert die Redundanz.

Ferner können Variable und Funktionen in verschiedene Sicherheitsbereiche eingestuft werden und somit aus dem unbefugten Zugriff von außen oder auch der abgeleiteten Klassen genommen werden. Dadurch wird sichergestellt, daß Manipulationen nur über die definierten Schnittstellen gemacht werden können (vgl. "Bestandteile einer Klasse"). Dies kommt besonders dann vorteilhaft zum Tragen, wenn mehrere Entwickler an demselben Projekt zusammenarbeiten und sich so mehr oder weniger auf die Einhaltung der Spezifikationen verlassen müssen.

Jede Klasse ist also ein abgeschlossenes Objekt, das für sich unabhängig von anderen Objekten definiert und behandelt werden kann.

- Alles, was ein Objekt *"**weiß**"*, wird durch seine Variablen ausgedrückt.
- Alles, was ein Objekt *"**kann**"*, wird durch seine Funktionen ausgedrückt.

Ein Objekt bzw. eine Klasse kann also als Datenspeicher betrachtet werden, der die Kenntnis zur Datenmanipulation besitzt.

Bestandteile einer Klasse

Eine Klassendefinition enthält Deklarationen von Variablen und Funktionen. Eine Klassendefinition bietet außerdem drei verschiedene Sicherheitsstufen für Variablen und Funktionen: "**private**", "**public**" und "**protected**".

```
class CFILE
{
        private:
              long ptr;                    // Dateizeiger
              char name[FNAME_LEN];        // Dateiname
              short state;                 // Dateistatus
              short mode;                  // Dateimodus
        public:
              CFILE(void);                 // Konstruktor
              ~CFILE(void);                // Destruktor
              short Open(char *,short);    // "Öffnen" Funktion
              short Close(void);           // "Schließen" Funktion
              short Read(char *,short);    // "Lesen" Funktion
              short Write(char *,short);   // "Schreiben" Funktion
};
```

Die obige **CFILE-**Klasse besitzt vier private Elementvariablen (Objekte) und sechs Funktionen, die diese Objekte manipulieren. Jeder Zugriff auf die Elementvariablen muß zwangsläufig über diese Klassenfunktionen ausgeführt werden. Das "**private**"**-**Schlüsselwort sorgt für die Kapselung der Daten (Information Hiding), indem es der Anwendung, die die Klassen verwendet, den Zugriff auf alles in diesem privaten Bereich verwehrt. Nur die Elementfunktionen selbst können auf diese privaten Objekte zugreifen.

Damit geht C++ weit über die Handhabung der Daten in C hinaus. Dort werden Strukturen, sonstige Variablen und Funktionen getrennt definiert, und jeder hat prinzipiell Zugriff auf alles. Die Vergangenheit hat gezeigt, daß davon auch kräftig Gebrauch gemacht wird, mit allen damit verbundenen Problemen. C++ kapselt die Repräsentation der Daten und stellt standardisierte Schnittstellen für den Zugriff bereit.

Änderungen an den Datenstrukturen wirken sich damit nicht auf die Zugriffsmechanismen aus. Durch Anpassung der Zugriffsfunktionen kann weiterhin sichergestellt werden, daß die Schnittstelle den bisherigen Spezifikationen entspricht - unabhängig von der Repräsentation der Daten im Speicher.

- In C++ werden Informationen **gekapselt** und über definierte **Schnittstellen** bereitgestellt.
- Änderungen der internen **Datenstrukturen** können daher erfolgen, ohne daß die **Schnittstellen** nach außen andere Verhaltensweisen zeigen.

Instanzierung von Klassen

Eine Instanz einer Klasse wird auf zwei Arten erzeugt. Die erste ist die Deklaration einer Klasseninstanz auf dieselbe Weise wie in Standard C eine Variable erzeugt wird:

```
short x;                // Erzeuge eine C Variable "x" vom Typ Integer
CFILE LogFile;          // Erzeuge eine Instanz der CFILE Klasse
```

Die zweite Methode ist, eine Klasseninstanz zu erzeugen, indem man Speicher für eine neue Instanz mittels eines Zeigers und dem C++-Schlüsselwort "**new**" alloziert:

```
CFILE * pLogFile;       // Erzeugt einen Zeiger auf die CFILE Klasse
pLogFile = new CFILE;   // Erzeuge eine Instanz und weise dem Zeiger zu
```

Das "new"-Schlüsselwort ist ein verbessertes "calloc" oder "malloc". Es berechnet die Größe des Speicherblocks, der benötigt wird, um das Objekt zu erzeugen und alloziert den notwendigen Speicher. Ebenso wie bei "calloc" oder "malloc" muß der bestellte Speicher später auch wieder freigegeben werden. Dies geschieht mit dem Operator "**delete**".

```
delete pLogFile         // Speicher wieder freigeben
```

Es obliegt dem Programmierer, dies zu erledigen. Ansonsten bleibt der bestellte Speicher bis zum Ende der Anwendung belegt (mit der Konsequenz, daß bei vermehrtem Gebrauch von "**new**" eventuell irgendwann kein Speicher mehr zur Verfügung steht).

Sollte es dem Betriebssystem nicht möglich sein, den benötigten Speicher bereitzustellen, liefert der "**new**"**-**Operator wie die alten C-Funktionen einen NULL-Zeiger. Auch diese Prüfung liegt beim Programmierer und sollte nicht vergessen werden.

- **„new“** kann „NULL“ liefern, wenn kein Speicher vorhanden ist (sollte immer geprüft werden).
- **„delete“** muß aufgerufen werden, um den Speicher wieder freizugeben (daran muß immer gedacht werden).

Konstruktoren und Destruktoren

Standardkonstruktor und einfache Konstruktoren

Die CFILE-Klasse besitzt eine CFILE() Funktion als **Konstruktor** und eine ~CFILE() Funktion als **Destruktor**. Der **Konstruktor** ist eine Funktion mit demselben Namen wie die Klasse, die bei der Erzeugung einer Instanz der Klasse automatisch aufgerufen wird und deren Zweck es ist, alle notwendigen Initialisierungen auszuführen, die von der Klasse benötigt werden.

Minimalkonstruktor

Es sollte immer zumindest ein **Konstruktor** definiert werden. Besitzt eine Klasse keinen **Konstruktor**, generiert der Compiler einen sogenannten **Minimalkonstruktor**, mit dessen Hilfe die Objekte erzeugt werden. Dieser wird implizit aufgerufen.

Er besitzt jedoch keine Funktionalität! Somit führt der Compiler für neue Objekte keine Initialisierung durch. Der Compiler ist nicht verpflichtet, die Variablen mit irgendwelchen Werten zu belegen. Letztendlich werden die Elementvariablen irgendwelche zufälligen Werte aus dem Speicher annehmen.

- Die **Elemente** einer Klasse bleiben immer **uninitialisiert**.
- Um den **Elementen** definierte Werte zuzuordnen, verwendet man **Konstruktoren**.

Konstruktoren mit Parametern

Wie jede Funktion, akzeptiert auch ein **Konstruktor** Parameter. Die Parameter werden bei der Instanzierung des Objektes übergeben:

```
class CFILE
{
                ...
        private:
                long ptr;                       // Dateizeiger
        public:
                CFILE(char *pFileName);
};
CFILE LogFile("LogFileName");                   // Konstruktor mit Parameter
```

Es können mehrere **Konstruktoren** angegeben werden. Der Compiler sucht dann den Passenden selbst aus. Dieses Verfahren nennt sich "Überladung" oder "Overloading". Es wird im Kapitel "Überladung" besprochen.

- **Konstruktoren** akzeptieren auch Parameter.
- Es können mehrere **Konstruktoren** angegeben werden.

Initialisierung von Konstanten mit Initialisierungslisten

In einer Klasse können sich neben Member-Objekten auch Konstanten befinden. Diese können aber nicht innerhalb der Klasse z.B. mittels

```
class CFILE
{
        private:
                const int blocksize = 512;    // Konstante
                ...
        private:
                CFILE(char *pFileName);
                ...
```

```
};
```

initialisiert werden. Es gibt zunächst keinen Speicher, in den die 512 geschrieben werden könnte. Dieser wird erst verfügbar, wenn das Objekt existiert. Auch eine spätere Zuweisung der Art

```
blocksize = 512;                    // Konstante
```

(beispielsweise innerhalb des **Konstruktors**) wird scheitern, da der Wert eine Konstante ist.

Abhilfe schaffen hier die **Initialisierungslisten**. Sie bieten die einzige Möglichkeit, Konstanten bei der Erzeugung eines Objektes Werte zuzuweisen.

```
short CFILE::CFILE(bsize) : blocksize(bsize)
{
        ...
}
```

Selbstverständlich kann auch ein fester Wert angegeben werden:

```
short CFILE::CFILE() : blocksize(512)
{
        ...
}
```

Die Definition der **Klassenfunktionen** hier, ist ein Vorgriff auf das nächste Kapitel "Deklaration von Klassenfunktionen". Das Thema **Initialisierungslisten** hat eine wichtige Bedeutung bei der Konstruktion von eingebetteten Objekten (siehe "Umfassende Klassen und Member-Objekte").

- **Initialisierungslisten** helfen beim Vorbelegen von **konstanten Werten.**

Kopierkonstruktor

Damit eine Klasse kopiert werden kann, muß sie einen **Kopierkonstruktor** besitzen. Der **Kopierkonstruktor** ist optional und hat das Format

```
Klassenname(Klassenname &)
```

Das "&" Zeichen ist in diesem Zusammenhang als "**Referenzoperator**" sehr wichtig (siehe "Referenzvariablen").

Falls kein **Kopieroperator** definiert wurde, kopiert der Compiler ein Objekt 1:1, also bitweise, auf das andere um. Dies ist vergleichbar mit einer memcpy()-Operation zweier Strukturen. In den meisten Fällen mag dies ausreichend sein, falls nicht, muß der notwendige Code im **Kopierkonstruktor** implementiert werden. Es kann nur genau einen **Kopierkonstruktor** geben, er kann also nicht Überladen werden (vgl. "Überladung").

- Kompliziertere **Objekte** lassen sich einfacher **kopieren**, wenn ein **Kopierkonstruktor** verwendet wird.
- Es kann nur genau einen **Kopierkonstruktor** geben.

Destruktoren

Ein **Destruktor** hat ebenfalls denselben Namen wie die Klasse, jedoch mit einer '~' (Tilde) davor. Der **Destruktor** wird verwendet, um die notwendigen Aufräumarbeiten während der Vernichtung der Klasse auszuführen (so z.B. sicherzustellen, daß die Datei geschlossen der mit "**new**" allozierter Speicher freigegeben wird).

Im Gegensatz zu **Konstruktoren** erlauben **Destruktoren** keine Parameter. Der **Destruktor** wird immer automatisch mit der Vernichtung des Objektes aktiviert, entweder durch den Aufruf von "**delete**" oder wenn ein Objekt ungültig wird (beispielsweise ein lokales Objekt innerhalb einer Funktion). Daher wäre es auch gar nicht möglich, Parameter zu übergeben.

- **Destruktoren** erledigen die „**Aufräumarbeiten**" beim Löschen und Zerstören eines Objektes.
- Es kann also nur genau eine **Destruktor-**Funktion geben. Überladene **Destruktoren** existieren nicht.

Zusammenfassung

Für **Konstruktoren** gilt also:

- **Konstruktoren** sind Klassenfunktionen.
- Der **Konstruktorname** ist Klassenname.
- Kein Rückgabedatentyp, auch nicht "void".
- **Konstruktoren** erlauben Parameter.
- Durch Überladung (Overloading) sind viele **Konstruktoren** möglich (siehe "Funktionsüberladung").
- **Konstruktoren** werden im "**public"-**Bereich deklariert und wie gewöhnliche Funktionen definiert.
- Zweck: Ein Objekt wird explizit initialisiert bei der Definition.

Für **Destruktoren** gilt also:

- **Destruktoren** sind Klassenfunktionen.
- der **Destruktorname** ist ~Klassenname.
- Ein Rückgabedatentyp, auch nicht "void".
- keine Parameter möglich (und auch notwendig).
- nur ein **Destruktor** möglich, keine Überladung.
- **Destruktoren** werden im "**public**" Bereich deklariert und wie gewöhnliche Funktionen definiert.
- Zweck: Durchführung von Aufräumarbeiten.

Klassenfunktionen

Deklaration von Klassenfunktionen

Eine **Klassendefinition** (wie eine Klasse CFILE...) befindet sich üblicherweise in einer Include-Datei. Die tatsächlichen Funktionen der Klasse werden in einer C++-Quelldatei definiert. Jeder **Klassenfunktion** wird der Klassenname vorangestellt zu der sie gehört, sowie das Symbol '::'.

```
class CFILE
{
        private:
              char  name[FNAME_LEN];       // Dateiname
              ...
        public:
              // Definition der Klassenfunktion innerhalb der Klasse
              short Open(char * pFileName, short newmode);
              ...
};

short CFILE::Open(char * pFileName, short newmode)
{
        mode = newmode;           // Zugriff auf privates Datenelement

        strcpy(name, pFileName);
        // Öffnen jetzt ausführen
        ...
        return (Status);
}
```

Aufruf von Klassenfunktionen

Klassenfunktionen werden wie einfache C-Funktionen aufgerufen. Der Unterschied ist, daß sie einen ähnlichen Syntax verwenden wie er für Elemente einer Struktur verwendet wird.

```
CFILE LogFile;                              // Instanz von CFILE erzeugen
CFILE * pInputFile;                         // Zeiger auf Instanz CFILE
PInputFile = new CFILE;                     // Instanz von CFILE erzeugen
LogFile.Open("LogFile",O_APPEND);           // Logfile öffnen
PInputFile -> Open("InputDat",O_READONLY); // InputFile öffnen
PInputFile -> Read(buffer,sizeof(buffer)); // InputFile lesen
```

Im obigen Beispiel wird kein Dateizeiger an die CFILE-Funktionen übergeben, wie es in den Standard C-Dateifunktionen der Fall ist. Dies ist so, da jede Instanz von CFILE seine eigenen internen Steuerinformationen innerhalb des privaten Bereichs der Klasse verwaltet.

- C++ vereinfacht die **Schnittstellen** zwischen **Klassen** und **Anwendungen**, da **Klassen** in sich vollständig abgeschlossen sind. Sie enthalten alle **Eigenschaften** und/oder Objekte, die die **Klasse** beschreiben.

Überladung

Überladene Klassenfunktionen

Analog zu einfachen Funktionen können auch Klassenfunktionen **überladen** werden. Der dafür verwendete Syntax ist derselbe: Die Funktionen werden in der Klasse mehrfach deklariert und der notwendige Code in die C++-Quelldatei eingefügt.

```
class CFILE
{
        private:
              char  name[FNAME_LEN];       // Dateiname
              ...
        public:
              // Definition der Klassenfunktion innerhalb der Klasse
              short Write(char * pText);
              short Write(void * pData, int iDataLen);
              ...
};

short CFILE:: Write(char * pText)
{
        // Textblock jetzt schreiben
        ...
        return (Status);
}

short CFILE:: Write(void * pData, int iDataLen)
{
        // Datenblock der Länge iDataLen jetzt schreiben
        ...
        return (Status);
}
```

Überladene Konstruktoren

Wie bereits im Kapitel "**Konstruktoren und Destruktoren**" erwähnt, können auch **Konstruktoren** überladen werden. Da **Konstruktoren** grundsätzlich auch nur "normale" Klassenfunktionen sind, bedürfen sie keiner Sonderbehandlung in diesem Fall.

```
class CFILE
{
              ...
        private:
              Long ptr;                    // Dateizeiger
        public:
              // Überladener Konstruktor
              CFILE(char *pFileName);
              CFILE(char *pFileName, mode);

};
CFILE LogFile("LogFileName");              // Konstruktor mit Parameter
CFILE LogFile("LogFileName", SH_DENYRW);   // Konstruktor mit Parameter
```

- Der Compiler wählt dann den passenden **Konstruktor** aus den **Überladenen** aus. Dies kann bei komplizierteren Klassen sehr hilfreich sein.

Funktionen mit Vorgabewerten bei Argumenten

Im dem Beispiel zu "**Überladene Konstruktoren**" wäre es eher unsinnig gewesen den Operator zu überladen, die Verwendung von **Vorgabenwerten** (vgl. "Defaultwerte bei Argumenten") wäre die elegantere Lösung gewesen:

```
class CFILE
{
            ...
      private:
            Long ptr;                    // Dateizeiger
      public:
            // Konstruktor mit Defaultwerten
            CFILE(char *pFileName, int mode = O_RDONLY);
};

CFILE LogFile("LogFileName");              // Konstruktor mit Parameter
CFILE LogFile("LogFileName", O_RDWR);      // Konstruktor mit Parameter
```

- **Konstruktoren** sind eigentlich "normale" **Klassenfunktionen**, daher trifft obige Aussage auch auf diese zu. Jede **Klassenfunktion** kann Defaultwerte besitzen.

Inline Funktionen

Wenn eine Klasse eine sehr einfache Aufgabe ausführen soll, kann diese als **Inline Funktion** definiert werden. Eine **Inline Funktion** ist eine erweiterte Version der Funktionsdeklaration innerhalb der Klasse, mit Anfang- und Ende-Klammern als Umfassung der **Inline-**Befehle.

```
class CFILE
{
        private:
              char  name[FNAME_LEN];       // Dateiname
              ...
        public:
              CFILE(char *pFileName)
              { strcpy(name, pFileName); }
              ...
};
```

Das obige Beispiel zeigt den CFILE-Konstruktor als **Inline Funktion** implementiert. Der Compiler wird den Code an jeder Stelle einsetzen, an der die Funktion verwendet wird. Sie verhält sich quasi wie ein Makro. Dadurch wird der Gesamtcode größer, die Ablaufgeschwindigkeit jedoch erhöht. Dies lohnt sich besonders, wenn die Funktion innerhalb von Schleifen wiederholt aufgerufen wird.

Alternativ kann die **Inline Funktion** auch außerhalb der Klasse definiert werden. Dazu dient der "**inline**"**-**Operator:

```
class CFILE
{
        private:
              char  name[FNAME_LEN];       // Dateiname
              ...
        public:
              CFILE(char *pFileName);
              ...
};

Inline CFILE(char *pFileName)
{
        strcpy(name, pFileName);
}
```

- **Inline-**Funktionen sollten sich auf wenige Befehle (am besten nur einen) beschränken. Es ist auch möglich, **Inline Funktionen** rekursiv zu verwenden. In diesem Fall wird der **Inline Operator** allerdings vom Compiler ignoriert, die Funktion also als "ganz normale" behandelt.

Felder von Objekten

Bisher wurden Objekte immer als Individuen betrachtet. Tatsächlich können aber auch **Felder (Arrays)** von Objekten angelegt werden, sofern für diese Klasse ein Konstruktor definiert ist, wird dieser auch eingesetzt. Bezüglich Konstruktoren und Feldern von Objekten gibt es aber einige Sonderbehandlungen.

Initialisierung statischer Felder von Objekten

Die Angabe:

```
CFILE LogFile[5];                          // Erzeugung eines Arrays
```

erzeugt ein Feld (**Array**) des Objektes CFILE. Diese Definition erfordert es, daß entweder kein definierter Konstruktor oder ein Standardkonstruktor ohne Parameter existiert. Der Compiler würde dann diesen Konstruktor einsetzen.

Ist das nicht ausreichend, dann können die Elemente auch explizit initialisiert werden. Für jedes Feldelement wird der Konstruktor aufgerufen:

```
CFILE LogFile[3] =                         // Erzeugung eines Arrays
{
      CFILE("meinfile.txt", SH_DENYWRITE),
      CFILE("file2.txt", O_RDONLY),
      CFILE("file3.txt", O_RDONLY)
}
```

Dazu gibt es mehrere Varianten. Die folgende Form nutzt die verschiedenen, überladenen Konstruktoren der Klasse:

```
CFILE LogFile[3] =                         // Erzeugung eines Arrays
{
      CFILE("meinfile.txt", SH_DENYWRITE),
      CFILE("file2.txt"),
      CFILE("file3.txt")
}
```

Die folgende Version ist eine Kurzschreibweise:

```
CFILE LogFile[3] =                         // Erzeugung eines Arrays
{
      CFILE("meinfile.txt", SH_DENYWRITE);
      "file2.txt",
      "file3.txt"
}
```

In dieser Version wird für das zweite und dritte Element der Konstruktor eingesetzt, der nur ein Parameter vom Typ "**Text**" besitzt.

Initialisierung dynamischer Felder von Objekten

Anstatt **Felder von Objekten** statisch zu erzeugen, können sie auch mittels des Operators "**new**" alloziert werden. In diesem Fall entsteht aber das Problem, daß keine Konstruktoren explizit angegeben werden können.

```
CFILE *pLogFile new CFILE[5];              // Erzeugung eines Arrays
```

Es wird grundsätzlich der Standardkonstruktor ausgeführt, eine explizite Angabe der Parameter fällt weg.

Es gibt aber zwei Möglichkeiten, dynamisch erzeugte **Arrays** zu initialisieren:

- Definition eines Standardkonstruktors, ohne Parameter, der die notwendigen Aufgaben erfüllt.
- Definition einer "gewöhnlichen" Funktion (z.B. **init()**), die im Anschluß an die Felderzeugung explizit aufgerufen wird. C++ stellt hier keine Unterstützung zur Verfügung. Diese Funktion **init()** müßte über eine for-Schleife auf jedes Feldelement angewendet werden.

Zugriff auf Felder

Der Zugriff erfolgt wie auch unter C üblich durch Angabe des Operators: Klammer "(" und ")". Weitere Zugriffe, z.B. der Aufruf einer Elementfunktion, müssen mit dem Operator-Pfeil "->" oder Punkt "." gemacht werden:

```
CFILE LogFile[3];                              // Instanz von CFILE erzeugen
CFILE * pInputFile;                            // Zeiger auf Instanz CFILE
pInputFile = new CFILE[5];                     // Instanz von CFILE erzeugen
LogFile[1].Open("LogFile",O_APPEND);           // Logfile öffnen
pInputFile[0] -> Open("InputDat",O_READONLY);  // InputFile öffnen
pInputFile[0] -> Read(buffer,sizeof(buffer));  // InputFile lesen
```

Abgeleitete Klassen

Eine der mächtigsten Merkmale von C++ ist die Verwendbarkeit von Klassen als Grundsteine von völlig neuen Klassen.

```
class CBROWSE : CFILE            // CBROWSE von CFILE abgeleitet
{
          private:
                    short curline;
                    ...
          public:
                    CBROWSE(void);
                    ~CBROWSE(void);
                    OpenFile(char *);
};
```

Im obigen Beispiel hat die **CBROWSE-**Klasse Zugriff nicht nur auf alle eigenen Elemente, sondern auch auf alle **CFILE-**Klassenfunktionen, die als "**public**" oder "**protected**" in **CFILE** definiert wurden. Die folgende Tabelle zeigt die Sicherheitsbereiche für die Basis- und abgeleitete Klasse.

Basisklasse	**Abgeleitete Klasse**
Private	Nicht sichtbar in der abgeleiteten Klasse.
Protected	Sichtbar als "**private**" in der abgeleiteten Klasse.
Public	Sichtbar als "**protected**" in der abgeleiteten Klasse.

Im obigen Beispiel wäre die **CBROWSE-Klasse** in der Lage auf alle Daten und Funktionen zuzugreifen, die in der **CFILE-Klasse** als "**protected**" oder "**public**" definiert wurden. Die Anwendung wiederum wäre nicht in der Lage, auf irgendwelche der Daten oder Funktionen in der **CFILE** zuzugreifen, ohne eine der "**public**"**-**Klassenfunktionen der **CBROWSE**-**Klasse** zu verwenden. Dies ist das Standard-Vererbungsprotokoll für die Sicherheitsbereiche bei Klassen.

Die abgeleitete Klasse erbt alle Elemente aus der Basisklasse mit den Ausnahmen:

Konstruktoren
Die **Basis-Konstruktoren** stehen in der abgeleiteten Klasse nicht mehr zur Verfügung: Sie werden also nicht vererbt, sie sind aber über eine syntaktische Erweiterung immer noch im Zugriff (siehe "Namenskonflikte" unter "Mehrfachvererbung" und "Virtuelle Funktionen").

Destruktoren
Hier gilt das gleiche wie für **Konstruktoren**, sie werden nicht vererbt. Aber ein Standardverhalten des Compilers ruft bei der Löschung eines Objektes automatisch den **Destruktor** aus der Oberklasse auf.

Zuweisungsoperatoren
Verfügt eine Klasse über einen **überladenen Zuweisungsoperator** (Operator-Gleichheitszeichen =), dann wird dieser nicht vererbt. Er muß speziell angepaßt und für die abgeleitete Klasse redefiniert werden.

Anpassung von Klassenvererbung

Die Standard-Sicherheitsvorschriften können in der Definition der abgeleiteten Klasse übergangen werden:

```
class CBROWSE : public CFILE      // CBROWSE von CFILE abgeleitet
{
            private:
                        short curline;
                        ...
            public:
                        CBROWSE(void);
                        ~CBROWSE(void);
                        OpenFile(char *);
};
```

Im obigen Beispiel sind alle "**public**"**-**Funktionen von **CFILE** ebenfalls "**public**" für Anwendungen, die die **CBROWSE-**Klassen verwenden.

Die folgende Tabelle zeigt das vollständige Vererbungsschema:

Zugriff auf Basisklasse	**Zugriffsmodifizierer (Ableitungsart)**	**Zugriffsrechte auf abgeleitete Klasse**
Public	***Public***	***Public***
Private	Public	Kein Zugriff
Protected	***Public***	***Protected***
Public	Private	Private
Private	Private	Kein Zugriff
Protected	Private	Private
Public	Protected	Protected
Private	Protected	Kein Zugriff
Protected	Protected	Protected

In 95% der Fälle werden die beiden **fett** und ***kursiv*** hervorgehobenen Fälle verwendet. Alle anderen haben praktisch keine Bedeutung.

Virtuelle Funktionen

Virtuelle Funktionen bieten für Funktionen der Basisklasse die Möglichkeit, die passenden Eigenschaften oder Verhaltensweisen für die aktuell abgeleitete Klasse anzunehmen.

```
class derived : public base1, private base2
{
        private:
              char  name[FNAME_LEN];       // Dateiname
              ...
        public:
              CFILE(char *pFileName)
              { strcpy(name,pFileName); }
              virtual short Reset(void);
};

class CBROWSE : CFILE                      // CBROWSE, abgeleitet von CFILE
{
        private:
              short curline;
              ...
        public:
              CBROWSE(void);
              ~CBROWSE(void);
              OpenFile(char *);
              short Reset(void);
};

short CBROWSE::Reset(void)
{
         CFILE::Reset();
         curline = 0;
}
```

Virtuelle Funktionen können also in den abgeleiteten Klassen überschrieben werden. Ohne das Schlüsselwort "**virtual**" ergäbe dieser Versuch eine Fehlermeldung. Es ist jedoch weiterhin möglich, die Funktion der Basisklasse aufzurufen, wie im Beispiel gezeigt. Dazu muß aber die Klasse, der die Funktion angehört, explizit erwähnt werden: **`CFILE::Reset()`**.

- Ohne explizite Angabe der Klasse wird immer die Funktion der eigenen Klasse angesprochen, im obigen Fall würde dies zu einer endlosen Rekursion und zu Stacküberlauf führen. Dies ist ein beliebter Fehler. Sehr wichtig wird diese syntaktische Erweiterung bei der Mehrfachvererbung (siehe "Namenskonflikte" unter "Mehrfachvererbung").

Konstruktoren und Destruktoren abgeleiteter Klassen

Initialisierungslisten

Da **Konstruktoren** nicht vererbt werden, muß es eine Möglichkeit geben, die **Konstruktoren** der Basisklassen aufzurufen und mit den korrekten Parametern zu versorgen.

Man bedient sich hier demselben Syntax wie zur Initialisierung von **Member-Variable**, jedoch mit einer etwas anderen Bedeutung. Der **Konstruktor** der abgeleiteten Klasse trägt Sorge dafür, daß die Parameter für den **Konstruktor** der Basisklasse ausreichend versorgt sind.

```
class base
{
        public:
              base(char *);                 // Konstruktor
              base();                       // Konstruktor
              ~base();                      // Destruktor
};

class derived : base
{
        public:
              derived(int i, char *) :
              ~derived();
};

// derived-Konstruktor gibt die notwendigen Parameter an den base-Konstrukor
// weiter.

derived::derived(int i, char *n) : base(n)
{
        // hier kann I und n verwendet
        // werden
}

derived::~derived()
{
        // ...
}
```

Der **Operator-Doppelpunkt :** leitet eine **Initialisierungsliste** ein. Nach dem **Operator-Doppelpunkt :** beim **Konstruktor** für die abgeleitete Klasse kann eine Kommaliste folgen. Sie wird, wie bei den **Member-Objekten**, **Initialisierungsliste** genannt.

- Die **Initialisierungsliste** dient ausschließlich dem Initialisieren von **Klassen-Membern**, unabhängig davon, ob es **Member-Variable** oder **Konstanten** sind, ob Vererbung vorliegt oder nicht. Die Syntax ist immer die gleiche und wird für alle Aspekte der Initialisierung von Objekten und deren Elementen eingesetzt.

Aufrufreihenfolge

Die in der Initialisierungsliste aufgeführten **Konstruktoren** werden nach einer festen Reihenfolge ausgeführt. Im Beispiel ist es zwar nicht explizit genannt, aber es können auch weitere **Konstruktoren** für den Fall der mehrfachen Vererbung aufgelistet sein.

- **Basisklassen-Konstruktoren** werden vor den **Konstruktoren** der abgeleiteten Klassen aufgerufen.

- **Destruktoren** werden zuerst für die abgeleitete Klasse, dann für die **Basisklasse** aufgerufen.

Die Ausführungsreihenfolge von **Konstruktoren** und **Destruktoren** ergibt sich notgedrungen. Bei den **Konstruktoren** ist die Idee dahinter, daß zuerst die allgemeine Initialisierung der **Basisklasse** gemacht werden soll und anschließend die spezielle Initialisierung durch den **Konstruktor** der abgeleiteten Klasse. Für die **Destruktoren** ist es umgekehrt: Zuerst wird der **Destruktor** der abgeleiteten Klasse und dann der **Destruktor** der Basisklasse ausgeführt.

- Nutzen Sie die Möglichkeit, in der **Initialisierungsliste** des **Konstruktors** einer abgeleiteten Klasse, einen **Konstruktor** der unmittelbar vorangehenden Klasse aufzurufen. Da Sie diesem **Konstruktor** der Basisklasse in der **Initialisierungsliste** Parameter übergeben können, ersparen Sie sich unnötige Funktionsaufrufe. Sie können damit Ihr Programm erheblich optimieren und beschleunigen.

Virtuelle Destruktoren

Wenn für jede Basis- und abgeleitete Klasse **Destruktoren** vereinbart wurden, dann werden diese in umgekehrter Reihenfolge zu den **Konstruktoren** aufgerufen. Endet die Lebensdauer eines Objektes, dann wird zuerst der **Destruktor** der abgeleiteten Klasse und anschließend der **Destruktor** der Basisklasse aufgerufen.

Nun sind aber **dynamische Objekte** im Spiel, die mit "**new**" erzeugt wurden und mit "**delete**" entfernt werden. Es kann ein Problem auftreten, wenn "**delete**" über dem Basiszeiger aufgerufen wird. Der Compiler erzeugt einen Aufruf des **Destruktors** der Basisklasse auch dann, wenn der Basisklassenzeiger auf ein Objekt einer von der Basisklasse abgeleiteten Klasse zeigt.

Das Problem wird aus der Welt geschafft, indem man den **Destruktor** als "**virtual**" deklariert, um virtuelle **Destruktoren** zu erhalten. Damit werden alle **Destruktoren** der abgeleiteten Klassen ebenfalls virtuell, obwohl sie nicht den gleichen Namen wie der **Destruktor** der Basisklasse besitzen.

Das ist eine Ausnahme und nur für **Destruktoren** gültig. Wenn "**delete**" auf einen Basisklassenzeiger angewendet wird, wird der richtige **Destruktor** aufgerufen, unabhängig davon, auf welches Objekt der Zeiger gerade zeigt. Also ist der dynamische Typ des Zeigers ausgewertet worden, ein Basisprinzip des **Polymorphismus** (siehe "Dynamischer und statischer Zeigertyp").

Mehrfachvererbung

In den meisten Fällen reicht die direkte **Vererbung** von Klassen in der Praxis aus. In anderen Fällen sind die nachzubildenden realen Systeme aber dafür zu komplex. Die **Mehrfachvererbung** erlaubt das Erzeugen neuer Klassen aus mehreren Basisklassen.

Beispiele

- Die Klasse **Spielzeugauto** leitet sich aus **Spielzeug** und **Auto** ab.
- Die Klasse **Steuerzahler** leitet sich aus **Mitarbeiter** und **Einkommen** ab.
- Die Klasse **Dialogfenster** leitet sich aus **Fenster** und **Textfeld** ab.

Die abgeleiteten Klassen vereinen das Verhalten von allen in ihnen vereinten Basisklassen. Sie erben alle Variablen und Funktionen aus den Basisklassen. Einschränkungen gibt es durch die Vererbungsgesetze. Die Gesetze sind jedoch die gleichen wie bei der wiederholten Vererbung. Die Zugriffsmodifizierer **public**, **protected** und **private** sind ebenfalls anwendbar.

Die Besonderheiten werden in den folgenden Abschnitten beschrieben.

Syntax der mehrfachen Vererbung

Eine mehrfach vererbte Klasse wird durch folgenden Syntax erzeugt:

```
class derived : public base1, private base2
{
                ...
};
```

Der Operator ":" tritt hier auf, um die **Vererbung** zu vereinbaren. Nach dem Operator kann eine beliebige Liste von Klassen stehen. Vor jeder Klasse ist der Zugriffsmodifizierer anzufügen. Ein Fehlen wird mit dem Schlüsselwort "**private**" aufgefüllt.

- Die **Sichtbarkeit** der Klassenelemente wird wie bei der **Einfachvererbung** bestimmt. Ebenso die Zugriffe auf Klassenelemente unterscheiden sich nicht gegenüber der wiederholten Vererbung.

Konstruktoren bei Mehrfachvererbung

Initialisierungslisten

Nicht nur die Syntax für die **Mehrfachvererbung** wiederholt sich, auch die Syntax für den Einsatz der **Konstruktoren** aus den Basisklassen ist analog zu den **Initialisierungslisten**. Zu jeder Basisklasse kann ein **Konstruktor** angegeben werden, der dann bei der Objektgenerierung ausgeführt wird. Wird für eine Basisklasse kein **Konstruktor** genannt, erfolgt ein impliziter Aufruf des **Standardkonstruktors** der jeweiligen Basisklasse.

Der Name jeder Basisklasse muß bei der Initialisierung durch den **Konstruktor** der abgeleiteten Klasse angegeben werden.

Aufrufreihenfolge

- Die **Konstruktoren** der Basisklassen werden in derselben Reihenfolge aufgerufen, wie sie in der Definition des **Konstruktors** der abgeleiteten Klasse aufgeführt sind.
- Wenn noch weitere Elemente zu initialisieren sind (**Member-Objekte** usw.), so rücken diese zeitlich nach hinten während der **Initialisierung**.
- **Destruktoren** werden bei der **wiederholten Vererbung** in der umgekehrten Reihenfolge aufgerufen.

```
class base1
{
        public:
              base1(int, char *);
              // Implementierung
};

class base2
{
        public:
              base2(int, float);
              // Implementierung
};

class derived : base1, base2
{
        public:
              derived(char *s, int i) :
                    base1(i, s),
                    base2(i, 5.31);
              // Weitere Klassenelemente
};
```

Namenskonflikte

Eine **mehrfach abgeleitete Klasse** enthält indirekt mehrmals Basisklassen. Wenn in zwei der Basisklassen das gleiche Element beschrieben ist (Funktion, Variable), dann kann der Compiler nicht mehr eindeutig die Zugriffe auf die Elemente zuordnen. Es liegt ein Konfliktfall auf Namensbasis vor.

Die mehrfache Vererbung führt potentiell zu Namenskonflikten unter den folgenden Voraussetzungen:

- Gleiche Namen der **Funktionen** und **Variablen** in den **Basisklassen**.
- Gleiche öffentliche **Schnittstellen** in den **Basisklassen**.

In C++ kennt man genau eine Lösungsmöglichkeit: Explizite Angabe der zugehörigen **Basisklasse** vor dem Name der Funktion oder der Variablen ist notwendig. ***Für diese explizite Angabe ist einzig und allein der Programmierer verantwortlich.***

```
class fruit
{
        public:
              virtual char *identify()
              {
                    return "fruit";
              }
              ...
}

class tree
{
        public:
              virtual char *identify()
              {
                    return "tree";
              }
              ...
}

class apple : public fruit, public tree
{
              ...
}

apple *az = new apple;                    // Zeiger vereinbaren
az -> identify();                         // Fehler durch Mehrdeutigkeit
```

Es gibt zwei Möglichkeiten zur Fehlerbehebung:

```
az -> fruit :: identify();                // Übliche Lösung per Klassennamen
((fruit *) az) -> identify();             // Unkonventionelle Lösung per
                                          // Casting
```

- **Mehrdeutigkeiten** müssen vom Entwickler selbst aufgelöst werden. Er hat vom System keine Unterstützung zu erwarten.

Container-Klassen und umfassende Klassen

Container-Klassen oder **umfassende Klassen** sind Klassen, die die Definition einer anderen Klasse oder Struktur enthalten. Diese ist dann nur innerhalb dieser **Container-Klasse** bekannt.

Ein Beispiel könnte eine Klasse sein, die einen binären Baum implementiert:

```
class CTREE
{
          private:
                    struct STNODE                    // Die umschlossene Klasse
                    {
                          PVOID pvData;
                          struct STNODE *pstLeft;
                          struct STNODE *pstRight;
                    };
                    typedef struct STNODE TNODE_T;
                    typedef TNODE_T *TNODE_P;
                    typedef TNODE_T **TNODE_PP;
                    TNODE_P pstHead;
                    TNODE_P pstNode;
                    ...
          public:
                    CTREE(VOID);
                    ~CTREE(VOID);
                    SHORT Delete(PVOID);             // Eintrag entfernen
                    SHORT Find(PVOID,PPVOID);        // Eintrag finden
                    SHORT Insert(PVOID);             // Eintrag einfügen
                    ...
};

typedef CTREE * CTREEP;
typedef CTREE ** CTREEPP;
```

Im Beispiel des binären Baumes war es nicht zweckmäßig, daß jeder Knoten des Baumes eine eigene Instanz der **CTREE-**Klasse ist. Deshalb ist jeder Knoten in der **CTREE-**Klasse enthalten und die **CTREE-**Klasse manipuliert alle **STNODE**-Strukturen, die darin enthalten sind.

Umfassende Klassen und Member-Objekte

Selbstverständlich können auch Klassen als Datenelemente einer anderen Klasse definiert werden. Diese bezeichnet man als **Member-Objekte** oder **eingebettete Objekte**. Die deklarierende Klasse wird als **umfassende Klasse** bezeichnet.

```
class CADRESS
{
            private:
                        char street[40];
                        char City[40];
                        char ZIP[15];
                        ...
            public:
                        CADRESS (VOID);
                        ~ CADRESS (VOID);
                        ...
};

class CPERSON
{
            private:
                        CADRESS address;         // Eingebettete Klasse
                        char name[40];
                        int age;
                        ...
            public:
                        CPERSON (VOID);
                        ~ CPERSON (VOID);
                        ...
};
```

Die Klasse "**CPERSON**" hat hier als Eigenschaft die Klasse "**CADRESS**" als **eingebettetes Objekt**.

Initialisierungslisten

Probleme gibt es jetzt bei der Initialisierung dieser **Member-Objekte**. Für jedes zur Laufzeit erzeugte Objekt der umfassenden Klasse muß automatisch ein **Member-Objekt** erzeugt werden. Wenn nun der **Konstruktor** des **Members** Parameter verlangt, muß es eine Möglichkeit geben, diese über die **Konstruktoren** der umfassenden Klasse an die **Member-Objekte** weiterzureichen.

Dazu wird der Syntax für **Konstruktoren** erweitert:

```
CPERSON( Typ p1, Typ p2, ...) : CADRESS(p1, p2)
{
        // Definition des Konstruktors
}
```

- Nach dem Zeichen : muß das zu initialisierende Objekt unter Angabe der Parameternamen für seinen **Konstruktor** genannt werden. Selbstverständlich muß in der Klasse des Objektes ein passender **Konstruktor** definiert sein.
- Die Parameter für das zu **initialisierende Objekt** müssen nicht mehr mit einem Datentyp deklariert werden, denn das ist schon in der Parameterliste des **Konstruktors** der umfassenden Klasse gemacht worden. Es genügt die namentliche Nennung der Parameter.
- Beachten Sie die **Namensgleichheit** zwischen den beiden Parameterlisten. Der Compiler macht hier eine Zuordnung über **Namensgleichheit**, um die Parameterwerte an den **Konstruktor** des **Member-Objektes** weiterzureichen.
- Die Parameter des **Konstruktors** des **Member-Objektes** müssen nicht in derselben Reihenfolge angegeben werden.
- Es können auch Berechnungen mit den Parametern durchgeführt werden oder diese an Funktionen übergeben, deren Ergebnis dann vom **Konstruktor** verwendet wird.
- Für Punkt 4. Und 5. muß dann lediglich ein passender **Konstruktor** (der die jeweils erzeugten Typen akzeptiert) im **Member-Objekt** definiert sein.

Beispiel:

```
CPERSON( Typ p1, Typ p2, ...) : CADRESS(p1, p2 * 3, "Test", func(p1))
{
        // Definition des Konstruktors
}
```

Ausführungsreihenfolge

Member-Konstruktoren werden ausgeführt, ***bevor*** der Rumpf des **Konstruktors** der umfassenden Klasse betreten wird. Die Reihenfolge der **Member-Konstruktoren** untereinander ist nicht festgelegt. Hier wählt der Compiler eine für ihn optimale Reihenfolge aus.

Beim Löschen werden zunächst der **Destruktor** der umfassenden Klasse und ***danach*** die **Member-Destruktoren** aufgerufen.

Abstrakte Klassen

Abstrakte Klassen dienen dazu, in einer **Klassenhierarchie** semantisch ähnlichen Klassen eine gemeinsame Funktionsschnittstelle per Deklaration zuzuordnen. In einer **abstrakten Klasse** wird für die daraus abgeleiteten Klassen eine oder mehrere Funktionsschnittstellen vorgegeben. Die Definition der deklarierten Funktionsschnittstelle wird in den abgeleiteten Klassen mit klassenspezifischen Implementierungen nachgeholt. Die deklarierten Funktionen werden in der **abstrakten Klasse** nicht definiert, sondern erst in den abgeleiteten Klassen.

Eine **abstrakte Klasse** gilt als eine unvollständige Klasse und deshalb können keine Objekte dieser Klasse angelegt werden. Eine solche Klasse wird im wesentlichen eingeführt, um eine gemeinsame funktionale Schnittstellendeklaration zu erreichen, ohne diese Funktionen sofort definieren zu müssen. Zusätzlich ist es möglich, den für den **Polymorphismus** notwendigen Basiszeiger vom Typ der abstrakten Klasse zu vereinbaren.

Eine Elementfunktion wird als "**rein virtuelle Funktion**" oder auch "**pure virtual function**" bezeichnet, wenn sie folgendermaßen deklariert ist:

```
virtual Typ Funktionsname (Parameter) = 0;
```

Wichtig ist das Schlüsselwort "**virtual**" und der **Initialisierer** "= 0". Der **Initialisierer** soll ausdrücken, daß eine Definition nicht in der **abstrakten Klasse** vorgenommen wird.

- Wenn eine Klasse mindestens **eine rein virtuelle Funktion** besitzt, wird sie **abstrakte Klasse** genannt. Von einer **abstrakten Klasse** können keine Objekte angelegt werden, denn die Klasse ist in ihrer Beschreibung unvollständig.

Man beachte, daß eine **abstrakte Klasse** aber über **Konstruktoren** und **Destruktoren** verfügen kann, auch wenn keine Objekte der Klassen jemals existieren. Die **Konstruktoren** kommen aber mit der Vererbung und der Initialisierungsliste anderer **Konstruktoren** zum Einsatz. Alle weiteren Aspekte wie Vererbung und Sichtbarkeit bleiben bestehen und verhalten sich wie bereits beschrieben.

Was wird mit den **rein virtuellen Funktionen** beabsichtigt?

- **Rein virtuelle Funktionen** werden in der **abstrakten Klasse** nur deklariert, um sie in der abgeleiteten Klasse redefinieren zu können. Sie erfüllen also keinen anderen Zweck, als ein **polymorphes Interface** für die abgeleitete Klasse zur Verfügung zu stellen. Hier spielen **Polymorphismus**, **Vererbung** und **Funktionen-Schnittstelle** eng zusammen. Die **abstrakte Klasse** ist die "Top"-Klasse für den **Polymorphismus**. Man kann die **abstrakte Klasse** auch als reine Verwaltungs- und Organisationsklasse betrachten.

```
class GraphObj                              // Basisklasse
{
        public:
              virtual void draw() = 0;      // Rein virtuelle Funktion
};

class Rectangle : public GraphObj           // Abgeleitete Klasse
{
        public:
              void draw();                  // Hier muß Definition erfolgen
};


// Weitere Klassen für sinnvollen Polymorphismus
```

```
int main(void)
{
        GraphObj *ptr;                  // Basiszeiger
        // GraphObj gobj;               // nicht möglich
        Rectangle robj;                 // Objekt
        ptr = (GraphObj *)&robj;        // Zuweisung auf Basiszeiger
        ptr -> draw();                  // Aufruf: Rectangle::Draw()
}
```

Weitere Sprachelemente

C++ Eingabe/Ausgabe

Die meisten Programme kommunizieren in irgendeiner Weise durch Tastatur und Bildschirm. Die Sprache C bietet drei Variablen, **stdin**, **stdout** and **stderr**, die für die Kommunikation mit diesen "Standardgeräten" verwendet werden. Diese werden geöffnet, bevor die Anwendung durch die Ausführung von **main()** beginnt. Diese drei Werte sind **FILE-**Zeiger und können prinzipiell jede Dateioperation der C-Bibliothek ausführen.

C++-Ausgaben wurden vereinfacht gegenüber der **printf-**Funktionsfamilie von C. C++ definiert vier Klassen: **cin**, **cout**, **cerr** and **clog**, die mit den C-Dateizeigern **stdin**, **stdout**, **stderr** and **buffered stderr** korrespondieren. C++ formatiert die Ausgaben automatisch anhand des aktuellen Variablentyps.

```
/* In C */
printf("%s = %d\n", szRate, sRate);
// In C++
cout << szRate << " = " << sRate << endl;
// Eine andere C++ Alternative
cout << form("%s = %d\n", szRate,sRate);
/* In C */
scanf("%d",&sRate);
// In C++
cin >> sRate;
```

cout ist ein **ostream**-Objekt, das mit dem "Standardausgabegerät" verbunden wird (üblicherweise der Bildschirm), bevor die Programmausführung beginnt. Werte, die mit der ostream-Operator << eingefügt werden, werden in Zeichen konvertiert und anhand des Typs des eingefügten Objektes auf den Bildschirm geschrieben.

cin ist ein **istream-**Objekt, das mit dem "Standardeingabegerät" verbunden wird (üblicherweise die Tastatur), bevor die Programmausführung beginnt. Werte, die mit dem **istream**-Operator >> extrahiert werden, werden vom Standardeingabegerät gelesen und anhand des Typs des extrahierten Objektes interpretiert.

- Aber Vorsicht! Für C++ gibt es keine genaue Definition, wie die Ausgaben zu formatieren sind. Es bleibt dem Compiler überlassen, wie viele Leerzeichen vor einer Zahl ausgegeben werden etc. Daher ist die Methode mit der form()-Elementfunktion meist vorzuziehen, und die Vorteile von C++ sind wieder dahin.

Außerdem arbeiten die **cin-** und **cout-**Klassen mit dem Standardgerät, nämlich der Konsole. Das hat zur Folge, daß sie für Windowsprogrammierung so gut wie gar keine Bedeutung haben.

Referenzvariablen

C++ bietet einen Syntax für Programmierer, die sich mit dem C-Zeiger-Syntax nicht wohlfühlen, einen einfacheren Weg Anwendungen mit Zeigern zu schreiben.

```
/* In C */
short Afunc(short * psShort)
{
   *psShort++;
}

// In C++
short Afunc(short & Short)
{
   Short++;
}
```

Operator Überladung

Überladung von Operatoren

Da die C++-Klassen detailliert die Eigenschaften von und Operationen mit der Klasseninstanz definieren, erlaubt C++ die **Neudefinierung** oder **Überladung** aller Standardoperatoren (z.B.. '+', '-', '*', '/', '++', '--') für die aktuelle Klasse.

```
class CSTRING
{
      private:
           char * pChar;
           int len;
           ...
      public:
           CSTRING(const char * = 0);        // Defaultwert setzen
           ~CSTRING(delete pChar);
           void operator+(char *)
};

CSTRING::operator+(char *pC)
{
      char * pBuf;
      pBuf = new char[len=strlen(pC)+len];
      strcpy(pBuf,pChar);
      strcat(pBuf,pC);
      delete pChar;
      pChar = pBuf;
}

CSTRING Str("ABC");
Str + "DEF";                                 // Jetzt ist pChar = 'ABCDEF'
```

- Das **Überladen** von Operatoren bedeutet immer die Simulation des überladenen Operators. Das **Überladen** bietet einen Mechanismus, diese Operationen abzukürzen. Es genügt also nicht, einfach eine **Überladung** zu definieren, der notwendige Code muß ebenfalls erstellt werden.

Genauso wie bei der **Funktionsüberladung** ist es auch hier ein Anliegen, mit demselben Operator verschiedene Datentypen bearbeiten zu können, ohne jeweils neue Namen zu vergeben. Sowohl Idee als auch syntaktische Ausführung zwischen **Funktionsüberladung** und **Operatorüberladung** sind vergleichbar.

- Die Präzedenz (Ausführungsreihenfolge) und der Syntax der Operatoren kann nicht geändert werden. Der Operator Stern * z.B. hat immer eine höhere Präzedenz als der Operator Plus +.
- Neue Operator-Symbole (Tokens) können nicht definiert werden. Der gegebene Operatorumfang kann nicht erweitert werden.
- Binäre Operatoren sind binär, unäre Operatoren sind unär zu überladen. Die Wertigkeit muß also berücksichtigt werden, d.h., zum Überladen eines Operators muß mindestens ein Parameter der Operatorfunktion einen Klassentyp haben. Der Operator Plus + für Int-Typen kann man deshalb nicht zur Subtraktion redefinieren.
- In der Definition erscheint der Operator ::. Er bindet die Operator-Definition an die Klasse. Eine Operator-Definition kann selbstverständlich auch ohne Existenz der Klasse erfolgen. Dann entfällt der Operator ::. Für eine Operator-Überladung in einer Klasse gelten die Sichtbarkeitsregeln wie bei der Elementfunktion.

Liste überladbarer Operatoren

Es sind praktisch alle C++-Operatoren auch **überladbar**. Die folgende Tabelle listet diese auf:

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

Alle Operatoren haben eine bestimmte Wertigkeit. Sie sind unär (! für NOT), behandeln also nur einen Operanden, oder sie sind binär (<< für Bitschieben links), behandeln also zwei Operanden.

Einige Operatoren treten jedoch sowohl unär als auch binär auf:

+	-	*	&

Folgende Operatoren sind nicht überladbar:

.	Zugriff auf Element eines Objektes
.*	Zeiger auf Elementoperator
::	Zugriffsoperator
?:	Operator für bedingte Ausdrücke

Erweiterungen der Initialisierungslisten

- **Initialisierungslisten** treten nicht nur bei **Member-Objekten** oder **Mehrfachvererbung** auf (siehe "Initialisierungslisten" und "Initialisierung von Konstanten mit Initialisierungslisten"). Es gibt vielfältige Anwendungsmöglichkeiten, die hier aufgelistet sind.

Initialisierbare Klassenelemente

In jeder **Initialisierungsliste** können die unterschiedlichen **Klassenelemente** und **Konstruktoren** genannt werden.

Ein Ausschnitt eines **Konstruktors** einer abgeleiteten Klasse:

```
derived::derived(char *n, int m, float o, int c, char v)
: base1(n), base2(m), obj(o, c), constant(c), variable(v)
{
        // Definition des Konstruktors
}
```

- **base1** und **base2** sind **Konstruktoren** aus den Basisklassen. Es wird jeweils ein Argument übergeben, weil die **Konstruktoren** der Basisklasse entsprechend definiert sind.
- **obj** ist ein **eingebettetes Objekt** in der abgeleiteten Klasse. Der **Konstruktor** seinerseits muß passend ausgelegt sein: zwei Parameter **o** und **c** mit den Typen **float** und **int**. Der **Konstruktor** ist in der Klasse definiert, deren Element **obj** ist.
- **constant** ist eine Konstante in der abgeleiteten Klasse. Diese kann nur so mit dem Wert **c** initialisiert werden.
- **variable** ist eine Variable in der abgeleiteten Klasse. Diese kann auch so initialisiert werden, der übliche Weg wäre über eine Zuweisung im **Konstruktorrumpf**.
- Ob die genannten Namen der **Initialisierungsliste Konstruktoren**, **eingebettete Objekte** oder **Konstanten** sind, geht aus der Liste nicht hervor. Dazu muß die Klassenbeschreibung inspiziert werden.

Der Bereichsoperator "::"

- Der zweifache Doppelpunkt "::" dient auch als sogenannter **Bereichsoperator**. In **K&R** und **ANSI C** gibt es keinen korrespondierenden Konstrukt. Er erlaubt Zugriff auf globale Variablen und Funktionen auch, wenn es lokale Variablen oder Funktionen desselben Namens gibt. Die Anwendung von "::" vor dem Variablen- oder Funktionsnamen sagt dem Compiler, daß die Verwendung des globalen anstatt des lokalen Objektes erwünscht ist.

Diese Technik erlaubt die Verwendung der globalen Objekte zu jedem Zweck. Sie können für Berechnungen und als Funktionsparameter etc. verwendet werden. Es ist kein guter Programmierstil, diesen Konstrukt allzu ausgiebig zu verwenden, da es den Code schlecht lesbar macht. Es wäre besser, einen anderen Variablenamen zu verwenden, anstatt denselben Namen wiederzuverwenden.

Es kann jedoch Fälle geben, in denen die Verwendung des **Bereichsoperators** sinnvoll oder sogar unumgänglich ist.

Zeiger auf Objekte

- Objekte einer abgeleiteten Klasse haben auch den Typ der abgeleiteten Klasse, der als spezieller Typ der Basisklasse aufgefaßt wird. Diese Betrachtungsweise wird auch in C++ genutzt; dies wird bei Konvertierungen von Zeigern und Objekten deutlich.

Zeiger auf Basisklasse

Ein Zeiger auf eine abgeleitete Klasse kann ohne explizite **Typkonvertierung** einer Zeigervariablen vom Typ "**Zeiger auf Basisklasse**" zugewiesen werden. Das entspricht einem Zugriff nach unten in der Vererbungshierarchie. Das bedeutet, daß ein Objekt einer abgeleiteten Klasse genauso wie ein Objekt der Basisklasse behandelt werden kann, sofern es über Zeiger referenziert wird.

Aber hier muß aufgepaßt werden. Die Umkehrung gilt nämlich nicht. Die Zuweisung einer Zeigervariablen vom Typ "**Zeiger auf Basisklasse**" zu einer Zeigervariablen vom Typ "**Zeiger auf abgeleitete Klasse**" erfordert explizite **Typkonvertierung** durch Casting. Der Zugriff nach oben in der Vererbungshierarchie ist nicht erlaubt.

Das eben beschriebene Prinzip ist am folgenden Beispiel dargestellt. Die Klassen **B** und **D** stehen für die Basisklasse und abgeleitete Klasse:

```
class B { ... };                    // Basisklasse
class D : public B { ... };         // abgeleitete Klasse
B *bp;                              // Zeiger auf Basisklasse
D *dp;                              // Zeiger auf abgeleitete Klasse
bp = dp;                            // OK, implizite Konvertierung
dp = bp;                            // Fehler
dp = (D *)bp;                       // OK, explizite Konvertierung
```

Regeln

- Ein spezieller (unten in der Vererbungshierarchie) Typ kann einem allgemeinen Typ (oben in der Vererbungshierarchie) zugewiesen werden.
- Für die explizite **Typkonvertierung** spielt die Vererbungshierarchie eine ausschlaggebende Rolle. In ihr ist verankert, was unter speziell oder generell zu verstehen ist.
- Eine implizite **Konvertierung** ist jedoch nur auf **public**-Basisklassen möglich. Der Zugriffsmodifizierer ist auf **public** eingestellt.
- Die implizite **Typkonvertierung** kann vom Compiler vorgenommen werden.
- Für die explizite **Typkonvertierung** ist der Entwickler verantwortlich für den Cast-Operator.
- Unsachgemäße Handhabung dieser Techniken führt in der Regel zu verwirrenden Ergebnissen.

Dynamischer und statischer Zeigertyp

Jeder Zeiger in C++ kann einen dynamischen und einen statischen Typ annehmen (Zeigertyp). Der statische Typ wird bei der Zeigerdeklaration fest vorgegeben und ändert sich über die gesamte Laufzeit nicht. Dieser statische Typ ist fix an die Zeigervariable gebunden.

Der dynamische Typ ist für jede Zeigervariable vielfältig und abhängig vom jeweiligen Einsatz. Bei jeder Zuweisung kann der dynamische Typ des Zeigers sich ändern. Er ist abhängig vom Typ der rechten Seite der Zuweisung. Aber - hier muß Vorsicht walten: Ein Zeiger kann nicht jeden beliebigen dynamischen Typ annehmen, sondern die Typenauswahl ist beschränkt durch die Vererbungshierarchie.

Sehen Sie sich dazu das vorhergehende Beispiel noch einmal an.

This-Zeiger

Der **this**-Zeiger ist ein spezieller Zeiger, der jeder Elementfunktion einer Klasse zur Verfügung steht. Er zeigt immer auf das Objekt, zu dem die Elementfunktion gehört.

this ist ein Schlüsselwort, kann aber als Zeigervariable eingesetzt werden. Bei jedem Aufruf einer Elementfunktion erhält diese vom Compiler implizit die Adresse des zugehörigen Objektes zugewiesen. Auf diesen Adresswert kann der Entwickler explizit, jedoch nur lesend, über den Namen **this** zugreifen. Der Zeiger **this** hat seine Hauptaufgabe bei den Parameterübergaben in Elementfunktion. Jeder Zugriff über Elementfunktion auf die Variablen eines Objektes wird durch die implizite Dereferenzierung dieses Zeigers erst möglich.

Auch wenn der Zugriff auf die Variablen oder Funktionen der Klasse implizit immer über den **this**-Zeiger erfolgt so darf dieser auch explizit eingesetzt werden. Dieser verborgene **this**-Zeiger war bei allen Beispielen vorhanden, nur wurde nicht darauf hingewiesen.

```
void zeit::drucke_zeit()
{
        cout << _stunde;                    // implizites this
        cout << this -> _stunde;            // explizites this
        cout << (*this)._stunde;            // explizites this mit
                                            // Dereferenzierung.
                                            // this ist tatsächlich ein Zeiger
}
```

Der **this**-Zeiger kann auch als Rückgabewert einer Elementfunktion dienen. Das Funktionsergebnis ist ein Bezug auf das aktuelle Objekt der Klasse. Eine Elementfunktion kann das Objekt, zu dem sie aufgerufen wurde, als Funktionsergebnis zurückgeben. Die Anweisung für den Rückgabewert dazu sieht so aus:

```
return * this;
```

Eine Funktion muß in der dargestellten Art vollständig definiert werden:

```
zeit & zeit::ausgabe(void)
{
        // Zugriff auf Elemente mit this
        return *this;
}
```

Der Friend Operator

Eine Funktion außerhalb der Klasse kann als **friend-Funktion** (**Freundfunktion**) der Klasse definiert werden und so Zugriff auf die privaten Elemente der Klasse erhalten. Das öffnet das Schutzschild rund um die Daten der Klasse ein wenig, sollte also nur sehr selten angewendet werden. Es gibt Fälle, in denen ein Programm durch die Verwendung solcher **Freundfunktionen** leichter zu verstehen ist und diese einen kontrollierten Zugriff auf die Daten erlauben. Man kann einzelnen Funktionen diesen Status verleihen, aber auch Elementen anderer Klassen oder gar ganzen Klassen. Weder ein **Konstruktor** noch ein **Destruktor** kann allerdings aus verständlichen Gründen eine **Freundfunktion** sein.

Friend-Klasse

Eine Klasse, bei der sämtliche Methoden Zugriff auf die Elemente einer anderen Klasse haben, wird als **Friend-Klasse** (**Freundklasse**) bezeichnet. **Freundklasse** muß innerhalb der Klasse als solche deklariert werden, die Zugriff auf ihre Elemente gewährt - beispielsweise:

```
class MyClass
{
        friend class MyOtherClass;          // MyOtherClass als
                                            // friend deklariert
        private:
             int topSecret;
}

class MyOtherClass
{
        public:
             void change( MyClass yc );
}

void MyOtherClass::change( MyClass yc )
{
        yc.topSecret++;                     // Zugriff ist erlaubt
}
```

- Eine derartige "Freundschaft" beruht nur dann auf Gegenseitigkeit, wenn beide Klassen die jeweils andere als **friend** deklarieren: Im zuvor verwendeten Beispiel hat **MyOtherClass** zwar Zugriff auf die Elemente von **MyClass**, aber nicht umgekehrt.
- Die Eigenschaft **friend** wird weder vererbt noch ist sie transitiv: Eine von **MyOtherClass** abgeleitete Klasse hat also nur über die Methoden von **MyOtherClass** Zugriff auf Elemente von **MyClass**; wenn **MyOtherClass** eine dritte Klasse **MyThirdClass** als **friend** deklariert, haben die Methoden von **MyThirdClass** keine zusätzlichen Zugriffsmöglichkeiten auf Elemente von **MyClass**.

Friend-Funktionen

Eine Routine, die nicht Element einer Klasse ist, aber dennoch Zugriff auf die als **private** und **protected** deklarierten Elemente dieser Klasse hat, wird als **Friend-Funktion** bezeichnet. **Friend-Funktionen** müssen innerhalb der Klasse als solche deklariert werden, die Zugriff auf ihre Elemente gewährt - beispielsweise:

```
class Komplex
{
        public:
                    friend Komplex operator+( Komplex first, Komplex second
                    );
        private:
                    float real, imag;
```

```
};
```

Zugriffs-Spezifizierer (**public**, **private** und **protected**) spielen in diesem Zusammenhang keine Rolle: Die Deklaration einer Routine als **friend** kann also an beliebigen Punkten innerhalb einer Klassendefinition erscheinen.

- Zu beachten ist, daß **Friend-Funktionen** keine Methoden sind. Sie bekommen keinen Zeiger auf das jeweilige Objekt übergeben und werden wie normale Funktionen deklariert - also ohne vorangestellten Klassennamen.

Beispiel:

```
Komplex operator+( Komplex first, Komplex second )
{
         return Komplex( first.real + second.real,
         first.imag + second.imag );
}
```

Die in diesem Beispiel definierte **Friend-Funktion** operator+ muß die beteiligten Objekte also über vollständige Adressen (first.real anstelle von real) ansprechen. Wie zu sehen, hat sie aber Zugriff auf die Felder real und imag, die von der Klasse Komplex als private deklariert sind. Da die Deklaration als **friend** innerhalb der jeweiligen Klasse stattfindet, erscheint das Schlüsselwort selbst nicht in der Definition der Funktion.

Statische Klassenelemente

Statische Elemente charakterisieren nicht wie die bisher kennengelernten Elemente einen Teil eines Objektes, sondern sie gehören zu der sie beschreibenden Klasse. Solche Elemente stehen dann nicht nur einem Objekt zu seiner individuellen Verfügung bereit, sondern sind lediglich einmal vorhanden für die gesamte Klasse. Wenn sich der Wert eines statischen Elementes ändert, können sofort alle Objekte dieser Klasse auf den geänderten Wert zugreifen.

Stellen Sie sich vor, sie wollen eine Klasse **Bankkonto** erstellen, zur Beschreibung von Konten einer Bank. Jedes einzelne Objekt repräsentiert dann ein individuelles Bankkonto eines Bankkunden. Die Klasse soll folgende Eigenschaften haben:

- dynamische Variable für Name und Kontostand.
- statische Variable für Zinssatz. Der Zinssatz soll für alle Konten gleich sein.
- Zugriffsfunktionen, um die Variablen zu lesen und zu schreiben.

Zur Lösung dieser Aufgabe muß man zwei Aspekte betrachten: statische Variable und statische Funktionen.

Statische Klassenvariable

Es erscheint sicherlich nicht sinnvoll, unter diesen Voraussetzungen die Variable **Zinssatz** in jedem Objekt zu halten. Drei Gründe sprechen dagegen:

- Die Variable **Zinssatz** soll allgemeinen Zugriff von allen Objekten der Klasse **Bankkonto** erlauben und darf nicht individuell gehalten werden.
- Tritt eine Wertänderung dieser Variablen ein, so muß nur einmal an einer Stelle geändert werden und alle Objekte haben sofort Zugriff auf den neuen Wert. Das erspart viel Aufwand, denn es muß nicht in jedem Objekt der Wert geändert werden.
- Zusätzlich spart es Speicherplatz, denn diese statische Variable ist im System nur einmal vorhanden. Besonders bei großen Objektmengen ist das ein Punkt zum Nachdenken.

Wird eine Variable einer Klasse mit dem Schlüsselwort **static** als statisches Element deklariert, dann wird für die gesamte Klasse und alle Objekte dieser Klasse diese statische Variable nur einmal angelegt. Dennoch kann sie von den Elementfunktionen der Klasse wie eine ganz normale Klassenvariable angesprochen werden. Darüber hinaus gelten weiterhin die definierten Zugriffsrechte. Wird eine Klassenvariable als **private** deklariert, dann haben - wie bekannt - nur die Elementfunktionen darauf Zugriff.

Da statische Klassenvariablen bereits vor dem ersten Objekt und vor dem ersten **Konstruktoraufruf** existieren, muß für sie eine besondere Art der Initialisierung geschaffen werden. Ein weiteres Hindernis wäre außerdem, ein **Konstruktor** wird mehrfach aufgerufen und würde die statische Klassenvariable eventuell mehrfach mit unterschiedlichen Werten initialisieren.

- Syntaktisch gesehen werden statische Klassenvariablen wie globale Elemente initialisiert. Lediglich der Klassenname muß der statischen Klassenvariablen vorangestellt werden. Es spielt für die Initialisierung auch keine Rolle, ob das statische Datenelement mit **public**, **protected** oder **private** deklariert ist. Diese Sichtbarkeit wirkt sich nur beim Zugriff aus, nicht bei der Initialisierung. Ein Beispiel soll diese Verhältnisse verdeutlichen:

```
class Bankkonto
{
        private:
             static float Zinssatz;          // Statische Deklaration als
                                             // private
        public:
             float ertrag()                  // Berechnungsfunktion
             {
             // Berechnung mit Zinssatz
             }
}
                                             // Initialisierung
float Bankkonto::zinssatz = 0.07;

int main(void)
{
        Bankkonto bkobj;
        cout << bkobj.ertrag();
        bkobj.zinssatz;                      // Fehler, da Variable private
        return 0;
}
```

- Die Initialisierung der statischen Variablen muß immer in einer Quellcode-(CPP)Datei erfolgen, nicht in einem Header (H). Header werden meist an mehreren Stellen eingebunden und die Variable würde dann mehrfach initialisiert. Dies spiegelt sich dann

in Linker-Fehlermeldungen wider.

Zusammenfassung

- Grundsätzlich besitzt jedes Objekt einer Klasse eine eigene Kopie der in der Klasse deklarierten Variablen. In einigen Fällen ist es jedoch sinnvoll, daß alle Objekte einer Klasse gemeinsame Variable besitzen (entspricht einem gemeinsamen Speicherbereich auf Klassenebene).
- Ein statisches Klassenelement existiert nur einmal, unabhängig davon, wie viele Objekte einer Klasse existieren.
- Ein statisches Klassenelement wird mit **static** innerhalb der Klasse deklariert, gleichzeitig werden die Zugriffsrechte zugewiesen.
- Ein statisches Klassenelement wird außerhalb der Klassendeklaration initialisiert. Eine fehlende Initialisierung führt zum Linker-Fehler.
- Auch wenn das statische Klassenelement eine **private**-Variable ist, kann man bei der Initialisierung mit dem Operator :: zugreifen und den Speicherplatz vorbelegen.
- Ein statisches Klassenelement wird separat von der Klasse gespeichert.

Statische Klassenfunktion

Elementfunktionen, die ausschließlich auf statische Elemente zugreifen, können auch mit dem Schlüsselwort **static** versehen werden und werden **statische Klassenfunktionen** genannt.

```
class Bankkonto
{
        private:
             Static float Zinssatz;          // Statische Deklaration als
                                             // private
        public:
             Static float lese_zinssatz()    // Berechnungsfunktion
             {
                  return Zinssatz;
             }
}
                                             // Initialisierung
float Bankkonto::zinssatz = 0.07;

void main()
{
        cout << lese_zinssatz();             // Aufruf der statischen Funktion
        Bankkonto bkobj;                     // Jetzt erst Objekt anlegen
        cout << bkobj.zinssatz;              // Fehler, da Variable private
}
```

Die **statischen Elementfunktionen** haben eine interessante Besonderheit: Eine **statische Elementfunktion** kann ohne Objektteil aufgerufen werden. Das ist möglich, weil die Ausführung dieser **statischen Elementfunktion** von der Existenz dynamischer Variablen der Objekte unabhängig ist. **Statische Elementfunktionen** haben prinzipiell keinen Zugriff auf dynamische Elemente der Objekte ihrer Klasse und müssen zudem ohne **this**-Zeiger auskommen.

Demzufolge können **statische Elementfunktionen** vor der Existenz eines ersten Objektes die **statischen Klassenelemente** initialisieren.

Neue Sprachelemente

Funktions- und Klassen-Templates (Schablonen)

Schablonen (**Templates**) gehören zum Sprachumfang neuerer C++-Implementierungen. Sie sind nicht in C verfügbar.

```
template < template-argumente-liste > deklaration
```

Templates, auch "generische" oder "parametrisierte" Typen genannt, dienen zur Erzeugung einer Familie verwandter Funktionen oder Klassen.

Funktions-Templates

Sehen wir uns die Funktion max(x, y) an, die das größere der beiden Argumente zurückgibt. x und y können von einem beliebigen, sortierbaren Typ sein. Da C++ eine streng auf Typen ausgerichtete Sprache ist, müssen die Argumenttypen zum Zeitpunkt der Compilierung deklariert werden.
Ohne Templates sind viele überladene Versionen von max erforderlich, d.h., eine Version für jeden unterstützten Datentyp, obwohl der Programmcode im Prinzip für jede Version identisch ist. Jede Version vergleicht die Argumente miteinander und gibt das größere zurück.

Um dieses Problem zu umgehen, könnte ein Makro verwendet werden:

```
#define max(x,y) ((x > y) ? x : y)
```

Die Verwendung von #define umgeht jedoch die Typüberprüfung, die ein wesentlicher Vorteil von C++ gegenüber C ist. Der Einsatz von Makros ist in C++ nahezu überflüssig. Die Aufgabe von max(x, y) liegt im Vergleichen kompatibler Typen. Das Makro seinerseits erlaubt jedoch leider auch einen Vergleich von int- und struct-Typen, die nicht kompatibel sind. Dies führt in der Regel zu seltsamen bis katastrophalen Ergebnissen, äußert sich aber nicht in einer Compiler-Fehlermeldung.

Ein weiteres Problem bei der Verwendung des Makros liegt darin, daß eine Ersetzung ausgeführt wird, wo dies gar nicht erwünscht ist. Wenn Sie statt dessen ein **Template** verwenden, können Sie eine Schablone für gleichartige überladene Funktionen definieren, indem Sie den Datentyp selbst als Parameter verwenden:

```
template <class T> T max(T x, T y)
{
        return (x > y) ? x : y;
};
```

Der Datentyp wird durch das **Template-Argument** <class T> repräsentiert. Beim Einsatz in einer Anwendung generiert der Compiler die entsprechende Funktion gemäß dem im Aufruf verwendeten Datentyps:

```
int i;
Myclass a, b;
int j = max(i,0);                    // Integer-Argumente
Myclass m = max(a,b);                // Argumente vom Typ Myclass
```

Für <class T> kann jeder beliebige Typ (nicht nur Klassen) verwendet werden. Der Compiler ruft dann den entsprechenden operator>() auf, so daß Sie max mit Argumenten jedes beliebigen Typs einsetzen können, für den der operator>() definiert ist.

Klassen-Templates

Mit einem **Klassen-Template** (auch **generische Klasse** oder **Klassengenerator** genannt) kann

eine **Schablone** für Klassendefinitionen erstellt werden. **Generische Container-Klassen** sind dafür ein gutes Beispiel. Sehen Sie sich das folgende Beispiel einer Vektorklasse an (ein eindimensionales Array). Die grundlegenden Operationen (Einfügen, Löschen, Indizieren usw.), die für den Typ ausgeführt werden, sind immer dieselben, unabhängig davon, ob Sie einen Vektor mit Integerwerten oder mit einem anderen Typ haben. Der Elementtyp wird als Typ-Parameter für die Klasse behandelt und das System erzeugt typsichere Klassendefinitionen: Wie bei **Funktions-Templates** kann eine explizite **Template-Klassendefinition** die automatische Definition eines gegebenen Typs überschreiben:

```
class Vector<char *> { ... };
```

Das Symbol Vector muß immer den Datentyp in spitzen Klammern bei sich haben. Es kann nicht alleine stehen; ausgenommen sind nur einige Fälle in der ursprünglichen **Template-Definition**.

Einführung in die Windows API und das Windows SDK

Dieser Abschnitt des Seminars soll ein Verständnis für die Grundlagen und Beweggründe der **MFC** wecken. Die **Einführung in die Windows API** (Windows Application Programming Interface = Windows Schnittstelle zur Anwendungsprogrammierung) erlaubt Einblicke in die Hintergründe von Windows und stellt außerdem eine gute Übungsmöglichkeit zur Handhabung der Werkzeuge von **VC++ 5.0** dar. Sie lernen die wichtigsten Funktionen der **Windows API** kennen. Später werden sich alle diese Dinge in **MFC** wiederfinden. Dies ist besonders dann wichtig, wenn Sie standardmäßige bereitgestellte Funktionalitäten von **MFC** abwandeln wollen.

Die Windows API und ihre Tücken

- Microsoft Windows wurde in C entwickelt und besitzt daher eine **C Schnittstelle**, über die Programme mit dem Betriebssystem kommunizieren. Diese **Schnittstelle** oder **API** ist im Laufe der Jahre gewachsen und dementsprechend finden sich viele verwirrende Sachverhalte. Da inzwischen prinzipiell drei Versionen von Windows auf dem Markt koexistieren (Windows 3.1x, Windows 95 und Windows NT), ist das Chaos langsam aber sicher perfekt.

Ein Beispiel:

Das folgende Codefragment realisiert den Ausdruck einiger Daten aus einem **MFC** Programm. Der Code wurde aus einem älteren C Programm übernommen und ist dadurch gemischt **MFC** und "normale" C **Windows API**. **MFC** erlaubt dies ohne weiteres, auch wenn es wie hier dargestellt sicherlich nicht der beste Stil ist. Das alleine stellt also überhaupt kein Problem dar, allerdings wurden einige veraltete "Escape" Funktionen weiter verwendet. Diese wurden schon für Windows 3.1 ausdrücklich als veraltet gekennzeichnet, jedoch werden sie weiter unterstützt, da sonst einige Windows 3.0 Programme nicht mehr laufen würden.

Das Problem: Der Code funktioniert unter Windows 3.1, 3.11 und Windows NT 4.0. Unter Windows 95 funktioniert er nicht!

```
BOOL ucDrucken(HDC hDC)
{
    ...
    // Ausdruck starten
    if ((rc = StartDoc(hDC, &docinfo)) <= 0)
    {
        ReportError(rc, "StartDoc()");
    }
    // Neue Seite anlegen
    else if (StartPage(hDC) <= 0)
    {
        ReportError(SP_ERROR, "StartPage()");
    }
    else
    {
        ...
        // Datei drucken
        PrintHeader(hDC, ++u_side);
        for (u = 0; u < MAXCOUNT; u++)
        {
            // Textausgabe hier z.B. mit TextOut()...

            if (!((u + 1) % nLinesPerPage))
            {
                if ((rc = Escape(hDC, NEWFRAME, 0, NULL, NULL)) <= 0)
```

```
                {
                    ReportError(rc, "Escape()");
                    break;
                }
                PrintHeader(hDC, ++u_side);
            }
        }
        // Wieder neue Seite anfangen
        if ((rc = Escape(hDC, NEWFRAME, 0, NULL, NULL)) <= 0)
        {
            ReportError(rc, "Escape()");
        }
    }
    ...
    return TRUE;
}
```

Erst, wenn die ESCAPE Funktionen durch die moderneren Pendants ersetzt werden, funktioniert der Code immer:

```
            // Textausgabe hier z.B. mit TextOut()...

            if (!((u + 1) % nLinesPerPage))
            {
                // Seitenende anzeigen
                if ((rc = EndPage(cDlg.m_pd.hDC)) <= 0)
                {
                    ReportError(rc, "EndPage()");
                    break;
                }
                // Jetzt neue Seite beginnen
                if (StartPage(cDlg.m_pd.hDC) <= 0)
                {
                    ReportError(SP_ERROR, "StartPage()");
                    break;
                }
                PrintHeader(hDC, ++u_side);
            }
        }
        // Seitenende
        if ((rc = EndPage(cDlg.m_pd.hDC)) <= 0)
        {
            ReportError(rc, "EndPage()");
        }
    }
    ...
    return TRUE;
}
```

Nun kann man ganz zurecht argumentieren, daß man eben nicht die veralteten Funktionen hätte verwenden dürfen. Aber anderseits müßte auch ein Windows 98 noch diese unterstützen, sofern es von sich behaupten möchte, daß es zu Windows 3.1 abwärtskompatibel ist. Letzten Endes handelte es sich hier um eine 16 Bit Anwendung, die für Windows 3.1 und 95 konzipiert war.

Ein weiteres Beispiel:

Die beiden Funktionen **TextOut** und **DrawText** sind verwandt und unterscheiden sich nur geringfügig im Verhalten. **DrawText** erlaubt einige erweiterte Formatierungen (links/rechtsbündig, zentriert, automatischer Umbruch). Die Parameter unterscheiden sich jedoch sehr heftig voneinander:

```
BOOL TextOut(
HDC hdc,            // handle of device context
int nXStart,        // x-coordinate of starting position
int nYStart,        // y-coordinate of starting position
LPCTSTR lpString,   // address of string
int cbString        // number of characters in string
);

int DrawText(
HDC hDC,            // handle to device context
LPCTSTR lpString,   // pointer to string to draw
int nCount,         // string length, in characters
LPRECT lpRect,      // pointer to structure with formatting dimensions
UINT uFormat        // text-drawing flags
);
```

Die eine Funktion nimmt außer dem obligatorischen Gerätekontext, dem Textstring und der Länge des Strings die gewünschte Position als X und Y Wert. Bei der anderen muß man die Position als **RECT** Struktur übergeben. Das macht in diesem Fall Sinn, denn sonst könnte die Funktion keinen Umbruch oder Blocksatz machen. Aber die Konsequenz ist, daß man selbst als "alter Hase" immer wieder die Parameter der Funktionen in der API nachschauen muß.

Übungen

Ziele des Kapitels:

- Erlernen der Handhabung der Oberfläche von **Visual C++ 5.0**
- Umgang mit C, CPP, RC und anderen Dateien unter **Visual C++ 5.0**
- Einführung in die Grundlagen der Windowsprogrammierung
- Sinn und Zweck von Rückruf- (Callback-) Funktionen
- Gerätekontexte und Handles

Sie brauchen die hier gezeigten Beispiele nicht alle abzutippen. Die Quellfiles werden ihnen auf der Seminar-CD bzw. in Ihrem Anwenderverzeichnis bereitgestellt. Es sind jedoch noch keine Arbeitsbereiche und Projekte angelegt. Sie müssen also zunächst einen neuen Arbeitsbereich anlegen und dann innerhalb dieses Arbeitsbereiches die Übungen als neue Projekte.

Das erste Beispiel werden wir als Übung jedoch vollständig durchführen. Dies gibt uns die Gelegenheit, die Oberfläche von **Visual C++** genauer kennenzulernen.

Aufgabe 9: Anlegen eines neuen Arbeitsbereiches "SDK Schulung"

Wenn Sie nicht mehr wissen, wie das geht: Schauen Sie einfach unter Aufgabe 1 nach - dort wurde der Vorgang bereits ausführlich besprochen.

Übung 1 "HELLOWIN"

Aufgabe 10: Anlegen eines neuen Projektes "HELLOWIN" im Arbeitsbereiche "SDK Schulung" und Hinzufügen der notwendigen Quelldateien

Wenn Sie nicht mehr wissen, wie das geht: Schauen Sie einfach unter Aufgabe 2 nach - dort wurde der Vorgang bereits ausführlich besprochen.

Vorgehensweise:

Das Projekt ist vom Typ "Win32 Applikation", da es sich um eine einfach 32 Bit Windows Anwendung ohne MFC Unterstützung handelt.

Neue Quelldateien können Sie dem Projekt hinzufügen, indem Sie im **FileView** mit der rechten Maustaste auf das gewünschte Projekt klicken und im Pop-Up Menü "Dateien dem Projekt hinzufügen" anwählen. Im "Datei öffnen" Dialog können die benötigten Quelldateien ausgewählt werden.

Das Programm, das "Hallo Welt!" sagt ist schätzungsweise so alt wie die Computertechnik selbst. Aber die Zeit der Dreizeiler ist mit Windows leider vorbei. Ein Windowsprogramm, und sei es noch so einfach, besteht grundsätzlich aus mindestens zwei Teilen:

1. Der Hauptteil (main() oder WinMain())
2. Der Rückruffunktion

- Die Rückruffunktion ist das Herz des Windowsprogrammes, sie ist der Pulsschlag, der das Programm am Leben erhält. Das Betriebssystem ruft die Rückruffunktion jeden Programmes zu allen möglichen Gelegenheiten auf - wenn der Anwender mit der Maus in ein Fenster klickt, wenn das Fenster neu gezeichnet werden muß etc. Das Programm muß mit geeigneten Aktionen darauf reagieren.

```
int PASCAL WinMain (HANDLE hInstance, HANDLE hPrevInstance,
```

```
                    LPSTR lpszCmdParam, int nCmdShow)
{
     static char szAppName[] = "HelloWin" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASS    wndclass ;

     if (!hPrevInstance)
     {
          wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
          wndclass.lpfnWndProc   = WndProc ;
          wndclass.cbClsExtra    = 0 ;
          wndclass.cbWndExtra    = 0 ;
          wndclass.hInstance     = hInstance ;
          wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
          wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
          wndclass.hbrBackground = GetStockObject (WHITE_BRUSH) ;
          wndclass.lpszMenuName  = NULL ;
          wndclass.lpszClassName = szAppName ;

          RegisterClass (&wndclass) ;
     }
```

Bis zu dieser Stelle wurde das Programm fertig initialisiert. Die wichtigsten Dinge sind die Registrierung der Fensterklasse, wobei die Rückruffunktion angemeldet wird, der Programmname, der Icon des Programmes, der gewünschte Cursor sowie die Hintergrundfarbe des Fensters.

Ein Wort zu **hPrevInstance**: Jedes Programm bekommt vor dem Start vom Betriebssystem ein sogenanntes **Instanz-Handle** verpaßt, das sich in **hInstance** wiederfindet. Wurde das Programm bereits vorher schon mal gestartet und läuft noch, so enthält **hPrevInstance** das Handle dieser vorherigen Instanz (sonst ist das Handle **NULL**).

Es gibt mehrere Möglichkeiten, dies zu verwenden:

- Ist bereits eine Instanz vorhanden, muß die Fensterklasse nicht nochmals registriert werden.
- Falls gewünscht, kann hier das mehrfache Starten eines Programmes unterbunden werden - mit Hilfe der Instanz kann man das vorige Programm in den Vordergrund holen und das aktuell neu gestartete wieder beenden. (Beispiel: Windord startet bei Doppelklick im Explorer auf ein DOC File. Wiederholt man dies mit anderen Word Dokumenten, wird die bereits gestartete Instanz von Winword wiederverwendet).

Ansonsten kann man den Parameter getrost ignorieren.

```
     hwnd = CreateWindow (szAppName,        // Fensterklassenname
          "Das erste Programm",             // Titelleiste
          WS_OVERLAPPEDWINDOW,              // Fensterstil
          CW_USEDEFAULT,                    // Fensterposition, x
          CW_USEDEFAULT,                    // Fensterposition, y
          CW_USEDEFAULT,                    // horizontale Größe
          CW_USEDEFAULT,                    // vertikale Größe
          NULL,                             // Handle Parent-Window
          NULL,                             // Handle Menü
          hInstance,                        // Handle Programmkopie
          NULL) ;                           // Spezialparameter
     ShowWindow (hwnd, nCmdShow) ;          // Anzeigen
     UpdateWindow (hwnd) ;                  // Aktualisieren
```

Jetzt ist das Fenster erzeugt worden und wird angezeigt. Der **nCmdShow** Parameter definiert, wie das Fenster angezeigt werden soll: Normal, minimiert oder maximiert. Es ist nicht

unbedingt notwendig, den Parameter auszuwerten, es gehört lediglich zum guten Ton.

```
      while (GetMessage (&msg, NULL, 0, 0))
      {
            TranslateMessage (&msg) ;
            DispatchMessage (&msg) ;
      }
      return msg.wParam ;
}
```

Jetzt läuft das Programm. Die **while** Schleife arbeitet die von Windows gesendeten Nachrichten ab. Im Fachjargon nennt sich die Technik, Nachrichten auf diese Weise zu bearbeiten "Nachrichtenpumpe". Würde man sich diese Zeilen sparen, würde das Programm hier wieder ohne viel Federlesens beendet und es wäre - wenn überhaupt - nur ein kurzes "Zappen" zu sehen.

```
long EXPORT FAR PASCAL WndProc (HWND hwnd, UINT message,
                        UINT wParam, LONG lParam)
{
      HDC         hdc ;
      PAINTSTRUCT ps ;
      RECT        rect ;
```

Die Parameter haben folgende Bedeutung:

Parameter	Funktion
HWND hwnd	Handle des aufrufenden Fensters
UINT message	Art der Nachricht
UINT wParam	WORD Parameter der Nachricht
LONG lParam	LONG Parameter der Nachricht

Eine Aufstellung der Nachrichten und Parameter finden Sie in der Windows API Dokumentation.

Folgende Konstruktion wertet die beiden wichtigsten Nachrichten aus: **WM_PAINT** und **WM_DESTROY**. Windows schickt immer dann eine **WM_PAINT** Nachricht, wenn das Programm seinen Anzeigebereich ganz oder teilweise neu zeichnen muß. Dies ist der natürlich der Fall, wenn das Programm gerade gestartet wurde, aber auch, wenn der Ausgabebereich zeitweilig teilweise oder ganz von einem anderen Fenster überdeckt wurde.

- An dieser Stelle sei übrigens gleich erwähnt, daß jedes Fenster seine Rückruffunktion besitzen muß, ein größeres Programm besteht also immer aus mehr als einer Rückruffunktion.

```
      switch (message)
      {
            case WM_PAINT :
                  hdc = BeginPaint (hwnd, &ps) ;
                  GetClientRect (hwnd, &rect) ;
                  DrawText (hdc, "Hello, Windows!", -1, &rect,
                        DT_SINGLELINE | DT_CENTER | DT_VCENTER) ;
                  EndPaint (hwnd, &ps) ;
                  return 0 ;
```

Der Vorgang des Zeichnens läuft immer auf dieselbe Weise ab:

- Wir teilen Windows mittels **BeginPaint** mit, daß wir jetzt gerne zeichnen möchten. Diese Funktion liefert uns auch gleich den sogenannten Gerätekontext, den wir zum Zeichnen

benötigen.

- Wir besorgen uns den Bereich, in dem wir zeichnen möchten oder müssen
- Wir führen die Zeichenoperationen aus.
- Wir teilen Windows mit, daß wir alles erledigt haben: **EndPaint**.
- Wir verlassen die Funktion über **return 0**.

Bemerkungen:

Aus reiner Faulheit verwenden wir hier **GetClientRect**. Diese Funktion liefert uns den gesamten Bereich des Fensters - eigentlich müßten wir schauen, welcher Teil des Fensters neu gezeichnet werden muß. Das ist eventuell nur eine kleine Ecke und geht viel natürlich viel schneller.

Es passiert jedoch nichts, wenn man außerhalb des zu aktualisierenden oder sogar sichtbaren Bereiches zeichnet. Windows fängt dies ab, allerdings kostet es natürlich Zeit.

Alle GDI Funktionen benötigen den Gerätekontext des Ausgabegerätes. Er identifiziert das Gerät: Bildschirm, Drucker oder auch eine Bitmap im Speicher.

- Der Aufruf von **EndPaint** ist sehr wichtig, damit Windows die für die vorhergehenden Operationen belegten Ressourcen wieder freigeben kann.

Es ist ebenfalls wichtig 0 als Rückgabewert der Funktion zu verwenden. Dies informiert Windows, daß wir bereits alles erledigt haben. Geben wir eine andere Zahl zurück, wird Windows noch einige Standardoperationen machen und die Ergebnisse sehen eventuell nicht sehr zufriedenstellend aus.

```
            case WM_DESTROY :
                  PostQuitMessage (0) ;
                  return 0 ;
            }
            return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

Der Rest der Funktion handhabt die **WM_DESTROY** Nachricht. Sie wird einem Programm geschickt, wenn es sich verkrümeln soll. Die typische Reaktion ist, sich selbst eine Nachricht zu schicken: **PostQuitMessage (0)** sagt dem Programm, daß es sich nun beenden kann. Alternativ kann man auch **PostMessage (hwnd, WM_QUIT, 0, 0);** verwenden.

- Äußerst wichtig ist es, alle nicht behandelten Nachrichten mittels **DefWindowProc** "nach oben" durchzureichen, damit Windows diese weiter bearbeiten kann. Ansonsten können sehr seltsame Effekte das Ergebnis sein.
- Vollständig behandelte Nachrichten können durch Rückgabe von 0 an den Aufrufer beendet werden. Dann muß **DefWindowProc** nicht aufgerufen werden.

Wie man sieht, ist allein das "**Hallo Welt!" Programm** schon ein fast abendfüllendes Projekt. Die recht umständliche Vorgehensweise mag zunächst ziemlich seltsam anmuten. Aber nur mit dieser Technik sind größere Windowsprogramme realisierbar und programmtechnisch zu handhaben.

Es gibt auch andere Ansätze z.B. bei OS/2 mittels einer reinen Nachrichtenschleife ohne Rückruffunktionen. Aber auch in diesem Fall werden den Programmen Nachrichten gesendet und diese müssen sie entsprechend bearbeiten. Dies funktioniert aber nur in einem Betriebssystem mit echtem "**preemptiven Multitasking**", also wenn die Prozessorleistung durch einen Dispatcher auf alle laufenden Prozesse gleichmäßig verteilt wird. Verlangt ein Prozess mehr Rechenleistung als ihm zusteht, kann ihm diese durchaus verweigert werden. Jeder Prozess erhält bestimmte Zeitscheiben, innerhalb denen er laufen kann. Danach wird er "suspendiert", also eingefroren, und ein anderer Prozess läuft für seine Zeitscheibe ab.

Das war zur Zeit von Windows 3.x mit den verfügbaren Prozessoren nicht ohne weiteres machbar. Daher verwendete Microsoft das sogenannte "**kooperative Multitasking**". Hier läuft eigentlich nur ein einziger Prozess, während ein Programm seine Nachrichtenschleife

abarbeitet, kann kein anderer Prozess laufen. Läuft es innerhalb dieses Vorganges in eine Endlosschleife, ist das komplette Windows außer Gefecht gesetzt und nur noch ein Reset hilft aus der Misere.

Unter Windows 95 und NT werden die "alten" 16 Bit Programme in einer 16 Bit Teiltask ausgeführt, die, sobald das Programm abstürzt, ohne Einfluß auf das Gesamtsystem stehen bleiben und beendet werden kann. Neue 32 Bit Programm laufen jedoch im echten preemptiven Multitasking.

Übung 2 "SYSMETS"

Damit sich ein Programm darüber informieren kann, auf was für einem System es genau läuft, bietet Windows eine ganze Reihe von Funktionen an. Die wichtigsten werden vom folgenden Programm ausgewertet und angezeigt.

Der WinMain Code unterscheidet sich nicht wesentlich vom "Hallo Welt!" Programm. In der Rückruffunktion jedoch steckt eine ganze Menge zusätzliche Funktionalität.

```
long EXPORT FAR PASCAL WndProc(HWND hwnd, UINT message,
                         UINT wParam, LONG lParam)
{
     static short cxChar, cxCaps, cyChar, cxClient, cyClient, nMaxWidth,
                  nVscrollPos, nVscrollMax, nHscrollPos, nHscrollMax ;
     char         szBuffer[50] ; // f r wsprintf und Oem_Ansi
     HDC          hdc ;
     short        i, x, y, nPaintBeg, nPaintEnd, nVscrollInc, nHscrollInc ;
     PAINTSTRUCT  ps ;
     TEXTMETRIC   tm ;
     switch (message)
     {
            case WM_CREATE :
                  hdc = GetDC (hwnd) ;
                  GetTextMetrics (hdc, &tm) ;
                  cxChar = (short)tm.tmAveCharWidth ;
                  cxCaps = (short)((tm.tmPitchAndFamily & 1 ? 3 : 2) *
                  cxChar / 2) ;
                  cyChar = (short)(tm.tmHeight + tm.tmExternalLeading) ;
                  ReleaseDC (hwnd, hdc) ;
                  nMaxWidth = 40 * cxChar + 18 * cxCaps ;
                  return 0 ;
            case WM_SIZE :
                 cyClient = HIWORD (lParam) ;
                 cxClient = LOWORD (lParam) ;
                 nVscrollMax = max (0, (int)(NUMLINES + 2 - cyClient /
                 cyChar)) ;
                 nVscrollPos = min (nVscrollPos, nVscrollMax) ;
                 SetScrollRange (hwnd, SB_VERT, 0, nVscrollMax, FALSE) ;
                 SetScrollPos (hwnd, SB_VERT, nVscrollPos, TRUE) ;
                 nHscrollMax = max (0, 2 + (nMaxWidth - cxClient) / cxChar)
                 ;
                 nHscrollPos = min (nHscrollPos, nHscrollMax) ;
                 SetScrollRange (hwnd, SB_HORZ, 0, nHscrollMax, FALSE) ;
                 SetScrollPos (hwnd, SB_HORZ, nHscrollPos, TRUE) ;
                 return 0 ;
           case WM_VSCROLL :
                 switch (wParam)
                 {
                      case SB_TOP :
                            nVscrollInc = -nVscrollPos ;
                            break ;
                      case SB_BOTTOM :
                            nVscrollInc = nVscrollMax - nVscrollPos ;
```

```
                    break ;
               case SB_LINEUP :
                    nVscrollInc = -1 ;
                    break ;
               case SB_LINEDOWN :
                    nVscrollInc = 1 ;
                    break ;

               case SB_PAGEUP :
                    nVscrollInc = min (-1, -cyClient / cyChar) ;
                    break ;
               case SB_PAGEDOWN :
                    nVscrollInc = max (1, cyClient / cyChar) ;
                    break ;

               case SB_THUMBTRACK :
                    nVscrollInc = LOWORD (lParam) - nVscrollPos ;
                    break ;

               default :
                    nVscrollInc = 0 ;
          }
          nVscrollInc = max (-nVscrollPos,
           min (nVscrollInc, nVscrollMax - nVscrollPos)) ;
          if (nVscrollInc)
          {
               nVscrollPos += nVscrollInc ;
               ScrollWindow (hwnd, 0, -cyChar * nVscrollInc, NULL,
               NULL) ;
               SetScrollPos (hwnd, SB_VERT, nVscrollPos, TRUE) ;
               UpdateWindow (hwnd) ;
          }
          return 0 ;
     case WM_HSCROLL :
          switch (wParam)
          {
               case SB_LINEUP :
                    nHscrollInc = -1 ;
                    break ;
               case SB_LINEDOWN :
                    nHscrollInc = 1 ;
                    break ;

               case SB_PAGEUP :
                    nHscrollInc = -8 ;
                    break ;

               case SB_PAGEDOWN :
                    nHscrollInc = 8 ;
                    break ;

               case SB_THUMBPOSITION :
                    nHscrollInc = LOWORD (lParam) - nHscrollPos ;
                    break ;
               default :
                    nHscrollInc = 0 ;
          }
          nHscrollInc = max (-nHscrollPos,
                     min (nHscrollInc, nHscrollMax - nHscrollPos)) ;
          if (nHscrollInc)
          {
               nHscrollPos += nHscrollInc ;
```

```
                ScrollWindow (hwnd, -cxChar * nHscrollInc, 0,
                NULL,NULL) ;
                SetScrollPos (hwnd, SB_HORZ, nHscrollPos, TRUE) ;
           }
           return 0 ;
      case WM_KEYDOWN :
           switch (wParam)
           {
                case VK_HOME :
                     SendMessage (hwnd, WM_VSCROLL, SB_TOP, 0L) ;
                     break ;

                case VK_END :
                     SendMessage (hwnd, WM_VSCROLL, SB_BOTTOM, 0L) ;
                     break ;

                case VK_PRIOR :
                     SendMessage (hwnd, WM_VSCROLL, SB_PAGEUP, 0L) ;
                     break ;

                case VK_NEXT :
                     SendMessage (hwnd, WM_VSCROLL, SB_PAGEDOWN, 0L) ;
                     break ;

                case VK_UP :
                     SendMessage (hwnd, WM_VSCROLL, SB_LINEUP, 0L) ;
                     break ;
                case VK_DOWN :
                     SendMessage (hwnd, WM_VSCROLL, SB_LINEDOWN, 0L) ;
                     break ;

                case VK_LEFT :
                     SendMessage (hwnd, WM_HSCROLL, SB_PAGEUP, 0L) ;
                     break ;
                case VK_RIGHT :
                     SendMessage (hwnd, WM_HSCROLL, SB_PAGEDOWN, 0L) ;
                     break ;
           }
           return 0 ;

      case WM_PAINT :
           hdc = BeginPaint (hwnd, &ps) ;
           nPaintBeg = max (0, nVscrollPos + ps.rcPaint.top / cyChar
           - 1) ;
           nPaintEnd = min (NUMLINES,
           nVscrollPos+ps.rcPaint.bottom/cyChar);
           for (i = nPaintBeg ; i < nPaintEnd ; i++)
           {
                x = cxChar * (1 - nHscrollPos) ;
                y = cyChar * (1 - nVscrollPos + i) ;
                TextOut (hdc, x, y, sysmetrics[i].szLabel,
                     lstrlen (sysmetrics[i].szLabel)) ;
                Oem_Ansi(sysmetrics[i].szDesc,szBuffer); // -> ANSI
                TextOut (hdc, x + 22 * cxCaps, y,

                     szBuffer, lstrlen(szBuffer)) ;
                SetTextAlign (hdc, TA_RIGHT | TA_TOP) ;
                TextOut (hdc, x + 22 * cxCaps + 40 * cxChar, y,
                     szBuffer, wsprintf (szBuffer, "%5d",
                GetSystemMetrics (sysmetrics[i].nIndex))) ;
                SetTextAlign (hdc, TA_LEFT | TA_TOP) ;
```

```
                }
                EndPaint (hwnd, &ps) ;
                return 0 ;
          case WM_DESTROY :
                PostQuitMessage (0) ;
                return 0 ;
          }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

Übung 3 "HEXCALC"

Dieses kleine Programm besitzt gegenüber den vorigen eine kleine Besonderheit. Es verwendet eine weitere Komponenten von Windowsprogrammen: Ressourcen. Diese werden in speziellen Files mit der Endung .RC abgelegt. Die darin verwendete Script-Sprache könnte durchaus auch von Hand erstellt werden.

In der Regel wird man aber einen der graphischen Editoren (Ressourcen-Editoren) verwenden, um sie zu erstellen. **Visual C++** stellt passende Editoren für Fenster, Dialoge und auch Bitmaps oder Icons bereit, die dem Entwickler das Leben sehr erleichtern.

Über den ResourceView können Sie sich den Dialog im Dialogeditor ansehen.

```
/*----------------------------
Ressource-Datei für HEXCALC
----------------------------*/

#include <windows.h>

HexCalc ICON hexcalc.ico
HexCalc DIALOG 32768, 0, 102, 122
STYLE WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX
CLASS "HexCalc"
CAPTION "Hexadezimalrechner"
{
        PUSHBUTTON "D",        68,  8,  24, 14, 14
        PUSHBUTTON "A",        65,  8,  40, 14, 14
        PUSHBUTTON "7",        55,  8,  56, 14, 14
        PUSHBUTTON "4",        52,  8,  72, 14, 14
        PUSHBUTTON "1",        49,  8,  88, 14, 14
        PUSHBUTTON "0",        48,  8, 104, 14, 14
        PUSHBUTTON "0",        27, 26,   4, 50, 14
        PUSHBUTTON "E",        69, 26,  24, 14, 14
        PUSHBUTTON "B",        66, 26,  40, 14, 14
        PUSHBUTTON "8",        56, 26,  56, 14, 14
        PUSHBUTTON "5",        53, 26,  72, 14, 14
        PUSHBUTTON "2",        50, 26,  88, 14, 14
        PUSHBUTTON "Korr.",    8, 26, 104, 32, 14
        PUSHBUTTON "C",        67, 44,  40, 14, 14
        PUSHBUTTON "F",        70, 44,  24, 14, 14
        PUSHBUTTON "9",        57, 44,  56, 14, 14
        PUSHBUTTON "6",        54, 44,  72, 14, 14
        PUSHBUTTON "3",        51, 44,  88, 14, 14
        PUSHBUTTON "+",        43, 62,  24, 14, 14
        PUSHBUTTON "-",        45, 62,  40, 14, 14
        PUSHBUTTON "*",        42, 62,  56, 14, 14
        PUSHBUTTON "/",        47, 62,  72, 14, 14
        PUSHBUTTON "%",        37, 62,  88, 14, 14
        PUSHBUTTON "=",        61, 62, 104, 32, 14
        PUSHBUTTON "&&",       38, 80,  24, 14, 14
        PUSHBUTTON "|",        124, 80,  40, 14, 14
        PUSHBUTTON "^",        94, 80,  56, 14, 14
```

```
        PUSHBUTTON "<",         60, 80,  72, 14, 14
        PUSHBUTTON ">",         62, 80,  88, 14, 14
}
```

Übung 4 "SPINCUBE"

"SPINCUBE" besitzt jetzt zum ersten Mal auch ein Menü. Auch Menüs sind in der RC Datei organisiert und **Visual C++** stellt einen eigenen Editor dafür zur Verfügung.

Neu an diesem Beispiel:

Demonstration der Zeichenfunktionen, Verwendung einer DLL.

Übung 5 "DYNDLG"

Demonstration der Zeichenfunktionen, Verwendung einer dynamisch nachgeladenen DLL.

Übung 6 "DEVCAPS"

Zeigt die verschiedenen Parameter, die man Windows abfragen kann.

Übung 7 "COLORS"

Ein Common Dialog - Beispiel für die verschiedenen vordefinierten Dialoge in Windows.

Programmierung mit MFC

Ziele des Kapitels:

- Erlernen der Konzepte von **MFC**
- Umgang mit den Werkzeugen
- Kennenlernen der wichtigsten Klassen (Anwendung, Fenster, Datenbankzugriffe, Kontrollelemente).
- Erstellung von Online-Hilfen.

Das erste MFC Programm

In diesem Abschnitt werden wir ein einfaches **MFC** Programm erstellen, ohne die Hilfen von **Visual C++ 5.0** zu verwenden. Dies soll die Analogien zu "normalen" SDK Programmen aufzeigen und vor allem verdeutlichen, wie stark **MFC** die Entwicklung von Windowsprogrammen vereinfacht.

Die Klassenbibliothek **MFC** 4.21 ist eine Sammlung vom über 200 C++ Klassen für verschiedene Anwendungsbereiche. Der Schwerpunkt liegt bei der Entwicklung von Windows 95/NT Anwendungen. **MFC** bietet dafür reichhaltige Unterstützung. Weitere Einsatzbereiche sind allgemeine Datenstrukturen, zu denen Listen, Arrays oder auch Strings gehören. Aber allen gemeinsam ist der objektorientierte Gedanke der Wiederverwendbarkeit von Quellcode in neuen Projekten.

Da die C++ Klassen sehr an die Windows API angelehnt sind, kann die **MFC** ohne größeren Aufwand in eine bestehende Windows-Anwendung eingebunden werden. Das Desgin von Windows Anwendungen wird durch den Einsatz der **MFC** im Vergleich zur "puren" API Programmierung deutlich vereinfacht. Der Entwickler muß nicht mehr die Fülle der API Funktionen und ihre Eigenarten (vgl. "Die Windows API und ihre Tücken") kennen.

Mit hoher Wahrscheinlichkeit findet er eine passende Klasse oder zumindest eine Klasse, aus der er durch Ableitung die gewünschte Lösung für sein aktuelles Problem gewinnen kann. Deshalb ist es sehr sinnvoll, sich genau mit dem Aufbau und Inhalt der **MFC** zu beschäftigen, um die Mehrfachentwicklung zu vermeiden und Entwicklungen effizient durch den Einsatz vorhandener Klassen voranzutreiben.

Anmerkung

Es ist eine Namenskonvention, die von der C++ Konvention abweicht:

- Alle Klassennamen in **MFC** 4.21 beginnen mit einem großen C gefolgt von einer Bezeichnung, die möglichst aussagekräftig das Wesen der Klasse beschreibt.
- Außerdem gilt die Vereinbarung, daß alle Elementfunktionen mit einem Großbuchstaben beginnen.
- Die Klassenvariablen werden mit einem Kleinbuchstaben beschrieben und beginnen mit m_.
- Alle Variablen werden nach der Ungarischen Notation mit Präfixen versehen (siehe "Die Ungarische Notation").

Application Architecture Class

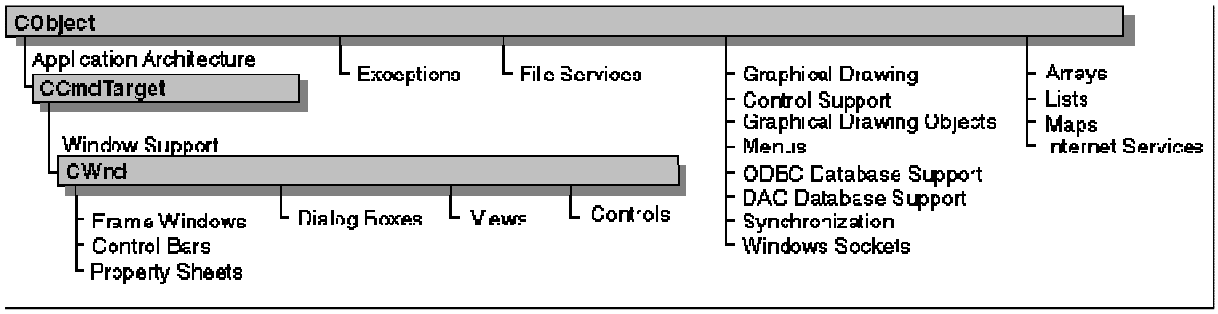


Abbildung 26: Kategorien der MFC Hierarchie

Die **Application Architecture Class** unterstützt Sie in der Strukturierung Ihrer Anwendung, indem Klassen für bestimmte Architekturen von Windows-Anwendungen vorgegeben sind. Sie können wählen zwischen einer Klasse, die Sie bei der Gestaltung von Dokumentenfenstern oder die Sie mit Datenstrukturen für Druckeraufträgen unterstützt.

CObject

Die Basisklasse, aus der alle weiteren Klassen mit **MFC** abgeleitet werden, ist die Klasse **CObject**. Sie dient als Wurzelklasse (Root-Class) nicht nur für Klassen innerhalb von **MFC**, sonder auch für die Klassen aus Ihrer Anwendung. Jede Klasse der **MFC** oder Ihrer Anwendung geht auf diese Klasse zurück. Die Klasse **CObject** bietet einige Basisdienste an, die Folgendes umfassen:

- **Unterstützung für Serialisierung**. Kurz: Aus Objekten muß man einen Datenstream machen und diesen abspeicherbar aufbereiten und umgekehrt. Also aus einem Datenstream ein Objekt generieren, das dann abgespeichert werden kann. Die Serialisierung ist ein Grundpfeiler für dauerhafte Objektspeicherung (Persistenz).
- **Runtime Klasseninformationen**. Unter diesem Basisdienst sind die Möglichkeiten zusammengefaßt, zur Laufzeit für ein Objekt seine Klasse zu bestimmen und allen Objekten die Informationen mitzugeben, daß sie speicher- oder ladbar sind.
- **Diagnoseangaben über Objekte und deren Laufzeitverhalten**. Dieser Basisdienst ist besonders wertvoll für das Debugging. Wenn sie für ein Objekt Speicherplatz oder Adresse untersuchen müssen, dann können Sie die entsprechenden Informationen aufbereiten lassen.
- **Kompatibilität mit sogenannten Collection Classes**, also Klassen, die ihrerseits Objekte aufnehmen können, um Datenstrukturen zu errichten.

Sie werden sicherlich selten direkt mit dieser Klasse im Rahmen der Vererbung arbeiten, aber indirekt habe Sie über Vererbung jederzeit Zugriff auf die Elemente der Klasse.

CWinApp

Diese Klasse dient als Basisklasse für alle Windows-Anwendungen. Jede Windows-Anwendung, die Sie entwickeln, nutzt die Basisklasse **CwinApp**. Wegen der Eindeutigkeit von Objekten, kann ihre Anwendung nur ein Objekt der Basisklasse **CWinApp** enthalten.

Diese Klasse stellt einige Elemente zur Verfügung, die für jede Anwendung lebensnotwendig sind. Das sind Elementfunktionen zum Setzen des Cursors oder zum Laden der Icons. Zusätzlich sind Zeigervariable vorhanden, die auf Objekte Ihrer Anwendung verweisen. Im später vorgestellten Beispiel werden wir diese Elemente näher kennenlernen.

Window Support

Die Klassenhierarchie der **Window Support Klassen** ist vollständig unter der Klasse **CObjekt** eingeordnet. In dieser Hierarchie sind alle Klassen zusammengefaßt, die mit der Gestaltung der Fenster in einer Windows-Anwendung zu tun haben. Es sind Klassen für Dialogboxen,

Scrollbars, Buttons etc. (Zur Erinnerung: Jedes Kontrollelement in Windows stellt ein Fenster mit eigenem Handle etc. dar).

CWnd

Sie ist die Basisklasse für alle Fenster Ihrer Anwendung. Aus dieser Klasse können Sie legal spezielle Fensterklassen ableiten. Diese Klasse stellt die elementaren Funktionen zur Verfügung, die Sie für einen ordnungsgemäßen Ablauf der Windows-Anwendung erwarten. **CWnd** ist die "Verwaltungsklasse" für alle Fenster Ihrer Anwendung. deshalb verfügt sie über Funktionen, die die gesamte Bandbreite von Fensterverwaltung und -handling abdecken.

CFrameWnd

Die Klasse **CFrameWnd** als Ableitung von **CWnd** bietet die volle Funktionalität eines Fensters für Single Document Interface (SDI) oder eines Pop-Up-Fensters. Es ist zweckmäßig, die Anwendung unter dieser Klasse einzurichten. Die Klasse **CFrameWnd** verwaltet auch das Hauptmenü, das jedem Fenster zugeordnet ist. **CFrameWnd** hat spezielle Funktionen, um das typische Verhalten von Fenstern zu bieten, wie Positionieren der Scrollbars, An- und Abschalten der unten angeordneten Statusleiste. Sie ist auch in der Klage, die Aktualisierung von Fensterelementen im Hintergrund durchzuführen, wenn keine Anwendungsfunktion aktiv ist.

Sonstige Klassen

Die sonstigen Klassen liefern allgemeine Dienstleistungen und machen sich damit unabhängig von den windowsorientierten Applikationen. Die enthaltenen Klassen könnten auch in einer reinen MS DOS Anwendung Einsatz finden.

Wie bei dem ersten beiden Abschnitten. ist auch hier eine Klasse **CObject** die Basisklasse. im wesentlichen besteht die Hierarchie hier aus mehreren thematisch abgeschlossene Bereichen, einigen Hierarchien und einigen Einzelklassen. Die Einzelklassen sind zwar nicht in die Vererbungshierarchie eingeschlossen, gehören thematisch jedoch dazu.

Die Themenbereiche sind:

- Dateizugriffe
- Ausnahmenbehandlung
- Collection-Klassen
- Verschiedene Klassen

Auf der folgenden Seite finden Sie eine Gesamtübersicht über die in **MFC 4.21** vorhandenen Klassen.

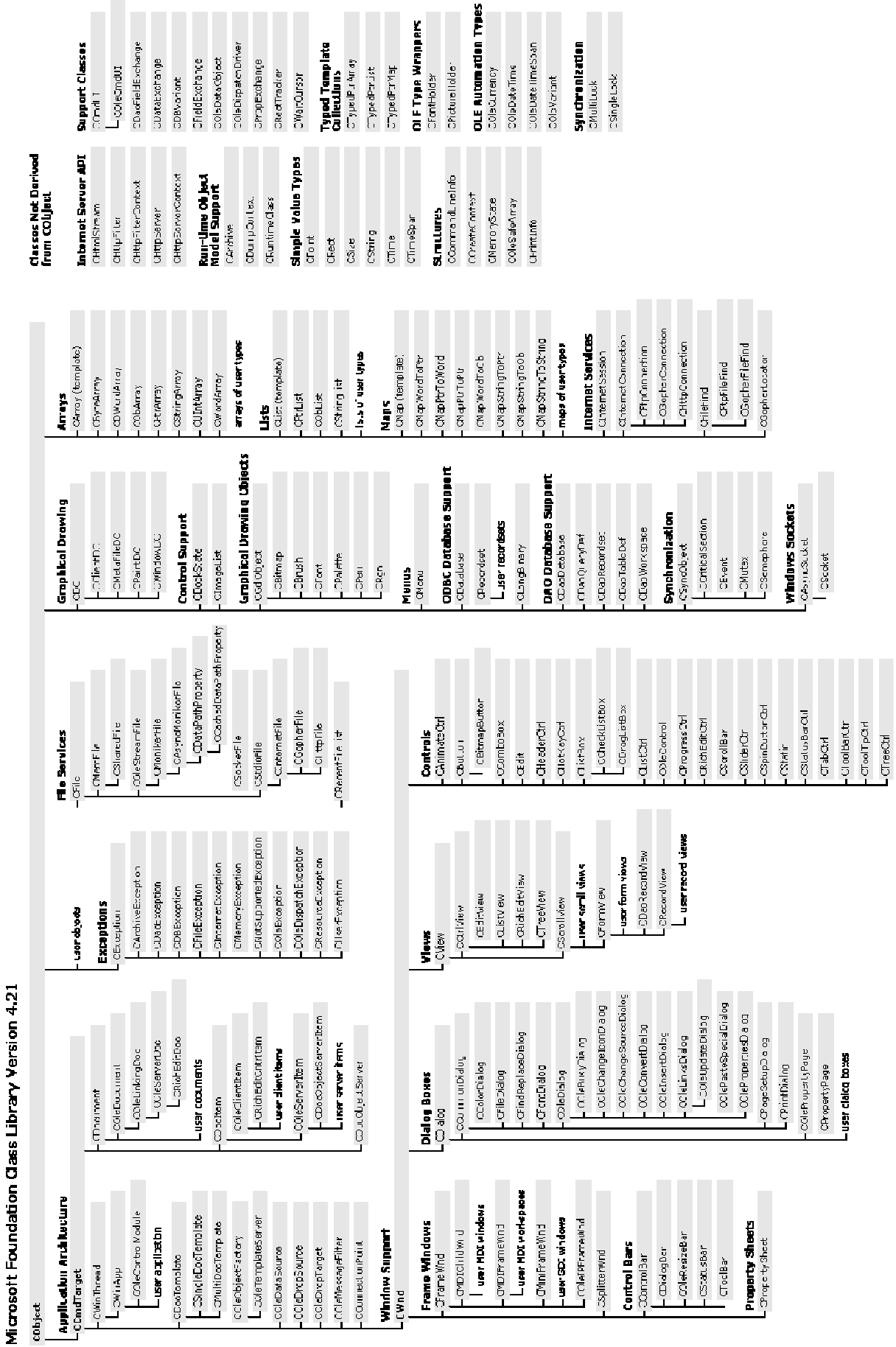


Abbildung 27: MFC 4.21 Übersicht

Eine einfache MFC Anwendung

Um ein minimales **MFC** Programm zu schreiben, brauchen Sie mindestens zwei Klassen für Ihre Anwendung. Das bedeutet, sie benötigen zwei Basisklassen aus der **MFC**, aus denen abzuleiten ist.

1. Schritt
Definition einer Fensterklasse, welche aus **CFrameWnd** abgeleitet ist.

Die erste Klasse der Anwendung ist die Klasse **Challo** und wird aus der **MFC**-Klasse **CFrameWnd** abgeleitet. Mit **CFrameWnd** werden alle Eigenschaften eines Fensters in die Anwendung eingebracht wie Pop-Up-Fähigkeit oder Verschiebbarkeit. Die wichtigsten Elementfunktionen werden durch Überladung oder durch Neuimplementierung an die jeweiligen Situation angepaßt. Diese Klasse hat jedoch noch nichts mit der Anwendung zu tun. Unabhängig wie die Anwendung später beschaffen sein wird, die Klasse **CFrameWnd** muß in das Programm eingebunden sein.

In diesem Beispiel ist nur der Konstruktor **CHallo** zu definieren. Auch diese ist minimal implementiert und benutzt die geerbte Funktion **Create** aus der Klasse **CFrameWnd**. Diese Funktion ist mit vier Parametern ausgestattet:

NULL:	Null-Zeiger für Klassennamen. Es wird kein Klassenname vergeben.
"Hello World!"	String, dieser wird als Titel über das zu erzeugende Fenster geschrieben.
WS_OVERLAPPEDWINDOW	Konstante, um überlappende Fenster zu ermöglichen.
rectDefault	Standardangabe von Koordinaten, an denen das Fenster auf dem Bildschirm zu positionieren ist.

2. Schritt:
Definition einer Applikationsklasse, welche aus **CWinApp** abgeleitet ist.

Die Applikationsklasse heißt **CHalloApp** und wird aus der **MFC**-Klasse **CWinApp** abgeleitet. Die Applikationsklasse erbt alle Variablen und Funktionen, die unverändert akzeptiert werden können. Eine Ausnahme gibt es: Die Elementfunktion **InitInstance()** muß für die Fensterverwaltung der Applikationsfenster ausgestattet werden. Diese Funktion ist virtuell deklariert mit dem Hintergedanken, daß von der Klasse **CHalloApp** weitere Applikationsklassen abgeleitet werden.

Die Funktion **InitInstance()** enthält drei Anweisungen:

- Initialisieren des Zeigers **m_pMainWnd**, der in der Klasse **CFrameWnd** deklariert ist.
- Starten des Fensteraufbaus und Fenster anzeigen.
- Rückgabewert ist TRUE, weil eine erfolgreiche Initialisierung in diesem Beispiel vorgegeben ist.

3. Schritt:
Generieren des Applikationsobjektes.

Von der Applikationsklasse **CHalloApp** wird ein Objekt generiert. Der Konstruktor wird aufgerufen und über die Ausführung der Funktion **InitInstance** wird die Anwendung aktiv. Die Funktion **InitInstance** wird durch interne Initialisierungsabläufe indirekt aufgerufen.

Aufgabe 11: Erstellen Sie eine MFC Anwendung "Hallo Welt!"
*Erstellen Sie anhand obiger Vorgaben eine einfache **MFC** Anwendung, die "Hallo Welt" ausgibt.*

Vorgehensweise:
Erstellen Sie zunächst einen neuen Arbeitsbereich "**MFC**" und darin ein Projekt "**Hallo Welt**" von Typ Win32 Anwendung. Erstellen Sie in diesem Projekt eine Quelldatei "HALLO.CPP" und übernehmen Sie den folgenden Code:

```
#include <afxwin.h>

class CHallo : public CFrameWnd
{
public:
          CHallo()
          {
                  Create(NULL,    "Hello    World!",    WS_OVERLAPPEDWINDOW,
                  rectDefault);
          }
};

class CHalloApp : public CWinApp
{
public:
          virtual BOOL InitInstance();
};

BOOL CHalloApp::InitInstance()
{
          m_pMainWnd = new CHallo();
          m_pMainWnd->ShowWindow(m_nCmdShow);
          return TRUE;
}

ChalloApp HalloApp;
```

Wenn Sie das Beispiel compilieren, werden Sie feststellen, daß sich der Code zwar umsetzen aber nicht linken läßt. Der Grund dafür ist, daß die notwendigen Runtime-Bibliotheken für MFC fehlen:

```
--------------------Konfiguration: MFC - Win32 Debug--------------------
Kompilierung läuft...
hello.cpp
Linker-Vorgang läuft...
nafxcwd.lib(thrdcore.obj)  :  error  LNK2001:  Nichtaufgeloestes  externes
Symbol __endthreadex
nafxcwd.lib(thrdcore.obj)  :  error  LNK2001:  Nichtaufgeloestes  externes
Symbol __beginthreadex
Debug/MFC.exe : fatal error LNK1120: 2 unaufgeloeste externe Verweise
Fehler beim Ausführen von link.exe.

MFC.exe - 3 Fehler, 0 Warnung(en)
```

Mittels "Projekt/Einstellungen" bzw. ALT-F7 können wir dieses Problem lösen. Ändern Sie unter den Projekteinstellungen auf dem "Allgemeines" Tabellenreiter die Option "Microsoft Foundation Classes" von "MFC wird nicht verwendet" in eine der beiden anderen Optionen. "Als gemeinsam genutzte DLL" ist dabei vorzuziehen, da so Platz gespart wird - die gemeinsam genutzten Routinen befinden sich dann in einer DLL und werden von anderen MFC Programmen mitverwendet.

Ein Wörterbuch auf SQL Basis

In diesem Kapitel werden Sie ein vollständige Anwendung in **Visual C++** erstellen. Es handelt sich dabei um ein Wörterbuch auf SQL Basis. Sie werden die wichtigsten Komponenten von **MFC** kennenlernen: Fensterklassen, Controls, Datenbankzugriffe.

Aufgabe 12: Legen Sie einen neuen Arbeitsbereich "Dictionary" an und darin ein Projekt "Dictionary"

Erstellen Sie einen neuen Arbeitsbereich "Dictionary" in Ihrem Verzeichnis. Legen Sie ein neues Projekt "Dictionary" innerhalb des Arbeitsbereiches an.

Vorgehensweise

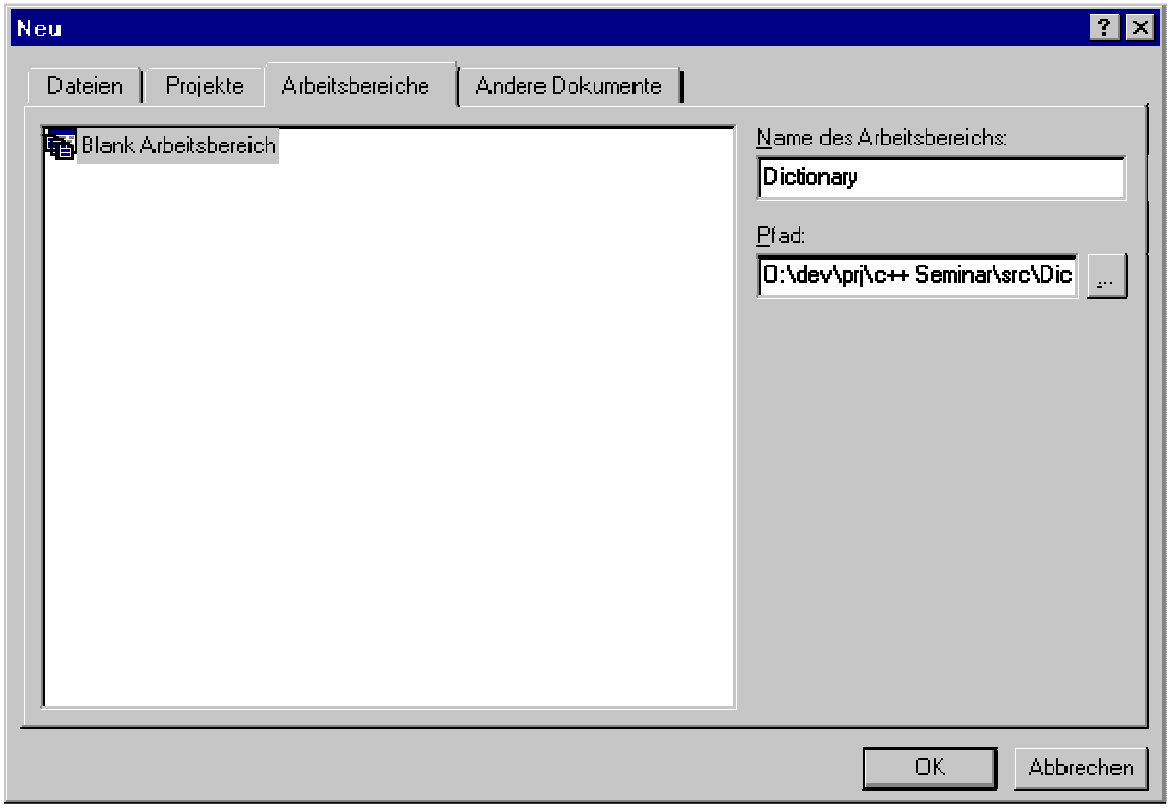


Abbildung 28: Neuanlage Arbeitsbereich "Dictionary"

Klicken Sie auf "Datei/Neu" oder drücken Sie **STRG-N**. Wählen Sie auf dem Fenster den Karteireiter "Arbeitsbereiche". Geben Sie als Arbeitsbereichsnamen "Dictionary" an. Klicken Sie dann "OK".

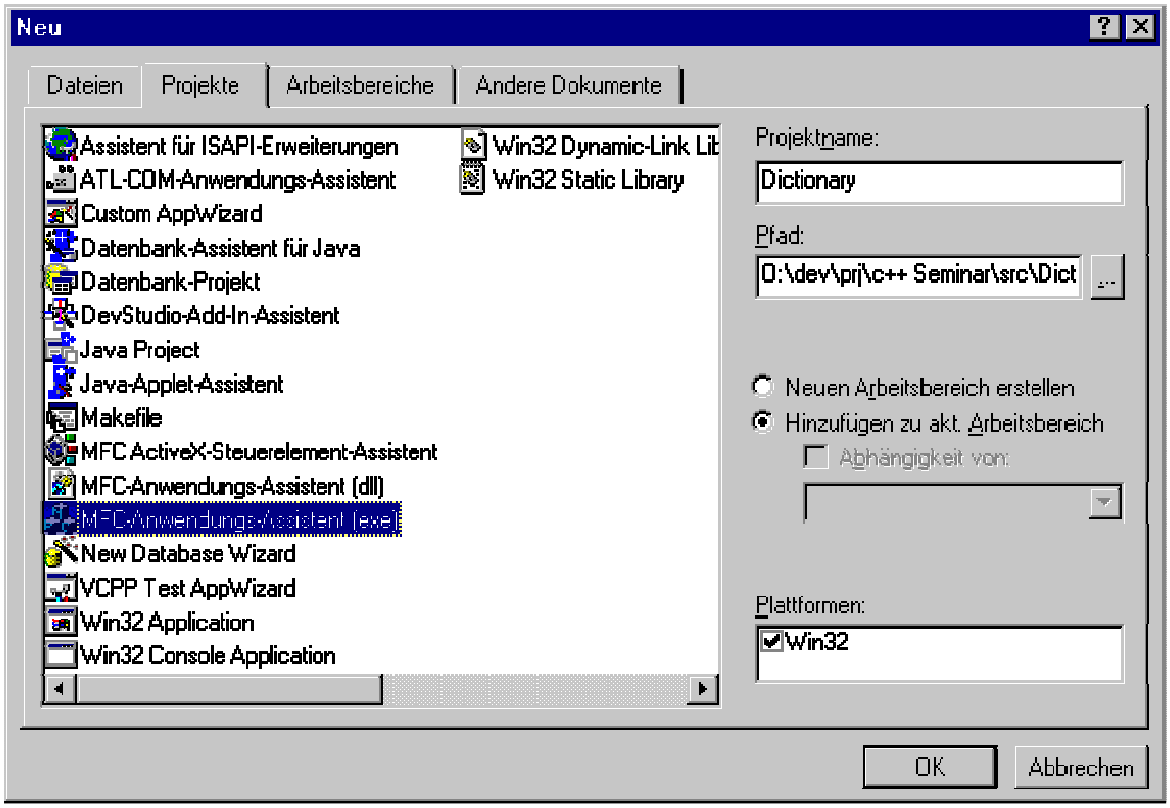


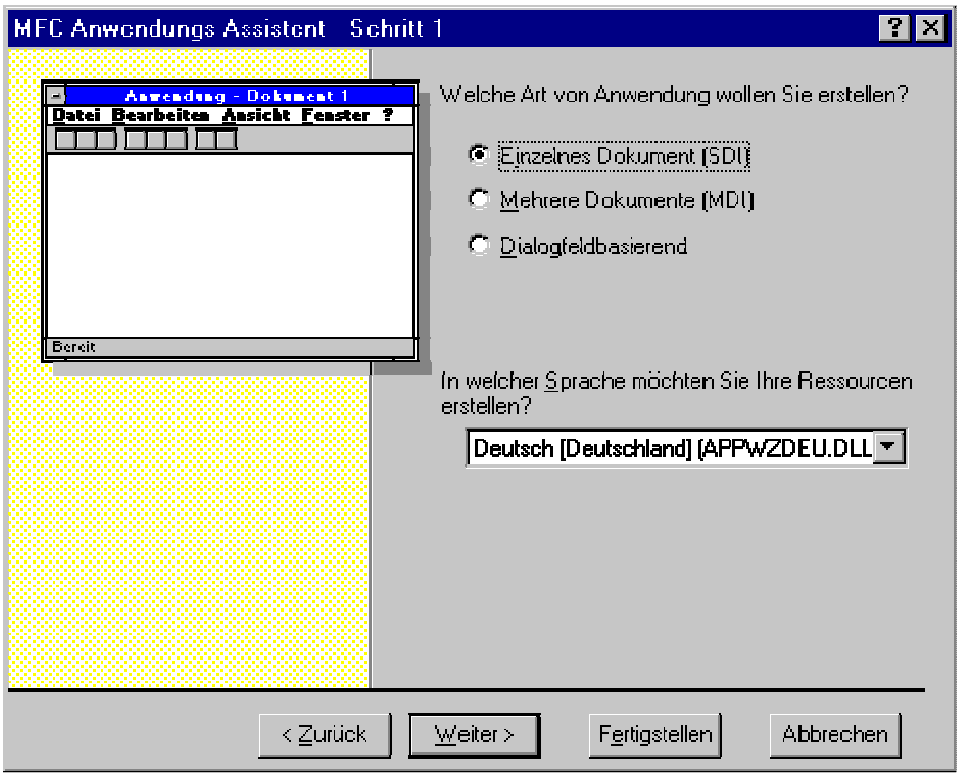
Abbildung 29: Anlegen einer MFC Anwendung

Klicken Sie auf "Datei/Neu" oder drücken Sie **STRG-N**. Wählen Sie auf dem Fenster den Karteireiter "Projekte". Wählen Sie links den **MFC** Anwendungsassistenten aus und geben Sie oben als Projektname "Dictionary" an. Klicken Sie dann "OK".

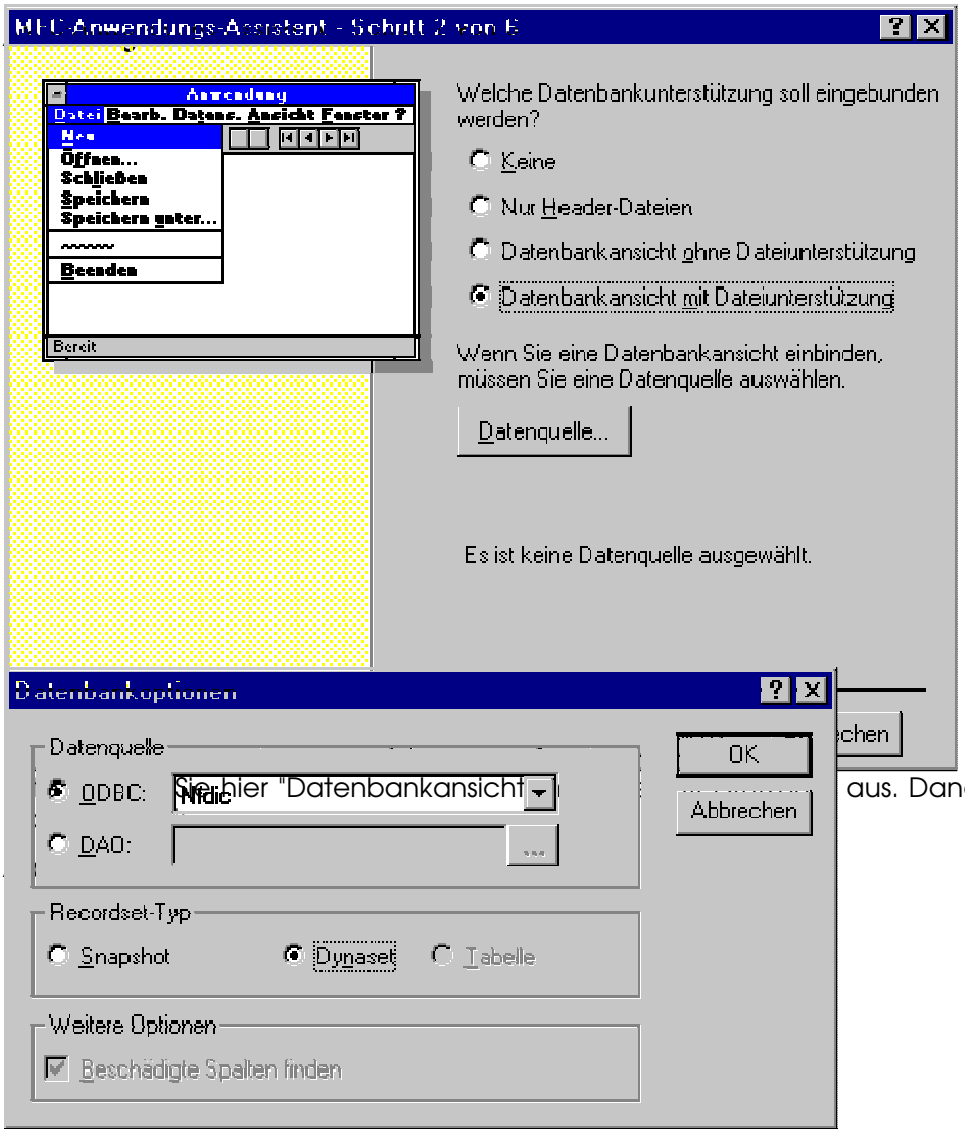
Achtung: Verwenden Sie im Dateinamen (und am besten auch Arbeitsbereichnamen) keine Umlaute und Sonderzeichen! Visual C++ findet sonst Teile des Projektes nicht.

Visual C++ startet nun den **MFC** Assistenten:

Abbildung 30: Der MFC-Assistent 1. Schritt



Auf der ersten Seite ändern wir die Art der Anwendung in "Einzelnes Dokument (SDI).



aus. Danach klicken Sie auf

Hier wählen Sie bitte aus der Liste der Datenquellen in der Box "NFDIC" aus. Der Recordset Typ sollte auf "Dynaset" geändert werden. Wenn Sie dann auf "OK" klicken, benötigt der Assistent noch zusätzlich das Paßwort für den Server-Zugriff. (Bei der Einrichtung des Treibers kann ein Standardlogin definiert werden. In diesem Fall wird diese Abfrage nicht angezeigt, sofern die Angaben stimmen).

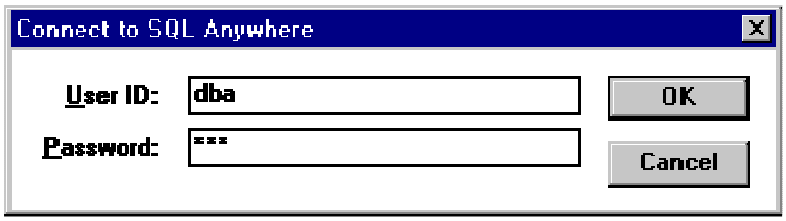


Abbildung 33: Verbindung mit SQL Server

Die User ID ist "**dba**", das Passwort lautet "**sql**" (dies ist Standard für die verwendete Datenbank Sybase SQL Anywhere 5.0). Damit kann der Assistent eine Verbindung zur Datenbank aufbauen und die darin gespeicherten Tabellen abrufen:

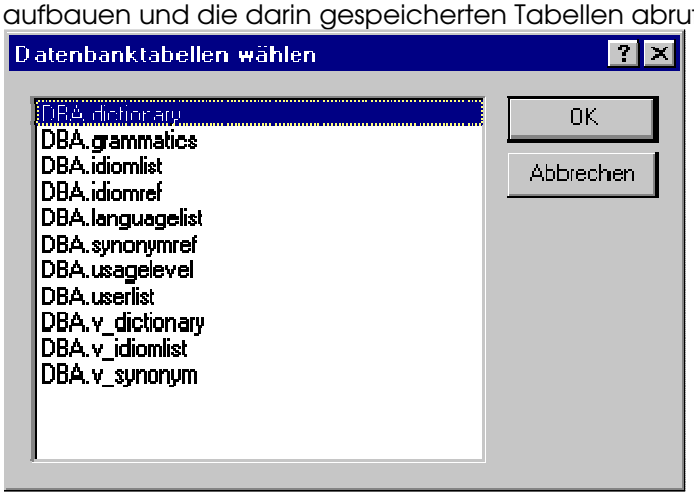


Abbildung 34: Angabe der gewünschten Tabelle

Die Tabelle "**DBA.dictionary**" enthält die gewünschten Daten. Markieren Sie diese also und klicken Sie auf "OK".

Jetzt erscheint wieder der gewohnte Assistent. Im Schritt 3 müssen wieder keine Änderungen vorgenommen werden, ebensowenig im 5. und 6. Schritt. Nur im Schritt 4 muß die Online-Hilfe aktiviert werden:

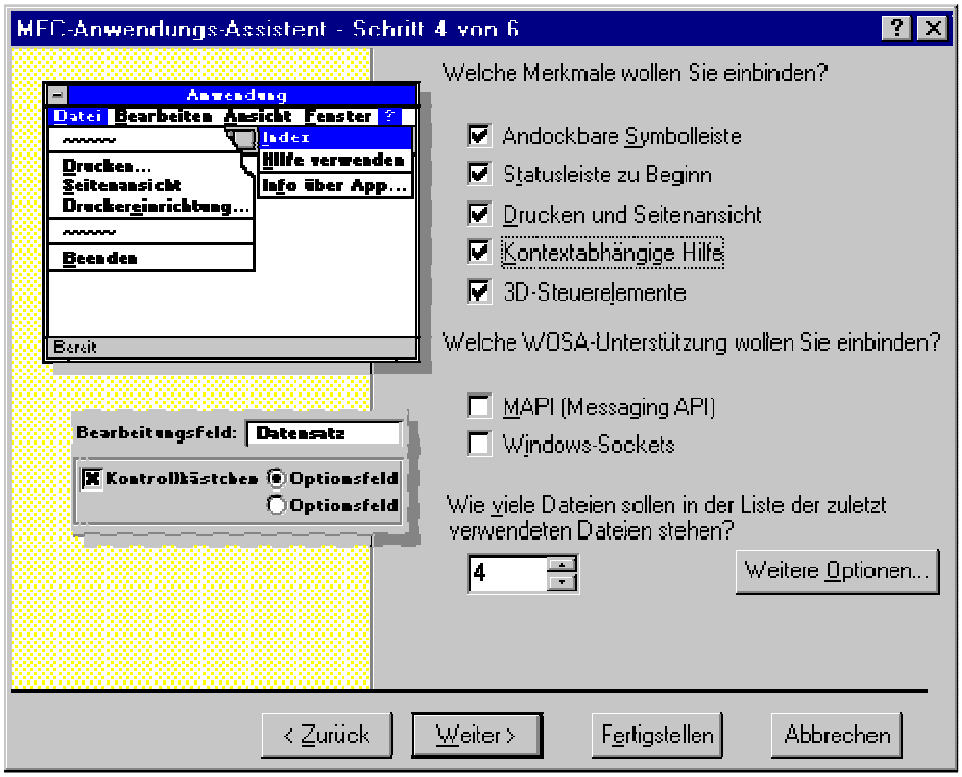


Abbildung 35: Der MFC-Assistent 3. Schritt

Damit ist der Assistent bereit, unsere Anwendung zu generieren. Er zeigt nur die gemachten Einstellungen als Zusammenfassung dar. Diese sollte bei Ihnen so ähnlich wie folgt aussehen:

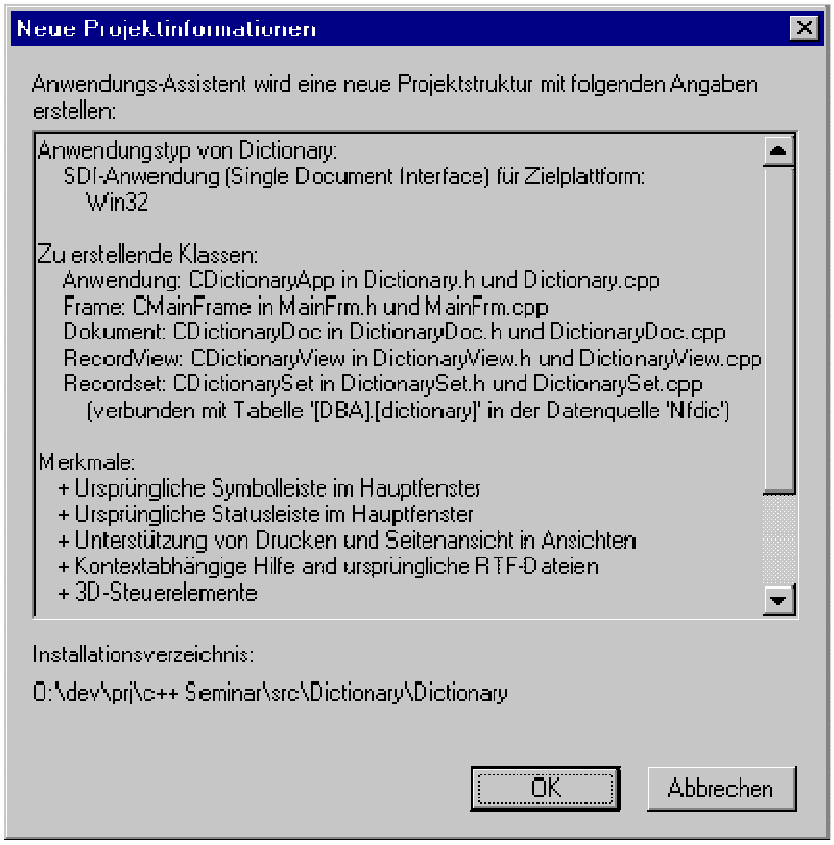


Abbildung 36: Projektinformationen

Die Bestandteile der generierten Minimalanwendung

Der **MFC** Assistent hat uns diesmal ein SDI Programm erstellt, also ein Programm, das immer nur ein einziges Dokument darstellen kann (Single Document Interface). Dies ist für die beabsichtigte Anwendung - das Wörterbuch - der geschickteste Ansatz.

Das generierte Minimalprogramm besteht aus folgenden Klassen:

CDictionaryApp
Dies ist das von **CWinApp** Hauptobjekt abgeleitete Hauptobjekt des Programmes.

CMainFrame
Das Rahmenfenster, auf dem die eigentlichen Dialoge anzeigt werden, abgeleitet von **CFrameWnd**.

CDictionaryView
Das eigentliche Fenster, auf dem die Kontrollelemente plaziert werden, abgeleitet von **CRecordView**. Es enthält auch das Datenbankobjekt als Member. Innerhalb dieses Fensters spielt sich das eigentliche Leben des Programmes ab.

CDictionaryDoc
Dieses von **CDocument** abgeleitete Objekt ist ebenfalls Member von **CDictionaryView**. **CDocument** ermöglicht die notwendige Funktionalität für benutzerdefinierte Dokumentenklassen. Ein Dokument repräsentiert eine Dateneinheit, die der Anwender typischerweise mit der File Öffnen Kommando aufmacht und mit dem Datei Speichern Kommando sichert. **CDocument** unterstützt Standardoperationen wie das Erzeugen eines Dokumentes, Laden und Speichern.

CDictionarySet
Hierbei handelt es sich um ein Datenbankobjekt, das von **CRecordSet** abgeleitet ist. Es erlaubt den Zugriff auf die Datenbank.

Datenbankanbindung automatisieren

Beim Start fragt die Anwendung augenblicklich jedesmal wieder Benutzername und Paßwort für die Datenbank ab. Das ist natürlich sehr unbequem. Daher codieren wir das Login zunächst fest in die **CDictionarySet** ein. Das Datenbankobjekt ist innerhalb von **CDictionaryDoc** definiert und wird dort initialisiert.

Um auf die Datenbank zugreifen zu können, benötigt **CDictionarySet** eine ODBC Verbindung zur Datenbank. Die dazu verwendete Klasse heißt **CDatabase** und ist ein Member-Objekt von **CDictionarySet.** Die Verbindung kann bei der Instanzierung von **CDictionarySet** übergeben werden. Ansonsten wird sie beim Öffnen der Datenbank mit **Open()** automatisch erzeugt. In diesem Fall werden aber die benötigten Parameter vom Anwender abgefragt, was wir vermeiden wollen.

Anmerkung: Der von uns verwendete ODBC Treiber (Sybase SQL Anywhere 5.x oder Sybase Apaptive Server 6.0) erlaubt es, Login und Paßwort beim Einrichten der ODBC Datenquelle anzugeben. In diesem Fall erscheint die Abfrage natürlich nicht, jedoch erkaufen wir dies mit zwei Nachteilen:

1. Wir verraten Login und Paßwort. Jeder Anwender kann es mit dem ODBC Setup nachschauen.
2. Wird die Datenquelle einmal gelöscht, müssen viel mehr Parameter wiederhergestellt werden.
3. Wenn irgendwann einmal auf eine andere Datenbank umgestellt werden soll, unterstützt deren Treiber diese Option eventuell nicht.

Aufgabe 13: Legen Sie eine Methode Connect() in der Klasse CDictionarySet an

*Legen Sie eine neue Methode Connect() in der Klasse **CDictionarySet** an und bauen Sie darin die Verbindung zur Datenbank auf.*

Vorgehensweise
Klicken Sie mit der rechten Maustaste im **ClassView** auf **CDictionarySet**. Wählen Sie dort "Member-Funktion hinzufügen".

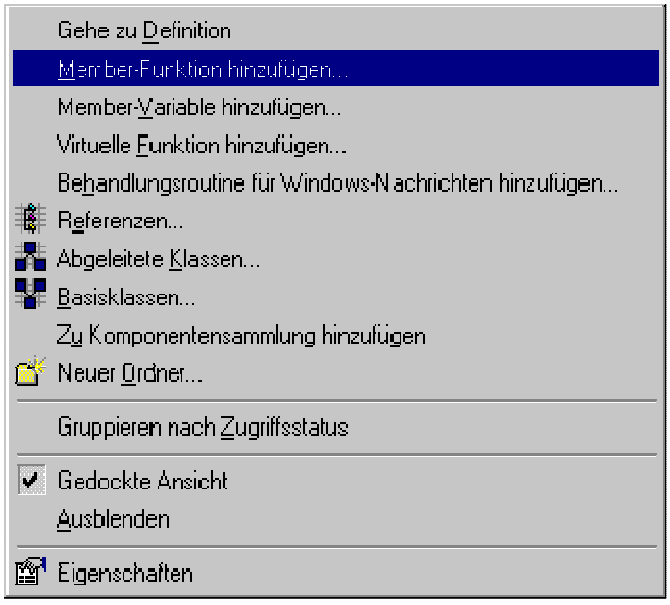


Abbildung 37: Pop-Up Menü des ClassView

Nun kann die Funktion angegeben werden.

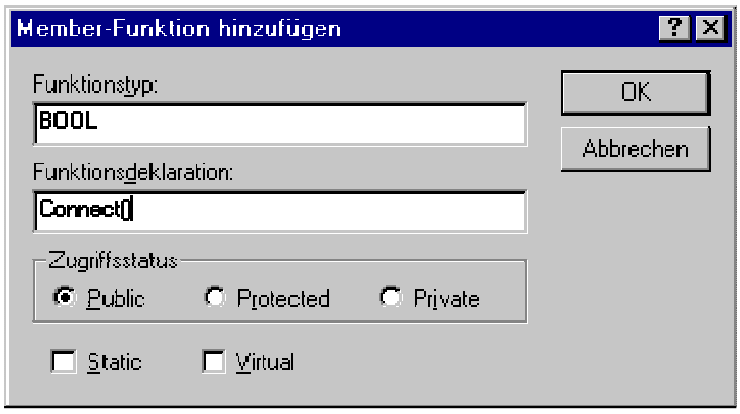


Abbildung 38: Member-Funktion "Connect()" hinzufügen

Das Ergebnis ist eine leere Funktion, der wir nun die notwendige Funktionalität verpassen:

```
BOOL CdictionarySet::Connect()
{
	if (m_pDatabase == NULL)
	{
		// Nur, wenn Objekt noch nicht existiert
		m_pDatabase = new CDatabase;
		// Wichtig: Wir müssen später auch wieder aufräumen
		m_bManualDatabase = TRUE;
```

```
        }

        if (!m_pDatabase->IsOpen())
        {
                // Verbindung aufbauen
                return m_pDatabase->OpenEx(
                        "DSN=NFDIC;UID=dba;PWD=sql",
                        CDatabase::noOdbcDialog);
        }
        else
        {
                return TRUE;
        }
}
```

Die Funktion setzt außerdem eine Variable **m_bManualDatabase**. Diese muß natürlich erst definiert werden.

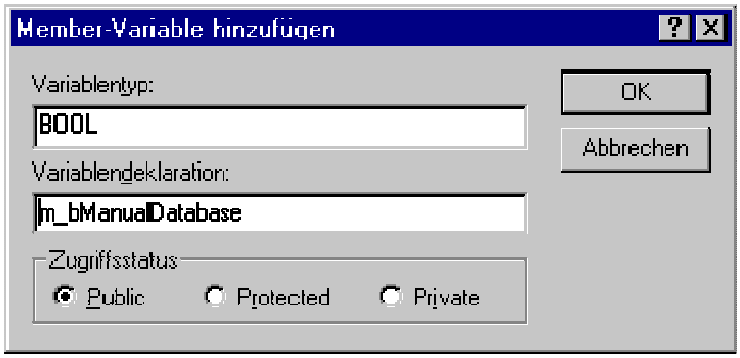


Abbildung 39: Member-Variable "m_bManualDatabase" hinzufügen

Einen typischen C/C++ Fehler sollten wir jetzt nicht machen, nämlich zu vergessen, das mit **new** allokierte Objekt nicht zu vernichten, wenn die Anwendung beendet wird. Dazu legen wir in **CDictionarySet** einen Destruktor an und programmieren ihn folgendermaßen:

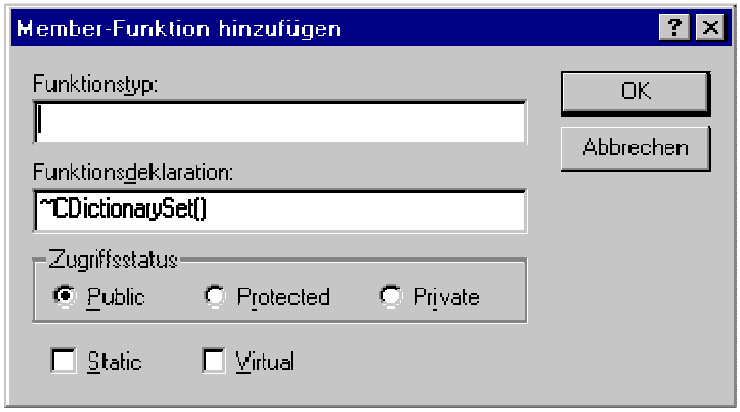


Abbildung 40: Member-Funktion hinzufügen: Destruktor für CDictionarySet

```
CdictionarySet::~CDictionarySet()
{
        if (m_pDatabase != NULL && m_bManualDatabase)
        {
                // Wichtig: Objekt müssen wir auch wieder vernichten
                delete m_pDatabase;
                m_pDatabase = NULL;
                m_bManualDatabase = FALSE;
        }
```

```
}
```

Ebenfalls nicht vergessen sollte man, daß Member-Variablen mit der Erzeugung eines Objekts nicht initialisiert werden. Der obige Code wird mit fast hundertprozentiger Sicherheit nicht funktionieren, da der Inhalt von **m_bManualDatabase** höchtswahrscheinlich nicht **FALSE** ist, wenn wir dies nicht innerhalb des Konstruktors sicherstellen.

```
CdictionarySet::CDictionarySet(CDatabase* pdb)
            : CRecordset(pdb)
{
    // ... wie bisher
    m_bManualDatabase = FALSE;
}
```

Nun fehlt letztlich eine kleine Änderung in **CDictionaryView**, dort muß Connect() noch aufgerufen werden. Öffnen Sie **ClassView**, erweitern Sie **CDictionaryView** und doppelklicken Sie auf **OnInitialUpdate()**. **Visual C++** zeigt ihnen jetzt die Methode an, die beim Aktivieren des Dokumentviews aufgerufen wird. Dort wird der Recordset übernommen. Jetzt kann dort die neue **Connect()** Funktion aufgerufen werden:

```
void CDictionaryView::OnInitialUpdate()
{
      m_pSet = &GetDocument()->m_dictionarySet;
      m_pSet->Connect();

      CRecordView::OnInitialUpdate();
}
```

Damit ist alles notwendige getan. Beim Start des Wörterbuches wird die Datenbank automatisch geladen, ohne daß der Anwender weitere Aktionen durchführen muß. Das Wörterbuch sieht jetzt folgendermaßen aus:

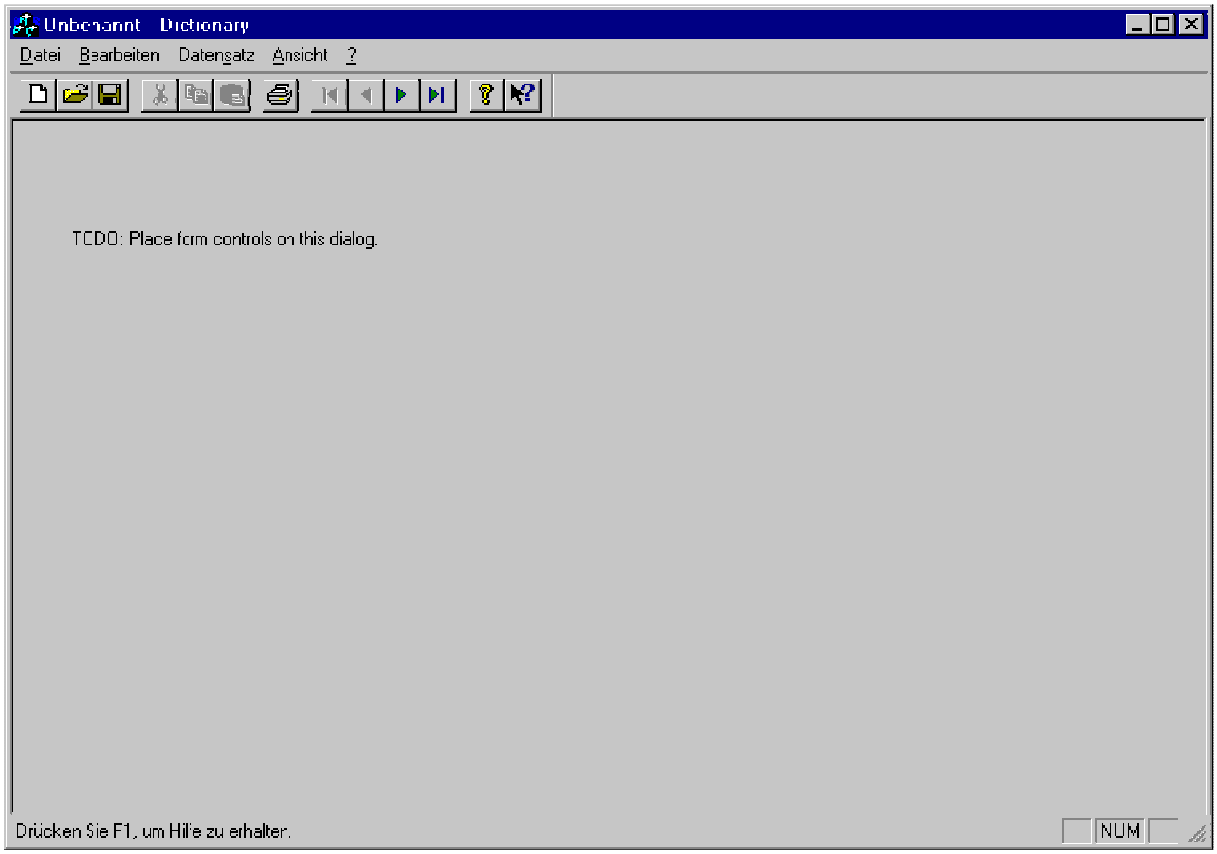


Abbildung 41: Das Hauptfenster der generierten Minimalanwendung

Spielen Sie ruhig ein wenig mit dem Programm. Sie werden feststellen, daß eine ganze Menge Funktionalität bereits vorhanden ist. Die Online-Hilfen sind soweit realisiert, daß bei den meisten Menüpunkten und Buttons die korrekten Beschreibungen angezeigt werden oder zumindest ein Standardtext eingeblendet wird, der leicht anzupassen ist.

Datensatzbewegung, Drucken, Hilfen, Seitenansicht etc. sind schon vorbereitet. Die jeweiligen Methoden müssen nur noch erweitert werden. Dies hat uns vorneherein einen Riesenberg an Arbeit erspart.

Anzeigen der Daten in gebundenen Kontrollelementen (Bound Controls)

Das Formular ist zwar vorhanden, aber ohne Kontrollelemente ziemlich langweilig. Plazieren Sie also zunächst eine Gruppenbox, vier Edit-Elemente, ein Button, ein Kombinationsfeld und eine Checkbox wie auf der Abbildung auf dem Formular.

Aufgabe 14: Definieren Sie Steuerelemente auf dem Form, ordnen Sie sie an und benennen Sie sie sinnvoll

Vorgehensweise:

Laden Sie zunächst den leeren Dialog **IDC_DICTIONARY_FORM** im **ResourceView** unter Dialoge. Mit der Toolpalette können sie die Controls auswählen und auf der Oberfläche des Formulars absetzen.

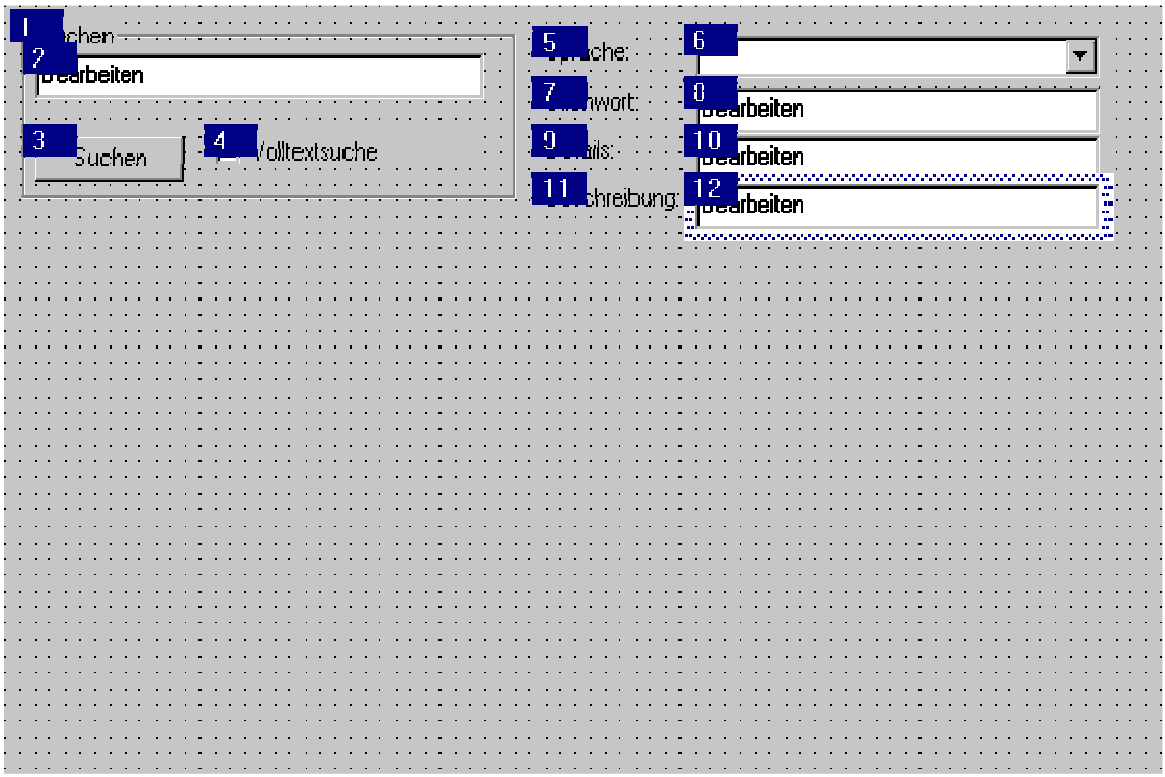


Abbildung 42: Der Dialogeditor

Im Ergebnis wird sicherlich die Reihenfolge der Kontrollelemente mit der erwünschten nicht übereinstimmen. Der Editor bietet dafür eine Funktion, die über Layout/Tabulator-Reihenfolge bzw. **STRG-D** ausgelöst werden kann. Der Editor zeigt die Reihenfolge wie auf der Abbildung an. Durch Klicken auf die Kontrollelemente kann sie geändert werden. Ein Klick außerhalb eines Kontrollelementes bricht den Vorgang ab.

Bei der Reihenfolge sollte darauf geachtet werden, daß die Textfelder jeweils direkt vor den zugehörigen Eingabefeldern angeordnet werden. Nur dann ist es möglich, mit ALT Tastenkombinationen in das richtige Feld zu springen. Dazu muß der gewünschte Buchstabe (Shortcut) mit einem Kaufmannsund "&" markiert werden. Soll das "&" angezeigt werden, gibt man es doppelt an: "**&Volltext && freie Suche**" würde als "**Volltext & freie Suche**" erscheinen, mit ALT-V wäre das Feld zu aktivieren.

Hinweis: Windows stört sich dabei nicht daran, wenn Shortcuts doppelt vergeben werden. Es wird dann allerdings immer nur der erste in der Reihenfolge angesprungen, alle weiteren sind unnerreichbar.

Der Editor vergibt Standardnamen für die Controls, die allerdings meist nicht sehr aussagekräftig sind. Daher sollten sie nun von Hand sinnvoll zugewiesen werden. Außerdem müssen einige Verhaltensweisen (Eigenschaften) geändert werden:

- Kombinationsfeld ändern: Dropdown-Listenfeld (Reiter "Formate"). Danach kann aus der angezeigten Liste genau ein Wert ausgewählt werden, freie Eingabe ist nicht möglich.
- Suchen-Edit: Großbuchstaben (Reiter "Formate"). Sieht hier besser aus, Groß- und Kleinschreibung wird sowieso ignoriert.
- Suchen-Button: Standardschaltfläche (Reiter "Formate"). Durch RETURN wird der Suchvorgang ausgelöst. Es kann immer nur ein solches Button pro Dialog geben, da sonst nicht unterschieden werden kann, welches beim Drücken von RETURN gemeint ist.

Nr.	Feld	Inhalt	Beschriftung	Name
1	Gruppenfeld (Group Box)		Suchen	IDC_STATIC_SEARCH
2	Eingabefeld (Single Line Edit)	Suchwert		IDC_EDIT_SEARCH
3	Schaltfläche (Button)		Suchen	IDC_BUTTON_SEARCH
4	Kontrollkästchen (Checkbox)	Freie Suche	Volltextsuche	IDC_CHECK_FULLTEXT
5	Textfeld		Sprache	IDC_STATIC_LANGUAGE
6	Kombinationsfeld (Combo Box)	Sprache		IDC_COMBO_LANGUAGE
7	Textfeld		Stichwort:	IDC_STATIC_HEADWORD
8	Eingabefeld	Übersetzung		IDC_EDIT_HEADWORD
9	Textfeld		Details	IDC_STATIC_DETAILS
10	Eingabefeld	Details		IDC_EDIT_DETAILS
11	Textfeld		Beschreibung	IDC_STATIC_DESCRIPTION
12	Eingabefeld	Beschreibung		IDC_EDIT_DESCRIPTION

Die Eigenschaften eines Feldes bekommen Sie, indem Sie rechts auf das Feld klicken und danach "Eigenschaften" aus dem Pop-Up-Menü anwählen.

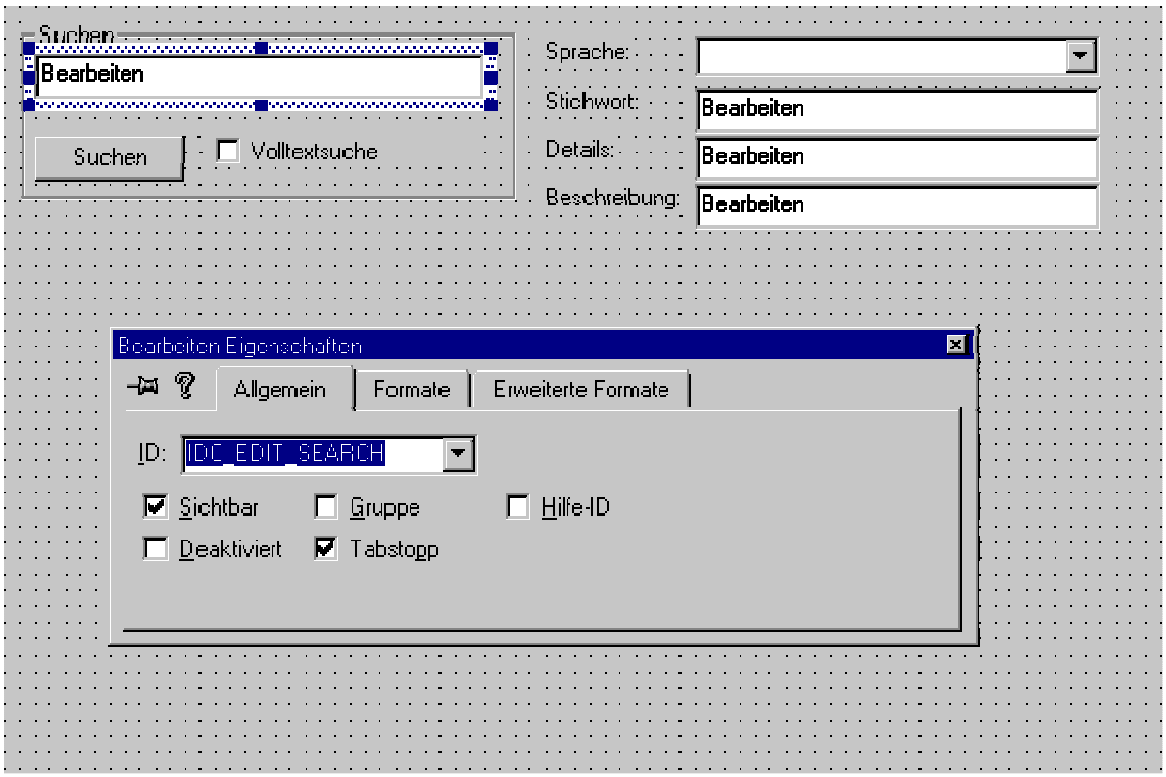


Abbildung 43: Ändern der Eigenschaften eines Kontrollelementes

Jetzt zeigt sich unser Programm bereits mit einem ganz ansehnlichen "Gesicht", aber leider steht nirgendwo etwas drin zu lesen. Dazu müssen die Kontrollelemente an die Datenbank gebunden werden. Dies nehmen wir im nächsten Abschnitt in Angriff.

Verbinden der Controls mit der Datenbank

Der Klassen-Assistent erweitert den Dialogdatenaustausch (DDX, Dialog Data Exchange), indem er es ermöglicht, Steuerelemente auf einem Formular oder einem Dialogfeld indirekt einem "Fremdobjekt" zuzuordnen, also speziell einem **CRecordset**- oder **CDaoRecordset-Objekt**. In diesem Abschnitt wird erklärt:

Ziele dieses Abschnitts:

- was Fremdobjekte sind
- wie man Fremdobjekte benutzt

Fremdobjekte

Zum Verständnis von Fremdobjekten sollte DDX für ein Dialogfeld oder eine Formularansicht sowie dessen Erweiterung betrachtet werden. An dieser Stelle wird eine Formularansicht auf der Basis der Klasse **CFormView** vorgestellt.

DDX ordnet die Steuerelemente in einer Formularansicht Datenelementen in der von **CFormView** abgeleiteten Klasse zu, die der Dialogvorlagen-Ressource zugeordnet sind. Einzelne Steuerelemente in einer Formularansicht entsprechen jeweils genau einem Datenelement der Formularansichtsklasse. Zwei Einheiten werden verbunden: die Formularansicht auf dem Bildschirm und die Formularansichtsklasse. (Dialogfelder funktionieren ebenso.)

Bei DDX für Fremdobjekte werden nicht zwei Einheiten, sondern drei verbunden. Die folgende Abbildung zeigt diese Verbindung mit einer Datensatzansicht auf dem Bildschirm auf der einen Seite, einer **CRecordView-Klasse** in der Mitte und einem dritten Objekt, dem Fremdobjekt, auf der anderen Seite: einem **CRecordset-Objekt**. (Dies könnten ein **CDaoRecordView**- oder ein **CDaoRecordset-Objekt** sein.)

DDX für Fremdobjekte

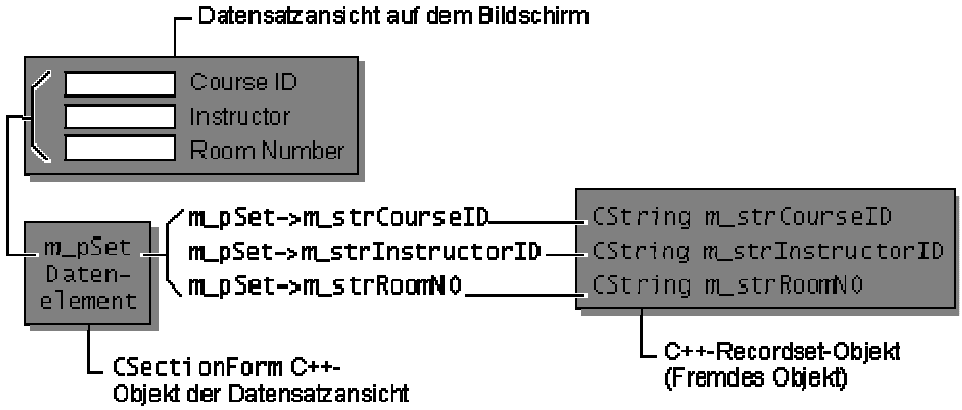


Abbildung 44: DDX für Fremdobjekte

Die Datensatzansicht-Klasse in der vorangehenden Abbildung enthält eine Member-Variable, einen Zeiger, dessen Typ der Klasse des fremden Objekts entspricht. Die DDX-Zuordnung verläuft von drei Datensatzansicht-Steuerelementen zu Elementen des Fremdobjekts durch den Zeiger, wie durch die folgende DoDataExchange-Funktion gezeigt:

```
void CSectionForm::DoDataExchange(CdataExchange* pDX)
{
```

```
    CRecordView::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CSectionForm)
    DDX_FieldText(pDX, IDC_COURSE, m_pSet->m_strCourseID, m_pSet);
    DDX_FieldText(pDX, IDC_INSTRUCTOR, m_pSet->m_strInstructor, m_pSet);
    DDX_FieldText(pDX, IDC_ROOMNO, m_pSet->m_strRoomNo, m_pSet);
    //}}AFX_DATA_MAP
}
```

Beachten Sie den indirekten Verweis auf Datensatzgruppen-Felder durch den Zeiger m_pSet auf ein Datensatzgruppen-Objekt CSections:

```
m_pSet->m_strCourseID
```

Beachten Sie außerdem, daß **m_pSet** als das vierte Argument in jedem DDX_FieldText-Aufruf wiederholt wird.

Der Klassen-Assistent und Fremdobjekte

Sie legen die "fremde" Klasse und das Fremdobjekt auf der Registerkarte Klassen-Info des Klassen-Assistenten fest. Das Feld Fremd-Klasse gibt die Klasse des Fremdobjekts an (**CRecordset** für ODBC oder **CDaoRecordset** für DAO). Das Feld Fremdvariable gibt die Zeigervariable an, die auf ein Objekt dieser Klasse verweist.

Hinweis: Sie können nur genau ein fremdes Objekt pro Klasse anlegen. Davon unberührt allerdings beliebig viele "von Hand", die aber für **VC++** dann keinerlei tiefere Bedeutung haben.

Wenn Sie eine Klasse auswählen, die auf einer Dialogvorlagen-Ressource auf der Registerkarte Member-Variablen des Klassen-Assistenten basiert, sucht der Klassen-Assistent die entsprechende Zeigervariable in der Klasse und setzt ihren Typ in das Feld Fremd-Klasse. Auch der Variablenname wird in dieses Feld übertragen. Sie können diese Werte ändern.

Wenn Sie mit dem Anwendungs-Assistent eine **CRecordView**- oder **CDaoRecordView-Klasse** anlegen und dann den Klassen-Assistent verwenden, um eine DDX-Verbindung anzugeben, bemerkt der Klassen-Assistent das Vorhandensein eines Zeigers zu einem von **CRecordset** oder **CDaoRecordset** abgeleiteten Objekt in der Ansichtsklasse automatisch. Der Name der Zeigervariablen wird in das Feld Fremdvariable übertragen.

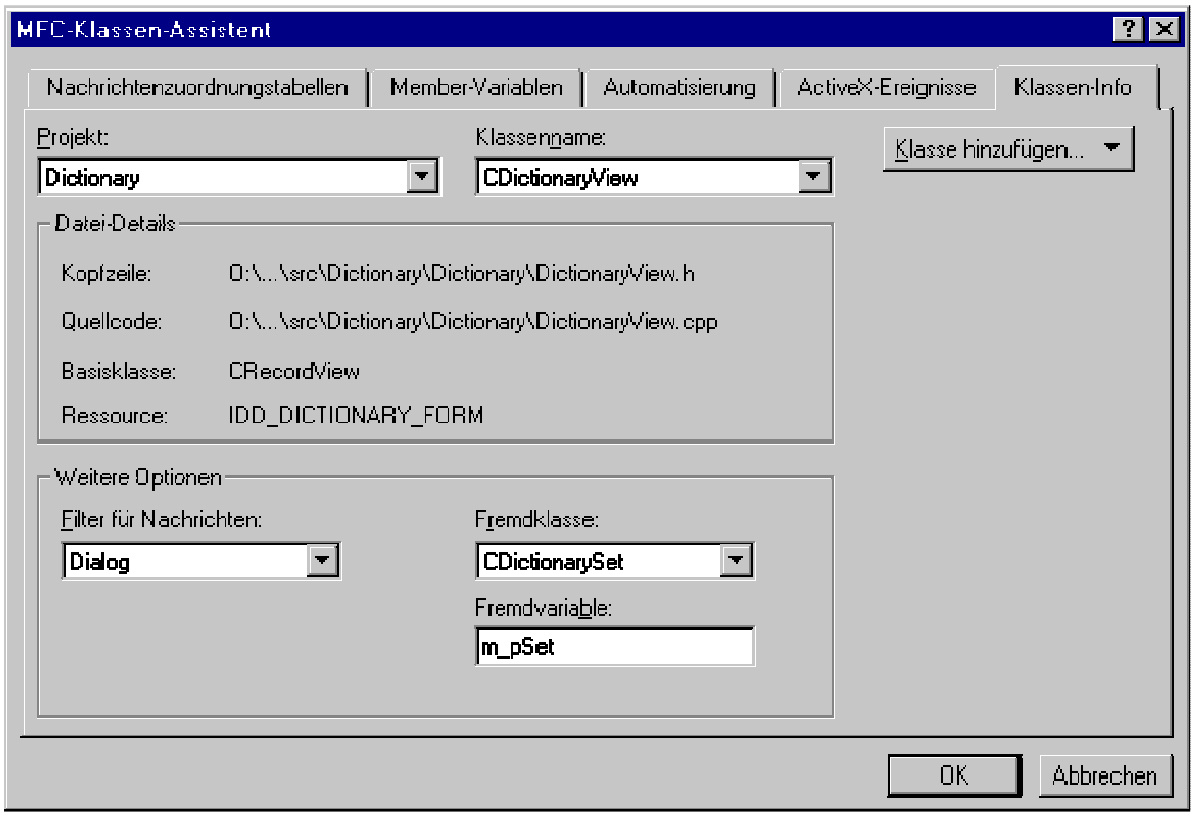


Abbildung 45: Klassen-Assistent: Zuordnung von Formularsteuerelementen zu Datensatzgruppen-Feldern

Ein Datenbankformular auf der Basis der Klassen **CRecordView** (für **MFC ODBC-Klassen**) oder **CDaoRecordView** (für **MFC DAO-Klassen**) verwendet den Dialogdatenaustausch (DDX, Dialog Data Exchange), um Daten zwischen den **Steuerelementen** des Formulars und den **Felddatenelementen** des mit dem Formular verbundenen **Datensatzgruppen-Objekts** auszutauschen. In diesem Abschnitt wird erklärt, wie Sie den Klassen-Assistent verwenden, um die DDX-Verbindung zwischen den Steuerelementen und der Datensatzgruppe herzustellen.

Diese Verbindung unterscheidet sich von dem normalen Einsatz von DDX, bei dem die Steuerelemente in einem Dialogfeld direkt mit den Datenelementen der zugeordneten Dialogklasse verbunden werden. Der Dialogdatenaustausch für ein Datensatzansicht-Objekt erfolgt indirekt. Die Verbindung erfolgt ausgehend von **Formularsteuerelementen** über das **Datensatzansicht-Objekt** hin zu den **Felddatenelementen** des zugeordneten **Datensatzgruppen-Objekts**. Weitere Informationen finden Sie in den Artikeln ***Datensatzansichten*** und ***Klassen-Assistent: Fremdobjekte*** in der Online-Hilfe.

In der folgenden Tabelle wird der Zuordnungsvorgang zusammengefaßt.

Zuordnung von Datensatzansicht-Steuerelementen zu einer Datensatzgruppe

Ziel	**Vorgehensweise**
Angeben des Formular-Steuerelements	Wählen Sie die Datensatzansichtsklasse auf der Registerkarte Member-Variablen in Klassen-Assistent sowie eine der Steuerelement-IDs des Formulars aus.
Angeben des Datensatzgruppen-Datenelements, zu dem eine Verbindung hergestellt werden soll	Verwenden Sie das Dialogfeld Variable hinzufügen, um ein Datensatzgruppen-Datenelement indirekt auszuwählen.

Bei der folgenden Anleitung wird angenommen, daß Sie das Erstellen eines

Datenbankformulars bereits durchgeführt haben (siehe **Aufgabe 14**). Insbesondere benötigen Sie eine von **CRecordset** oder **CDaoRecordset** abgeleitete Klasse und einige **Felddatenelemente**. Außerdem sollten Sie eine Dialogvorlagen-Ressource sowie Klassen angelegt haben, die von **CRecordView** oder **CDaoRecordView** abgeleitet sind, und die **Datensatzansichts-Klasse** der **Datensatzgruppen-Klasse** zugeordnet haben. Ihre **Datensatzansichts-Klasse** ist also mit der Dialogvorlagen-Ressource verbunden, zu der Sie Steuerelemente mit dem Dialog-Editor von **Visual C++** hinzugefügt haben.

All dies ist bereits vom **MFC-Wizard** für uns erledigt worden, die Felder haben wir ebenfalls angelegt. Damit Start frei für die Verbindung mit der Datenbank!

- So weisen Sie die **Fomularsteuerelemente** der **Datensatzgruppe** zu

1. Wählen Sie die Registerkarte **Member-Variablen** des **Klassen-Assistenten** aus.
2. Wählen Sie im Feld **Klassenname** den Namen Ihrer **Datensatzansichts-Klasse** aus.
3. Wählen Sie im Feld **Steuerelement-IDs** eine **Steuerelement-ID** aus.
4. Klicken Sie auf die Schaltfläche **Variable hinzufügen**, um eine Variable zu bezeichnen, die dem **Steuerelement** zugeordnet wird.
5. Wählen Sie im Dialogfeld **Member-Variable hinzufügen** einen Variablennamen aus, indem Sie ein **Datensatzgruppen-Datenelement** auf dem Dropdown-Listenfeld den Namen der **Member-Variable** auswählen.

   Wichtig: Verwenden Sie das **Dropdown-Listenfeld**, um einen Datenelementnamen aus der zugeordneten Datensatzgruppe auszuwählen (ein Fremdobjekt). Die angezeigten Namen haben die Form **m_pSet->m_recordsetVarName**. Sie können die Namen auch selbst eingeben, die Auswahl aus der Dropdown-Liste ist jedoch schneller und genauer.

   Alle Variablen der Datensatzgruppen-Klasse werden in dem Feld angezeigt, nicht nur die Variablen des gegenwärtig ausgewählten Datentyps.
6. Klicken Sie auf OK, um das Dialogfeld Member-Variable hinzufügen zu **schließen**.
7. Wiederholen Sie die Schritte 3 bis 6 für jedes Steuerelement in der Datensatzansicht, das Sie einem Datensatzgruppen-Felddatenelement zuordnen möchten.
8. Klicken Sie auf OK, um den Klassen-Assistent zu schließen.

Indem Sie einen Namen von der Dropdown-Liste auswählen, geben Sie eine Verbindung an, die über das Datensatzansicht-Objekt zu einem Felddatenelement des zugeordneten Datensatzgruppen-Objekts verläuft.

Tips

1. Wenn Sie den Klassen-Assistenten ausführen, während ein Dialog-Editor von **Visual C++** geöffnet ist, können Sie den Vorgang folgendermaßen abkürzen: Wählen Sie ein Steuerelement auf dem Formular aus, drücken Sie die **STRG-TASTE**, und doppelklicken Sie mit der Maus. Hierdurch wird das Dialogfeld Member-Variable hinzufügen des Klassen-Assistenten geöffnet. Hier können Sie ein Datensatzgruppen-Felddatenelement mit dem Steuerelement verbinden. (Wenn Sie diese Abkürzung verwenden, bevor für die Dialogvorlage eine Klasse angelegt wurde, öffnet und zeigt der Klassen-Assistent das Dialogfeld "Klasse hinzufügen" an.)
2. Wenn Sie eine einfache Regel befolgen, während Sie Steuerelemente auf dem Datensatzansichts-Formular positionieren, kann der Klassen-Assistent schon im voraus das wahrscheinlichste Datensatzgruppen-Element im Dialogfeld auswählen. Die Regel lautet, den statischen Textbezeichner für das Steuerelement in der Aktivierreihenfolge vor das entsprechende Steuerelement zu setzen.
3. Sie können auch eine Kombination aus **STRG-TASTE** und Doppelklicken für Schaltflächen verwenden. In diesem Fall erstellt der Klassen-Assistent eine Nachrichtenbehandlungsfunktion für die Benachrichtigung BN_CLICKED in Ihrer Datensatzansichts-Klasse. Sie können den Code bearbeiten, um die durch diese Schaltfläche ausgelösten Aktionen zu implementieren.

Aufgabe 15: Verbinden der Kontrollelemente mit den Datenbankfeldern

Verbinden Sie die Kontrollelemente mit den Datenbankfeldern anhand folgender Tabelle:

Kontrollelement	Feld
IDC_EDIT_HEADWORD	m_headword
IDC_EDIT_DETAILS	m_headword1
IDC_EDIT_DESCRIPTION	m_description

Die DoDataExchange() Methode sieht danach folgendermaßen aus:

```
void CDictionaryView::DoDataExchange(CDataExchange* pDX)
{
      CRecordView::DoDataExchange(pDX);
      //{{AFX_DATA_MAP(CDictionaryView)
      DDX_FieldText(pDX, IDC_EDIT_HEADWORD, m_pSet->m_headword, m_pSet);
      DDX_FieldText(pDX, IDC_EDIT_DETAILS, m_pSet->m_headword1, m_pSet);
      DDX_FieldText(pDX, IDC_EDIT_DESCRIPTION, m_pSet->m_description,
      m_pSet);
      //}}AFX_DATA_MAP
}
```

Setzen von Parametern und Sortierungsfolge der Datenbank

Natürlich macht es selten Sinn, aus einer Datenbank alle Datensätze herauszulesen. Normalerweise möchte man immer die Anzahl der Datensätze sinnvoll einschränken können. Die **CRecordSet-Klasse** bietet dazu einen einfachen Mechanismus.

Aufgabe 16: Ermöglichen Sie es, aus der Datenbank ein Auswahlmenge herauszulesen

Vorgehensweise:

Sie müssen zunächst der Datenbankklasse eine Variable hinzufügen, die den neuen Parameter aufnimmt. Außerdem muß der Klasse bei der Initialisierung gesagt werden, wie die Auswahlmenge aussehen soll.

Erstellen Sie eine neue Member-Variable **m_strHeadwordParam** vom Typ CString) in CDictionarySet (Rechtsklick auf die Klasse im **ClassView**, dann "Member-Variable hinzufügen". Danach müssen wir DoFieldExchange() erweitern:

```
void CDictionarySet::DoFieldExchange(CFieldExchange* pFX)
{
       //{{AFX_FIELD_MAP(CDictionarySet)
       pFX->SetFieldType(CFieldExchange::outputColumn);
       RFX_Long(pFX, _T("[dictionarykey]"), m_dictionarykey);
       RFX_Long(pFX, _T("[languagelistkey]"), m_languagelistkey);
       RFX_Text(pFX, _T("[headword]"), m_headword);
       RFX_Text(pFX, _T("[headword1]"), m_headword1);
       RFX_Text(pFX, _T("[description]"), m_description);
       RFX_Byte(pFX, _T("[deleteable]"), m_deleteable);
       //}}AFX_FIELD_MAP

       // Parameter setzen
       // Dieser Wert wird in der Datensatzauswahlformel verwendet
       // (m_strFilter Parameter)
       pFX->SetFieldType(CFieldExchange::param);
       RFX_Text(pFX, _T("HeadwordParam"), m_strHeadwordParam);
}
```

Bitte beachten Sie: Die Änderungen müssen außerhalb der **AFX_FIELD_MAP** Sequenz erfolgen,

sonst werden sie von **Visual C++** eventuell überschrieben. Wichtig ist auch, daß dem **MFC** Rahmenwerk mittels **SetFieldType()** mitgeteilt wird, daß es sich hierbei um ein Parameter handelt. Sie können beliebig viele Parameter verwenden, deren Anzahl muß nur mit der Anzahl der Platzhalter in der Filteranweisung **`m_pSet->m_strFilter`** übereinstimmen und die Reihenfolge muß ebenfalls dieselbe sein.

Details zu **SetFieldType()** finden Sie in der Onlinehilfe. (Siehe **CRecordSet::SetFieldType** und **CRecordSet::m_nParams**)

OnInitialUpdate muß dann folgendermaßen angepaßt werden:

```
void CDictionaryView::OnInitialUpdate()
{

        m_pSet = &GetDocument()->m_dictionarySet;
        m_pSet->Connect();

        // Das "Dictionary" Ergebnis-Set parametrisieren und sortieren
        m_pSet->m_strFilter = "headword like ?";
        m_pSet->m_nParams = 1;
        m_pSet->m_strHeadwordParam = "A%";
        m_pSet->m_strSort = "headword";

        CRecordView::OnInitialUpdate();
}
```

Jetzt werden ausschließlich Wörter angezeigt, die mit dem Buchstaben "A" beginnen. Zum Verständnis hier eine kurze Beschreibung der Parameter und deren Bedeutung:

Anweisung	Funktion
`m_pSet->m_strFilter = "headword like ?";`	Teilt dem SQL Server mit, daß das Feld "**Headword**" dem später übergebenen Kriterium entsprechen muß.
`m_pSet->m_nParams = 1;`	Teil der **CRecordSet-Klasse** mit, daß wir ein Parameter haben (genau ein "?" in **m_strFilter**)
`m_pSet->m_strHeadwordParam = "A%";`	Setzt die Filteranweisung, die anstelle des Platzhalters "?" eingesetzt wird.
`m_pSet->m_strSort = "headword";`	Setzt die Sortierungsreihenfolge

Die **CRecordSet** Klasse stellt daraus eine korrekte SQL Abfrage zusammen und sendet sie an den SQL-Server. Das Resultat kann dann mit ihrer Hilfe ermittelt werden.

Bemerkung: Bis hier hin haben wir noch nicht allzuviel gewonnen, um das Wörterbuch einsetzen zu können, muß die Abfrage vom Anwender gesteuert werden. Es muß also möglich sein, **`m_pSet->m_strHeadwordParam`** zu ändern und die neuen Ergebnisse anzuzeigen. Dies werden wir in Aufgabe 18 in Angriff nehmen. Als kleinen Zwischenschritt vervollständigen wir jedoch zunächst die bestehende Anzeige und füllen noch das Kombinationsfeld mit der Sprachenliste.

Füllen des Kombinationsfeldes

Das Kombinationsfeld soll die verfügbaren Sprachen anzeigen. Diese können aus der Tabelle **LanguageList** geholt werden. Leider stellt **Visual C++** keine Funktionalität zur Verfügung, um den Inhalt eines Kombinationsfeldes (Combobox) mit einer Datenbanktabelle zu verbinden. Es bleibt also nur die Wahl, die Daten "von Hand" in die Kombinationsfelder zu schieben.

Eine elegantere (und OOP-mäßigere) Lösung wäre natürlich, die notwendige Funktionalität in einer Ableitung der CComboBox Klasse zu realisieren. Davon sehen wir aber ab, um das Beispiel nicht zu kompliziert zu machen.

Aufgabe 17: Füllen Sie das Kombinationsfeld mit dem Inhalt der Tabelle LanguageList

Vorgehensweise:

Zunächst muß das Kombinationsfeld an eine Variable gebunden werden, damit die notwendigen Manipulationen vorgenommen werden können. Außerdem muß ein neues Datenbankobjekt für den Zugriff auf die Tabelle angelegt werden. Danach können die Daten in einer Schleife in das Kombinationsfeld eingetragen werden. Da in der Tabelle **Dictionary** nicht die Sprache im Klartext, sondern nur ein numerischer Schlüssel auf die Tabelle **LanguageList** steht, tragen wir in das sichtbare Feld den Klartext ein, binden aber den numerischen Schlüssel unsichtbar als Datum daran.

Zunächst legen wir als eine neue Member-Variable an, die als **CComboBox** Objekt an das Kombinationsfeld gebunden wird. Es gibt zwei Wege, dies zu erledigen:

1. Starten Sie den Dialog-Editor, drücken Sie **STRG** und klicken Sie auf das Kombinationsfeld. Es erscheint der Dialog in Abbildung 46: Member-Variable "m_ctrlLanguage" hinzufügen und an Control binden. Passen Sie dort die Angaben dieser Abbildung gemäß an.
2. Starten Sie den **ClassWizard** (**STRG-W**), wählen Sie dort den Reiter "**Member Variablen**" und darauf **IDC_COMBO_LANGUAGE** an. Nun können Sie über das Button "**Variable hinzufügen**" den gewünschten Dialog aktivieren. Passen Sie dort die Angaben dieser Abbildung gemäß an.

Abbildung 46: Member-Variable "m_ctrlLanguage" hinzufügen und an Control binden

Danach starten wir den Klassenassistenten (**STRG-W**) und legen eine neue Klasse vom Typ **CRecordSet** an. Die Klasse soll **CLanguageSet** heißen. Den vorgeschlagenen Dateinamen akzeptieren wir unverändert. Der Assistent fragt nun automatisch nach der Datenquelle und der zugehörigen Tabelle. Auch hier ist die Datenquelle wieder **NFDIC**. Benutzername und Paßwort sind wiederum "**dba**" und "**sql**".

Als nächstes können wir die Tabelle auswählen:

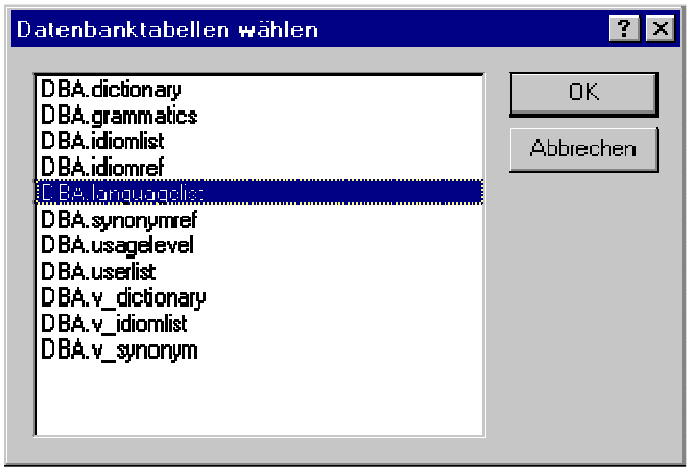


Abbildung 47: Datenbanktabellen auswählen

Damit sind alle notwendigen Angaben gemacht. Der Klassenassistent zeigt jetzt die neue Klasse an. Die Datenbankfelder sind aber noch nicht an Variablen gebunden, so daß nicht auf sie zugegriffen werden kann.

Wir haben die Möglichkeit, nur einzelne Spalten der Datenbanktabelle an Variablen zu binden, oder einfach alle. Die letztere Möglichkeit ist normalerweise nicht sinnvoll. Es sollten immer nur die Spalten gebunden werden, die auch benötigt werden, da sonst unnötige Zugriffs- und Datenaustauschoperationen erfolgen (die sehr zeitraubend sein können).

Wir benötigen nur die Spalten "**languagelistkey**" und "**description**". In der folgenden Abbildung wurden einfach alle Spalten verbunden. Sollten sich die Tabellen ändern, kann über das Button "Spalten aktualisieren" die Änderung übernommen werden. Das erfordert unter Umständen das Entfernen von Variablen (wenn Spalten gelöscht wurden) oder das Hinzufügen.

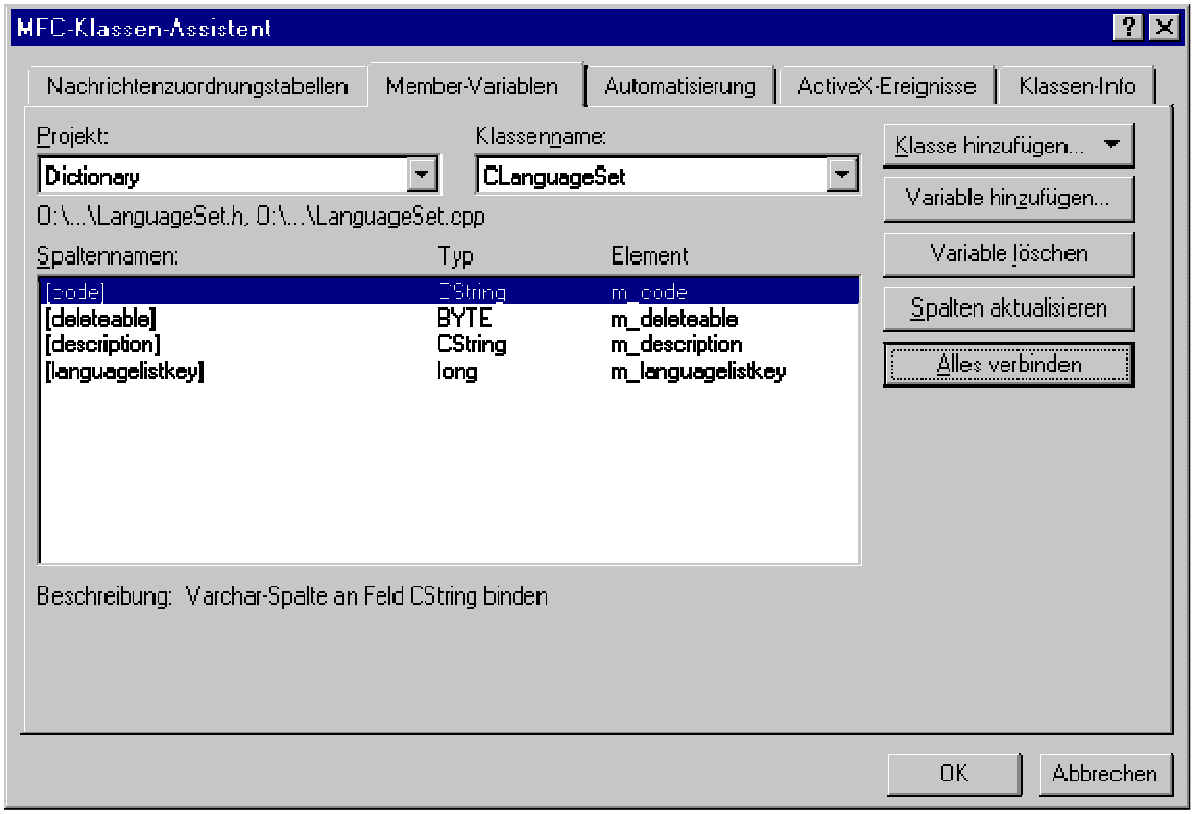


Abbildung 48: Der MFC-Klassen Assistent

Jetzt bleibt noch das Füllen des Kombinationsfeldes:

```
void CDictionaryView::OnInitialUpdate()
{

    int nIndex;

    // ... wie bisher

    // Kombinationsfeld mit allen Sprachen füllen
    // Wir verwenden hier dieselbe Datenbankverbindung wie CDictionarySet
    CLanguageSet LanguageSet(m_pSet->m_pDatabase);
    if (!LanguageSet.Open())
        return;

    CRecordView::OnInitialUpdate();

    m_ctrlLanguage.ResetContent();

    if (LanguageSet.IsOpen())
    {
        while (LanguageSet.IsEOF() != TRUE)
        {
            // Angezeigt wird der Text
            nIndex = m_ctrlLanguage.AddString(
                LanguageSet.m_description);

            // Als Datum wird aber der Schlüssel auf
            // LanguageList verwendet.

            if (nIndex != CB_ERR && nIndex != CB_ERRSPACE)
                m_ctrlLanguage.SetItemData( nIndex,
                LanguageSet.m_languagelistkey );
            LanguageSet.MoveNext();
```

```
        }
    }

}
```

Compilieren Sie nun den Code. Der Compiler fällt auf die Nase, weil er die neue Klasse **CLanguageSet** nicht kennt. Natürlich haben wir vergessen, den Header mit ihrer Definition einzubinden:

```
#include "LanguageSet.h"
```

Diese Zeile gehört an den Anfang von **DictionaryView.cpp**. Jetzt läßt sich das Programm compilieren und starten. Das Kombinationsfeld enthält tatsächlich die Sprachen (bisher Deutsch und Englisch), aber es wechselt nicht, wenn man sich zwischen den Datensätzen bewegt. Leider müssen wir auch die Positionierung "zu Fuß" realisieren.

Legen Sie also nun eine Funktion **SetComboBoxes(BOOL bSaveAndValidate)** in CDictionaryView an und fügen Sie folgenden Code ein:

```
void CDictionaryView::SetComboBoxes(BOOL bSaveAndValidate)
{
    int nIndex, nCount;

    // Das ist ein bißchen trickreich: in der Datenbank steht der Index
    // auf die LanguageList Tabelle. Diese Werte stehen auch im
    // Datenelement der Combobox. Also suchen wir den Wert heraus

    if (bSaveAndValidate)
    {
        // Daten sollen in Datenbank geschrieben werden
        long nValue =
        (long)m_ctrlLanguage.GetItemData( m_ctrlLanguage.GetCurSel() );

        if (nValue != CB_ERR)
            m_pSet->m_languagelistkey = nValue;
    }
    else
    {
        // Datenbankbewegung: Korrekte Zeile im Kombinationsfeld
        // suchen und setzen

        nCount = m_ctrlLanguage.GetCount();

        if (nCount != CB_ERR)
        {
            for (nIndex = 0; nIndex < nCount; nIndex++)
            {
                if (m_pSet->m_languagelistkey ==
                (long)m_ctrlLanguage.GetItemData( nIndex ))
                {
                    m_ctrlLanguage.SetCurSel(nIndex);
                    break;
                }
            }
        }
    }
}
```

Nach dem Füllen des Kombinationsfeldes muß positioniert werden.

```
void CdictionaryView::OnInitialUpdate()
{
    // ... wie bisher

    SetComboBoxes(FALSE);
}
```

Außerdem den Datenaustausch erweitern, damit bei Änderungen das Feld positioniert wird:

```
void CDictionaryView::DoDataExchange(CDataExchange* pDX)
{
     CRecordView::DoDataExchange(pDX);
     //{{AFX_DATA_MAP(CDictionaryView)
     DDX_Control(pDX, IDC_COMBO_LANGUAGE, m_ctrlLanguage);
     DDX_FieldText(pDX, IDC_EDIT_HEADWORD, m_pSet->m_headword, m_pSet);
     DDX_FieldText(pDX, IDC_EDIT_DETAILS, m_pSet->m_headword1, m_pSet);
     DDX_FieldText(pDX, IDC_EDIT_DESCRIPTION, m_pSet->m_description,
     m_pSet);
     //}}AFX_DATA_MAP

     SetComboBoxes(pDX->m_bSaveAndValidate);
}
```

Damit ist das Werk getan - die Felder zeigen alle die korrekten Werte, wenn die Datensätze gewechselt werden und Änderungen werden außerdem korrekt geschrieben.

Suchfunktion realisieren

Was wäre ein Wörterbuch ohne Suchfunktion? Der vorletzte Schritt zum vollständigen Wörterbuch ist also die Realisierung der Suche.

Aufgabe 18: Belegen Sie das "Suchen" Button mit einer Suchfunktion

Vorgehensweise:

Zunächst muß der Klick auf das "Suchen" Button mit einem Ereignis-Handler verknüpft werden. Außerdem müssen die beiden Kontrollelemente **IDC_EDIT_SEARCH** und **IDC_CHECK_FULLTEXT** jeweils an eine Variable gebunden werden, damit auf deren Werte zugegriffen werden kann. Legen Sie also zunächst mittels **ClassWizard** diese beiden gebundenen Werte an:

Control ID	Variable	Typ
IDC_EDIT_SEARCH	m_strSearch	Cstring
IDC_CHECK_FULLTEXT	m_bFulltext	BOOL

Starten Sie nun den Dialogeditor, klicken Sie mit der rechten Maustaste auf "Suchen" Button und wählen "Ereignisse" an (als Abkürzung genügt auch ein linker Doppelklick auf das Button). Markieren Sie das Ereignis "**BN_CLICKED**" und klicken Sie auf "Behandlungsroutine hinzufügen":

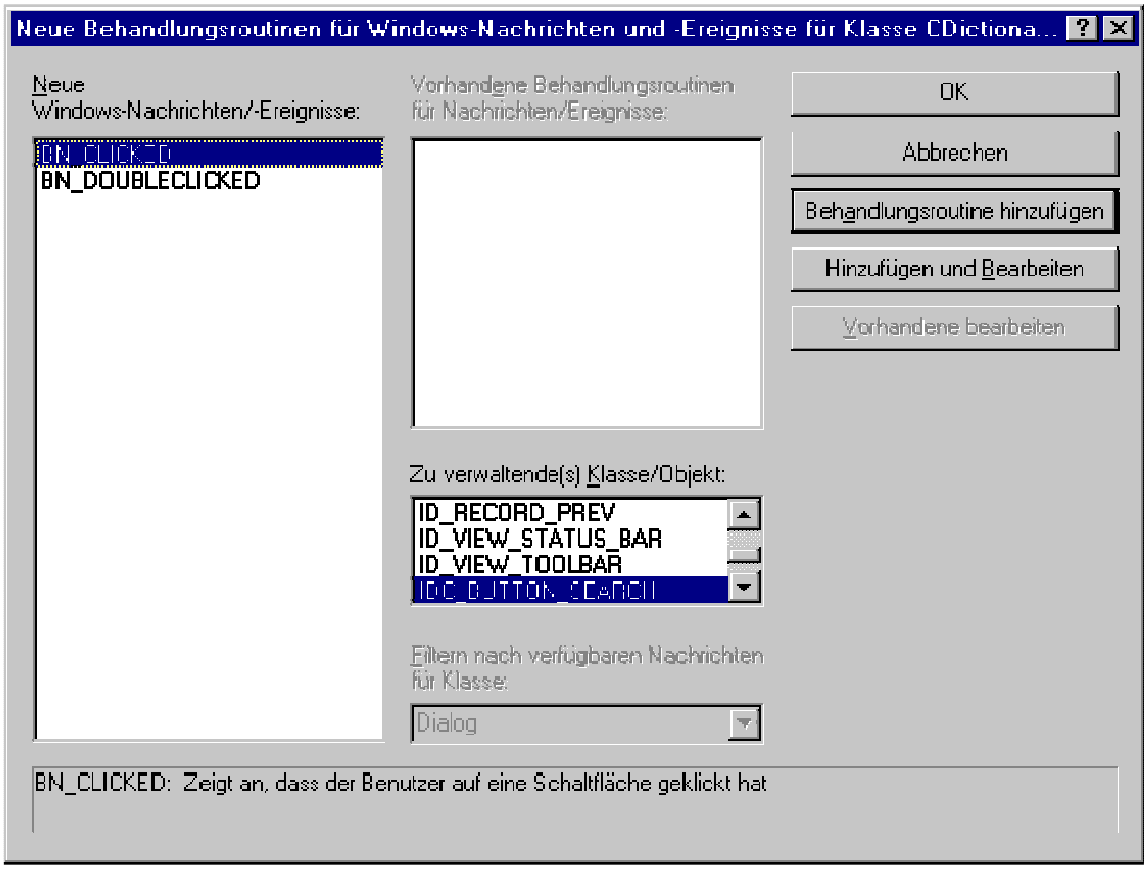


Abbildung 49: Pushbutton mit Funktionen belegen

Der Name der Funktion wird wie vorgeschlagen übernommen (OnButtonSearch). Die Funktion kann danach über den **ClassView** geladen werden.

Folgende Änderungen nehmen wir vor:

```
void CDictionaryView::OnButtonSearch()
{
      UpdateData(TRUE);

      m_pSet->m_strHeadwordParam =
           (m_bFulltext ? "%" : "") + m_strSearch + "%";

      if( !m_pSet->CanRestart( ) )
           return;    // Unable to requery

      m_pSet->Requery();

      // Daten übernehmen

      UpdateData(FALSE);
}
```

Zunächst wird mit **`UpdateData(TRUE)`** der aktuelle Stand aus den Controls ausgelesen (ansonsten enthalten die Variable noch nicht die aktuellen Werte!). Danach wird ein neue Parameterstring zusammengebaut und ein Requery() ausgelöst. Schließlich muß mittels **`UpdateData(FALSE)`** der geänderte Inhalt der Datenbank in die Controls geladen werden.

"Sicherheitsmaßnahmen" in der Entwicklungsversion

Damit ist der erste Teil des Programmes fertig. Wir ändern jetzt noch eine Kleinigkeit, damit

nicht aus Versehen Änderungen am Datenbestand vorgenommen werden können. An Ende von **OnInitUpdate()** setzen wir dazu folgenden Code ein:

```
void CDictionaryView::OnInitialUpdate()
{

    // ... wie bisher

    // Damit in der unfertigen Version mit dem Datenbestand kein Unfug
    // gemacht werden kann, deaktivieren wir noch die Kontrollelementen.
    // Das Kombinationsfeld ist kein Problem, da als Variable vorhanden
    m_ctrlLanguage.EnableWindow( FALSE );
    // Den anderen Kontrollelementen senden wir eine Nachricht
    SendDlgItemMessage(IDC_EDIT_HEADWORD, WM_ENABLE, FALSE, 0L);
    SendDlgItemMessage(IDC_EDIT_DETAILS, WM_ENABLE, FALSE, 0L);
    SendDlgItemMessage(IDC_EDIT_DESCRIPTION, WM_ENABLE, FALSE, 0L);
}
```

Wie Sie sehen, müssen wir manchmal doch auf die alten API-Funktionen zurückgreifen. Hier hilft uns **MFC** nur insofern, als wir eine Memberfunktion **`SendDlgItemMessage`** zur Verfügung haben, die uns den ersten Parameter, das Handle des Fensters, erspart.

Sollte es erforderlich sein, eine Nachricht an ein anderes Fenster zu schicken, muß die Original-API-Funktion verwendet werden. Diese steht uns nach wie vor zur Verfügung, indem wir den Bereichsoperator :: verwenden:

```
::SendDlgItemMessage(hWnd, IDC_CONTROL, WM_ENABLE, FALSE, 0L);
```

sendet eine Nachricht **WM_ENABLE** Nachricht an das Fenster, das sich mit dem Handle **hWnd** identifiziert. Diese Vorgehensweise ist keineswegs illegal, sonder durchaus üblich und erwünscht.

Eine Philosophie von **MFC** ist nämlich, daß API Funktionen nur dann als Member einer Klasse auftreten, wenn dies sinnvoll und vorteilhaft ist.

Die Übersetzung

Jetzt wird es spannend, denn endlich werden wir ein Tab mit den Übersetzungen einrichten.

Aufgabe 19: Realisieren Sie die Anzeige der Übersetzungen

Vorgehensweise:

Es sollen drei Tabs (Registerkarten) angezeigt werden, die folgende Daten enthalten:

Beschriftung	**Inhalt**	**Tabelle**
Übersetzung	Übersetzungen des Stichwortes als Liste	v_dictionary
Details	Eine einzelne Übersetzung im Detail	v_dictionary
Redewendungen	Liste mit Redewendungen zum Stichwort	v_idiomlist

Die Registerkarte ist leider ein etwas komplizierteres Gebilde, es besteht aus mehreren Dialogen, die die einzelnen Seiten (Karteireiter) repräsentieren und je nach dem, welche Seite gerade angezeigt wird, ein- oder ausgeblendet werden. Es ist aber auch möglich, nur einzelne Kontrollelemente auf die verschiedenen Ebenen zu legen. Diese Vereinfachung verringert insbesondere in unserem Fall den Aufwand immens.

Als ersten Schritt setzen wir im Dialog-Editor das Tab auf das Fenster unterhalb der bisherigen Kontrollelemente und benennen es in **IDC_TAB_TRANSLATION** um (Zur Erinnerung: rechter Mausklick auf das Kontrollelement und in dem Pop-Up-Menü "Eigenschaften" wählen). Danach erzeugen wir eine daran gebundene Member-Variable vom Typ **CTabControl**, die wir

m_TabTranslation nennen. Der einfachste Weg dorthin ist, bei gedrückter **STRG-Taste** im Dialogeditor auf die Registerkarte Doppelzuklicken und in dem dann eingeblendeten Dialog die notwendigen Angaben zu machen.

Compilieren und starten wir die Anwendung jetzt, wir das Tab zwar angezeigt, aber es sind keine einzelnen Seiten sichtbar. Diese müssen im Code definiert werden. Dazu passen wir wieder mal **`OnInitialUpdate()`** an:

```
void CDictionaryView::OnInitialUpdate()
{
     // ... wie bisher

     // Tab-Control einrichten
     TC_ITEM tie;
     tie.mask = TCIF_TEXT;

     tie.pszText = "Übersicht";
     m_TabTranslation.InsertItem(0, &tie);

     tie.pszText = "Details";
     m_TabTranslation.InsertItem(1, &tie);

     tie.pszText = "Redewendungen";
     m_TabTranslation.InsertItem(2, &tie);
}
```

Jetzt wird das Tab, noch ohne Kontrollelemente, schon so angezeigt, wie wir wünschen. Auch der Wechsel zwischen den Seiten ist möglich. Nun muß noch die Anzeige der Kontrollelemente auf den Seiten realisiert werden.

Damit das Tab verschiedene Reiter anzeigen kann, muß es einen Nachrichtenhandler besitzen, der auf den Wechsel der Tabseiten reagiert. Die zugehörige Nachricht heißt **TCN_SELCHANGE**. Verbinden wir also gleich einen Handler (standardmäßig **`OnSelchangeTabTranslation`**) damit. Wir benötigen die Funktion allerdings erst später (zur Erinnerung: einen ähnlichen Vorgang hatten wir bereits beim Verbinden des "Suchen" Buttons mit dem BN_CLICKED Ereignis).

Anzeige der Übersetzungen

Damit die Übersetzungen angezeigt werden, muß ein neuer Dialog angelegt werden, der dann auf dem Tabellenreiter angezeigt wird. Außerdem eine weitere Datenbankklasse für den notwendigen Datenzugriff.

Aufgabe 20: Erstellen Sie die Datenbankklassen für die Übersetzung

Vorgehensweise:

Wir erzeugen zunächst die notwendigen Datenbankklassen. Dies haben wir bereits für **CLanguageSet** getan, der Ablauf ist derselbe. Folgende Tabellen werden mit der jeweiligen **CRecordSet-Klasse** verknüpft, wobei alle Datenbankfelder verbunden werden:

Tab	**SQL Tabelle**	**MFC-Klasse**
Übersetzung	v_dictionary	Cv_dictionarySet
Details	v_dictionary	Cv_dictionarySet
Redewendungen	v_idiomlist	Cv_idiomlistSet

Die Vorgehensweise ist wieder:

1. Start des Klassenassistenten mit **STRG-W**
2. Wechsel auf das Tab "**Member-Variablen**"
3. Klicken Sie auf "**Klasse hinzufügen/Neu**"
4. Geben Sie als Klassennamen einen der beiden obigen Namen an
5. Geben Sie als Basisklasse **CRecordSet** an
6. Klicken Sie OK - jetzt werden Sie nach der Datenquelle gefragt
7. Geben Sie **NFDIC** an. Benutername und Paßwort sind "**dba**" und "**sql**"
8. Wählen Sie die zum Klassennamen zugehörige Tabelle aus.
9. Schließen Sie den Assistenten mit "OK".

Wiederholen Sie obige Vorgänge für die zweite Klasse.

Hinweis: Hier gilt ebenfalls das bereits beim Einrichten von **CLanguageSet** Gesagte - eigentlich sollten nur genau die Datenbankfelder gebunden werden, die auch später notwendig sind. Jedes unnötige Feld ist Resourcenverschwendung, angefangen vom Speicherbedarf bis hin zu zusätzlicher Netzwerkbelastung für Daten, die sowieso "weggeworfen" werden. Denken Sie also bitte daran, daß hier der Übersichtlichkeit willen absichtlich ineffektiv programmiert wurde!

Aufgabe 21: Fügen Sie die beiden Recordsets in die CDictionaryView ein

Die beiden neuen Klassen sollen als Member in **CDictionaryView** *verfügbar sein, damit über Controls auf sie zugegriffen werden kann. Die Klassen müssen also definiert und geöffnet werden, außerdem bei Datensatzbewegungen mitgeführt (sie enthalten die Übersetzungen und Redewendungen für jedes Stichwort).*

Vorgehensweise:

Klicken Sie im **ClassView** mit der rechten Maustaste auf **CDictionaryView**. Wählen Sie aus dem Pop-Up Menü "Member Variable hinzufügen" und erzeugen Sie folgende Variable:

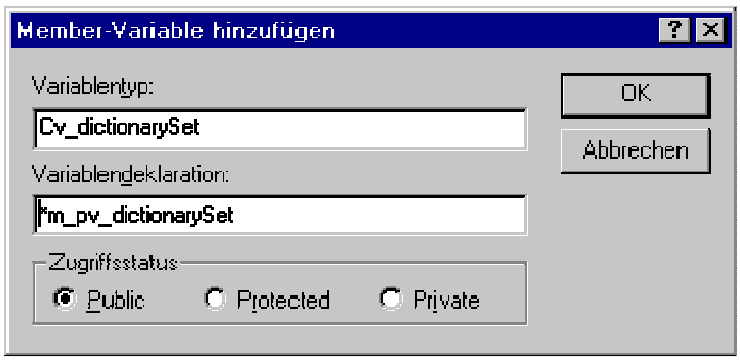


Abbildung 50: Member-Variable "*m_pv_dictionarySet" hinzufügen

Wiederholen Sie den Vorgang für den zweiten Recordset:

Abbildung 51: Member-Variable "*m_pv_idiomlistSet" hinzufügen

Damit sind die beiden Variablen definiert. Natürlich müssen die passenden Header-Files per Include eingebunden werden. Die Entwicklungsumgebung von VC++ "denkt" für uns daran, sofern sie die Klasse im Variablentyp identifizieren kann. Ein Blick in "**CDictionaryView.h**" zeigt uns das.

```
// DictionaryView.h : Schnittstelle der Klasse CdictionaryView
//
/////////////////////////////////////////////////////////////////////////////
//

#if
!defined(AFX_DICTIONARYVIEW_H__68941F7E_D0BC_11D1_A9B7_0020AF0B533E__INCLUD
ED_)
#define
AFX_DICTIONARYVIEW_H__68941F7E_D0BC_11D1_A9B7_0020AF0B533E__INCLUDED_

#include "v_dictionarySet.h"          // Hinzugefügt von ClassView
#include "v_idiomlistSet.h"           // Hinzugefügt von ClassView
```

Den notwendigen Code, um die beiden zusätzlichen Tabellen mitzuführen, packen wir am besten in eine zusätzliche Funktion **PositionTranslation**. Legen Sie also eine neue Memberfunktion dieses Namens an. Außerdem muß eine ganze Menge neuer Code geschrieben werden.

```
void CdictionaryView::OnInitialUpdate()
{
```

```
      // ... wie bisher
      // Zusätzliche Tabellen öffnen (gleiche Datenbankverbindung)
      m_pv_dictionarySet = new Cv_dictionarySet;
      m_pv_dictionarySet->m_pDatabase = m_pSet->m_pDatabase;
      m_pv_dictionarySet->m_strFilter = "dictionarykey2 = ?";
      m_pv_dictionarySet->m_nParams = 1;
      m_pv_dictionarySet->m_strSort = "headword";
      m_pv_idiomlistSet = new Cv_idiomlistSet;
      m_pv_idiomlistSet->m_pDatabase = m_pSet->m_pDatabase;
      m_pv_idiomlistSet->m_strFilter = "dictionarykey = ?";
      m_pv_idiomlistSet->m_nParams = 1;
      m_pv_idiomlistSet->m_strSort = "idiom";
      PositionTranslation();
}

CDictionaryView::~CDictionaryView()
{
      // Dynamisch erzeugte Objekte wieder aufräumen
      if (m_pv_dictionarySet != NULL)
         delete m_pv_dictionarySet;
      if (m_pv_idiomlistSet != NULL)
         delete m_pv_idiomlistSet;
}

void CdictionaryView::PositionTranslation()
{
       //Übersetzungen positionieren und Query abschicken
       m_pv_dictionarySet->m_nDictionarykey2Param = m_pSet-
       >m_dictionarykey;
       m_pv_idiomlistSet->m_nDictionarykeyParam = m_pSet->m_dictionarykey;
       if (!m_pv_dictionarySet->IsOpen())
       {
          // Beim ersten Aufruf Recordset öffnen
          m_pv_dictionarySet->Open();
       }
       else
       {
          // Ansonsten neu positionieren
          if( m_pv_dictionarySet->CanRestart( ) )
                  m_pv_dictionarySet->Requery();
       }
       if (!m_pv_idiomlistSet->IsOpen())
       {
          // Beim ersten Aufruf Recordset öffnen
          m_pv_idiomlistSet->Open();
       }
       else
       {
          // Ansonsten neu positionieren
          if( m_pv_idiomlistSet->CanRestart( ) )
                  m_pv_idiomlistSet->Requery();
       }
}

void CDictionaryView::OnButtonSearch()
{
       // wie bisher

       // Übersetzungen
       PositionTranslation();
}
```

Wenn wir jetzt versuchen, den Code zu compilieren, dann fehlen die beiden in **`PositionTranslation()`** referenzierten Membervariablen **`m_nDictionarykey2Param`** in **`m_pv_dictionarySet`** sowie **`m_nDictionarykeyParam`** in **`m_pv_idiomlistSet`**. Diese müssen wir zunächst anlegen.

Damit die Parameteranweisungen ausgewertet werden, muß wie bereits in **Aufgabe 16** der Parameter in den beiden **Recordset** Klassen übernommen werden. Als legen Sie zunächst in **Cv_dictionarySet** den Parameter als Variable an. Hier handelt es sich um einen numerischen Wert, daher verwenden wir **long**.

Abbildung 52: Member-Variable "m_nDictionaryKey2Param" hinzufügen

Dann muß auch die **`DoFieldExchange`** Funktion der Klasse angepaßt werden, damit der Parameter ausgewertet wird.

```
void Cv_dictionarySet::DoFieldExchange(CFieldExchange* pFX)
{
    // ... wie bisher
    // Parameter setzen
    // Dieser Wert wird in der Datensatzauswahlformel verwendet
    // (m_nDictionarykey2Param Parameter)
    pFX->SetFieldType(CFieldExchange::param);
    RFX_Long(pFX, _T("Dictionarykey2Param"), m_nDictionarykey2Param);
}
```

Der Klasse **Cv_idiomlistSet** fügen wir ebenfalls einen Parameter Typ **long** hinzu und passen die **`DoFieldExchange`** Funktion an.

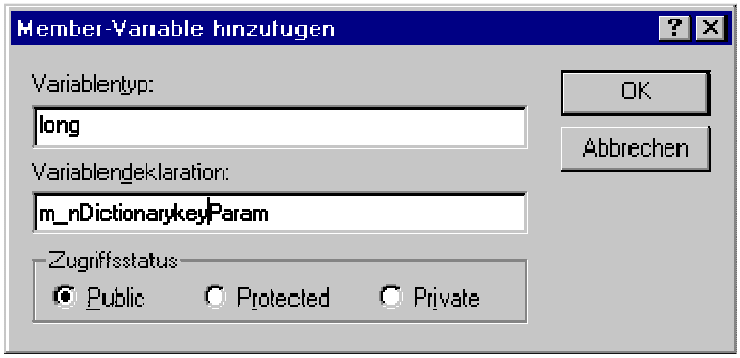


Abbildung 53: Member-Variable "m_nDictionaryKeyParam" hinzufügen

```
void Cv_idiomlistSet::DoFieldExchange(CFieldExchange* pFX)
{
    // ... wie bisher
    // Parameter setzen
    // Dieser Wert wird in der Datensatzauswahlformel verwendet
    // (m_nDictionarykeyParam Parameter)
```

```
    pFX->SetFieldType(CFieldExchange::param);
    RFX_Long(pFX, _T("DictionarykeyParam"), m_nDictionarykeyParam);
}
```

Aufgabe 22: Zeigen Sie die Übersicht in einem Listenfeld auf dem ersten Tab an

Auf dem ersten Tab soll ein Listenfeld erscheinen, auf der die Übersetzung als Liste angezeigt wird.

Vorgehensweise:

Zunächst plazieren wir das Listenfeld auf dem Dialog. Wir benennen es sofort in **IDC_LIST_OVERVIEW** um. Schalten Sie außerdem in den Eigenschaften unter "Formate" die Einstellung "sortiert" aus. Dies wird vom SQL Server erledigt. Klicken Sie dazu im **ResourceView** unter Dialoge auf **IDD_DICTIONARY_FORM** und übernehmen Sie das Listenfeld von der Toolbox. Stören Sie sich nicht daran, daß das Tab das Feld vorläufig überdeckt. Es ist auch nicht notwendig, es zu vergrößern. Die Anpassung werden wir im Code vornehmen.

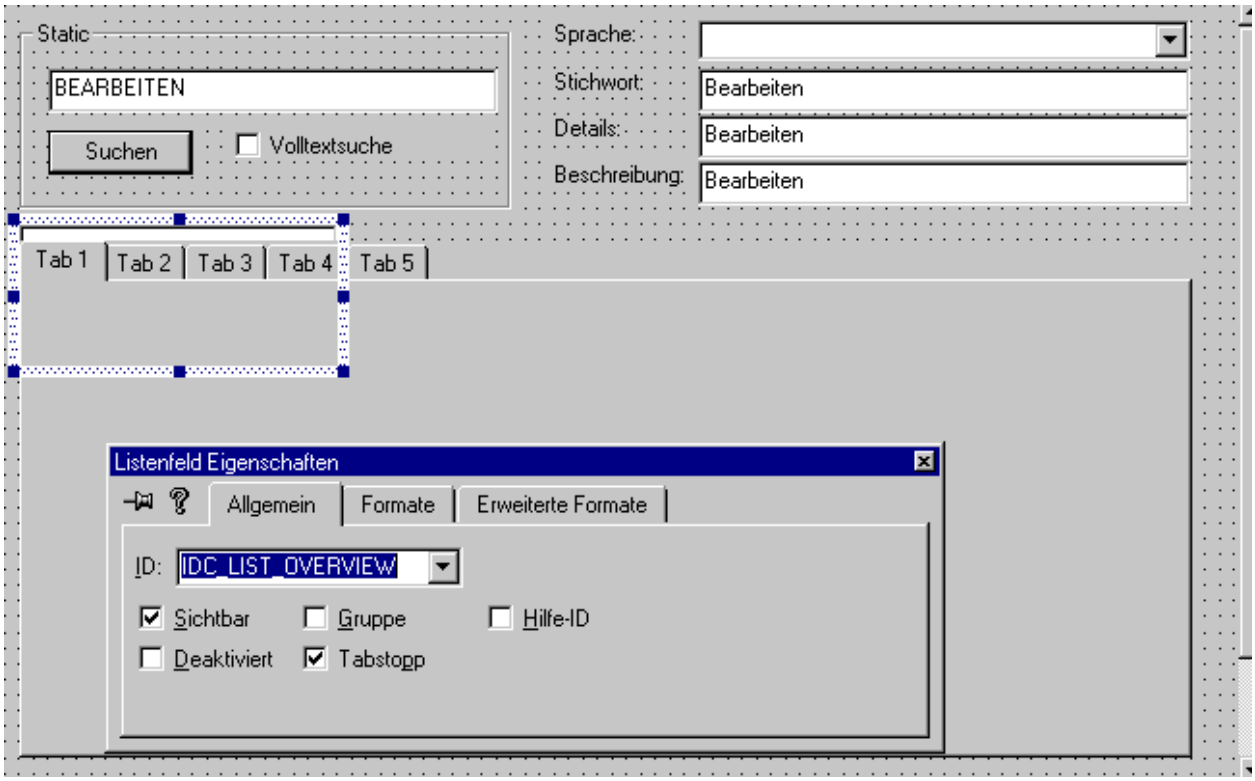


Abbildung 54: Das Ausgabefenster des Wörterbuchs

Natürlich muß auch für dieses Control wieder eine Variable mitgeführt werden, damit mit MFC Klassen auf es zugegriffen werden kann. (Verwenden Sie dazu wieder **ClassWizard**).

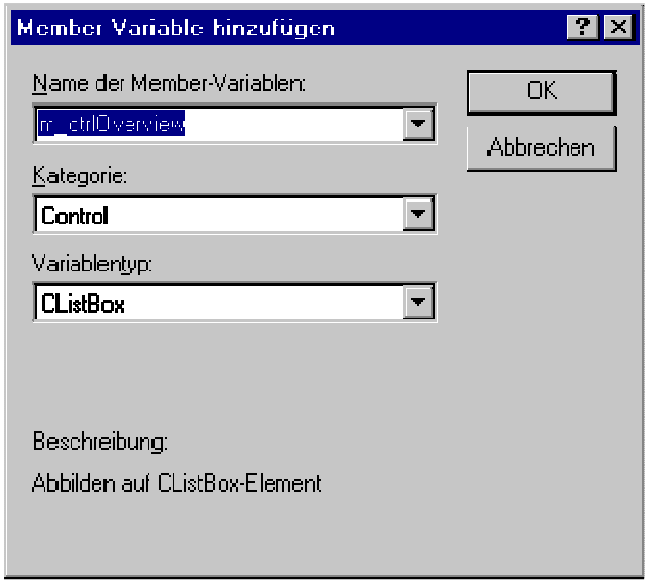


Abbildung 55 : Member-Variable "m_CtrlOverview" hinzufügen und an Control binden

Mit Hilfe dieser Variablen kann das Listenfeld gefüllt werden. Dazu muß die Funktion PositionTranslation etwas erweitert werden.

```
void CdictionaryView::PositionTranslation()
{
      // ... wie bisher

      // Füllen der Übersetzung (Übersicht)
      m_ctrlOverview.ResetContent();
      if (m_pv_dictionarySet->IsOpen())
      {
           while (m_pv_dictionarySet->IsEOF() != TRUE)
           {
                // Anzeigen der Übersetzung
                m_ctrlOverview.AddString(
                     m_pv_dictionarySet->m_headword);
                m_pv_dictionarySet->MoveNext();
           }
      }
}
```

Damit die Anzeige des Listenfeldes wechselt, muß die Funktion **PositionTranslation** immer wieder angestoßen werden, wenn der Datensatz gewechselt wird. Das wird durch die Funktion **OnMove** erledigt, die virtuell in der Oberklasse definiert ist. Mit Hilfe des ClassWizard erstellen wir sie auch in **CDictionaryView**.

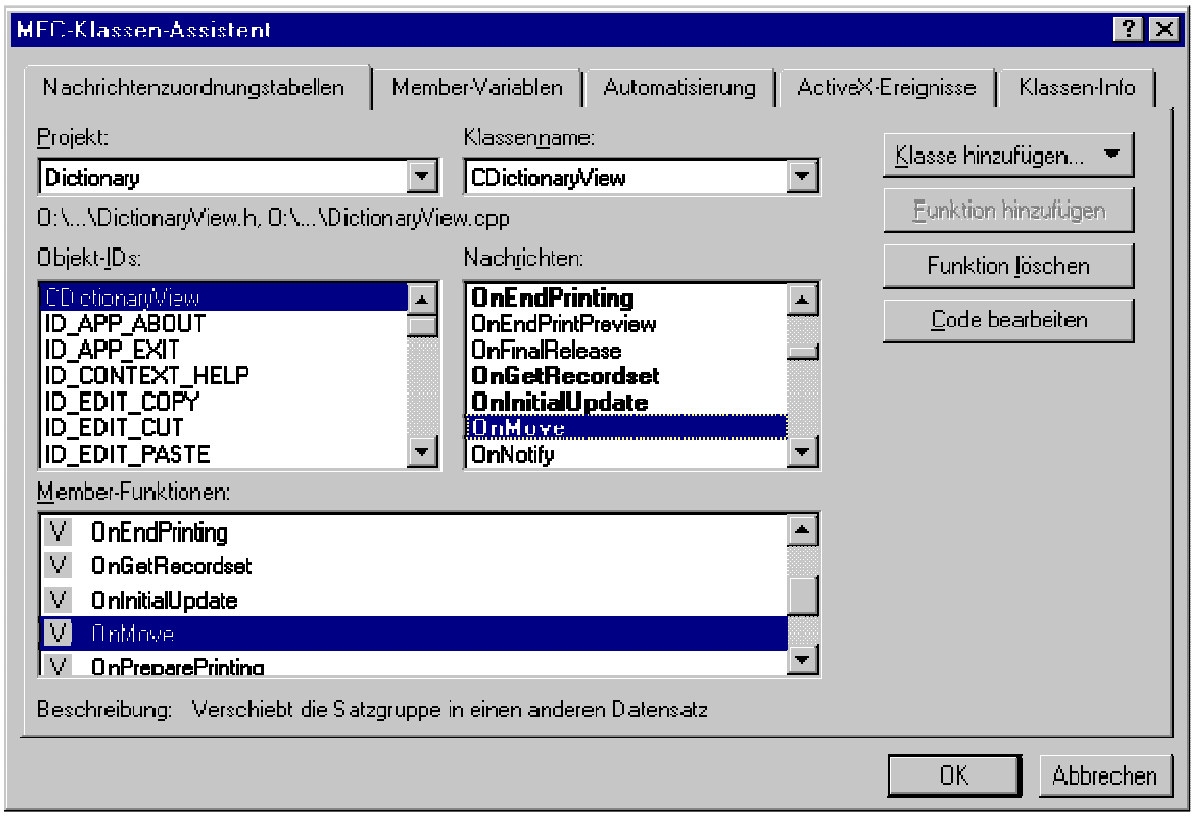


Abbildung 56: Der MFC-Klassen-Assistent

```
BOOL CDictionaryView::OnMove(UINT nIDMoveCommand)
{
      BOOL bResult = CRecordView::OnMove(nIDMoveCommand);

      // Anzeigen der Übersetzungen
      if (bResult)PositionTranslation();
           return bResult;
}
```

Aufgabe 23: Verwenden Sie das dritte Tab für die Liste der Redewendungen

Auf dem letzten Tab soll eine Listbox erscheinen, auf der die Redewendungen zum Stichwort als Liste angezeigt wird.

Vorgehensweise:

Das Tab mit den Redewendungen wird analog angelegt. Kopieren Sie einfach das Listenfeld für die Übersetzungen und ändern Sie den Namen in **IDC_LIST_IDIOMS**.

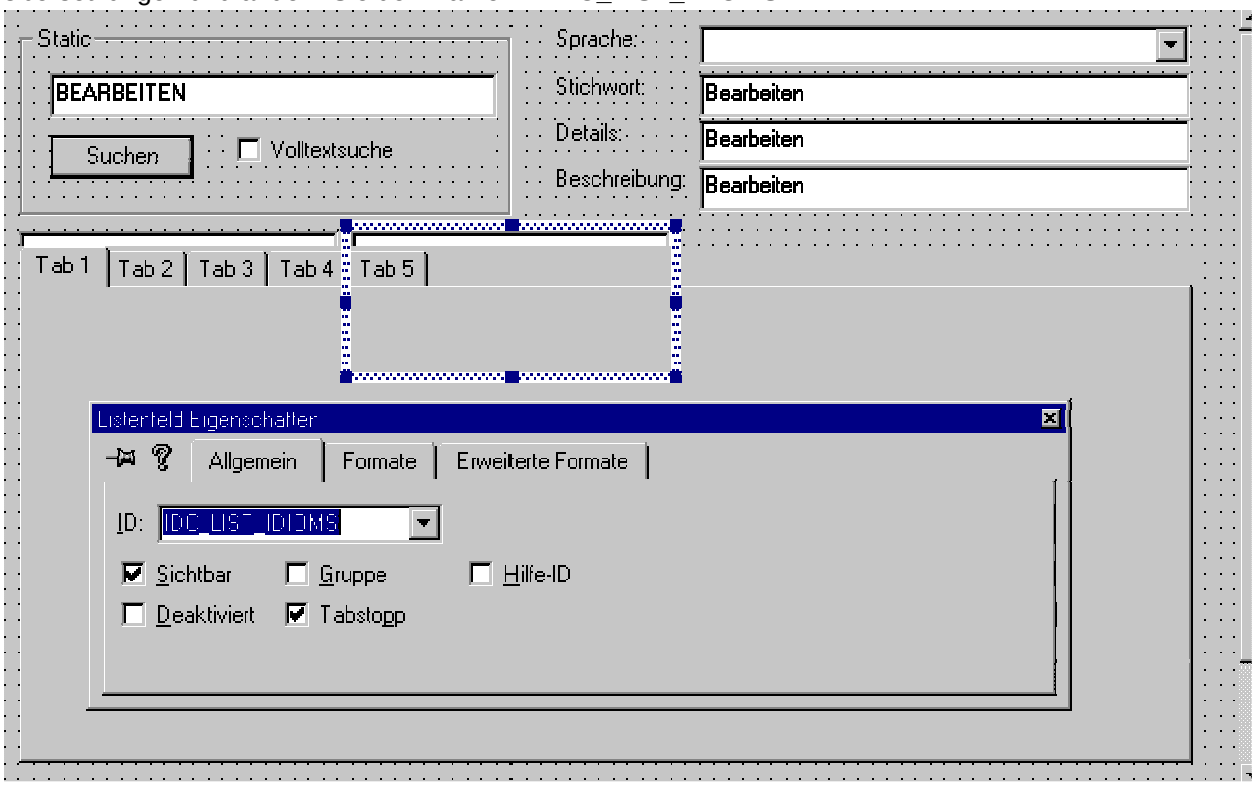


Abbildung 57: Änderung von Control-Namen

Auch hier wird wieder eine Variable für den Zugriff benötigt:

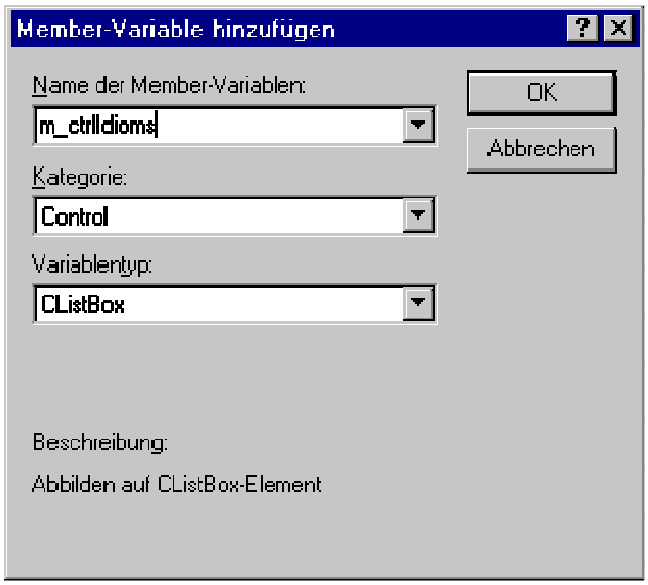


Abbildung 58: Member-Variable "m_ctrlIdioms" hinzufügen und an Control anbinden

Das Füllen der Liste erfolgt fast auf das I-Tüpfelchen wie das Füllen der Übersetzung. Ein paar neue Zeilen in PositionTranslation reichen aus:

```
void CDictionaryView::PositionTranslation()
{
        // ... wie bisher
        // Füllen der Redewendungen
        m_ctrlIdioms.ResetContent();
        if (m_pv_idiomlistSet->IsOpen())
        {
                  while (m_pv_idiomlistSet->IsEOF() != TRUE)
                  {
                            // Angezeigt wird der Text
                            m_ctrlIdioms.AddString(
                                        m_pv_idiomlistSet->m_Idiom);
                            m_pv_idiomlistSet->MoveNext();

                  }
        }
}
```

Die Listen werden jetzt immer gleichzeitig angezeigt und liegen an der falschen Stelle auf dem Fenster. Positionieren wir sie deshalb neu:

```
void CDictionaryView::OnInitialUpdate()
{
      // ... wie bisher

      // Größe und Position des Listenfeldes

      RECT Rect, itemRect;
      // Position des Tabs in Bildschirmkoordinaten
      m_TabTranslation.GetWindowRect( &Rect );
      // Umrechnen in Clientkoordinaten für das Fenster
      ScreenToClient( &Rect ) ;
      // Höhe des Reiters einrechnen
      m_TabTranslation.GetItemRect( 0, &itemRect ) ;
      Rect.top += itemRect.bottom + 5;
      Rect.bottom -= 5;
      Rect.left += 5;
      Rect.right -= 5;
      // Listenfelder positionieren
      m_ctrlOverview.MoveWindow( &Rect );
      m_ctrlIdioms.MoveWindow( &Rect );

      // Erstes Tab anwählen und Controls ein/ausblenden
      m_TabTranslation.SetCurSel(0);
      UpdateTabs(0);
}
```

Die Funktion **UpdateTabs** ist neu. Ohne sie würden alle Ebenen des Karteireiters gleichzeitig angezeigt und auf den Wechsel der Laschen überhaupt nicht reagiert. Legen Sie also eine neue Memberfunktion in **CDictionaryView** an:

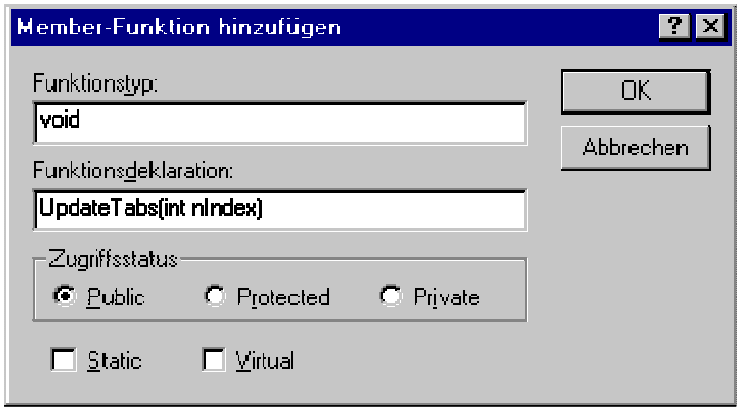


Abbildung 59: Member-Funktion "UpdateTabs()" hinzufügen

```
void CDictionaryView::UpdateTabs(int nIndex)
{
	switch (nIndex)
	{
		case 0:

			m_ctrlOverview.ShowWindow( SW_SHOW );
			m_ctrlIdioms.ShowWindow( SW_HIDE );
			break;

		case 1:

			m_ctrlOverview.ShowWindow( SW_HIDE );
			m_ctrlIdioms.ShowWindow( SW_HIDE );
			break;

		case 2:

			m_ctrlOverview.ShowWindow( SW_HIDE );
			m_ctrlIdioms.ShowWindow( SW_SHOW );
			break;
	}
}
```

Eine kleine Änderung an der Methode **OnSelchangeTabTranslation()** muß jetzt noch vorgenommen werden, damit die jeweiligen Ebenen je nach dem, welche Lasche aktiviert wurde, richtig ein- und ausgeblendet werden:

```
void CDictionaryView::OnSelchangeTabTranslation(NMHDR* pNMHDR, LRESULT*
pResult)
{
	int nIndex = m_TabTranslation.GetCurSel();

	UpdateTabs(nIndex);

	*pResult = 0;
}
```

Damit wird beim Wechsel der Tabs die richtige Liste angezeigt, und der Inhalt an das jeweils angewählte Fremdwort angepaßt. Jetzt ist das Wörterbuch endlich verwendbar!

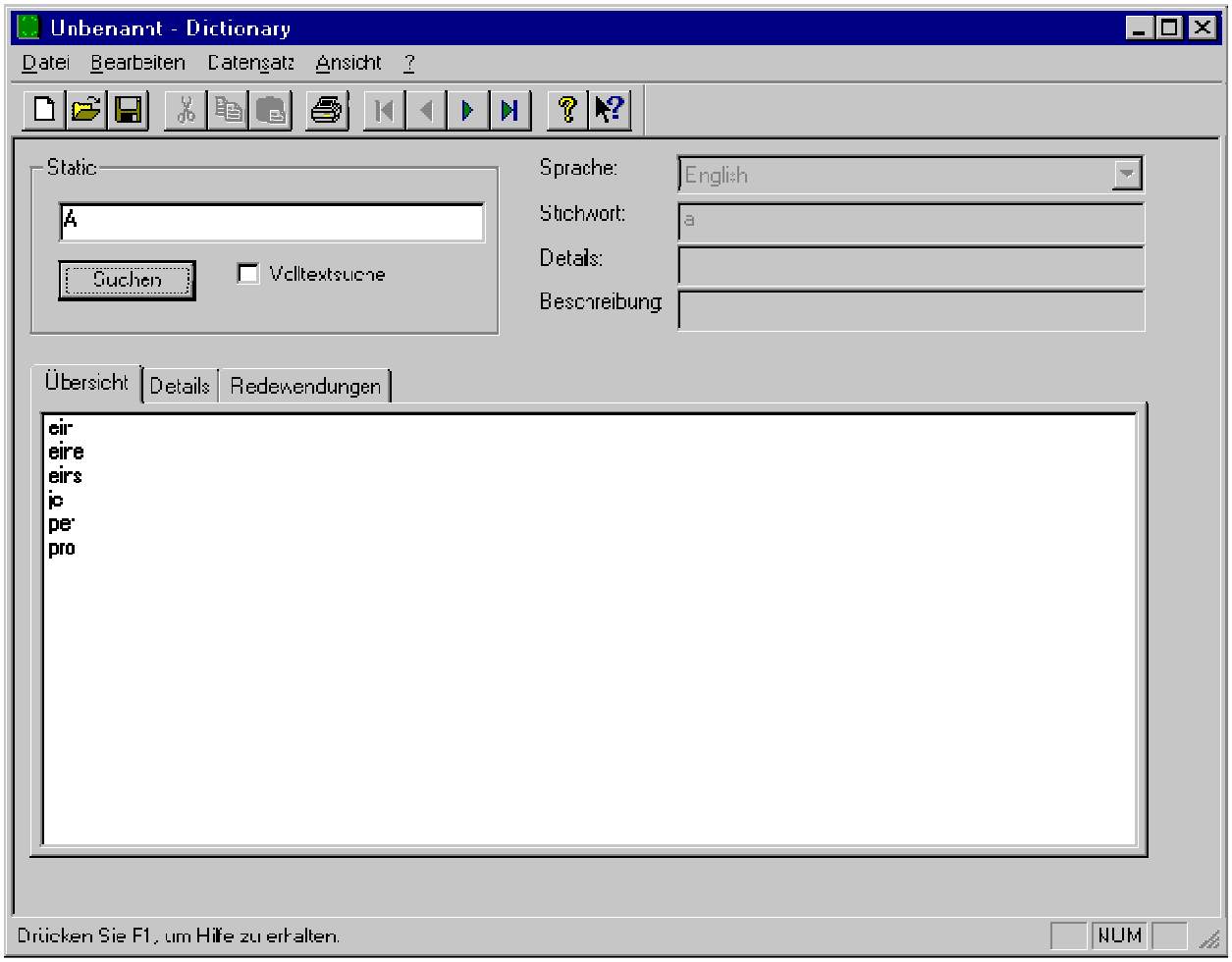


Abbildung 60: Das Wörterbuch

Aufräumen und Schönheitskorrekturen

Das Beispielprojekt Wörterbuch ist nun fast vollständig. Es gibt noch aber ein paar Dinge, an die es Hand anzulegen gilt. Mit dem im Seminar erworbenen Wissen können Sie diese Aufgaben aber leicht selber vervollständigen. Die Todo-Liste sieht folgendermaßen aus:

- Anpassung der Online-Hilfen
- Belegen des zweiten Tabs mit dem Detaildaten
- Entfernen der überflüssigen Menüpunkte aus dem Hauptmenü

Tips:

Zu 2: Positionieren Sie die Controls "ganz normal" auf dem Tab und blenden Sie diese wie die beiden Listen mit **ShowWindow** ein und aus. Der Datenaustausch erfolgt wie gehabt mit **DoFieldExchange**, jedoch müssen Sie ihn "von Hand" programmieren (außerhalb des von **VC++** gewarteten Fragmentes). Statt **m_pSet** gehört natürlich **m_pv_dictionarySet** als Parameter in die Funktion.

Zu 3: Es gibt einige Menüpunkte, die nicht relevant sind: Speichern, Speichern unter und Laden beispielsweise. Diese können Sie entfernen. Die Druckfunktionen können Sie deaktivieren (falls Sie die folgende Wunschliste noch bearbeiten wollen).

Es gibt noch eine Liste von Wünschen, die etwas anspruchsvoller sind. Diese können Sie, wenn sie sattelfester mit VC++ sind, später angehen:

1. Realisierung der Druckfunktionen
2. Erstellung von Zusatzfenstern zur Wartung der Sprachen und der zusätzlichen Tabellen.

Anhang

Abbildungsverzeichnis

Abbildung 1: MS Visual C++ Master Setup 13
Abbildung 2: Visual C++ Setup Installationsoptionen 14
Abbildung 3: Visual C++ Programmgruppe 15
Abbildung 4: Visual C++ Process Viewer 16
Abbildung 5: WinDiff 17
Abbildung 6: MS Visual C++ Logo 19
Abbildung 7: Developer Studio mit zwei installierten Programmiersprachen 20
Abbildung 8: Das Ausgabe-Fenster 21
Abbildung 9: Der Arbeitsbereich 21
Abbildung 10: Neuen Arbeitsbereich einrichten 27
Abbildung 11: Neues Projekt einrichten 29
Abbildung 12: MFC-Anwendungsassistent 30
Abbildung 13: Neue Projektinformationen 33
Abbildung 14: Kontextmenü des Arbeitsbereichs 34
Abbildung 15: Popup Menü für Ordner 35
Abbildung 16: Das Menü "Erstellen" 36
Abbildung 17: Menüleiste Arbeitsbereich 36
Abbildung 18: Das generierte Minimalprogramm 37
Abbildung 19: ClassWizard Fehlermeldung 41
Abbildung 20: Klasseninformationen bereinigen 41
Abbildung 21: Der ClassWizard 42
Abbildung 22: ClassWizard mit zwei neuen Nachrichten 43
Abbildung 23: Der Editor mit den neuen Haltepunkten 45
Abbildung 24: Befehle des Debuggers 46
Abbildung 25: C++ Logo 50
Abbildung 26: Kategorien der MFC Hierarchie 114
Abbildung 27: MFC 4.21 Übersicht 116
Abbildung 28: Neuanlage Arbeitsbereich "Dictionary" 119
Abbildung 29: Anlegen einer MFC Anwendung 120
Abbildung 30: Der MFC-Assistent 1. Schritt 121
Abbildung 31: Der MFC-Assistent 2. Schritt 121
Abbildung 32: Datenbankoptionen 121
Abbildung 33: Verbindung mit SQL Server 122
Abbildung 34: Angabe der gewünschten Tabelle 122
Abbildung 35: Der MFC-Assistent 3. Schritt 123
Abbildung 36: Projektinformationen 124
Abbildung 37: Pop-Up Menü des ClassView 126
Abbildung 38: Member-Funktion "Connect()" hinzufügen 126
Abbildung 39: Member-Variable "m_bManualDatabase" hinzufügen 127
Abbildung 40: Member-Funktion hinzufügen: Destruktor für CDictionarySet 127
Abbildung 41: Das Hauptfenster der generierten Minimalanwendung 129
Abbildung 42: Der Dialogeditor 130
Abbildung 43: Ändern der Eigenschaften eines Kontrollelementes 131
Abbildung 44: DDX für Fremdobjekte 132
Abbildung 45: Klassen-Assistent: Zuordnung von Formularsteuerelementen zu Datensatzgruppen-Feldern 134
Abbildung 46: Member-Variable "m_ctrlLanguage" hinzufügen und an Control binden 138
Abbildung 47: Datenbanktabellen auswählen 139
Abbildung 48: Der MFC-Klassen Assistent 140
Abbildung 49: Pushbutton mit Funktionen belegen 143
Abbildung 50: Member-Variable "*m_pv_dictionarySet" hinzufügen 147
Abbildung 51: Member-Variable "*m_pv_idiomlistSet" hinzufügen 147
Abbildung 52: Member-Variable "m_nDictionaryKey2Param" hinzufügen 149
Abbildung 53: Member-Variable "m_nDictionaryKeyParam" hinzufügen 149

Abbildung 54: Das Ausgabefenster des Wörterbuchs..... 150
Abbildung 55 : Member-Variable "m_CtrlOverview" hinzufügen und an Control binden..... 151
Abbildung 56: Der MFC-Klassen-Assistent 152
Abbildung 57: Änderung von Control-Namen..... 153
Abbildung 58: Member-Variable "m_ctrlIdioms" hinzufügen und an Control anbinden 153
Abbildung 59: Member-Funktion "UpdateTabs()" hinzufügen 155
Abbildung 60: Das Wörterbuch..... 156
Abbildung 61: Die Struktur der SQL Datenbank 165

Abkürzungen und Fachbegriffe

Es gibt einen ganze Handvoll von Begriffen aus der Computertechnik, die den meisten Anwendern zumindest geläufig sind, aber eigentlich so gut wie nichts sagen. Deshalb folgt hier eine Liste mit Begriffen, die im Verlaufe des Seminars ziemlich oft verwendet werden. Einige der Begriffe beschreiben verwandte Dinge und zeugen eigentlich eher vom Einfallsreichtum der Entwickler was das Erfinden von wohlklingenden Namen betrifft.

API (Application Programming Interface)

Eine **API** oder **Programmiererschnittstelle** ist eine Schnittstelle zu einer anderen Software, die ein Programmierer verwenden kann, um Software für diese zu erstellen oder die Fremdsoftware zu verwenden.

- ➔ Es handelt sich meist um eine Sammlung von Funktionsdeklarationen, mit denen mit der Fremdsoftware kommuniziert werden kann. Das ist bei der **Windows API** der Fall.
- ➔ Es kann aber ebenso eine Liste von Nachrichten, die dem anderen Programm geschickt werden kann, handeln. Das wird bei Windows Programmen oft so gehandhabt.
- ➔ Schließlich gibt es auch **APIs**, die aus Sammlungen von Funktionen bestehen. Diese stehen entweder als Source oder als Library zur Verfügung und erlauben den direkten Zugriff beispielsweise auf Datenbank-Dateien einer anderen Anwendung.

SDK (Software Development Kit)

SDK oder **Software Development Kit** bezeichnet eine Sammlung von Informationen und meist auch Quellcodes, die die Entwicklung von Software für eine andere Software erleichtern. Die **Windows API** ist z.B. auch ein **Software Development Kit** für die Entwicklung von Windows Programmen.

ODBC (Open Database Connectivity)

1992 von Microsoft zusammen mit anderen Firmen eingeführter Standard. **ODBC** (**Open Database Connectivity**, zu deutsch etwa "Offene Datenbankverbindungen") ist eine **Programmiererschnittstelle** (**API**) zu den Daten. Diese können z.B. auf einem SQL Server liegen. Es kann sich aber auch um dBASE Dateien (*.dbf) oder sogar reine Textdateien (*.txt, *.csv, *.asc) handeln.

Die tatsächlichen Zugriffe werden von Treibern ausgeführt, die dazu installiert und konfiguriert werden müssen. Diese Treiber sind auf genau einer Seite normiert, nämlich durch die **ODBC API**. Was der Treiber sonst alles anstellen muß, um auf "sein" Datenbankformat zuzugreifen, braucht den Entwickler letztlich nicht zu kümmern. Der Vorteil dieses Systems ist es, daß diese Treiber zumindest in gewissem Rahmen beliebig austauschbar sind und die Anwendungen somit nicht an ein bestimmtes Datenbankformat gebunden sind.

DAO (Data Access Objects)

DAO (**Data Access Objects**) ist eine Sammlung Klassen, die Zugriffe auf die populäre Microsoft Access Datenbanken erlauben. Der Vorteil gegenüber **ODBC** ist, daß eine Ebene (der Treiber) wegfällt und damit eine bessere Leistung erreicht wird. Der Nachteil ist, daß **DAO** augenblicklich nur MS Access unterstützt und die Anwendung somit an ein bestimmtes Datenbankformat gebunden ist.

OLE DB

Eine allgemein einsetzbare Anzahl Schnittstellen, die es den Programmierern erlauben, Datenzugriffe als Komponenten mittels des **Component Object Models** (**COM**) zu machen. **OLE DB** (OLE = **Object Linking and Embedding**) ermöglicht es Anwendungen, einen standardisierten Zugriff auf Daten zu haben, die in **DBMS** (**Database Management Systemen**) und nicht **DBMS** (z.B. dBASE) gespeichert sind, während sie trotzdem den Vorteil einer

Datenbanktechnologie haben, ohne dazu Daten von ihrem Ort zu einem **DBMS** verschieben zu müssen.

ActiveX/OLE

ActiveX bzw. früher **OLE** (**Object Linking and Embedding**) beschreibt eine Technologie, die es ermöglicht, fremde Anwendungen in eigene einzubinden und dort als Teil der eigenen laufen zu lassen. So kann man z.B. ein excelkompatibles **ActiveX** Control erwerben, mit dem man bequem die Leistung von Excel Tabellen in eigene Anwendungen einbauen kann. Diese Controls können nach dem Bausteinprinzip eingebunden werden.

Man spricht eigentlich von **ActiveX** Komponenten, wenn diese netzwerk- oder internetfähig sind.
ActiveX basiert auf dem **Component Object Model** (**COM**).

Component Object Model (COM)

Eine offene Architektur zur plattformübergreifenden Entwicklung von Client/Server Anwendungen basierend auf objektorientierter Technik. Clients haben Zugriff auf ein Objekt mittels einer Schnittstellen, die in dem Objekt implementiert ist. **COM** ist nicht an eine Sprache gebunden, so kann jede Sprache, die **ActiveX** Komponenten erzeugen kann, auch **COM** Applikationen erstellen.

MFC (Microsoft Foundation Class)

Die **Microsoft Foundation Class** (**MFC**) ist eine Sammlung von über 100 C++ Klassen für verschiedene Anwendungsgebiete. Der Schwerpunkt liegt bei der Entwicklung von Windows 3.1 und Windows 95/NT Anwendungen. Damit stellt **MFC** u.a. einen vollgültigen Ersatz für die pure Windows **API** Programmierung dar. Mit großer Wahrscheinlichkeit findet der Entwickler eine Klasse, die seinen Ansprüchen genügt oder sich per Ableitung für seine Wünsche anpassen läßt.

Die Ungarische Notation

f	a flag (boolean, logical). The qualifier (see below) should describe the condition that will cause this flag to be set (e.g. fError would be clear if there were no error, set if one exists). This tag may refer to a single bit, a byte, or a word; often it will be an object of type BOOL (defined by the application, usually as int). Usually the object referred to will contain either 1 (fTrue, TRUE) or 0 (fFalse, FALSE). In some instances, other values may be used, either for efficiency or historical reasons; such a use usually indicates that another type may be more appropriate.
ch	a one-byte character. Note that this is not adequate for Kanji.
st	a Pascal-type string (first byte is count, remainder is the actual characters). Typically refers to a pointer to the actual memory. This should be the most common type of string used in the Applications group; it is more efficient than an sz (below).
sz	a zero-terminated string, or a pointer to it. These are most often used to interface to an operating system (or equivalent) that requires them; for most other uses, an st is preferable. Unfortunately, C string constants are normally zero-terminated, so it takes a little more effort to use st's; the effort is worth it. The Applications Development compiler provides ways to make strings constants st's.
fn	a function. Since about the only thing you can do with a function is take its address, this almost always has a "p" prefix (see below). For this reason, in some applications fn is itself used to mean pointer to a function.
w	a word (typically 16 bits). For most purposes, this is an incorrect usage, since the usage of the word is specific to a particular type of word, and should be so distinguished. Correct usages are generally limited to generic subroutines (e.g. sort an array of words) that can deal with a number of different types; another common use is in conjunction with the prefix c (see below), to produce a count of words (the size) for some object. The exact meaning of w is also somewhat loose; it sometimes means a signed quantity and sometimes unsigned.
b	a byte (typically 8 bits). The same warnings apply to this as to w.
l	a long (typically 32 bits). The same warnings apply to this as to w.
u	an unsigned word (typically 16 bits). The same as w, except this is always unsigned.
bit	a single bit. Typically used to specify bits used within other types. This concept is usually better handled with the "f" and "sh" prefixes (see below).
v	a void. This corresponds to the C definition of void, meaning that the type is not specified. This type will never be used without a "p" prefix since it is not possible to have an unspecified type for a variable; conceivably ther are additional prefixes (e.g. ppv), but such a usage is unlikely. It is perfectly valid to assign a pv to a pointer of any other type, or vice versa. The major use of this type is for generic subroutines (such as allocate and free) which return or take as arguments pointers of various types.

Die Struktur der SQL Datenbank

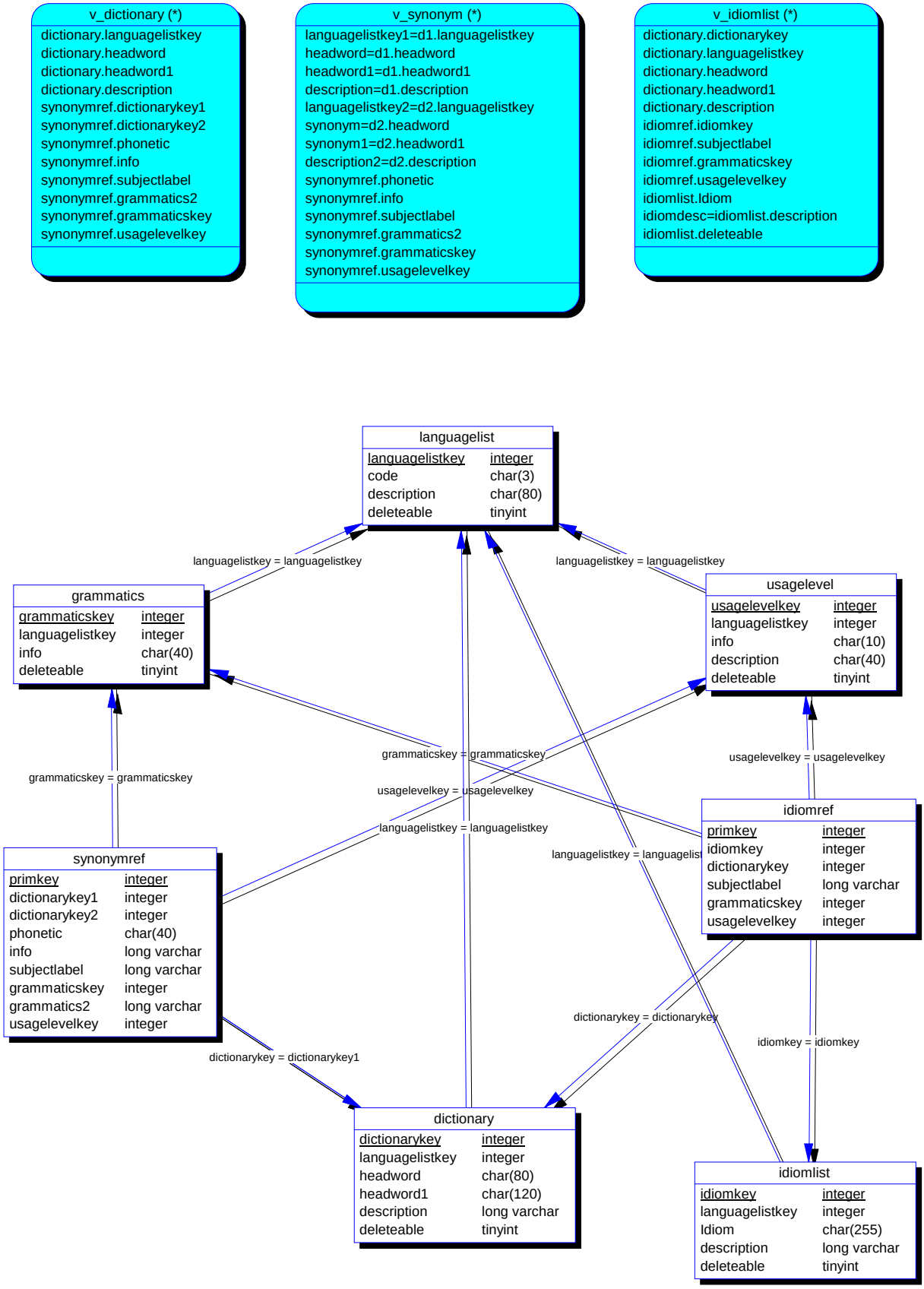


Abbildung 61: Die Struktur der SQL Datenbank

Literaturhinweise

ANSI	American National Standard for Information Systems -- Programming Language -- C, ANSI X3.159-1989
K&R I	Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language, Prentice Hall, 1978, ISBN 0-13-110163-3.
K&R II	Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language, Second Edition, Prentice Hall, 1988, ISBN 0-13-110362-8, 0-13-110370-9.
K&R III	Dennis M. Ritchie. The Development of the C Language . In Second History of Programming Languages conference, Cambridge, Mass., Apr. 1993.
X3J16	The CD, which X3J16/WG21 released as the first "official" baseline for the eventual standard, can be found at the following address: ftp://research.att.com/dist/c++std/WP for PostScript and PDF formats
MFC	Herbert Schildt, MFC programming from the ground up, MacGraw-Hill, 1996, ISBN 0-07-882222-X, 599 Seiten
W31	Charles Petzold, Programmierung unter Windows 3.1, Microsoft Press, 1992, ISBN 3-86063-317-1
OOP I	Ute Claussen. Objektorientiertes Programmieren. Springer Verlag, 1993. ISBN 3-540-55748-2.
OOP II	William Ford and William Topp. Data Structures with C++. Prentice-Hall, Inc., 1996. ISBN 0-02-420971-6.
CPP I	Bjarne Stroustrup. The C++ Programming Language. Addison-Wesley, 2nd edition, 1991. ISBN 0-201-53992-6.