Visual C++.NET

Visual C++.NET Das Buch

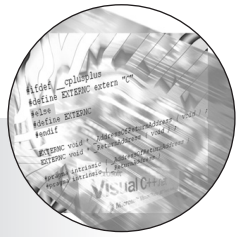
Ron Nanko

Projektmanagement: Simone Schneider
DTP: starke&partner, Willich
Endkontrolle: Mathias Kaiser Redaktionsbüro, Düsseldorf
Umschlaggestaltung: Guido Krüsselsberg, Düsseldorf
Farbreproduktionen: Fischer GmbH, Willich
Belichtung, Druck und buchbinderische Verarbeitung: Media Print GmbH, Paderborn

ISBN 3-8155-0525-9
1. Auflage 2002

Kurzübersicht

Inhaltsverzeichnis



Vorwort

Herzlich Willkommen in der Welt von VC++.net. Viel ist darüber gerätselt worden, wie wohl die Veränderungen am Ende des Entwicklungsprozesses tatsächlich aussehen würden, nachdem die ersten Betas Anfang 2001 die Runde machten.

Das Ergebnis ist recht erfreulich ausgefallen: Die Vorgängerversion wurde in puncto Bedien- und Benutzerfreundlichkeit um ein Vielfaches übertroffen, neue Anwendungs- und Klassenassistenten machen das Programmieren zur wahren Freude.

Das vorliegende Buch beschäftigt sich insbesondere mit diesen Neuerung im Rahmen der MFC-Programmierung unter VC++.net, richtet sich allerdings nicht nur an Umsteiger, sondern insbesondere auch an Neueinsteiger, die zwar über C++-Kenntnisse verfügen, bislang den Schritt in die Entwicklung von Windows-Applikationen jedoch nicht vollzogen haben.

So werden außer Grundkenntnissen in C++ keine weiteren Anforderungen an den Leser gestellt – er wird jedoch nach der Lektüre in der Lage sein, professionelle Windows-Anwendungen zu entwickeln.

Bei der Konzeption des Buchs kam es uns in erster Linie auf eine sinnvolle Gliederung und eine behutsame Einführung in das Wirrwarr der zahllosen Klassen und Methoden an, die dem Neueinsteiger bei seinen ersten Gehversuchen begegnen.

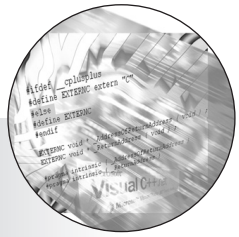
Wert gelegt wurde auch darauf, keine trockenen Programmbeispiele zu präsentieren, sondern stets eine gewisse Experimentierfreude durch die Auswahl der zu entwickelnden Anwendungen zu erzeugen.

Abgerundet wird das Buch schließlich durch ein vergleichsweise umfangreiches Kapitel zur Fehlersuche in Programme – ein Themenkomplex, dem in der restlichen Literatur meist nur wenig Aufmerksamkeit gewidmet wird.

Abschließend möchte ich Simone Schneider und Ute Dick vom SYBEX-Verlag für die gute Zusammenarbeit danken und Ihnen viel Spaß beim Lesen dieses Buches wünschen.

Ron Nanko

Schwackendorf, im März 2002



Installation und Funktionstest

1

Das Studio ist tot, lang lebe das Studio

Dieses erste Kapitel behandelt die Installation und erste Inbetriebnahme des neuen *Visual Studio .NET*. Es werden dabei sämtliche zur Verwendung nötigen Schritte erläutert und anhand von Screenshots, die während einer tatsächlichen Installation angefertigt wurden, illustriert.

Es ist ein ungeschriebenes Gesetz, dass die Installation neuer Produkte immer mit etwas Aufwand und Zeitbedarf verbunden ist – dieser Arbeitseinsatz nimmt mit neuen Programmversionen für gewöhnlich immer um wenigstens ein Viertel zu.

VS.NET – ein Schwergewicht?

Installationsaufwand

Visual Studio .NET bildet hier keine Ausnahme – der gesamte Prozess vom Starten des Setup Tools bis zum ersten Einsatz des Studios kann, je nach Rechner, durchaus bis zu eine Stunde dauern, insbesondere dann, wenn zunächst noch weitere *Windows Components* wie zum Beispiel der *Internet Explorer* aktualisiert werden müssen.

Der Aufwand lohnt sich allerdings. Mit der neuesten Version der Visual-Studio-Reihe hat sich ein würdiger Thronfolger gefunden, der, ganz im neuen XP Look, sämtliche für den Entwickler anfallenden Arbeiten in einem überschaubaren und wohl geordneten Rahmen zur Verfügung stellt. Worum es sich hierbei im Einzelnen handelt, würde allein ganze Bücher füllen, sodass innerhalb dieses Werks nur auf die für den Visual C++ .NET-Entwickler relevanten Sachverhalte eingegangen werden kann.

Die mehrere CDs – oder eventuell eine DVD – umfassende Visual Studio .NET-Programmversion, die also nun auf Ihrem Schreibtisch liegt, sollte Sie nicht verschrecken. Gerade diejenigen Entwickler, die bereits mit dem Vorgänger (Visual Studio 6) gearbeitet haben, werden sich schnell einleben.

Rasche Eingewöhnung dank durchdachten Aufbaus

Aber auch Neueinsteiger, die nach dem Studium der Sprache C++ nun in die visuelle Programmierung unter Windows eintauchen wollen, dürften sich rasch eingewöhnen – immerhin stecken die Erfahrungen vieler tausend Benutzer der älteren Versionen in dem neuen Produkt. Wie so oft gilt auch hier: Übung macht den Meister – und am Ende des Buchs wird jeder in der Lage sein, eigene Windows-Programme zu schreiben und sich, mit diesem Grundwissen ausgestattet, die vielen weiteren Bereiche der Entwicklungswelt zu erschließen.

Vor den Erfolg haben die Götter jedoch die Arbeit gesetzt: die folgenden Seiten beschreiben daher den kompletten Installationsvorgang, der im Wesentlichen zwar intuitiv, aber trotz allem allein wegen seines Umfangs gerade für den Visual Studio-Neuling verwirrend sein mag.

Voraussetzungen für eine erfolgreiche Installation

Im Laufe der verschiedenen *Betas* und *Release Candidates* (Microsofts Testversionen für einen breitangelegten Funktionstest vor Auslieferung des fertigen Studios) schwankten die angegeben Minimalvoraussetzungen für eine erfolgreiche Verwendung von Visual Studio .NET mitunter recht stark.

Voraussetzungen für einen produktiven Einsatz

Mittlerweile haben sich die Wogen geglättet und Microsoft gibt eine Konfiguration ähnlich der Folgenden als sinnvolle Kombination an (sinnvoll heißt hier natürlich: lauffähig, jedoch ist mit Leistungseinschränkungen in der Benutzungsgeschwindigkeit zu rechnen):

- PC mit Pentium II Prozessor, 450 MHz (Pentium III mit 600 MHz wird empfohlen)
- Microsoft Windows 2000 (Server oder Professional), Microsoft Windows XP (Home oder Professional) oder Microsoft Windows NT 4.0 Server. Mit Windows 95 und 98 (gleich welche Version) ist das Visual Studio .NET *nicht* lauffähig!
- Benötigter Speicherausbau:
  - Windows 2000 Professional: 96 MByte (128 MByte empfohlen)
  - Windows 2000 Server: 192 MByte (256 MByte empfohlen)
  - Windows XP Professional: 128 MByte (160 MByte empfohlen)
  - Windows NT 4.0 Server: 96 MByte (128 MByte empfohlen)
- Festplattenplatz:
  - 500 MByte auf dem Systemlaufwerk (das Laufwerk, auf dem auch das Betriebssystem installiert ist, für gewöhnlich also C:)
  - 3 GByte auf dem Installationslaufwerk (das Laufwerk, auf das das Studio installiert werden soll)
- Weiteres: CD-ROM-Laufwerk, Maus oder kompatibles Eingabegerät, VGA-kompatible Grafikkarte und passender Monitor

Hinweis zu den Angaben

Es hat nichts mit übertriebener Vorsicht zu tun, wenn an dieser Stelle gleich vor den eben genannten Hardwarevoraussetzungen gewarnt wird: natürlich werden hier Spezifikationen angegeben, die zwar zum Arbeiten ausreichend sind, jedoch bei einigen Arbeitsschritten sicherlich zu einer unbefriedigenden Performance führen werden. Als Programmierer wird man häufig Programme kompilieren müssen, hier macht sich ein leistungsfähiger Prozessor stark bemerkbar. In Anbetracht von Projekten, die möglicherweise mehrere hunderttausend Zeilen Quelltext enthalten, erscheint ein 600-MHz-Rechner doch eher unterdimensioniert.

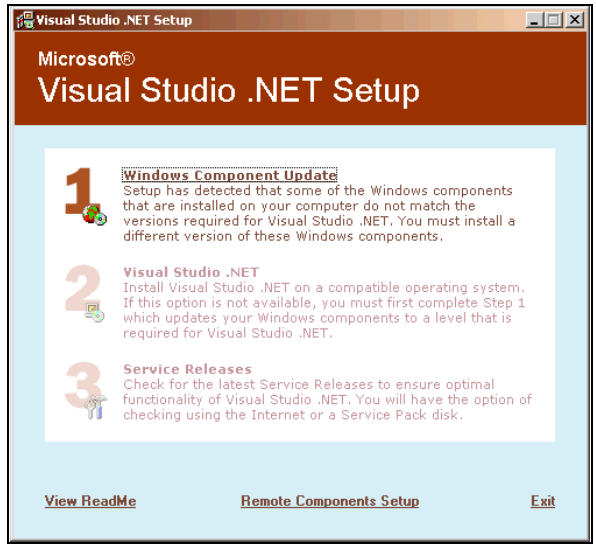
Speicherbedarf des Studios

Das Gleiche gilt für den angegebenen Speicherausbau, 256 MByte sollte der ernsthafte Entwickler durchaus sein eigen nennen, nach oben sind hier keine Grenzen gesetzt.

Starten der Installation

Doch lassen Sie uns nun entspannt am Anfang beginnen – schieben Sie die Installations-CD des Visual Studio .NET in Ihr CD- Laufwerk ein und lassen Sie sich vom Setup-Assistenten begrüßen:

Abb. 1.1
Der Begrüßungsschirm des Visual Studio .NET-Setups



Stufenweise Installation

Das sich öffnende Assistentenfenster enthält die für eine erfolgreiche Installation durchzuführenden Schritte. Genauer gesagt unterteilt sich die Einrichtung des Visual Studio .NET in eine vorbereitende, eine Haupt- und eine nachbereitende Stufe, hier einfach von eins bis drei durchnummeriert.

Die vorbereitende Stufe

Als Schritt 1 tituliert, handelt es sich bei der ersten Stufe salopp gesagt um die Modernisierung eines alt und träge gewordenen Windows-Betriebssystems. Ist dieser Punkt bei Ihnen nicht eingegraut, bedeutet das für Sie, dass auch Ihrem System wichtige Updates fehlen. Wählen Sie also diesen Punkt, um nähere Angaben darüber zu erhalten, welche Maßnahmen zur Behebung dieses Umstands erforderlich sind.

Um eine gerade in Arbeitsgruppen mit verschiedensprachigen Mitarbeitern häufig eintretende Situation zu erläutern, ist der folgende Screenshot aus der

englischen Visual Studio .NET Version herausgegriffen, die unter einem deutschen Windows-Betriebssystem installiert werden soll:

Abb. 1.2
Fehlende Windows-Komponenten und Sprachkonflikte

Fehlende Komponenten

Verschiedene Sprachversionen

Zunächst macht Sie die Meldung darauf aufmerksam, dass die erforderlichen Komponenten nicht in der passenden Sprachversion im Lieferumfang des Visual Studio .NET enthalten sind. Dieser Umstand tritt natürlich ebenso auf, wenn Sie die deutsche Version auf einem englischen Windows-System installieren möchten und so weiter. In jedem Fall benötigen Sie dann die für das Betriebssystem passenden Komponenten, in der jeweiligen Landessprache, zu beziehen über die Microsoft Homepage unter *www.microsoft.com* – wohin Sie der *Download Components*-Button auch prompt führt.

Im Normalfall jedoch dürften Sie die deutsche Visual Studio Version unter einem deutschen Windows-Betriebssystem installieren. In diesem Fall sind die Komponenten auf einer der Visual Studio .NET-CDs bereits enthalten.

In diesem konkreten Fall fehlt ein installiertes *Windows 2000 Service Pack 2* – das System unter dem hier installiert wird, ist Windows 2000 – sowie die neueste Version des Internet Explorers, der, kaum verwunderlich, eng mit dem Studio verzahnt ist.

Installation des neuesten Service Packs

Folgen Sie den Anweisungen auf dem Bildschirm, was in diesem Fall zunächst nichts anderes heißt, als das aktuellste Windows 2000 Service Pack nachzuinstallieren. Im Rahmen dieses Updates werden zahlreiche Sicherheitslöcher Ihres Systems gestopft und einige Komponenten aktualisiert, sowohl was Bugs als auch die Leistungsfähigkeit von Treibern und so weiter anbetrifft.

Vorsicht beim Einspielen neuer Service Packs

Es ist sicherlich angebracht, vor der Installation des Service Packs sämtliche relevanten Benutzerdaten zu sichern, da doch erhebliche Eingriffe in die Windows-Interna durchgeführt werden – genaugenommen werden nämlich sämtliche auszutauschenden Komponenten lediglich mit ihrer neuen Version überschrieben, was dann auch die Größe der einzelnen Service Packs erklärt. Der Grund dafür ist einfach: Sie können sich selbst neue Windows-Installations-CDs mit den jeweils aktuellsten Service Packs vorbereiten, in dem Sie einfach die ursprünglichen Dateien mit denen der Updates austauschen. Damit haben Sie bei einer Neuinstallation immer sofort die aktuellste Version vorrätig und müssen nicht erst noch weitere Updates einspielen, was einen nicht unerheblichen Zeitvorteil bringt.

Ist das Service Pack erfolgreich installiert, können Sie erneut den ersten Schritt des Installationsprozesses anwählen und erhalten in diesem konkreten Fall dann folgende Mitteilung:

Abb. 1.3
Das Service Pack ist installiert, doch der Explorer ist noch nicht up-to-date

Wieder erhalten Sie die Meldung, dass die Komponenten nicht in der zum Betriebssystem passenden Sprache vorhanden sind. (Zur Erinnerung: Dieses Problem tritt nur bei der Vermischung von verschiedenen Sprachversionen bezüglich Betriebssystem und Visual Studio .NET auf.)

Optionale Komponenten mit freier Sprachwahl

In diesem Fall jedoch handelt es sich um eine so genannte *optionale Komponente*, die entsprechend nicht notwendigerweise dieselbe Sprache wie der Betriebssystemkern haben muss.

Sie können nun selbst entscheiden, ob Sie die den Visual Studio .NET CDs beiliegenden Komponentenversionen nutzen möchten, oder doch lieber selbst eine Aktualisierung durchführen, beispielsweise durch das Herunterladen der Komponente in der Ihnen genehmen Sprache.

Herunterladen neuer Komponenten

Letzteres geschieht durch Anwählen des Punkts *Download Components*, der Sie direkt auf eine passende Downloadseite – selbstverständlich im Rahmen der Microsoft Homepage – führt.

Wollen Sie hingegen die fremdsprachige Version verwenden, reicht ein Klick auf die *Continue*-Schaltfläche aus.

Lizenzvereinbarungen

Im Laufe der Visual Studio .NET-Installation werden Sie mehr als einmal auf die wohl bekannten Lizenzvereinbarungen zwischen Ihnen und Microsoft stoßen, die allesamt akzeptiert werden müssen, um den Einrichtungsvorgang fortsetzen zu können.

Annehmen der Lizenzvereinbarung

In diesem Fall geht es um das Zustimmen zu den Vereinbarungspunkten bezüglich der zu installierenden Windows-Komponenten. Lesen Sie sich den Text durch und bestätigen Sie, dass Sie die Vorschriften akzeptieren durch Anwahl des Punkts *I accept the agreement* durch anschließenden Klick auf die Schaltfläche *Continue*.

Eine Anwahl von *I do not accept the agreement* beendet die Installation vorzeitig.

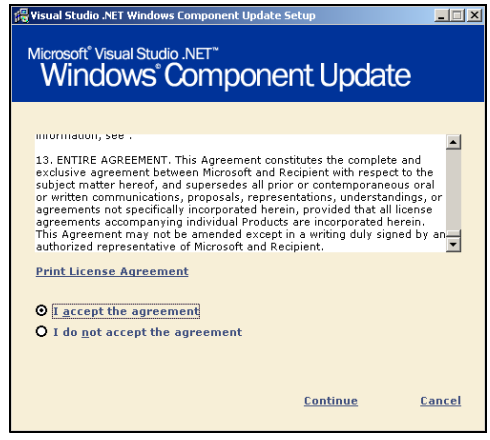
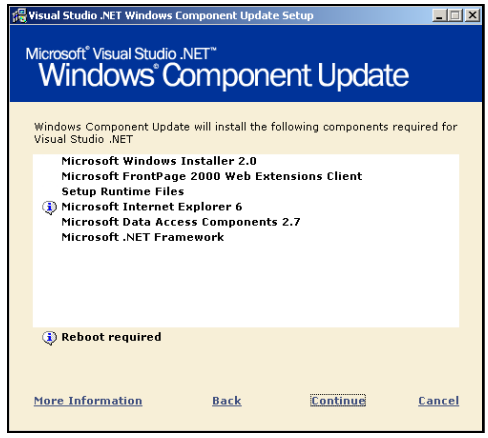


Abb. 1.4
Die Lizenzvereinbarung

Übersicht über den Installationsprozess

Der eigentliche Installationsprozess für die zu aktualisierenden Windows-Komponenten wird nach der Annahme der Lizenzvereinbarungen Microsoft-typisch noch nicht gestartet, sondern zunächst in einer Art kurzen Übersicht zusammengefasst:

Abb. 1.5
Die zu installierenden Komponenten



Die Bedeutung der einzelnen Komponenten

Im Folgenden soll kurz dargestellt werden, wofür die einzelnen Komponenten dienen, damit Sie nicht blindlings Software in Ihr System schaufeln müssen.

- Microsoft Windows Installer 2.0: diese Komponente dient zum Erzeugen benutzerdefinierter Installationen für Ihre selbst geschriebenen Programme und liegt nun in der neuen Version 2.0 vor. Die wesentlichen Unterschiede zur Vorgängerversion liegen in der Unterstützung von Windows XP und 64-Bit-Applikationen. Weiterhin ist es nun möglich, Systemeigenschaften direkt abzufragen, beispielsweise ob das Setup-Programm unter einem XP-Betriebssystem gestartet wurde und so weiter. Eine Reihe von Bugfixes und weiteren nützlichen Erweiterungen runden die neue Version ab.

- *Microsoft FrontPage 2000 Web Extensions Client*: Eine FrontPage 2000 Erweiterung, die allerdings demnächst durch die neue FrontPage 2002 Version überflüssig werden dürfte.

- *Microsoft Data Access Components 2.7 (kurz MDAC)*: Die MDAC bilden den Kern von Microsofts UDA (*Universal Data Access*) Strategie, die es erlaubt, beliebige Informationen aus beliebigen Datenquellen in eigenen Applikationen zugriffsbereit und verarbeitbar zu machen. Die MDAC stellen sozusagen die Treibersoftware zwischen der Applikation und der externen Datenquelle dar.

- *Microsoft Internet Explorer 6*: Der jüngste Spross der Explorerfamilie, der allgemein bekannt sein dürfte und keiner weiteren Erläuterung bedarf.
- *Microsoft .NET Framework*: Der Kern des gesamten .NET Frameworks besteht „nur" aus einer Reihe von neuen Windows-Komponenten, die es allerdings in sich haben. Eine genauere Beschreibung wird sich im weiteren Verlauf dieses Buchs befinden.

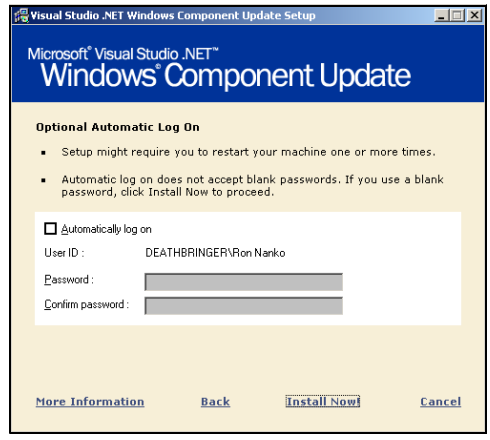


Abb. 1.6
Möglichkeit zum automatischen Login

Automatischer Login

Beim Betrachten der Liste der zu installierenden Komponenten ist Ihnen vielleicht aufgefallen, dass neben dem Internet Explorer ein Hinweissymbol zu sehen war, dessen Erläuterung von einem erneuten Bootvorgang spricht. Je nachdem, welche Komponenten auf Ihrem System noch installiert werden müssen, könnten es durchaus eine ganze Reihe von Schritten sein, die einen neuen Bootvorgang bedingen.

Wie schon angesprochen, braucht die Installation von Visual Studio.NET seine Zeit, umso angenehmer fällt da das Fenster auf, das sich zeigt, sobald Sie die Informationsseite mit *Continue* verlassen haben.

Zeit sparen bei der Installation

In dem erscheinenden Dialog finden Sie eine Maske wieder, die Sie mit Ihren Login-Informationen füllen können. Das Setup trägt dann dafür Sorge, dass erforderliche Bootvorgänge voll automatisch ablaufen und Sie in der Zwischenzeit Kaffee trinken gehen können, während der Rechner seine Arbeit erledigt – vergessen Sie aber nicht, auch den Erlaubnisschalter *Automatically Log On* zu aktivieren, wenn Sie von dieser Möglichkeit Gebrauch machen wollen.

Ein Klick auf *Install Now* startet schließlich den Vorgang.

Sicherheitsbedenken beim automatischen Login

Es ist natürlich klar, dass es nicht umsonst die Möglichkeit gibt, den automatischen Login zu unterbinden, denn schließlich handelt es sich hierbei um eine durchaus praktische Idee, die davon zeugt, dass auch die Microsoft Entwickler hinreichend häufig die Installation der Komponenten über sich ergehen lassen mussten.

Im Normalfall gibt es keinen Grund, der gegen ein Ausnutzen dieser Vereinfachung spricht, bedenken Sie jedoch, dass Sie auch dann noch einmal neu eingeloggt werden, wenn der Installationsprozess abgeschlossen wurde – nach dem finalen Reboot, sozusagen.

Haben Sie tatsächlich die Zeit während der Installation zu einer Kaffeepause oder Ähnlichem genutzt und steht Ihr Rechner in einem nicht verschlossenen Büro, ist er nun ungeschützt und jedermann zugänglich! Unter solchen Voraussetzungen bietet es sich an, den automatischen Login zu untersagen und dann lieber die etwas längere Installationszeit – bedingt durch erforderliche manuelle Eingaben – in Kauf zu nehmen.

Der Installationsverlauf

Der Installationsprozess wird nun durch eine Reihe von visuellen Markierungen (Fortschrittsbalken, farbige Pfeile und Häkchen) dargestellt und dokumentiert:

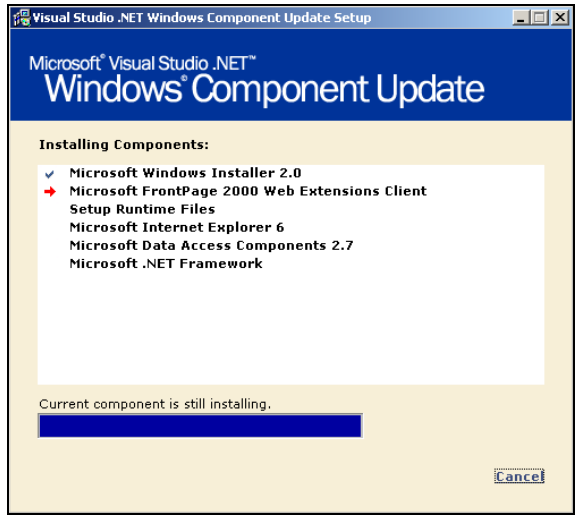


Abb. 1.7
Der laufende Installationsvorgang für die Komponenten

Abschluss der Komponenteninstallation

Rückmeldung des Installationsprogramms

Nach einer Reihe von Neustarts – abhängig von den nachzuinstallierenden Komponenten – wird sich das Setup-Programm wieder bei Ihnen melden und Auskunft darüber geben, ob Probleme bei der Installation aufgetreten sind.

Dieses sollte normalerweise nicht der Fall sein und deutet normalerweise auf einen schwerwiegenden Fehler hin. Allgemein gilt, dass der Setup-Prozess des Visual Studio .NET im Laufe der von Microsoft verteilten Test- und Zwischenversionen immer ausgefeilter wurde und Probleme meist immer mit vorhandenen früheren Testversionen zusammenhingen.

Stellen Sie bei Schwierigkeiten mit der Installation sicher, dass keine Altlasten aus dem Betastadium des Visual Studio .NET mehr vorhanden sind oder entfernen Sie diese gegebenenfalls. Prüfen Sie weiterhin, ob auf dem Systemlaufwerk – das auch das Betriebssystem enthält – hinreichend Speicherplatz vorhanden ist.

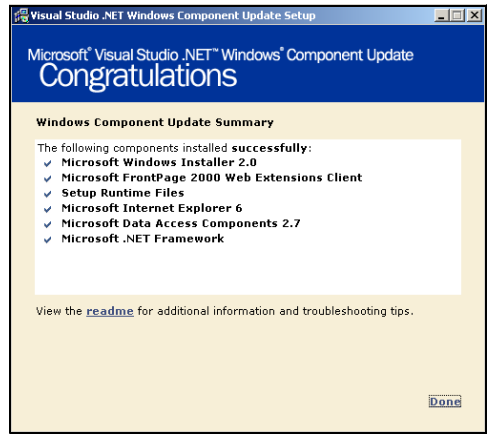


Abb. 1.8
Die erfolgte und erfolgreiche Installation wird gemeldet

Die zweite Runde – Installation des eigentlichen Studios

Bislang haben wir nur vorbereitende Maßnahmen getroffen, die zur Inbetriebnahme des Visual Studio .NET notwendig, aber nicht hinreichend sind. Wenn Sie an dieser Stelle angekommen sind, haben Sie insbesondere erst die Windows-Komponenten auf den aktuellsten Stand gebracht und dem .NET Framework Einzug in Ihr System gewährt.

Fortsetzen der Installation

Jetzt kommt es darauf an, ein Werkzeug einzurichten, mit dem auf dieses Framework in einer effektiven und effizienten Weise zugegriffen werden kann.

Nach der erfolgten Installation der Windows-Komponenten finden Sie sich zunächst im Hauptauswahlbildschirm des Setup-Assistenten wieder:

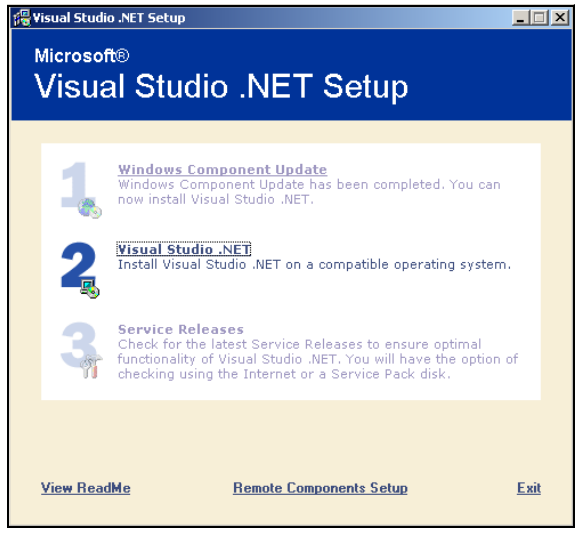


Abb. 1.9
Der Installationsassistent wartet auf die Ausführung des zweiten Schritts

Starten des Hauptinstallationsvorgangs

Status der Installation

Wie Sie sehen können, ist der erste Punkt mittlerweile ausgegraut, was darauf hindeutet, dass Sie diese erste Hürde erfolgreich gemeistert haben. Es ist nun erforderlich, den zweiten Punkt auszuwählen und dort das eigentliche Visual Studio .NET zu installieren.

Es präsentiert sich ein weiteres Fenster, das in drei Reiter aufgeteilt ist, die allerdings nicht frei auswählbar sind, sondern der Reihe nach komplett abgearbeitet werden müssen. „Abarbeiten“ heißt hierbei nichts anderes, als Ihre Benutzerinformationen einzutragen und weiterführende Angaben zu machen, die für die erfolgreiche Installation notwendig sind.

Weitere Lizenzvereinbarungen

Sie treffen nun auf die zweite Lizenzvereinbarung, die sich dieses Mal auf das Visual Studio .NET selbst bezieht. Auch hier besteht die Wahl wiederum darin, die Vereinbarungen zu akzeptieren oder abzulehnen. Während ein Ablehnen zum Beenden des Setup-Vorgangs führt, erlaubt Ihnen das Annehmen das Fortfahren des Vorgangs.

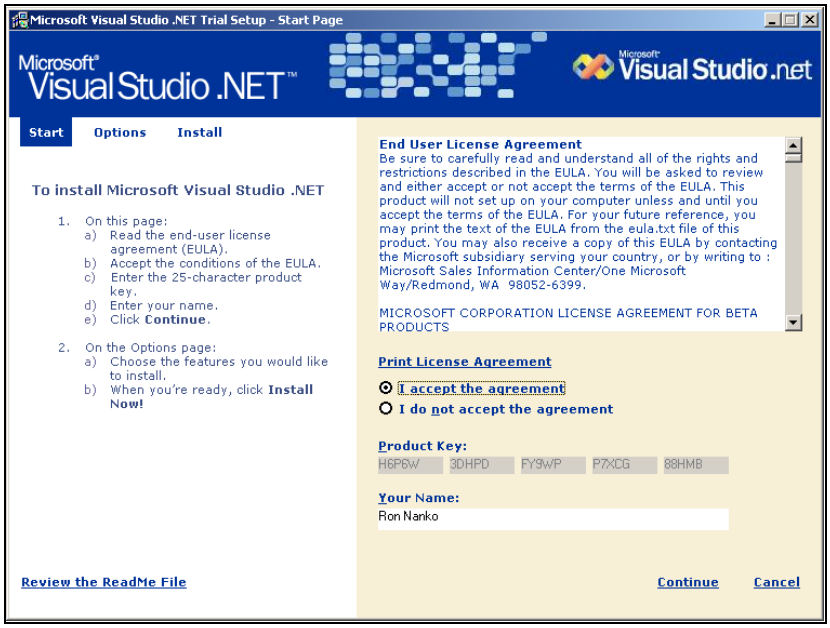


Abb. 1.10
Die erste Seite der Hauptinstallation

Neben dem Lizenzvertrag sind noch Ihr Name sowie die Produkt ID Ihrer Visual Studio .NET Version in die dafür vorgesehenen Felder einzutragen.

Angeben der Produkt ID

Die .NET Trial Version

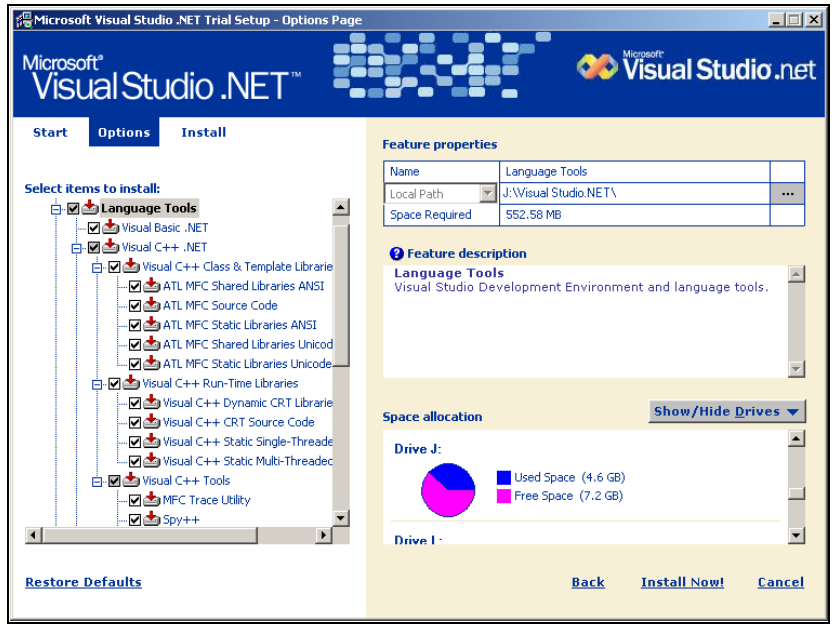
Haben Sie noch eine Trial Version des Studios vorliegen, sind die Daten für die Produkt ID bereits fest gesetzt.

Bedenken Sie jedoch, dass die von Microsoft im Vorfeld herausgegebene Trial Version Ende Februar 2002 Ihre Gültigkeit verloren hat, jetzt also nicht mehr lauffähig ist.

Die Optionsseite

Nach einem Anklicken von *Continue* gelangen Sie auf die zweite Seite des Installationsdialogs:

Abb. 1.11
Die Optionsseite



Auswählen von Laufwerken und Verzeichnissen zur Installation

Eigentlich ist der Name *Optionsseite* etwas hochtrabend für die möglichen Einstellungen, in Wirklichkeit können hier nur der Installationsumfang und die zu verwendenden Installationspfade eingetragen werden.

Visual Studio .NET braucht ca. 300 MByte an Plattenplatz auf Ihrem Systemlaufwerk (in der Regel C:) sowie bei voller Installation weitere 3 GByte, die mehr oder weniger frei verteilt werden können – einzelne Programmpakete sind dabei immer komplett auf einem Laufwerk zu installieren.

Die installierbaren Komponenten

Im linken Teil des Fensters sehen Sie die zur Verfügung stehenden Optionen, das Anklicken der Plus-Symbole erweitert den Darstellungsbaum an der betreffenden Stelle, eine Anwahl einer Minus-Symbols zieht die untergeordneten Punkte wieder zusammen.

Festlegen der zu installierenden Komponenten

Auf diese Weise können Sie jeden Punkt des Baums bequem erreichen und einzeln an- oder abwählen. Nur die angewählten Komponenten werden im Folgenden installiert, die übrigen sind aber auch nachträglich über den Assistenten noch hinzufügbar.

Sobald Sie ein Element des Baums auswählen, werden rechts oben Informationen über den benötigten Platzbedarf und das derzeitig eingestellte Installationsverzeichnis ausgegeben, das Sie beliebig anpassen dürfen. Dabei gelten Einstellungen von höher eingeordneten Komponenten auch automatisch für die untergeordneten – stellen Sie also beispielsweise bei *Language Tools* als

Verzeichnis *C:\MeinVerzeichnis* ein, werden auch die untergeordneten Komponenten *Visual Basic. NET*, *Visual C++ .NET* und so weiter unter diesem Verzeichnis – möglicherweise in eigenen Unterverzeichnissen – abgelegt.

Eine kurze Beschreibung der installierbaren Komponenten

Um sich entscheiden zu können, welche Komponenten installiert werden sollten, müssen Sie wissen, womit Sie es im Einzelnen zu tun haben. Eine Übersicht über die installierbaren Komponenten ist daher im Folgenden abgedruckt.

Visual Basic .NET

Visuelle Programmierung mit Visual Basic

Visual Basic .NET: Den größten Wandel bei der Umstellung auf .NET dürfte Visual Basic durchgemacht haben. In der Version 6 noch verlacht als Einsteigersprache, hat sich Visual Basic nun zu einer interessanten Möglichkeit der Programmentwicklung unter Windows gemausert. Die gravierendste Neuerung ist mit Sicherheit die nun tatsächlich durchgezogene objektorientierte Struktur der Sprache, insbesondere bestehen nun alle bekannten Möglichkeiten wie Vererbung und so weiter. Es ist jetzt unter anderem mit VB.NET möglich, verteilte Anwendungen, *Webservices*, *WebForms* und *WinForms* zu erzeugen. (Im weiteren Verlauf werden diese Begriffe noch näher erläutert.)

Visual C++.NET

Visual C++ – alter Bekannter im neuen Gewand

Visual C++.NET: C++ galt immer als das Schlachtross unter den Programmiersprachen, da es durch seine vielfältigen Einsatzmöglichkeiten zwar zu hochoptimierter Programmierung einlud und performante Applikationen erlaubte, aber bei ungünstiger Wahl der Programmkonstrukte leicht Flaschenhälse an kritischen Programmstellen einstreuen konnte. Mit der jetzt vorliegenden Version hat sich an dieser Situation zunächst einmal nur so viel geändert, dass Microsoft die neue Generation als noch leistungsfähiger und noch flexibler darstellt. Zu den neuen Features gehören *Managed Extensions*, *Attribut-Programmierung* und *Webapplikationsprogrammierung*, das Ganze verpackt in einem neuen Kleid, der neuen IDE des Visual Studio .NET.

Ergänzungen für Visual C++

- *Visual C++-Klassen & Template Bibliotheken*: Wie es der Name schon andeutet, verbergen sich hinter diesem Punkt eine Reihe wichtiger Bibliotheken, nämlich genau denen, die für die Entwicklung von Webservices, MFC-Applikationen und der Entwicklung von COM-Komponenten erforderlich sind.

- Visual C++ Laufzeitbibliotheken: die einzelnen Laufzeitbibliotheken (common run-time libraries, kurz CRT) von Visual C++. Im Einzelnen sind dieses die static single-threaded CRT-Bibliothek, die static multi-threaded CRT-Bibliothek, die dynamische CRT-Bibliothek und der CRT Source Code.

Visual C++ Tools

Visual C++ Tools: Seit jeher sind im Studio eine Reihe von nützlichen Tools enthalten, die in der neuen Version gründlich überholt, aufpoliert und verbessert wurden.

ActiveX-Steuerelemente

- *ActiveX Control Test Container*: Bei der Entwicklung von ActiveX-Kontrollen kommt es insbesondere auf das Testen und Debuggen derselben an. Hierbei unterstützt Sie der ActiveX Control Test Container, der hilfreiche Informationen während Fehlersuche in den Kontrollen bereitstellt.

- *Error Lookup Tool*: Ein unerlässliches Tool. Compiler, Linker und andere Visual Studio Komponenten kommunizieren miteinander und mit dem Benutzer durch Angabe von Zahlencodes, sobald Fehler auftreten. Diese sind in einer Datenbank abgelegt und mit einem Prosa-Text versehen, der eine genauere Beschreibung des Fehlers enthält, insbesondere also einen, der vom Menschen gelesen und klar verstanden werden kann. Das *Error Lookup Tool* dient nun dazu, diese Informationen automatisch auszulesen und zu präsentieren, sobald eine Fehlermeldung auftritt.

Debugging

- *MFC Trace Utility*: *Trace Flags* (Nachverfolgungs-Flags) können in der Datei *AFX.INI* gesetzt oder gelöscht werden, um anzugeben, welche Informationen während einer Debug-Sitzung an den Programmierer ausgegeben werden sollen. Um die Zahl der fehlerhaften Eintragungen in dieser Datei zu minimieren, dient das MFC Trace Utility, das einen Zugriff auf die einzelnen Flags bietet.

- *OLE/COM Object Viewer*: Zeigt die Schnittstellen von COM-Objekten an. Dazu gehören sowohl Registrierungseintragungen als auch Implementations- und Aktivierungsdetails.

Fensterüberwachung

- *Spy++*: Dient zum Überwachen von Fenstern. Sie können mit diesem Tool ein (oder mehrere) Fenster selektieren, die dann auf eingehende Nachrichten überprüft werden. Die folgenden Nachrichtentypen können überwacht werden: DDE, Zwischenablage, Maus, Tastatur, Buttons, Non-Client Bereich, Kombo-Kontrollen, Edit-Kontrollen, Listbox-Kontrollen, Statische Kontrollen, applikationsspezifische Mitteilungen, andere Mitteilung, die nicht in eine der eben genannten Kategorien passen.

- *SQL Debug Tool*: Dient zur Fehlersuche in SQL-Server-Applikationen.

- *WebDebug Tool*: Dient zur Fehlersuche in Webserver- und -client-Anwendungen

Visual C#.NET

C# – Die neue Programmiersprache

Visual C#.NET: Ein junger Spross der Sprachenfamilie ist C# (sprich: c sharp). Microsoft selbst bezeichnet diese Sprache als „logische Konsequenz aus der

Entwicklung von C, C++ und Java“. Kurz gesagt, erfüllt C# alle Forderungen, die man an eine moderne Programmiersprache stellen würde und ist komplett auf die Entwicklung von Anwendungen auf das .NET Framework ausgerichtet.

Weitere Komponenten

- *.NET Framework SDK*: Die Dokumentation für Entwickler, die mit dem .NET Framework arbeiten.
- *Crystal Reports*: Dient zum Einfügen von grafischen Repräsentationen von Daten in Ihre Anwendungen. Als Schlagwort sei hier interaktiver Inhalt genannt, also insbesondere Anschauungsmaterial, das für Präsentationen aufbereitet werden soll.
- *Tools zur Verteilung von Applikationen*: Enthält Module zum Aufbereiten von Applikationen zur Weitergabe sowie eine frei verwendbare Bibliothek mit Grafiken, Videos, Cursors und Icons.
- *Server Components*: Wenn Sie planen, Web-Applikationen zu entwickeln und Ihren eigenen Rechner sowohl als Client als auch als Server zu verwenden, müssen die Server-Komponenten installiert werden.
- *Remote Debugger*: Es ist bei zahlreichen Applikationen praktisch, wenn der Vorgang der Fehlersuche nicht direkt auf der Maschine durchgeführt wird, auf der auch die Anwendung läuft. Als Beispiel seien hier Vollbildapplikationen wie zum Beispiel Spiele genannt. Der Remote Debugger erlaubt es Ihnen, einen zweiten Rechner, der dann via Netzwerk mit dem Computer, der das Programm ausführt, verbunden sein muss, als Debug-Plattform zu benutzen.
- *Web Development*: Zur Erzeugung von ASP.NET-Web-Applikationen, müssen Sie diese Module installieren. Voraussetzung ist weiterhin das Vorhandensein von Internet Information Services (IIS) sowie FrontPage Server Extensions (FPSE).
- *SQL Server Desktop Engine*: Hier hinter verbirgt sich die *Microsoft Server Desktop Engine* (kurz *MSDE*), die einen SQL-Server darstellt. Erforderlich, falls Sie keinen anderen SQL-Server zur Verfügung haben. MSDE unterstützt weiterhin XML.
- Dokumentationen: Die komplette Dokumentation, inklusive Referenzen, Schritt-für-Schritt-Anleitungen und Beispielen rund um Visual Studio .NET und das .NET Framework.

Online-Hilfe

Abschließen der Installation

Sobald Sie sich für alle zu installierenden Komponenten entschieden (wenn Sie nicht sicher sind, wählen Sie, genügend Platz vorausgesetzt, ruhig alle Kompo-

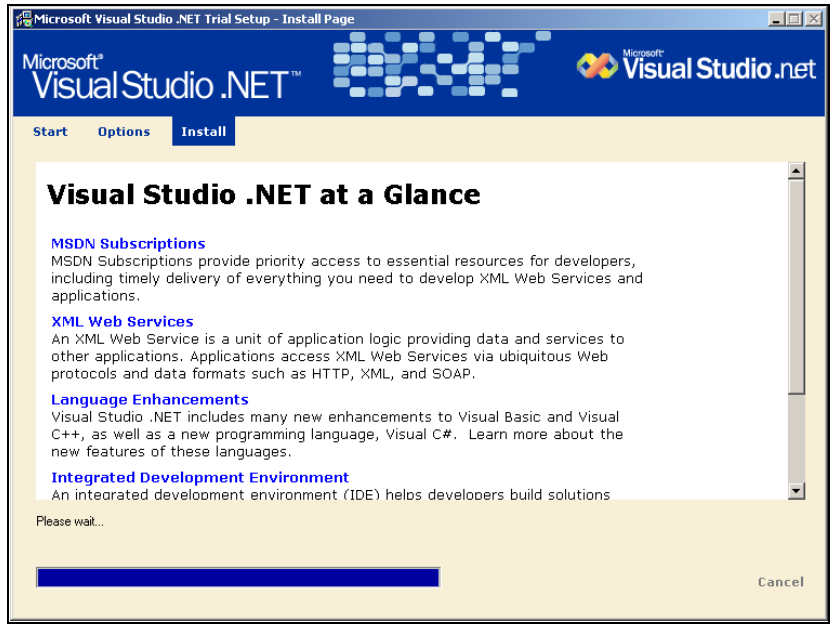
nenten an) und die gewünschten Verzeichnisse ausgewählt haben, können Sie den eigentlichen Installationsvorgang durch einen Klick auf die Schaltfläche *Install Now* starten.

Beobachten des Installationsvorgangs

Es präsentiert sich ein weiteres Fenster, das den Installationsverlauf visualisiert. Nach dem Erzeugen eines Setup-Skripts werden dann zunächst eine ganze Reihe von Registrierungsschlüsseln angelegt und Dateien in die spezifizierten Verzeichnisse kopiert. Dieser Vorgang kann je nach Plattengeschwindigkeit und ausgewählten Komponenten durchaus eine halbe Stunde und länger dauern.

Reichlich Zeit also, um einen Blick auf die im Fenster angebotenen weiterführenden Informationen zu werfen. Sie sollten diesen Service durchaus nutzen, da es hier allerlei Interessantes zu erfahren gibt, beispielsweise wie man an eine *MSDN Subscription* (also ein Abonnement der aktuellsten Referenzen, Dokumentationen und Artikel aus der Entwicklergemeinde) beziehen kann oder was es an Neuerungen in der integrierten Entwicklungsumgebung (kurz IDE des Visual Studio .NET) gibt.

Abb. 1.12
Der Bildschirm während der Hauptinstallation



Zusammenfassung des Installationsergebnisses

Nach der erfolgten Installation erwartet Sie ein Fenster ähnlich dem folgenden:

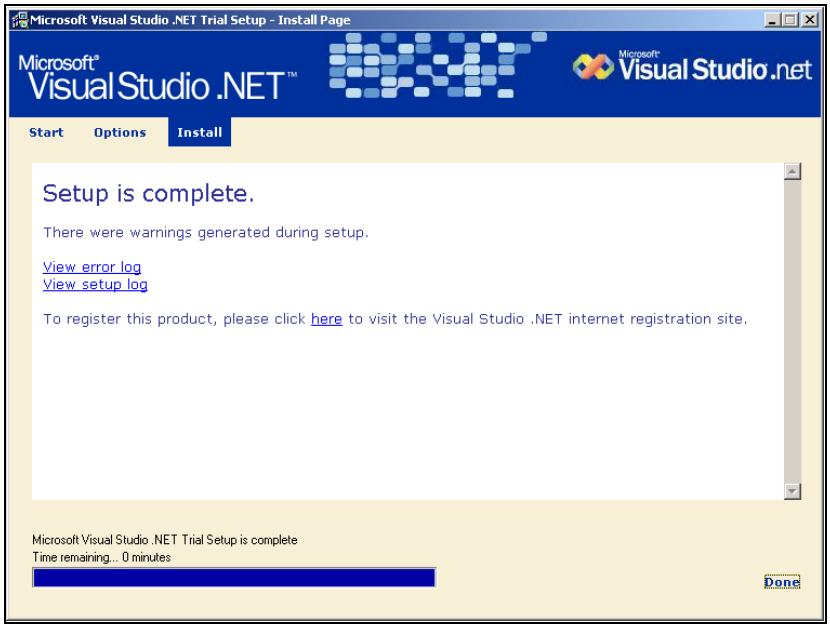


Abb. 1.13
Das Ergebnis einer fertigen Visual Studio .NET-Installation

In diesem Fall teilt das Setup-Programm mit, dass Fehler – oder besser: Warnungen – während der Installation aufgetreten sind. Das sollte im Normalfall nicht vorkommen und dient an dieser Stelle nur zur Demonstration. Sollten bei Ihnen Fehler aufgetreten sein, können Sie dann an dieser Stelle einen Blick in den Installations-Log werfen und sehen, was schief gegangen ist.

Fehler bei der Installation

Können Sie den Fehler dann aus irgendeinem Grund nicht selbstständig beheben – zum Beispiel, weil eine Datei auf CD defekt ist – wenden Sie sich an den Microsoft Support, der sich in diesem Fall eingehend mit Ihrem Problem beschäftigen kann und wird.

Phase 3 – Alles auf den neuesten Stand bringen

Nach erfolgreicher Installation präsentiert sich erneut der Startbildschirm:

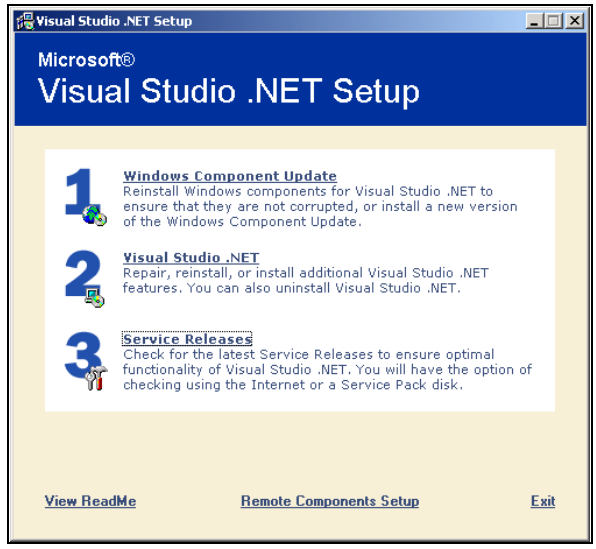


Abb. 1.14
So sieht der Startbildschirm im dritten Anlauf aus

Verwirrenderweise sind nun alle drei Punkte anwählbar, doch sollte Sie dieser Umstand nicht irritieren: Sobald der dritte Punkt aktivierbar ist, haben Sie während der Installation alles richtig gemacht und sind nun in der Lage, die Windows-Komponenten erneut zu installieren, falls Sie durch einen Datenverlust (zum Beispiel nach einem Virenbefall) korrumpiert wurden).

Aktualisierung einer Installation

Interessanter ist an dieser Stelle jetzt der Punkt 3, der es erlaubt, die gerade abgeschlossene Installation auf den neuesten Stand zu bringen. Dieses dient dem Umstand, dass es im Laufe der Zeit Fehlerkorrekturen geben kann – und es ist davon auszugehen, dass Visual Studio .NET (das ja auch als Visual Studio 7 bezeichnet wird) durchaus einige Jahre als Ihre Hauptentwicklungsplattform dienen könnte, wie es eventuell schon die Vorgängerversion getan hat.

Innerhalb dieser Zeit werden von Microsoft immer wieder Verbesserungen und Bugfixes nachgeliefert werden, die über diesen Punkt 3 zugriffsbereit gemacht werden. Wählen Sie daher jetzt diesen letzten Schritt durch einfaches Anklicken aus.

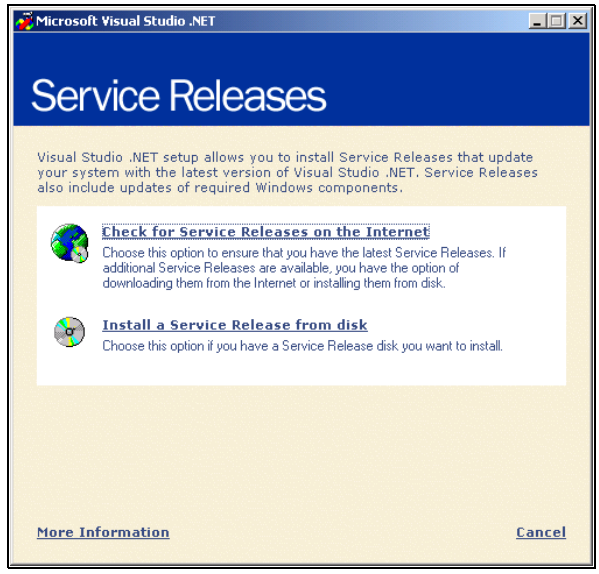


Abb. 1.15
Die Möglichkeiten zum Aktualisieren der Installation

Haben Sie eine Update-CD vorliegen, können Sie hier den Punkt *Installieren von einer Service CD* wählen, ansonsten prüfen Sie, ob im Internet eine neue Version vorliegt, indem Sie den Punkt *Prüfen auf neue Versionen im Internet* auswählen.

Sollte einer der beiden Punkte zutreffen, führen Sie die Anweisungen aus, die auf dem Bildschirm ausgegeben werden, wodurch Ihre Version, falls nötig, auf den aktuellen Stand gebracht werden kann.

Herzlichen Glückwunsch, Sie haben die Installationshürden erfolgreich gemeistert! Im Folgenden soll es um einen kleinen Funktionstest des gerade installierten Produkts gehen.

Die erste Hürde ist genommen

Test der Funktionalität

Nachdem Visual Studio .NET nun installiert ist, wollen Sie es sicherlich auch gleich einmal austesten. Nun, im Laufe des Buchs werden Sie mit vielen Feinheiten der Entwicklungsumgebung vertraut gemacht, sodass sich die restlichen Seiten dieses Kapitels nur mit dem ersten Starten des Studios beschäftigen sollen.

Starten des neuen Studios

Beginnen Sie damit, das neue Studio zu starten, Sie werden es in Ihrem Startmenü finden. Es öffnet sich eine Startseite, die Ihnen in dieser Form noch nicht bekannt sein dürfte.

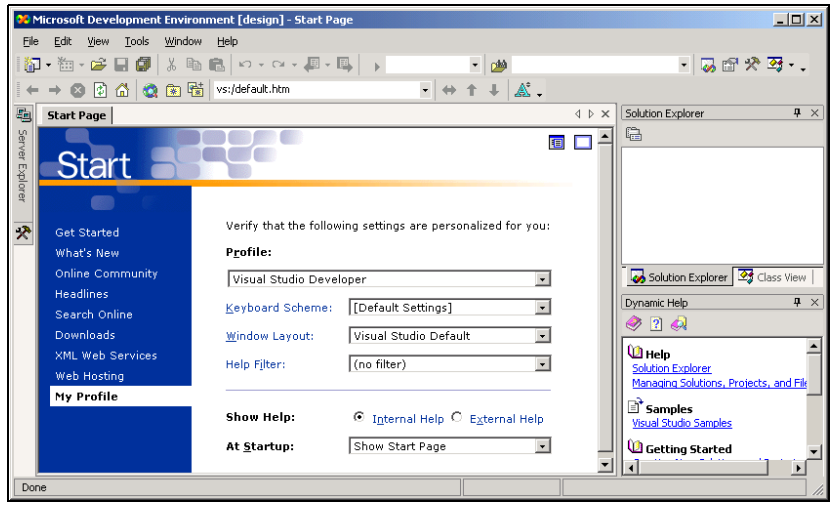


Abb. 1.16
Die unangetastete Startseite

Einrichten eines Profils

Hier bietet sich die Möglichkeiten, Ihr Profil als Entwickler einzurichten. Unter Profil versteht das Studio ihr Haupteinsatzgebiet, also ob Sie beispielsweise C++-Programmierer, Visual Basic-Entwickler oder vielleicht ein Allround-Talent sind, das mit allen Sprachen gleichstark arbeiten möchte.

Sie können entweder aus einer Reihe von vorgefertigten Profilen wählen, oder Ihre eigenen Einstellungen nach Lust und Laune festsetzen – dabei enthält jeder Punkt eine Reihe von Einstellmöglichkeiten.

Die einzelnen Punkte im Startfenster

Die Punkte im Einzelnen:

- *Tastatur Layout*: Hier tragen Sie ein, welche Shortcuts auf der Tastatur wie belegt sein sollen.
- *Fenster Layout*: Gibt an, wie die Fenster und Toolbars innerhalb der IDE angeordnet werden sollen.
- *Hilfe Filter*: Spezifiziert, welche Hilfethemen bei einer Suche innerhalb der Dokumentationen automatisch ausgeblendet werden sollen. Geben Sie einen Visual C++-Filter an, werden nur Themen aufgelistet, die mit Visual C++ in einem Zusammenhang stehen, nicht etwa solche, die sich mit einer Visual Basic-Spezialität beschäftigen.
- *Hilfe anzeigen*: Hier können Sie wählen, ob Hilfstexte intern (also innerhalb der IDE) oder extern (in einem eigenen Fenster) angezeigt werden sollen. Dieses ist sicherlich Geschmackssache, wer einen aufgeräumten Desktop mag, wählt die interne Variante.

- *Beim Start*: Gibt an, was Sie beim Starten des Studios auf dem Bildschirm sehen möchten. Voreingestellt ist *Startseite*, dies ist die Seite, die Sie gerade betrachten.

Sinnvolle Einstellungen für das Buch

Für dieses Buch bietet es sich an, wenn Sie das voreingestellte *Profil Visual C++-Entwickler auswählen, das die drei erstgenannten Punkte allesamt auf Visual C++ setzt. So ist sichergestellt, dass Sie keine Diskrepanzen zu den im Buch beschriebenen Fensteranordnungen haben und womöglich wichtige Fenster erst lange suchen müssen, die standardmäßig nicht dargestellt werden.*

Die neu eingestellte Startseite sieht nun so aus:

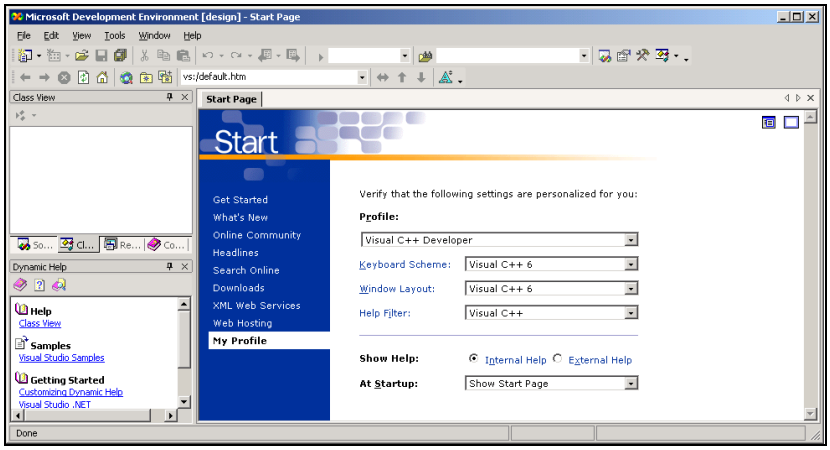


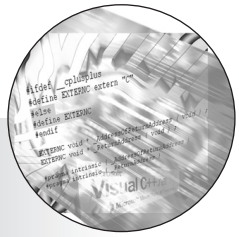
Abb. 1.17
Die Startseite wurde geeignet initialisiert

Wie Sie sehen, gibt es auf der Startseite noch eine ganze Reihe weiterer Punkte, die hier nicht näher erläutert werden sollen und im Wesentlichen zusätzliche Hilfestellungen zur Verwendung des Studios, Download-Links, News zum VS.NET und so weiter anbieten.

Es sei dem Leser überlassen, an dieser Stelle frei in den einzelnen Unterpunkten herumzustöbern.

Ausblick

Im Laufe des nächsten Kapitels werden Sie eine erste Anwendung mit dem Visual Studio erzeugen, die auf den so genannten *Microsoft Foundation Classes* (MFC) basiert und bereits die vollständige Funktionalität einer Windows-Anwendung aufweist.



Ein erstes Projekt

2

Ein erstes Projekt mit dem neuen Visual Studio .NET

Wie bereits angesprochen, sollen sich die folgenden Seiten mit dem Erstellen einer MFC-Anwendung befassen, die zwar funktional nicht viel hermachen wird, aber doch den kompletten Ablauf vom Start eines neuen Projekts bis zum Testlauf darlegen wird.

Portierung von alter Software

Gerade Umsteiger vom Visual Studio 6.0 werden sich dadurch schnell zurechtfinden und insbesondere ihre alten Projekte im neuen Studio weiterpflegen können – es soll nicht verschwiegen werden, dass es an einigen Stellen sicherlich trotz allem Portierungsschwierigkeiten geben wird (das bleibt bei neuen Versionen nie aus), doch muss man diese im Einzelfall betrachten, was im Rahmen dieses Buches natürlich leider nicht möglich ist.

Die Entscheidung, trotz der neuen Möglichkeiten zunächst mit einer MFC-Anwendung zu beginnen, liegt ganz einfach darin begründet, dass die MFC immer noch zu den meist verbreiteten Klassenbibliotheken für Windows gehören und auch von Microsoft weiterhin als gültig und nützlich anerkannt werden.

Probleme mit .NET

Diesen Umstand erkennt man nicht zuletzt daran, dass es mit VC++ zwar theoretisch möglich ist, eine vollwertige .NET-Anwendung zu erzeugen, man hierfür aber sämtliche designtechnischen Vorgänge per Hand durchführen müsste.

C# vs. C++

Während C# beispielsweise Editoren zum Anlegen von Dialogen auf Basis des .NET Frameworks bietet, fehlen diese für VC++ – da es kaum anzunehmen ist, dass Microsoft diesen nicht unerheblichen Umstand einfach übersehen hat, muss vermutet werden, dass C# im Bereich der Windows-Entwicklung auf längere Sicht gesehen eine echte Alternative zu VC++ werden könnte.

Letzteres verliert seinen Sinn jedoch nie wirklich, da mit ihm die Entwicklung von so genannten *unmanaged*, nicht verwalteten Anwendungen möglich ist, also Applikationen, die beispielsweise ihren Speicher frei selbst verwalten können.

Das erste Projekt mit dem neuen Studio

Das Projekt, das im Rahmen dieses Kapitels erzeugt werden soll, dient in erster Linie dazu, aufzuzeigen, welche Möglichkeiten das Studio zum Anlegen neuer Windows-Projekte offeriert. Wir wollen uns hier konkret mit den MFC-Anwendungen beschäftigen, da diese in der Entwicklergemeinde weit verbreitet sind und daher viel Support in Form von fertigen Programmgerüsten und Informationen auf den zahlreichen einschlägigen Webseiten bieten.

Lernkurve und nötiges Wissen

Für den absoluten Neueinsteiger in die Windows-Programmierung dürften einige der im Folgenden genannten Fachbegriffe zunächst unklar sein, es wird versucht, alles im Rahmen des zur Verfügung stehenden Platzes hinreichend anzureißen.

Sollten Sie an einer Stelle Verständnisschwierigkeiten haben, lesen Sie zunächst einfach weiter und kehren zu einem späteren Zeitpunkt an diese Stelle zurück. Wichtige Thematiken werden auch im weiteren Verlauf dieses Buches noch näher besprochen.

Projekt anlegen

Um ein neues Projekt zu erzeugen, wählen Sie aus dem Menü *Datei* den Punkt *Neu* > *Neues Projekt*, woraufhin sich ein Dialogfeld ähnlich dem folgenden öffnet:

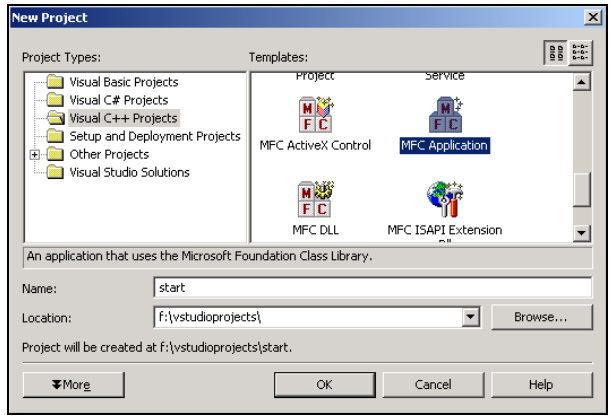


Abb. 2.1
Das Dialogfeld zum Erzeugen eines neuen Projekts

Einstellmöglichkeiten

Hier finden Sie eine ganze Reihe von Einstellungs- und Auswahlmöglichkeiten:

Ein Visual C++-Projekt

- *Projekttyp*: Der zu erzeugende Projekttyp. Dieser hängt insbesondere davon ab, welche Sprache verwendet werden soll. Verwenden Sie hier zunächst einfach den Punkt *Visual C++ Projekte*.
- *Vorlagen*: Nach Auswahl des Projekttyps erscheinen im Fenster rechts daneben eine Reihe von Vorlagen, die ähnlich wie beispielsweise bei Word-Dokumenten dazu dienen, ein Projekt bereits so vorzubereiten, dass es sofort einsetzbar und optimal auf das angestrebte Ziel ausgerichtet ist. Wählen Sie als Vorlage *MFC-Applikation*.
- *Name*: Der Name des zu erzeugenden Dokuments. Tragen Sie hier zum Beispiel *start* ein.

- *Pfad*: Der Pfad, in dem das Projekt eingefügt werden soll. Es wird im spezifizierten Pfad automatisch ein Unterverzeichnis mit dem Namen des Projekts angelegt. Geben Sie als Pfad also beispielsweise *f:\vstudioprojects* ein und als Projektname *start*, werden Sie das erzeugte Projekt im Verzeichnis *f:\vstudioprojects\start* wiederfinden.

Die weiteren Optionen sind an dieser Stelle nicht relevant. Tätigen Sie nun die beschriebenen Einstellungen und bestätigen Sie sie mit *OK*.

Der gute alte MFC-Application Wizard

Es öffnet sich der einigen von Ihnen vielleicht schon vom Visual Studio 6 bekannt Applikationsassisent, jedoch in komplett neuem Gewand und mit neuen Einstellmöglichkeiten. Die erste Seite wirkt allerdings noch harmlos und stellt lediglich eine Übersicht über die derzeit eingestellten Optionen für die zu erzeugende Anwendung dar.

Voreingestellte Anwendungsparameter

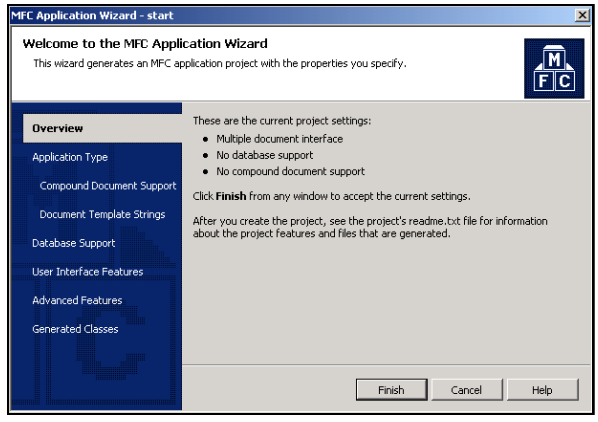
Das sind im Falle der MFC-Anwendungen:

- MDI als Benutzerschnittstelle
- Kein Datenbanksupport
- Keine Unterstützung für Verbunddokumente.

Sind diese Einstellungen für Sie zufrieden stellend, können Sie jederzeit durch Anklicken von *Fertig stellen* das Erzeugen des Projekts in Auftrag geben. Anderenfalls sind in der linken Leiste eine Reihe von Punkten vorgegeben, die angeklickt werden können und weitere Optionen bieten.

Es sei darauf hingewiesen, dass Sie auf dieser ersten Seite keine Möglichkeit haben, die Einstellungen zu verändern, es handelt sich tatsächlich nur um eine reine Informationsseite.

Abb. 2.2
Die Zusammenfassung über die Einstellungen



Festlegen des Anwendungstyps

Auf der ersten Einstellungsseite – also der zweiten Seite in der Assistentenleiste – lassen sich bereits einige relevante Optionen für die zu entstehende Anwendung einrichten.

SDI, MDI und Dialoge

Als Erstes ist der eigentliche Anwendungstyp erwähnenswert. Hier wird, unterschieden zwischen *Single Document* (*SDI*) und *Multiple Document* (MDI), dazu kommt die Möglichkeit, eine dialogfeldbasierte Anwendung zu erzeugen, die darüber hinaus als HTML-Dialog ausgerichtet sein kann.

MDI, SDI und dialogfeldbasierte Anwendungen

Wenn Sie bereits einige Zeit mit Windows-Applikationen zu tun hatten, kennen Sie sicherlich schon alle drei oben genannten Anwendungstypen, wenn auch vielleicht nicht unter deren speziellem Namen.

Der einfachste Anwendungstyp ist die dialogfeldbasierte Applikation, dabei werden sämtliche Vorgänge innerhalb eines – wie es der Name schon sagt – Dialogfelds abgehandelt. Ein Beispiel hierfür ist der Windows beiliegende Rechner. Allgemein ist es immer dann sinnvoll, eine dialogfeldbasierte Anwendung zu erzeugen, wenn kein konkretes Dokument zu bearbeiten ist, sondern vielleicht nur Einstellungen zu setzen oder einfache Aktionen durchzuführen sind.

SDI-Anwendungen (Single Document Interface) bieten ein Fenster, in dem ein Dokument geöffnet sein kann. Lädt der Benutzer ein zweites Dokument, wird das erste geschlossen, das neue nimmt seinen Platz ein. Ein Beispiel hierfür ist der Windows-eigene Editor (Notepad).

MDI-Applikationen (Multiple Document Interface) erlauben das gleichzeitige Öffnen beliebig vieler Dokumente – frühere Versionen von Word sind ein Beispiel hierfür, aber auch das Visual Studio selbst.

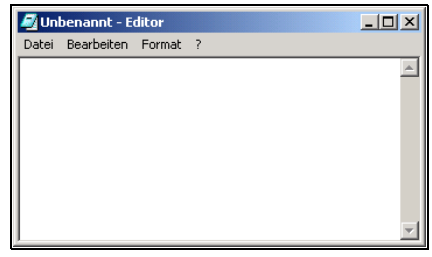


Abb. 2.3
SDI-Anwendung (hier: Editor)

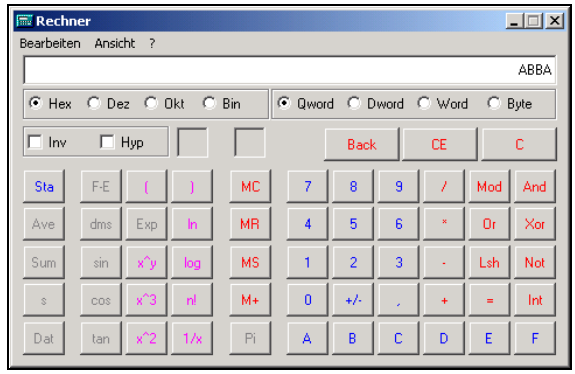


Abb. 2.4
Dialogfeldbasierte Anwendung (hier: *Rechner*)

Abb. 2.5
MDI-Anwendung (hier: *Opera*)

Ein neuer Anwendungstyp

Toplevel-Dokumente

Eine Neuerung ist die Möglichkeit, Anwendungen mit mehreren Toplevel-Dokumenten (ähnlich Microsoft Word, bei dem jedes geöffnete Dokument ein eigenständiges Fenster ist) zu erzeugen.

Toplevel bezeichnet in diesem Fall salopp gesagt den Verwandtschaftgrad zwischen Fenstern und bezieht sich auf ein Fenster, das keine übergeordneten Objekte besitzt.

Genaugenommen handelt es sich bei einer Toplevel-Anwendung um eine MDI-Applikation, die die Dokumente im SDI-Stil verwaltet.

Englische Bezeichnungen

Die Sprache der Windows-Entwickler ist englisch, daran gibt es nichts zu rütteln. Selbst in deutschen Büchern wird mehr und mehr mit den englischen Originalbegriffen gearbeitet, zum einen der Einfachheit halber (es müssen nicht künstlich wirkende Eindeutschungen vorgenommen werden), zum anderen wegen der Konsistenz mit englischsprachigen Dokumentationen.

Auch in dem vorliegenden Werk soll mit dieser Konsistenz nicht gebrochen werden, sodass Sie in vielen Fällen direkt mit den englischen Begriffen konfrontiert werden. Dabei wird immer ein deutsches Synonym, oder, in Ermangelung eines solchen, eine passende Erklärung abgegeben, wie oben im Fall von Toplevel.

Dokument-/Ansicht-architektur

Wie in früheren Versionen kann eingestellt werden, ob eine *Dokument/Ansicht-architektur* für das Projekt zu verwenden ist und in welcher Sprache die Ressourcen-Datei gehalten werden soll – dabei handelt es sich nicht um den Quelltext, sondern lediglich um die Sprache, in der Fehlermeldungen oder Menüpunkte ausgegeben werden.

Weitere Einstellungsmöglichkeiten

Baumstrukturen innerhalb von Fenstern

Weiterhin auf dieser Seite befinden sich Möglichkeiten zur Angabe, ob eine an den Explorer angelehnte Anwendung (mit einer Baumstruktur im linken Teil des Fensters) oder eine herkömmliche MFC-Applikation erzeugt werden soll und ob die MFC in einer gemeinsamen (engl. shared) oder einer statischen (engl. static) Bibliothek zu verwenden sind.

Gemeinsame oder statische MFC-Bibliotheken

Die Entscheidung, ob Sie die MFC-Bibliotheken gemeinsam oder statisch nutzen wollen, hängt nicht zuletzt davon ab, in welchem Kundenkreis und auf welchem Weg Sie Ihre Anwendung verbreiten wollen.

Der Vorteil von gemeinsam verwendeten Bibliotheken liegt auf der Hand: Die Bibliothek ist nur einmal auf dem System vorhanden, belegt also wenig Speicherplatz auf der Festplatte, und kann von beliebig vielen MFC-Anwendungen benutzt werden.

Das ist aber auch genau der Nachteil dieses Verfahrens: die Bibliothek muss auf dem System vorhanden sein. Erfahrungen zeigen, dass dieses bei weitem nicht bei allen Windows-Besitzern der Fall ist, sodass bei einer Distribution die Bibliotheken im Zweifelsfall mitgeliefert werden sollten.

Die statische Bibliothek wird hingegen direkt mit der Applikation zu einem EXE-Paket verschnürt. Hierdurch steigt die Größe der Anwendung nicht unbeträchtlich, mehrere MByte können im Extremfall hinzukommen.

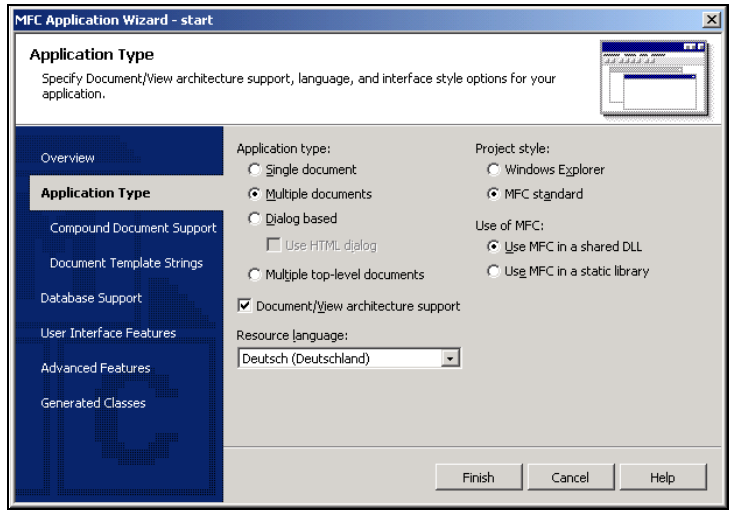
Als Faustregel gilt: Wenn Sie auf Nummer sicher gehen wollen, verteilen Sie statisch gelinkte (also mit den Bibliotheken verbundene) Anwendungen. Werden die Bibliotheken auf dem Zielsystem ohnehin verfügbar sein, sind gemeinsam verwendete Bibliotheken die richtige Einstellungsmöglichkeit.

Gemeinsame Verwendung im ersten Projekt

Für das Projekt, das im Rahmen dieser Einführung erzeugt wird, brauchen Sie keine Einstellungen zu verändern, insbesondere nicht die zur gemeinsamen Benutzen der MFC-Bibliotheken, da die resultierende Anwendung ohnehin nur bedingt zur Weitergabe geeignet ist und Sie im Rahmen des Visual Studio .NET natürlich die passenden Bibliotheken auf Ihrem Rechner vorfinden werden.

Sie sollten nun das folgende Bild vor Augen haben:

Abb. 2.6
Der Applikationstyp



Einstellungen für Verbunddokumente

Wählen Sie nun den nächsten Punkt, *Verbunddokumentunterstützung*. Dort gibt es wiederum eine ganze Reihe von Einstellmöglichkeiten. Verbunddokumente (*Compound Documents* im Englischen) sind Dokumente, die Daten verschiedener Formate wie Text, Grafik, Videosequenzen und so weiter enthalten.

Supporteinstellungen für Verbunddokumente

Je nach Umfang der benötigten Unterstützung können Sie hier den passenden Support einstellen, die Auswahl reicht dabei von *Keine*, über *Container* und *Mini Server* hin zu *Full Server* und *Container/Full Server*.

Als zusätzliche Optionen bietet sich die Möglichkeit, den Dokument-Server bzw. Container als aktiv zu kennzeichnen. Welche Möglichkeiten sich hierdurch im Einzelnen ergeben, würde an dieser Stelle zu weit führen und soll daher nicht näher erläutert werden – es sei vielmehr auf entsprechende weiterführende Fachliteratur verwiesen.

Stellen Sie auch auf dieser Seite wieder alle für Sie interessanten und für das Projekt notwendigen Punkte ein, für das einfache Projekt im Rahmen dieses Buch, können Sie die Einstellungen so belassen, wie sie sind: Die Unterstützung für Verbunddokument wird deaktiviert.

Es ergibt sich somit folgendes Einstellungsbild:

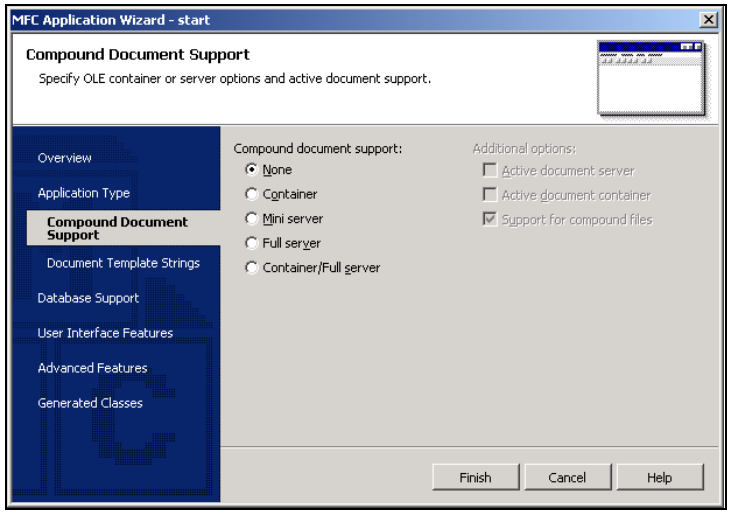


Abb. 2.7
Einstellungen für Verbunddokumente

Zeichenkettenvorlagen

Stringkonstanten

Ein Dialogfeld, das unter dem Visual Studio 6 gut versteckt war, findet sich hier direkt als Punkt, wenn auch untergeordnet, innerhalb der Auswahlleiste: die ehemaligen *Erweiterten Optionen* heißen nun *Dokumentvorlagen-Strings*, bieten aber dieselben Einstellungsmöglichkeiten wie das alte Pendant:

- *Dateinamenerweiterung*: Die Erweiterung für Dokumente, die mit dieser Applikation erzeugt werden.
- *Filetyp ID*: ID für Dateien, die von diesem Programm angelegt werden.

Nicht lokalisierte Strings

Diese beiden ersten Strings werden als nicht-lokalisierte Zeichenketten bezeichnet. Damit meint man Zeichenketten, die in jeder Sprache den gleichen Wert haben. Das ist sinnvoll, damit zum Beispiel Textdokumente im englischsprachigen Raum DOC (für document), im deutschsprachigen Raum aber DOK (für Dokument) heißen. So vorzugehen, würde eine wahre Schwemme an Dateiendung mit sich führen, bedenkt man die vielen verschiedenen Sprachen, die weltweit existieren.

- *Language*: der Sprachstring, der für diese Applikation verwendet werden soll. Für Deutschland ist dieses herkömmlicherweise *Deutsch (Deutschland)*.
- *Doc Type Name*: Der Dokumenttypname dieses Dokuments.
- *File New Short Name*: Name (Kurzform), der angezeigt wird, wenn ein neues Dokument dieses Typs erzeugt werden soll (beispielsweise über einen Shortcut im Explorer oder Ähnliches).
- *Filter Name*: Name für einen Filter, der Zusammen mit diesem Dokument angewendet wird.
- *Main Frame Caption*: Der Titeltext für das Hauptfenster der Applikation.
- *File Type Long Name*: Name (Lange Form), der angezeigt wird, wenn ein neues Dokument dieses Typs erzeugt werden soll (beispielsweise über einen Shortcut im Explorer oder Ähnliches).

Lokalisierte Strings

Im Gegensatz zu den ersten beiden Strings sind die zuletzt beschriebenen Zeichenketten solche, die zu lokalisieren sind.

Zu lokalisierende Zeichenketten sind solche, die in verschiedenen Sprachen unterschiedliche Darstellungen annehmen können sollen. Das macht natürlich nur dort Sinn, wo der Text lediglich als visuelles Merkmal eingesetzt wird und nicht etwa zur Unterscheidung von Dateitypen oder ähnlichem (siehe auch Kasten zu nicht lokalisierten Strings).

Ein gutes Beispiel für zu lokalisierende Strings ist die Beschriftung der Titelzeile eines Dialogs einer Anwendung, der zum Einstellen einiger Optionen dient. Sie als deutscher Programmierer tragen dort vielleicht Bearbeitungsoptionen ein, während ein englischer Kollege eher Edit Options schreiben würde.

Um diesem Umstand Rechnung zu tragen, werden die lokalisierten Strings auch in einer Ressourcen-Datei und nicht etwa fest in das Programm hineinübersetzt ausgeliefert, um ein schnelles Anpassen ohne Neukompilierung der Applikation zu ermöglichen).

Belassen Sie auch in diesem Fall die Einstellungen wieder so, wie sie sind, es sei denn, Sie möchten gleich konkret ein neues Projekt erzeugen, mit dem Sie weiter arbeiten möchten – dieses empfiehlt sich allerdings nur für Benutzer, die bereits mit den MFC gearbeitet haben.

Wenn alles unverändert bleibt, ergibt sich folgendes Bild:

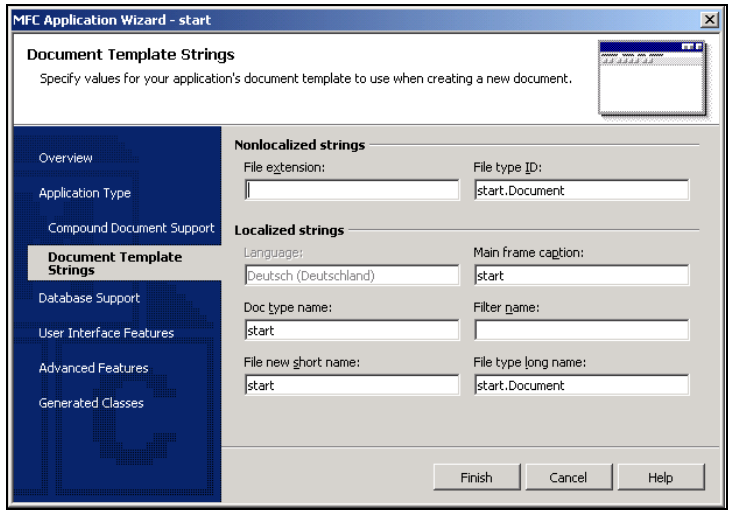


Abb. 2.8
Globale Zeichenketten für die Applikation

Datenbankunterstützung

Das nun folgende Fenster (zu Erreichen durch Anwahl des Schriftzugs *Datenbankunterstützung* im linken Navigationsbalken) ermöglicht Ihnen Einstellungen zur Verwendung von Datenbanken innerhalb des zu erzeugenden Projekts.

Einstellungen für die Datenbankunterstützung

Auch hier gilt wieder, dass einige Dialoge des alten Studios zusammengefasst und erweitert wurden. Die Punkte im Einzelnen:

- *Datenbank Support*: Gibt an, welche Art von Datenbankunterstützung Sie wählen können. Die Einstellmöglichkeiten reichen von *keine* über *Nur Header Files* bis hin zu *Datenbankansicht ohne Dateiunterstützung* und *Datenbankansicht mit Dateiunterstützung*. Der Unterschied bei den

Wahlmöglichkeiten besteht darin, wie viel an Quelltext der Assistent Ihnen bereits in vorgefertigter Art und Weise zur Verfügung stellt und welche Funktionalität diese Quelltexte enthalten sollen.

- *Generate attributed database class*: Erzeugt automatische eine mit Attributen versehene Klasse für die zu verwendende Datenbank zur einfacheren Verwendung.
- *Bind All Columns*: Bindet automatisch alle Spalten der Datenbank.
- Typ: Hier stehen *Dynaset* und *Snapshot* zur Verfügung. Der Unterschied besteht darin, dass beim *Dynaset* Benachrichtigungen an einen Benutzer gesandt werden, wenn ein anderer gleichzeitig mit denselben Daten arbeitet. Der *Snapshot*typ bietet diese Möglichkeit nicht, ist dafür deutlich einfacher einzusetzen.
- Clienttyp: Hier können Sie zwischen dem altgedienten ODBC (*Open Database Connectivity*) und der relative neuen, COM basierenden Schnittstelle *OLE DB* wählen.
- Auswählen einer Datenquelle: Ermöglicht weitere Angabe zur Spezifizierung der Quelldatenbank, die innerhalb der Applikation verwendet werden soll

Für das vorliegende Projekt benötigen wir keine Datenbankunterstützung, sodass Sie die Einstellungen unverändert übernehmen können.

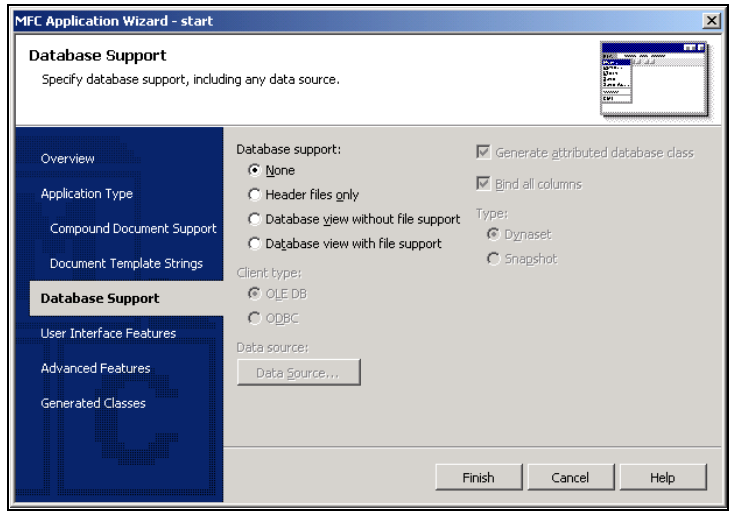


Abb. 2.9 **Datenbankunterstützung**

Benutzeroberfläche (User Interface)

Hier finden sich Einstellungen die Fenster betreffend, mit denen ein Benutzer agieren kann.

Aufgrund der vielzähligen Optionen, die dieses Fenster bietet, soll zur Abwechslung der Screenshot am Anfang der Erläuterungen stehen:

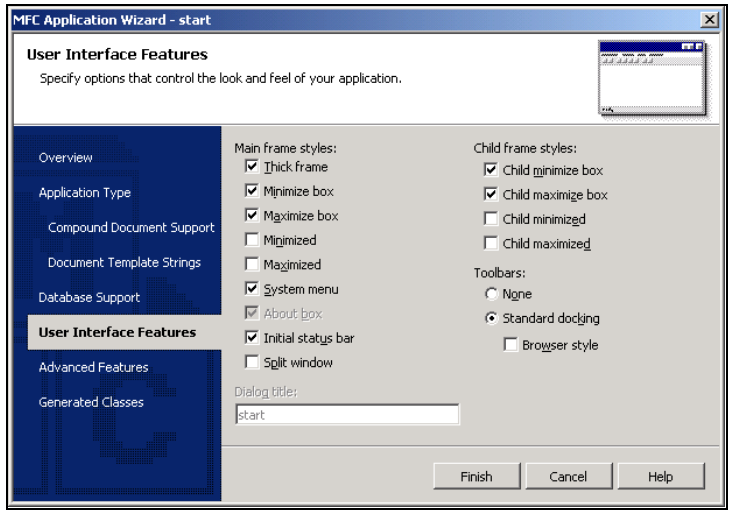


Abb. 2.10
Einstellungen für die Benutzeroberfläche

Main Frame Stilattribute

Rahmeneinstellungen

- *Thick Frame*: Hauptfenster hat einen dicken Rahmen.
- *Minimize Box*: Hauptfenster hat eine Box zum Minimieren.
- *Maximize Box*: Hauptfenster hat eine Box zum Maximimieren.
- *Minimized*: Hauptfenster ist initial minimiert.
- *Maximized*: Hauptfenster ist initial maximiert.
- *System Menu*: Hauptfenster besitzt ein System Menü (erreichbar durch Anklicken des Applikations-Icons).
- *Über Box*: Fenster hat eine *Über*-Box.

Zusätzliche Attribute

- *Initial Status Bar*: Fenster hat beim Start eine Statuszeile.
- *Split Window*: Hauptfenster ist gesplittet.
- *Dialog Title*: Einstellung für Dialoge, enthält den Namen des Dialogs, der in der Titelzeile dargestellt wird.

Child Frame Stilattribute:

- *Child minimize Box*: Kindfenster haben Schaltfläche zur Minimierung des Fensters.
- *Child maximize Box*: Kindfenster haben Schaltfläche zur Maximierung des Fensters.
- *Child minimized*: Kindfenster sind initial minimiert.
- *Child maximized*: Kindfenster sind initial maximiert.

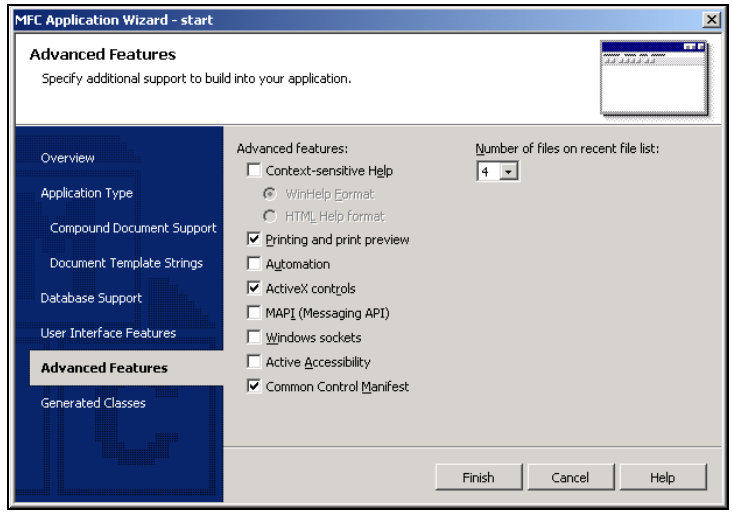
Werkzeugleisten

- *None*: Anwendung hat keine Werkzeugleisten
- *Standard Docking*: Anwendung hat Werkzeugleisten, die sich wie gewöhnliche Toolbars an den Rändern andocken lassen
- *Browser Style*: Fenster hat Werkzeugleisten, die in Ihrem Aussehen denen aus Browsern gleichen.

Erweiterte Einstellungen

Der vorletzte Dialog in der Reihe der Konfigurationsdialoge für die zu erzeugende Anwendung beinhaltet erweiterte Einstellungen für das Verhalten und die Features der Applikation:

Abb. 2.11
Erweiterte Optionen



- *Kontextsensitive Hilfe*: Gibt an, ob es eine Hilfsoption innerhalb der Anwendung geben soll, die nähere Informationen zur Bedienung auf Wunsch des Benutzers liefert. Es stehen hierfür das bewährte *WinHelp* und das neuer *HTMLHelp*-Format zur Verfügung.

  Einbinden von Hilfedokumenten

- *Drucken und Druckvorschau*: Entscheidet darüber, ob die Anwendung das Drucken von Dokumenten unterstützen soll.

- *Automation*: Entscheidet darüber, ob die Anwendung Automation unterstützen soll.

- *ActiveX Controls*: Gibt an, ob die Anwendung ActiveX-Kontrollen verwenden können soll.

- *Messaging API (MAPI)*: Diese wird benötigt, wenn die Anwendung Mailnachrichten verwalten können soll. Dieses ist bei Anwendungen sinnvoll, die häufig ihre Dokumente per E-Mail versenden müssen.

  Mailversand

- *Windows Sockets*: Gestattet den Zugriff per TCP/IP auf externe und interne Netzwerke

- *Active Accessibility*: Bietet Unterstützung bei der Darstellung von Inhalten in Form von Lesehilfen etc, beispielsweise zur Kompensierung von Sehbehinderungen und ähnlichem.

- *Common Control Manifest*: Dient zur Verwendung der Common Control Bibliothek, die mit XP ausgeliefert wurde.

- *Anzahl der Dateien in Liste der zuletzt verwendeten Dateien*: Gibt die Anzahl der Dateien an, die im Startmenü am unteren Ende angezeigt werden sollen – dieses sind gerade die Dateien, die zuletzt bearbeitet wurden.

  Verwendete Dokumente

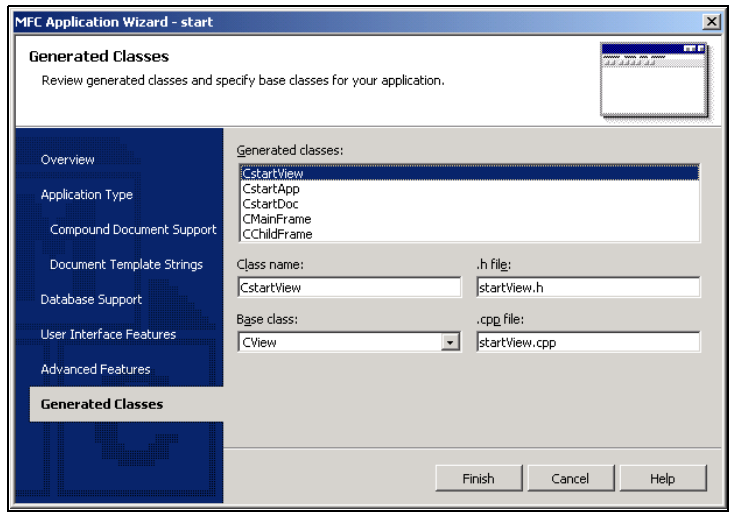
Lassen Sie auch in diesem Fall die Einstellungen so, wie sie sind, es sei denn, Sie haben einen Grund, sie zu ändern (zum Beispiel, um eine Option direkt auszutesten, die im Rahmen dieses Beispiels nicht vorkommen wird).

Ansonsten ist es jetzt an der Zeit, den letzten Punkt unter Augenschein zu nehmen.

Übersicht über die zu erzeugenden Klassen

Der letzte Dialog zeigt Ihnen an, welche Klassen vom MFC-Assistenten angelegt werden, Sie haben hier noch die Möglichkeit, die Basisklasse für die Ansichtsklasse festzulegen und die Header- sowie Implementationsdateinamen zu verändern:

Abb. 2.12
Zu erzeugende Klassen



Was es mit den Ansichtsklassen auf sich hat, werden Sie im Rahmen des nächsten Kapitels erfahren, wenn es darum geht, tiefer in die MFC-Entwicklung einzusteigen. Für jetzt sei nur gesagt, dass hiermit die Art, in der ein Dokument dargestellt wird, festgelegt werden kann.

Ende der Rundreise

Wenn Sie an dieser Stelle angelangt sind, haben Sie sämtliche Dialogfenster gesehen, in denen Einstellungen für das vom Assistenten zu erzeugende Projekt festgelegt werden können.

Sollten Sie schon mit dem Visual Studio 6.0 gearbeitet haben, werden Sie die meisten Optionen wiedererkannt haben, auch wenn sich die eine oder andere an einer vermeintlich seltsamen Position befunden haben mag.

Einige Erweiterungen sind ebenfalls dazu gekommen, die den ohnehin schon guten MFC-Anwendungsassistenten der Vorgängerversion abzurunden verstehen.

Das Projekt erzeugen

Was jetzt noch fehlt, ist der eigentliche Erzeugungsschritt, klicken Sie also einmal auf *Fertig stellen* und Sie werden sehen, wie der Assistent die benötigten Files erzeugt und das Projekt für Sie öffnet.

Der Solution Explorer

Was früher als *Arbeitsbereich* bekannt war, heißt jetzt *Solution Explorer* – stören Sie sich für den Anfang nicht an diesem Namen, es wird in Kürze darauf eingegangen werden.

Der Solution Explorer selbst bietet Ihnen nun das folgende Bild, das bis auf den ungewohnten Namen recht vertraut aussieht:

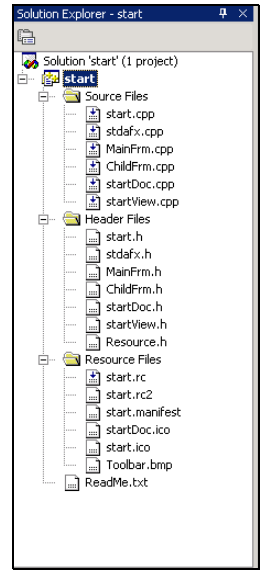


Abb. 2.13
Der Solution Explorer, der früher einmal Arbeitsbereich hieß

Kompilieren ...

Nun soll nicht mehr lange gezögert werden, schließlich wollen Sie auch sehen, dass die Anwendung ihren Dienst versieht und Sie sich nicht umsonst durch die Einstellungsseiten gemüht haben.

Starten des Programms

Starten Sie den Übersetzungsvorgang durch Anwahl des Menüpunkts *Project* > *Erstellen* – Visual Studio 6.0 Benutzer beachten an dieser Stelle insbesondere, dass das Kompilieren deutlich schneller vonstatten geht als mit der vergleichbare älteren Compilerversion. Natürlich erhalten Sie auch eine Ausgabe, die im Folgenden dargestellt ist:

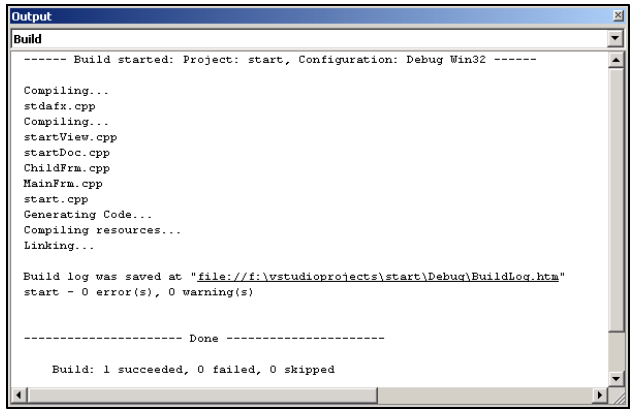


Abb. 2.14
Die Ausgaben während des Kompiliervorgangs

Auffallend: Ein Log des Übersetzungsvorgangs wird gespeichert, das kann nützlich sein, wenn Fehler beim Übersetzen auftreten, die später nachvollzogen werden soll.

... und starten

Fehlerfreie Übersetzung

Da das Projekt fehlerfrei übersetzt werden konnte, ist es nun an der Zeit, es zu starten. Öffnen Sie dazu das *Menü Debug* und wählen Sie den Punkt *Run* – das Programm wird daraufhin innerhalb des Debuggers gestartet.

Weitere Startmöglichkeiten

Alternativ hätten Sie die Anwendung auch außerhalb des Debuggers über den passenden Menüpunkt starten können (oder auch direkt über die beim Übersetzungsvorgang erzeugte EXE-Datei), doch das Ergebnis ist in diesem Fall dasselbe.

Das fertige Programm tut genau das, was man von ihm erwartet: Es ist eine vollständige Windows-Anwendung, jedoch ohne Inhalt, es tut einfach nichts.

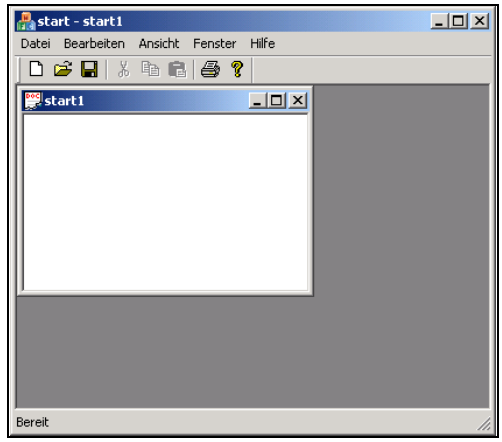
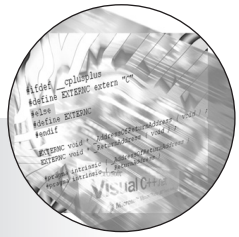


Abb. 2.15
Das lauffähige Programm



Grundlagen für die MFC-Entwicklung

3

Eine kurze Einführung in die Windows-Programmierung

Wer noch nie zuvor Anwendungen für ein multitaskingfähiges Betriebssystem mit grafischer Oberfläche entwickelt hat, wird sich am Anfang der Bemühungen, in diesen Sektor einzusteigen, etwas schwer tun.

Fachbegriffe in der Windows-Welt

Das Konzept von *Fenstern*, *Nachrichten*, *Semaphoren* und *Threads* – um nur eine kleine Anzahl von wichtigen Kernpunkten zu nennen – gehört sicherlich zu den umfangreichsten Themen, die es für einen einsteigenden Entwickler zu lösen gilt.

Ressourcenaufteilung

Befanden sich seine Programme (beispielsweise zu MS-DOS-Zeiten) noch allein in einem abgeschlossenen Raum, muss er sich nun mit anderen Anwendungen – und im übertragenen Sinne deren Entwicklern – die meist spärlichen Ressourcen teilen und Obacht walten lassen, dass das eigene Programm nicht etwa Bereiche von anderen Anwendungen in Anspruch nimmt oder gar zerstört.

Moderne Betriebssysteme

Zum Glück ist das Zerstören anderer Applikationen heutzutage kaum noch möglich, vielfach sogar gar nicht mehr durchführbar, da die Betriebssysteme selbst darauf achten, dass Anwendungen nur den ihnen zugehörigen Bereich verwenden und nicht in den Adressraum anderer Programme eindringen können.

Das erlaubt es dem Entwickler, sich mehr auf die eigentlich wichtigen Aspekte der Windows-Programmierung zu kümmern, nämlich das Verwalten von Fenstern und die Interaktivität mit dem Anwender.

Fensterverwaltung

Kapselung von API-Funktionalität durch die MFC

Wenn hier von Fensterverwaltung gesprochen wird, handelt es sich im Wesentlichen um eine buchführerische Tätigkeit, denn die gesamten Interna – Anlegen des Fensters, Reaktion auf Benutzereingaben wie Größenveränderungen und so weiter – werden innerhalb der MFC weitestgehend vor dem Benutzer versteckt und nur offen gelegt, sobald der Anwender eine von der Norm abweichende Tätigkeit durchführen möchte.

Sie haben im letzten Kapitel gesehen, wie mit einigen wenigen Mausklicks eine komplette Windows-Anwendung erzeugt wurde, die zumindest von der Bedienung her eine vollständige Applikation ergab.

Unterschiedliche Fenstertypen

Es ist allerdings trotzdem notwendig, dass Sie einmal genaueres Augenmerk auf die verschiedenen Fenstertypen richten, mit denen man es unter Windows zu tun hat.

Die folgende Grafik stellt einige davon, ohne Anspruch auf Vollständigkeit, zusammen mit weiteren wichtigen Begriffen dar:

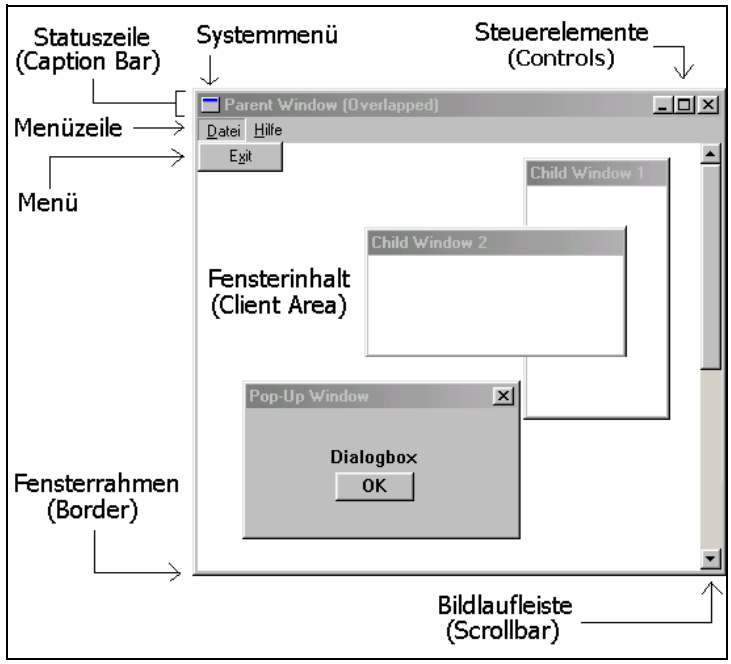


Abb. 3.1
Verschiedene Fenstertypen und wichtige Begriffe

Eigenschaften von Fenstern

Eine interessante Eigenschaft von Fenstern ist es, dass sie, sozusagen rekursiv, aus weiteren Fenstern aufgebaut sind. Prinzipiell ist jedes Element, das sich innerhalb eines Fensters befindet, wiederum ein Fenster, im obigen Beispiel zählen beispielsweise auch die Steuerelemente oder die Bildlaufleiste dazu.

Fenster und andere Fenster

Wenn wir sagen, dass eigentlich alle Komponenten eines Fensters wiederum Fenster sind, so geschieht dieses aus dem einfachen Grund, dass die Handhabung der verschiedenen Teilaspekte immer in der gleichen Art und Weise abgehandelt werden.

Wenn Sie also wissen, wie man ein Fenster öffnet, verschiebt oder auf seinen Status hin überprüft, können Sie dieses Wissen auch direkt auf sämtliche weiteren Komponenten anwenden.

Insbesondere ist auch das Konzept der Nachrichten, das im weiteren Verlauf dieses Kapitels beschrieben wird, auf sämtliche Komponenten anwendbar.

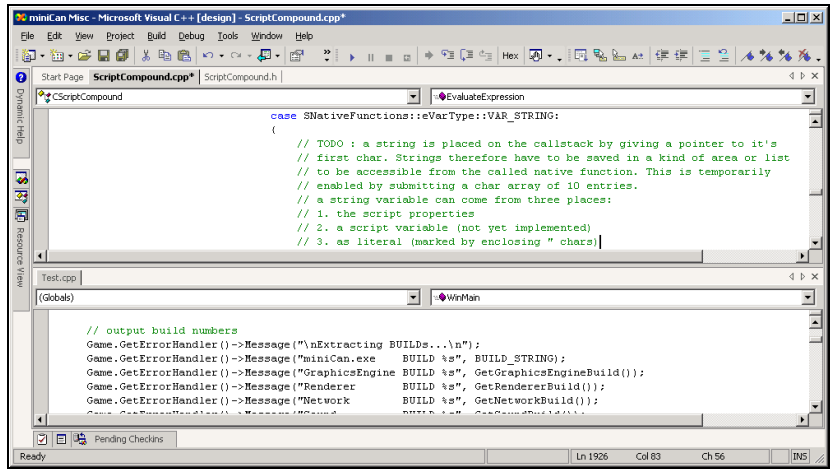
Weitere Bezeichnungen

Fensterhierarchien

Es gibt Hierarchien unter den Fenstern, diese werden dadurch bestimmt, welche Fenster von welchen *Elternfenstern* erzeugt oder umschlossen werden.

Wenn Sie sich zum Beispiel das Visual Studio anschauen, sind die beiden in diesem Zusammenhang wichtigsten Fensterarten sichtbar, sobald Sie zum Beispiel das Projekt aus dem letzten Kapitel geöffnet haben und eine Quelltextdatei betrachten:

Abb. 3.2
Startup – Mainframe und Kindfenster



Zum einen sehen Sie in der Abbildung das *Hauptrahmenfenster* (engl. *Mainframe*). Es ist ein Toplevel-Fenster, da es keine übergeordneten Fenster gibt, die diesen Mainframe enthalten.

Gleichsam existieren eine Reihe von untergeordneten Fenstern, die als *Kindfenster* oder *Child Windows* bezeichnet werden – in diesem Fall zwei Fenster, die Quelltexte darstellen, sowie eine ganze Reihe von Toolbars und weiteren Kontrollelementen.

Verwandtschaftsverhältnisse

Beachten Sie, dass diese Bezeichnungen nicht nur eine Abstammungs- sondern insbesondere auch eine Existenzberechtigung beinhalten: Wird ein Elternfenster (zum Beispiel der Visual Studio .NET Mainframe) geschlossen, endet gleichzeitig auch die Existenz der untergeordneten Kindfenster!

Das Nachrichtenkonzept von Windows

Um effektive Windows-Programmierung betreiben zu können, ist es unabdingbar, die Grundprinzipien des Nachrichtenkonzepts begriffen zu haben, das sich durch das gesamte Betriebssystem hindurchzieht.

Am einfachsten kann man sich die Materie klarmachen, indem man einmal betrachtet, wie Windows arbeitet.

Auslastung des Systems

Nehmen wir zum Beispiel das Visual Studio .NET – starten Sie es und warten Sie, bis sich das Hauptfenster aufgebaut hat. Wenn Sie nun keine Eingabe tätigen und mit der Maus keine Elemente an- oder Aktionen auswählen, was wird passieren? Richtig, nichts – und das schlägt sich auch in der Auslastung des Systems nieder.

Der Task-Manager

Öffnen Sie den *Task-Manager* (durch Rechtsklick in die Startzeile und Auswahl des Punkts *Task-Manager* oder durch gleichzeitiges Drücken von Alt+Strg+Entf und nachfolgende Auswahl des Punkts *Task-Manager*) und schauen Sie sich unter der Registerkarte *Prozesse* die CPU-Nutzung des Prozesses *devenv.exe* an:

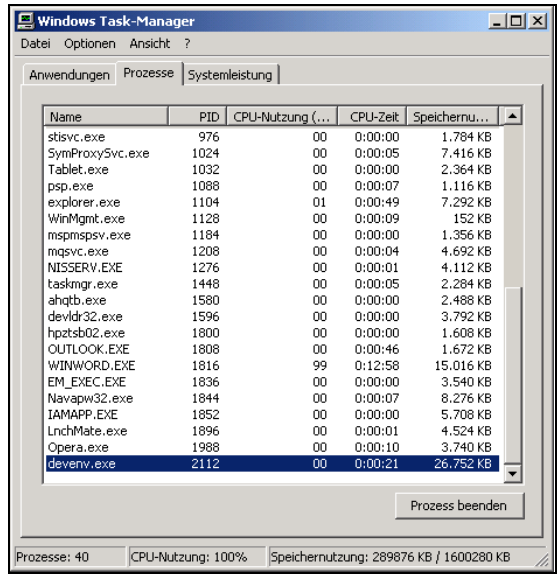


Abb. 3.3
Der Task-Manager zeigt, dass das Studio nichts tut

Die Auslastung liegt bei ungefähr Null Prozent, was ein klares Indiz dafür ist, dass die Entwicklungsumgebung derzeit auch versteckt im Hintergrund, unbemerkt von den Augen des Anwenders, nichts tut.

Wenn Sie jetzt einen Menüpunkt – zum Beispiel den zum Erzeugen eines neuen Projekts – auswählen, reagiert das Studio aber prompt. Wie funktioniert das?

Ereignisse als Auslöser von Aktionen

Winterschlaf

Das Auswählen eines Menüpunkts macht erst einmal nichts Esoterisches, sondern löst ein so genanntes Ereignis aus. In Wirklichkeit werden sogar eine ganze Anzahl von Ereignissen ausgelöst, aber das soll an dieser Stelle für die Betrachtung nicht weiter relevant sein.

Ein Ereignis ist zunächst nichts anderes als eine bestimmte Aktion, die der Benutzer ausführt. Umgangssprachlich könnte man diese Aktionen zum Beispiel so beschreiben:

- „Mauszeiger wurde an Position (x, y) bewegt“
- „Linke Maustaste wurde gedrückt“
- „Zeichen 'X' wurde eingetippt“
- „Fenster wurde an Position (x, y) verschoben“

Verständigung mit Windows

Nun versteht Windows die deutsche Umgangssprache leider nicht, sodass anstelle der Prosatexte natürlich, wie häufig bei der Beschreibung von immer wiederkehrenden Sachverhalten oder Werten innerhalb eines Programms, entsprechende Zahlenwerte einzusetzen sind. Wie Sie später sehen werden, existieren für jedes mögliche Ereignis Konstanten, die aus Ihren Programmen heraus eingesetzt werden können.

Nachrichten zum Anzeigen von Ereignissen

Ein Ereignis allein ist nicht allzu viel wert, denn das alleinige Eintreten eines solchen erzeugt erst einmal keine Reaktion von Seiten eines Programms.

Über Ereignisse in Kenntnis setzen

Das liegt daran, dass die Anwendung zunächst noch nichts von dem eingetretenen Ereignis (engl. Event) wissen kann – lediglich das Betriebssystem hat zur Kenntnis genommen, dass der Benutzer eine Aktion durchführen will (oder besser: schon durchgeführt hat, auch wenn die Behandlung derselben noch nicht vollzogen wurde).

Tritt ein Ereignis auf, hat das Betriebssystem alle Hände voll zu tun: zunächst muss geprüft werden, um welche Art von Ereignis es sich handelt und welche Anwendungen davon betroffen sind.

Wurde beispielsweise die Maus über einem Zeichenprogrammfenster bewegt, muss zum einen der Mauszeiger verschoben werden, um dem Benutzer eine direkte Reaktion auf sein Verschieben des Eingabegeräts zu signalisieren, zum anderen sollte das Zeichenprogramm auch über das Eintreten dieser Statusveränderung informiert werden – vielleicht will der Anwender gerade etwas zeichnen.

Der Nachrichtenpuffer

Die Geister scheiden sich darüber, ob es korrekt ist zu sagen, dass Windows-Anwendungen von eingehenden Nachrichten in Kenntnis setzt.

Puffer zum Zwischenspeichern von Nachrichten

In der Tat ist es so, dass jede Applikation über einen so genannten Nachrichtenpuffer verfügt, der zur Entgegennahme von Windows-Nachrichten gedacht ist und die Botschaften solange speichert (eben „puffert“), bis sie vom zugehörigen Programm verarbeitet wurden. Die Gesamtheit der Puffer wird als Systemnachrichtenpuffer bezeichnet:

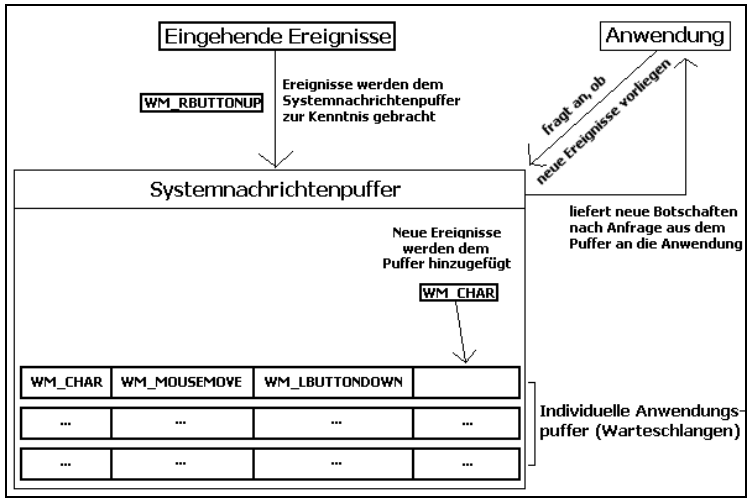


Abb. 3.4
Der Systemnachrichtenpuffer

Nachrichtenverarbeitung

Hier liegt auch der eigentliche Haken, denn die Verarbeitung liegt vollständig in den Händen der Anwendung – sie ist dafür verantwortlich, zu prüfen, ob neue Botschaften vorliegen. Windows schiebt die Nachrichten zwar in den Puffer, zieht sich damit allerdings auch schon aus der Verantwortung zurück.

Das heißt im Extremfall, dass eine Anwendung, die nie ihren Puffer auf den Eingang neuer Nachrichten hin überprüft, niemals auch nur eine einzige Botschaft zu Gesicht bekommen wird und sich gegenüber Benutzereingaben als blind und taub darstellen wird.

Das Hauptprogramm von Windows-Anwendungen

Weiter oben wurde gezeigt, dass Windows-Anwendungen im Ruhezustand nichts tun – das ist nicht ganz richtig, in Wirklichkeit sehen die Hauptprogramme von Windows-Programmen meist so aus, dass sie aus einem fest vorgegebenen Ablaufsplan bestehen, der sich in drei Schritte gliedert: Initialisierung, Nachrichtenverarbeitung und Herunterfahren der Anwendung.

Gerade der zweite Schritt ist es, der im Ruhezustand einer Anwendung die ganze Zeit durchlaufen wird. Durch geschickte Betriebssystemorganisation ist es möglich, diesen Prozess sehr arbeitssparsam durchzuführen.

Im Idealfall wird die Anwendung sofort benachrichtigt, sobald eine neue Nachricht in den Nachrichtenpuffer eingeht. Sie erkennt dieses allerdings nicht durch die eigentliche Aktion des Einfügens der Nachricht, sondern dadurch, dass der Nachrichtenpuffer nicht mehr leer ist. In der Zwischenzeit wird die Anwendung quasi auf Standby geschaltet und führt gegebenenfalls nur Hintergrundaktivitäten aus, die allerdings dem Programm und nicht windows-internen Zwecken dienen.

Vorteil der Pufferung

Gründe für einen Puffer

Man könnte sich jetzt fragen, warum überhaupt ein zwischengelagerter Puffer notwendig ist; warum schickt man die Ereignisse nicht direkt an die jeweilige Anwendung?

Einsparen von CPU-Zeit

Das liegt an drei wesentlichen Gründen. Der Erste wurde bereits weiter oben genannt: Die Anwendung kann in eine Art Ruhezustand übergehen, bis die nächste Botschaft eintrifft. Würde man Nachrichten direkt an Anwendungen weiterleiten, müsste sie kontinuierlich prüfen, ob Nachrichten vorrätig sind. Dieses Problem ließe sich gegebenenfalls lösen, indem Windows (oder ein anderes Betriebssystem) die Anwendung zunächst aufwecken und ihr dann die Botschaft zuweisen würde.

Schwerwiegender ist ein weiterer Grund, der mit dem eben gesagten eng zusammenhängt: Angenommen, Windows würde Botschaften an Applikationen direkt durchreichen – wie würden sie dann entgegengenommen werden? Vielleicht in der Art, dass sie akzeptiert und direkt verarbeitet werden?

Gründe gegen ein direktes Weiterreichen von Nachrichten

Was, wenn eine Nachricht vielleicht 10ms zur Abarbeitung braucht, und Windows jetzt so lange warten muss, bis das Programm bereit zur Entgegennahme weiterer Nachrichten wäre? Sagen wir hypothetisch weiter, dass der Benutzer dabei ist, die Maus zu bewegen – pro Mausbewegung vergehen nun diese 10 ms, da es sich um ein rechenintensives Programm handelt und viele Dinge berechnet werden müssen, wenn sich die Mausposition ändert (betrachten Sie zum Beispiel komplexe CAD-Applikationen – Computer Aided Design -). Da die Anwendung pixelgenaue Mausinformationen braucht, würde wir für eine Strecke von zehn Bildpunkten bereits 100 ms benötigen. Bei einer Auflösung von 1.024*768 bräuchten Sie dann über zwölf Sekunden, um die Maus von der linken oberen in die rechte untere Ecke zu bewegen. Es ist leicht einsichtig, dass dieses im heutigen Zeitalter nicht mehr tragbar wäre.

Optimierungen durch den Puffer

Redundanz

Der letzte Punkt hängt wiederum mit dem gerade eben Gesagten zusammen – vielfach sind pixelgenaue Mausbewegungen gar nicht erforderlich, sondern vielmehr störend, insbesondere wenn man bedenkt, dass schon bei kleineren Verschiebungen Unmengen an Nachrichten eintreten würden, die durch die Applikation dann auch noch verarbeitet werden müssten.

Windows nutzt nun die Pufferung, um zu sehen, ob eine neu hinzuzufügende Nachricht nicht vielleicht mit einer anderen, schon im Puffer befindlichen – und somit noch nicht abgearbeiteten – Botschaft verschmolzen werden könnte. Befindet sich im Puffer beispielsweise eine Nachricht „Bewege den Mauscursor an Position (100, 100)" und will Windows die Botschaft „Bewege den Mauscursor an Position (120, 120)" in den Puffer schreiben, wäre es beispielsweise sinnvoll, die erste Nachricht gleich komplett durch die zweite zu ersetzen.

Es gibt in diesem Zusammenhang eine ganze Reihe von Optimierungen, die Windows durchführen kann, von denen die Anwendung im Nachhinein nichts bemerkt, aber doch deutlich weniger Botschaften zu bearbeiten hat, als es ohne den intelligenten Eingriff des Betriebssystems der Fall wäre.

Der Vorgang der Optimierungen lässt sich grafisch wie in der folgenden Abbildung zusammenfassen:

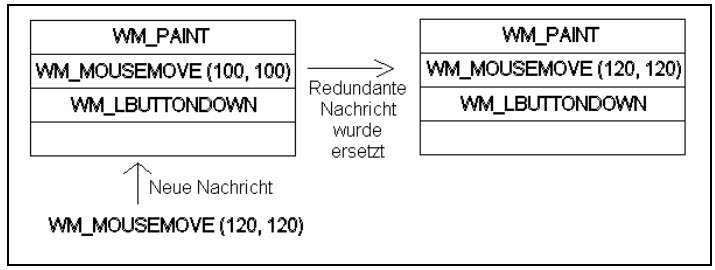


Abb. 3.5
Redundanzprozess

Aufbau von Nachrichten

Sie haben jetzt schon viel von Ereignissen und den sie umschließenden Nachrichten gehört, doch wie ist so eine Nachricht überhaupt aufgebaut?

Unter Windows besteht eine Nachricht aus drei Komponenten:

- Dem Nachrichtencode (dieser besagt, welches Ereignis eingetreten ist und wird durch eine positive ganze Zahl beschrieben)
- Einem 32-Bit-Parameter, der als *wParam* bezeichnet wird. Vor dem Erscheinen von Windows 95 handelte es sich hierbei noch um einen 16-Bit-Parameter – hieraus resultiert auch das w (für Word = 2 Byte = 16 Bit) in der Benennung.

- Einem 32-Bit-Parameter, der als *lParam* bezeichnet wird (l für Long = 2 Word = 4 Byte = 32 Bit)

Aufbau des Nachrichtencodes

Die eigentliche Nachricht steckt im Nachrichtencode, der normalerweise ein ganzzahliger positiver Ausdruck ist – hierfür stehen dann die bereits angesprochenen Konstanten zur Verfügung. Die Nachricht *WM_PAINT*, beispielsweise, die ein Fenster zum Neuzeichnen seines Inhalts auffordert, hat den Wert 15, sodass dieser im Nachrichtencode zu finden sein würde. Dieser Wert wird im Verzeichnis *Vc7/PlatformSDK/Include* in der Datei *WinUser.h* definiert.

WM steht dabei für Window-Message (also Windows-Nachricht), während PAINT die eigentliche Aktion skizziert.

Weitere Nachrichtencodes

Das Wörtchen „normalerweise" im oberen Abschnitt deutet es bereits an: auch hier gibt es Ausnahmen von der Regel. Windows wäre ein ärmliches Betriebssystem, wenn es nur die fest definierten Nachrichtencodes zur Verfügung stellen würde.

Statt dessen ist es dem Entwickler freigestellt, eigene Nachrichtencodes zu erzeugen und diese dann an Anwendungen zu verschicken. Es versteht sich dabei von selbst, dass außer den Applikationen, die speziell daraufhin entwickelt wurden, keine anderen Programme mit diesen Nachrichten etwas anfangen können.

Unbekannte Nachrichtencodes werden von Anwendungen mit einer Standardbehandlung abgehakt – diese tut im Zweifelsfall gar nichts und sorgt nur dafür, dass die Botschaft aus dem Nachrichtenpuffer entfernt wird.

Die Parameter der Nachrichten

Welche Aussagen die beiden Parameter treffen, hängt von der Nachricht selbst ab. Im Falle einer Mausbewegung zum Beispiel (*WM_MOUSEMOVE*) enthält *wParam* einen Wert, der beschreibt, ob während der Bewegung Maus- oder Funktionstasten gedrückt wurden, während *lParam* die neue Mausposition enthält.

Im Einzelfall müssen Sie für die Nachrichten in der Online-Hilfe von MS-VC++ prüfen, welche Bedeutung die Parameter haben. Zum praktischen Nachschlagen befindet sich aber eine Übersicht sämtlicher Windows-Nachrichten mit kurzer Beschreibung im Anhang dieses Buchs.

Konventionen bei der Windows-Programmierung

Wenn Sie Programme für Windows schreiben (und insbesondere mit der im weiteren beschriebenen Klassenbibliothek MFC), werden Sie auf einige Ihnen vielleicht seltsam anmutende Bezeichner wie *lpszText* oder *puiNumber* stoßen.

Verständigung unter Programmierern

Hierbei handelt es sich um die so genannten *ungarische Notation* oder den ungarischen Programmierstil. Es wird dringend angeraten, dass Sie sich auch an diese Konventionen halten, damit Sie sich an die Schreibweisen gewöhnen und andere Programme schneller verstehen können und insbesondere auch für andere Entwickler verständliche Quelltexte schreiben.

Im Wesentlichen handelt es sich um eine Erweiterung von Variablen- und Objektnamen, die aus einfachen kurzen Präfixen bestehen, die wiederum den Typ einer Variablen genauer beschreiben.

Die folgende Tabelle gibt einen Überblick über die in diesem Buch verwendeten Präfixe:

Tabelle 3.1 **Ungarische Notation im Überblick**

Kennung	Bedeutung	Entspricht in C++
b	BOOL	bool
by	BYTE oder UCHAR	byte oder unsigned char
c	char	char
cx, cy	SHORT, benutzt als Größenangabe	short
d	DOUBLE	double
dw	DWORD oder ULONG	long oder unsigned long
e	Enumerierungsvariable	
fn	Funktion	Funktion
h	Handle	Allgemein ein Datum zum Zugriff auf Objekte. Unter Windows meist ein ganzzahliger Ausdruck.
i	int	int
l	long	long

Kennung	Bedeutung	Entspricht in C++
n	Integerzahl, bei der die tatsächliche Größe irrelevant ist	Alle ganzzahligen Variablentypen, wie zum Beispiel *int*, *short* oder *long*
p	Zeiger. Früher wurde noch eine Unterscheidung zwischen sp und lp (*short point* und *longpointer*) getroffen, heutzutage geht man generell von 32-Bit-Zeigern aus.	Zeiger. Aus *int *Zahl* würde im ungarischen Programmierstil *int *pZahl*.
s	String	char[x]
sz	String, der mit NULL terminiert ist	char[x], wobei char[x-1] = NULL.
uc	UCHAR	unsigned char
w	WORD oder UINT	unsigned integer
x, y	SHORT, benutzt als Koordinate	short
E	enum	enum
S	struct	struct
C	class	Class
m_	Member einer Klasse	
in_	Eingabeparameter einer Funktion, wird von der Funktion nicht verändert	
out_	Eingabeparameter einer Funktion, wird von der Funktion bewusst verändert (nur sinnvoll bei Referenzen oder Zeigertypen)	

Ein erstes MFC-Programm

Nach der ganzen Theorie soll in diesem Buch aber auch die Programmierung nicht zu kurz kommen, weshalb es an dieser Stelle mit einem ersten selbst geschriebenen MFC-Programm losgehen soll. Im vorangegangenen Kapitel haben Sie ja bereits eine Anwendung mithilfe des Assistenten erzeugt und sich mit den wesentlichen Einstellungsmöglichkeiten während dieses Vorgangs vertraut gemacht.

Einige Punkte werden sicherlich noch hier und da unklar sein, doch ist dieses für einen einsteigenden MFC-Entwickler ganz natürlich: Der schieren Fülle an Informationen ist nur nach und nach beizukommen, und es ist ein ungeschriebenes Gesetz, dass selbst gestandene Windows-Entwickler jeden Tag etwas Neues über die API, das MFC-Klassengeflecht oder neue entstehende Technologien lernen.

Unklarheiten durch Üben beseitigen

Machen Sie sich also nichts daraus, wenn hin und wieder einige Teilaspekte nicht hundertprozentig klar werden – lesen Sie statt dessen weiter und kehren Sie zu einem späteren Zeitpunkt zu diesen Inhalten zurück.

Ein sanfter Einstieg

Um Ihren ersten wirklichen Kontakt – bislang haben Sie ja noch keine MFC-Quelltexte zu Gesicht bekommen, geschweige denn editiert – so angenehm wie möglich zu machen, werden wir auf den folgenden Seiten mit einer zwar in sich kompletten, aber eher leichtgewichtigen Applikation beginnen, die nichts anderes tut, als ein Fenster zu öffnen und einen Text darin auszugeben.

Wenn Ihnen dieses Vorhaben als zu leicht erscheint, kann ich Sie dahingehend beruhigen, dass wir nicht auf Assistenten zurückgreifen werden, sondern alles per Hand erledigen, was nötig ist, um eine MFC-Anwendungen von Grund auf zu erzeugen.

Arbeit ohne Assistenten

Diese Vorgehensweise ist sinnvoll, wenn man bedenkt, dass vom Assistenten erzeugte Quelltexte bereits mehrere hundert Zeilen Code aufweisen – da ist es gut, wenn man bekannte Anker wiederfindet, die man schon einmal selbst per Hand bearbeitet hat.

Beginnen werden wir mit dem Anlegen eines neuen Projekts, öffnen Sie daher das *Datei*-Menü und wählen dort den Punkt *Neu* > *Projekt*. Es öffnet sich das schon bekannte Fenster zur Auswahl des gewünschten Projekttyps.

Anlegen des neuen Projekts

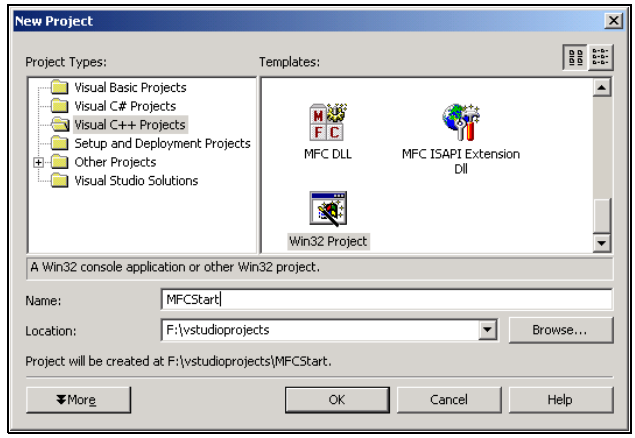
Unter *Projekttypen* muss zunächst *Visual C++ Projekte* aktiviert werden, um die korrekten Vorlagen zum Vorschein zu bringen.

Suchen Sie aus der Liste im rechten Teil des Dialogs den Punkt *Win32 Project* und tragen Sie als Namen *MFCStart* ein (alternativ befindet sich das in diesem Abschnitt erzeugte Programm auch auf der Buch-CD im Verzeichnis *\Kapitel3\MFCStart*).

Anlegen eines Win32-Projekts

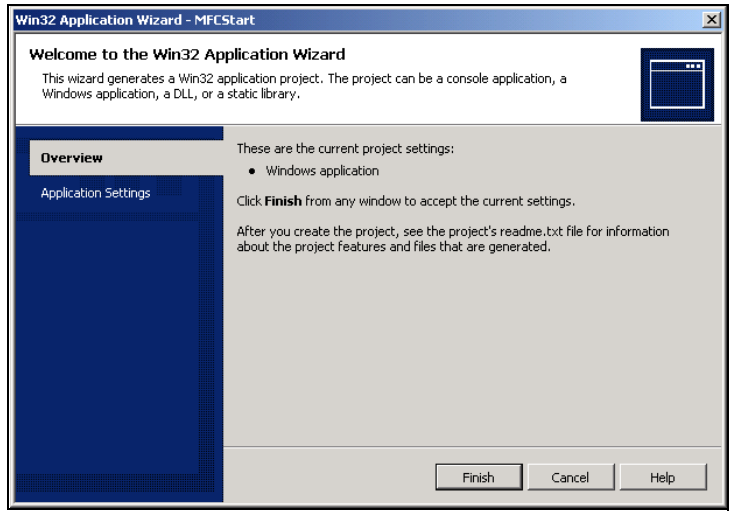
Es ergibt sich das folgende Bild:

Abb. 3.6
Erzeugen Sie ein einfaches Win32-Projekt



Sobald Sie auf *OK* klicken, öffnet sich ein neues Fenster, das den ersten Schritt eines Assistenten anzeigt, der Sie beim Anlegen der Win32-Applikation unterstützen soll – zugegeben, ganz ohne Assistent kommen wir dann doch nicht aus:

Abb. 3.7
Die Übersicht über die zu erzeugende Win32-Anwendung



Das Fenster gibt an, dass eine Windows-Anwendung erzeugt werden soll, was prinzipiell wenig hilfreich ist, denn dazu zählen viele Dinge. Interessant wird diese Aussage erst im entsprechenden Kontext, der über die Auswahl *Einstellungen* im linken Navigationsrahmen angezeigt werden kann:

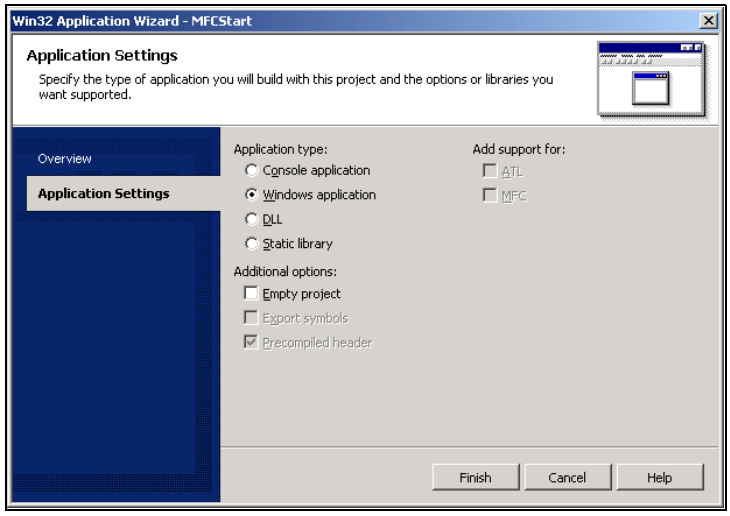


Abb. 3.8
Die Einstellungsmöglichkeiten für Win32-Anwendungen

Applikationstypen

Die folgenden Applikationstypen können ausgewählt werden (dabei erzeugt der Assistent jeweils eine Projektdatei und je nach Programmtyp mehrere vorgefertigte Prototyp-Dateien, falls unter dieses unter den Zusatzoptionen nicht deaktivert wird):

DOS-Fenster-Anwendung

- *Konsolen-Applikation*: Erzeugt eine Anwendung, die in einem MS-DOS-Fenster läuft, also insbesondere in den meisten Fällen ohne Fenster auskommt, jedoch vielleicht mit umfangreicheren Kommandozeilen versorgt werden kann.

Fensteranwendung

- *Windows-Applikation*: Eine echte Windows-Anwendung, die mit Fenstern ausgestattet werden und sämtliche Vorzüge der Win32-Programmierung nutzen kann. Außerdem ist es möglich, die MFC zu verwenden.

Bibliotheken

- *DLL*: Erzeugt eine dynamische *Bibliothek* (engl. *dynamic link library*), die einen Funktionalitätssatz enthält, der von anderen Anwendungen genutzt werden kann. Dynamische Bibliotheken werden erst zur Laufzeit einer Applikation an das Programm gebunden und können von mehreren Anwendungen gleichzeitig genutzt werden.

- *Statische Bibliothek*: Ähnlich der DLL, ist die statische Variante zum direkten Anbinden an Programme gedacht. Dieses ist meist dann sinnvoll, wenn die Bibliothek sehr spezifisch ist und mit großer Wahrscheinlichkeit nicht von anderen Anwendungen genutzt wird.

Unterstützung für zusätzliche Technologien

Sie können für Ihre Anwendungen weitere Unterstützung direkt durch den Assistenten anbinden lassen. Dieses ist allerdings nur für Konsolen-Anwendungen und statische Bibliotheken sinnvoll, da hier der Prozess manuell zu aufwendig wäre:

COM-Objekte

- *ATL*: Diese Bibliothek (ATL steht für *Active Template Library*) dient zur Entwicklung von COM-Komponenten. Auf sie wird im weiteren Verlauf des Buchs noch näher eingegangen. ATL-Unterstützung kann nur für Konsolen-Anwendungen hinzugefügt werden.

Klassenframework

- *MFC*: Die *Microsoft Foundation Classes* (MFC) sind ein objektorientiertes Klassenframework zur Entwicklung von Windows-Anwendungen. Das Aktivieren der Unterstützung ist allerdings nur für Konsolen-Anwendungen und statische Bibliotheken erforderlich, bei einer Windows-Applikation – wie in unserem Fall – ist dieser Schritt nicht nötig.

Zusätzliche Optionen

Als letzten Abschnitt der Dialogseite gibt es eine Reihe von weiteren Optionen, die nachstehend aufgeführt sind:

- *Leeres Projekt*: Erzeugt nur das eigentliche Projekt, aber keine Dateien, die sich in diesem befinden. Sämtliche zum Projekt gehörenden Files müssen per Hand hinzugefügt werden.

- *Export Symbols*: Aktivieren Sie diesen Punkt, wenn Sie eine DLL erzeugen und von dort aus Symbole exportieren möchten.

Beschleunigung von Kompiliervorgängen

- *Precompiled Headers*: Die vorkompilierten Header dienen in erster Linie dazu, den Kompiliervorgang enorm zu beschleunigen. Während ohne diese Option jedes Headerfile möglicherweise einige hundert Mal eingelesen und übersetzt werden müsste, ermöglicht es der Einsatz von vorkompilierten Headerdateien, diesen zeitraubenden Prozess nur einmal durchzuführen und dann mit bereits übersetzen Headern weiterzuarbeiten. Die Option kann nur für statische Bibliotheken aktiviert werden – in unserem Fall werden die vorkompilierten Header automatisch verwendet.

Erzeugungsprozess einleiten

Für unser Projekt brauchen keine Einstellungen verändert zu werden, übernehmen Sie also die vordefinierten Werte und klicken dann auf die *Fertigstellen* Schaltfläche.

Automatische Bereitstellung von Dateien

Der Assistent bereitet das Projekt sowie eine Reihe von Dateien vor, die nun zum Editieren bereit stehen. Schon zu diesem Zeitpunkt existiert eine vollwertige Applikation, wie Sie leicht durch Kompilieren und Ausführen der neu entstandenen Sources verifizieren können (Menü *Build* > *Build Solution* oder F7 zum Kompilieren, dann F5 oder *Debug* > *Start* zum Ausführen des Programms):

Abb. 3.9
Die vom Assistenten angelegte Anwendung

Wie schon beim MFC-Beispiel präsentiert sich also eine vielleicht nicht atemberaubende, aber vollständig funktionierende Anwendung, die darüber hinaus auch noch eine Infobox mit Copyright-Informationen und zwei vorgefertigte Menüs bereit hält.

Natürlich handelt es sich hierbei noch um eine reine Win32-Anwendung – und genau dieser Umstand soll im Folgenden behoben werden.

Der Solution Explorer

Übersicht im Solution Explorer

Werfen Sie an dieser Stelle einen Blick in den Solution Explorer, der im Grunde nichts anderes darstellt als eine Sammlung sämtlicher zu einem Projekt gehörender Dateien. Dazu zählen in erster Linie verständlicherweise die Quelltexte, aber auch Ressourcen wie Icons oder einfache Textdateien mit Hinweisen oder Todo-Listen.

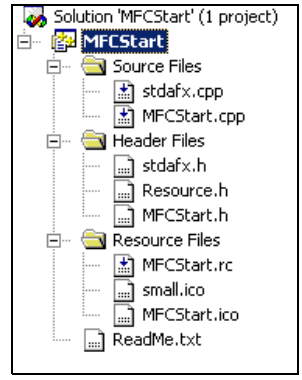


Abb. 3.10
Der Solution Explorer

Projekte im Solution Explorer

Mehr noch, eine als Solution bezeichnete Komplettlösung (*Solution* ist nichts anderes als das englische Wort für *Lösung*) eines Problems, bzw. einer Aufgabenstellung, kann auch beliebig viele unterschiedliche Projekte enthalten.

Unterteilung von Projekten

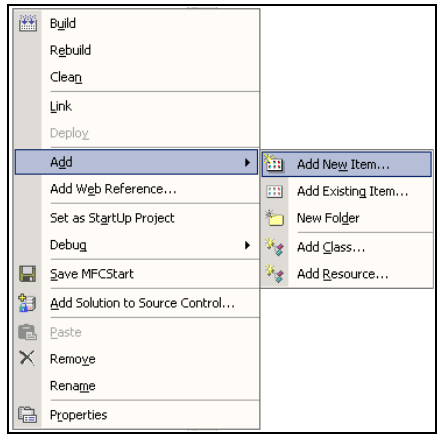
Dieses ist vor allem dann nützlich, wenn Sie es mit großen Aufgaben zu tun haben, deren Unterteilung in kleinere Aufgaben eine Zuordnung der einzelnen Teilgebiete in separate Projekte rechtfertig.

Solution Explorer Optionen

Hinzufügen neuer Dateien

Sie können im Solution Explorer nach Belieben Ordner oder Dateien zu einem Projekt hinzufügen, indem Sie die betroffene Solution auswählen und mit einem Rechtsklick das zugehörige Kontextmenü einblenden. Wählen Sie hier den Befehl *Hinzufügen* sowie im erscheinenden Untermenü den gewünschten Inhalt:

Abb. 3.11
Das Kontextmenü eines Projekts



Hinzufügen von Dateien zu einem Projekt

Neben dem Anlegen von Ordnern zur verbesserten Übersichtlichkeit innerhalb eines Projekts ist es möglich, bestehende Dateien beliebigen Typs in den Rahmen des Projekts einzubinden.

Interessant ist die Möglichkeit, neue Dateien zu erzeugen und einzufügen. Bei Auswahl dieses Punkts bietet sich dem Entwickler ein neues Dialogfeld, das passende Einstellmöglichkeiten zulässt:

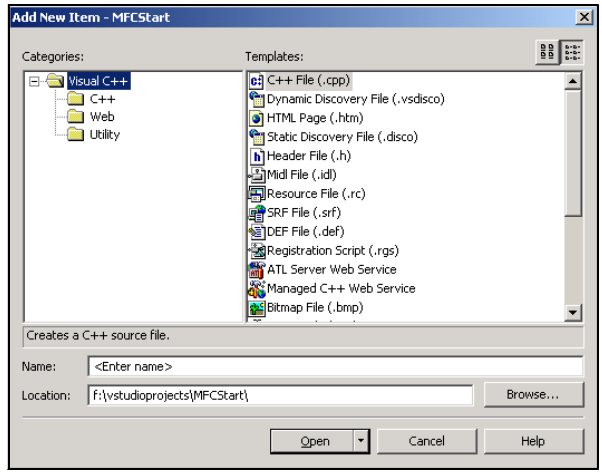


Abb. 3.12
Dialogfeld zum Hinzufügen neuer Dateien

Die Auswahl an unterschiedlichen Dateitypen ist ganz erheblich, normalerweise sollten Sie den jeweils gerade erforderlichen Datentyp auffinden können.

Dateitypen

Dieses erspart den lästigen Umweg über eine externe Applikation, die dann möglicherweise extra mühsam gestartet werden muss, wenn beispielsweise eine neue Bitmap- oder eine HTML-Datei angelegt werden soll.

Neue Klassen in ein Projekt integrieren

Ebenfalls eine angenehme Möglichkeit bietet sich im Solution Explorer zum Anlegen neuer Klassen und zugehöriger Dateien. Um diese Funktionalität auszutesten, wählen Sie im Explorer das *MFCStart*-Projekt aus, öffnen das Kontextmenü und wählen den Punkt *Add* > *New Class*. Es öffnet sich das nachstehend abgebildete Dialogfeld:

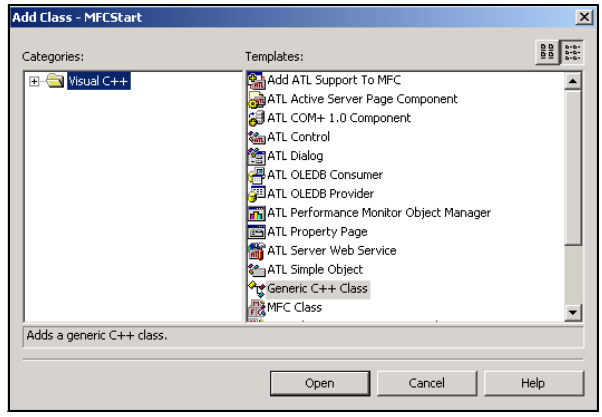


Abb. 3.13
Das Dialogfeld zum Erzeugen neuer Klassen

Klassentypen

Wie für die Assistenten des Studios üblich, haben Sie eine ganze Reihe von Möglichkeiten, von welchem Typ die zu erzeugende Klasse sein soll. Welche Auswahlmöglichkeiten es hier im Einzelnen gibt, ist an dieser Stelle nicht weiter relevant, eine Beschreibung würde den Rahmen des Buchs sprengen.

Herkömmliche Klassen hinzufügen

Wählen Sie einfach den Punkt *Generic C++ Class*, was eine herkömmliche C++-Klasse ohne Schnörkel oder Verzierungen beschreibt. Ein Druck auf den *Öffnen*-Knopf öffnet einen Klassenerzeugungsassistenten, der Ihnen die Möglichkeit bietet, einen Namen für die neue Klasse, sowie die beiden zu erzeugenden Dateien (Header- und Implementationdatei) festzulegen.

Ferner können Sie spezifizieren, ob die neue Klasse von einer anderen, bereits bestehenden Klasse abzuleiten ist, und in welchem Modus *(public, protected, private*) dieses zu geschehen hat.

Generische Klassen sind übrigens nicht dafür vorgesehen, abgeleitete Klassen von MFC-Klassen zu werden, hierfür stehen weitere Klassentypen zur Verfügung.

Die Einstellmöglichkeit, die Klasse als Inline zu erzeugen (Implementationen landen mit im Headerfile) oder ihr einen virtuellen Destruktor mitzugeben, runden den Klassenassistenten ab.

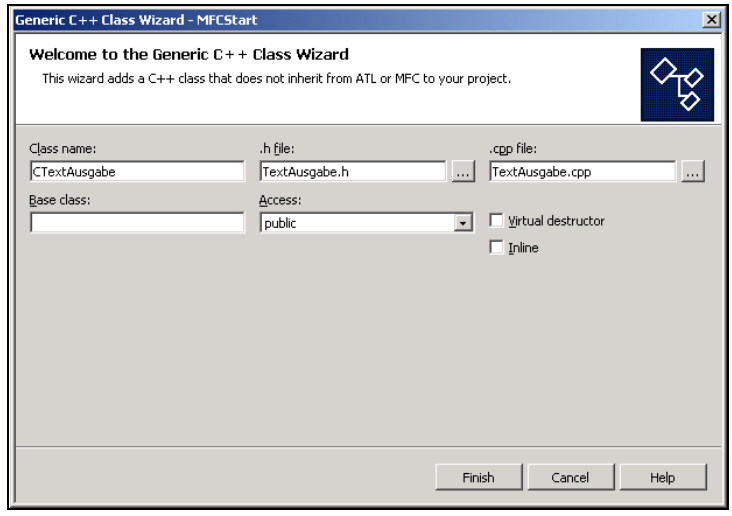


Abb. 3.14
Die Einstellmöglichkeiten im Klassenassistenten

Ein Objekt zur Textausgabe

In Ihrem aktuellen Projekt soll eine Textausgabe im Fenster getätigt werden. Zu Demonstrationszwecken soll das Objekt einer Klasse dafür verantwortlich sein, diesen Text bereitzustellen. Das ist vielleicht in diesem Rahmen etwas übertrieben, veranschaulicht aber die allgemeine Vorgehensweise beim assistentengestützten Anlegen neuer Klassen mit dem Visual Studio.

Klasse zur Textverwaltung für die erste Applikation

Name für die neue Klasse

Tragen Sie daher nun als Namen für die neue Klasse *CTextAusgabe* ein – Sie werden feststellen, dass die Bezeichnungen für die Header- und die Implementationsdatei automatisch ergänzt werden.

Abgeleitet werden soll unsere Klasse nicht, ebenso wenig ist sie als *Inline* zu spezifizieren oder mit einem virtuellen Destruktor zu versehen.

Anlegen der Klassendateien

Klicken Sie auf *Fertig stellen* und der Assistent legt die genannten Dateien an. Sie können sich durch einen schnellen Blick in den Solution Explorer davon überzeugen, dass sowohl die Header- als auch die Implementationsdatei in den dafür vorgesehenen Ordnern abgelegt wurden:

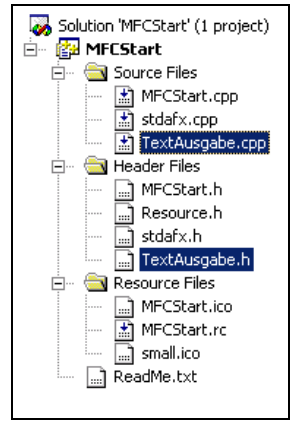


Abb. 3.15
Die neu hinzugefügten Dateien

Öffnen Sie durch Doppelklick auf die Dateinamen die beiden Files *Textausgabe.cpp* und *Textausgabe.h*. Es öffnen sich zwei neue Fenster, die die Inhalte der eben neu angelegten Files präsentieren.

Vielleicht sagt Ihnen die Art der Fensterverwaltung beim Visual Studio.NET nicht zu. Dieses ist aber kein Problem, denn Sie kann auf die vom Visual Studio 6 bekannte Variante umgestellt werden. Öffnen Sie hierzu das *Tools*-Menü und wählen Sie dort den Punkt *Optionen*:

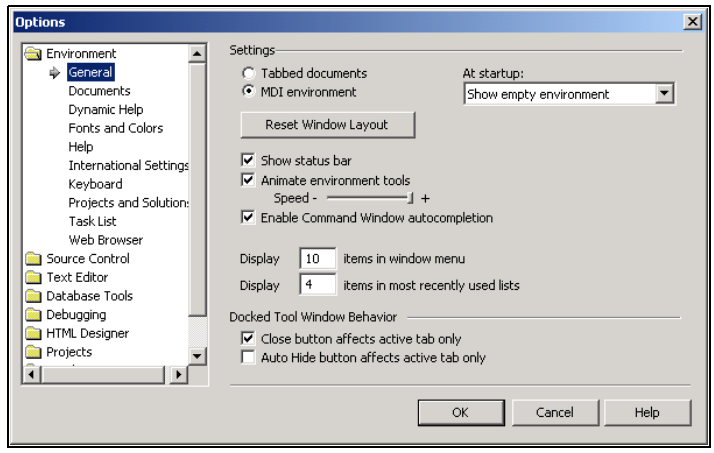


Abb. 3.16
Unter zahlreichen Optionen verbirgt sich auch die Darstellungsart der Fenster

Darstellungsoptionen

Bei den Einstelloptionen unter *Umgebung* > *Allgemein* können Sie zwischen *Tabbed Documents* und einer MDI-Umgebung wählen. Den Unterschied zwischen diesen beiden Formen zeigen die beiden folgenden Abbildungen auf:

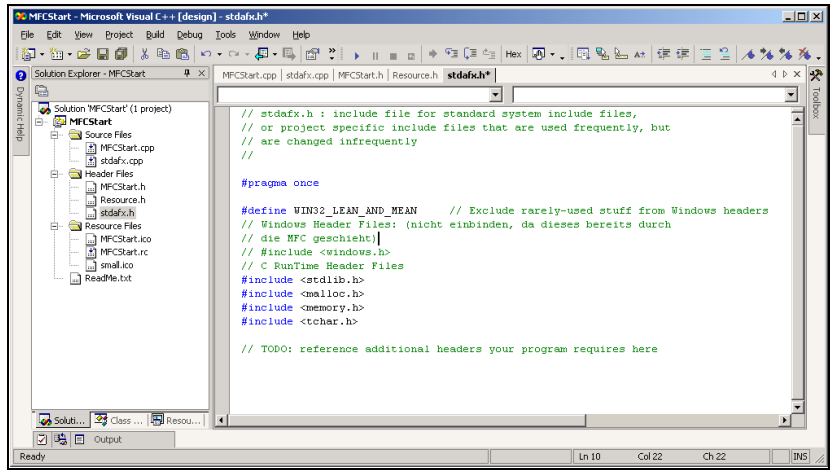


Abb. 3.17
Die voreingestellte *Tabbed Documents*-Variante

Wie Sie sehen, sind die Fenster bei der *Tabbed Windows*-Version in einer Übersichtsleiste am oberen Rand aufgeführt, während immer nur eins sichtbar im Vordergrund verweilt und dabei den gesamten zur Verfügung stehenden Raum einnimmt.

Die MDI-Variante stellt sämtliche geöffneten Fenster innerhalb des freien Quelltextbereichs dar – eventuell in minimierter Form als kleiner rechteckiger Kasten mit Titelleiste und Steuerelementen – und erlaubt das freie Verschieben innerhalb dieser Sektion.

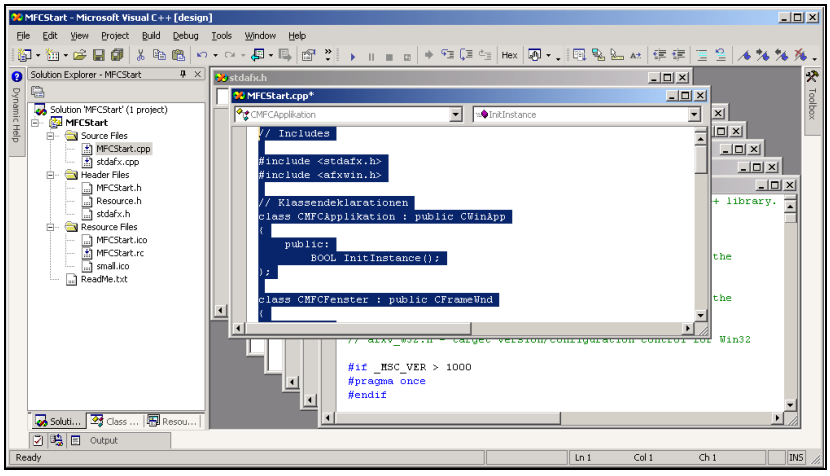


Abb. 3.18
Viele Entwickler empfinden die MDI-Darstellungsvariante als übersichtlicher

Inhalte der neuen Klassendateien

Die neu angelegten Klassen sind recht übersichtlich geraten und enthalten derzeit noch keine Funktionalität. Ergänzen Sie die Headerdatei *Textausgabe.h* wie folgt:

Listing 3.1
Textausgabe.h

```
#pragma once

const int MAXIMALE_STRINGLAENGE = 256;

class CTextAusgabe
{
public:
   CTextAusgabe(void);
   ~CTextAusgabe(void);

   void StringSetzen(char *in_lpszString);
   char *StringAuslesen();

private:
   char m_lpszAusgabe[MAXIMALE_STRINGLAENGE];
};
```

Zunächst fällt auf, dass die Datei mit einem *#pragma once* beginnt. Diese Direktive sorgt dafür, dass die Headerdatei nur einmal eingebunden wird.

#pragma once

Angenommen, Sie haben eine Datei *A.cpp*, welche die beiden Includefiles *B.h* und *C.h* einbindet. Wenn *B.h* nun *C.h* ebenfalls einbindet, würde *C.h* zweifach eingelesen und damit die Definitionen als doppelt angesehen werden.

#pragma once sorgt dafür, dass jede Headerdatei tatsächlich nur einmal eingelesen wird. Ist dieses bei einem Kompiliervorgang bereits der Fall gewesen, wird das Übersetzen der Datei übersprungen.

Die Klasse selbst bietet nichts Aufregendes: zwei Memberfunktionen zum Initialisieren und Auslesen eines Chararrays – diese werden im Laufe dieses Buchs häufig salopp als Strings bezeichnet werden – sowie das Array selbst, dessen Größe durch eine definierte Konstante festgelegt ist.

Die CTextausgabe-Implementationsdatei

Editieren Sie nun die Implementationsdatei *TextAusgabe.cpp*:

Listing 3.2 **TextAusgabe.cpp**

```
#include "StdAfx.h"
#include "textausgabe.h"

CTextAusgabe::CTextAusgabe(void)
{
   strcpy(m_lpszAusgabe, "String nicht
initialisiert!");
}

// CTextAusgabe::StringSetzen
//
// Setzt den auszugebenden String neu.
//
// Rückgabewerte: keine

void CTextAusgabe::StringSetzen(char *in_lpszString)
{
   if (NULL != in_lpszString)
   {
      if (strlen(in_lpszString) <
MAXIMALE_STRINGLAENGE)
      {
         strcpy(m_lpszAusgabe,
in_lpszString);
      }
      else
      {
         strcpy(m_lpszAusgabe, "Zu langer
            String uebergeben!");
      }
   }
   else
   {
      strcpy(m_lpszAusgabe, "NULL Zeiger uebergeben!");
   }
}
```

```
// CTextAusgabe::StringAuslesen
//
// Gibt den auszulesenden String aus.
//
// Rückgabewerte: Zeiger auf den auszulesenden,
// nullterminierten String

char *CTextAusgabe::StringAuslesen()
{
   return (m_lpszAusgabe);
}

CTextAusgabe::~CTextAusgabe(void)
{
}
```

Beschreibung der Implementation

Auch die Implementationsdatei ist recht einfach gehalten. Erwähnt werden muss jedoch gleich die erste Zeile *#include „StdAfx.h*": Sie dient zum Einbinden der für die vorkompilierten Header benötigten Includedatei.

Bedingungen für vorkompilierte Headerdateien

Wenn Sie ein Projekt mit vorkompilierten Headern bearbeiten, muss jede Implementationsdatei diesen Ausdruck (bzw. mit dem entsprechenden zur Vorkompilierung gehörenden Includedateinamen) als erste Zeile enthalten, da sonst der Übersetzungsvorgang direkt wieder abgebrochen wird.

Im Konstruktor der Klasse wird der auszugebende String auf einen initalen Wert gesetzt, der dafür sorgt, dass ein Entwickler es sofort bemerkt, wenn mit einem nicht initialisierten String gearbeitet wird.

Nicht initialisierte Werte und Debugging

Es ist gerade in der Windows-Entwicklung sehr wichtig, dass Sie sich selbst diese kleinen Tricks von Anfang an angewöhnen, da die Fehlersuche oft nur unter großem Aufwand betrieben werden kann.

Wenn Sie jedoch bei einigen Werten von Ihnen selbst verteilte Dummysymbole wieder erkennen, ist es oftmals eine Sache von Sekunden, Fehler zu berichtigen.

Es stehen weiterhin zwei Methoden zum Auslesen und Setzen des Ausgabestrings zur Verfügung, von denen Letztere wieder extensiven Gebrauch von Sicherheitsmaßnahmen gegen inkorrekte Eingabeparameter enthält.

Die ursprünglichen Dateien unter der Lupe

Neben den eben gerade eingefügten Dateien hatte ja bereits der Anwendungsassistent eine ganze Reihe von weiteren Files selbstständig erzeugt. Der folgende Abschnitt durchleuchtet im Schnelldurchlauf die Bedeutung der einzelnen Dateien:

Projektdatei

- *MFCStart.vcproj*: Die eigentliche Projektdatei für die MFC-Startanwendung. Es enthält grundlegende Informationen darüber, mit welcher Version des Visual Studio dieses Projekt angelegt wurde, sowie allgemeine Konfigurationseinstellungen, die während der Assistentenphase oder danach in den Projekteinstellungen getätigt wurden.

Ressourcen

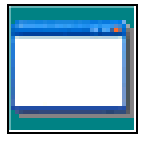
- *MFCStart.rc*: Diese Datei enthält eine Auflistung der *Ressourcen*, die vom Projekt *MFCStart* verwendet werden. Zu den Ressourcen einer Anwendung zählen unter anderem Icons, Bitmaps, Zeichensätze oder Mauszeiger, kurz alles, was nicht direkt im Programm als Quelltext gespeichert wird, sondern als externe Datenquelle eingebunden wird (die dann wiederum statisch zur Anwendung gebunden werden kann).

- *Resource.h*: Diese Datei hängt eng mit der Ressourcendatei zusammen und enthält IDs, die als konstante Platzhalter für die einzelnen Ressourcen eines Projekts stehen. Wenn Sie wissen, dass ein Knopf in einem Dialogfeld als ID die Konstante *ID_KNOPF* hat, können Sie ihn darüber direkt ansprechen und müssen nicht erst seine native Bezeichnung (eine positive ganze Zahl) heraussuchen.

- *MFCStart.ico*: Ein Icon (kleines Bild), das als Applikationssymbol verwendet wird (Größe: 32*32 Pixel). Diese Ressource wird über *MFCStart.rc* eingebunden.

- *Small.ico*: In der Regel das gleiche Bild wie MFCStart.ico, nur in einer kleineren (16*16) Version. Wird ebenfalls über *MFCStart.rc* eingebunden.

Abb. 3.19
Das Applikations-Icon



Vorkompilierte Header

- *StdAfx.h, StdAfx.cpp*: Dateien, die zum Erzeugen der vorkompilierten Headerfiles benötigt werden. Vorkompilierte Header (*precompiled headers*, kurz PCH) dienen, wie bereits angesprochen, zur Beschleunigung von Kompiliervorgängen.

Hauptprogramm

- *MFCStart.cpp*: Die Implementationsdatei, die das Hauptprogramm der Win32-Anwendung enthält. Das Ergebnis dieses Programms haben Sie bereits gesehen, hier nun der Quelltext:

Listing 3.3
Die Win32-Applikation als Quelltext

```
// MFCStart.cpp : Defines the entry point for the
// application.
//

#include "stdafx.h"
#include "MFCStart.h"
#define MAX_LOADSTRING 100

// Global Variables:
HINSTANCE hInst;

// current instance
TCHAR szTitle[MAX_LOADSTRING];
// The title bar text
TCHAR szWindowClass[MAX_LOADSTRING];

// the main window class name
// Forward declarations of functions included in this // code
module:

ATOM               MyRegisterClass(HINSTANCE hInstance);
BOOL               InitInstance(HINSTANCE, int);
LRESULT CALLBACK   WndProc(HWND, UINT, WPARAM,LPARAM);
LRESULT CALLBACK   About(HWND, UINT, WPARAM, LPARAM);

int APIENTRY _tWinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPTSTR    lpCmdLine,
                     int       nCmdShow)
{
   // TODO: Place code here.
   MSG msg;
   HACCEL hAccelTable;

   // Initialize global strings
   LoadString(hInstance, IDS_APP_TITLE,
szTitle, MAX_LOADSTRING);
   LoadString(hInstance, IDC_MFCSTART,
 szWindowClass, MAX_LOADSTRING);
   MyRegisterClass(hInstance);

   // Perform application initialization:
   if (!InitInstance (hInstance, nCmdShow))
   {
      return FALSE;
   }

   hAccelTable = LoadAccelerators(hInstance,(LPCTSTR)IDC_MFCSTART);
```

```
   // Main message loop:
   while (GetMessage(&msg, NULL, 0, 0))
   {
      if (!TranslateAccelerator(msg.hwnd,
hAccelTable, &msg))
      {
         TranslateMessage(&msg);
         DispatchMessage(&msg);
      }
   }

   return (int) msg.wParam;
}
```

Registrieren einer Fensterklasse

```
//  FUNCTION: MyRegisterClass()
//
//  PURPOSE: Registers the window class.
//
//  COMMENTS:
//
//    This function and its usage are only necessary
// if you want this code to be compatible with Win32
// systems prior to the 'RegisterClassEx'
// function that was added to Windows 95. It is
// important to call this function
// so that the application will get 'well formed'
// small icons associated
// with it.
//

ATOM MyRegisterClass(HINSTANCE hInstance)
{
   WNDCLASSEX wcex;

   wcex.cbSize = sizeof(WNDCLASSEX);

   wcex.style = CS_HREDRAW | CS_VREDRAW;
   wcex.lpfnWndProc = (WNDPROC)WndProc;
   wcex.cbClsExtra = 0;
   wcex.cbWndExtra = 0;
   wcex.hInstance = hInstance;
   wcex.hIcon = LoadIcon(hInstance,
(LPCTSTR)IDI_MFCSTART);
   wcex.hCursor = LoadCursor(NULL,IDC_ARROW);
   wcex.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
   wcex.lpszMenuName = (LPCTSTR)IDC_MFCSTART;
   wcex.lpszClassName = szWindowClass;
   wcex.hIconSm = LoadIcon(wcex.hInstance, (LPCTSTR)IDI_SMALL);
```

```
   return RegisterClassEx(&wcex);
}

//   FUNCTION: InitInstance(HANDLE, int)
//
//   PURPOSE: Saves instance handle and creates main
//   window
//
//   COMMENTS:
//
//        In this function, we save the instance
//    handle in a global variable and
//        create and display the main program window.
//
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
   HWND hWnd;

   hInst = hInstance; // Store instance handle in our
// global variable

   hWnd = CreateWindow(szWindowClass, szTitle,
WS_OVERLAPPEDWINDOW,
CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
 NULL, NULL, hInstance, NULL);

   if (!hWnd)
   {
      return FALSE;
   }

   ShowWindow(hWnd, nCmdShow);
   UpdateWindow(hWnd);

   return TRUE;
}

//  FUNCTION: WndProc(HWND, unsigned, WORD, LONG)
//
//  PURPOSE:  Processes messages for the main window.
//
//  WM_COMMAND- process the application menu
//  WM_PAINT- Paint the main window
//  WM_DESTROY- post a quit message and return
//
//
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
WPARAM wParam, LPARAM lParam)
```

Initialisierung

Die Nachrichten-verarbeitende Funktion

```
{
   int wmId, wmEvent;
   PAINTSTRUCT ps;
   HDC hdc;

   switch (message)
   {
   case WM_COMMAND:
      wmId    = LOWORD(wParam);
      wmEvent = HIWORD(wParam);
      // Parse the menu selections:
      switch (wmId)
      {
      case IDM_ABOUT:
         DialogBox(hInst,
(LPCTSTR)IDD_ABOUTBOX, hWnd,
 (DLGPROC)About);
         break;

      case IDM_EXIT:
         DestroyWindow(hWnd);
         break;
      default:
         return DefWindowProc(hWnd, message,
wParam, lParam);
      }
      break;
   case WM_PAINT:
      hdc = BeginPaint(hWnd, &ps);
      // TODO: Add any drawing code here...
      EndPaint(hWnd, &ps);
      break;
   case WM_DESTROY:
      PostQuitMessage(0);
      break;
   default:
      return DefWindowProc(hWnd, message,
 wParam, lParam);
   }
   return 0;
}
```

Nachrichtenverarbeitung für die Infobox

```
LRESULT CALLBACK About(HWND hDlg, UINT message,
WPARAM wParam, LPARAM lParam)
{
   switch (message)
   {
   case WM_INITDIALOG:
```

```
      return TRUE;

   case WM_COMMAND:
      if (LOWORD(wParam) == IDOK ||
LOWORD(wParam) == IDCANCEL)
      {
         EndDialog(hDlg, LOWORD(wParam));
         return TRUE;
      }
      break;
   }
   return FALSE;
}
```

Kurze Beschreibung des Inhalts von *MFCStart.cpp*

Trotz seiner Kürze hat es der entstandene Quelltext in sich. An dieser Stelle sollen gar nicht alle Details besprochen werden, schließlich soll zügig mit der Entwicklung einer MFC-Applikation begonnen werden.

Trotzdem ist es sinnvoll einmal kurz zu reflektieren was eingangs dieses Kapitels über die Windows-Programmierung gesagt wurde und wie die praktische Umsetzung im Rahmen der Win32-API-Programmierung aussieht – zumal in diesem Zusammenhang auch an einem konkreten Beispiel einige weitere Konzepte erläutert werden können.

Die WinMain

Fehlende ***main()***-Funktion

Zunächst fällt auf, dass das Programm offensichtlich keine *main()*-Funktion hat, wie Sie vermutlich von der Shell-Programmierung her bekannt sein dürfte. Unter Windows heißt das passende Gegenstück *WinMain*, oder, wie in diesem Fall *_tWinMain*. Sie stellt das Hauptprogramm jeder Windows-Applikation dar.

Listing 3.4
Prototyp der *WinMain*-Funktion

```
int APIENTRY _tWinMain(HINSTANCE hInstance,
            HINSTANCE hPrevInstance,
            LPTSTR    lpCmdLine,
int nCmdShow );
```

Die vier übergebenen Parameter sind wichtig zu kennen:

Tabelle 3.2
Die Parameter der Funktion *WinMain*

Parameter	Bedeutung
hInstance	Kennzahl für die aktuelle Programminstanz, wird automatisch von Windows beim Starten der Anwendung vergeben.
hPrevInstance	NULL (stammt aus Win16-Zeiten und wird nicht mehr verwendet)

Parameter	Bedeutung
lpCmdLine	Ein Zeiger auf die Kommandozeile, mit der das Programm aufgerufen wurde, ist analog zu argv aus main(int argc, char **argv).
nCmdShow	Gibt an, in welcher Form das Hauptfenster der Anwendung angezeigt werden soll (minimiert, Vollbild, Icon etc).

Wenn bei der Beschreibung oben von *Instanzen* geredet wird, meint man hiermit das im Speicher befindliche Abbild der zugrunde liegenden Applikationsdatei – hierbei handelt es sich normalerweise um EXE- Files.

Instanzen

Sie können eine (theoretisch) unbegrenzte Anzahl von Instanzen gleichzeitig starten, Windows kümmert sich um die Verwaltung und sorgt dafür, dass sich die einzelnen laufenden Anwendungen nicht gegenseitig in die Quere kommen – eine Ausnahme bilden Programme, die dazu gedacht sind, nebeneinander zu existieren und untereinander zu kommunizieren, etwa Hybridformen von Applikationen, die gleichzeitig Client- und Server bei Netzwerk-Anwendungen darstellen können.

Instanzen stehen im gleichen Verhältnis zu ihren Applikationsdateien wie Objekte zu Klassen.

Erzeugen des Fensters

Innerhalb von *WinMain* wird durch den sukzessiven Aufruf von *MyRegisterClass* (der Datentyp *ATOM* entspricht übrigens dem Datentyp *WORD*) und *InitInstance* zunächst eine so genannte Fensterklasse registriert und dann ein Fenster vom eben registrierten Typ erzeugt.

Fensterklassen

Fensterklassen charakterisieren den grundlegenden Typ eines Fensters, der für alle Fenster dieser Klasse identisch sein soll, während bei der eigentlichen Erzeugung eines Fensters die individuellen Eigenschaften (Größe, Position etc.) spezifiziert werden können. Für nähere Angaben zu *RegisterClassEx* und der verwendeten *WNDCLASSEX* schlagen Sie im Anhang dieses Buchs nach.

Was genau die einzelnen Strukturelemente der Fensterklasse bedeuten ist an dieser Stelle zunächst nicht weiter interessant – im Anhang finden Sie jedoch der Vollständigkeit halber eine Übersicht über die einzelnen Bedeutungen.

Die Nachrichtenschleife

Wenn Sie sich an den Anfang dieses Kapitels zurückerinnern, werden Sie die nun in *WinMain* folgende Kontrollstruktur als Nachrichtenschleife wiedererkennen.

Sie ist es, die das eigentliche Herzstück der Anwendung ausmacht: Solange die Applikation läuft (also nicht auf Wunsch des Anwenders oder des Betriebssystems geschlossen wurde) wird die Nachrichtenschleife ständig ausgeführt, um eingehende Botschaften zu verarbeiten.

Dabei läuft die Abarbeitung in drei Schritten ab:

Nachrichten abholen

1. Entgegennehmen der Botschaft: Dieses geschieht in diesem Fall durch ein ständiges Aufrufen von *GetMessage* (diese Methode liefert o zurück, sobald die Anwendung geschlossen werden soll) und wartet ansonsten solange, bis eine Nachricht eintrifft. Das Ganze passiert in Absprache mit dem Betriebssystem und ist sehr ressourcenschonend, wie wir bereits bei der Untersuchung des Task-Managers gesehen haben.

Nachrichten übersetzen

2. *TranslateMessage*: Eingegangene Nachrichten müssen erst in ein für Windows verständliches, internes Format gebracht werden, dessen Spezifikationen für uns an dieser Stelle nicht weiter interessant sind.

Nachrichten weiterleiten

3. *DispatchMessage*: Dieser Aufruf leitet die eingetroffene Botschaft an eine speziell auf die Nachrichtenverarbeitung fokussierte Methode weiter

Woher weiß Windows nun, wohin die Nachricht geleitet werden soll?

Nun, beim Anlegen der Fensterklasse wird unter anderem auch eine so genannte *Callbackfunktion* spezifiziert, die immer dann aufgerufen wird, sobald eine Nachricht verarbeitet werden soll (durch den *DispatchMessage* Aufruf).

Callback-Funktionen

Funktionen dieser Art werden von Windows direkt aufgerufen, nicht von unserer Applikation (es gibt keinen direkten Aufruf innerhalb unseres Quelltexts).

Man spricht auch von asynchroner Abarbeitung, womit man sich auf den Zeitpunkt des Aufrufs bezieht: dieser kann jederzeit stattfinden und ist vom Programmierer nicht vorhersehbar.

Die Nachrichtenfunktion

Die Nachrichtenfunktion für das erzeugte Fenster heißt *WndProc* (Window Procedure, also Fensterprozedur) und genügt dem nachstehenden Aufbau:

Listing 3.5
Prototyp von Nachrichtenfunktionen

```
LRESULT CALLBACK WndProc(HWND hWnd,
UINT message,
WPARAM wParam,
LPARAM lParam);
```

Tabelle 3.3
Parameter der Nachrichtenfunktion

Parameter	Bedeutung
hWnd	Handle des Fensters, das die Nachricht empfangen hat. Muss übergeben werden, da dieselbe Nachrichtenfunktion Basis für beliebig viele Fenster sein kann.
message	Die empfangene Botschaft in Form eines positiven, ganzzahligen Werts, zu dem auch eine passende Konstante existiert (WM_XXX).
wParam	1. Parameter mit zusätzlichen Informationen, trägt je nach Nachricht unterschiedliche Informationen oder ist gar nicht in Gebrauch.
lParam	2. Parameter mit zusätzlichen Informationen, trägt je nach Nachricht unterschiedliche Informationen oder ist gar nicht in Gebrauch.

Fensterhandles

Nachrichtenfunktionen erhalten als ersten Parameter ein Handle (zu deutsch ungefähr Griff) auf das Fenster, das die Nachricht erhalten hat. Stellen Sie sich Handles einfach als einen anderen Namen für ein Objekt vor.

Handles werden von Windows automatisch ergeben, meist liefern die erzeugenden Funktionen – zum Beispiel CreateWindow wie in diesem Programm – die jeweiligen Werte zurück. Handles sind ganzzahlige positive Ausdrücke und werden von Windows in einer internen Liste verwaltet.

Spricht ein Programmierer ein Objekt über sein Handle an, prüft Windows in dieser Liste, welches Objekt gemeint ist – oder ob es überhaupt existiert – und gestattet somit den indirekten Zugriff.

Fensterhandles sind in der Regel im Nachhinein nicht mehr aus den Objekten auszulesen, speichern Sie sie also immer, sobald sie von erzeugenden Funktionen zurückgeliefert werden.

Der Inhalt der Nachrichtenfunktion selbst ist dann wieder einfach zu verstehen – vorausgesetzt man kennt die Bedeutung der einzelnen Fensternachrichten sowie die ihrer Parameter.

Eine Übersicht über verfügbar Windows-Messages finden Sie im Anhang dieses Buches, die konkrete Beschreibung der Parameter schlagen Sie bequem in der MSVC++-Online-Hilfe nach.

Verfügbare Windows-Messages

Sie können sich an dieser Stelle eine kurze Lesepause gönnen und ein wenig mit dem Win32-Programm herumspielen. Es ist wichtig, dass Sie ein Gefühl dafür bekommen, wie Windows-Programme intern arbeiten, wie Nachrichten versendet werden und wie auf sie reagiert wird.

Vielleicht schlagen Sie im Anhang dieses Buchs ein paar interessante Fenstermachrichten nach und testen aus, ob diese vom oben erzeugten Fenster verarbeitet werden können.

Aus Win32 wird MFC

Nach diesen Vorbetrachtung anhand eines einfaches Win32-Programms ist es nun an der Zeit, die eigentliche MFC-Applikation zu entwickeln. Öffnen Sie hierzu zunächst die Datei *StdAfx.h* und editieren Sie sie wie folgt:

Listing 3.6
Die veränderte *StdAfx.h*

```
// stdafx.h : include file for standard system include
// files, or project specific include files that are
// used frequently, but
// are changed infrequently

#pragma once

// Exclude rarely-used stuff from Windows headers
#define WIN32_LEAN_AND_MEAN

// Windows Header Files: (nicht einbinden, da dieses
// bereits durch die MFC geschieht)
// #include <windows.h>
// C RunTime Header Files
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include <tchar.h>

// TODO: reference additional headers your program
// requires here
```

Achten Sie insbesondere darauf, dass die Zeile *#include <windows.h>* auskommentiert wird, da es ansonsten Probleme mit den im Folgenden verwendeten MFC-Klassen gibt, die ihrerseits auch bereits *windows.h* einbinden – da die Applikation ursprünglich eine Win32-Anwendung ist, hat der Assistent diese Zeile nicht schon selbst weggelassen.

In *StdAfx.h* werden eine Reihe von Includedateien eingebunden, die in nahezu jedem Projekt benötigt werden.

Die neue Implementationsdatei

Öffnen Sie als Nächstes die *MFCStart.cpp* Datei und löschen Sie ihren kompletten Inhalt (oder kommentieren Sie ihn alternativ aus) – er wird nicht mehr benötigt. Tippen Sie statt dessen die folgenden Zeilen ein (alternativ befindet sich die veränderte Datei auch auf der dem Buch beiliegenden CD unter *\Kapitel3\MFCStart\MFCStart.cpp*):

Listing 3.7
Ein erster MFC-Test

```
// Includes
#include <stdafx.h>
#include <afxwin.h>

// Klassendeklarationen
class CMFCApplikation : public CWinApp
{
   public:
      BOOL InitInstance();
};

class CMFCFenster : public CFrameWnd
{
   public:
      CMFCFenster();

   protected:
      afx_msg void OnPaint();
      DECLARE_MESSAGE_MAP()
};

// globale Variablen
CMFCApplikation dieApplikation;

// die Nachrichtentabelle
BEGIN_MESSAGE_MAP(CMFCFenster, CFrameWnd)
   ON_WM_PAINT()
END_MESSAGE_MAP()

// CMFCApplikation::InitInstance()
//
// InitInstance initialisiert die Anwendung
// (Fenstererzeugung
// und -darstellung)
//
// Rückgabewerte: true -> Kein Fehler
```

```
BOOL CMFCApplikation::InitInstance()
{
   m_pMainWnd = new CMFCFenster;

   m_pMainWnd->ShowWindow(m_nCmdShow);
   m_pMainWnd->UpdateWindow();

   return (true);
}

// CMFCFenster() [Konstruktor]
//
// Der Konstruktor der CMFCFenster Klasse erzeugt ein
// neues Fenster mit dem Text "Ein kleiner MFC Test"
// in der Titelzeile
//
// Rückgabewerte: keine

CMFCFenster::CMFCFenster()
{
   Create(NULL, "Ein kleiner MFC Test");
}

// CMFCFenster::OnPaint
//
// Die OnPaint Methode der Klasse CMFCFenster
// behandelt eingehende WM_PAINT Nachrichten und
// stellt den Inhalt der Fensters wieder her.
//
// Rückgabewerte: keine

void CMFCFenster::OnPaint()
{
   CPaintDC dc(this);

   RECT rect;
   dieApplikation.m_pMainWnd->GetClientRect(&rect);
   dc.DrawText("Dieser Text soll zentriert
ausgegeben werden", -1, &rect,
DT_SINGLELINE|DT_CENTER|DT_VCENTER;
}
```

Übersetzen des Programms

Da der Anwendungsassistent die Applikation ursprünglich als reine Win32-Anwendung ausgelegt hatte, ist ein Einsatz der MFC nicht vorgesehen.

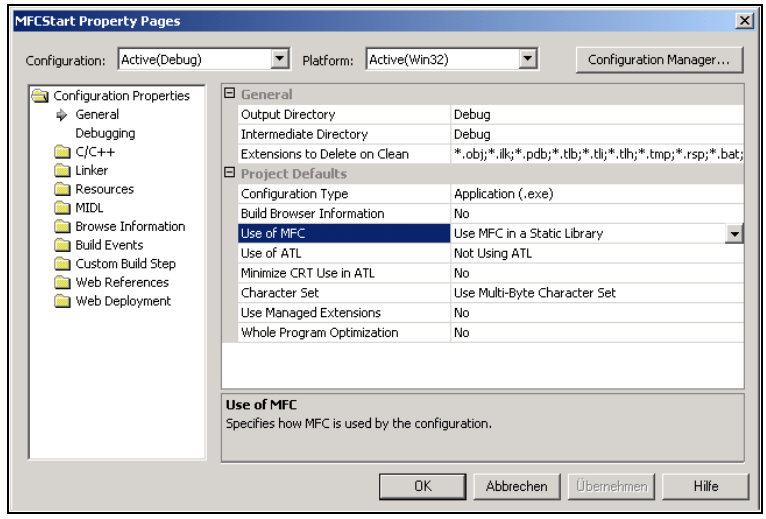
Sicherstellen einer erfolgreichen Übersetzung

Sie müssen selbst manuell dafür sorgen, dass die notwendigen Bibliotheken eingebunden werden, sonst kommt es zu Linkerfehlern.

Wählen Sie im Solution Explorer das *MFCStart*-Projekt aus, aktivieren Sie dessen Kontextmenü (rechte Maustaste) und wählen Sie den Punkt *Einstellungen*.

Es öffnet sich ein Dialogfeld ähnlich dem folgenden:

Abb. 3.20
Verwendung der MFC aktivieren



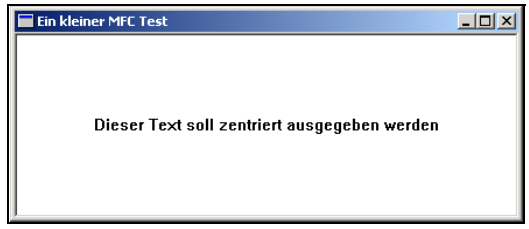
Stellen Sie die Verwendung der MFC wie im Bild angezeigt auf *Verwendung der MFC in einer statischen Bibliothek* und bestätigen Sie durch einen Klick auf *OK*.

Kompilieren des Projekts

Das Projekt sollte jetzt kompilierbar (F7) sein. Ist das nicht der Fall, vergleichen Sie es mit dem auf der Buch-CD befindlichen Projekt (*\Kapitel3\MFCStart*), um herauszufinden, welche Stellen falsch sind. Vermutlich handelt es sich nur um einen kleinen Fehler, doch gerade durch die lernt man sein Handwerkszeug richtig kennen.

Wenn der Übersetzungsvorgang erfolgreich beendet werden konnte, starten Sie das Programm (F5):

Abb. 3.21
Die erste MFC-Anwendung



Beschreibung des MFC-Programms

Nachdem Sie nun Ihr erstes MFC-Programm geschrieben haben, drängt sich Ihnen sicherlich die Frage auf, was genau im Einzelnen eigentlich da von Ihnen in Form von Quelltext entworfen wurde.

Vergleich mit dem Win32-Programm

Am einfachsten kann man diese Inhalte erklären, wenn man versucht, Analogien zum Win32-Beispiel zu finden und anhand dieser Gemeinsamkeiten und Unterschiede herauszukristallisieren.

Im vorangegangenen Abschnitt hatte die Betrachtung des Win32-Programms mit einer Untersuchung der *WinMain*-Funktion begonnen. Diese stellte das Hauptprogramm der Anwendung dar, erzeugte das Hauptfenster und enthielt die bereits mehrfach besprochene Nachrichtenschleife.

Das MFC-Anwendungsgerüst

Wenn Sie Ihren Blick über die MFC-Variante der Applikation schweifen lassen, wird Ihnen auffallen, dass keine explizit angegebene *WinMain*-Funktion existiert.

Verstecke *WinMain*

Das heißt nicht etwa, dass im Rahmen von MFC-Anwendungen andere Regeln gelten, vielmehr ist es so, dass die *WinMain*-Funktion implizit im so genannten MFC-Anwendungsgerüst (engl. *Application Framework*) verborgen ist.

Dieses Framework stellt die Basis der Entwicklung von Windows-Anwendungen dar, mit denen wir uns im weiteren Verlauf dieses Buchs noch genauer auseinandersetzen wollen.

Enthalten in ihm sind zahlreiche Funktionen, Klassen und Strukturen, die jede für sich Teilkomponenten bei der Entwicklung von fensterorientierten Programmen zur Verfügung stellen.

Gerade die einsatzbereiten Klassen des Anwendungsgerüsts sind jedoch nicht als Universalhilfsmittel zu bestimmten Problemen zu verstehen, sondern vielmehr als Ausgangspunkte für die Entwicklung von eigenen Lösungsansätzen zu spezifizierten Aufgaben.

Verwenden bereitgestellter Klassen

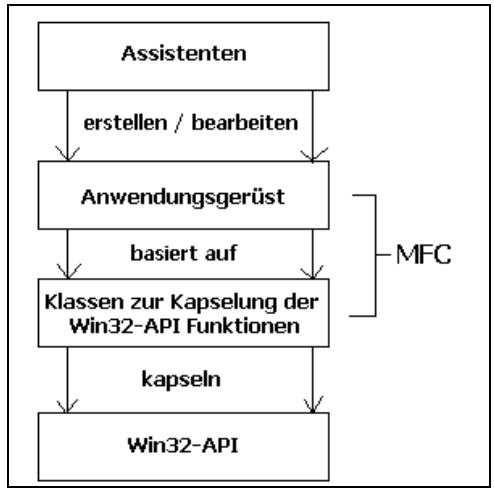
Benötigen Sie beispielsweise einen Dialog, der eine Sicherheitsabfrage – etwas in der Art wie *Möchten Sie den Vorgang wirklich fortsetzen?* – enthält, greifen Sie auf eine der bereits bestehenden Dialogklassen des Frameworks zurück und ergänzen Sie lediglich um die gerade benötigten Inhalte.

Die gesamte Handhabung des Dialogs hingegen überlassen Sie den vorgefertigten Klassen des MFC-Anwendungsgerüsts.

Einordnung des Anwendungsgerüsts

Das Anwendungsgerüst steht nun nicht frei im Raum, sondern stellt eine Komponente in einer Kette von Abstraktionsstufen dar. Im Kontext des Visual Studio, könnte man das Framework so einordnen, wie es in der nachstehenden Abbildung gezeigt wird:

Abb. 3.22
Das Anwendungsgerüst im Kontext



In diesem Schaubild unten beginnend finden Sie die Win32-API, mit der Sie schon kurzen Kontakt im einführenden Beispiel dieses Kapitels hatten. Die API umfasst sämtliche Funktionen und Strukturen, die zum Entwickeln von Windows-Programmen benötigt werden.

Nachteile prozeduraler APIs

Soweit so gut, doch hat sie für sich genommen einen entscheidenden Nachteil: Sie ist als prozedurale API nur noch schwer in den heutigen Kontext objektorientierter Sprachen wie C++ einzuordnen und scheint sich den modernen Programmierformen verschließen zu wollen.

MFC als objektorientierte Kapselung der Win32-API

An dieser Stelle treten die MFC auf den Plan und bieten eine Reihe von Klassen zur Kapselung der Win32-Funktionalität auf.

Als C++-Entwickler ist Ihnen das Prinzip sicherlich bekannt: anstatt von Grund auf neue Methoden und Klassen mit eigenständiger Funktionalität zu entwerfen, nimmt man häufig ein bestehendes System und bastelt lediglich deren Funktionalität zur Verfügung stellende Klassensysteme, die intern wiederum auf die alte Basis zugreifen.

Abstraktionsschichten

Ein gutes Beispiel für diese Abstraktionsschichten (engl. Abstraction Layers) sieht man an Projekten, die für unterschiedliche Plattformen (zum Beispiel Windows und X-Windows) entwickelt werden.

Um möglichst wenig Code neu schreiben zu müssen, wenn es um die Portierung von einem Endgerätetyp auf einen anderen geht, ist es wichtig, spezifische Funktionalitäten wie etwas das Erzeugen von Fenstern nur an zentralen Stellen innerhalb der Quelltexte zu verwenden, sodass nur diese konkreten Codefragmente umgeschrieben werden müssen.

Konkret bedeutet dieses meist, dass eine Schnittstelle entwickelt wird, die häufig als abstrakte Basisklasse (zum Beispiel IFenster*)entworfen ist und deren abgeleitete Klassen (in diesem Fall vielleicht* CWindowsFenster *und* CXWindowsFenster*) die für die jeweiligen Plattformen spezifischen Methoden definieren.*

Im Programm selbst, wenn es um den Zugriff auf ein Fenster geht, könnte man dann einfach einen Zeiger auf ein Objekt vom Typ IFenster *(Schnittstellen erhalten normalerweise das Präfix I für Interface) haben, das je nach Plattform auf ein Objekt vom Typ* CWindowsFenster *oder* CXWindows-Fenster *verweist.*

Heißt dieser Zeiger pFenster*, kann im Quelltext beispielsweise bequem per* pFenster>SchliesseFenster() *das Schließen des Fensters veranlasst werden, ohne sich an dieser Stelle Gedanken darüber machen zu müssen, ob die ausführende Basis Windows oder X-Windows heißt.*

Kleine Geschichte der MFC

Die MFC traten das erste mal im Jahre 1992 zur Erscheinung und haben seitdem einen expansiven Wandel hinter sich gebracht. Ausgehend von einer handvoll Klassen zur Verwaltung von Fenstern, stellen Sie jetzt tatsächlich ein komplettes Anwendungsgerüst zur Verfügung, das neben Kernpunkten wie der Dokument-/Ansichtarchitektur, auf die im weiteren Verlauf dieses Buchs noch eingegangen wird, auch eine Vielzahl an Hilfsklassen zur Verwendung von Zeichenketten, verketteten Listen und dergleichen mehr bietet.

Insgesamt umfassen die MFC mittlerweile über 200 Klassen, die Ihnen als Entwickler zur Verfügung stehen. Die Gesamtheit der Klassen wird dann als MFC-Anwendungsgerüst bezeichnet.

Assistenten als Hilfsmittel

Im Schaubild sind des Weiteren die so genannten Assistenten aufgeführt. Sie kenne diese sicherlich aus anderen Windows-Programmen und auch im Einführungskapitel haben wir bereits eine MFC-Anwendung erzeugt um zu sehen,

welche Möglichkeiten sich hier bieten und uns von der Leistungsfähigkeit der kleinen Helfer zu überzeugen.

Fleißige Helfer

Ein Assistent erzeugt nach Angaben des Benutzers, der diese für gewöhnlich in einer Reihe von Optionsdialogen seinen derzeitigen Bedürfnissen gemäß tätigt, ein Grundgerüst, das eine komplett funktionierende Anwendung bereitstellt und danach zur weiteren Bearbeitung als vollständiger Quelltext zur Verfügung steht (neben Anwendungen können auch weitere Komponenten wie DLLs mithilfe von Assistenten erzeugt werden).

In diesem Kapitel wird allerdings noch Hand angelegt, damit Sie sich später in den automatisch erzeugten Quelltexten schneller zurechtfinden.

Die WinMain in einer MFC-Applikation

Wie bereits angesprochen, ist keine *WinMain*-Funktion in den eben geschriebenen MFC-Quelltexten aufzufinden. Das liegt nicht etwa daran, dass sie nicht existiert, sondern vielmehr an dem Umstand, dass sie vor den Augen des Entwicklers versteckt wird.

Ihr Rumpf sieht im Quelltext so aus:

Listing 3.8 ***WinMain* in der MFC**

```
return AfxWinMain(hInstance, hPrevInstance, lpCmdLine,
 nCmdShow);
```

Wie Sie sehen, wird hier lediglich ein Aufruf von *AfxWinMain* getätigt, das in der Folge eine ganze Reihe von Initialisierungen und Aktionen durchführt:

Listing 3.9 ***AfxWinMain* im Detail**

```
int AFXAPI AfxWinMain(HINSTANCE hInstance,
HINSTANCE hPrevInstance,
            LPTSTR lpCmdLine,
int nCmdShow)
{
   ASSERT(hPrevInstance == NULL);

   int nReturnCode = -1;
   CWinThread* pThread = AfxGetThread();
   CWinApp* pApp = AfxGetApp();

   // AFX internal initialization
   if (!AfxWinInit(hInstance, hPrevInstance,
lpCmdLine, nCmdShow))
            goto InitFailure;

   // App global initializations (rare)
   if (pApp != NULL && !pApp->InitApplication())
      goto InitFailure;

   // Perform specific initializations
```

```
   if (!pThread->InitInstance())
   {
      if (pThread->m_pMainWnd != NULL)
      {
         TRACE0("Warning: Destroying non-
NULL m_pMainWnd\n");

pThread
->m_pMainWnd>DestroyWindow();
      }
      nReturnCode = pThread->ExitInstance();
      goto InitFailure;
   }
   nReturnCode = pThread->Run();

InitFailure:
#ifdef _DEBUG
   // Check for missing AfxLockTempMap calls
   if (AfxGetModuleThreadState()->m_nTempMapLock
!= 0)
   {
      TRACE1("Warning: Temp map lock count non-
zero (%ld).\n",
         AfxGetModuleThreadState()
->m_nTempMapLock);
   }
   AfxLockTempMaps();
   AfxUnlockTempMaps(-1);
#endif

   AfxWinTerm();
   return nReturnCode;
}
```

Das sieht nicht nur kompliziert aus, dass ist es auch. Zum Glück ist es nicht unbedingt erforderlich, jeden einzelnen Schritt nachvollziehen zu können, es ist nur wichtig zu wissen, wie generell der Ablauf aussieht, vom Starten des Programms bis zum Einstiegspunkt in das, was wir selbst geschrieben haben.

Nachdem eine Reihe von MFC-Anwendungsgerüst spezifischen Initialisierungen durchgeführt worden sind (*Afx* steht hier übrigens immer für Application Framework(s)), weiß das Framework schon einmal, mit welcher Anwendung es zu tun hat.

Das resultiert daraus, dass im MFC-Programm, dessen Quelltext Sie eingetippt haben, ein globales Objekt namens *dieApplikation* angelegt wurde:

Listing 3.10
Das Anlegen des globalen Anwendungsobjekts

```
CMFCApplikation dieApplikation;
```

Dieses Objekt wird als *Anwendungsobjekt* bezeichnet und stellt das einzige globale Objekt Ihrer gesamten Applikation dar. Warum ist es so wichtig, und woher weiß das MFC-Anwendungsgerüst das? Immerhin können Sie viele globale Objekte erzeugen, die für das Framework nicht relevant sind.

Die Applikationsklasse

Nun, *CMFCApplikation* ist von *CWinApp* abgeleitet worden, das wiederum von *CWinThread* abgeleitet ist. Geeignete Initialisierungen durch die Konstruktoren der Basisklassen bereiten dieses Objekt darauf vor, eine wichtige Rolle im restlichen Verlauf der MFC-Sources zu spielen.

Insbesondere stellt die Definition also sicher, dass *AfxWinMain* mit dem richtigen Ausgangsobjekt arbeitet und entsprechende Initialisierungen nicht im Sande verlaufen.

dieApplikation ist abgeleitet von einer Klasse *CWinThread* – von dieser stammt auch das in *AfxWinMain* verwendete Objekt *pThread* ab.

MFC-Klassenhierarchie

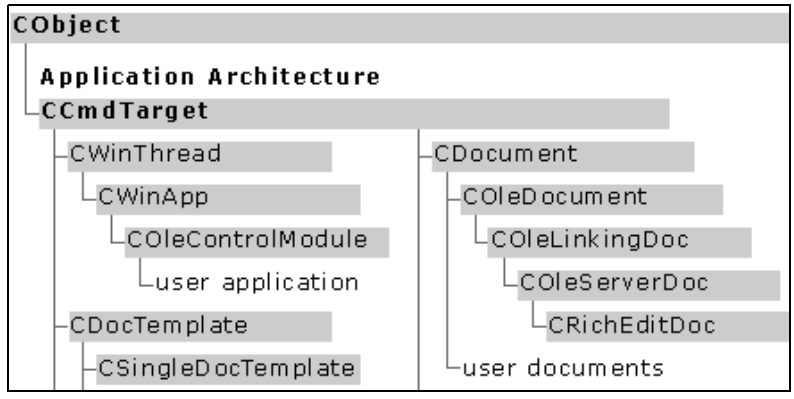
Dieses ist gerade der richtige Zeitpunkt, einen Blick auf die Klassenhierarchie der MFC zu werfen, die relativ strikt ausgelegt ist. Es gibt nämlich keine Mehrfachvererbung und sämtliche Klassen sind von einer Oberklasse *CObject* abgeleitet – dieser Umstand macht die Verwaltung von Zeigern auf Klassen natürlich recht angenehm.

Verwendung von virtuellen Methoden

Die virtuellen Methoden der rein abstrakten Basisklasse *CObject* werden dann geeignet in den abgeleiteten Klassen überschrieben.

Ein Auszug aus der MFC-Klassenhierarchie stellt die folgende Abbildung dar, hier sind insbesondere auch *CWinThread* und *CWinApp* aufgeführt. Beachten Sie auch die klare, übersichtliche Baumstruktur der einzelnen Unterklassen:

Abb. 3.23
Ein Auszug der MFC-Klassenhierarchie



Da Sie bei der MFC-Entwicklung häufig mit der Hierarchie herumhantieren werden, sollten Sie sich dieser Art der Klassendarstellung genau ansehen und nachvollziehen, in der Online-Hilfe beispielsweise sind Verwandtschaftsverhältnisse von Klassen auch in dieser Form bis hinauf zu *CObject* aufgeführt.

Aufruf der eigenen Quelltextpassagen

Zurück zum MFC-Quelltext. Wir kommen der Sache jetzt schon näher: nach den nötigen Initialisierungsschritten wurde ein Objekt vom Typ *CMFCApplikation* geeignet vorbereitet und relevante Zeiger – hier eben insbesondere *pThread* – sind geeignet initialisiert.

Es folgt ein Aufruf, der Sie aufhorchen lassen sollte:

```
pThread->InitInstance()
```

Listing 3.11
Aufruf der virtuellen Funktion *InitInstance*

Wie Sie sicher bemerkt haben, ist die *InitInstance*-Methode von Ihnen in der Klasse CMFCApplikation überschrieben worden. Hier noch einmal der zugehörige Quelltext:

Listing 3.12
CMFCApplikation::InitInstance

```
// CMFCApplikation::InitInstance()
//
// InitInstance initialisiert die Anwendung
// (Fenstererzeugung
// und -darstellung)
//
// Rückgabewerte: true -> Kein Fehler

BOOL CMFCApplikation::InitInstance()
{
   m_pMainWnd = new CMFCFenster;

   m_pMainWnd->ShowWindow(m_nCmdShow);
   m_pMainWnd->UpdateWindow();

   return (true);
}
```

Hier haben Sie es also mit der ersten selbst geschriebenen MFC-Methode zu tun, die vom Framework aus aufgerufen wird – mal abgesehen von der Definition des Anwendungsobjekts.

Erzeugen eines Fensters mit den MFC

Erinnern Sie sich noch, was in der *WinMain* der Win32-Anwendung als Erstes geschah?

Erzeugen des Anwendungsfensters

Richtig, das Anwendungsfenster wurde erzeugt, zunächst über die Registrierung einer passenden Fensterklasse, dann über den darauf abgestimmten Aufruf von *CreateWindow*().

Das *InitInstance* der MFC-Anwendung – und genauer der Klasse *CMFCApplikation* – erfüllt in diesem Fall keinen anderen Zweck; doch anstelle einen umfangreichen Registrierungsvorgang durchzuführen, finden wir nur eine einfache, lapidare Zeile vor:

Listing 3.13 **Das Erzeugen des Hauptfensters**

```
m_pMainWnd = new CMFCFenster;
```

Das Anlegen des neuen Objekts hat freilich das Aufrufen dessen Konstruktors zur Folge, der ebenfalls recht übersichtlich aufgebaut ist:

Listing 3.14 **Der Fenstererzeugungsaufruf**

```
Create(NULL, "Ein kleiner MFC Test");
```

Um zu verstehen, was hier passiert, muss man zunächst wissen, dass *CMFCFenster* von *CFrameWnd* abgeleitet ist. Diese Klasse stellt die Funktionalität einer Windows-SDI-Anwendung (also Verwaltung von je einem Dokument zur Zeit) oder eines Popup-Fensters zur Verfügung.

Initialisiert wird sie durch den Aufruf von *Create* – der Prototyp dieser Funktion ist im Folgenden abgedruckt:

Listing 3.15 **Prototyp *CFrameWnd::Create***

```
virtual BOOL Create
(
   LPCTSTR lpszClassName,
   LPCTSTR lpszWindowName,
   DWORD dwStyle = WS_OVERLAPPEDWINDOW,
   const RECT& rect = rectDefault,
   CWnd* pParentWnd = NULL,
   LPCTSTR lpszMenuName = NULL,
   DWORD dwExStyle = 0,
   CCreateContext* pContext = NULL
);
```

Die Parameter entscheiden dabei darüber, welches Aussehen das entstehende Fenster haben wird. Ihre Bedeutung im Einzelnen:

Tabelle 3.4 **Parameter**

Parameter	Bedeutung
lpszClassName	Der Klassenname für die zu verwendende, registrierte Fensterklasse (siehe auch *RegisterClass*). Wird kein Name angegeben, werden die *CFrameWnd*-Standardattribute für das Erzeugen des Fensters verwendet.
lpszWindowName	Name der zu erzeugenden Fensters, wird in der Titelleiste ausgegeben

Parameter	Bedeutung
dwStyle	Spezifziert die Windows-Style-Attribute. Für eine Übersicht über die zur Verfügung stehenden Attribute schauen Sie in den Anhang dieses Buchs.
rect	Rechteck, das die Größe und Position des Fensters spezifiziert. Wird *rectDefault* angegeben, gestatten Sie Windows, diese Daten selbstständig zu setzen.
pParentWnd	Gibt das Elternfenster für das zu erzeugende Fenster an. Sollte bei Toplevel-Fenstern auf NULL gesetzt werden.
lpszMenuName	Gibt den Name der zu verwendenden Menü-Ressource an.
dwExStyle	Spezifiziert die erweiterten Stilattribute für dieses Fenster. Für eine Übersicht der zur Verfügung stehenden Attribute schauen Sie in den Anhang dieses Buchs.
pContext	Zeiger auf eine *CcreateContext*-Struktur, darf auf NULL gesetzt werden. Eine Übersicht über diese Struktur finden Sie im Anhang dieses Buchs.

Analogien zwischen Win32-API und MFC

Das Erzeugen läuft also prinzipiell analog zum Win32-API-Pendant. Das ist natürlich auch einsichtig, denn die MFC kapseln ja nur deren Funktionalität. Insbesondere können Sie auch direkt auf Win32-Funktionen zugreifen – also sozusagen an den Möglichkeiten des MFC Frameworks vorbei – was allerdings in den meisten Fällen nicht unbedingt erforderlich oder ratsam ist.

Die Nachrichtenbehandlungsmethoden

Sie haben im Win32-Beispiel gesehen, wie eine Methode zum Behandeln von Nachrichten auszusehen hat. Dort werden eingehende Botschaften auf Ihren Typ hin geprüft und dann in einem großen Switch-Konstrukt passende Antwortbehandlungen durchgeführt.

Obwohl in der MFC-Variante diese Nachrichtenschleife nicht aufzufinden ist, können Sie doch eine verdächtig wirkende Zeile in der Klassendeklaration von *CMFCFenster* ausmachen:

```
DECLARE_MESSAGE_MAP()
```

Listing 3.16
Makro zum Anzeigen, dass eine Nachrichtentabelle existiert

Hierhinter verbirgt sich ein Makro, das, kurz gesagt, darauf hindeutet, dass irgendwo im Quelltext eine Übersicht über diejenigen Nachrichten versteckt ist, die von der umschließenden Klasse behandelt werden sollen.

Nachrichtentabelle

Sie ahnen es sicherlich schon, bei dieser Übersicht handelt es sich um die nachstehend noch einmal wiedergegebenen Zeilen:

Listing 3.17 **Eine Nachrichtentabelle mit einer zu behandelnden Nachricht**

```
BEGIN_MESSAGE_MAP(CMFCFenster, CFrameWnd)
   ON_WM_PAINT()
END_MESSAGE_MAP()
```

Die Message Map (Nachrichtentabelle) enthält sämtliche Fensternachrichten, auf die reagiert werden soll. Sie beginnt mit einem *BEGIN_MESSAGE_MAP*-Makroaufruf, der in Klammern zuerst den Klassennamen, zu dem diese Nachrichtentabelle gehört und dann den Namen der Klasse, von der diese wiederum abgeleitet ist, aufführt.

Es folgen die zu behandelnden Fensternachrichten, den Abschluss bildet das Makro *END_MESSAGE_MAP*.

Was inhaltlich hinter den Makros steht, ist an dieser Stelle nicht weiter interessant, Sie können sich bei Interesse die Implementationen in der schon angesprochenen Datei *afxwin.h* anschauen.

MFC-Nachrichtenmakros

Festlegen der zu behandelnden Nachrichten in Nachrichtentabellen

In der Nachrichtentabelle sind nun auch die einzelnen zu behandelnden Nachrichten durch Makrodefinitionen einzutragen. *ON_WM_PAINT*, wie im obigen Beispiel, bedeutet, wie eine *WM_PAINT* Nachricht behandelt werden soll.

Wie? Das ist das durch das *ON_WM_PAINT*-Makro bereits festgelegt: es wird eine Methode *OnPaint* in der Klasse *CMFCFenster* erwartet.

Diese Methoden- und Makronamen sind Vorgaben, die die MFC-Entwickler für sinnvoll hielten – es bleibt nichts anderes übrig, als sich an diese Richtlinien zu halten.

Das ist aber nicht weiter problematisch, da ein Großteil der Eintragungen in Nachrichtentabelle später von den Assistenten selbstständig durchgeführt werden – nur in Ausnahmefällen muss der Entwickler selbst Hand anlegen und hat es dann meistens mit einfach zu verstehenden Makros zu tun.

Die *OnPaint*-Methode

Nachdem Sie festgestellt haben, dass eine OnPaint-Methode zur Behandlung der WM_PAINT Nachricht zum Neuzeichnen eines Fensters benötigt wird, werden Sie sich vielleicht daran erinnern, eine solche auch zur Verfügung gestellt zu haben.

Sie ist in CMFCFenster wie folgt deklariert und implementiert:

```
afx_msg void OnPaint();
```

Listing 3.18
Deklaration von *OnPaint*

Die Methode *OnPaint* mit dem Rückgabetyp *void* ist mit einem weiteren Modifikator *afx_msg* versehen. Dieser hat allerdings keine Bewandnis, denn er ist als leerer String definiert.

Das *afx_msg*-Präfix

Es ist trotzdem Sitte, bei Behandlungsmethoden dieses Kürzel vorne vor die eigentliche Deklaration zu stellen, man erkennt damit sehr schnell, welchen Zweck eine Methode hat – wir haben es hier also mit der gleichen zugrunde liegenden Überlegung zu tun, wie es allgemein auch mit der ungarischen Notation der Fall ist.

Die Implementation von OnPaint sieht wie folgt aus:

Listing 3.19
Implementation von *OnPaint*

```
// CMFCFenster::OnPaint
//
// Die OnPaint Methode der Klasse CMFCFenster
// behandelt eingehende WM_PAINT Nachrichten und
// stellt den Inhalt der Fensters wieder her.
//
// Rückgabewerte: keine

void CMFCFenster::OnPaint()
{
   CPaintDC dc(this);

   RECT rect;
   dieApplikation.m_pMainWnd->GetClientRect(&rect);
   dc.DrawText("Dieser Text soll zentriert
ausgegeben werden", -1, &rect,
DT_SINGLELINE|DT_CENTER|DT_VCENTER;
}
```

Was hier inhaltlich geschieht, soll an dieser Stelle nicht weiter ausgeführt werden, da Sie diesbezüglich in den folgenden Kapiteln noch einiges über die hier verwendeten Gerätekontexte erfahren werden.

Kurz gesagt, wir die Größe des Fensters ermittelt und dann ein Text exakt in dessen Mitte ausgegeben.

Die Nachrichtenfunktion

Nachrichtenfunktion der Anwendung

Das gesamte MFC-Listing ist jetzt besprochen worden, doch fehlt derzeit noch ein ganz elementarer Bestandteil jeder Windows-Anwendung: die Nachrichtenfunktion zur Entgegennahme und Weiterleitung von Botschaften.

Hier schließt sich der Kreis und wir kommen zur *AfxWinMain*-Methode zurück. Nach der erfolgreichen Initialisierung der Anwendung, findet sich dort der folgende Aufruf:

Listing 3.20 **Starten der Nachrichtenschleife**

```
pThread->Run();
```

Trotz der Kürze hat es die *Run*-Methode von *CWinThread* in sich: hier drin befindet sich die gesamte Nachrichtenschleife, die dafür verantwortlich ist, dass Botschaften entgegengenommen und an die passenden Behandlungsmethoden weitergereicht werden.

Ruhezustand einer Applikation

Dazu bietet sie die Möglichkeit, die Anwendung in eine Art Ruhezustand zu versetzen. Immer, wenn die Applikation keine Nachrichten zu verarbeiten hat (das bedeutet normalerweise, dass der Benutzer gerade keine Eingaben macht), verzweigt Run in eine weitere Methode namens *OnIdle*.

Hier kann der Entwickler Funktionalitäten unterbringen, die immer dann ausgeführt werden sollen, wenn gerade nichts anderes anliegt. Für ein 3-D-Grafikprogramm wäre es hier zum Beispiel möglich, aufwendige Berechnungen durchzuführen, aber auch einfachere Aufgaben wie das Aktualisieren der Benutzeroberfläche (zum Beispiel das Darstellen von *Tooltips*) wird von hier aus initiiert.

Wichtig: findet kein Rücksprung von *OnIdle* zu *Run* statt, kann die Anwendung keine Nachrichten mehr auswerten, sie wird dem Benutzer als nicht mehr reaktionsfähig präsentiert. An dieser Stelle hilft dann nur noch ein Abbruch über den Task-Manager oder (in hartnäckigen Fällen) einen Reboot.

ExitInstance

Nun würde *Run* ewig laufen, wenn es nicht die Möglichkeit zum Beenden der Anwendung gebe. Diese tritt beispielsweise dann ein, wenn das Hauptfenster einer Applikation geschlossen wird.

Beendigung eines Programms

Ähnlich wie bei der Win32-Anwendung, deren *GetMessage*-Aufrufe bei einer Programmbeendigung das Verlassen der Nachrichtenschleife auslösen, wird auch die *Run*-Methode in einem solchen Fall verlassen, allerdings nicht, ohne zuvor einen Aufruf der *ExitInstance*-Methode durchzuführen.

Im Wesentlichen werden hier alle vom Applikationsgerüst angelegten Speicherbereiche wieder freigegeben und einige sonstige Aufräumarbeiten getätigt, die nichts direkt mit dem zu tun haben, was ein Entwickler noch selbstständig an Quelltexten zu einer Anwendung hinzugefügt hat.

Der komplette MFC-Programmablauf

Sie haben jetzt anhand eines ersten MFC-Beispiels den kompletten Ablauf einer solchen Windows-Anwendung kennen gelernt. Er ist in der nachstehenden Abbildung noch einmal grafisch zusammengefasst:

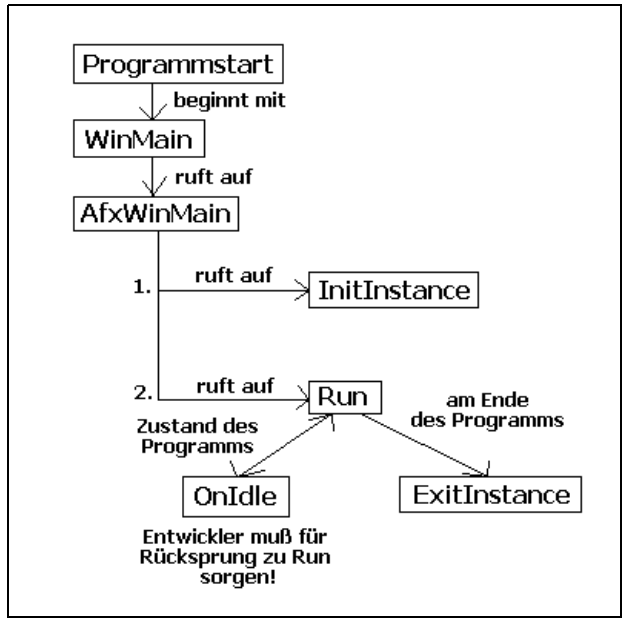


Abb. 3.24
Übersicht über den Ablauf einer MFC-Anwendung

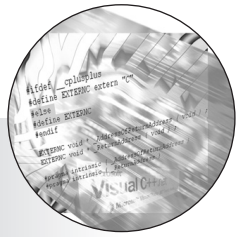
Zusammenfassung

Dieses Kapitel hat Ihnen aufgezeigt, wie grundlegende Windows-Programmierung funktioniert und welches Hintergrundwissen notwendig ist, um die ablaufenden Vorgänge zu verstehen. Nach einer Einführung in die Win32-Programmierung wurde der Wechsel zu einer MFC-Anwendung vollzogen, wobei das elementare Basiswissen zu solchen Applikationen angeführt wurde.

Vorschau

Nach diesen Grundlagen wird es in den folgenden Kapiteln mit der konkreten Programmierung moderner Software weitergehen, dabei lernen Sie das Anlegen von dialogfeldbasierten , SDI- und MDI-Anwendungen, das Erzeugen von GUIs – Menüs, Tooltips etc. –, die Auswertung von Benutzereingaben und so weiter.

Sollten Sie einige Sachverhalte im Laufe dieses Kapitels nicht verstanden haben, ist das nicht weiter tragisch, kehren Sie einfach zu einem späteren Zeitpunkt hierher zurück – einige Dinge werden sicherlich im Laufe der folgenden Kapitel klarer werden.



Entwicklung von dialogfeldbasierten Anwendungen

4

Grundlegender Entwurf einer MFC-Anwendung

Im Rahmen dieses Kapitels soll es um das Entwickeln einer einfachen dialogfeldbasierten Anwendung gehen.

Arten von Windows-Anwendungen

Es gibt im Wesentlichen drei Arten von Windows-Anwendungen, die sich insbesondere hinsichtlich der Darstellung und der Verwaltung von Dokumenten unterscheiden:

1. Dialogfeldbasierte Anwendungen: Diese bestehen aus einem Dialogfeld, das eventuell weitere Dialogfelder selbstständig bzw. auf Wunsch des Benutzers hin, öffnen kann. Normalerweise wird diese Art von Programmen benutzt, wenn es um einfache Sachverhalte wie das Konfigurieren einer Anwendung oder simple Tools wie eine Applikation zum Verschlüsseln von Dateien geht.

2. SDI-Anwendungen (Single Document Interface, *Einzeldokumentschnittstelle*) können jeweils ein Dokument zur Zeit geöffnet halten. Ein gutes Beispiel hierfür ist der Editor von Windows.

3. MDI-Anwendungen (Multiple Document Interface, *Mehrfachdokumentschnittstelle*) können eine theoretisch beliebige Anzahl von Dokumenten zur Zeit geöffnet haben. Ein Beispiel hierfür ist das Visual Studio, in dem Sie beispielsweise eine ganze Reihe von Quelltextdateien simultan geöffnet halten können.

Die dialogfeldbasierende Anwendung

Um die zugrunde liegenden Konzepte einer MFC-Applikation schnell zu begreifen, macht es Sinn, mit einem einfachen Programmtyp zu beginnen, der sich dadurch auszeichnet zwar sämtliche Möglichkeiten einer herkömmlichen dokumentorientierten Anwendung zu besitzen, aber aufgrund einer eher eingeschränkten Funktionalität die Aufmerksamkeit des lernenden Entwicklers auf spezielle Teilbereiche zu konzentrieren.

Entwurf eines Zeichenprogramms

Auf den folgenden Seiten dieses Kapitels werden Sie daher ein kleines Zeichenprogramm entwickeln, das innerhalb eines Dialogfelds den Benutzer in die Lage versetzt, eine Figur mittels Freihandstrichzeichnungen zu erzeugen.

Eine kleine Auswahl an Symbolen gestattet weiterführende Funktionalitäten, wie etwa die Auswahl des Zeichenwerkzeugs und das Löschen der Darstellungsoberfläche.

Anlegen des Projekts

MFC-Anwendung erzeugen

Starten Sie, falls noch nicht geschehen, das Visual Studio .NET und erzeugen Sie über den Menüpunkt *Datei* > *Neu* > *Projekt* ein neues Visual C++-Projekt auf Basis der *MFC-Anwendungsvorlage* (alternativ können Sie auch direkt das auf der Buch-CD im Verzeichnis *\Kapitel4\ZeichenProgramm* befindliche Projekt einladen und weiterverwenden).

Benennen Sie das Projekt *ZeichenProgramm* und bestätigen Sie die Auswahl durch Anwahl von *OK*.

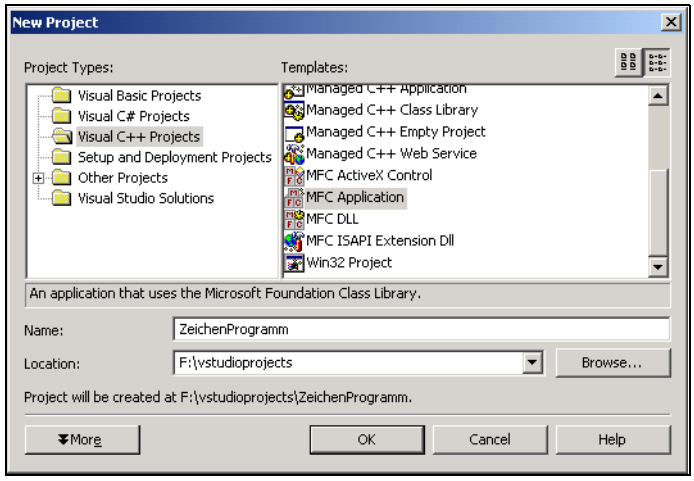


Abb. 4.1
Die korrekten Projekteinstellungen für das *ZeichenProgramm*-Projekt

Festlegen des Anwendungstyps

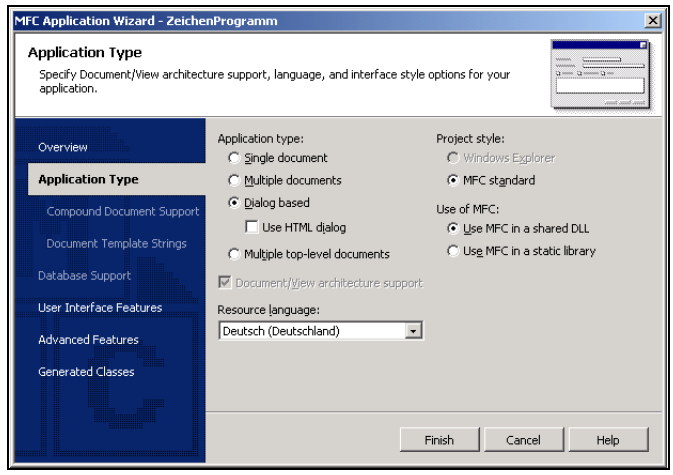
Die nun auftauchende Konfigurationsseiten haben Sie bereits in der Einführung zum Visual Studio .NET kennen gelernt, sodass hier nur noch auf die konkret für dieses Projekt notwendigen Einstellungen eingegangen werden soll.

Dialogfeldbasierter Anwendungstyp

Wählen Sie das Dialogfeld *Anwendungstyp* aus und stellen dort *Dialogfeldbasiert* ein. Ob Sie die MFC in einer statischen oder gemeinsam verwendeten Bibliothek verwenden wollen, bleibt im Rahmen dieses Buches Ihrem Geschmack überlassen, bedenken Sie aber, dass die Verwendung der gemeinsamen Bibliothek auf Rechnern Fehler verursachen wird, auf denen die MFC-Bibliotheken nicht installiert sind.

Die übrigen Einstellungen bleiben unverändert:

Abb. 4.2
Einstellen des Anwendungstyps



Erweiterte Einstellungen

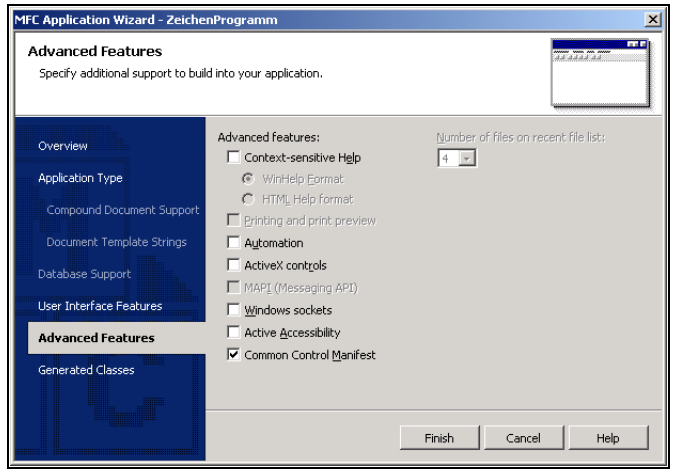
Die nächste Registerkarte, auf der Veränderungen vorgenommen werden müssen, ist die, die sich mit den erweiterten Einstellungen für die Applikation beschäftigt.

Deaktivieren von ActiveX-Steuerelementen

Da innerhalb des Projekts *ZeichenProgramm* keine *ActiveX Controls* verwendet werden sollen – mit denen sich ein späteres Kapitel dieses Buchs noch beschäftigen wird – können Sie den zugehörigen Punkt deaktivieren.

Der Rest der Einstellungen bleibt unverändert, sodass sich das folgende Bild für die Einstellungsseite ergibt:

Abb. 4.3
Die erweiterten Einstellungen für das *ZeichenProgramm*-Projekt



Überprüfen der Einstellungen

Wählen Sie schließlich den Punkt *Übersicht* aus, um zu verifizieren, dass sämtliche Einstellungen korrekt gemacht wurden:

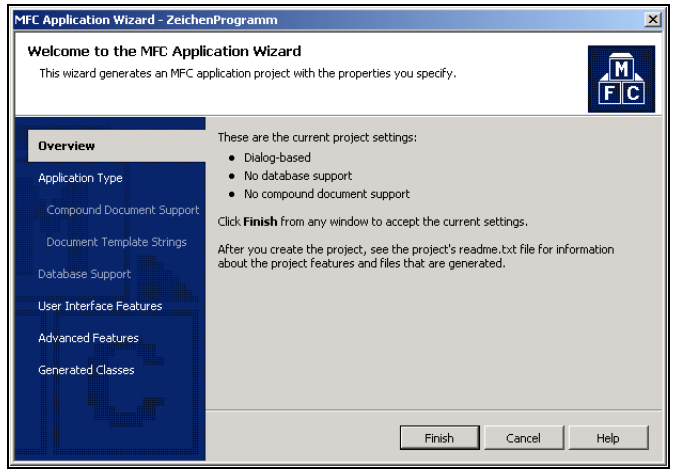


Abb. 4.4 **Übersicht über die gemachten Einstellungen**

Bestätigen Sie die Auswahl mit *Fertig stellen* und veranlassen Sie den Anwendungsassistenten dadurch, ein zu den Einstellungen passendes Grundgerüst zu erzeugen.

Kurzer Funktionstest

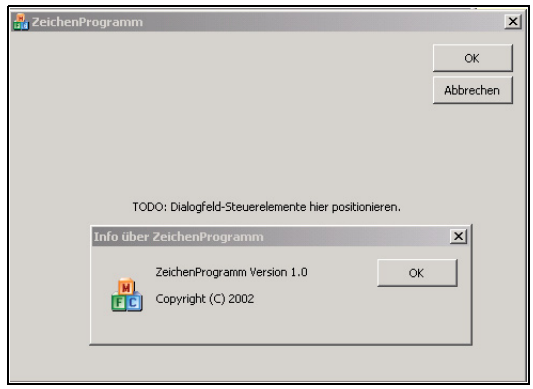
Kompilieren und starten Sie das vom Assistenten kreierte Projekt. Sie sehen nun das Hauptdialogfeld des neuen Programms.

Prüfen der Ausgabe

Im Systemmenü – erreichbar durch Anklicken des Applikationssymbols in der linken oberen Ecke – finden Sie einen Punkt namens *Info über ZeichenProgramm*. Wählen Sie ihn an. Es ergibt sich das folgende Bild 4.5.

Mehr Funktionalität – bis auf das Beenden des Programms durch Anwahl der *OK*- bzw. *Abbrechen*-Schaltfläche des Hauptdialogfelds – existiert derzeit noch nicht. Brechen Sie die Programmausführung ab und bringen Visual Studio .NET wieder in der Vordergrund.

Abb. 4.5
Das Hauptdialogeld und das Infodialogfeld im Überblick



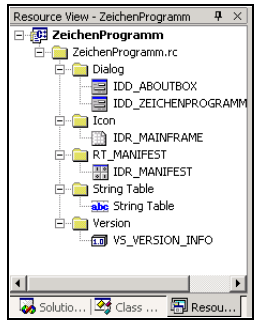
Ressourcen des Zeichenprogramms

Die Dialoge, die Sie während der Programmausführung zu Gesicht bekommen haben, stammen natürlich nicht von ungefähr, sondern wurden durch den Anwendungsassistenten angelegt.

Editieren von Dialogen

Um sie zu editieren, wählen Sie den Reiter *Ressourcenansicht* aus, den Sie am unteren Ende des Solution Explorers finden. Öffnen Sie sämtliche dort befindlichen Ordner, um einen Überblick über die zur Verfügung stehenden Ressourcen zu erhalten:

Abb. 4.6
Die Ressourcen für das geöffnete Projekt



Sie finden hier eine kleine Anzahl von Ressourcen, die allesamt im *ZeichenProgramm*-Projekt untergebracht sind. Wir wollen nun im Einzelnen schauen, wofür die einzelnen Ressourcen zuständig sind. Zum näheren Betrachten, was sich hinter einer Ressource verbirgt, ist es ausreichend, sie mit einem Doppelklick anzuwählen, woraufhin sich weitere Fenster zum Editieren der enthaltenen Daten öffnen.

Die Versionsressource

Zu einem Projekt gehören vielfältige Informationen, wenn man es richtig ein- und zuordnen möchte. Zu diesen Daten zählen beispielsweise der Name des Entwicklers – gegebenenfalls einer Entwicklerfirma –, eine kurze Beschreibung des Projektinhalts, aber auch die Versionsnummer und dergleichen mehr.

Applikations-informationen

Die Versionsressource enthält genau diese Arten von Informationen und es ist angebracht zu Beginn der Entwicklung eines Projekts diese eben genannten Angaben in entsprechender Form zu machen.

Gerade bei umfangreichen Projekten, die vielleicht innerhalb eines größeren Teams entwickelt werden, sind nämlich beispielsweise die gerade vorliegende Version der Quelltexte wichtig – stellen Sie sich eine Abteilung innerhalb einer Firma vor, die Programme auf Fehler prüft und dann feststellt, dass sie seit Tagen Fehlersuche in einer bereits veralteten Programmversion betreiben.

Ein Doppelklick auf die Versionsressource öffnet ein Fenster, in dem Sie die für Ihr Projekt notwendigen Einstellungen festlegen können:

ZeichenProgramm.rc (VS_VERSION_INFO - Version)

Key	Value
FILEVERSION	1, 0, 0, 1
PRODUCTVERSION	1, 0, 0, 1
FILEFLAGSMASK	0x3fL
FILEFLAGS	0x0L
FILEOS	VOS__WINDOWS32
FILETYPE	VFT_APP
FILESUBTYPE	VFT2_UNKNOWN
Block Header	Deutsch (Deutschland) (040704e4)
Comments	
CompanyName	Sybex GmbH
Copyright (C) 2002	(c) Sybex GmbH. Alle Rechte vorbehalten.
FileDescription	Programm zur Demonstration der Verwendung einer dialogfeldbasierten Applikation
FileVersion	1.0.0.1
InternalName	ZeichenProgramm.exe
LegalCopyright	
LegalTrademarks	
OriginalFilename	ZeichenProgramm.exe
PrivateBuild	
ProductName	ZeichenProgramm
ProductVersion	1.0.0.1
SpecialBuild	

Abb. 4.7
Die Einstellungs-optionen der Versions-ressource

Die Zeichenkettenressource

Eine wichtige Ressource ist die Zeichenkettenressource, da sie optimalerweise sämtliche innerhalb eines Projekts verwendete Texte enthalten sollte. Dazu zählen sowohl Menüeinträge als auch Beschriftungen von Schaltflächen oder Ausgaben in einem Fehlerfall.

Externe Texte

Der Vorteil liegt auf der Hand: durch das Entkoppeln der Anwendung von den Texten ist es leicht möglich, ein bestehendes Programm unkompliziert auf andere Sprachen zu übertragen.

Wurden die Texte statt dessen fest im Quelltext verankert, ist es sehr mühsam, diese per Hand herauszufiltern und entsprechend anzupassen – insbesondere eine Arbeit, die ein der Programmierung unkundiger Übersetzer gar nicht selbstständig durchführen könnte.

In der Praxis verzichtet man häufig darauf, die Beschriftungen von Schaltflächen oder anderen Steuerelementen dynamisch beim Programmstart aus der Zeichenkettenressource auszulesen und den jeweiligen Kontrollen zuzuweisen. Das liegt jedoch einfach daran, dass es mithilfe der Dialogfeldeditoren normalerweise ein Leichtes ist, die Beschriftung der Schaltflächen zu verändern. Außerdem behält man sich dadurch den Vorteil bei, die Beschriftungen in ihrem wirklichen Kontext zu sehen – es gibt vermutlich nichts Schlimmeres, als schlecht übersetzte Programmversionen: man stelle sich einen Bundesligamanager vor, der vom Englischen ins Deutsche übertragen wird und aus *Fans* mal eben *Ventilatoren* macht.

Die Zeichenkettenressource ist übersichtlich aufgebaut, sie besteht im Grunde nur aus einer Liste von Wertepaaren, die aus einer ID und dem eigentlichen String aufgebaut sind. Über die ID kann dann auf die Ressource zugegriffen werden, die Liste liefert die zugehörige Zeichenkette zurück.

Abb. 4.8
Die Zeichenkettenressource

Die Iconressource

Die Iconressource enthält sämtliche Icons, die im Rahmen der Anwendung benötigt werden, also Beispielsweise das Anwendungssymbol, das bei einem Hauptfenster in der linken oberen Ecke dargestellt wird.

Editieren von Icons

Das Microsoft Visual Studio .NET bietet einige rudimentäre Möglichkeiten zum Editieren dieser Symbole, gestattet es aber auch, diese mit anderen Programmen erzeugen zu lassen und dann in den Iconkontext zu übertragen.

Im Falle der vorliegenden Applikation *ZeichenProgramm* existiert nur ein einziges Symbol, nämlich gerade das bereits angesprochene Anwendungsicon (dessen ID *IDR_MAINFRAME* ist):

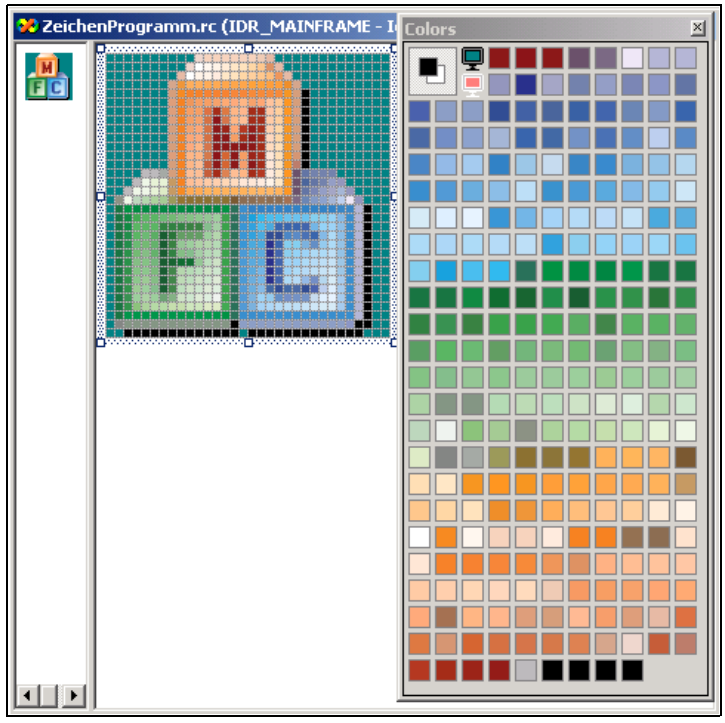


Abb. 4.9
Der Icon-Ressourcen-editor

Die Dialogressourcen

Die für das aktuelle Projekt wichtigsten Ressourcen dürften diejenigen sein, die das Aussehen und Verhalten der Dialoge – zum einen des Hauptdialogs, zum anderen des Infodialogs – beschreiben.

Sie finden diese Ressourcen im Ordner *Dialog*. Dabei heißt der Infodialog *IDD_ABOUTBOX*, der Hauptdialog *IDD_ZEICHENPROGRAMM_DIALOG*:

Namen der Dialoge

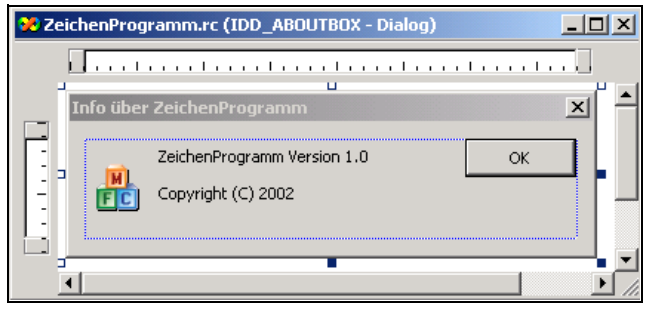
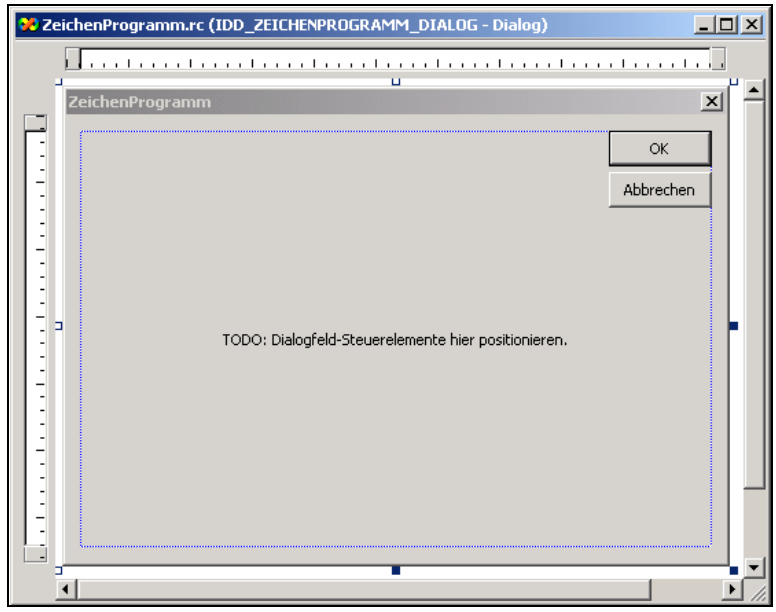


Abb. 4.10
IDD_ABOUTBOX

Abb. 4.11
IDD_ZEICHEN-PROGRAMM_DIALOG



Editieren von Dialogressourcen

Um Dialogressourcen editieren zu können, muss man zunächst einmal sehen, woraus sie überhaupt bestehen. Bei der Einführung in die Windows-Programmierung wurde darauf verwiesen, dass eigentlich sämtliche in einer Anwendung sichtbaren Elemente auch wieder Fenster sind.

Dieses Prinzip gilt natürlich auch für Dialoge, nur dass hier die einzelnen Fenster als *Kontrollen* (englisch *Controls*) oder *Steuerelemente* bezeichnet werden.

Hinzufügen und Editieren von Steuerelementen

Sie können in eine Dialogressource jedes beliebige Steuerelement einfügen, positionieren, in der Größe verändern und ihr passende Eigenschaften mit auf den Weg geben – dazu zählt auch das Festlegen einer für die Kontrolle einzigartigen ID, über die dann später vom Programm aus Zugriffe auf das Steuerelement stattfinden können.

Die Dialogfeldressourcen-Toolbox

Abb. 4.12
Die komplette Toolbox mit den verfügbaren Steuerelementen

Die Toolbox umfasst sämtliche Kontrollen, die Sie in Ihren Programmen einsetzen können. Sie können dieses Fenster über das Menü *View* > *Toolbox* (Strg+Alt+X) öffnen. Der folgende Abschnitt stellt die verfügbaren Steuerelemente vor, die Sie aus dieser Toolbox heraus in Ihre Dialoge einbauen können.

Kontrollelemente der Dialogfelder

Abb. 4.13
Knopf

Der gewöhnliche Knopf (auch *Schaltfläche* oder englisch *Button*) wird in der Regel für Aktionen benutzt, die sofort ausgeführt werden sollen. Prominentestes Beispiel sind sicherlich die *OK*- und *Abbrechen*-Knöpfe, wie sie in den meisten Dialogboxen vorkommen (unter anderem auch im Rahmen dieses Projekts).

Abb. 4.14
Checkbox

Die *Checkbox* – wird häufig auch als *Kontrollkästchen* bezeichnet – kommt immer dann zum Einsatz, wenn der Anwender eine Reihe von sich gegenseitig nicht ausschließenden Optionen anwählen kann. Gerade bei umfangreichen Applikationen bietet sich somit eine komfortable Möglichkeit, viele Funktionalitäten auf einen Blick darstellen zu können, anhand derer der User dann seine favorisierten Einstellungen wählt.

Abb. 4.15
Eingabefeld

Eins der wichtigsten Kontrollelemente dürfte das *Eingabefeld*, die so genannte *Editbox* (auf englisch häufig auch als *Editcontrol* bezeichnet) sein. Sie dient zur Aufnahme der von der Tastatur eingegebenen Daten und findet in vielen Bereichen ihre Anwendung, vor allem aufgrund ihrer vielfältigen Konfigurationsmöglichkeiten.

Abb. 4.16
Combobox

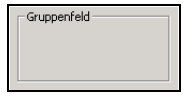
Die *Combobox* stellt sich im Ruhezustand als einfaches Editierfeld da, an dessen rechter Seite ein Dreieckssymbol angeordnet ist. Klickt man auf dieses, öffnet sich eine Tabelle der für dieses Feld gültigen Optionen. Weiterhin bietet sich die Möglichkeit, Comboboxen so anzulegen, dass auch neue Einträge hinzugefügt werden können. Diese neuen Einträge werden dann in der Regel beim nächsten Öffnen des Dialogfelds in der angehängten Auswahlliste ebenfalls angezeigt.

Es gibt eine erweiterte Combobox, die darüber hinaus die Möglichkeit bietet, Icons neben den einzelnen Texteinträgen darzustellen.

Abb. 4.17
Listenfeld

Das *Listenfeld* (*Listbox*) stellt in einer Liste eine Anzahl an möglichen Optionen bereit, die nicht editierbar sind. Reicht der für das Kontrollelement vorgesehene Platz nicht aus, wird automatisch eine *Scrollbar* an den rechten Rand angehängt, mit deren Hilfe in der Liste navigiert werden kann.

Abb. 4.18
Gruppenfeld



Das *Gruppenfeld* (englisch *Groupbox*) dient zum systematische Anordnen von Kontrollelementen, die zum selben Themenbereich gehören. Mit ihnen kann ein gewisser Grad an Ordnung auf umfangreichen Dialogfelder geschaffen werden.

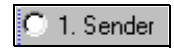


Abb. 4.19
Radiobutton

Im Gegensatz zur Checkbox steht der *Radio-Button* für sich gegenseitig ausschließende Optionen zur Verfügung. Dieses Kontrollelement wird immer in größerer Anzahl in die Dialogfelder eingestreut, um beispielsweise übersichtlich verschiedene Konfigurationsmöglichkeiten zur Verfügung zu stellen. Bei einem Spiel beispielsweise könnten Sie vielleicht zwischen einer 800*600 und einer 1.024*768 Bildauflösung wählen – beides gleichzeitig ist natürlich nicht praktikabel.

Abb. 4.20
Statischer Text

Das vermutlich einfachste Kontrollelement ist der statische Text (engl. *static text*). Er dient in erster Linie zum Beschriften der anderen, komplexeren Elemente, aber auch zur einfachen Textausgabe innerhalb von Dialogfeldern.

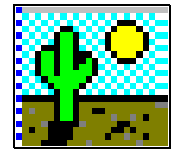


Abb. 4.21
Grafiken

Auch als Kontrollelement bezeichnet man innerhalb von Dialogfeldern einfache Grafiken, die meist nur zu Verschönerung dienen und keinen funktionellen Sinn haben. Gerade in den in nahezu allen Applikationen vorhandenen *About*-Boxen findet man diese Elemente.

Abb. 4.22
Horizontale Scrollbar

Abb. 4.23
Vertikale Scrollbar

Die von den Fenstern her bekannten *Scrollbars* gibt es sowohl in einer horizontalen wie auch in einer vertikalen Ausführung. Sie kommen immer dann zur Anwendung, wenn größere Ausschnitte oder Listen nicht komplett in den dafür vorgesehenen Bereich passen und die Ansicht daher verschoben werden muss.

Abb. 4.24
Slider

Der Slider (Schieberegler) findet seinen Einsatz in vielen Bereichen heutiger Applikationen. Sei es, dass mit ihm die Zoomgröße einer Grafikverarbeitungssoftware eingestellt, oder der Kompressionsgrad eines MP3-Enkoders festgelegt wird, seine Verwendung liegt immer dann nahe, wenn variable Werte nicht punktgenau festgelegt werden müssen, sondern es in erster Linie auf schnelle Bedienung ankommt. Meistens befindet sich ein Editierfeld in der unmittelbaren Umgebung, das den exakten Wert der momentanen Schiebereglerstellung angibt.

Abb. 4.25
Spin

Der Spin (nicht zu verwechseln mit dem vertikalen Scrollbar), dient zum schnellen Durchlaufen einer Reihe von Elementen. Er wird jedoch aufgrund von Listenfeldern und Comboboxen (die diese Funktionalität bereits beinhalten) nicht sehr häufig eingesetzt.

Abb. 4.26
Fortschrittsbalken

Der *Fortschrittsbalken* (*progress control*) zeigt innerhalb einer Anwendung an, wie weit die gerade ablaufende Funktion bereits fortgeschritten ist. Gerade bei Installationen wird diese Anzeige gerne eingesetzt, aber natürlich auch im Zusammenhang mit umfangreichen Berechnungsfunktionen. Voraussetzung für den Einsatz einer Fortschrittsanzeige ist das sinnvolle Zerlegen einer Tätigkeit in Kontrollpunkte, sodass jederzeit festgestellt werden kann, wie weit die Abarbeitung bereits fortgeschritten ist.

Abb. 4.27
Hotkeys

Hin und wieder ist es für den Benutzer eines Programms praktisch, wenn er sich selbst aussuchen kann, über welche Tastenkombination eine bestimmte Aktion der Applikation ausgeführt werden kann – als Beispiel sei ein Screenshot-Tool genannt, dass nach dem Drücken der jeweiligen Tastenkombination eine Kopie des Bildschirminhalts anfertigt. Zum Festlegen solcher Tastenkom-

binationen (*Hotkeys*) dient das Hotkey-Steuerelement. Nach der Fokussierung werden die vom Anwender gedrückten, für die jeweilige Aktion gewünschten, Tasten innerhalb der Kontrolle dargestellt und können dann vom Programm weiterbearbeitet werden.

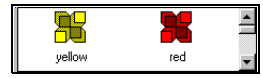


Abb. 4.28
Listkontrollfelder

Die *Listenkontrollfelder* (englisch *Listcontrols*) erlauben die Visualisierung von Texteinträgen mithilfe von Icons. Bekannt ist diese Darstellung unter anderem vom Windows-Arbeitsplatz.

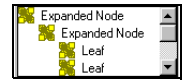


Abb. 4.29
Baumstruktuanzeige

Die *Baumstrukturanzeige* (*Tree Control*) ermöglicht die einfache Darstellung von untergeordneten Elementen wie sie zum Beispiel in einer Verzeichnisstruktur vorkommen. Das Programm *RegEdit*, das zum Editieren der Windows-Systemregistrierung dient, benutzt ebenfalls eine Tree Control zum Anzeigen der einzelnen Einträge.

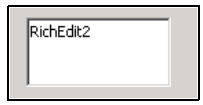
Abb. 4.30
Reiterbox

Die *Tab Control* (*Reiterbox*) dient zum platzsparenden Anordnen großer Informationsmengen oder Einstellungsmöglichkeiten innerhalb von Applikationen. Dieses Kontrollelement könnte als die digitale Repräsentation eines Karteikastens bezeichnet werden, bei dem jeder Reiter eine einzelne Karte darstellt.

Abb. 4.31
Beispiel für ein Animationssteuerelement

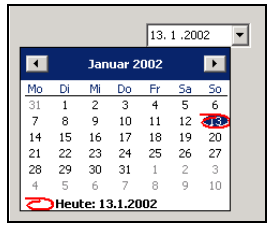
Mit dem *Animationssteuerelement* (englisch *Animation Control*) lassen sich in einem Bereich des Dialogfelds *AVI-Videos* abspielen. Ein Beispiel hierfür ist die sich bewegende Lupe beim Durchführen eines Suchvorgangs über die Standard-Suche im Windows-Explorer. Das Kontrollelement ist einigen Limitierungen unterworfen, unter anderem spielen sie keinen Sound ab, der in einem Video vorkommen mag

Abb. 4.32
***RichEdit*-Eingabefeld**



Ähnlich dem herkömmlichen Eingabefeld kann der Benutzer in einem *RichEdit-Eingabefeld* Texte eintippen, diese darüber hinaus aber auch formatieren, zum einen auf Zeichenebene, zum anderen durch Anwenden von Paragrafenformatierungen wie Linksausrichtung, Zentrieren des Texts und so weiter.

Abb. 4.33
Datumsauswahl-element

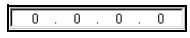


Mit dem *Datum/Zeitauswahl-Steuerelement* wird dem Benutzer die Möglichkeit gegeben, komfortabel ein Datum aus einem kalenderähnlich aufgebauten Feld auszuwählen. Dieses öffnet sich, sobald auf das in Kurzform dargestellten Datum geklickt wird, das sich in einem Dropdown-Menü befindet.

Abb. 4.34
Monatskalender

Das Steuerelement zum Darstellen eines Monatskalenders heißt im englischen *Month Calender Control*. Es zeigt den aktuellen Tag in eingekreister Form an und bietet die Möglichkeit, mithilfe von zwei Buttons durch die einzelnen Monate zu navigieren.

Abb. 4.35
IP-Kontrollelement



IP-Felder kommen vor allem in netzwerkorientierten Anwendungen zum Einsatz. Sie ermöglichen die einfache Eingabe einer IP-Adresse zur weiteren Verwendung durch das Programm.

Vorbereiten des Hauptdialogs

Für das in diesem Kapitel zu erzeugende Projekt wird wir nur eine sehr kleine Auswahl an unterschiedliche Kontrollen benötigen, doch reicht deren Einsatz aus, um die wesentlichen Einstellmöglichkeiten, die es beim Hinzufügen beziehungsweise Editieren von Steuerelementen gibt, aufzuzeigen.

Hinzufügen eines Gruppenfelds

Falls noch nicht geschehen, öffnen Sie jetzt das Hauptdialogfeld der Anwendung mit dem Namen *IDD_ZEICHENPROGRAMM_DIALOG*. Suchen Sie dann aus der Toolbox das Steuerelement *Gruppenfeld* heraus und platzieren es in etwa so wie in der nachfolgenden Abbildung (entweder durch eine Drag-&-Drop-Operation von der Toolbox in das Dialogfeld oder durch Auswahl des Elements und Aufziehen eines Kastens ähnlich der Markierung eines Bereichs in einem Zeichenprogramm):

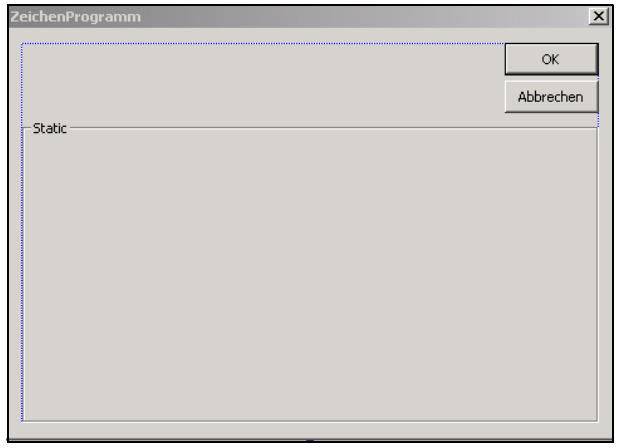


Abb. 4.36
Das eingefügte Gruppenfeld

Gruppenfeld als Zeichenfläche

Dieses Gruppenfeld soll uns als Zeichenfläche dienen, beschreibt also gerade den Bereich, den der Benutzer mit der Maus einfärben oder salopp gesagt, bemalen kann. Was zunächst wie ein einfacher Rahmen mit einer Beschriftung am oberen Rand aussieht, ist in Wirklichkeit ein komplexes Gebilde, bestehend aus einer Vielzahl von Attributen, Eigenschaften und Methoden.

Das liegt in letzter Instanz daran, dass es sich ja wie bereits mehrfach erwähnt, um ein eigenständiges Fenster handelt und somit eben auch entsprechende Beschreibungsmerkmale aufweisen muss.

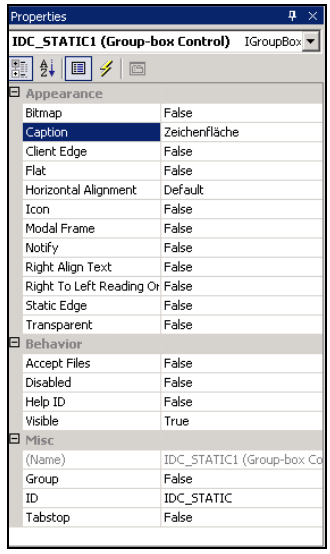
Dazu gehören insbesondere auch Behandlungsmethoden für Nachrichten, die diesem Fenster zugesandt werden können.

Gruppenfeld-eigenschaften

Einen Überblick über die Eigenschaften eines Gruppenfelds erhalten Sie durch einen Blick in das *Properties*-Fenster. Ist dieses zur Zeit nicht aktiviert, rufen Sie durch einen Rechtsklick das Kontextmenü des Hauptdialogs auf und wählen den Punkt *Eigenschaften*.

Sie sollten nun in der linken unteren Bildschirmecke das erwähnte Fenster vorfinden können. Klicken Sie jetzt das Gruppenfeld an und der Inhalt der *Property*-Übersicht wird mit den Werten dieses Steuerelements aktualisiert:

Abb. 4.37
Eigenschaften des Gruppenfelds



Verändern von Steuerelementeigenschaften

Aus der Vielzahl der möglichen Einstellungen, zu denen Sie jeweils eine kurze Erklärung am unteren Ende des Property-Fensters finden können, interessieren uns an dieser Stelle nur zwei.

Titel des Gruppenfelds ändern

Zum einen geht es dabei um den Text, der am oberen Rand des Steuerelements ausgegeben werden soll – Sie finden den zugehörigen Eintrag unter dem Punkt *Caption*. Tragen Sie hier das Wort *Zeichenfläche* ein und bestätigen Sie die Eingabe mit Enter. Der Inhalt wird sofort in die Dialogansicht übernommen.

Die andere interessante Einstellung verbirgt sich hinter der lapidaren Bezeichnung ID und kennzeichnet den Zahlenwert, über den die zugrunde liegende Kontrolle vom Rest des Programms aus referenziert werden kann.

Sie als Programmierer brauchen nicht direkt mit dem Zahlenwert herumzuhantieren, sondern erhalten den Zugriff über Konstanten, die gerade den entsprechenden Wert beschreiben.

ID_STATIC

Initial ist hier *ID_STATIC* eingetragen, was als Standardkonstante für sämtliche fixe, unveränderliche Elemente eines Dialogfelds gilt. Das heißt aber gleichzeitig, dass Sie auch nicht direkt auf diese Kontrolle zugreifen können, da es prinzipiell beliebig viele Steuerelemente mit einer ID von *ID_STATIC* geben kann.

Für das Zeichenprogramm ist es wichtig zu wissen, wie groß das Gruppenfeld ist, um dem Benutzer zu verbieten, über dessen Rand hinaus zu zeichnen. Daher ändern Sie die ID auf *IDC_ZEICHENFLAECHE1*, was es Ihnen in der Folge erlauben wird, die Position und Dimensionen des Steuerelements auszulesen.

Applikationsdateien

Nach diesen einleitenden Vorbereitungen wird es Zeit, einen Blick auf die vom Anwendungsassistenten erzeugten Dateien zu werfen. Nachdem Sie ja bereits eine MFC-Applikation per Hand entwickelt haben, ist es Ihnen während dieses Vorgangs möglich, Parallelen zwischen Ihren Quelltexten und den automatisch generierten Files zu ziehen.

Implementation der Applikationsklasse

Die Basis einer Anwendung ist ihr Applikationsobjekt, das, wie Sie wissen, die Instanz einer von *CWinApp* abgeleiteten Klasse ist. Die Deklaration für eben diese Kindklasse findet sich in der Datei *ZeichenProgramm.h* und ist im Folgenden abgedruckt:

Listing 4.1 ZeichenProgramm.h

```
// ZeichenProgramm.h : Hauptheaderdatei für die
// ZeichenProgramm-Anwendung
//

#pragma once

#ifndef __AFXWIN_H__
#error include 'stdafx.h' before including this
file for PCH
#endif

#include "resource.h"// Hauptsymbole


// CZeichenProgrammApp:
// Siehe ZeichenProgramm.cpp für die Implementierung
// dieser Klasse
//

class CZeichenProgrammApp : public CWinApp
{
public:
   CZeichenProgrammApp();

// Überschreibungen
   public:
   virtual BOOL InitInstance();

// Implementierung
```

```
   DECLARE_MESSAGE_MAP()
};

extern CZeichenProgrammApp theApp;
```

Implementierung der Applikationsklasse

Zwar ist die *CZeichenProgrammApp*-Klasse ein wenig umfangreicher als die von Ihnen selbst geschriebene *CMFCApplikation*, doch enthält sie inhaltlich nichts, was Ihnen unbekannt vorkommen dürfte.

InitInstance

Neben der *InitInstance*-Methode zum Initialisieren der Programminstanz sind noch ein Konstruktor für die Klasse sowie ein Deklarationsmakro zur Ankündigung einer Nachrichtentabelle in der Klasse enthalten.

Die Implementation selbst ist ebenfalls leicht verständlich:

Listing 4.2 ZeichenProgramm.cpp

```
// ZeichenProgramm.cpp : Definiert das
// Klassenverhalten für die Anwendung.
//

#include "stdafx.h"
#include "ZeichenProgramm.h"
#include "ZeichenProgrammDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif


// CZeichenProgrammApp

BEGIN_MESSAGE_MAP(CZeichenProgrammApp, CWinApp)
   ON_COMMAND(ID_HELP, CWinApp::OnHelp)
END_MESSAGE_MAP()


// CZeichenProgrammApp-Erstellung

CZeichenProgrammApp::CZeichenProgrammApp()
{
   // TODO: Hier Code zur Konstruktion einfügen
   // Alle wichtigen Initialisierungen in
// InitInstance positionieren
}

// Das einzige CZeichenProgrammApp-Objekt
CZeichenProgrammApp theApp;
```

```
// CZeichenProgrammApp Initialisierung

BOOL CZeichenProgrammApp::InitInstance()
{
   // InitCommonControls() ist für Windows XP
// erforderlich, wenn ein Anwendungsmanifest
// die Verwendung von ComCtl32.dll Version 6
// oder höher zum Aktivieren
   // von visuellen Stilen angibt. Ansonsten treten
// beim Erstellen von Fenstern Fehler auf.
   InitCommonControls();

   CWinApp::InitInstance();


   CZeichenProgrammDlg dlg;
   m_pMainWnd = &dlg;
   INT_PTR nResponse = dlg.DoModal();
   if (nResponse == IDOK)
   {
      // TODO: Fügen Sie hier Code ein, um das
// Schließen des Dialogfelds über OK zu
// steuern
   }
   else if (nResponse == IDCANCEL)
   {
      // TODO: Fügen Sie hier Code ein, um das
// Schließen des
      //  Dialogfelds über "Abbrechen" zu
// steuern
   }

   // Da das Dialogfeld geschlossen wurde, FALSE
// zurückliefern, sodass wir die
   //  Anwendung verlassen, anstatt das
// Nachrichtensystem der Anwendung zu starten.
   return FALSE;
}
```

Die Nachrichtentabelle der Applikationsklasse

Trotz der ausgezeichneten Kommentierung des Quelltexts durch den Anwendungsassistenten, müssen doch noch einige Kommentare zu den einzelnen Abschnitten gemacht werden.

ID_HELP und Hilfedateien

Die Nachrichtentabelle, um am Anfang zu beginnen, umfasst eine einzige zu behandelnde Nachricht, *ID_HELP*.

Eine *ID_HELP* Botschaft wird immer dann ausgelöst, wenn der Benutzer während des Programmablaufs F1 drückt. Sie kennen dieses Vorgehen sicherlich aus anderen Windows-Programmen – wenn Sie an einer Stelle nicht weiter wissen, oder beispielsweise erfahren möchten, was die einzelnen Einstellmöglichkeiten einer Programmoption bedeuten, öffnet das Drücken auf F1 häufig eine Hilfeseite mit weiterführenden Informationen.

Es ist klar, dass diese Informationen vom Entwickler des Programms selbst stammen müssen und nicht vom Anwendungsassistenten oder gar dem Programm selbst erzeugt werden können.

Im Falle des Zeichenprogramms beispielsweise existiert überhaupt keine Hilfedatei, die angezeigt werden könnte, weshalb der Versuch, eine Hilfestellung zu erhalten, mit einer schlichten Fehlermeldung quittiert wird.

Es versteht sich von selbst, dass das vorgeschlagene Neuinstallieren der Anwendung in unserem Fall kaum von Erfolg gekrönt sein wird: Solange Sie als Entwickler nicht selbstständig die gewünschte Hilfedatei anlegen, wird sie unauffindbar bleiben.

Wie Sie vorgehen müssen, um entsprechende Hilfetexte in Ihren Programmen unterzubringen, erfahren Sie in einem der späteren Kapitel, wenn es darum geht, die eigenen Programme um weitere Ressourcen wie Menüs oder Toolbars zu erweitern.

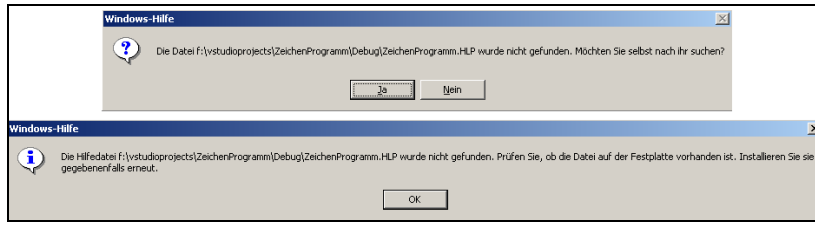


Abb. 4.38
Die Fehlermeldungen bei einer vorhandenen Hilfedatei

Initialisierung der Applikationsklasse

Zur Initialisierung der Applikationsklasse existieren prinzipiell zwei zentrale Anlaufstellen: zum einen der Konstruktor der Klasse, zum anderen die Methode *InitInstance*.

Initialisierungen außerhalb von Konstruktoren

Als C++-Programmierer sind Sie es vermutlich gewohnt, wichtige Initialisierungen einer Klasse im Konstruktor derselben abzulegen und nicht erst eine Initialisierungsmethode aufzurufen. Das macht natürlich auch Sinn, denn falls ein Entwickler, der mit Ihrer Klasse arbeitet – möglicherweise sind Sie dieses auch selbst – vergessen, die Initialisierungsmethode aufzurufen, ist möglicherweise die Funktionalität der verwendeten Objekte nicht gewährleistet.

Bei der Programmierung im Rahmen der MFC müssen Sie allerdings etwas mehr Vorsicht walten lassen. Häufig drängt sich der Wunsch auf, wichtige Fensterinitialisierungen – beispielsweise das Sichern von Größen- und Positionsangaben wie im Falle des Gruppenfelds – direkt beim Programmstart vorzunehmen und somit in den Konstruktor entweder der Applikations- oder der Dialogfeldklasse zu verfrachten.

Beide Stellen sind hierfür jedoch ungeeignet, wird das benötigte Fenster – und insbesondere die enthaltenen Steuerelemente – doch erst in einer der Initialisierungsmethoden erzeugt. Es wird unweigerlich zu einem Ausnahmefehler kommen, der gerade bei Anfängern der MFC-Programmierung häufig für viel Verwirrung sorgt.

Kurz gesagt sollten Sie tatsächlich nur Variableninitialisierungen im Rahmen von MFC-Klassen-Konstruktoren durchführen, niemals aber Funktionen aufrufen, die die Existenz von irgendwelchen Fensterobjekten bedingen!

Die *InitInstance*-Methode

Nachdem der Konstruktor bereits als nur bedingt geeignet zur Initialisierung bezeichnet wurde, geht das Augenmerk auf die *InitInstance*-Methode der Applikationsklasse über.

Diese unterscheidet sich nun doch ein wenig von dem, was Sie zuvor per Hand geschrieben haben, sodass eine nähere Betrachtung nötig wird.

Windows XP

Die ersten Zeilen beziehen sich auf ein mögliches Anwendungsmanifest, das im Rahmen von Windows XP die Verwendung von weiterführenden Kontrollelementstilen vorschreibt. Ein Weglassen dieses Aufrufs würde beim Erzeugen der jeweiligen Fenster Fehler verursachen.

Es folgt das Aufrufen der Basisklassenmethode *InitInstance*, die einige Initialisierungen durchführt, die allgemein für Klassen dieses Abstammungszweigs notwendig und wichtig sind.

Basisklassenmethoden

Das Aufrufen von Basisklassenmethoden ist innerhalb der MFC weit verbreitet. Der Grundgedanke dahinter ist, dass jede abgeleitete Klasse immer nur die Aktionen durchführt, die speziell für sie selbst notwendig sind und für alle globaleren Geschehnisse die Basisklassenmethode derselben, abgeleiteten Methode aufruft.

Je nach Ableitungstiefe können hierdurch eine ganze Reihe von Funktionsaufrufen ausgelöst werden und gerade bei der Verschachtelung von selbstgeschriebenen Klassen in der Vererbungshierarchie ist darauf zu achten, dass sich die einzelnen Methoden in ihrem Inhalt nicht gegenseitig widersprechen. Setzen Sie in einer abgeleiteten Methode eine Variable auf den Wert 0, in einer der Basisklassen aber auf den Wert 1, kann dieses zu schwer nachvollziehenden logischen oder tatsächlich einen Absturz hervorrufenden Fehlern führen.

In diesem Zusammenhang ist es auch wichtig, an welcher Stelle die Basisklassenmethode aufgerufen wird, ob also zu Beginn einer Funktion, während des Ablaufs oder vielleicht erst am Ende, da diese Reihenfolge durchaus zu unterschiedlichen Ergebnisse führen kann.

Bei Klassen, die sich direkt von MFC-Klassen ableiten richten Sie sich am besten nach den durch den Assistenten – gleich ob Klassen-, Funktions- oder Anwendungsassistenten – erzeugten Hinweisen. So steht normalerweise in jeder automatisch generierten Methode an einer Stelle ein Satz ähnlich „Hier Erweiterungen einfügen" oder schlicht „Todo".

Diese Positionen im Quelltext sorgen im Normalfall für eine reibungslose Verträglichkeit der Funktionsaufrufsfolge.

Anzeigen des Hauptdialogs

Jetzt wird es spannend: der Hauptdialog wird erzeugt und dargestellt.

Dialogkonstruktion

Zunächst muss man wissen, dass jeder Dialog, der im Ressourceneditor konstruiert wird, zunächst nichts anderes als eben dieses ist: eine Ressource.

Ressourcen sind im Falle von Dialogen mit Grafiken vergleichbar, wobei diese speziellen Bilder aus einem Hintergrund (dem Dialogfeldrahmen mit seiner Client Area) und darauf platzierten weiteren Grafiken (Knöpfe, Gruppenfelder und so weiter) bestehen.

Nun könnte man mit den Ressourcen an sich nichts weiter anfangen, daher ist es notwendig, zu jeder Ressource eine zugehörige Klasse zu schreiben, die definiert, was beim Anwählen der Steuerelemente passieren soll und wie sich der Dialog allgemein zu verhalten hat, wenn der Benutzer des Programms mit ihm interagiert.

Genaugenommen können Sie auch beliebig viele Klassen zu einer Dialogressource schreiben und so vollständig unterschiedliche Verhaltensweisen auf Basis desselben Dialogfelds implementieren. Inwieweit dieses sinnvoll ist hängt allerdings vom Einzelfall ab und ist allgemein kaum zu empfehlen.

Bei der Zeichenprogramm Applikation existiert eine Klasse *CZeichenProgramm-Dlg*, die die Implementierung eines Verhaltens für die bereits von Ihnen bearbeitete Dialogressource bereithält.

Verwendung des Dialogs

Um den Dialog jetzt also im Programm zu verwenden, brauchen Sie lediglich ein Objekt dieser Klasse anlegen und dann über einen einfachen Funktionsaufruf darzustellen.

Genau genommen haben Sie zum Darstellen sogar zwei Möglichkeiten. Welche Sie wählen, hängt davon ab, ob der Dialog modal oder nicht-modal darzustellen ist.

Modale und nicht-modale Dialoge

Man unterscheidet bei der Darstellung von Dialogen grundsätzlich zwischen den modalen und den nicht-modalen Dialogen.

Modale Dialoge werden durch den Funktionsaufruf *DoModal* angezeigt und haben als Rückgabewert die ID, die zum Beenden des Dialogs geführt hat – hört sich verwirrend an, besagt aber lediglich, dass beispielsweise *ID_OK* als Rückgabewert geliefert wird, wenn der Benutzer das Dialogfeld durch Anwahl von *OK* geschlossen hat, der gerade die ID *IDOK* hat. Gleichsam erhalten Sie ein Ergebnis von *IDCANCEL*, falls *Abbrechen* gewählt wurde. Es kann auch jede andere ID als Ergebnis herhalten, das ist abhängig davon, welche Möglichkeiten zum Beenden eines Dialogs Sie dem Benutzer zur Verfügung stellen wollen.

Das Hauptargument für ein modales Dialogfeld ist jedoch, dass es den restlichen Programmablauf einfriert. Das heißt, dass die aufrufende Funktion nicht weiter abgearbeitet wird, solange der Dialog noch aktiv ist. In der Tat ist es so, dass Sie in dieser Zeit absolut keine Möglichkeit haben, auf die restlichen Fenster der Anwendung zuzugreifen.

Ein Beispiel für einen modalen Dialog wäre der von vielen Anwendungen her bekannte *Öffnen/Speichern*-Dialog.

Vor- und Nachteile nicht modaler Dialoge

Im Gegensatz dazu erlauben es nicht-modale Dialoge, ganz normal mit der Anwendung im Hintergrund weiterzuarbeiten, während sie geöffnet sind. Das Öffnen selbst findet über einen Aufruf der Methode *CreateIndirect* statt. Dieses Verhalten ist bei nur recht wenigen Dialogen sinnvoll, beispielsweise bei einem *Suchen/Ersetzen*-Dialog.

Das Verwenden von nicht-modalen Dialogen birgt eine ganze Reihe von Schwierigkeiten, stellen Sie sich beispielsweise ein Aktion vor, die über ein Dialogfeld abgehandelt soll und mit den gerade im Dokument markierten Daten arbeitet. Ändert sich diese Markierung – was aufgrund der weiterhin aktivierbaren restlichen Anwendungsfenster ja ohne weiteres möglich ist – ist die gesamte Handlung des Dialogs unter Umständen nicht mehr gültig oder durchführbar.

Dankenswerterweise wird das Hauptdialogfeld der Zeichenprogramm-Applikation allerdings modal geöffnet, sodass wir uns an dieser Stelle mit den aus solchen nicht-modalen Dialogen resultierenden Problemen nicht weiter beschäftigen müssen.

Behandlung der Dialogbeendigung

Beendigung des Programms

Je nach Ergebnis der Dialogdarstellung – beziehungsweise je nach Beendigungsmethode durch den Benutzer – verzweigt *InitInstance* nun in einen zugehörigen Abarbeitungszweig, die jedoch beide ohne Inhalt sind. Das macht auch Sinn, da viele dialogfeldbasierende Anwendungen nach Beendigung des Hauptdialogs ebenfalls komplett beendet werden ohne noch weitere Aktionen (wie das Speichern von Daten oder Ähnlichem) durchzuführen.

Auch unser Zeichenprogramm hat nach dem Schließen nichts weiter zu tun, sodass hier keine Änderungen oder Erweiterungen einzufügen sind.

Die Hauptdialogklasse

Es ist nun schon einiges über die Klasse gesagt worden, die zur Darstellung des Hauptdialogs herangezogen wird. Grund genug, sich einmal ihren Aufbau im Detail anzusehen.

Hier zunächst die Headerdatei *ZeichenProgrammDlg.h*:

Listing 4.3 ZeichenProgramm-Dlg.h

```
// ZeichenProgrammDlg.h : Headerdatei
//

#pragma once


// CZeichenProgrammDlg Dialogfeld
class CZeichenProgrammDlg : public CDialog
{
// Konstruktion
public:
// Standardkonstruktor
   CZeichenProgrammDlg(CWnd* pParent = NULL);

// Dialogfelddaten
   enum { IDD = IDD_ZEICHENPROGRAMM_DIALOG };

   protected:
   virtual void DoDataExchange(CDataExchange* pDX);// DDX/DDV-
Unterstützung


// Implementierung
```

```
protected:
   HICON m_hIcon;

   // Generierte Funktionen für die
// Meldungstabellen
   virtual BOOL OnInitDialog();
   afx_msg void OnSysCommand(UINT nID,
LPARAM lParam);
   afx_msg void OnPaint();
   afx_msg HCURSOR OnQueryDragIcon();
   DECLARE_MESSAGE_MAP()
};
```

In dieser Deklaration finden sich einige neue Bestandteile, die in sämtlichen Dialogfeldklassen wiederzufinden und daher wichtig zu kennen sind.

Konstruktor und Ressourcenenumeration

Der Konstruktor ist noch vergleichsweise unspannend, dient er nur zur Initialisierung des Dialogs und erhält als Parameter einen Zeiger auf das Fenster, zu dem er gehören soll.

Definition von Platzhaltern

Die darauf folgende *Enumeration* ist als Vereinfachung und Vereinheitlichung anzusehen. Hier wird einer Konstanten *IDD* die ID der Dialogressource zugeordnet, die als Vorlage für die grafische Repräsentation dienen soll.

Dadurch, dass nun ein normierter Platzhalter *IDD* definiert wurde, können Sie leicht unterschiedliche Implementationen einer Dialogklasse realisieren, die vielleicht auf unterschiedlichen Dialogressourcen operieren, aber inhaltlich recht ähnlich agieren – und insbesondere dieselben IDs umfassen.

Man denke hier zum Beispiel an die Möglichkeit, einem Anwender eine Reihe von unterschiedlich aufgebauten Dialogfeldern zu präsentieren, die quasi als – derzeit ja sehr beliebte – Skin-Grundlagen zwar dieselben Funktionalitäten aufweisen, aber halt doch ursprünglich andere Dialogressourcen als Basis haben.

Datenaustausch und Verifizierung

Die ungefährlich aussehende Methode *DoDataExchange* hat es in sich – zum Glück bekommen Sie davon als Programmierer recht wenig mit, denn im Wesentlichen werden Sie nur mithilfe von Assistenten auf die Möglichkeiten dieser Funktion und der mit ihr verbundenen Strukturen zugreifen.

Datenaustausch mit Dialogen

Die durch *DoDataExchange* ermöglichte *DDX*/*DDV* (*Dialog Data Exchange* und *Dialog Data Validation*) Unterstützung dient kurz gesagt dazu, Werte zwischen dem Dialogfeld und Variablen auszutauschen und ihre Wertebereiche auf Gültigkeit zu überprüfen.

Als Beispiel sei ein Edit-Feld genannt, in das der Benutzer ganzzahlige Werte eintragen darf. Diese sollen allerdings in einem Bereich zwischen 0 und 12 bleiben.

Bei der Konstruktion der Dialogressource würden Sie diesem Edit-Feld eine Membervariable zuordnen, die den Wert bei einem Quittieren des Dialogs mit *OK* übernehmen soll.

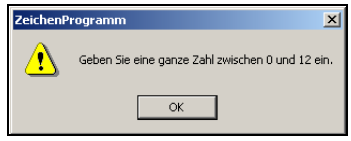
Angenommen, der Benutzer trägt in das Edit-Feld die Zahl 99 ein, so ist dieser Wert natürlich höher als der maximal Erwünschte. Hier kommt DDV zum Tragen. Vor dem Beenden des Dialogs werden sämtliche einstellbaren Werte auf Gültigkeit überprüft.

Prüfen ungültiger Werte

Bei unzulässigen Werten weist eine Meldung den Anwender darauf hin, dass seine Eingabe nicht korrekt war und verändert werden muss – der Dialog wird also an dieser Stelle nicht beendet, sondern wieder in den Vordergrund gebracht.

Sind die Werte hingegen allesamt korrekt, tritt DDX auf den Plan und transferiert die Informationen der Dialogfeld-Steuerelemente in die zugeordneten Membervariablen.

Abb. 4.39
DDV in Aktion



Es ist wichtig zu wissen, dass dieses Übertragen tatsächlich erst beim Beenden eines Dialogfelds automatisch durchgeführt wird, nicht etwa schon bei der Eingabe. Es ist jedoch auch während der Lebenszeit eines Dialogs möglich, die *DDX/DDV*-Funktionalität manuell auszulösen.

Behandlungsmethoden des Dialogs

Die restlichen Zeilen geben einen Überblick über die vom Hauptdialog behandelten Nachrichten. Dabei dient *OnInitDialog* zur Initialisierung des Felds – dieses ist analog zur *InitInstance* der Applikationsklasse zu betrachten –, *OnSysCommand* für die Behandlung von Systemkommandos (die Nachrichten die ausgelöst werden, wenn ein Menüeintrag des Systemmenüs angewählt wird, zum Beispiel die *Info Über...* Seite), *OnPaint* zum Neuzeichnen des Fensters nach dem Eingang einer *WM_PAINT* Nachricht (siehe unten) und zu guter Letzt *OnQueryDragIcon* – dessen zugehörige Nachricht dient zur Behandlung der Situation, wenn ein Fenster im minmierten Zustand verschoben werden soll. Die Methode liefert einen Verweis auf das währenddessen anzuzeigende Mauszeigerobjekt zurück.

Grau ist alle Theorie, daher hier nun die tatsächliche Implementation der eben kurz angerissenen Methoden:

Listing 4.4 ZeichenProgramm-Dlg.cpp

```
// ZeichenProgrammDlg.cpp : Implementierungsdatei
//

#include "stdafx.h"
#include "ZeichenProgramm.h"
#include "ZeichenProgrammDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif


// CAboutDlg-Dialogfeld für Anwendungsbefehl 'Info'

class CAboutDlg : public CDialog
{
public:
   CAboutDlg();

// Dialogfelddaten
   enum { IDD = IDD_ABOUTBOX };

   protected:
virtual void DoDataExchange(CDataExchange* pDX);
  // DDX/DDV-Unterstützung

// Implementierung
protected:
   DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
   CDialog::DoDataExchange(pDX);
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
END_MESSAGE_MAP()


// CZeichenProgrammDlg Dialogfeld
```

```
CZeichenProgrammDlg::CZeichenProgrammDlg(
CWnd* pParent /*=NULL*/)
   : CDialog(CZeichenProgrammDlg::IDD, pParent)
{
   m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

void CZeichenProgrammDlg::DoDataExchange(
CDataExchange* pDX)
{
   CDialog::DoDataExchange(pDX);
}

BEGIN_MESSAGE_MAP(CZeichenProgrammDlg, CDialog)
   ON_WM_SYSCOMMAND()
   ON_WM_PAINT()
   ON_WM_QUERYDRAGICON()
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()


// CZeichenProgrammDlg Meldungshandler

BOOL CZeichenProgrammDlg::OnInitDialog()
{
   CDialog::OnInitDialog();

   // Hinzufügen des Menübefehls "Info..." zum
// Systemmenü.

   // IDM_ABOUTBOX muss sich im Bereich der
// Systembefehle befinden.
   ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
   ASSERT(IDM_ABOUTBOX < 0xF000);

   CMenu* pSysMenu = GetSystemMenu(FALSE);
   if (pSysMenu != NULL)
   {
      CString strAboutMenu;
      strAboutMenu.LoadString(IDS_ABOUTBOX);
      if (!strAboutMenu.IsEmpty())
      {
         pSysMenu->AppendMenu(MF_SEPARATOR);
         pSysMenu->AppendMenu(MF_STRING,
IDM_ABOUTBOX, strAboutMenu);
      }
   }
```

```
   // Symbol für dieses Dialogfeld festlegen. Wird
// automatisch erledigt
   // wenn das Hauptfenster der Anwendung kein
// Dialogfeld ist
   SetIcon(m_hIcon, TRUE);//Großes Symbol verwenden
   SetIcon(m_hIcon, FALSE);//Kleines Symbol
// verwenden

   // TODO: Hier zusätzliche Initialisierung
// einfügen

   return TRUE;  // Geben Sie TRUE zurück, außer
// ein Steuerelement soll den Fokus
// erhalten
}

void CZeichenProgrammDlg::OnSysCommand(UINT nID,
LPARAM lParam)
{
   if ((nID & 0xFFF0) == IDM_ABOUTBOX)
   {
      CAboutDlg dlgAbout;
      dlgAbout.DoModal();
   }
   else
   {
      CDialog::OnSysCommand(nID, lParam);
   }
}

// Wenn Sie dem Dialogfeld eine Schaltfläche
// "Minimieren" hinzufügen, benötigen Sie
//  den nachstehenden Code, um das Symbol zu zeichnen.
// Für MFC-Anwendungen, die das
//  Dokument/Ansicht-Modell verwenden, wird dies
// automatisch ausgeführt.

void CZeichenProgrammDlg::OnPaint()
{
   if (IsIconic())
   {
      CPaintDC dc(this); // Gerätekontext zum
// Zeichnen

      SendMessage(WM_ICONERASEBKGND,
 reinterpret_cast<WPARAM>
(dc.GetSafeHdc()), 0);
```

```
      // Symbol in Clientrechteck zentrieren
      int cxIcon = GetSystemMetrics(SM_CXICON);
      int cyIcon = GetSystemMetrics(SM_CYICON);
      CRect rect;
      GetClientRect(&rect);
      int x = (rect.Width() – cxIcon + 1) / 2;
      int y = (rect.Height() – cyIcon + 1) / 2;

      // Symbol zeichnen
      dc.DrawIcon(x, y, m_hIcon);
   }
   else
   {
      CDialog::OnPaint();
   }
}

// Die System ruft diese Funktion auf, um den Cursor
// abzufragen, der angezeigt wird, während der
// Benutzer
//  das minimierte Fenster mit der Maus zieht.
HCURSOR CZeichenProgrammDlg::OnQueryDragIcon()
{
   return static_cast<HCURSOR>(m_hIcon);
}
```

Die *Info Über*-Dialogbox

Zu Beginn der Implementationsdatei wird zunächst die *Info Über*-Dialogboxklasse deklariert und definiert, die zum Darstellen des Informationsdialogs dient. Dieses ist nicht weiter spannend, und stellt bei genauem Hinsehen auch nur eine abgeleitete Klasse der Basisklasse *CDialog* dar, ohne weiterführende Funktionalitäten zu bieten.

Wir wollen uns also direkt um die im Anschluss daran befindlichen Methoden der Klasse *CZeichenprogrammDlg* kümmern.

Konstruktor der Klasse *CZeichenprogrammDlg*

Der Konstruktor der Klasse *CZeichenprogrammDlg* weist der Variablen *m_hIcon* das Handle des Applikations-Icons zu, das innerhalb der Ressourcen unter der Bezeichnung *IDR_MAINFRAME* abgelegt ist:

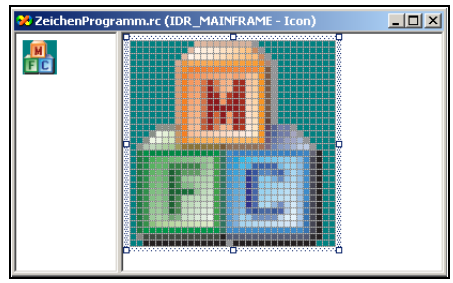


Abb. 4.40
Die Ressource *IDR_MAINFRAME*

Innerhalb der Dialogfeldklasse kann so bequem auf dieses Icon zugegriffen werden, ohne immer den Umweg über die Ressourcenangaben gehen zu müssen.

Die Methode *AfxGetApp* liefert dabei einen Zeiger auf das Applikationsobjekt zurück, über dessen *LoadIcon*-Funktion wiederum das Ressourcenhandle angefordert werden kann.

Zugriff auf das Applikationsobjekt

DoDataExchange und Message Map

Es folgt die Definition der Methode *DoDataExchange*, die, wie bereits angesprochen, für den Austausch zwischen Dialogfeldwerten und -variablen zuständig ist. In diesem einfachen Fall reicht das Aufrufen der Basisklassenmethode, die für die Ausführung des Datenaustauschs zuständig ist.

Hier könnten Sie beispielsweise noch weitere Prüfungen einfügen, die vielleicht über *DDX/DDV* nicht durchzuführen sind. Denken Sie aber in jedem Fall daran, die Basisklassenmethode aufzurufen, um das reibungslose Übertragen der Variablen zu gewährleisten.

Die anschließende Message Map fasst noch einmal die zu behandelnden Botschaften zusammen.

Initialisierung des Dialogs

In *OnInitDialog* finden sich die für den Dialog notwendigen Initialisierungen wieder. Den Anfang macht das Einfügen der *Info Über*-Box in das Systemmenü des Dialogs.

Zur Erinnerung: das Systemmenü erreichen Sie durch Anklicken des Applikations-Icons in der linken oberen Ecke des Fensters.

Zunächst wird geprüft, ob sich die ID des Menüeintrags im Bereich des Systemmenü *IDs* befindet. Sie werden in einem späteren Kapitel noch mehr über Menüs und deren Einsatz erfahren, für jetzt nur soviel: Systemmenü-Nachrichten werden über eine eigene Fensternachricht bekannt gemacht, nämlich gerade *WM_SYSCOMMAND*.

Bereich von Menüeintrags-IDs

Damit diese Nachricht aber auch gesendet wird, muss das auslösende Ereignis – in der Regel halt ein Menüeintrag – eine ID in einem dafür vorgesehen Wertebereich haben. IDs sind ja, wie Sie sicherlich noch wissen, lediglich positive ganzzahlige Werte, die zur einfacheren Verwendung durch eine Konstante beschrieben werden können.

Begrenzung des Systemmenüs

Diese Einschränkung im möglichen Wertebereich, der dafür sorgt, dass nicht beliebig viele Einträge in das Systemmenü vorgenommen werden können, was allerdings auch kaum sinnvoll sein dürfte, wird durch zwei aufeinanderfolgende *ASSERT*-Anweisungen auf Gültigkeit geprüft. *ASSERT*s werden im Rahmen dieses Buches im Kapitel *Debugging* noch ausführlich behandelt werden.

Die folgenden Zeilen enthalten wiederum eine Reihe von weiterführenden Aufrufen, die ebenfalls erst in den folgenden Kapiteln erläutert werden können. Kurz gesagt fordert der folgende Abschnitt das Systemmenü an – was durch einen zurückgelieferten Zeiger auf eben dieses quittiert wird – und versucht dann, die *Info Über*-Menüzeile in das Systemmenü einzugliedern.

Zur besseren Übersichtlichkeit wird bei dieser Gelegenheit auch gleich noch eine Trennzeile eingearbeitet, die den neuen Punkt von den bereits bestehenden Kommandos Verschieben und Schließen trennt.

Festlegen eines Applikations-Icons

Die restlichen Zeilen dieser *InitDialog*-Methode setzen das Applikations-Icon für die Anwendung fest, nämlich gerade auf das, welches im Konstruktor aus dem Ressourcenbereich der Applikation angefordert wurde.

Abhängig vom zweiten booleschen Parameter wird das große oder kleine Applikations-Icon festgelegt. Je nach Situation wird eines dieser Icons angezeigt. Hier legen Sie also fest, welche Symbole beispielsweise der Windows-Explorer anzeigen soll, wenn als Darstellungsart kleine Symbole oder große Symbole ausgewählt sind.

Als Rückgabewert der Funktion können Sie den Wert *true* angeben, falls Sie keinem Steuerelement des Dialogs explizit den Fokus zuweisen wollen (in diesem Fall setzt Windows den Fokus auf das erste gefundene Steuerelement des Dialogs) oder *false*, falls Sie einem Steuerelement den Fokus mithilfe der dafür zuständigen Funktion *SetFocus*, die an dieser Stelle nicht weiter beschrieben werden soll, zugewiesen haben – Windows lässt in diesem Fall den Fokus unverändert.

Versäumen Sie, den Fokus manuell zu setzen, geben aber *false* als Rückgabewert an, kann es zu einer Zugriffsverletzung bei der Ausführung der Anwendung kommen.

Behandlungsmethode für *WM_SYSCOMMAND*

Die Nachricht *WM_SYSCOMMAND* wird durch die Behandlungsmethode *OnSysCommand* bearbeitet. Nach einer Überprüfung, ob als ID die des *Info Über*-Dialogeintrags an die Methode übergeben wurde, wird bei einer positiven Beantwortung dieser Frage das Informationsdialogfeld aufgebaut.

Dies geschieht analog zum Aufruf des Hauptdialogs, also insbesondere durch einen Einsatz der Funktion *DoModal*.

Es ist an dieser Stelle nicht erforderlich zu prüfen, wie der Anwender das Dialogfeld beendet hat. Zum einen, weil es hierfür ohnehin nur eine Möglichkeit gibt, zum anderen, weil das Dialogfeld an sich unerheblich für die weitere Ausführung der Anwendung ist.

Möglichkeit der nichtmodalen Darstellung

Das Informationsdialogfeld könnte prinzipiell auch nicht modal dargestellt werden, sein Informationsgehalt und seine Funktionalität sind hinreichend unkritisch.

Stimmt die ID nicht mit der des Informationsdialogfelds überein, wird die Standardbehandlung aus der Basisklasse aufgerufen – in diesem Fall also insbesondere die ordnungsgemäße Abarbeitung der Kommandos *Verschieben* und *Schließen*.

Behandlungsmethode für *WM_PAINT*

Es wurde in der Einführung bereits angedeutet, dass die *WM_PAINT*-Nachricht dafür zuständig ist, den Client-Bereich eines Fensters mit denen durch die Anwendung erzeugten (also durch den Benutzer eingegeben oder eingeladenen oder durch die Applikation vorgeschriebenen) Daten zu füllen.

In diesem Kapitel, in dem es ja um das Schreiben eines kleinen Zeichenprogramms gehen soll, bezieht sich diese Aussage natürlich auf das Darstellen der gemalten Punkte und Striche.

Ausgangsform der *WM_PAINT*-Behandlungsmethode

In ihrer Rohform jedoch macht die Behandlungsmethode der WM_PAINT Nachricht nichts dergleichen, sie lässt die Client Area des Dialogfelds unbeschrieben.

Natürlich versteht man unter dem Client-Bereich eines Fensters normalerweise alles innerhalb des Fensters aber außerhalb von Statuszeilen, Toolbars und so weiter. Im Falle des Dialogfelds ist dieses ähnlich, doch überlagern Steuerelemente den Bereich der Client Area. Diese werden also in jedem Fall dargestellt, egal, was in der *OnPaint*-Methode geschrieben steht.

In *OnPaint* wird zunächst geprüft, ob die Anwendung minimiert ist, oder nicht. Ist sie es, braucht nur das Applikations-Icon in die verkleinerte Titelleiste der Anwendung gemalt zu werden.

Das ist zwar nicht sonderlich aufregend, bedingt aber trotzdem eine Reihe von Anweisungen, die vom Löschen der Titelzeile bis hin zur Ausrichtung des Symbols und schließlich dem Zeichen des Icons reichen.

Es ist an dieser Stelle nicht notwendig, weiter auf die einzelnen Zeilen einzugehen, da sie bei dem Großteil der Windows-Anwendungen, die Sie schreiben werden, nicht verändert zu werden brauchen.

Es bestünde an dieser Stelle allerdings durchaus die Möglichkeit, eine Art Animation durch ständig wechselnde Applikations-Icons zu erzeugen. Inwieweit dieser Blickfang von einem potenziellen Anwender gewünscht wird, sei dahingestellt.

WM_QUERYDRAGICON

Die letzte Methode, die in der Datei *CZeichenprogrammDlg.cpp* definiert ist, dient zum Festlegen des Mauszeigers, der dargestellt werden soll, wenn ein Fenster im minimierten Zustand verschoben wird.

Normalerweise ist hier die standardmäßig vorgegebene Methode gerade gut geeignet und muss nur in Ausnahmefällen überarbeitet werden.

Festlegen von eigenen Behandlungsmethoden

Die bereits bestehenden Behandlungsmethoden bieten eine gute Basis für das zu entwickelnde Zeichenprogramm, sind jedoch bei weitem nicht ausreichend, um die benötigte Funktionalität zu gewährleisten.

Reaktion auf Benutzereingaben

Überlegen Sie an dieser Stelle einmal, welche Aktionen der Benutzer durchführen muss, um im Zeichenprogramm eine freihändige Linie zu zeichnen – dabei einmal davon ausgehend, dass das Freihandwerkzeug sowie die gewünschte Farbe bereits ausgewählt sind und nur noch der Zeichenvorgang durchzuführen ist.

Ihnen sollten wenigstens die folgenden drei wichtigen Ereignisse eingefallen sein:

1. Der Benutzer drückt die linke Maustaste, um mit dem Zeichnen zu beginnen
2. Der Benutzer bewegt die Maus, entweder um den Mauszeiger für eine neu zu zeichnende Linie neu zu positionieren, oder um einen laufenden Zeichenvorgang bei gedrückter linker Maustaste fortzuführen.
3. Der Benutzer hatte die linke Maustaste bereits gedrückt, um einen Linienzug zu zeichnen und lässt die Maustaste nun wieder los.

Diese Ereignisse werden, wie es unter Windows üblich ist, im Rahmen von Fensternachrichten an das Dialogfeld gesandt, ohne derzeit von unserer Applikation bearbeitet zu werden.

Nicht bearbeitete Botschaften

Wenn wir sagen, dass unsere Applikation auf einige Fensternachrichten nicht reagiert, ist dieses nur halb richtig: tatsächlich sind für sämtliche möglichen Fensternachrichten Behandlungsmethoden vorgesehen, die in den Basisklassen des Dialogs (oder eines anderen Fensters) implementiert und im Falle einer entsprechenden eingehenden Nachricht abgearbeitet werden.

Es ist lediglich so, dass wir nicht explizit bestimmte Aktionen durchführen, sondern uns, ohne eine passende Methode zu überschreiben, auf die Basisklassenmethoden verlassen, die im Zuge wiederum nur eine Standardbehandlung abarbeiten – im Zweifelsfall also auch nichts tun.

Ermitteln von Fensternachrichten

Die drei oben genannten Punkte lassen sich durch die Windows-Nachrichten *WM_LBUTTONDOWN* (linke Maustaste gedrückt), *WM_LBUTTONUP* (linke Maustaste losgelassen) und *WM_MOUSEMOVE* (Mauszeiger wurde bewegt) beschreiben.

Für den einsteigenden MFC-Entwickler ist es immer etwas problematisch festzustellen, welche Fensternachrichten für welche Ereignisse zuständig sind.

Erlernen von Nachrichtenzugehörigkeiten

Während es bei Mausnachrichten noch einigermaßen schlüssig zu sein scheint, welche Möglichkeiten hier überhaupt existieren (Drücken der Maustasten, Bewegen der Maus, Drehen eines Mausrads), kann dieses bei komplexen Kontrollelementen, die als eigenständige Fenster ebenfalls Windows-Nachrichten empfangen können, durchaus komplizierter sein.

Allerdings ist dieses eine der Hürden, die gemeistert werden müssen, um tief in die MFC-Entwicklung eintauchen zu können. Es führt kein Weg daran vorbei, dass Sie sich einmal die Zeit nehmen, die einzelnen möglichen Fensternachrichten zu studieren und grob zu überfliegen, wofür jede einzelne zuständig ist – hierbei sind natürlich nur die Standard-Windows-Nachrichten gemeint, nicht etwa applikationsspezifische Botschaften.

Im Anhang dieses Buchs finden Sie eine Auflistung der meisten Windows-Nachrichten, wobei diese keinen Anspruch auf Vollständigkeit erheben kann oder will. Nutzen Sie diese Quelle als Basis für Ihre weiteren Recherchen, zum Beispiel im Rahmen der Visual Studio.NET Online-Hilfe.

Für dieses Projekt (und auch die im weiteren Verlauf des vorliegenden Buchs) werden Ihnen die benötigten Fensternachrichten allerdings jeweils explizit genannt werden.

Festlegen von neuen Behandlungsmethoden

Hinzufügen von Nachrichtenfunktionen

Um weitere Behandlungsmethoden zu einer bestehenden Anwendung hinzuzufügen, können Sie das bereits bekannte Eigenschaftendialogfeld des Visual Studio.NET benutzen. Ist dieses derzeit nicht aktiv, begeben Sie sich in die Ressourcenansicht, öffnen das Hauptdialogfeld und dort das zugehörige Kontextmenü und wählen den Punkt *Eigenschaften* aus der Liste der möglichen Kommandos.

Im erscheinenden Dialogfeld finden Sie ganz oben eine Liste mit Symbolen, wie sie auch im nachstehenden Screenshot festgehalten sind:

Abb. 4.41 **Die Symbole des *Eigenschaften*-Dialogs**

Es ist wichtig, dass Sie sich mit diesen Symbolen vertraut machen, um bei der späteren Entwicklung zügig mit Ihnen arbeiten zu können und nicht erst ständig grübeln müssen, welches Symbol welche Informationen verbirgt.

- Die ersten beiden Symbole stellen die Auflistungsreihenfolge der Eigenschaften im darunter befindlichen Fenster dar. Der linke Knopf listet diese Werte nach Kategorien sortiert auf und ist immer dann sinnvoll, wenn Sie nicht genau wissen, welche Eigenschaft ein bestimmtes Attribut beschreibt.

  Es ist dann häufig möglich, diesen Wert durch eine Untersuchung der passenden Oberkategorie zu finden. Wollen Sie beispielsweise festlegen, dass ein Dialog Dateien per Drag & Drop übernehmen kann, werden Sie schnell auf die Kategorie *Behaviour* (Verhalten) stoßen, in der es ein auf Ihre Problemstellung passendes Attribut *Accept Files* gibt.

  Der rechte Knopf listet die Eigenschaften alphabetisch ohne Kategorisierung auf – recht nützlich, wenn ein Attributname bekannt ist und Sie nicht erst umständlich innerhalb der Kategorien navigieren wollen.

Festlegen des anzuzeigenden Informationsgehalts

- Die folgenden drei Symbole regeln, welche Arten von Informationen angezeigt werden sollen. Die eigentlichen Eigenschaften eines Dialogs sind über den linken Knopf dieser Gruppe zu aktivieren. Hier werden feste Konstanten für die einzelnen Attribute festgelegt, beispielsweise der in der Titelzeile eines Dialogs anzuzeigende Text oder der im Dialogfeld zu verwendende Zeichensatz.

  Der mittlere Knopf listet sämtliche Steuerelemente auf, die ein Dialog besitzt. Sie bekommen so einen schnellen Überblick darüber, woraus Ihr Feld im Einzelnen besteht. Wählen Sie eine der Kontrollen in dieser Liste an, erscheint eine Zusammenfassung der Fensternachrichten, auf die dieses Steuerelement reagiert.

Der rechte Knopf schließlich listet die vom Dialogfeld verarbeiteten Fensternachrichten auf. Dabei sind hier sämtliche möglichen Fensternachrichten aufgeführt, doch nur diejenigen, zu denen auch ein Methodenname spezifiziert wurde, werden durch die Anwendung implementiert. Das Zeichenprogramm ist hier eingangs recht jungfräulich und besitzt bis auf *OnPaint*, *OnQueryDragIcon* und *OnSysCommand* keine zugeordneten Behandlungsfunktionen. Entsprechend leer präsentiert sich daher auch das Fenster mit den zur Behandlung bereitstehenden Fensternachrichten:

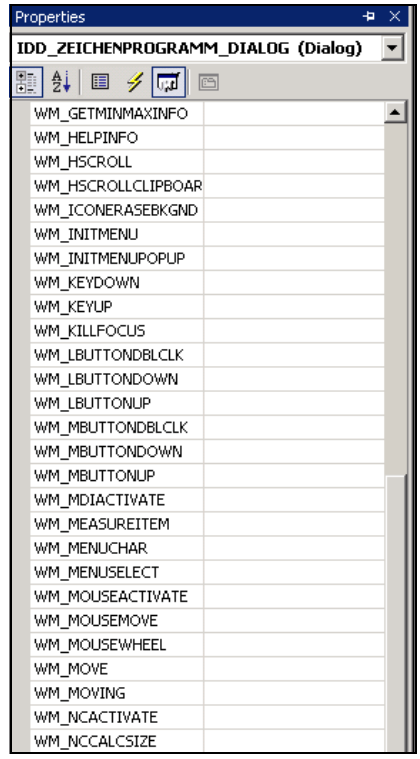


Abb. 4.42
Nachrichten für den Hauptdialog

Auswahl von neuen Behandlungsmethoden

Aufnehmen von Behandlungsfunktionen für die Benutzerinteraktion

Weiter oben wurden die drei Windows-Nachrichten genannt, die bearbeitet werden müssen, um dem Benutzer die Möglichkeit zum Zeichnen auf der Zeichenfläche zu geben. Um diese in die Anwendung aufzunehmen – das heißt, um eine Behandlungsfunktion für diese Nachrichten einzurichten – klicken Sie neben der Fensternachricht in die rechte Spalte der Tabelle und wählen den daraufhin erscheinenden Pfeil nach unten zur Anzeige einer Liste möglicher Optionen an.

Führen Sie dieses nun für die Nachricht *WM_LBUTTONDOWN* durch, Sie erhalten darauf eine Ansicht ähnlich der folgenden:

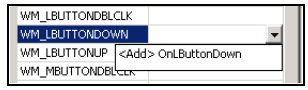


Abb. 4.43 **Hinzufügen einer Behandlungsmethode**

Hinzufügen einer neuen Methode

Als Option steht hier nur <*Add*> *OnLButtonDown* zur Verfügung, was nichts anderes bedeutet, als dass der Klassen- und Behandlungsmethodenassistent nach Anwahl dieses Kommandos eine Methode *OnLButtonDown* in das bestehende Anwendungsgerüst integrieren wird, die gerade zur Behandlung der nebenstehenden Botschaft *WM_LBUTTONDOWN* dienen wird.

Führen Sie diese Aktion nun aus, Sie werden daraufhin automatisch in der Datei *ZeichenprogrammDlg.cpp* an den neu eingefügten Quelltextbereich versetzt:

***Listing 4.5 OnLButtonDown*-Methode**

```
void CZeichenProgrammDlg::OnLButtonDown(UINT nFlags, CPoint point)
{
   // TODO: Add your message handler code here
// and/or call default

   CDialog::OnLButtonDown(nFlags, point);
}
```

Der Inhalt dieser Methode ist nun relativ nichtssagend, in der Tat tut sie nichts weiter, als die Basisklassenmethode *OnLButtonDown* der Klasse *CDialog* aufzurufen.

Es ist festgelegt, dass das Hinzufügen von Behandlungsmethoden zu einer Anwendung ohne weitere Editierung deren Inhalts keine Funktionsveränderung des ursprünglichen Programms hervorrufen darf.

Somit laufen Sie nicht Gefahr, ein lauffähiges Programm zu zerstören, nur weil Sie eine in Wirklichkeit gar nicht benötigte Behandlungsmethode aus Versehen in Ihr Projekt eingefügt haben.

Erweiterung der *OnLButtonDown*-Funktionalität

Da die vom Assistenten bereit gestellte Methode nichts tut, lässt sich auch schwer kontrollieren, ob sie überhaupt funktioniert, also beim Klicken mit der linken Maustaste ordnungsgemäß reagiert.

Ausgabe von Testzeilen

Hier bietet sich ein kleiner Trick an, den man immer dann einsetzen kann, wenn man prüfen möchte, ob eine Behandlungsmethode aufgerufen wird – Sie werden in einem späteren Kapitel eine elegantere Möglichkeit hierzu kennen lernen, doch wird sich die nun beschriebene Vorgehensweise auch häufig in anderem Kontext einsetzen lassen.

Überarbeiten Sie die *OnLButtonDown*-Methode nun wie folgt:

Listing 4.6
Ausgabetest

```
void CZeichenProgrammDlg::OnLButtonDown(UINT nFlags, CPoint point)
{
   // Darstellen einer MessageBox
   AfxMessageBox("Ein Test");

   // Standardbehandlung aufrufen
   CDialog::OnLButtonDown(nFlags, point);
}
```

Kompilieren Sie das neue Programm und führen es aus. Klicken Sie dann mit der linken Maustaste ein beliebigen Punkt innerhalb der Client Area an:

Abb. 4.44
Ergebnis des *AfxMessageBox*-Aufrufs

Offensichtlich wird die Maustastenaktivierung registriert und die zugehörige Fensternachricht ordnungsgemäß an die neue Methode weitergeleitet.

Das ist schon recht erfreulich, ermöglicht es uns, nun Schritt für Schritt eine sinnvolle Maustastenbehandlung zu implementieren.

Feststellen der Mausposition

Das Drücken der linken Maustaste soll einen Zeichenvorgang auslösen. Sie hatten hierfür eine Zeichenfläche mithilfe eines Gruppenfelds vorbereitet, innerhalb dessen Grenzen ein Anwender seiner Grafiken erzeugen kann.

Verwendung der *OnLButtonDown*-Parameter

Um nun feststellen zu können, ob der Benutzer in einen erlaubten Bereich geklickt hat, müssen Sie die Mausposition zum Zeitpunkt des Klicks abfragen. Praktischerweise wird diese Information der *OnLButtonDown*-Methode als Parameter *point* direkt mitübergeben:

Listing 4.7
Ausgabe der Mausposition

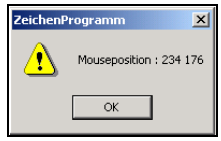
```
void CZeichenProgrammDlg::OnLButtonDown(UINT nFlags, CPoint point)
{
   // Position in String umwandeln
   char lpszAusgabe[256];
   sprintf(lpszAusgabe, "Mouseposition : %i %i",
 point.x, point.y);

   // Position ausgeben
   AfxMessageBox(lpszAusgabe);
```

```
    // Standardbehandlung aufrufen
    CDialog::OnLButtonDown(nFlags, point);
}
```

CPoint ist eine der Werkzeugklassen der MFC und ist äquivalent zum vielleicht bekannten Struct *POINT*. Die Klasse enthält also insbesondere eine X- und eine Y-Komponente als Integerwerte, die in dieser *OnLButtonDown*-Methode via *sprintf*-Funktion in einen String transferiert werden.

Ein Ergebnis des Programmablaufs (nach Anklicken eines Bereichs innerhalb der Zeichenfläche) könnte nun zum Beispiel so aussehen:

Abb. 4.45
Eine Beispielausgabe



Diese Mausposition ist relativ zur linken oberen Ecke der Client Area anzusehen (nur hier werden Mausklicks entgegen genommen).

Auslesen von Steuerelementinformationen

Nachdem Sie nun wissen, an welche Stelle ein Benutzer geklickt hat, ist es weiterhin wichtig zu prüfen, ob dieser Bereich innerhalb der Gruppenfeldumrandung liegt.

Bestimmen der Gruppenfeldgröße

Da Sie das Gruppenfeld in jeder beliebigen Größe aufgezogen haben könnten, bietet es sich hier nicht an, mit absoluten Zahlwerten zu arbeiten und beispielsweise zu sagen: *Jeder Klick im Rechteck von (10, 10) bis (100, 100) relativ zur linken oberen Ecke der Client Area ist gültig*.

Vielmehr müssen die hier fixierten Werte durch die tatsächlichen Koordinaten des Gruppenfelds ersetzt werden.

Wie Sie an diese Informationen gelangen können, zeigt die nächste Erweiterungsstufe von *OnLButtonDown*:

Listing 4.8
Ermitteln der Zeichenflächenkoordinaten

```
void CZeichenProgrammDlg::OnLButtonDown(UINT nFlags, CPoint point)
{
    // lokale Variablen deklarieren
    char lpszAusgabe[256];
    RECT FensterRechteck;

    // Position und Größe des Feldes auslesen
    GetDlgItem(IDC_ZEICHENFLAECHE)
->GetWindowRect(&FensterRechteck);
```

```
   // Ausgabe der ermittelten Informationen
   sprintf(lpszAusgabe, "Gruppenfeldposition : %i
 %i %i %i", FensterRechteck.left,
       FensterRechteck.top,
FensterRechteck.right,
FensterRechteck.bottom);

   AfxMessageBox(lpszAusgabe);

   // Koordinaten von Schirm- in Clientwerte
// umrechnen
   ScreenToClient(&FensterRechteck);
   sprintf(lpszAusgabe, "Gruppenfeldposition : %i
%i %i %i", FensterRechteck.left,
       FensterRechteck.top,
FensterRechteck.right,
FensterRechteck.bottom);

   AfxMessageBox(lpszAusgabe);

   // Standardbehandlung
   CDialog::OnLButtonDown(nFlags, point);
}
```

Diese Version sieht schon etwas komplizierter als ihre Vorgänger aus – was an einer kleinen Problematik des Koordinatengültigkeitsbereichs liegt, wie Sie gleich sehen werden.

Auslesen von Zeigern auf Steuerelemente

Zugriff auf Steuerelemente

Zunächst ist jedoch die Funktion *GetDlgItem* zu erwähnen, der als Parameter die ID eines Steuerelements übergeben wird, und die als Rückgabewert gerade einen Zeiger auf das zur ID gehörende Steuerelement zurückliefert.

Prüfung auf Fehler

Wird eine ungültige ID spezifiziert, ist der Rückgabewert NULL. Aus Gründen der Übersichtlichkeit wurde hier jedoch auf eine explizite Fehlerabfrage verzichtet. Das Gleiche gilt für die weiteren Beispiele dieses Buches. Der Leser ist angehalten, sich stets um eine vernünftige Absicherung seiner Programme zu sorgen, sodass fehlerhafte oder unerwartete Rückgabewerte möglichst früh abgefangen werden und nicht später im Programmablauf für Probleme sorgen.

Daten aus Steuerelementen auslesen

Haben Sie erst einmal einen Zeiger auf ein Steuerelement, können Sie über die Membermethode *GetWindowRect* leicht die so genannte *Boundingbox* der Kontrolle auslesen. Eine Boundingbox ist der weit verbreitete englische Begriff für ein eine Sache umschließendes Rechteck, beziehungsweise einen umschließenden Kasten im dreidimensionalen Raum.

Der Haken bei der Sache offenbart sich beim Anzeigen der ermittelten Werte: sie sind absolut beziehungsweise relativ zur linken oberen Ecke des Bildschirms, nicht zu der der Client Area.

Koordinatenumrechnungen

Es muss also geeignet umgerechnet werden, was die Funktion *ScreenToClient* erledigt, die als Argument einen Zeiger auf eine Rechteckstruktur übernimmt, die wiederum im Laufe der Funktionsabarbeitung auf die relativen Clientwerte umgerechnet wird.

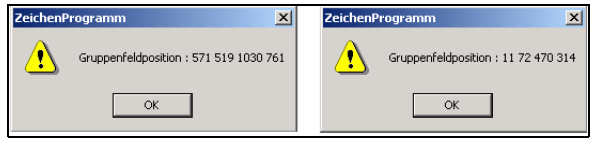
Systemausrichtungen

Bei der Entwicklung unter Windows müssen Sie stets darauf achten, mit den richtigen Werteinheiten zu arbeiten, insbesondere darauf, ob sie im richtigen System oder relativ zu einem sinnvollen Ursprung angegeben sind.

Im Falle des Mausklicks wäre es zum Beispiel wenig sinnvoll, den relativen Mausklickpunkt mit dem absoluten Gruppenfeldpunkt zu vergleichen – das Resultat wäre im günstigsten Fall verwirrend, aber vermutlich ohne Kenntnisse über diese unterschiedlichen Koordinatenangaben nur schwer zu korrigieren.

Kompilieren Sie das neue Programm und führen Sie es aus. Das Ergebnis der Werteumrechnung wird Ihnen nach einem Mausklick in die Client Area des Dialogs präsentiert:

Abb. 4.46
Position des Gruppenfelds



Abfragen der Gruppenfeldgrenzen

Nachdem nun sämtliche relevanten Werte feststehen (also die Position des Mausklicks und die Position und Dimension des Gruppenfelds), können Sie über einfache *if*-Vergleiche feststellen, ob nach einem Mausklick ein Punkt gezeichnet werden soll, oder nicht.

Eine mögliche Lösung für diese Fragestellung könnte wie folgt aussehen:

Listing 4.9
Zeichnen von Punkten

```
void CZeichenProgrammDlg::OnLButtonDown(UINT nFlags,
CPoint point)
{
   // lokale Variablen deklarieren
   RECT FensterRechteck;

   // Position und Größe des Feldes auslesen
   GetDlgItem(IDC_ZEICHENFLAECHE)
```

```
->GetWindowRect(&FensterRechteck);

   // Koordinaten von Schirm- in Clientwerte
// umrechnen
   ScreenToClient(&FensterRechteck);

   // Offset einrichten, um nicht die Ränder zu
// überzeichnen.
   // oberen Rand.
   int nOffset = 8;

   // prüfen, ob sich der Mausklick innerhalb der
// Zeichenfläche befand
   if ((point.x >= FensterRechteck.left + nOffset)
      && (point.x <= FensterRechteck.right -
nOffset)
      && (point.y >= FensterRechteck.top +
nOffset)
      && (point.y <= FensterRechteck.bottom -
nOffset))
   {
      CClientDC dc(this);
      dc.SetPixel(point.x, point.y, RGB(0,0,0));
   }

   // Standardbehandlung
   CDialog::OnLButtonDown(nFlags, point);
}
```

Beim Überprüfen der Klickposition wird in der obigen Methode ein *Offset* (also Abstand) von 8 Pixeln mit in die Überprüfung eingerichtet. Damit wird dem Umstand entgegengebeugt, dass es sich bei der Boundingbox des Gruppenfelds ja tatsächlich um ein umschließendes Rechteck handelt, das also auch die Beschriftung am oberen Rand des Felds beinhaltet. Dieser Bereich soll natürlich nicht übermalt werden können, sodass durch den Offset eine Marge von 8 Pixeln freigehalten wird. Das reicht bei den Standardzeichensatzeinstellungen gerade aus, um den Rand des Gruppenfelds von Zeicheneinflüssen frei zu halten.

Gerätekontexte

Nun, da Sie endlich wissen, ob der Benutzer einen gültigen Punkt angeklickt hat, ist es interessant zu wissen, wie man diesen Punkt letztendlich auch zeichnen kann. Hierzu ist es zunächst erforderlich, einen groben Überblick über das Konzept der Gerätekontexte zu erlangen.

Zeichenflächen in der Windows-Programmierung

Gerätekontexte sind inhaltlich eine Art Zeichenfläche, auf der beliebige Informationen eingetragen werden können – also beispielsweise die von einem Benutzer gezeichneten Punkte oder auch eine einfache Textausgabe.

Gerätekontexte kommen innerhalb des Konzepts der *GDI* (*Graphic Device Interface*, etwa: Schnittstelle für grafikfähige Ausgabegeräte) vor. Die GDI stellen eine Abstraktionsschicht zwischen Ihnen und dem tatsächlichen Ausgabegerät (einem Monitor etwa, oder einem Drucker) dar.

Durch diese Abstraktionsschicht ist es möglich, dass Sie als Entwickler mit einem Gerätekontext in beliebiger Weise arbeiten können und sich nicht darum sorgen müssen, wie die von Ihnen eingefügten Informationen auf dem Zielgerät ausgegeben werden. Der Vorteil liegt auf der Hand: befinden sich die gewünschten Daten erst einmal im Gerätekontext, können Sie (durch die internen Funktionen der GDI) ohne Probleme und vor allem ohne weiteren programmiertechnischen Aufwand auf jedes beliebige Ausgabegerät weitergeleitet werden.

Ausgabe auf verschiedene Medien

Wenn Sie also ein Zeichenprogramm geschrieben haben, dass eine Grafik auf dem Bildschirm darstellt, können Sie die zugehörige Grafik genauso gut auf den Drucker ausgeben, ohne irgendwelche Einstellungen an Ihrem Programm verändern zu müssen.

In den meisten Fällen ist es ausreichend, sich den Gerätekontext als eine Art Leinwand vorzustellen und so soll es innerhalb dieses Buches auch gehandhabt werden. Sind hiervon abweichende Betrachtungsweisen notwendig, werden Sie an entsprechender Stelle darauf hin gewiesen.

Punkte zeichnen auf dem Gerätekontext

Der Zeichenvorgang setzt sich zusammen aus dem Anfordern des Gerätekontexts für das Dialogfeld und dem eigentlichen Setzen des Punkts.

CClientDC

Das Anfordern geschieht über das Erzeugen eines neuen Objekts vom Typ *CClientDC*. Der Name deutet es bereits an, es geht hierbei um einen Gerätekontext, der den Client-Bereich des Dialogs virtuell repräsentiert. Der Konstruktorparameter *this* stellt klar, dass ein Gerätekontext für die eigene Instanz, in unserem Fall das Hauptdialogfeld, erbeten wird.

Wurde das Objekt erfolgreich initialisiert, können Sie seine Membermethoden aufrufen, von denen eine zum Setzen von Punkten dient. Sie benötigt als Parameter den relativen Offset des Punkts zum Clientursprung und die gewünschte Farbe des zu setzenden Punkts.

Farbwerte

Letzterer wird als *COLORREF*-Wert spezifiziert, Sie erzeugen solche Werte durch Verwendung des Makros *RGB*, das als Parameter den Rot-, Grün sowie Blauanteil des zu erzeugenden Farbwerts enthält. RGB (0, 0, 0) erzeugt beispielsweise gerade den RGB-Wert für die Farbe schwarz.

Starten Sie nun das Programm und versuchen Sie, einige Punkte zu setzen:

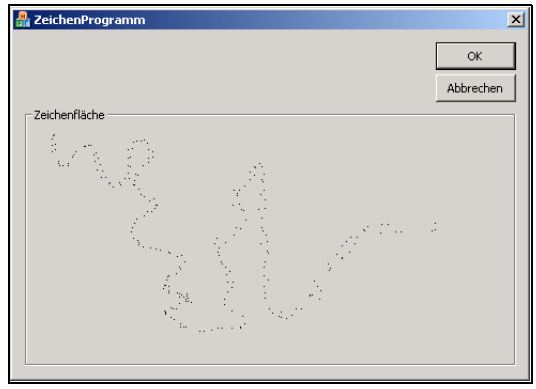


Abb. 4.47
Eine Beispielgrafik

Mausbewegungen registrieren

Bislang wird nur die *WM_LBUTTONDOWN* Nachricht behandelt, ein Anwender des Zeichenprogramms kann also nur einzelne Punkte setzen.

Zeichnen von Punkten

Diese Funktionalität soll nun dahingehend erweitert werden, dass nach dem Drücken der linken Maustaste auch nach Mausbewegungen an die neuen Mauspositionen Punkte gemalt werden, bis die Taste wieder losgelassen wird.

Fügen Sie daher nun, analog zur oben beschriebenen Vorgehensweise, Behandlungsmethoden für die Fensternachrichten *WM_MOUSEMOVE* und *WM_LBUTTONUP* ein. Ergänzen Sie weiterhin die Klassendeklaration von *CZeichenProgrammDlg* um eine boolesche Variable zur Aufnahme des aktuellen Status der linken Maustaste.

Ihre Headerdatei *ZeichenProgrammDlg.h* sollte dann wie folgt aussehen:

Listing 4.10
Die überarbeitete Headerdatei

```
// ZeichenProgrammDlg.h : Headerdatei
//

#pragma once


// CZeichenProgrammDlg Dialogfeld
class CZeichenProgrammDlg : public CDialog
{
// Konstruktion
public:
   CZeichenProgrammDlg(CWnd* pParent = NULL);
// Standardkonstruktor
```

```
// Dialogfelddaten
   enum { IDD = IDD_ZEICHENPROGRAMM_DIALOG };

   protected:
   virtual void DoDataExchange(CDataExchange* pDX);// DDX/DDV-
Unterstützung


// Implementierung
protected:
   HICON m_hIcon;
   bool  m_bLinkeMaustasteGedrueckt;

   // Generierte Funktionen für die
// Meldungstabellen
   virtual BOOL OnInitDialog();
   afx_msg void OnSysCommand(UINT nID, LPARAM
lParam);
   afx_msg void OnPaint();
   afx_msg HCURSOR OnQueryDragIcon();
   DECLARE_MESSAGE_MAP()
public:
   afx_msg void OnLButtonDown(UINT nFlags, CPoint
point);
   afx_msg void OnMouseMove(UINT nFlags, CPoint
point);
   afx_msg void OnLButtonUp(UINT nFlags, CPoint
point);

};
```

Speichern des Status der linken Maustaste

Die neue boolesche Variable wird im Konstruktor (Datei *ZeichenProgramm-Dlg.cpp*) initialisiert:

Listing 4.11
Initialisierung der Mausstatus-Variablen

```
CZeichenProgrammDlg::CZeichenProgrammDlg(
CWnd* pParent /*=NULL*/)
   : CDialog(CZeichenProgrammDlg::IDD, pParent)
{
   m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
   m_bLinkeMaustasteGedrueckt = false;
}
```

Vielleicht ist Ihnen beim bisherigen Zeichnen aufgefallen, dass der eingestellte Offset für die seitlichen und den Rand zu groß wirken. Dieser Umstand soll nun gleich mitbehoben werden, wenn wir in *OnLButtonDown* das Setzen der booleschen Variable *m_bLinkeMaustasteGedrueckt* einfügen:

Listing 4.12
Anpassen der Zeichenoffsets und Speichern des Mausstatus

```
void CZeichenProgrammDlg::OnLButtonDown(UINT nFlags,
  CPoint point)
{
   // lokale Variablen deklarieren
   RECT FensterRechteck;

   // Position und Größe des Feldes auslesen
   GetDlgItem(IDC_ZEICHENFLAECHE)
->GetWindowRect(&FensterRechteck);

   // Koordinaten von Schirm- in Clientwerte
// umrechnen
   ScreenToClient(&FensterRechteck);

// Offsets einrichten, um nicht die Ränder zu
// überzeichnen. Da
// die Gruppenbox oben einen breiten Rahmen hat
// (mit der Beschriftung
// Zeichenfläche) spendieren wir ihr einen
// eigenen Offset für den
// oberen Rand.
   int nTopOffset = 8;
   int nCommonOffset = 3;

   // prüfen, ob sich der Mausklick innerhalb der
// Zeichenfläche befand
   if ((point.x >= FensterRechteck.left + nCommonOffset)
      && (point.x <= FensterRechteck.right - nCommonOffset)
      && (point.y >= FensterRechteck.top + nTopOffset)
      && (point.y <= FensterRechteck.bottom - nCommonOffset))
   {
      CClientDC dc(this);
      dc.SetPixel(point.x, point.y, RGB(0,0,0));
      m_bLinkeMaustasteGedrueckt = true;
   }

   // Standardbehandlung
   CDialog::OnLButtonDown(nFlags, point);
}
```

Behandeln von Mausbewegungen

Um auf eine Mausbewegung zu reagieren, muss die Behandlungsmethode *OnMouseMove* editiert werden. Ihr Funktionsrumpf wurde vom Klassenassistenten angelegt, als Sie eine Nachrichtenbehandlung für *WM_MOUSEMOVE* zu Ihrem Projekt hinzugefügt haben.

OnMouseMove erhält, wie *OnLButtonDown*, den zum Client-Area-Ursprung relativen Offset, an dem sich der Mauszeiger nach einer Bewegung befindet.

Ergänzen der *OnMouseMove*-Methode

Editieren Sie die Datei (zu finden in *ZeichenProgrammDlg.cpp*):

Listing 4.13 **Behandlung von Mausbewegungen**

```
void CZeichenProgrammDlg::OnMouseMove(UINT nFlags, CPoint point)
{
   // wird die Maus bewegt und ist die linke
// Maustaste gedrückt,
   // soll erneut ein Punkt gezeichnet werden
   if (m_bLinkeMaustasteGedrueckt)
   {
      CClientDC dc(this);
      dc.SetPixel(point.x, point.y, RGB(0,0,0));
   }

   CDialog::OnMouseMove(nFlags, point);
}
```

Wie bei *OnLButtonDown* wird hier einfach der Gerätekontext des Client-Bereichs angefordert und ein schwarzer Pixel an die neue Mausposition gesetzt.

Loslassen der linken Maustaste

Beenden eines Zeichenvorgangs

Zum Schluss muss noch das Loslassen der linken Maustaste registriert werden, um einen eingeleiteten Zeichenvorgang wieder zu beenden. Lässt ein Anwender die linke Maustaste los, wird eine *WM_LBUTTONUP* Nachricht an das Fenster geschickt, editieren Sie die zugehörige Behandlungsmethode *OnLButtonUp* wie folgt:

Listing 4.14 **Loslassen der linken Maustaste**

```
void CZeichenProgrammDlg::OnLButtonUp(UINT nFlags, CPoint point)
{
   // linke Maustaste ist nicht mehr gedrückt
   m_bLinkeMaustasteGedrueckt = false;

   CDialog::OnLButtonUp(nFlags, point);
}
```

Testen Sie das so veränderte Programm aus und zeichnen Sie einige Figuren in das Fenster. Was fällt Ihnen dabei auf?

Fehler im Programm

Die nachstehend abgedruckt Abbildung gibt einen Überblick über die Fehler, die sich bei diesen Versuchen offenbaren werden:

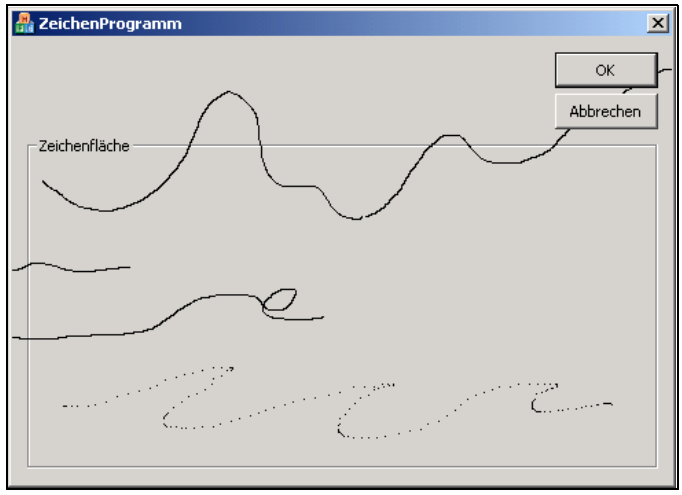


Abb. 4.48
Die derzeitigen Fehler des Programms im Überblick

Die drei sichtbaren Linienzüge sind auf folgende Art und Weise entstanden:

- Der obere Linienzug wurde korrekt innerhalb der Zeichenfläche begonnen, doch hat der Benutzer dann den Mauszeiger aus der Fläche hinausbewegt. Da in *OnMouseMove* keine weitere Positionsabfragen vorgenommen werden, konnte der Anwender so in den eigentlich nicht erlaubten Bereich hineinzeichnen.

  Fehler 1: Überschreiten gültiger Bereiche

- Der mittlere Linienzug wurde ebenfalls korrekt innerhalb der Zeichenfläche begonnen. Der Benutzer hat dann die Maus aus dem Fenster hinaus bewegt und dort, außerhalb des Dialogs, die Maustaste losgelassen. Offensichtlich wurde die *OnLButtonUp*-Methode nicht aufgerufen, denn beim Wiedereintritt des Mauszeigers in den Client-Bereich des Dialogs wurde der Zeichenvorgang trotz nicht gedrückter Maustaste fortgesetzt.

  Fehler 2: Weiterzeichnen nach Loslassen der Maustaste

- Der untere Linienzug wurde korrekt innerhalb der Zeichenfläche begonnen und beendet, doch hat der Benutzer den Mauszeiger mit schnellen Bewegungen geführt. Scheinbar wurden dabei nicht alle Positionen des Mauszeigers an das Fenster geliefert, sodass zwischen den einzelnen Punkten deutliche Abstände zu sehen sind.

  Fehler 3: Lücken beim Zeichnen

Diese drei Fehler sollen nun behoben werden.

Hinzufügen neuer Methoden

Wenn Sie in C++ zu einer Klasse eine neue Methode hinzufügen möchten, fügen Sie diese manuell in die Klassendeklaration ein und implementieren Sie dann normalerweise außerhalb dieser Deklaration für gewöhnlich in einer separaten Implementationsdatei – Unterschiede gibt es zum Beispiel bei Inline-Funktionen, doch sollen derartige Sonderfälle an dieser Stelle nicht betrachtet werden.

Das Visual Studio bietet eine Reihe von Hilfestellungen für die Arbeit mit Klassen und deren Funktionen, unter anderem eine Möglichkeit, neue Methoden mithilfe eines Assistenten zu entwerfen und automatisch in das bestehende Klassengerüst einzufügen.

Testen, ob gültige Zeichenposition vorliegt

Diese Vorgehensweise soll nun demonstriert werden, indem eine Methode zum Testen einer gültigen Zeichenposition geschrieben wird – scheinbar brauchen wir diesen Test sowohl in *OnLButtonDown* als auch in *OnMouseMove*, sodass es wenig Sinn machen würde, die betreffenden Zeilen in beide Methoden einzubinden.

Die Bedeutung von Funktionen bei der Windows-Programmierung

Es sollte jedem Programmierer klar sein, dass gleichartige Funktionalitäten nach Möglichkeit immer in Funktionen zusammengefasst werden sollten, die dann von den verschiedenen Stellen im Quelltext aufgerufen werden können.

Gerade bei der Windows-Programmierung ist dieses Verfahren umso wichtiger, da es häufig vorkommt, dass gleiche oder ähnliche Funktionalitäten an vielen unterschiedlichen Stellen innerhalb eines Programms benötigt werden.

Es gibt viele Faustregeln, nach denen man vorgehen kann um zu entscheiden, ab wann es sinnvoll ist, Funktionen zu schreiben, um Funktionalität zusammenzufassen. Allgemein gilt, dass es schon in dem Augenblick Sinn macht, in dem eine bestimmte Aktion mehr als einmal innerhalb eines Programms vorkommen soll, die mehr als eine Zeile besitzt.

Natürlich kommt es dabei jeweils auf den Einzelfall an, doch sollten Sie sich in Ihren Überlegungen diesbezüglich nie von der Sorge des Geschwindigkeitsverlusts der Anwendung leiten lassen – definieren Sie im Zweifelsfall die betreffenden Funktionen einfach als Inline.

In jedem Fall wird sich die Zeitersparnis – weniger Tipparbeit und vor allem weniger Wartungszeit, falls sich die Funktionalität einmal ändert – in sehr kurzer Zeit bemerkbar machen.

Die Solution-Klassen

Hinzufügen neuer Methoden

Um eine neue Methode in das bestehende Klassengerüst einzufügen, öffnen Sie zunächst die Klassenansicht Ihres Projekts (zu finden innerhalb des Arbeitsbereichs, der auch den Solution Explorer enthält).

Sie finden hier eine Übersicht über die bereits im Projekt befindlichen Klassen – erweitern Sie die Klasse *CZeichenProgrammDlg* und Sie erhalten eine Übersicht ähnlich der folgenden:

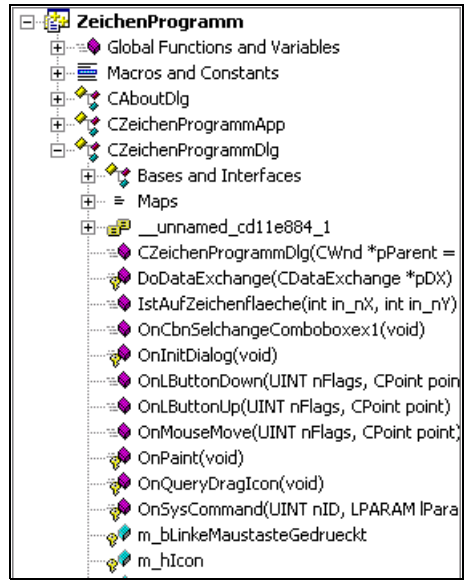


Abb. 4.49
Ansicht der Solution-Klassen

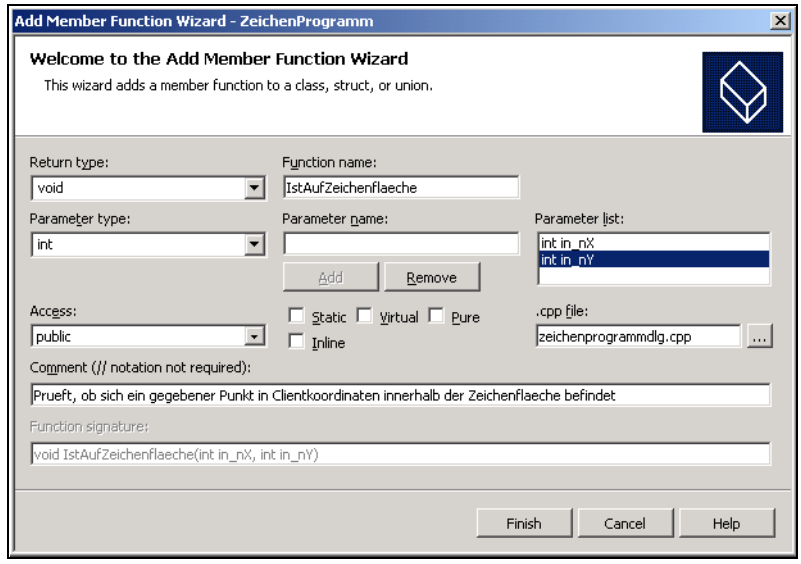
Übersicht über Klasseninhalte

Die Liste enthält sämtliche Methoden, Variablen, Basisklassen, Enumeration, Strukturen und dergleichen mehr (unter anderem auch eine Übersicht der von der Klassen verarbeiteten Fensternachrichten), die für die ausgewählte Klasse definiert sind.

Ein Rechtsklick auf einen Klassennamen öffnet ein Kontextmenü, aus dem Sie nun den Punkt *Add* > *New Function* auswählen. Dieses Kommando dient zum Einfügen einer neuen Methode in die Klasse.

Es öffnet sich das nachstehende Assistentenfenster:

Abb. 4.50
Dialogfeld zum Einfügen neuer Methoden



Festlegen eines Funktionsprototyps

Sie können hier sämtliche Einstellungen vornehmen, die zum Erzeugen eines Funktionsprototyps notwendig sind, angefangen vom gewünschten Rückgabetyp, über benötigte Parameter (die über *Add* in eine Liste aufgenommen und mit *Remove* von dort wieder entfernt werden) bis hin zu Zugriffsrechten, zusätzlichen Modifikatoren (*Inline, Virtual, Pure* und *Static*) und einem Kommentar, der über dem Funktionskopf ausgegeben wird.

Tragen Sie in dieses Feld als Funktionsnamen *IstAufZeichenflaeche*, als Rückgabewert void, zwei Integer-Parameter mit den Namen *in_nX* und *in_nY* sowie den Kommentar *Prueft, ob sich ein gegebener Punkt in Clientkoordinaten innerhalb der Zeichenflaeche befindet* ein und bestätigen die Einstellungen mit dem Anklicken von *Fertig stellen*.

Prüfen Sie dann in der Datei *ZeichenProgrammDlg.h*, ob die Funktion ordnungsgemäß eingebunden wurde. Sie sollten folgende Zeilen vorfinden:

Listing 4.15
Die Deklaration der neuen Methode

```
// Prueft, ob sich ein gegebener Punkt in Clientkoordinaten
innerhalb der Zeichenflaeche befindet
   bool IstAufZeichenflaeche(int in_nX, int in_nY);
```

Prüfen, ob sich ein Punkt innerhalb der Zeichenfläche befindet

Auch in *ZeichenprogrammDlg.cpp* wurde ein einfacher Funktionsrumpf eingefügt, erweitern Sie ihn entsprechend der nachstehenden Zeilen:

Listing 4.16
Die neue Methode
IstAufZeichenflaeche

```
// Prueft, ob sich ein gegebener Punkt in
// Clientkoordinaten innerhalb der Zeichenflaeche
// befindet
bool CZeichenProgrammDlg::IstAufZeichenflaeche(
int in_nX, int in_nY)
{
   // lokale Variablen deklarieren
   RECT FensterRechteck;

   // Position und Größe des Feldes auslesen
   GetDlgItem(IDC_ZEICHENFLAECHE)
->GetWindowRect(&FensterRechteck);

   // Koordinaten von Schirm- in Clientwerte
// umrechnen
   ScreenToClient(&FensterRechteck);

   // Offsets einrichten, um nicht die Ränder zu
// überzeichnen. Da
   // die Gruppenbox oben einen breiten Rahmen hat
// (mit der Beschriftung
   // Zeichenfläche) spendieren wir ihr einen
// eigenen Offset für den
   // oberen Rand.
   int nTopOffset = 8;
   int nCommonOffset = 3;

   // prüfen, ob sich der Mausklick innerhalb der
// Zeichenfläche befand
   if ((in_nX >= FensterRechteck.left +
nCommonOffset)
      && (in_nX <= FensterRechteck.right -
nCommonOffset)
      && (in_nY >= FensterRechteck.top +
nTopOffset)
      && (in_nY <= FensterRechteck.bottom -
nCommonOffset))
   {
// der Punkt befindet sich auf der
// Zeichenflaeche
      return (true);
   }
   else
   {
      // der Punkt ist ausserhalb des
// beschreibbaren Bereichs
      return (false);
}
}
```

Es handelt sich hierbei gerade um die Zeilen, die auch in *OnLButtonDown* schon zum Einsatz kamen, jedoch liefert die Methode *IstAufZeichenflaeche* als Rückgabe den Wahrheitsgehalt der Aussage, ob sich die übergebene Position innerhalb der Zeichenfläche befindet (*true* bedeutet dabei ein positives Beantworten der Frage, *false*, dass sich der Punkt außerhalb des Bereichs befindet).

Verwenden der neuen Prüfmethode

Passen Sie weiterhin die Methode *OnLButtonDown* auf die neuen Gegebenheiten an:

Listing 4.17 **Überabeitete Version von *OnLButtonDown***

```
void CZeichenProgrammDlg::OnLButtonDown(UINT nFlags, CPoint point)
{
   if (IstAufZeichenflaeche(point.x, point.y))
   {
      CClientDC dc(this);
      dc.SetPixel(point.x, point.y, RGB(0,0,0));
      m_bLinkeMaustasteGedrueckt = true;
   }

   // Standardbehandlung
   CDialog::OnLButtonDown(nFlags, point);
}
```

Auch *OnMouseMove* kann nun um eine Prüfung der aktuellen Mausposition erweitert werden. Dieses ist dank der neuen Methode sehr bequem zu realisieren:

```
void CZeichenProgrammDlg::OnMouseMove(UINT nFlags,
 CPoint point)
{
   // wird die Maus bewegt und ist die linke
// Maustaste gedrückt,
   // soll erneut ein Punkt gezeichnet werden
   if ((m_bLinkeMaustasteGedrueckt) &&
(IstAufZeichenflaeche(point.x, point.y)))
   {
      CClientDC dc(this);
      dc.SetPixel(point.x, point.y, RGB(0,0,0));
   }

   CDialog::OnMouseMove(nFlags, point);
}
```

Testen Sie die neue Variante aus, werden Sie feststellen, dass der weiter oben beschriebene erste Fehler erfolgreich behoben wurde.

Mausnachrichten nach Verlassen des Fensters

Listing 4.18
Die überarbeiteten Mausbehandlungsmethoden

Bei der einführenden Erklärung zu den Fensternachrichten wurde bereits gesagt, dass Windows entscheidet, welches Fenster die Botschaften zugestellt bekommt, nämlich gerade diejenigen, die mit dem erzeugenden Ereignis direkt in Zusammenhang stehen.

Probleme bei fensterspezifischen Nachrichten

Insofern ist es klar, dass der gewählte Ansatz (Benutzer lässt Maustaste los, boolesche Variable *m_bLinkeMaustasteGedrueckt* wird auf *false* gesetzt) so nicht funktionieren kann, wenn der Anwender sich außerhalb des Dialogfelds entscheidet, die Maustaste nicht weiter zu drücken – die zugehörige *WM_LBUTTONUP*-Nachricht geht dann an das Fenster, das sich zu diesem Zeitpunkt gerade unterhalb des Mauszeigers befindet.

Windows bietet jedoch die Möglichkeit, den Gültigkeitsbereich für Mausnachrichten auf ein Fenster zu begrenzen, indem dieses die Maus exklusiv anfordert.

Fixieren der Maus

Die dazu verwendete Funktion heißt *SetCapture* und sorgt dafür, dass auch Nachrichten, die außerhalb des Dialogs entstehen, an ihn weitergeleitet werden. Dieses geschieht so lange, bis durch einen Aufruf von *ReleaseCapture* die Exklusivrechte wieder abgegeben werden.

Fügen Sie die passenden Aufrufe nun in die *OnLButtonDown*- und *OnLButtonUp*-Methoden ein:

Listing 4.19
Festhalten der Maus

```
void CZeichenProgrammDlg::OnLButtonDown(UINT nFlags,
 CPoint point)
{
   // prüfen, ob sich der Punkt auf der
// Zeichenflaeche befindet
   if (IstAufZeichenflaeche(point.x, point.y))
   {
      // ok, Gerätekontext anfordern
      CClientDC dc(this);

      // Pixel zeichnen
      dc.SetPixel(point.x, point.y, RGB(0,0,0));

      // merken, dass Maustaste gedrückt ist
      m_bLinkeMaustasteGedrueckt = true;

      // Maus festhalten
      SetCapture();
   }

   // Standardbehandlung
   CDialog::OnLButtonDown(nFlags, point);
}
```

Listing 4.20
Maus freigeben

```
void CZeichenProgrammDlg::OnLButtonUp(UINT nFlags, CPoint point)
{
   // linke Maustaste ist nicht mehr gedrückt
   m_bLinkeMaustasteGedrueckt = false;

   // Maus freigeben
   ReleaseCapture();

   CDialog::OnLButtonUp(nFlags, point);
}
```

Testen Sie das Programm nun erneut aus, auch der zweite Fehler ist behoben, verbleibt noch die Problematik der versetzten Punkte.

Redundanz bei Mausnachrichten

Optimierungen bei Windows-Nachrichten

Bei der Besprechung der Nachrichtenpuffer unter Windows wurde bereits angedeutet, dass das Betriebssystem Optimierungen dahingehend vornimmt, dass gleiche Botschaften, die in rascher Folge eintreffen und noch nicht verarbeitet wurden, zu einer einzigen zusammengefasst werden.

Das bietet eine nicht zu unterschätzende Geschwindigkeitssteigerung, führt in unserem Fall aber auch zu Problemen, da die Zwischenpunkte, die während einer Mausbewegung erreicht werden, bei schnellen Bewegungen nicht mehr vom Programm registriert werden können (natürlich hängt auch die Einstellung der Mausgeschwindigkeit ganz erheblich mit diesem Phänomen zusammen, einige Punkte werden bei schnellen Bewegungen tatsächlich nie erreicht).

Wenn Sie schon einmal mit einem anderen Zeichenprogramm gearbeitet haben, wird Ihnen aufgefallen sein, dass die Freihandzeichenwerkzeuge dort das hier sichtbare Phänomen nicht aufweisen. Das liegt nicht etwa daran, dass diese einen Trick kennen, um höhere Auflösungsgenauigkeiten zu erzwingen, sondern schlicht daran, dass in Wirklichkeit gar keine Punkte, sondern Linien gezeichnet werden.

Tricks für verbundene Zeichnungen

Stellen Sie sich die weiter oben abgedruckte Punktgrafik mit durch Linien verbundenen Punkten vor. Da die einzelnen Pixel nicht allzu weit auseinander liegen, käme es einem Betrachter tatsächlich so vor, als würde ein runder, geschwungener Kurvenzug dargestellt werden.

Es soll auch für unser Projekt gelten: Linien statt Punkte, doch dafür sind zunächst einige Vorkehrungen zu treffen, insbesondere muss nämlich der zuletzt gezeichnete Punkt gespeichert werden, da eine Linie ja immer einen Start- und einen Endpunkt besitzt. Der gespeicherte Punkt stellt dann den Startpunkt dar, die bei einer Mausbewegung angesteuerte neue Position den Endpunkt.

Hinzufügen neuer Variablen

Wie für Methoden existiert auch ein Assistent zum Hinzufügen neuer Variablen. Dieser soll nun genutzt werden, um zwei Membervariablen zur Aufnahme der zuletzt bekannten Zeichenposition einzufügen.

Öffnen Sie die Klassenansicht, klicken Sie mit der rechten Maustaste auf die *CZeichenProgrammDlg*-Klasse und wählen Sie das Kommando *Add* > *New Variable* aus dem erscheinenden Kontextmenü:

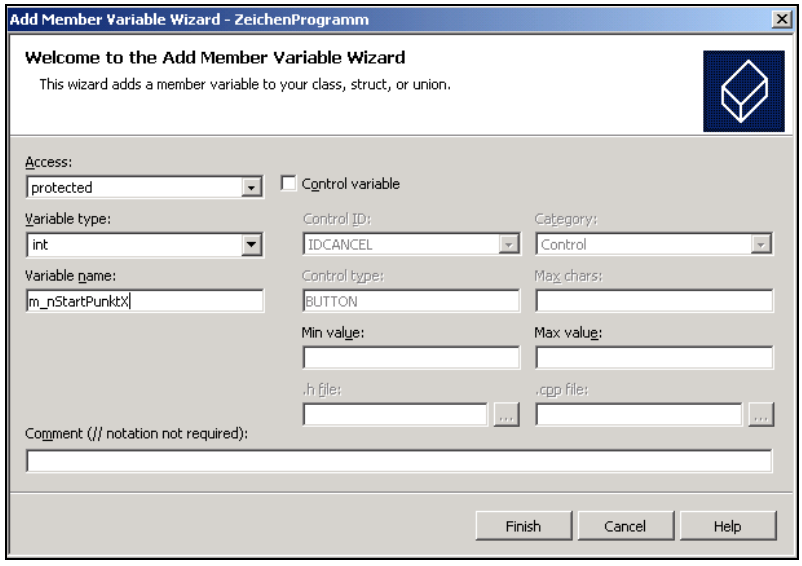


Abb. 4.51
Dialogefeld zum Hinzufügen neuer Variablen

Dieser Dialog ähnelt dem für die Methoden und bietet wiederum sämtliche Einstellmöglichkeiten, die man sich wünschen könnte – und darüber hinaus einige mehr, die Ihnen derzeit noch nichts sagen werden, und zu denen wir in einem späteren Kapitel zurückkehren.

Einstellungen für neue Variablen

An dieser Stelle sei nur auf die Felder *Min Value* und *Max Value* hingewiesen, die gerade die bei DDX/DDV-Besprechung erwähnten Grenzwerte für gültige Daten angeben. Sie sind an dieser Stelle nicht weiter relevant.

Tragen Sie als Variablentyp *int* und als *-namen m_nStartPunktX* ein, der Zugriffsbereich wird mit *Protected* angegeben. Bestätigen Sie die Angaben durch Anklicken von *Fertig stellen* und wiederholen Sie den Vorgang für eine Variable *m_nStartPunktY*.

Prüfen Sie in *ZeichenprogrammDlg.h*, ob die Variablen ordnungsgemäß eingefügt wurden:

Listing 4.21 **Die angelegten Variablen**

```
protected:
   int m_nStartPunktX;
   int m_nStartPunktY;
```

Speichern von Mauszeigerpositionen

Anpassen von Behandlungsmethoden

Die beiden Methoden, die nun überarbeitet werden müssen sind *OnLButtonDown* und *OnMouseMove*, denn dort werden Zeichenoperationen ausgeführt und neue Mauszeigerpositionen verarbeitet. Diese sollen nun auch gesichert werden:

Listing 4.22 **Speichern einer Startposition**

```
void CZeichenProgrammDlg::OnLButtonDown(UINT nFlags,
CPoint point)
{
   // prüfen, ob sich der Punkt auf der
// Zeichenflaeche befindet
   if (IstAufZeichenflaeche(point.x, point.y))
   {
      // ok, Gerätekontext anfordern
      CClientDC dc(this);

      // Pixel zeichnen
      dc.SetPixel(point.x, point.y, RGB(0,0,0));

      // merken, dass Maustaste gedrückt ist
      m_bLinkeMaustasteGedrueckt = true;

      // Maus festhalten
      SetCapture();

      // Startposition merken
      m_nStartPunktX = point.x;
      m_nStartPunktY = point.y;
   }

   // Standardbehandlung
   CDialog::OnLButtonDown(nFlags, point);
}
```

Zeichnen von Linienzügen

Tiefer gehende Umstellungen bedingt die Methode *OnMouseMove*, da Sie hier die Darstellung vom Setzen einzelner Pixel auf das Ziehen von Linien umstellen müssen:

Listing 4.23
Linien statt Punkte zeichnen

```
void CZeichenProgrammDlg::OnMouseMove(UINT nFlags,
CPoint point)
{
   // wird die Maus bewegt und ist die linke
// Maustaste gedrückt,
   // soll erneut ein Punkt gezeichnet werden
   // (aufgrund von Nachrichtenreduktion wird eine
   // Linie gezeichnet, um Abstände zwischen zwei
   // Punkten zu interpolieren)
   if ((m_bLinkeMaustasteGedrueckt) &&
(IstAufZeichenflaeche(point.x, point.y)))
   {
      // Gerätekontext anfordern
      CClientDC dc(this);

      // Startpunkt anfahren
      dc.MoveTo(m_nStartPunktX, m_nStartPunktY);

      // Linie bis zum Endpunkt ziehen
      dc.LineTo(point);

      // neue Position speichern
      m_nStartPunktX = point.x;
      m_nStartPunktY = point.y;
   }

   CDialog::OnMouseMove(nFlags, point);
}
```

Das Zeichnen einer Linie erweist sich als relativ einfach. Zunächst fahren Sie mit der Gerätekontextfunktion *MoveTo* den Punkt an, an dem die Startposition der Linie liegen soll.

Initialisieren eines Zeichenvorgangs

Es ist dem Benutzer nur möglich, Linien zu zeichnen, wenn er zuvor auf einen Startpunkt durch Drücken der linken Maustaste innerhalb eines gültigen Bereichs auf der Zeichenfläche gesetzt hatte.

Somit sind auch die Integerwerte *m_nStartPunktX* und *m_nStartPunktY* bereits geeignet initialisiert, sobald die *WM_MOUSEMOVE*-Nachricht behandelt wird.

Der eigentliche Zeichenvorgang beginnt dann durch den Aufruf von *LineTo*, das eine Linie zum angegeben Punkt (gerade der neuen Mausposition) zieht.

Dieser wird dann in der Folge als neuer Startpunkt festgelegt und erlaubt so das Kreieren geschlossener Linienzüge, wie Sie leicht nachvollziehen können, wenn Sie das fehlerbereinigte Programm nun kompilieren und starten:

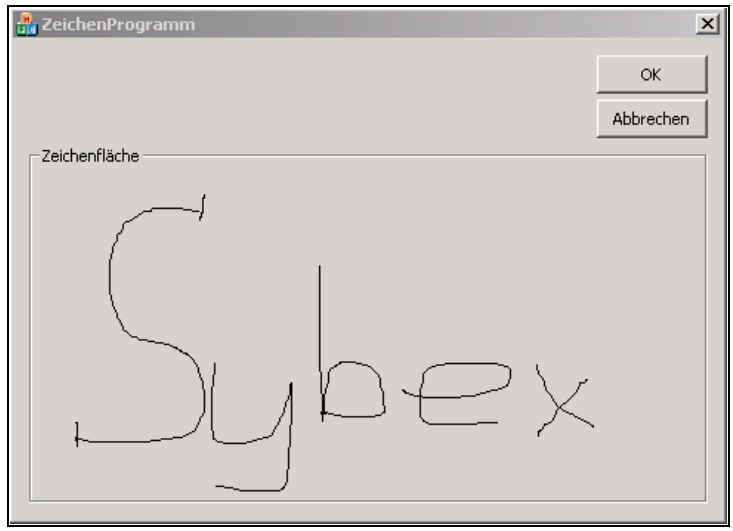


Abb. 4.52 **Ergebnis des Linienzeichnens**

Farbauswahl

Erweiterung des Programms

Das Zeichenprogramm in seiner bisherigen Form ist noch relativ trist und soll ein wenig aufgepeppt werden. Dazu bietet es sich an, eine Möglichkeit zum Auswählen von unterschiedlichen Farben anzubieten.

Wir können diese Programmerweiterung nutzen, um etwas vertrauter mit dem Hinzufügen neuer Steuerelemente, dem Nutzen der Dialogfeld-Layoutfunktionen und nicht zuletzt der Behandlung von Kontrollelement-Nachrichten zu werden.

Öffnen Sie in der Ressourcenansicht den Hauptdialog und fügen vier Schaltflächen hinzu (dabei brauchen Sie nicht sonderlich auf eine korrekte Platzierung achten). Ändern Sie deren Beschriftungen mithilfe der Eigenschaftentabelle auf *Schwarz*, *Rot*, *Grün* und *Blau* und vergeben Sie in derselben Tabelle die IDs *IDC_SCHWARZ*, *IDC_ROT*, *IDC_GRUEN* und *IDC_BLAU* an die neuen Knöpfe.

Obwohl Sie mit einer deutschen Version des Visual Studio .NET arbeiten, kann es doch zu Problemen führen, wenn Umlaute in IDs verwendet werden – schreiben Sie also statt ä, ü, ö und ß immer ae, ue, oe und ss.

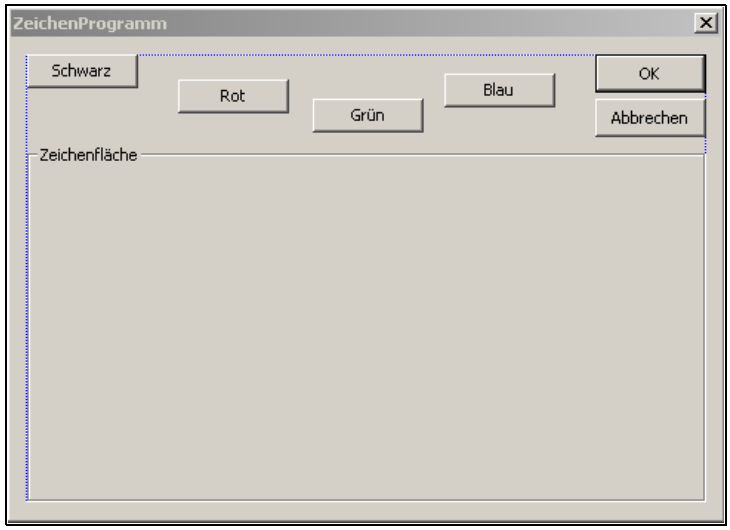


Abb. 4.53
Die neuen Buttons in ungeordneter Form

Layout-Optionen für Dialogfelder

Die Buttons befinden sich zwar nun auf dem Dialogfeld, sehen aber nicht unbedingt sonderlich schön sortiert aus.

Formatieren von Dialogen

Sie finden allerdings ein Menü namens *Format*, wenn Sie mit Dialogen arbeiten, das eine ganze Reihe von Formatierungshilfsmitteln zur Verfügung stellt, unter anderem auch solche zum Ausrichten von Steuerelementen zueinander oder zum Verteilen von Kontrollen über einen bestimmten Bereich.

Markieren Sie eine der Schaltflächen, drücken die Taste Strg und markieren dann die restlichen. Sie werden bemerken, dass alle Schaltflächen bis auf eine weiß umrahmt sind, die letzte jedoch blau. Dieses ist die Kontrolle, nach der die anderen ausgerichtet werden, wenn Sie eine der Formatierungsfunktionen benutzen. Sie können bei gedrückt gehaltener Taste Strg auch nachträglich noch zwischen den bereits markierten Elementen hin und her wechseln, um das Ankerobjekt zu verändern.

Wählen Sie dann zum Beispiel das Menükommando *Format* > *Align* > *Tops*, um alle Buttons auf eine Höhe zu bringen. Spielen Sie ein wenig mit den Formatierungsmöglichkeiten herum und richten Sie die neuen Schaltfläche nach Ihren Wünschen auf dem Dialogfeld aus. In meinem Fall sieht das Ergebnis so aus:

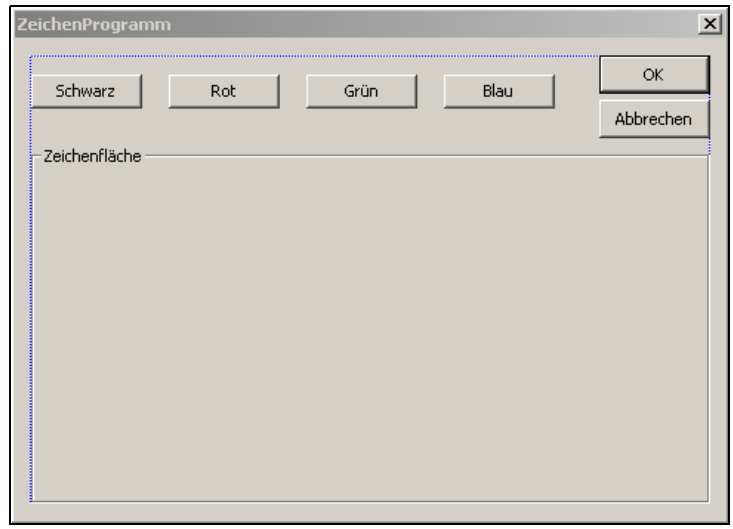


Abb. 4.54
Die Buttons wurden sinnvoll ausgerichtet

Behandlungsmethoden für die Buttons

Wenn Sie das Programm in seiner jetzigen Form austesten, werden die Buttons zwar dargestellt, doch bergen sie noch keine Funktionalität. Sie müssen erst wieder Behandlungsmethoden für die einzelnen Schaltflächen einfügen, dies geschieht analog zu denen des Dialogfelds.

Hinzufügen neuer Behandlungsmethoden

Gehen Sie zurück in den Ressourceneditor und wählen Sie dort eine der Schaltflächen aus. Wählen Sie im *Eigenschaften*-Fenster die Übersicht über die verfügbaren Fensternachrichten aus und fügen Sie eine Methode für *BN_CLICKED* auf die bekannte Weise hinzu.

Eine *BN_CLICKED*-Nachricht wird gerade dann erzeugt, wenn die zugehörige Schaltfläche angeklickt wird.

Fügen Sie nun entsprechende Behandlungsmethoden für die drei übrigen neuen Schaltflächen hinzu.

Übrigens, wenn Sie später einmal vom Ressourceneditor direkt an die Quelltextstelle springen wollen, die zur Behandlung einer Nachricht vorgesehen ist, wählen Sie das betreffende Fenster (also in diesem Fall den Dialog oder ein Steuerelement) aus und wählen in der Nachrichtenliste die betreffende aus.

Es erscheinen dann zwei Auswahlmöglichkeiten für diese Botschaft:

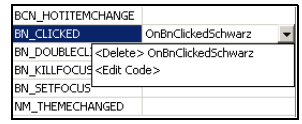


Abb. 4.55
Löschen und Editieren von Behandlungsmethoden

Wählen Sie hier *Löschen*, um die Behandlungsmethode zu löschen, oder *Editieren*, um zur zugehörigen Stelle im Quelltext zu springen.

In letzterem Fall werden Sie die automatisch generierten, leeren Funktionsrümpfe der vier neuen Methoden finden:

Automatisch angelegte Behandlungsmethoden

Listing 4.24
Die hinzugefügten Behandlungsmethoden

```
void CZeichenProgrammDlg::OnBnClickedSchwarz()
{
   // TODO: Add your control notification handler
// code here
}

void CZeichenProgrammDlg::OnBnClickedRot()
{
   // TODO: Add your control notification handler
// code here
}

void CZeichenProgrammDlg::OnBnClickedGruen()
{
   // TODO: Add your control notification handler
// code here
}

void CZeichenProgrammDlg::OnBnClickedBlau()
{
   // TODO: Add your control notification handler
// code here
}
```

Speichern der aktuellen Farbe

Die vom Benutzer ausgewählte Farbe soll in einer Membervariable festgehalten werden. Sie haben bereits gesehen, dass dazu eine Variable vom Typ *COLORREF* notwendig ist. Fügen Sie diese in die Headerdatei ein – je nach Geschmack manuell oder über den Assistenten zum Hinzufügen neuer Variablen – und bezeichnen Sie sie als *m_AktuelleFarbe*:

Listing 4.25
Neue Zeile in *Zeichen-ProgrammDlg.h*

```
COLORREF m_AktuelleFarbe;
```

Diese Variable sollte initialisiert werden, damit der Anwender beim Programmstart seine Linien mit einer definierten Farbe zeichnen kann. Wählen Sie hier schwarz und editieren Sie zu diesem Zweck den Konstruktor von *CZeichenProgrammDlg* in der Datei *ZeichenprogrammDlg.cpp*:

Listing 4.26
Der neue Konstruktor

```
CZeichenProgrammDlg::CZeichenProgrammDlg(CWnd* pParent
 /*=NULL*/)
   : CDialog(CZeichenProgrammDlg::IDD, pParent)
   , m_nStartPunktX(0)
   , m_nStartPunktY(0)
{
   m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
   m_bLinkeMaustasteGedrueckt = false;
   m_AktuelleFarbe = RGB(0,0,0);
}
```

Automatische Initialisierung von Variablen

Hier zeigt sich ein weiterer Unterschied zwischen normaler und automatischer Hinzufügung von Variablen: Durch Assistenten eingefügte Variablen werden automatisch mit einem Startwert initialisiert, in diesem Fall die von Ihnen eingebauten Integerwerte *m_nStartPunktX* und *m_nStartPunktY*, die jeweils mit 0 initialisiert wurden.

Farbige Punkte

Um nun auch farbige Punkte zeichnen zu können, muss in *OnLButtonDown* das Setzen von Pixeln überarbeitet werden:

Listing 4.27
Setzen von farbigen Punkten

```
void CZeichenProgrammDlg::OnLButtonDown(UINT nFlags, CPoint point)
{
   // prüfen, ob sich der Punkt auf der
// Zeichenflaeche befindet
   if (IstAufZeichenflaeche(point.x, point.y))
   {
      // ok, Gerätekontext anfordern
      CClientDC dc(this);

      // Pixel zeichnen – verwendet werden soll
// die aktuell
      // ausgewählte Farbe
      dc.SetPixel(point.x, point.y,
m_AktuelleFarbe);

      // merken, dass Maustaste gedrückt ist
      m_bLinkeMaustasteGedrueckt = true;

      // Maus festhalten
```

```
      SetCapture();

      // Startposition merken
      m_nStartPunktX = point.x;
      m_nStartPunktY = point.y;
   }

   // Standardbehandlung
   CDialog::OnLButtonDown(nFlags, point);
}
```

Andere Farben auswählen

Behandlungsmethoden zum Auswählen von Farben

Um diese Funktionalität schnell testen zu können, sollen nun die Funktionsrümpfe der *BN_CLICKED*-Behandlungsmethoden für die einzelnen Schaltflächen mit Leben gefüllt werden:

Listing 4.28 **Einstellen der Farbwerte**

```
void CZeichenProgrammDlg::OnBnClickedSchwarz()
{
   m_AktuelleFarbe = RGB (0, 0, 0);
}

void CZeichenProgrammDlg::OnBnClickedRot()
{
   m_AktuelleFarbe = RGB (255, 0, 0);
}

void CZeichenProgrammDlg::OnBnClickedGruen()
{
   m_AktuelleFarbe = RGB (0, 255, 0);
}

void CZeichenProgrammDlg::OnBnClickedBlau()
{
   m_AktuelleFarbe = RGB (0, 0, 255);
}
```

Sie sind jetzt in der Lage, farbige Startpixel zu setzen, doch die Linien sind immer noch schwarz. Editieren Sie daher nun die *OnMouseMove*-Methode, die ebenfalls in der Datei *ZeichenprogrammDlg.cpp* zu finden ist:

Listing 4.29 **Erzeugen und Auswählen von Stiften**

```
void CZeichenProgrammDlg::OnMouseMove(UINT nFlags,
CPoint point)
{
   // wird die Maus bewegt und ist die linke
// Maustaste gedrückt,
   // soll erneut ein Punkt gezeichnet werden
   if ((m_bLinkeMaustasteGedrueckt) &&
```

```
(IstAufZeichenflaeche(point.x, point.y)))
   {
      // Gerätekontext anfordern
      CClientDC dc(this);

      // Stift vorbereiten
      CPen StandardPen(PS_SOLID, 1,
m_AktuelleFarbe);
      CPen *pOldPen =
dc.SelectObject(&StandardPen);

      // Startpunkt anfahren
      dc.MoveTo(m_nStartPunktX, m_nStartPunktY);

      // Linie bis zum Endpunkt ziehen
      dc.LineTo(point);

      // alten Stift wiederherstellen
      dc.SelectObject(pOldPen);

      // neue Position speichern
      m_nStartPunktX = point.x;
      m_nStartPunktY = point.y;
   }

   CDialog::OnMouseMove(nFlags, point);
}
```

Vielleicht hatten Sie sich bereits darüber gewundert, dass beim Zeichnen von Linien keine Farbe explizit angegeben werden brauchte. Das liegt daran, dass ein Gerätekontext eben nicht nur eine Leinwand ist, sondern auch sämtliche Hilfsmittel zur Verfügung stellt, die ein Maler benötigen würde, wenn man das Konzept einmal in die Realität übersetzt.

Dazu gehören unter anderem auch Stifte und es ist ein Stift, mit dem Linien über die *LineTo*-Methode gezeichnet werden.

GDI-Objekte

Zeichenwerkzeuge

Ein solcher Stift ist eins von zahlreichen GDI-Objekten, die jeweils durch eine eigene Klassen im Rahmen der MFC repräsentiert werden. Die Klasse zur Arbeit mit Stiften trägt den passenden Namen *CPen*.

Die einzelnen Parameter, die für GDI-Objekte verwendet werden können, finden Sie im Anhang dieses Buchs aufgelistet, an dieser Stelle sei nur gesagt, dass der im obigen Listing stehende Aufruf einen Stift mit der aktuell eingestellten Farbe erzeugt, der eine Zeichendicke von einem Pixel hat und das Ziehen ununter-

terbrochener Linien erlaubt – sie könnten Ihren Stift auch so erzeugen, dass er automatisch gestrichelte oder gepunktete Linien zeichnet.

Dem Erzeugen eines Stifts folgt meistens die Verwendung in einem Gerätekontext. Dort kann immer nur ein GDI-Objekt eines bestimmten Typs (zum Beispiel ein Stift) zurzeit aktiv sein, daher liefert die überladene *SelectObject*-Methode einen Zeiger auf den bis dahin aktivierten Stift zurück.

Weitere GDI-Objekte sind beispielsweise Stempel, die verwendet werden können, um Flächen mit Mustern zu füllen. Auch diese würden mit der *SelectObject*-Methode dem Gerätekontext bekannt gemacht werden, weshalb diese Funktion im Rahmen der Gerätekontextanwendung eine große Rolle spielt.

Abräumen von GDI-Objekten

Es ist wichtig, dass Sie vor Verlassen der Methode den alten Stift wieder aktivieren, da ansonsten die lokal angelegten Stifte nicht ordnungsgemäß abgeräumt werden könnten. Sie verbleiben dann im Systemspeicher und werden über kurz oder lang die wertvollen Systemressourcen über Gebühr belasten.

So, nun haben Sie es endlich geschafft – es können farbige Bilder gezeichnet werden. Kompilieren und starten Sie das Programm:

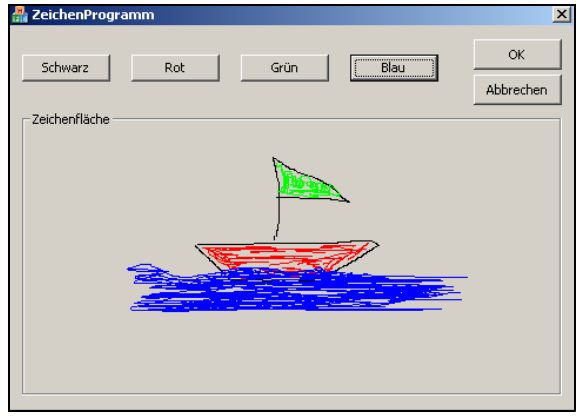


Abb. 4.56
Farbige Grafiken

Wiederherstellen des Bildinhalts

Ein Problem besteht immer noch: Zeichnen Sie ein paar Striche auf die Zeichenfläche und verdecken Sie das Dialogfeld mit einem anderen Fenster. Bringen Sie das Dialogfeld wieder zum Vorschein – die Striche sind verschwunden. Wieso?

Nun, Windows kann prinzipiell nicht wissen, was Ihre Anwendung in einem Fenster darstellen soll oder will – das ist Aufgabe der Applikation selbst. Im bisherigen Beispielprogramm werden jedoch nur direkt Daten in das Dialogfeld gezeichnet.

Zerstören von Bildinhalten

Wischt man mit einem Fenster – ähnlich einem Schwamm – über den Dialog, wird der überschriebene Hintergrund nicht wieder hergestellt. Sie ahnen es schon: Fenster werden nicht etwa irgendwo im Speicher gesichert, wenn Sie hin- und hergeschoben und möglicherweise überlagert werden, damit das Betriebssystem sie später wieder herstellen könnte.

Der Grund dafür ist auch recht schnell ermittelt: gehen Sie einmal von einer Bildschirmauflösung von 1.024*768 Pixeln bei einer Farbtiefe von 32 Bit aus. Sie kommen dann auf ungefähr 3 MByte, die ein maximiertes Fenster an reiner Grafikdarstellung benötigen würde. Nun sind unter Windows normalerweise eine ganze Reihe von Fenstern geöffnet, viele von Ihnen ebenfalls maximiert – der Speicherbedarf wäre ganz erheblich.

Aufforderung zum Neuzeichnen

Deshalb geht Windows den schon skizzierten Weg, den Anwendungen die Verantwortlichkeit zur Wiederherstellung ihres Fensterinhalts zu übertragen.

Das Betriebssystem sendet dann eine *WM_PAINT*-Nachricht an das betreffende Fenster – diese behandeln Sie im Zeichenprogramm in der Methode *OnPaint*, wenngleich derzeit auch noch ohne tatsächlichen Ausführungsinhalt.

Interessanterweise achtet Windows übrigens darauf, dass nur Bereiche neu gezeichnet werden, die tatsächlich wiederhergestellt werden müssen – Sie können also in *OnPaint* immer den kompletten Fensterinhalt wieder herstellen, tatsächlich gezeichnet werden nur benötigte Randgebiete, was eine nicht unerhebliche Geschwindigkeitssteigerung mit sich bringt.

Speichern von Linienzügen

Wie speichert man nun am sinnvollsten die gezeichneten Figuren? Ohne komplizierte Zusammenfassungsalgorithmen zu bemühen – beispielsweise könnte man in einigen Fällen zwei Linien zu einer zusammenfassen, wenn alle drei beteiligten Punkte (Startpunkt erste Linie, Endpunkt erste Linie, Endpunkt zweite Linie) auf einer Geraden liegen – muss tatsächlich jede einzelne Zeichenoperation irgendwie gesichert werden.

Dazu wollen wir uns einer der Containerklassen der MFC bedienen, die zu diesem Zweck ausgezeichnete Dienste leisten wird: *CList*.

MFC-Containerklassen

Die MFC bieten drei Arten von *Containerklassen* an, die jede für sich ihre eigenen Vorzüge bietet. Containerklassen sind, wenn man so will, eine Struktur zur Aufnahme einer unbestimmten Anzahl von Objekten eines definierten Typs.

Verkettete Listen

Sie haben sicherlich schon einmal mit verketteten Listen gearbeitet, eine ähnliche Funktionalität stellt die MFC-Containerklasse *CList* zur Verfügung. Sie kapselt die Möglichkeiten einer verketteten Liste und stellt Sie Ihnen zur bequemen Anwendung zur Verfügung.

Die allgemeine Syntax, um ein Objekt einer Containerklasse anzulegen, ist diese:

```
CCONTAINERTYP <TYP, PARAMETERTYP> VARIABLENNAME;
```

Dabei gibt *CCONTAINERTYP* den Namen der Containerklasse (zum Beispiel *CList*) an, *TYP* den Typ der in der Liste stehenden Daten (zum Beispiel eine Struktur) und *PARAMETERTYP* den Typ, der verwendet werden soll, wenn Elemente der Liste an Funktionen als Parameter übergeben werden. Dieses ist in der Regel der gleiche Typ wie bei *TYP*, eventuell als Referenz dargestellt.

Eine komplette Übersicht über die Möglichkeiten von Containerklassen im Rahmen der MFC würde an dieser Stelle den Rahmen sprengen, Sie sind jedoch im Anhang dieses Buchs aufgeführt.

Navigieren in Listen

Stellen Sie sich ein Objekt des Typs *CList* einfach als eine anfangs leere Liste von Funktionstypen vor. Sie können über diese Liste iterieren, indem Sie eine weitere Variable vom Typ *POSITION* einführen, die jeweils auf ein konkretes Element innerhalb dieser Liste zeigt.

Beim Auslesen eines Objekts aus der Liste wird die *POSITIONs*-Variable auf das jeweils nächste Objekt gesetzt, können keine Werte mehr ausgelesen werden, ist dieses auch direkt über diesen Navigationswert auszulesen.

Aber welche Daten sollen in die Liste eingetragen werden?

Da wir für einen Linienzug zwei Koordinaten speichern müssen, bietet sich eine Verwendung der ebenfalls in den MFC vorhandenen Klasse *CRect* an, die eine Rechteckstruktur analog zu *RECT* repräsentiert.

Es sollen die Startkoordinaten in den Komponenten *left* und *top* und die Endkoordinaten in den Komponenten *right* und *bottom* gespeichert werden. Ferner muss noch die Farbe festgehalten werden, in der eine Linie gezeichnet wurde.

Einbinden der benötigten Variablen und Funktionen

Eine entsprechende Struktur und die weiteren benötigten Variablen sowie einige vereinfachenden Zugriffsfunktionen finden Sie in der folgenden Headerdatei – sie sind manuell eingefügt worden:

Listing 4.30
Die neue Headerdatei für den Hauptdialog

```
// ZeichenProgrammDlg.h : Headerdatei
//

#pragma once
```

```
#include <afxtempl.h>

struct SLinienzug
{
   CRect m_Koordinaten;
   COLORREF m_Farbe;
};

// CZeichenProgrammDlg Dialogfeld
class CZeichenProgrammDlg : public CDialog
{
// Konstruktion
public:
   CZeichenProgrammDlg(CWnd* pParent = NULL);
// Standardkonstruktor

// Dialogfelddaten
   enum { IDD = IDD_ZEICHENPROGRAMM_DIALOG };

   protected:
   virtual void DoDataExchange(CDataExchange* pDX);// DDX/DDV-
Unterstützung


// Implementierung
protected:
   HICON m_hIcon;
   bool  m_bLinkeMaustasteGedrueckt;

   // Generierte Funktionen für die
// Meldungstabellen
   virtual BOOL OnInitDialog();
   afx_msg void OnSysCommand(UINT nID, LPARAM
lParam);
   afx_msg void OnPaint();
   afx_msg HCURSOR OnQueryDragIcon();
   DECLARE_MESSAGE_MAP()
public:
   afx_msg void OnLButtonDown(UINT nFlags, CPoint
 point);
   afx_msg void OnMouseMove(UINT nFlags, CPoint
point);
   afx_msg void OnLButtonUp(UINT nFlags, CPoint point);

   // Prueft, ob sich ein gegebener Punkt in
// Clientkoordinaten innerhalb der
// Zeichenflaeche befindet
   bool IstAufZeichenflaeche(int in_nX, int in_nY);
```

```
   // Setzt den Iterator auf den Listenanfang
// zurück
   void ResetListe();

   // Liest den nächsten Linienzug aus
   bool NaechstenLinienzugAuslesen(SLinienzug
&out_Linienzug);

   // Fügt einen Linienzug zur Liste hinzu
   void LinienzugHinzufuegen(SLinienzug
&in_Linienzug);

   // Loescht saemtliche Linien aus der Liste
   void ListeLoeschen();

protected:
   int m_nStartPunktX;
   int m_nStartPunktY;

   COLORREF m_AktuelleFarbe;

   CList<SLinienzug, SLinienzug&> m_LinienListe;
   POSITION m_PositionInListe;

public:
   afx_msg void OnBnClickedSchwarz();
   afx_msg void OnBnClickedRot();
   afx_msg void OnBnClickedGruen();
   afx_msg void OnBnClickedBlau();
};
```

Implementationen der Zugriffsfunktionen

Die Implementationen der Zugriffsfunktionen gehören in die Datei *ZeichenprogrammDlg.cpp*. Sie werden im Folgenden der Reihe nach erläutert:

Listing 4.31
Zurücksetzen der Liste

```
// Setzt den Iterator auf den Listenanfang zurück
void CZeichenProgrammDlg::ResetListe()
{
   m_PositionInListe =
m_LinienListe.GetHeadPosition();
}
```

Die Methode *ResetListe* setzt die Liste auf den Anfang zurück, genauer gesagt die Positionsvariable. *GetHeadPosition* ermittelt gerade die Position des ersten Elements einer *CList*. Der nächste Auslesevorgang liefert dann eben dieses Element der Liste zurück, soweit die Liste nicht leer ist.

Auslesen von Elementen

Das Auslesen wird über die Methode *NaechstenLinienzugAuslesen* realisiert:

Listing 4.32 **Auslesen von Listenelementen**

```
// Liest den nächsten Linienzug aus
bool
CZeichenProgrammDlg::NaechstenLinienzugAuslesen(
SLinienzug &out_Linienzug)
{
   // existieren noch weitere Elemente in der
// Liste?
   if (m_PositionInListe)
   {
      // objekt kopieren
      out_Linienzug =
 m_LinienListe.GetNext(m_PositionInListe);

      // alles OK
      return (true);
   }
   else
   {
      // es existieren keine weiteren Elemente
      return (false);
   }
}
```

Das Auslesen eines Elements beginnt mit einer Prüfung, ob das *POSITIONs*-Objekt gültig ist. Hat es den Wert *NULL*, befindet es sich zur Zeit auf keiner gültigen Listenelementposition, was in der Zeichenprogrammanwendung ein Indiz dafür ist, dass keine weiteren Elemente ausgelesen werden können. Dieses ist auch der Fall, wenn die Liste leer ist.

Existiert noch ein Element, wird dieses über die *CList*-Methode *GetNext* ausgelesen, was automatisch das *POSITIONs*-Objekt ein Element weiter bewegt und auf NULL setzt, falls das Ende der Liste erreicht ist.

Ein Rückgabewert von *true* bedeutet, dass ein Wert ausgelesen werden konnte, *false*, dass keine weiteren Elemente in der Liste existieren.

Hinzufügen von Elementen in die Liste

Um Elemente auslesen zu können, müssen erst einmal welche hinzugefügt werden können. Dieses erledigt die Methode *LinienzugHinzufuegen*:

Listing 4.33 **Hinzufügen von Elementen**

```
// Fügt einen Linienzug zur Liste hinzu
void
CZeichenProgrammDlg::LinienzugHinzufuegen(SLinienzug
&in_Linienzug)
```

```
{
   // übergebenes Objekt kopieren und in Liste
// einfügen
   m_LinienListe.AddTail(in_Linienzug);
}
```

Die *CList*-Methode *AddTail* fügt ein übergebenes Element vom Datentyp der Liste, der über die Initialisierung des *CList*-Objekts festgelegt wurde, an das Ende der Liste ein. Es gibt auch weitere Methoden zum Einfügen von Objekten wie zum Beispiel *AddHead*, das Objekte an den Anfang der Liste einfügt.

Beachten der Reihenfolge beim Zeichnen

Die Zeichenoperationen müssen aber in der Reihenfolge nachvollzogen werden, wie sie der Benutzer ursprünglich auch vollzogen hatte, dazu ist es sinnvoll, die Liste sequenziell abzuarbeiten, weshalb ein Einfügen an das Ende erforderlich ist.

Löschen der Liste

Um mit einer komplett neuen Zeichenoberfläche starten zu können, sollte es möglich sein, die Liste zu löschen. Sie werden gleich noch eine weitere Schaltfläche in das Programm integrieren, die für genau diese Operation zur Verfügung stehen wird:

Listing 4.34
Löschen einer Liste

```
// Loescht saemtliche Linien aus der Liste
void CZeichenProgrammDlg::ListeLoeschen()
{
   // alle Elemente löschen
   m_LinienListe.RemoveAll();
}
```

ListeLoeschen entfernt alle Elemente unter Verwendung der *RemoveAll CList*-Methode aus dem *CList*-Objekt.

Speichern von Punkten

Die Daten, die gespeichert werden sollen, sind an zwei Stellen abzufangen. Zum einen bei *OnLButtonDown*, zum anderen bei *OnMouseMove*.

Markieren von Punkten innerhalb der Liste

Da wir die Positionen der Linienzüge in einer Rechteckstruktur speichern wollen, *OnLButtonDown* aber nur einzelne Pixel setzt, verwenden wir den einfachen Trick, die gleichen Koordinaten in den *left/top* und den *right/bottom* Komponenten der Rechteckstruktur abzulegen.

Beim Wiederherstellen des Bilds kann diese Gleichheit dann festgestellt und gegebenenfalls statt einer Linie ein Punkt gezeichnet werden:

Listing 4.35 **Speichern von Punkten**

```
void CZeichenProgrammDlg::OnLButtonDown(UINT nFlags,
CPoint point)
{
   // prüfen, ob sich der Punkt auf der
// Zeichenflaeche befindet
   if (IstAufZeichenflaeche(point.x, point.y))
   {
      // ok, Gerätekontext anfordern
      CClientDC dc(this);

      // Pixel zeichnen – verwendet werden soll
// die aktuell
      // ausgewählte Farbe
      dc.SetPixel(point.x, point.y,
m_AktuelleFarbe);

      // merken, dass Maustaste gedrückt ist
      m_bLinkeMaustasteGedrueckt = true;

      // Maus festhalten
      SetCapture();

      // Startposition merken
      m_nStartPunktX = point.x;
      m_nStartPunktY = point.y;

      // Position dieses Punktes in der
// Linienliste speichern
      SLinienzug Linienzug;

      Linienzug.m_Koordinaten.left = point.x;
      Linienzug.m_Koordinaten.right = point.x;
      Linienzug.m_Koordinaten.top = point.y;
      Linienzug.m_Koordinaten.bottom = point.y;
      Linienzug.m_Farbe = m_AktuelleFarbe;

      LinienzugHinzufuegen(Linienzug);
   }

   // Standardbehandlung
   CDialog::OnLButtonDown(nFlags, point);
}
```

Wie Sie sehen, wird eine neue *Slinienzug*-Strukturvariable deklariert und dann mit den gerade aktuellen Werten gefüllt und zur Liste hinzugefügt. Die *left/top*- und *right/bottom*-Komponentenpaare erhalten jeweils dieselben Werte.

Speichern von Linien

Die tatsächlichen Linien werden in *OnMouseMove* erzeugt, müssen also auch an dieser Stelle gesichert werden:

Editieren von *OnMouseMove* zum Sichern der Linien

Listing 4.36 **Speichern der Linienzüge**

```
void CZeichenProgrammDlg::OnMouseMove(UINT nFlags,
CPoint point)
{
   // wird die Maus bewegt und ist die linke
// Maustaste gedrückt,
   // soll erneut ein Punkt gezeichnet werden
   if ((m_bLinkeMaustasteGedrueckt) &&
(IstAufZeichenflaeche(point.x, point.y)))
   {
      // Gerätekontext anfordern
      CClientDC dc(this);

      // Stift vorbereiten
      CPen StandardPen(PS_SOLID, 1,
m_AktuelleFarbe);
      CPen *pOldPen =
 dc.SelectObject(&StandardPen);

      // Startpunkt anfahren
      dc.MoveTo(m_nStartPunktX, m_nStartPunktY);

      // Linie bis zum Endpunkt ziehen
      dc.LineTo(point);

      // alten Stift wiederherstellen
      dc.SelectObject(pOldPen);

      // Linienzug in Liste einfügen
      SLinienzug Linienzug;
      Linienzug.m_Koordinaten.left =
m_nStartPunktX;
      Linienzug.m_Koordinaten.top =
m_nStartPunktY;
      Linienzug.m_Koordinaten.right = point.x;
      Linienzug.m_Koordinaten.bottom = point.y;
      Linienzug.m_Farbe = m_AktuelleFarbe;

      LinienzugHinzufuegen(Linienzug);

      // neue Position speichern
      m_nStartPunktX = point.x;
      m_nStartPunktY = point.y;
   }
```

```
   CDialog::OnMouseMove(nFlags, point);
}
```

Hier gibt es keine weiteren Überraschungen, die Vorgehensweise ist analog zu der bei den gespeicherten Punkten.

Wiederherstellen von Fensterinhalten

Nachdem nun die Werte ordnungsgemäß gespeichert werden, sollen Sie beim Wunsch nach einer Fensteraktualisierung auch wiedergegeben werden.

Zeichnen der gespeicherten Linien und Punkte

Begeben Sie sich im Quelltext dazu zur Methode *OnPaint* (Datei *ZeichenprogrammDlg.cpp*) und editieren Sie diese wie folgt:

***Listing 4.37* Neuzeichnen des Fensters**

```
void CZeichenProgrammDlg::OnPaint()
{
   if (IsIconic())
   {
      CPaintDC dc(this); // Gerätekontext zum
// Zeichnen

      SendMessage(WM_ICONERASEBKGND,
 reinterpret_cast<WPARAM>(dc.GetSafeHdc()),
0);

      // Symbol in Clientrechteck zentrieren
      int cxIcon = GetSystemMetrics(SM_CXICON);
      int cyIcon = GetSystemMetrics(SM_CYICON);
      CRect rect;
      GetClientRect(&rect);
      int x = (rect.Width() - cxIcon + 1) / 2;
      int y = (rect.Height() - cyIcon + 1) / 2;

      // Symbol zeichnen
      dc.DrawIcon(x, y, m_hIcon);
   }
   else
   {
      // Linienzugvariable vorbereiten
      SLinienzug Linienzug;

      // Liste zurücksetzen
      ResetListe();

      // Gerätekontext vorbereiten
      CPaintDC dc(this);
```

```
      // Stifte vorbereiten
      CPen SchwarzerStift(PS_SOLID, 1,
RGB(0, 0, 0));
      CPen RoterStift(PS_SOLID, 1,
RGB(255, 0, 0));
      CPen GruenerStift(PS_SOLID, 1,
RGB(0, 255, 0));
      CPen BlauerStift(PS_SOLID, 1,
RGB(0, 0, 255));

      // alten Stift speichern
      CPen *pAlterStift =
dc.SelectObject(&SchwarzerStift);

      // Jeden Linienzug bearbeiten
      while
(NaechstenLinienzugAuslesen(Linienzug))
      {
         // handelt es sich um einen Punkt
// oder eine Linie?
         if ((Linienzug.m_Koordinaten.left
== Linienzug.m_Koordinaten.right)
&& (Linienzug.m_Koordinaten.top == Linienzug.m_Koordinaten.bottom))
         {
            // einfacher Punkt

dc.SetPixel(
Linienzug.m_Koordinaten.left,
Linienzug.m_Koordinaten.top,
Linienzug.m_Farbe);
         }
         else
         {
            // passenden Stift setzen
            if (Linienzug.m_Farbe ==
RGB(0, 0, 0))
            {
            dc.SelectObject(
&SchwarzerStift);
            }
            else
            {
if (Linienzug.m_Farbe
== RGB(255, 0, 0))
               {
               dc.SelectObject(
&RoterStift);
```

```
                }
                else
                {
                   if
(Linienzug.m_Farbe == RGB(0, 255, 0))
                   {
                   dc.SelectObject(
&GruenerStift);
                   }
                   else
                   {
                   dc.SelectObject(
&BlauerStift);
                   }
                }
             }

             // Linie. Startpunkt anfahren
             dc.MoveTo(
Linienzug.m_Koordinaten.left, Linienzug.m_Koordinaten.top);

             // Linie zeichnen
             dc.LineTo(
Linienzug.m_Koordinaten.right,
Linienzug.m_Koordinaten.bottom);
          }
      }

      // ursprünglichen Stift zurücksetzen
      dc.SelectObject(pAlterStift);

      // Standardbehandlung aufrufen
      CDialog::OnPaint();
   }
}
```

Die Methode ist nun schon etwas umfangreicher geworden, aber eigentlich nicht allzu kompliziert zu verstehen.

Zunächst wird die LinienListe zurückgesetzt, um die *POSITIONs*-Variable mit einem gültigen Startwert zu versehen. Daraufhin legt das Programm eine Reihe von Stiften an, die jeweils den Farben entsprechen, welche die Linienzüge besitzen können.

Verwenden von unterschiedlichen Stiften

In diesem Fall liegen festdefinierte Farben vor, also ist dieser Weg möglich, wären die Farben frei definierbar gewesen, müsste man den Stift tatsächlich vor jedem Zeichenvorgang neu mit den passenden Werten erzeugen – auf diese vereinfachte Weise sparen wir also ein wenig Abarbeitungszeit.

Nachdem der ursprüngliche Stift des Gerätekontexts gespeichert wurde – Sie erinnern sich, dieser muss am Ende der Methode zurückgesetzt werden – werden der Reihe nach alle in der Liste befindlichen Elemente ausgelesen.

Je nachdem, ob die *left/top*- und *right/bottom*-Komponenten denselben Wert haben, wird ein Punkt oder eine Linie in der gespeicherten Farbe beziehungsweise mit dem vorbereiteten Stift gezeichnet.

Testen Sie nun die neue Funktionalität am laufenden Programm aus.

Fensterinhalt wieder löschen

Zu guter Letzt soll noch eine Schaltfläche zum Löschen des Fensterinhalts in das Zeichenprogramm eingearbeitet werden. Öffnen Sie dazu den Ressourceneditor und das Hauptdialogfeld und platzieren einen neuen Button, den Sie mit *Loeschen* beschriften und die ID *IDC_LOESCHEN* geben.

Editieren Sie die entstehenden Memberfunktion wie folgt:

Listing 4.38 **Implementation zum Löschen der Zeichenfläche**

```
// Zeichenfläche löschen
void CZeichenProgrammDlg::OnBnClickedLoeschen()
{
   ListeLoeschen();
   InvalidateRect(NULL, true);
}
```

Der Aufruf von *ListeLoeschen* entfernt sämtliche Elemente aus dem Listenobjekt. Nun muss allerdings noch das Fenster zum Neuzeichnen aufgefordert werden, was ja bekannterweise durch eine *WM_PAINT*-Nachricht initiiert wird. Dieses kann durch Verwendung der Funktion *InvalidateRect* erzwungen werden, die den Zeiger auf eine *RECT*-Struktur als Parameter erhält, die wiederum das Rechteck des Client-Bereichs angibt, das neu zu zeichnen ist. Geben Sie hier *NULL* an, wird das komplette Dialogfeld neu gezeichnet.

Löschen des Client-Bereichs

Der zweite Parameter bestimmt, ob der Client-Bereich des Dialogfelds vor dem neuen Zeichenvorgang gelöscht werden soll. *True* sorgt für ein vorhergehendes Löschen, *false* lässt den Inhalt unangetastet. In unserem Fall ist *true* die richtige Alternative, da sonst die schon bestehenden Linien trotz des Löschens der Daten bestehen bleiben würden.

Das fertige Programm

Sie haben im Rahmen dieses Kapitels ein zwar schlichtes, aber funktionales Zeichenprogramm entwickelt, das neben dem Zeichnen von Linienzügen auch das Speichern der eingegebenen Daten, das Wiederherstellen des Fensters nach einer *WM_PAINT*-Nachricht, das Löschen der Zeichenfläche und das Auswählen verschiedenere Farbwerte implementiert.

Weiterhin haben Sie gelernt, Steuerelementwerte direkt aus dem Programm heraus auszulesen (*GetDlgItem*), neue Behandlungsmethoden für Dialoge und Kontrollen hinzuzufügen und die Maustasten sowie -bewegungen abzufragen sowie mit Gerätekontexten und GDI-Objekten zu arbeiten.

Weitere Features für das Zeichenprogramm

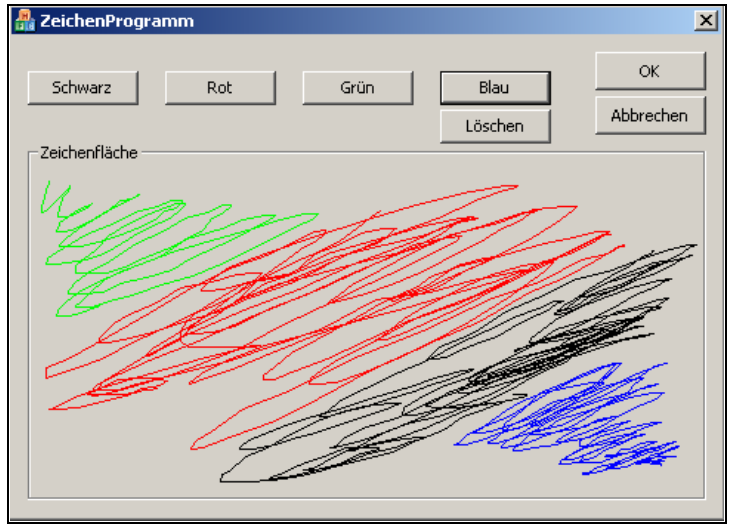
Sie sollten sich jetzt die Zeit nehmen, das Zeichenprogramm nach eigenen Vorstellungen zu erweitern. Zum Beispiel könnten Sie eine Funktion zum Malen von Rechtecken oder Kreisen einbauen, oder eine größere Menge an Farben zur Verfügung stellen.

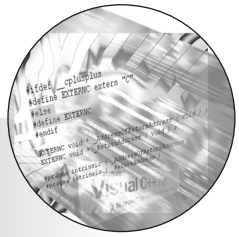
Was immer Sie tun, es ist auf jeden Fall sinnvoll genutzte Zeit, denn die Grundkonzepte dieses Kapitels ermöglichen es Ihnen, nahezu jede beliebige Anwendung zu programmieren.

In den weiteren Abschnitten des vorliegenden Buchs geht es dann um weitere Feinheiten und insbesondere die Verwendung von nicht auf Dialogfelder basierenden Fenstern.

Zum Abschluss noch ein Screenshot vom fertigen lauffähigen Programm:

Abb. 4.57
Das fertige Programm





Entwickeln von SDI-Anwendungen

5

Unterschiede zu dialogfeldbasierenden Applikationen

Im vorangegangenen Kapitel wurde eine Anwendung auf der Basis eines Dialogfelds realisiert, die mit einer geringen Anzahl an Dateien und Klassen auskam.

Während viele kleinere Tools durchaus nach diesem Konzept entwickelt werden können, sind doch häufig umfangreichere Projektentwürfe notwendig, um eine gestellte Aufgabe durch ein MFC-Programm realisieren zu können.

Umfangreichere Projekte

Dieses liegt zum einen darin begründet, dass umfangreiche Daten gespeichert werden müssen – beispielsweise in einer Datenbank-ähnlichen Anwendung – oder die Darstellung der Dokumentdaten nicht auf trivialem Wege durchführbar ist. Ein Beispiel für den letztgenannten Fall ist ein Programm, dass die eingegebenen Daten auf mehrere verschiedene Arten anzeigen kann, vielleicht einmal in Textform und einmal als Balkendiagramm.

Für solche Anwendungsgebiete müssen andere Strukturen herhalten, die im Rahmen der Windows-Programmierung als SDI- und MDI-Programme (Single bzw. Multiple Document Interface) bezeichnet werden.

Entwicklung einer SDI-Applikation

Der vorliegende Abschnitt beschäftigt sich mit der SDI-Variante und wird eine vollständige lauffähige Anwendung hervorbringen, die neben den Möglichkeiten zum Speichern von Dokumentdaten auch weitere wichtige Windows-Themen wie das Erzeugen von Menüs oder die Ausgabe von Grafik besprechen wird.

Zunächst ist es wichtig zu erfahren, wie Daten in einer MFC-SDI-Applikation gespeichert und dann (theoretisch besprochen) dargestellt werden.

Die Document/View-Architecture

Sie haben bereits das MFC-Anwendungsgerüst kennen gelernt, dessen Kern die *Dokument-/Ansichtarchitektur* ist. Hierbei handelt es sich um ein Konzept, das durch seine Einfachheit und Vorteile überzeugen kann.

Stellen Sie sich eine umfangreiche Anwendung vor, die ihre Daten auf eine beliebige Weise speichert und die Ausgabe dieser Informationen durch direkten Zugriff vornimmt.

Datenorganisation

Nun entsteht noch in der Entwicklungsphase die Notwendigkeit, die Daten in anderer Art und Weise zu organisieren, beispielsweise weil wichtige Schlüsselinformationen weiter aufgeteilt oder zusammengefasst werden sollen.

Das Ergebnis ist, dass neben den Methoden zur eigentlichen Datenverwaltung auch die zur Anzeige neu geschrieben werden müssen, da sich durch die eingangs gewählte enge Verzahnung zwischen Darstellung und Informationsba-

sis ein roter Faden durch das komplette Projekt zieht und die Umstellung Folgefehler in allen möglichen Unterkomponenten hervorruft.

Ein deutlich geeigneterer Weg ist es, die zu einem Projekt gehörenden Daten in einem eigenen Bereich abzulegen und dann nur über Anforderungsfunktionen auf diese zu zugreifen.

Separates Speichern von Daten

Dadurch, dass die Daten separat gespeichert werden – im Rahmen der MFC- und der Dokument-/Ansichtarchitektur spricht man hier von der Dokumentenklasse – ist es nun möglich, weitere unabhängige Ansichtsobjekte zu erzeugen – diese werden durch Ansichtsklassen repräsentiert –, die ihrerseits über geeignete Zugriffsmethoden an die benötigten Elemente herankommen, aber keinen direkten Zugriff auf die zugrunde liegenden Daten haben.

Blackbox-Prinzip bei Dokumentdaten

Man sagt, die Ansichtsfunktionen benötigen keine Kenntnis über den tatsächlichen Aufbau der Informationen. Ein Geburtsdatum könnte intern als Ganzzahl oder als String gespeichert werden, das Ansichtsobjekt fordert vom Dokument aber immer einen bestimmten Typ an, beispielsweise eine Stringrepräsentation.

Die Dokumentenklasse nimmt nun das tatsächliche vorliegende Format und konvertiert es geeignet in das angeforderte Format.

Entscheidet sich der Entwickler später, die interne Repräsentation zu verändern, braucht er nur die Anforderungsfunktionen geeignet, um zu schreiben, die Ansichtsklasse selbst kann in unveränderter Form ihren Dienst weiterversehen.

Ansichtsklassen als Puffer zwischen Benutzer und Daten

Einen weiteren wichtigen Zweck erfüllen die Ansichtsklassen, wenn es um die Darstellung von Informationen und die Manipulation derselben durch den Benutzer geht.

Umleitung der Benutzereingaben

In vielen Fällen ist es nicht wünschenswert, dass ein Benutzer Daten direkt verändern kann – vielleicht ist die interne Repräsentation mit den Eingaben eines Anwenders auch gar nicht kompatibel, wie es im obigen Beispiel mit dem Geburtsdatum gezeigt wurde (dieses könnte als Ganzzahl gespeichert sein, während der Benutzer nur Strings eingeben kann).

Die Ansichtsklasse nimmt daher zunächst nur die Eingaben des Anwenders entgegen und überträgt diese dann in geeigneter Form an die Dokumentenklasse. Auch hier greift wieder das Prinzip von Zugriffsfunktionen, die eine Abstraktion von den tatsächlichen Daten darstellen und einen transparenten bidirektionalen Zugriff innerhalb der Projekte erlauben.

Mehrere Ansichtsklassen für mehrere Darstellungen

Multiple Ansichtsmöglichkeiten

Ein weiterer unmittelbarer Vorteil – neben der Wartbarkeit – ist die Möglichkeit, ein und dieselbe Datenmenge auf theoretisch unbegrenzt viele Arten darzustellen.

Es ist ausreichend, weitere Ansichtsklassen zu erzeugen und diese mit dem Dokument zu verknüpfen, um daraus eine Vielzahl an Informationsaufbereitungen zu erzeugen, wie es die folgende Übersicht veranschaulicht.

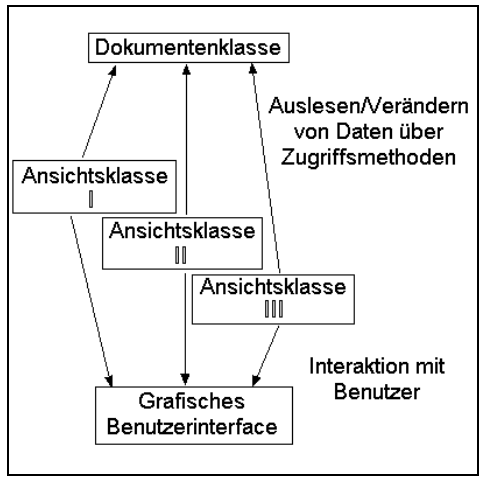


Abb. 5.1 **Zusammenhang zwischen Dokumentenklassen, Ansichtsklassen und Benutzerschnittstelle**

Das vorliegende Kapitel beschäftigt sich, wie bereits angekündigt, mit einer einfachen SDI-Anwendung, die neben einer Dokumentenklasse genau eine Ansichtsklasse enthält.

Das Apfelmännchen-Projekt

Um das Prinzip der SDI-Anwendungen anschaulich zu erklären, sollen auch gleich die Grundlagen der Grafikausgabe unter Windows erklärt werden.

Darstellung grafischer Elemente

Viele neue Programmierer machen sich keine oder nur wenige Vorstellungen davon, wie aufwendig die Darstellung von grafischen Elementen wirklich ist. Es bietet sich daher gleich von Beginn an, die Grenzen der Möglichkeiten aufzuzeigen, um so von vornherein die Entwicklung von anspruchsvollen Anwendungen in die richtige Bahn zu lenken.

Inhalt des neuen Projekts

Um die Theorie nicht allzu trocken werden zu lassen, soll der geneigte Leser trotzdem durch eine ansprechende Optik belohnt werden – konkret geht es in diesem Beispiel daher um die Berechnung und Darstellung von so genannten *Apfelmännchen* (auch *Mandelbrot-Mengen* genannt), deren mathematischer Hintergrund ebenfalls grundlegend erklärt werden soll.

Am Ende der Bemühungen wird dann eine Art Betrachtungsprogramm stehen, dass es erlaubt, einen beliebigen Ausschnitt der Apfelmännchengrafik zu betrachten, zu drucken und zu speichern, sowie durch eine *Gummibandfunktion* bestimmte Bereiche auszuwählen und zu vergrößern. Das Ganze wird abgerundet durch den Einsatz von Menüs, Toolbars und Tooltips.

Erzeugen des Projekts

Anlegen des neuen Projekts

Legen Sie ein neues Projekt über den Menüpunkt *Datei* > *Neu* > *Projekt* an, wählen Sie ein Visual C++-MFC-Projekt mit dem Namen *Apfelmaennchen* aus und stellen Sie als Applikationstyp *Single Document* ein.

Sie finden das fertige Projekt auch auf der Buch-CD im Verzeichnis *\Kapitel5\Apfelmaennchen*.

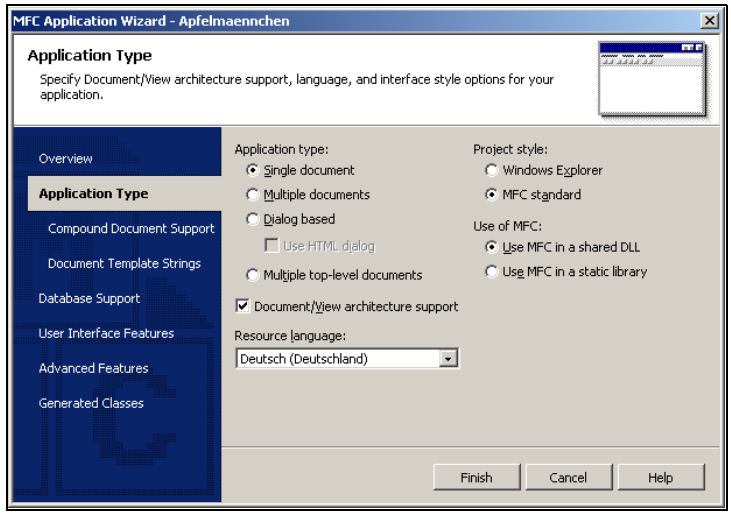


Abb. 5.2
Nur eine Einstellung muss verändert werden

Die restlichen Einstellungen bleiben unverändert.

Erzeugte Dateien

Begutachtung der entstandenen Dateien

Neben den schon aus dem vorhergegangenen Projekt bekannten Dateien werden bei einer SDI-Anwendung sechs Files erzeugt, die im Rahmen dieses Kapitels besondere Beachtung verdienen:

- *MainFrm.h* und *MainFrm.cpp*: die Header- und Implementationsdatei für das Hauptfenster der Applikation. Dieses enthält neben einem Client-Bereich, der durch die gleich vorgestellte Ansichtsklasse ausgefüllt wird, Menüs und Toolbars sowie eine Statusleiste.

- *ApfelmaennchenDoc.h* und *ApfelmaennchenDoc.cpp*: die Dokumentenklasse der Anwendung.
- *ApfelmaennchenView.h* und *ApfelmaennchenView.cpp*: die (einzige) Ansichtsklasse der Anwendung.

Es ist wichtig, einen Überblick über die Inhalte dieser Klassen zu erhalten, weshalb sie auf den folgenden Seiten kurz dargestellt werden sollen.

Die *CMainFrame*-Klasse

Die *CMainFrame*-Klasse symbolisiert das Hauptrahmenfenster der Applikation:

Listing 5.1 MainFrm.h

```
class CMainFrame : public CFrameWnd
{

protected: // Nur aus Serialisierung erstellen
   CMainFrame();
   DECLARE_DYNCREATE(CMainFrame)

// Attribute
public:

// Operationen
public:

// Überschreibungen
public:
   virtual BOOL PreCreateWindow(CREATESTRUCT& cs);

// Implementierung
public:
   virtual ~CMainFrame();
#ifdef _DEBUG
   virtual void AssertValid() const;
   virtual void Dump(CDumpContext& dc) const;
#endif

protected:  // Eingebundene Elemente der Steuerleiste
   CStatusBar  m_wndStatusBar;
   CToolBar    m_wndToolBar;

// Generierte Funktionen für die Meldungstabellen
protected:
   afx_msg int OnCreate(LPCREATESTRUCT
lpCreateStruct);
   DECLARE_MESSAGE_MAP()
};
```

Makros und Methoden von *CMainFrame*

Makros in *CMainFrame*

Sie werden zunächst über das Makro *DECLARE_DYNCREATE* stolpern, das allerdings im alltäglichen Umgang für den Programmierer nicht weiter von Bedeutung ist. Kurz gesagt gestattet es dem Anwendungsgerüst, zur Laufzeit dynamische Objekte des spezifizierten Klassentyps anzulegen – die genauen Voraussetzungen und Notwendigkeiten sind allerdings tatsächlich für das weitere Studium dieses Buches nicht relevant.

Die überschriebene Methode *PreCreateWindow* wird aufgerufen, kurz bevor das Hauptfenster erzeugt wird, *OnCreate* dient als Reaktion auf eine eingehende *WM_CREATE*-Nachricht.

Debug-Funktionen

Die Funktionen *AssertValid* und *Dump* dienen zum Debugging, also der Fehlersuche, innerhalb Ihrer Programme. Sie werden im Kapitel zum Debugger noch gesondert behandelt und detailliert besprochen.

Zu guter Letzt finden Sie zwei Membervariablen namens *m_wndStatusBar* und *m_wndToolBar*, die jeweils die Statuszeile beziehungsweise die Werkzeugleiste des Hauptfensters beschreiben.

MainFrm.cpp

Die vom Anwendungsassistenten erzeugte Implementationsdatei sieht so aus:

Listing 5.2 MainFrm.cpp

```
// MainFrm.cpp : Implementierung der Klasse CMainFrame
//

#include "stdafx.h"
#include "Apfelmaennchen.h"

#include "MainFrm.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif


// CMainFrame

IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
   ON_WM_CREATE()
END_MESSAGE_MAP()

static UINT indicators[] =
{
   ID_SEPARATOR,           // Statusleistenanzeige
```

```
   ID_INDICATOR_CAPS,
   ID_INDICATOR_NUM,
   ID_INDICATOR_SCRL,
};


// CMainFrame Erstellung/Zerstörung

CMainFrame::CMainFrame()
{
   // TODO: Hier Code für die Memberinitialisierung
// einfügen
}

CMainFrame::~CMainFrame()
{
}

BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
   if( !CFrameWnd::PreCreateWindow(cs) )
      return FALSE;
   // TODO: Ändern Sie hier die Fensterklasse oder
// die Darstellung, indem Sie
   //  CREATESTRUCT cs modifizieren.

   return TRUE;
}

int CMainFrame::OnCreate(LPCREATESTRUCT
lpCreateStruct)
{
   if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
      return -1;

   if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT,
WS_CHILD | WS_VISIBLE | CBRS_TOP
      | CBRS_GRIPPER | CBRS_TOOLTIPS |
CBRS_FLYBY | CBRS_SIZE_DYNAMIC) ||
      !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
   {
      TRACE0("Symbolleiste konnte nicht erstellt
werden\n");
      return -1;      // Fehler bei Erstellung
   }

   if (!m_wndStatusBar.Create(this) ||
      !m_wndStatusBar.SetIndicators(indicators,
```

```
        sizeof(indicators)/sizeof(UINT)))
   {
      TRACE0("Statusleiste konnte nicht erstellt
werden\n");
      return -1;      // Fehler bei Erstellung
   }
   // TODO: Löschen Sie diese drei Zeilen, wenn Sie
// nicht möchten, dass die Systemleiste
// andockbar ist
   m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
   EnableDocking(CBRS_ALIGN_ANY);
   DockControlBar(&m_wndToolBar);

   return 0;
}


// CMainFrame Diagnose

#ifdef _DEBUG
void CMainFrame::AssertValid() const
{
   CFrameWnd::AssertValid();
}

void CMainFrame::Dump(CDumpContext& dc) const
{
   CFrameWnd::Dump(dc);
}

#endif
```

Beschreibung der Implementationsdatei

IMPLEMENT_DYNCREATE gehört zum *DECLARE_DYNCREATE*-Makro – stellt also quasi dessen Gegenstück dar – und soll hier nicht weiter erläutert werden.

Statuszeilen-informationen

Das folgende *indicators*-Array enthält eine Aufzählung der in der Statuszeile des Fensters vorhandenen Komponenten und wird später in diesem Kapitel noch näher untersucht, wenn es um das Hinzufügen eigener Daten in dieser Informationsquelle geht.

Einleitende Fenster-vorbereitungen

PreCreateWindow erlaubt es, spezifizierte Angaben zur Erzeugung des Fensters zu machen, beispielsweise Stilflags zu manipulieren und Ähnliches. Das wird standardmäßig nicht benötigt, sodass die Methode zwar vorhanden ist, aber selbst keine Änderungen an dem übergebenen *CreateStruct*-Parameter vornimmt. Schauen Sie in die Online-Hilfe, um nähere Informationen über die hier zu Verfügung stehenden Möglichkeiten zu finden.

Die *OnCreate*-Methode

Erzeugung eines Rahmenfensters

Hier wird das eigentliche Rahmenfenster erzeugt (durch Weiterleitung des *lpCreateStruct* Parameters an die Basisklassenmethode *CFrameWnd::OnCreate*), sowie die beiden Leisten (Status- und Werkzeugleiste) erzeugt.

Die hierzu eingesetzten Methoden *CreateEx* und *Create* finden Sie im Anhang dieses Buches erläutert.

Abschließend wird die Werkzeugleiste als andockbar definiert und gleich an das Hauptfenster angebunden. Sie könnten hier eine nicht andockbare Werkzeugleiste erzeugen, indem Sie die drei zugehörigen Zeilen auskommentieren oder löschen.

Diagnosemethoden

Die beiden Diagnosemethoden *AssertValid* und *Dump* werden im Kapitel über den Debugger noch näher besprochen. Für den Augenblick beachten Sie bitte, dass diese Methoden nur definiert werden, wenn ein Debug-Build des Projekts erzeugt wird.

Bedingte Kompilierung von Debug-Funktionalitäten

Die Präprozesseranweisungen *#ifdef _DEBUG ... #endif* machen die Methoden in Release-Builds für den Compiler unsichtbar und somit vom Programm aus nicht erreichbar. Somit wird dafür gesorgt, dass Release-Builds unter Umständen deutlich schneller als ihre Debug-Build-Versionen ablaufen können.

Die Dokumentenklasse

Die generierte Dokumentenklasse

Da das Hauptfenster der Anwendung ohne Daten noch relativ leer aussieht, müssen Sie eine Möglichkeit vorsehen, Informationen zu speichern. Der Anwendungsassistent hat zu diesem Zweck bereits eine Dokumentenklasse namens *CApfelmaennchenDoc* angelegt, die hierfür hervorragend geeignet ist:

Listing 5.3 ApfelmaennchenDoc.h

```
class CApfelmaennchenDoc : public CDocument
{
protected: // Nur aus Serialisierung erstellen
   CApfelmaennchenDoc();
   DECLARE_DYNCREATE(CApfelmaennchenDoc)

// Attribute
public:

// Operationen
public:

// Überschreibungen
   public:
   virtual BOOL OnNewDocument();
   virtual void Serialize(CArchive& ar);
```

```
// Implementierung
public:
   virtual ~CApfelmaennchenDoc();
#ifdef _DEBUG
   virtual void AssertValid() const;
   virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generierte Funktionen für die Meldungstabellen
protected:
   DECLARE_MESSAGE_MAP()
};
```

OnNewDocument und *Serialize*

Die Klasse *CApfelmaennchenDoc* wirkt in ihrer Ausgangsform noch nackt und leer. Das liegt daran, dass sie zwar ein Grundgerüst für die später tatsächlich benötigte Dokumentenklasse bildet, inhaltlich aber bis auf zwei Methoden noch nichts enthält, was auf die aufzunehmenden Daten schließen lassen würde.

Dokumentenverwaltung

Diese beiden Methoden jedoch, *OnNewDocument* und *Serialize*, sind für SDI-Anwendungen zwei der wichtigsten Dokumentverwaltungsfunktionen überhaupt. Ihre Implementation ist in der Datei *ApfelmaennchenDoc.cpp* zu finden:

Listing 5.4 Apfelmaennchen-Doc.cpp

```
#include "stdafx.h"
#include "Apfelmaennchen.h"

#include "ApfelmaennchenDoc.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif


// CApfelmaennchenDoc

IMPLEMENT_DYNCREATE(CApfelmaennchenDoc, CDocument)

BEGIN_MESSAGE_MAP(CApfelmaennchenDoc, CDocument)
END_MESSAGE_MAP()


// CApfelmaennchenDoc Erstellung/Zerstörung

CApfelmaennchenDoc::CApfelmaennchenDoc()
```

```
{
   // TODO: Hier Code für One-Time-Konstruktion
// einfügen

}

CApfelmaennchenDoc::~CApfelmaennchenDoc()
{
}

BOOL CApfelmaennchenDoc::OnNewDocument()
{
   if (!CDocument::OnNewDocument())
      return FALSE;

   // TODO: Hier Code zur Reinitialisierung
// einfügen
   // (SDI-Dokumente verwenden dieses Dokument)

   return TRUE;
}

// CApfelmaennchenDoc Serialisierung

void CApfelmaennchenDoc::Serialize(CArchive& ar)
{
   if (ar.IsStoring())
   {
      // TODO: Hier Code zum Speichern einfügen
   }
   else
   {
      // TODO: Hier Code zum Laden einfügen
   }
}

// CApfelmaennchenDoc Diagnose

#ifdef _DEBUG
void CApfelmaennchenDoc::AssertValid() const
{
   CDocument::AssertValid();
}

void CApfelmaennchenDoc::Dump(CDumpContext& dc) const
{
   CDocument::Dump(dc);
}
#endif
```

Implementationen von *CApfelmaennchenDoc*

Genau wie die Deklarationen wirken die Implementationen der beiden Methoden noch unfertig – und sind es auch.

Neue Dokumente

OnNewDocument dient in SDI-Anwendungen dazu, die Daten eines Dokuments zu reinitialisieren. Angenommen, Sie hätten bereits einige Informationen in ein Dokument eingetragen und erzeugen dann über den von Windows-Programmen bekannten Punkt *Datei > Neu* ein neues Dokument, werden die alten Daten gegebenenfalls gespeichert und dann verworfen.

Verwerfen bedeutet hierbei, dass *OnNewDocument* aufgerufen wird und möglicherweise Variablenwerte des Dokuments zurücksetzt – Sie bringen das Dokument also gewissermaßen in einen Startzustand; bei einer Textverarbeitung könnte das das Initialisieren des Textinhalts mit einem leeren String sein.

Laden und Speichern von Daten

Serialize, auf der anderen Seite, dient zum Laden und Speichern (*Serialisierung* genannt) eines Dokuments. Was hier genau geschieht, wird im weiteren Verlauf des Kapitels noch näher geklärt.

Bleibt noch eine Betrachtung der Ansichtsklasse für neue Dokumente des Typs Apfelmännchen.

CApfelmaennchenView – eine Ansichtsklasse

Die Ansichtsklasse teilt sich, wie zu erwarten, in eine Header- und eine Implementationsdatei auf. Sie bildet ein Grundgerüst, um die im Dokument enthaltenen Daten auf den Schirm beziehungsweise in das Fenster zu bringen.

Hier zunächst die Headerdatei:

Listing 5.5 Apfelmaennchen-View.h

```
class CApfelmaennchenView : public CView
{
protected: // Nur aus Serialisierung erstellen
   CApfelmaennchenView();
   DECLARE_DYNCREATE(CApfelmaennchenView)

// Attribute
public:
   CApfelmaennchenDoc* GetDocument() const;

// Operationen
public:

// Überschreibungen
   public:
   virtual void OnDraw(CDC* pDC);  // Überladen, um
// diese Ansicht darzustellen
virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
```

```
protected:
   virtual BOOL OnPreparePrinting(CPrintInfo*
pInfo);
   virtual void OnBeginPrinting(CDC* pDC,
CPrintInfo* pInfo);
   virtual void OnEndPrinting(CDC* pDC, CPrintInfo*
pInfo);

// Implementierung
public:
   virtual ~CApfelmaennchenView();
#ifdef _DEBUG
   virtual void AssertValid() const;
   virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generierte Funktionen für die Meldungstabellen
protected:
   DECLARE_MESSAGE_MAP()
};
```

Zugriff auf das Dokument, Drucken und Zeichnen

Die Ansichtsklasse enthält eine Reihe von nützlichen Methoden, allen voran *GetDocument*, die einen Zugriff auf die Dokumentenschnittstelle gestattet.

Über den zurückgelieferten Zeiger kann dann bequem via Zugriffsfunktionen auf die abstrahierten Daten des Dokuments zugegriffen werden.

Visualisierung von Dokumentinhalten

Neben dem Zugriff ist die Darstellung die eigentliche Hauptaufgabe der Ansichtsklasse. Die überschriebene Methode *OnDraw* dient zur Ausgabe der ermittelten Daten auf einen Gerätekontext – dabei ist es (mehr oder weniger) unerheblich, ob dieser einen Bildschirm oder eine Druckausgabe repräsentiert.

Es gibt jedoch noch einige zusätzliche Operationen für die konkrete Ausgabe von Informationen auf den Drucker – namentlich sind dies *OnPreparePrinting*, *OnBeginPrinting* und *OnEndPrinting*. Sie können beispielsweise genutzt werden, um eine bestimmte Sortierung beim Drucken zu erreichen, oder sonstige Einstellungen in diesem Bereich zu tätigen.

PreCreateWindow schließlich wird analog zu *CFrameWnd::PreCreateWindow* aufgerufen, gerade bevor das Fenster erzeugt wird. Hier wäre es also wieder möglich, passende Stilflags zu setzen und so weiter.

Implementation von *CApfelmaennchenView*

Die Implementation dieser Methoden befindet sich in der Datei *ApfelmaennchenView.cpp*.

Listing 5.6 Apfelmaennchen-Doc.cpp

```
#include "stdafx.h"
#include "Apfelmaennchen.h"

#include "ApfelmaennchenDoc.h"
#include "ApfelmaennchenView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif


// CApfelmaennchenView

IMPLEMENT_DYNCREATE(CApfelmaennchenView, CView)

BEGIN_MESSAGE_MAP(CApfelmaennchenView, CView)
   // Standarddruckbefehle
   ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
   ON_COMMAND(ID_FILE_PRINT_DIRECT,
CView::OnFilePrint)
   ON_COMMAND(ID_FILE_PRINT_PREVIEW,
CView::OnFilePrintPreview)
END_MESSAGE_MAP()

// CApfelmaennchenView Erstellung/Zerstörung

CApfelmaennchenView::CApfelmaennchenView()
{
   // TODO: Hier Code zum Erstellen einfügen

}

CApfelmaennchenView::~CApfelmaennchenView()
{
}

BOOL
CApfelmaennchenView::PreCreateWindow(CREATESTRUCT& cs)
{
   // TODO: Ändern Sie hier die Fensterklasse oder
// die Darstellung, indem Sie
   //  CREATESTRUCT cs modifizieren.

   return CView::PreCreateWindow(cs);
```

```
}

// CApfelmaennchenView-Zeichnung

void CApfelmaennchenView::OnDraw(CDC* /*pDC*/)
{
   CApfelmaennchenDoc* pDoc = GetDocument();
   ASSERT_VALID(pDoc);

   // TODO: Code zum Zeichnen der systemeigenen
// Daten hinzufügen
}


// CApfelmaennchenView drucken

BOOL
 CApfelmaennchenView::OnPreparePrinting(CPrintInfo*
pInfo)
{
   // Standardvorbereitung
   return DoPreparePrinting(pInfo);
}

void CApfelmaennchenView::OnBeginPrinting(CDC*
/*pDC*/, CPrintInfo* /*pInfo*/)
{
   // TODO: Zusätzliche Initialisierung vor dem
// Drucken hier einfügen
}

void CApfelmaennchenView::OnEndPrinting(CDC* /*pDC*/,
 CPrintInfo* /*pInfo*/)
{
   // TODO: Bereinigung nach dem Drucken einfügen
}


// CApfelmaennchenView Diagnose

#ifdef _DEBUG
void CApfelmaennchenView::AssertValid() const
{
   CView::AssertValid();
}

void CApfelmaennchenView::Dump(CDumpContext& dc) const
{
```

```
   CView::Dump(dc);
}

CApfelmaennchenDoc* CApfelmaennchenView::GetDocument() const //
Nicht-Debugversion ist inline
{
   ASSERT(m_pDocument
->IsKindOf(RUNTIME_CLASS(CApfelmaennchenDoc)));
   return (CApfelmaennchenDoc*)m_pDocument;
}
#endif
```

Behandlungsmethoden von Druckmenüeinträgen

Jede Ansichtsklasse ist von der Basisklasse *CView* abgeleitet, die bereits eine ganze Reihe von Methoden enthält, die standardmäßig beispielsweise beim Drucken Verwendung finden.

Nachrichten zur Druckausgabe

So enthält die Implementationsdatei *ApfelmaennchenDoc.cpp* in ihrer Messagemap auch einige Zuweisungen von *CView*-Behandlungsmethoden an die Menüelementereignisse, die durch die Komponenten *ID_FILE_PRINT* (Drucken mit Optionen), *ID_FILE_PRINT_DIRECT* (Drucken ohne Optionen) und *ID_FILE_PRINT_PREVIEW* (Seitenansicht) ausgelöst werden.

In der Regel brauchen Sie an dieser Stelle nicht tätig zu werden, es sei denn, Sie wünschen alternative Behandlungen dieser drei Aktionen.

Aus den *CView*-Methoden heraus werden dann vom Framework die Methoden *OnPreparePrinting* (vorbereitende Aktionen), *OnBeginPrinting* (Initialisierungen direkt vor Druckbeginn) und *OnEndPrinting* (Aufräumarbeiten nach Beendigung des Druckvorgangs) aufgerufen.

Diese sind eingangs leer und, wie Sie später noch sehen werden, brauchen im einfachsten Fall auch keine weitere Funktionalität aufzunehmen.

Sonstige Methoden von *CApfelmaennchenView*

Der Rest von *CApfelmaennchenView* sieht eingangs noch recht übersichtlich aus – neben der *GetDocument*-Methode zum Zugriff auf das eigentliche Dokument, existiert eine Funktion *OnDraw*, die derzeit noch nichts anderes macht, als über das eben genannte *GetDocument* einen Zeiger auf das Dokument zu erhalten.

Der eigentliche Zeichenvorgang ist derzeit noch nicht implementiert, schließlich kann der Anwendungsassistent nicht wissen, welche Art von Informationen Sie ausgeben möchten.

Auch in *CApfelmaennchenView* sind wieder die Diagnosemethoden *AssertValid* und *Dump* vorhanden.

Zusammenfassung der Klassen

Nachdem Sie jetzt einen kurzen Überblick über die Inhalte der Klassen erhalten haben, soll ihre Bedeutung noch einmal kurz zusammengefasst werden:

- *CMainFrame*: Hauptfensterklasse, enthält Werkzeugleiste, Menüleiste und Statuszeile.
- *CApfelmaennchenDoc*: Dokumentenklasse. Enthält Methoden zur Serialisierung (Laden/Speichern) von Dokumenten sowie zum Neuinitialisieren von Daten.
- *CApfelmaennchenView*: Ansichtsklasse. Enthält Methoden zur Vor- und Nachbereitung von Druckvorgängen sowie jeweils eine Funktion zum Zugriff auf das Dokumentobjekt und zum Zeichnen der gewünschten Informationen.

Ausfüllen der *OnDraw*-Methode

Es sind nun die einzelnen vorhandenen Dateien vorgeführt worden – höchste Zeit also, selbst erste behutsame Versuche durchzuführen, etwas an diesem zunächst vielleicht beeindruckend wirkenden System zu verändern.

Zeichnen von Apfelmännchen

Begonnen werden soll mit dem Zeichnen der Apfelmännchen, ohne dabei Rücksicht auf irgendwelche Benutzereingaben oder Datenbestände zu nehmen.

Das Zeichnen innerhalb der *OnDraw*-Methode einer Ansichtsklasse funktioniert prinzipiell genauso wie Sie es schon bei den Dialogfeldern im letzten Kapitel gesehen haben, die *OnDraw*-Methode stellt dabei den für den Bildschirm passenden Gerätekontext zur Verfügung – genau genommen auch den für den Drucker, falls eine Printausgabe getätigt werden soll, aber damit wollen wir uns erst zu einem späteren Zeitpunkt beschäftigen.

Berechnen von Apfelmännchen

Apfelmännchen (die auch Mandelbrot-Mengen genannt werden), werden nach der einfach anmutenden Formel

$$X_{n+1} = (X_n * X_n) + C$$

berechnet. Dabei steht C für eine komplexe Zahl.

Ein wenig Schulmathematik

Es ist für die folgenden Überlegungen nicht wichtig, ob Sie wissen, wie mit komplexen Zahlen gerechnet wird, stellen Sie sich einfach das aus der Schule bekannte Gauss'sche Koordinatensystem vor, bei dem eine (normalerweise mit X bezeichnete) horizontale Achse von einer (mit Y titulierten) vertikalen Achse geschnitten wird. Der Schnittpunkt dieser beiden Geraden ist genau an der Position (0, 0).

Eine komplexe Zahl C ist in unserem Fall einfach ein beliebiger Punkt innerhalb dieses Systems. Etwas weiter in die Mathematik hineingreifend sei erwähnt, dass die X-Achse den Realteil von C, die Y-Achse den Imaginärteil beschreibt.

Für ein Apfelmännchen wird nun jeder einzelne Punkt (vereinfacht gesprochen) herangezogen und in die oben stehende Formel gesteckt. Weiterhin wird für X_0 der Wert 0 angenommen.

Ausgehend von einem Punkt C und X_0 mit dem Wert 0 lässt sich nun X_1 berechnen, daraus dann wiederum mit demselben C X_2 und so weiter. Das Ganze wiederholt man so lange, bis X_n eine bestimmte Grenze (für Apfelmännchen nimmt man hier in der Regel den Wert 4.0 an) überschreitet.

Sie schauen nun auf den Index bei X und wissen dann, wie viele Iterationen nötig waren, um den Grenzwert zu überschreiten. Bei manchen Cs wird der Grenzwert allerdings nie erreicht, weshalb man nach einer bestimmten Iterationstiefe (zum Beispiel 10) die Berechnung abbricht und diesen (höchsten) Wert als Index annimmt.

Bei schwarz-weißen Apfelmännchen werden nur die Punkte gezeichnet, bei denen die höchste Iterationstiefe errechnet wurde. Alle anderen Punkte bleiben weiß.

Bei farbigen Apfelmännchen wird abhängig von der Iterationstiefe eine andere Farbe gewählt, normalerweise ein Farbverlauf, wobei Koordinaten mit der höchsten Iterationstiefe in der Regel schwarz gezeichnet werden.

Soviel zur Theorie, kommen wir zur praktischen Implementation.

Implementation der Apfelmännchen-Berechnung

Wie oben beschrieben wurde, muss für jeden Punkt in der Koordinatenebene bestimmt werden, wie viele Iterationen durchlaufen werden können.

Unterteilung der Berechnung in Funktionsaufrufe

Hier bietet es sich sicherlich an, eine Funktion zu schreiben, die für einen gegebenen Punkt genau diesen Wert berechnet. Da es hierbei um eine Verarbeitung von Dokumentdaten handelt und diese nur für die Darstellung benötigt werden (insbesondere sollen die Werte nicht gespeichert, sondern gleich geeignet gezeichnet werden), gehört die Methode in die Ansichtsklasse.

Fügen Sie daher nun eine Methode *GetNumberOfIterations* in die *CApfelmaennchenView*-Klasse ein und zwar zunächst die Deklaration in *ApfelmaennchenView.h*. Es bleibt dabei Ihnen überlassen, ob Sie den Funktionsassistenten, den Sie in einem vorangegangenen Kapitel kennen gelernt haben, verwenden, oder den Code manuell eintippen:

Listing 5.7
Funktionsdeklaration

```
int GetNumberOfIterations(double in_dX, double in_dY,
 int in_nMaxIter);
```

GetNumberOfIterations soll die eigentliche Rechnung durchführen. Da es hier um die Verrechnung von komplexen Zahlen geht, wird einfach die Implementation vorgegeben – für Mathematiker sollten die Zeilen leicht verständlich sein, alle anderen seien an entsprechende Fachliteratur verwiesen:

Listing 5.8 **Implementation von *GetNumberOfIterations***

```
int CApfelmaennchenView::GetNumberOfIterations(double
 in_dX, double in_dY,
 int in_nMaxIter)
{
   // temporäre Variablen deklarieren
   double dSX, dSY, dZX, dZY;
   int i;

   // Variablen initialisieren
   dZX = in_dX;
   dZY = in_dY;
   dSX = dZX * dZX;
   dSY = dZY * dZY;
   i = 0;

   // iterieren
   while ((i <= in_nMaxIter) && (dSX + dSY <= 4.0))
   {
      dZY = ((dZX + dZX) * dZY) + in_dY;
      dZX = dSX – dSY + in_dX;
      dSX = dZX * dZX;
      dSY = dZY * dZY;
      i++;
   }

   // Iterationstiefe zurückgeben
   return (i);
}
```

Hier wird die Anzahl der Iterationen bestimmt, und als Rückgabewert an den Aufrufer übermittelt.

Zeichnen der Mandelbrot-Menge

Dieser befindet sich in der Methode *OnDraw* der Klasse *CApfelmaennchenView*. Wir hatten bereits festgestellt, dass diese Funktion zur Darstellung der Dokumentdaten verwendet wird.

Zeichnen eines ersten Apfelmännchens

Für einen ersten Versuch soll ein Mandelbrot mit einer Größe von 200*200 Pixel gezeichnet werden:

Listing 5.9
Die veränderte *OnDraw*-Methode

```
void CApfelmaennchenView::OnDraw(CDC* pDC)
{
   CApfelmaennchenDoc* pDoc = GetDocument();
   ASSERT_VALID(pDoc);

   // TODO: Code zum Zeichnen der systemeigenen
// Daten hinzufügen

   // Größe der Ausgabefläche festlegen
   int nSize = 200;

   // maximale Iterationstiefe bestimmen
   int nMaxIter = 10;

   // Startposition für die Ausgabe festlegen
   POINT Position;
   Position.x = 0;
   Position.y = 0;

   // Bereich, in dem das Apfelmännchen berechnet
// werden soll
   double dXStart, dYStart, dXEnd, dYEnd;
   dXStart = -2.0;
   dYStart = -2.0;
   dXEnd   = 2.0;
   dYEnd   = 2.0;

   // momentane Position in der Mandelbrotmenge
   double dX, dY;
   dX = dXStart;
   dY = dYStart;

   // schrittweite in der Mandelbrotmenge
   double dSX, dSY;
   dSX = (dXEnd – dXStart) / nSize;
   dSY = (dYEnd – dYStart) / nSize;

   // alle Zeilen berechnen
   while (Position.y < nSize)
   {
      // neue Zeile, x-Wert zurücksetzen
      Position.x = 0;
      dX = dXStart;

      // jeden Pixel der Zeile berechnen
      while (Position.x < nSize)
      {
         // Iterationswert bestimmen
```

```
            int nIterations =
GetNumberOfIterations(dX, dY,
nMaxIter);

            // nur einen Punkt setzen, wenn die
// maximale Iterationstiefe
// überschritten wurde
            if (nIterations >= nMaxIter)
            {
               pDC->SetPixel(Position,
RGB(0,0,0));
            }

            // zum nächsten Bildpunkt
// fortschreiten
            Position.x++;
            dX += dSX;
         }

         // nächste Zeile
         Position.y++;
         dY += dSY;
      }
}
```

Erklärungen zu *OnDraw*

Zunächst werden hier eine Reihe von Initialisierung durchgeführt, die den Bereich und die Auflösung des zu zeichnenden Mandelbrots bestimmt.

Der gewählte Bereich (-2.0, -2.0) bis (2.0, 2.0) ist gerade der klassische Ausschnitt, der auch in Büchern gerne gewählt wird. Er sieht aus wie in der nachstehenden Abbildung:

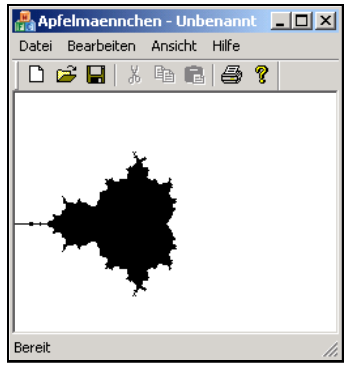


Abb. 5.3
Erste Ergebnisse des Mandelbrot-Programms

Annahmen über die Darstellung

Im weiteren Verlauf von *OnDraw* gehen wir davon aus, dass pro Pixel genau ein Punkt im Koordinatensystem steht. Wenn wir den Bereich von (-2.0, -2.0) – (2.0, 2.0) abdecken wollen, entspricht ein Pixel gerade einem 0,02*0,02 Einheiten großen Punkt im Koordinatensystem.

Es werden zwei Zeilen- und Spaltenzähler verwendet, die jeweils die aktuelle Position auf der Zeichenfläche und im Koordinatensystem festhalten und dann geeignet die *GetNumberOfIterations*-Funktion aufrufen. Ist der Rückgabewert größer oder gleich der Variablen *nMaxIter*, wird ein schwarzer Punkt gezeichnet.

Obwohl dieses Vorgehen funktioniert, fällt doch sofort ein Problem auf: was, wenn das Fenster kleiner als 200*200 Pixel ist? Oder deutlich größer? Wie bringe ich das Mandelbrot auf eine ansprechende Größe, beispielsweise gerade durch Ausnutzung der kompletten Zeichenfläche?

Abbildungsmodi unter Windows

An dieser Stelle treten die so genannten Abbildungsmodi auf den Plan. Hierbei handelt es sich um Beschreibungen, wie Pixel in Datenform in ein Fenster übertragen werden.

Der Ursprung der Zeichenfläche

Bislang sind wir davon ausgegangen, dass sich der Punkt (0, 0) unserer Zeichenfläche in der linken oberen Ecke befindet und die X- und Y-Werte beim Zeichnen nach rechts beziehungsweise nach unten vergrößern.

Offensichtlich ist dieses auch tatsächlich der Fall, doch warum eigentlich?

Kurz gesagt, gerade diese Vorstellung von der virtuellen Leinwand (in Form unseres Gerätekontexts und somit in zweiter Instanz des tatsächlichen Fensterinhalts) ist bei Windows standardmäßig voreingestellt.

Was zum schnellen Austesten durchaus Vorteile bringt, entpuppt sich bei näherer Überlegung aber als Problem: Stellen Sie sich vor, Sie möchten ein Quadrat zeichnen, dass 320*200 Pixel groß ist. Ist das zugrunde liegende Fenster genauso groß, wird es den gesamten Bereich ausfüllen.

Zahlenspielereien

Vergrößern Sie das Fenster nun auf 640*480 Pixel, füllt das Quadrat nur noch ein Viertel des Platzes aus.

Noch dramatischer wird es, wenn Sie die Grafik auf einem herkömmlichen Drucker mit zum Beispiel 600 dpi (Dots per Inch, also Punkte pro Zoll, wobei ein Zoll 2,54 cm sind) Auflösung aus. Das Quadrat ist gerade einmal noch ungefähr 1,25 cm^2 groß.

Oder andersherum: Um eine komplette DIN-A4-Seite (210 mm*297 mm) mit einem ausgedruckten Rechteck zu füllen, müsste dieses nach unserer bisherigen Methode ungefähr 4.960 * 7.015 Pixel groß sein.

Es bedarf keiner großen Vorstellungskraft, um sich zu denken, dass dieser Weg sicherlich als suboptimal bezeichnet werden darf.

Festlegen des Koordinatenursprungs

Der voreingestellte Abbildungsmodus, der uns nun die ganzen Kopfschmerzen bereitet, heißt *MM_TEXT*. Wie bereits festgestellt, deckt sich bei ihm die linke obere Ecke mit dem Punkt (0, 0).

Sie können diesen Umstand jedoch durch einen einfachen Aufruf ändern und die linke obere Ecke auf einen beliebigen anderen Punkt im kartesischen Koordinatensystem umlegen.

Verschieben des Ursprungs

Hierfür gibt es die Methode *SetWindowOrg*, die als Parameter den Ursprung (den Wert für die linke obere Ecke) in Form von X- und Y-Parametern übergeben bekommt.

Sehen Sie sich die folgenden zwei Zeilen Quelltext an:

```
pDC->SetWindowOrg(100, 50);
pDC->TextOut(100, 50, "Versetzte Ausgabe");
```

Sie könnten diese Zeilen in *OnDraw* einfügen, um zu sehen, was passiert. Die folgende Abbildung veranschaulicht Ihnen dieses aber auch hinreichend genau und erspart das umständliche Abändern der Methode des Beispielprojekts:

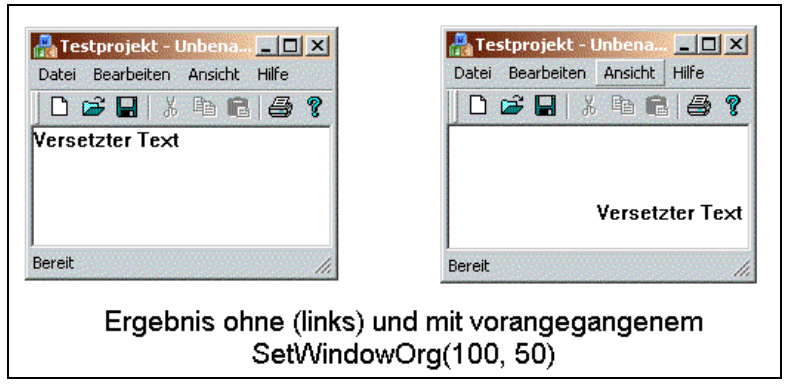


Abb. 5.4 **Das Ergebnis von *SetWindowOrg***

Ergebnis ohne (links) und mit vorangegangenem SetWindowOrg(100, 50)

Wie Sie sehen können, sorgt das Verändern des Koordinatenursprungs für eine Veränderung der Ausgabe. Man spricht hier von logischen Koordinaten. (100, 50) ist eine solche logische Koordinate, die sich nach einem Aufruf der oben stehenden *SetWindowOrg*-Zeile mit der Gerätekontext-Koordinate (0, 0) deckt.

Logische Einheiten

Rechnen mit logischen Einheiten

Wenn es logische Koordinaten gibt, liegt es nahe, dass auch logische Einheiten existieren, die unabhängig vom Gerätekontext sind und dort lediglich in einer geeigneten Form übersetzt werden.

Tatsächlich gibt es eine ganze Reihe von logischen Einheiten, die in der folgenden Tabelle mit ihren zugehörigen Abbildungsmodi zusammengefasst sind:

Tabelle 5.1 **Abbildungsmodi**

Abbildungsmodus	Logische Einheit in diesem Modus entspricht
MM_TEXT	1 Pixel
MM_LOENGLISH	0,01 Zoll (= 0,0254 cm)
MM_HIENGLISH	0,001 Zoll (= 0,00254 cm)
MM_LOMETRIC	0,1 mm
MM_HIMETRIC	0,01 mm
MM_TWIPS	1/1440 Zoll
MM_ANISOTROPIC	Beide Achsen frei wählbar
MM_ISOTROPIC	Frei wählbar, aber verzerrungsfrei

Die Verwendung dieser Abbildungsmodi ist denkbar einfach: Der Gerätekontext wird mit einem Funktionsaufruf von *SetMapMode* und dem Namen des Abbildungsmodus aus der obigen Tabelle passend gesetzt.

Danach kann einfach wie gehabt gezeichnet werden. Es ist jedoch zu beachten, dass in allen Abbildungsmodi außer *MM_TEXT* die y-Koordinate nach unten hin abnimmt. Ist der Gerätekontextursprung – seine linke obere Ecke – auf (0, 0) gesetzt und wollen Sie einen Pixel an die Stelle (0, 10) setzen, müssen Sie als Koordinate (0, -10) angeben.

Beispiel zu den Abbildungsmodi

Abbildungsmodus-Exkurs

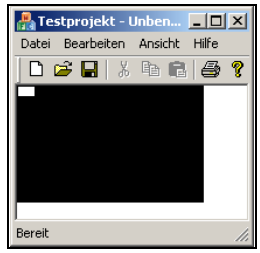
Um ein besseres Gefühl für die Abbildungsmodi zu bekommen, sei ein kleines Beispiel angeführt, dass zwei Rechtecke mit den gleichen Zahlwerten, jedoch in unterschiedlichen Modi zeichnet:

```
CRect test(0, 0, 320, -200);
pDC->SetMapMode(MM_LOMETRIC);
pDC->SelectStockObject(BLACK_BRUSH);
pDC->Rectangle(test);

pDC->SetMapMode(MM_HIMETRIC);
```

```
pDC->SelectStockObject(WHITE_BRUSH);
pDC->Rectangle(test);
```

Das eingangs definierte Rechteck wird einmal im *MM_LOMETRIC* Abbildungsmodus (hier entspricht eine logische Einheit 0,1mm) in Schwarz und einmal im *MM_HIMETRIC* (logische Einheit ist 0,01mm) in Weiß gezeichnet.

Das Ergebnis sehen Sie hier:



Das zweite gezeichnete Rechteck ist tatsächlich um den Faktor 10 kleiner als das erste (schwarze) Rechteck.

Passende Ergebnisse

Mehr noch: Wenn Sie zum Beispiel das schwarze Rechteck am Bildschirm mit einem Lineal ausmessen, werden Sie feststellen, dass es tatsächlich ungefähr 3,2 cm breit und 2,0 cm hoch ist. Das liegt an der Verzerrungsungenauigkeit des Monitors – bei einem Ausdruck erhalten Sie schon deutlich präzisere Messungen.

Frei skalierbare Abbildungsmodi

Neben den Abbildungsmodi, die eine feste logische Einheitengröße vorgegeben, gibt es noch die beiden Modi *MM_ANISOTROPIC* und *MM_ISOTROPIC*, die auch nicht unerwähnt bleiben sollen.

Bei diesen Modi können Sie die Größe einer logischen Einheit frei bestimmen. Das folgende Beispiel zeigt dieses anhand des *MM_ANISOTROPIC*-Abbildungsmodus:

```
// Fenstergröße ermitteln
CRect Rectangle;
GetClientRect(Rectangle);

// Mapping Mode setzen
pDC->SetMapMode(MM_ANISOTROPIC);
pDC->SetWindowExt(5000, 5000);
pDC->SetViewportExt(Rectangle.right,Rectangle.bottom);
pDC->SetViewportOrg(Rectangle.right / 2,
Rectangle.bottom / 2);
```

```
// Ellipse zeichnen
pDC->Ellipse(CRect(-2500, -2500, 2500, 2500));
```

Hier wird zunächst die Größe der Client Area des aktuellen Fensters bestimmt und dann der Mapping Mode auf *MM_ANISOTROPIC* gesetzt.

Vorteile der frei skalierbaren Modi

Der Vorteil von *MM_ANISOTROPIC* und *MM_ISOTROPIC* besteht ja darin, dass die Größe einer logischen Einheit frei gewählt werden kann. Genau das geschieht jetzt mit dem Aufruf von *SetWindowExt*, das die Ausmaße des Fensters angibt (unabhängig von der tatsächlichen Fenstergröße).

Das heißt in diesem konkreten Fall, dass das Fenster 5.000 Einheiten in der Breite und 5.000 Einheiten in der Höhe darstellen kann.

Weiterhin wird der tatsächlich sichtbare Bereich angegeben. Das geschieht über die Viewport-Methoden *SetViewportExt* und *SetViewportOrg*. Der sichtbare Bereich soll gerade das gesamte Fenster sein, weswegen an dieser Stelle die zuvor über *GetClientRect* ermittelten Werte übergeben werden.

Der Ursprung dieses sichtbaren Bereichs wird mit der nachfolgenden Zeile gerade auf die Mitte des Fensters gesetzt und sorgt somit dafür, dass der Ursprung (0, 0) des Viewports mit dem Punkt (2500, 2500) des Fensters übereintrifft.

Das tatsächliche Zeichnen der Ellipse (die eigentlich ein Kreis sein müsste, da beide Größenangaben identisch sind) ergibt dann das folgende Ergebnis:

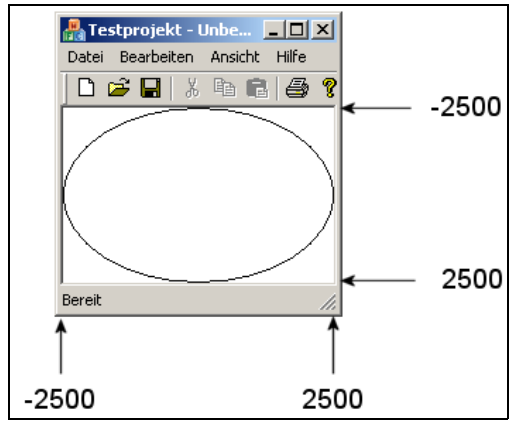


Abb. 5.5
MM_ANISOTROPIC

Hier zeigt sich auch gleich der Nachteil des *MM_ANISOTROPIC*-Abbildungsmodus: Die Darstellung wird verzerrt, da beide Achsen frei skalierbar sind.

MM_ISOTROPIC

Unterschiede zwischen *Anisotropic* und *Isotropic*

Möchten Sie ein unverzerrte Abbildung erreichen, können Sie den *MM_ISOTROPIC*-Abbildungsmodus verwenden. Wenn Sie das obige Beispiel heranziehen, und lediglich die Zeile

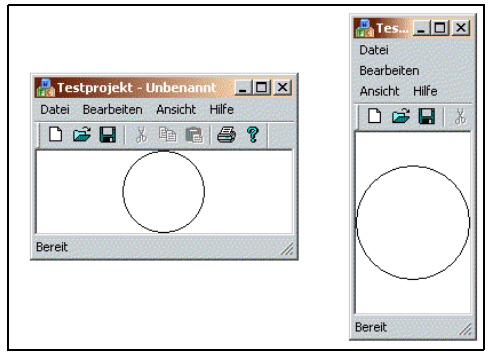
```
pDC->SetMapMode(MM_ANISOTROPIC);
```

durch

```
pDC->SetMapMode(MM_ISOTROPIC);
```

ersetzen, erhalten Sie eine Abbildung ähnlich der folgenden:

Abb. 5.6
MM_ISOTROPIC



Der *MM_ISOTROPIC*-Abbildungsmodus sorgt also dafür, dass die Darstellung verzerrungsfrei bleibt, auch wenn dieses bedeutet, dass Teile der Client Area nicht bezeichnet werden.

Es ist sehr wichtig, dass Sie sich mit den verschiedenen Abbildungsmodi auskennen. Spielen Sie daher ein wenig damit herum, setzen Sie unterschiedliche Abbildungsmodi, Viewport- und Fensterursprungskoordinaten und Ausmaße ein, bis Sie ein Gefühl für die Möglichkeiten bekommen haben.

Wahl eines Abbildungsmodus für das Apfelmännchen-Programm

Um das neu gelernte Wissen über die verschiedenen Abbildungsmodi gleich in die Praxis umzusetzen, soll nun auch im Apfelmännchen-Programm ein passender Modus gesetzt werden.

Vorbereiten des Gerätekontexts

Während in den einfachen Beispielen weiter oben die entsprechenden Funktionsaufrufe direkt in der *OnDraw*-Methode getätigt wurden, sollen sie im

Apfelmännchenprojekt dort eingefügt werden, wo sie zweckmäßiger Weise hingehören: in die Methode *OnPrepareDC*.

Diese ist in *CView* definiert und wird aufgerufen, bevor das Framework die *OnDraw*-Methode der Ansichtsklasse ausführt. Praktischerweise wird *OnPrepareDC* bereits der zu verwendende Gerätekontext übergeben, sodass hier die notwendigen Einstellungen vorgenommen werden können und der Funktionsumfang von *OnDraw* auf die tatsächliche Ausgabe begrenzt werden kann.

Hinzufügen von *Override*-Funktionen

Sie haben bereits gesehen, wie Behandlungsmethoden für Fensternachrichten mithilfe des *Eigenschaften*-Fensters des Visual Studio automatisch hinzugefügt werden können.

Override-Funktionen

Eine ähnliche Option besteht für das Anlegen so genannten *Override-Funktionen*, also Methoden, die in einer Basisklasse definiert wurden und nun in einer abgeleiteten Klasse überschrieben werden sollen.

Öffnen Sie dazu zunächst die Klassenansicht des Projekts und suchen dort die Klasse *CApfelmaennchenView* heraus. Stellen Sie weiterhin sicher, dass das *Eigenschaften*-Fenster geöffnet ist.

Sie sollten nun einen Überblick über die Klasse erhalten und können in der oberen Knopfleiste des *Eigenschaften*-Fensters den Punkt *Overrides* auswählen, woraufhin sich eine Liste aller überschreibbaren Basisklassenmethoden öffnet.

Suchen Sie in dieser Liste die Methode *OnPrepareDC* heraus, klicken Sie in die rechte Spalte und wählen Sie den Punkt <*Add*> *OnPrepareDC* aus:

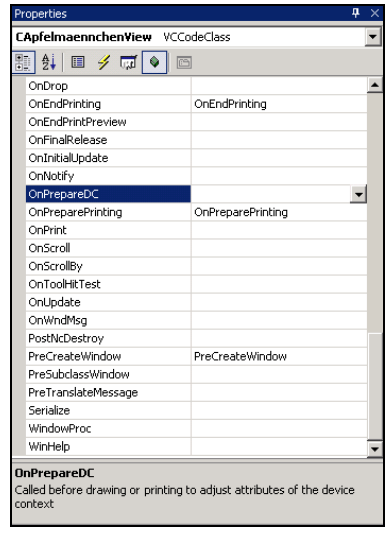


Abb. 5.7
Die *Override*-Methoden

Editieren von *OnPrepareDC*

Einrichten des Abbildungsmodus

In *OnPrepareDC* soll der Abbildungsmodus für das Apfelmännchenprogramm eingerichtet werden. Begeben Sie sich an die Stelle im Quelltext, an die der Assistent die *OnPrepareDC*-Methode eingefügt hat und editieren Sie den Funktionsrumpf wie folgt:

Listing 5.10 OnPrepareDC

```
void CApfelmaennchenView::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
   // lokale Variablen vorbereiten
   CRect ClientRect;
   int nAufloesung;

   // nur interessant, wenn nicht gedruckt wird
   if (!pDC->IsPrinting())
   {
      // Clientarea Größe ermitteln
      GetClientRect(ClientRect);

      // Mapmode einstellen. ANISOTROPIC ist
// sinnvoll,
      // da das Fenster beliebig in der Größe
// veraendert werden
      // kann
      pDC->SetMapMode(MM_ANISOTROPIC);

      // Auflösung berechnen
      nAufloesung = (ClientRect.Width() >
ClientRect.Height()) ?
         ClientRect.Width() :
ClientRect.Height();

      // Zeichenflächengröße festlegen
      pDC->SetWindowExt(nAufloesung,
nAufloesung);

      // Ursprung auf linke obere Ecke festlegen
      pDC->SetWindowOrg(0, 0);

      // Abbildungsbereich festlegen
      pDC->SetViewportExt(ClientRect.Width(),
ClientRect.Height());

      // Abbildungsursprung auf linke obere Ecke
// festlegen
      pDC->SetViewportOrg(0, 0);
   }
```

```
   // Standardbehandlung aufrufen
   CView::OnPrepareDC(pDC, pInfo);
}
```

Der gewählte Abbildungsmodus für das Apfelmännchen-Programm

In diesem Fall wurde *MM_ANISOTROPIC* als Abbildungsmodus gewählt, da wir uns nicht für eine verzerrungsfreie Darstellung des Apfelmännchens interessieren.

Ausfüllen des gesamten Bildbereichs

Vielmehr soll der gewählte Ausschnitt stets den kompletten Client-Bereich des Ausgabefensters ausfüllen. Daher wird mittels *SetViewportExt* der komplette Client-Bereich ausgewählt und als Auflösung die größere Dimension (Breite oder Höhe) gewählt.

Diese Änderungen machen es auch erforderlich, dass die *OnDraw*-Methode überarbeitet wird, denn sie arbeitet derzeit ja noch mit den festen Breiten- und Höhenangaben von 200 Pixeln, die ursprünglich festgelegt wurden:

```
void CApfelmaennchenView::OnDraw(CDC* pDC)
{
   CApfelmaennchenDoc* pDoc = GetDocument();
   ASSERT_VALID(pDoc);

   // Größe der Ausgabefläche festlegen
   CRect ClientRect;
   GetClientRect(ClientRect);
   int nAufloesung = ClientRect.Width() >
ClientRect.Height() ?
                ClientRect.Width() :
ClientRect.Height();

   // Anzahl der Gitterpunkte bestimmen
   int nGitterpunkte = nAufloesung + 1;

   // maximale Iterationstiefe bestimmen
   int nMaxIter = 10;

   // Startposition für die Ausgabe festlegen
   POINT Position;
   Position.x = 0;
   Position.y = 0;

   // Bereich, in dem das Apfelmännchen berechnet
// werden soll
   double dXStart, dYStart, dXEnd, dYEnd;
   dXStart = -2.0;
```

```
   dYStart = -2.0;
   dXEnd   = 2.0;
   dYEnd   = 2.0;

   // momentane Position in der Mandelbrotmenge
   double dX, dY;
   dX = dXStart;
   dY = dYStart;

   // Schrittweite in der Mandelbrotmenge
   double dSX, dSY;
   dSX = (dXEnd – dXStart) / nAufloesung;
   dSY = (dYEnd – dYStart) / nAufloesung;

   // alle Zeilen berechnen
   while (Position.y < nGitterpunkte)
   {
      // neue Zeile, x-Wert zurücksetzen
      Position.x = 0;
      dX = dXStart;

      // jeden Pixel der Zeile berechnen
      while (Position.x < nGitterpunkte)
      {
         // Iterationswert bestimmen
         int nIterations =
GetNumberOfIterations(dX, dY,
nMaxIter);

         // nur einen Punkt setzen, wenn die
// maximale Iterationstiefe
// überschritten wurde
         if (nIterations >= nMaxIter)
         {
            pDC->SetPixel(Position,
RGB(0,0,0));
         }

         // zum nächsten Bildpunkt
// fortschreiten
         Position.x++;
         dX += dSX;
      }

      // nächste Zeile
      Position.y++;
      dY += dSY;
   }
}
```

Änderungen in *OnDraw*

Zunächst wird in *OnDraw* die aktuelle Fenstergröße ermittelt und danach die Zahl der zu berechnenden Punkte (die Auflösung) kalkuliert.

Berechnen der Bildauflösung

Außerdem ist die Schrittweite im Koordinatensystem des Mandelbrots nun ebenfalls abhängig von diesem Wert (*nAufloesung*). Die restlichen Berechnungen bleiben unverändert, lediglich die Schleifenbedingungen verändern sich.

Wenn Sie das veränderte Programm nun kompilieren und starten, werden Sie feststellen, dass sich das Apfelmännchen der Fenstergröße anpasst. Beachten Sie hierbei insbesondere, wie lange die tatsächlichen Zeichenvorgänge dauern. Ein großes Fenster benötigt schon einige Sekunden, um komplett aufgebaut zu werden.

Benutzerdefinierte Apfelmännchen

Erweiterung um Benutzerinteraktion

Der nächste Schritt wird es sein, dem Benutzer zu gestatten, selbstständig den darzustellenden Bereich und die maximale Iterationstiefe festzulegen. Weiterhin soll eine Option existieren, die es erlaubt zu entscheiden, ob die Mandelbrote farbig oder schwarz-weiß darzustellen sind.

Hierfür bietet es sich an, einen eigenen Dialog anzulegen, der diese Einstellmöglichkeiten anbietet. Sie haben bereits mit dialogfeldbasierenden Applikationen gearbeitet, aber wie kann nun ein neuer Dialog in eine bestehende SDI-Anwendung integriert werden?

Öffnen Sie hierzu die Ressourcenansicht Ihres Projekts und öffnen durch einen Rechtsklick das Kontextmenü des Ordners *Dialogs*:

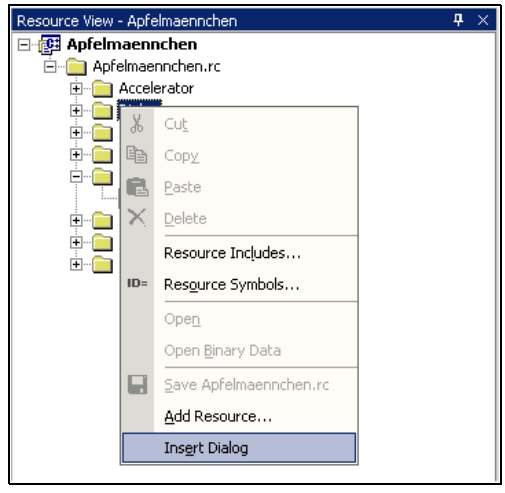
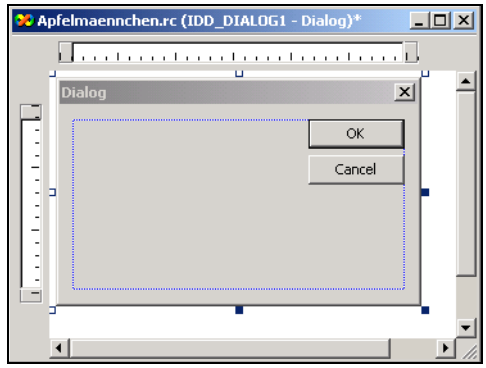


Abb. 5.8
Einfügen eines neuen Dialogs

Hinzufügen eines neuen Dialogs

Wählen Sie hier den Menüpunkt *Insert Dialog*. Ein Assistent fügt nun den neuen Dialog in Ihr Projekt ein und präsentiert ein zunächst leeres Dialogfeld, das dem der dialogfeldbasierenden Anwendung des letzten Kapitels ähnelt:

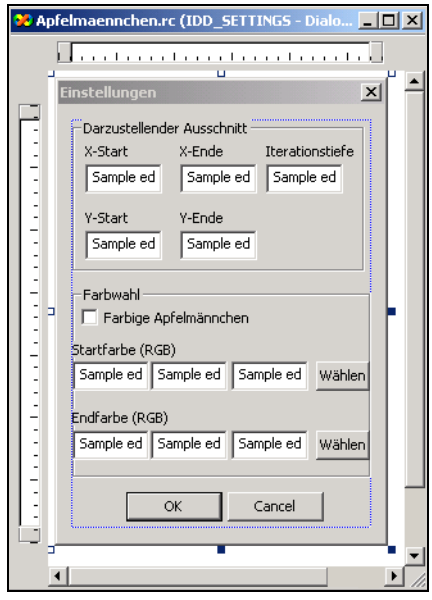
Abb. 5.9
Der neue Dialog



Editieren des neuen Dialogs

Editieren Sie das Dialogfeld so, dass es der folgenden Abbildung entspricht:

Abb. 5.10
Die Edit-Felder des Dialogs



Entnehmen Sie die zu vergebenden IDs der nachstehenden Tabelle:

Tabelle 5.2
IDs für Dialogfeldkomponenten

Komponente	ID
X-Start	IDC_XSTART
Y-Start	IDC_YSTART
X-Ende	IDC_XEND
Y-Ende	IDC_YENDE
Iterationstiefe	IDC_MAXITER
Farbige Apfelmännchen	IDC_COLOR
Startfarbe R	IDC_STARTRED
Startfarbe G	IDC_STARTGREEN
Startfarbe B	IDC_STARTBLUE
Endfarbe R	IDC_ENDRED
Endfarbe G	IDC_ENDGREEN
Endfarbe B	IDC_ENDBLUE
Startfarbe wählen	IDC_BUTTON1
Endfarbe wählen	IDC_BUTTON2

Bezeichnen Sie das Dialogfeld selbst als *IDD_SETTINGS*.

Anlegen einer Klasse für den neuen Dialog

Ein Dialogfeld als solches ist zunächst nicht mehr als eine Vorlage und wird es durch eine zugehörige Klasse zu einem verwendbaren Objekt.

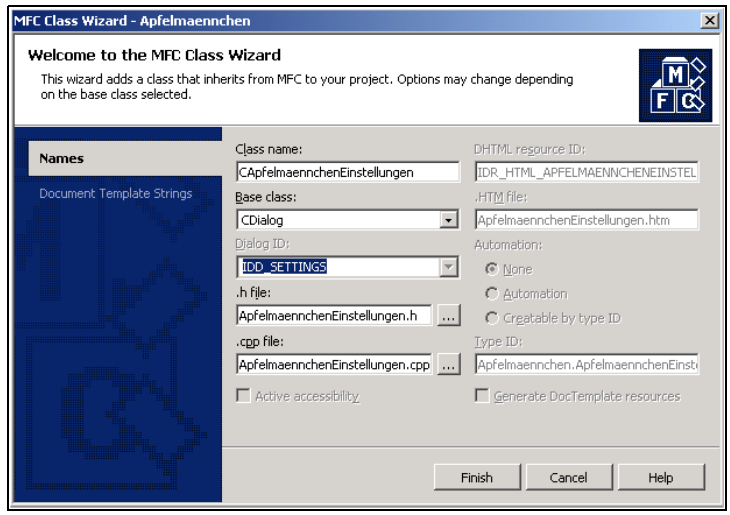
Ergänzen des Projekts um eine neue Dialogklasse

Klicken Sie mit der rechten Maustaste in den neuen Dialog und wählen aus dem auftauchenden Kontextmenü den Punkt *Add New Class* aus.

Hiermit wird der Klassenassistent der MFC aufgerufen, der es Ihnen erlaubt, eine neue Klasse in das Projekt zu integrieren, die auf Basis der Dialogfeldvorlage ein lauffähiges Dialogfeld implementiert.

Tragen Sie als Klassennamen *CApfelmaennchenEinstellungen* ein. Die restlichen Optionen können unverändert übernommen werden. Achten Sie jedoch darauf, dass bei *.h File ApfelmaennchenEinstellungen.h*, bei *.cpp File Apfelmaennchen-Einstellungen.cpp* und bei *Dialog ID IDD_SETTINGS* eingestellt ist:

Abb. 5.11
Anlegen einer neuen Dialogklasse



Bestätigen Sie die Angaben mit *Finish*.

Die erzeugte Dialogfeldklasse

Nehmen Sie sich nun die Zeit, einen Blick auf die erzeugten Dateien zu werfen. Hier ist als Erstes die Datei *ApfelmaennchenEinstellungen.h* zu nennen:

Listing 5.11
Apfelmaennchen-Einstellungen.h

```
#pragma once


// CApfelmaennchenEinstellungen dialog

class CApfelmaennchenEinstellungen : public CDialog
{
   DECLARE_DYNAMIC(CApfelmaennchenEinstellungen)

public:
   // standard constructor
   CApfelmaennchenEinstellungen(CWnd* pParent =
 NULL);

   virtual ~CApfelmaennchenEinstellungen();

// Dialog Data
   enum { IDD = IDD_SETTINGS };

protected:
   virtual void DoDataExchange(CDataExchange* pDX);
   // DDX/DDV support
```

```
   DECLARE_MESSAGE_MAP()
};
```

Diese Zeilen sind mit Ihrem jetzigen Wissen leicht zu verstehen, kommen Sie Ihnen doch sicherlich aus der Arbeit mit dem dialogfeldbasierenden Zeichenprogramm des letzten Kapitels bekannt vor.

ApfelmaennchenEinstellungen.cpp

Die Implementationsdatei *ApfelmaennchenEinstellungen.cpp* ist ebenfalls leicht zu durchschauen:

Listing 5.12 Apfelmaennchen-Einstellungen.cpp

```
// ApfelmaennchenEinstellungen.cpp : implementation
// file
//

#include "stdafx.h"
#include "Apfelmaennchen.h"
#include "ApfelmaennchenEinstellungen.h"


// CApfelmaennchenEinstellungen dialog

IMPLEMENT_DYNAMIC(CApfelmaennchenEinstellungen,
 CDialog)
CApfelmaennchenEinstellungen::
CApfelmaennchenEinstellungen(CWnd* pParent /*=NULL*/)
   : CDialog(CApfelmaennchenEinstellungen::IDD,
pParent)
{
}

CApfelmaennchenEinstellungen::
~CApfelmaennchenEinstellungen()
{
}

void CApfelmaennchenEinstellungen::
DoDataExchange(CDataExchange* pDX)
{
   CDialog::DoDataExchange(pDX);
}


BEGIN_MESSAGE_MAP(CApfelmaennchenEinstellungen,
 CDialog)
END_MESSAGE_MAP()
```

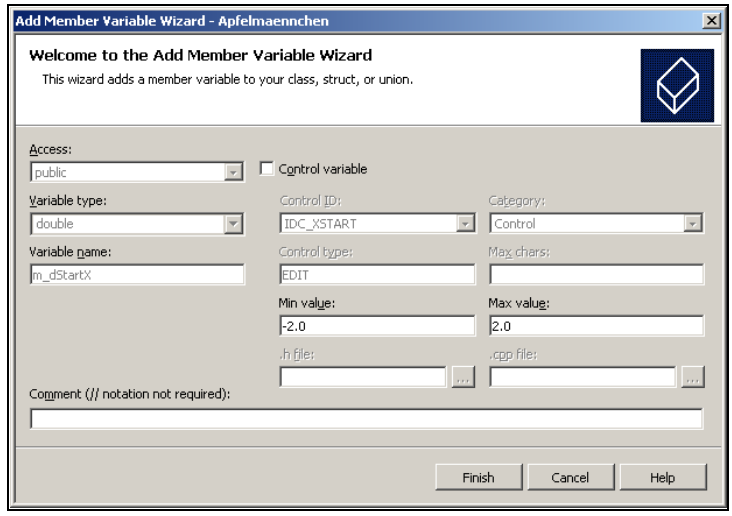
Festlegen von Variablen

In ihrer jetzigen Form ist die Dialogfeldklasse noch unausgereift. Sie ist zwar funktional, bietet aber noch keine vernünftigen Zugriffsmöglichkeiten auf die Werte Ihrer Steuerelemente.

Variablen an Komponenten binden

Der nächste Schritt besteht also darin, Variablen an die einzelnen Komponenten zu binden. Öffnen Sie wieder das Dialogfeld innerhalb des Ressourceneditors und klicken Sie das zu *X-Start* gehörende Edit-Feld mit der rechten Maustaste an. Wählen Sie den Punkt *Add Variable*. Es erscheint das folgende Dialogfeld:

Abb. 5.12
Eigenschaften der Variablen



Es ist nicht notwendig, sich mit den Steuerelementen selbst zu beschäftigen, beim Apfelmännchen-Projekt interessieren schlicht die tatsächlich eingestellten Werte der einzelnen Komponenten.

Echte Werte statt Kontrollvariablen

Deaktivieren Sie also gegebenenfalls die *Checkbox Control Variable* und wählen statt dessen als Variablentyp *double* aus und tragen als Variablennamen *m_dStartX* ein.

Sinnvolle Werte für die Startvariable rangieren von -2.0 bis 2.0, sodass Sie diese Werte nun bei *min value*, bzw. *max value* angeben.

Erzeugen Sie die Variable mit einem Klick auf die *Finish*-Schaltfläche.

Wiederholen Sie den Vorgang auf Basis der nachstehenden Tabelle und fügen für sämtliche editierbaren Komponenten Variablen ein:

Tabelle 5.3
Variablen für die Steuerelemente des Dialogfelds

ID	Variablentyp	Variablenname	Wertebereich
IDC_XSTART	double	m_dStartX	-2.0 bis 2.0
IDC_YSTART	double	m_dStartY	-2.0 bis 2.0
IDC_XEND	double	m_dEndX	-2.0 bis 2.0
IDC_YEND	double	m_dEndY	-2.0 bis 2.0
IDC_MAXITER	int	m_nMaxIter	1 bis 100
IDC_COLOR	bool	m_bColor	
IDC_STARTRED	int	m_nStartRed	0 bis 255
IDC_STARTGREEN	int	m_nStartG	0 bis 255
IDC_STARTBLUE	int	m_nStartB	0 bis 255
IDC_ENDRED	int	m_nEndR	0 bis 255
IDC_ENDGREEN	int	m_nEndG	0 bis 255
IDC_ENDBLUE	int	m_nEndB	0 bis 255

Durch das Angeben von Wertebereichen für die einzelnen Variablen wird automatisch sichergestellt, dass nur gültige Werte durch den Benutzer eingetragen werden können (Stichwort DDX/DDV).

Einsatz des Dialogfelds

Der Einsatz des neuen Dialogfelds gestaltet sich als denkbar einfach. Öffnen Sie die Datei *ApfelmaennchenView.cpp* und fügen Sie dort die folgende Includezeile ein:

```
#include "ApfelmaennchenEinstellungen.h"
```

Hierdurch wird sichergestellt, dass die Klasse *CApfelmaennchenEinstellungen* bekannt ist und Instanzen von ihr angelegt werden können.

Fügen Sie als Nächstes eine Behandlungsmethode für die WM_LBUTTON-DOWN-Methode in die *CApfelmaennchenView*-Klasse ein und editieren Sie Ihren Rumpf wie folgt:

```
void CApfelmaennchenView::OnLButtonDown(UINT nFlags,
 CPoint point)
{
   // TODO: Add your message handler code here
// and/or call default
```

```
    CApfelmaennchenEinstellungen dlg;
    dlg.DoModal();

    CView::OnLButtonDown(nFlags, point);
}
```

Austesten der aktuellen Version

Kompilieren und testen Sie das neue Programm. Durch einen Linksklick in die Apfelmännchen-Grafik (oder besser: in den Client-Bereich des Hauptfensters) wird der neue Dialog aufgerufen und dargestellt.

Ungültige Benutzereingaben werden abgefangen und moniert.

Allerdings nützen diese Eingaben derzeit noch nichts, da die neuen Werte nirgendwo gespeichert werden – sobald der Gültigkeitsbereich des Dialogs verlassen wird, verfallen auch die Informationen, die möglicherweise eingetragen wurden.

Ablegen von Werten im Dokument

Bislang wurde die Dokumentenklasse der Anwendung noch recht stiefmütterlich behandelt – dieses soll sich nun ändern.

Speichern der für das Apfelmännchen relevanten Werte

In der Klasse sollen von jetzt an die zur Darstellung eines Apfelmännchens notwendigen Werte eingetragen werden. Dazu müssen geeignete Speicherstrukturen sowie Zugriffsmethoden vorhanden sein.

Um das Beispiel nicht unnötig kompliziert zu machen, werden für jede abzulegende Information eine Variable und zwei Zugriffsmethoden bereitgestellt.

Öffnen Sie die Datei *ApfelmaennchenDoc.h* und fügen Sie dort die folgenden Zeilen in die Klassendeklaration von *CApfelmaennchenDoc* ein:

***Listing 5.13* Neue Deklarationen**

```
public:
    void SetStartX(double in_dStartX);
    void SetStartY(double in_dStartY);
    void SetEndX(double in_dEndX);
    void SetEndY(double in_dEndY);
    void SetMaxIter(int in_nMaxIter);
    void SetColored(bool in_bColored);
    void SetStartRed(int in_nColor);
    void SetStartGreen(int in_nColor);
    void SetStartBlue(int in_nColor);
    void SetEndRed(int in_nColor);
    void SetEndGreen(int in_nColor);
    void SetEndBlue(int in_nColor);

    double   GetStartX();
    double   GetStartY();
    double   GetEndX();
```

```
   double   GetEndY();
   int      GetMaxIter();
   bool     IsColored();
   int      GetStartRed();
   int      GetStartGreen();
   int      GetStartBlue();
   int      GetEndRed();
   int      GetEndGreen();
   int      GetEndBlue();

private:
   void InitializeValues();

   double   m_dStartX;
   double   m_dStartY;
   double   m_dEndX;
   double   m_dEndY;
   int      m_nMaxIter;
   int      m_nStartRed;
   int      m_nStartGreen;
   int      m_nStartBlue;
   int      m_nEndRed;
   int      m_nEndGreen;
   int      m_nEndBlue;
   bool     m_bColor;
```

Implementationen für die neuen Methoden

Begeben Sie sich daraufhin in die Datei *ApfelmaennchenDoc.cpp* und ergänzen den Quelltext gemäß den folgenden Zeilen, die jeweils durch kurze erklärende Kommentare aufgelockert sein werden.

```
CApfelmaennchenDoc::CApfelmaennchenDoc()
{
   // TODO: Hier Code für One-Time-Konstruktion
// einfügen
   InitializeValues();
}
```

Variablen-initialisierungen

Der Konstruktor von *CApfelmaennchenDoc* soll in erster Linie die Variablen der Klasse auf sinnvolle Startwerte setzen. Da dieses aber auch immer dann geschehen soll, wenn ein neues Dokument erzeugt wird, macht es Sinn, die Initialisierungen in eine eigene Methode *InitializeValues* auszulagern, sodass sie von mehreren Stellen des Quelltexts aus angesprungen werden können.

Initialisierung von Dokumenten

Im Gegensatz zu dem, was man annehmen könnte, wird bei SDI-Anwendungen nicht für jedes neu erzeugte Dokument ein neues Objekt vom Dokumententyp angelegt. Vielmehr sorgt ein Aufruf von OnNewDocument *dafür, dass die Dokument-Variablen neu initialisiert werden.*

InitializeValues

Die *InitializeValues*-Methode sieht wie folgt aus:

```
void CApfelmaennchenDoc::InitializeValues()
{
   m_dStartX    = -2.0;
   m_dStartY    = -2.0;
   m_dEndX      = 2.0;
   m_dEndY      = 2.0;
   m_nMaxIter   = 10;

   m_nStartRed  = 0;
   m_nStartGreen= 0;
   m_nStartBlue = 0;

   m_nEndRed    = 255;
   m_nEndGreen  = 255;
   m_nEndBlue   = 255;

   m_bColor     = true;
}
```

Farbige vs. schwarzweiße Apfelmännchen

Neben dem Einsetzen der bekannten Standardwerte (nämlich zufälligerweise gerade denen, die auch in der ersten *OnDraw*-Methode Verwendung fanden), wird weiterhin festgelegt, dass farbige Apfelmännchen gezeichnet werden sollen (*m_bColor*) und dabei ein Farbverlauf von Schwarz nach Weiß einzusetzen ist. Das sorgt dafür, dass um die innerste Schicht des Apfelmännchens (gerade die Punkte, an denen der maximale Iterationswert überschritten wird und die daher schwarz gezeichnet werden) ein weißer Rahmen für eine edle Optik sorgt.

OnNewDocument

Das Dokument neu initialisieren

Das Initialisieren gehört, wie schon angesprochen, nicht nur in den Konstruktor, sondern vor allem auch in die *OnNewDocument*-Methode, die beim Erzeugen eines neuen Dokuments verwendet wird:

```
BOOL CApfelmaennchenDoc::OnNewDocument()
{
   if (!CDocument::OnNewDocument())
      return FALSE;

   // TODO: Hier Code zur Reinitialisierung
// einfügen
   // (SDI-Dokumente verwenden dieses Dokument)
   InitializeValues();

   return TRUE;
}
```

Zugriffsmethoden auf die Dokumentvariablen

Die Zugriffsmethoden auf die Dokumentvariablen fallen wegen der niedrigen Komplexität der Apfelmännchen-Anwendung relativ unkompliziert und übersichtlich aus.

Vereinfachter Zugriff auf Dokumentinhalte

Sie bieten jeweils schreibenden beziehungsweise lesenden Zugriff auf die einzelnen Variablen. Das ist nur bedingt schöner Programmierstil, vereinfacht aber das doch schon recht umfangreiche Projekt:

```
void CApfelmaennchenDoc::SetStartX(double in_dStartX)
{
   m_dStartX = in_dStartX;
}

void CApfelmaennchenDoc::SetStartY(double in_dStartY)
{
   m_dStartY = in_dStartY;
}

void CApfelmaennchenDoc::SetEndX(double in_dEndX)
{
   m_dEndX = in_dEndX;
}

void CApfelmaennchenDoc::SetEndY(double in_dEndY)
{
   m_dEndY = in_dEndY;
}

void CApfelmaennchenDoc::SetMaxIter(int in_nMaxIter)
{
   m_nMaxIter = in_nMaxIter;
}
```

```
void CApfelmaennchenDoc::SetColored(bool in_bColored)
{
   m_bColor = in_bColored;
}

void CApfelmaennchenDoc::SetStartRed(int in_nColor)
{
   m_nStartRed = in_nColor;
}

void CApfelmaennchenDoc::SetStartGreen(int in_nColor)
{
   m_nStartGreen = in_nColor;
}

void CApfelmaennchenDoc::SetStartBlue(int in_nColor)
{
   m_nStartBlue = in_nColor;
}

void CApfelmaennchenDoc::SetEndRed(int in_nColor)
{
   m_nEndRed = in_nColor;
}

void CApfelmaennchenDoc::SetEndGreen(int in_nColor)
{
   m_nEndGreen = in_nColor;
}

void CApfelmaennchenDoc::SetEndBlue(int in_nColor)
{
   m_nEndBlue = in_nColor;
}

doubleCApfelmaennchenDoc::GetStartX()
{
   return (m_dStartX);
}

doubleCApfelmaennchenDoc::GetStartY()
{
   return (m_dStartY);
}

doubleCApfelmaennchenDoc::GetEndX()
{
   return (m_dEndX);
}
```

```
double   CApfelmaennchenDoc::GetEndY()
{
   return (m_dEndY);
}

int   CApfelmaennchenDoc::GetMaxIter()
{
   return (m_nMaxIter);
}

bool  CApfelmaennchenDoc::IsColored()
{
   return (m_bColor);
}

int   CApfelmaennchenDoc::GetStartRed()
{
   return (m_nStartRed);
}

int   CApfelmaennchenDoc::GetStartGreen()
{
   return (m_nStartGreen);
}

int   CApfelmaennchenDoc::GetStartBlue()
{
   return (m_nStartBlue);
}

int   CApfelmaennchenDoc::GetEndRed()
{
   return (m_nEndRed);
}

int   CApfelmaennchenDoc::GetEndGreen()
{
   return (m_nEndGreen);
}

int   CApfelmaennchenDoc::GetEndBlue()
{
   return (m_nEndBlue);
}
```

Weitere Überarbeitungen von *CApfelmaennchenEinstellungen*

Das Dokument ist jetzt prinzipiell bereit, die Daten aufzunehmen, die von einem Benutzer in das Optionsdialogfeld eingegeben werden. Allerdings haben wir uns noch nicht um die Behandlung der beiden Farbauswahlknöpfe gekümmert.

Benutzerdefiniertes Auswählen einer Farbe

Diesem Umstand soll nun Rechnung getragen werden. Öffnen Sie die Datei *ApfelmaennchenEinstellungen.h* und Sie fügen dort die folgende Zeile ein:

```
private:
   bool GetColor(COLORREF &out_Color);
```

GetColor wird eine Hilfsfunktion zum Einlesen einer Farbe.

Fügen Sie weiterhin Behandlungsmethoden für die beiden Schaltflächen des Optionsfelds hinzu (durch Doppelklick auf die Schaltfläche und Bestätigen des Namens der zu erzeugenden Methode).

Standarddialogfelder

Öffnen Sie die Datei *ApfelmaennchenEinstellungen.cpp* und ergänzen Sie die Includezeilen zunächst um die nachstehende Zeile:

Listing 5.14 Apfelmaennchen-Einstellungen.cpp

```
#include "afxdlgs.h"
```

Sie ist notwendig, damit Sie auf die so genannten Standarddialoge zugreifen können, die durch die MFC automatisch zur Verfügung gestellt werden.

Zur Verfügung stehende Standarddialoge

Windows bietet eine ganze Reihe solcher Standarddialogfelder. Im Einzelnen sind dies:

- *CColorDialog*: Dialog zur Auswahl einer Farbe.
- *CFileDialog*: Dialog zur Auswahl einer oder mehrerer Dateien, bzw. zum Eingeben eines Dateinamens.
- *CFindReplaceDialog*: Dialog zum Suchen/Ersetzen innerhalb von Dokumenten.
- *CFontDialog*: Dialog zur Auswahl eines Zeichensatzes.
- *CPageSetupDialog*: Dialog zum Einrichten von Seiten.
- *CPrintDialog*: Dialog zum Anpassen von Druckeinstellungen.

Sie finden eine Übersicht über die einzelnen Standarddialoge und ihre Verwendung im Anhang dieses Buches.

Der *CColorDialog* zum Auswählen von Farben

Verwenden eines *CColorDialogs*

Genauer ansehen wollen wir uns hier, aus gegebenem Anlass, den Dialog zur Auswahl einer Farbe. Seine Anwendung entspricht der von selbst definierten Dialogfeldern, wie Sie in der Implementation von *GetColor*, die Sie nun in *ApfelmaennchenEinstellung.cpp* ergänzen, sehen können:

```
bool CApfelmaennchenEinstellungen::GetColor(COLORREF
 &out_Color)
{
   // Farbdialog vorbereiten
   CColorDialog dlg;

   // wurde eine Farbe ausgewählt?
   if (dlg.DoModal() == IDOK)
   {
      // Farbe kopieren
      out_Color = dlg.GetColor();
      return (true);
   }
   else
   {
      return (false);
   }
}
```

Der Dialog wird zunächst als lokales Objekt erzeugt und dann modal dargestellt. Wird er durch einen Druck auf die *OK*-Schaltfläche beendet, kann man über die Membermethode *GetColor* den ausgewählten Farbwert auslesen. Dieser wird in Form eines *COLORREF*-Werts gespeichert, der, wie Sie bereits wissen, die Komponenten einer Farbe in Form von Rot-, Grün- und Blauanteil enthält.

Makros zum Auslesen von Farbinformationen

Auslesen der Farbanteile

Bislang haben Sie nur selbst *COLORREF*-Variablen mit Werten gefüllt, indem das RGB-Makro eingesetzt wurde. Zum Auslesen der einzelnen Farbkomponenten gibt es nun die drei Makros *GetRValue*, *GetGValue* sowie *GetBValue*, deren Einsatz in den nachstehenden Behandlungsmethoden für die beiden Dialogschaltflächen des Optionsdialogfelds aufgezeigt wird:

```
void
CApfelmaennchenEinstellungen::OnBnClickedButton1()
{
   // Variable deklarieren
   COLORREF Color;

   // Farbwert ermitteln
   if (GetColor(Color))
   {
```

```
      // Farbwert kopieren
      m_nStartRed = GetRValue(Color);
      m_nStartG = GetGValue(Color);
      m_nStartB = GetBValue(Color);

      // Farbwerte aktualisieren
      UpdateData(false);
   }
}

void
CApfelmaennchenEinstellungen::OnBnClickedButton2()
{
   // Variable deklarieren
   COLORREF Color;

   // Farbwert ermitteln
   if (GetColor(Color))
   {
      // Farbwert kopieren
      m_nEndR = GetRValue(Color);
      m_nEndG = GetGValue(Color);
      m_nEndB = GetBValue(Color);

      // Farbwerte aktualisieren
      UpdateData(false);
   }
```

Durch einen Aufruf der selbst definierten Funktion *GetColor* wird der vom Benutzer ausgesuchte Farbwert an die *COLORREF*-Variable *Color* übertragen. Die einzelnen Farbanteile werden dann in die zugehörigen Variablen mithilfe der *GetXValue*-Makros übertragen.

Aktualisieren von Dialogfeldinhalten

Der abschließende Aufruf von *UpdateData* überträgt die neuen Variablenwerte in die Steuerelemente, da hier sonst noch die alten Daten angezeigt würden.

Testen Sie das Programm nun erneut aus und öffnen Sie den Farbdialog durch Anwahl einer der beiden Schaltflächen. Es öffnet sich das von anderen Windows-Programmen her bekannte Feld:

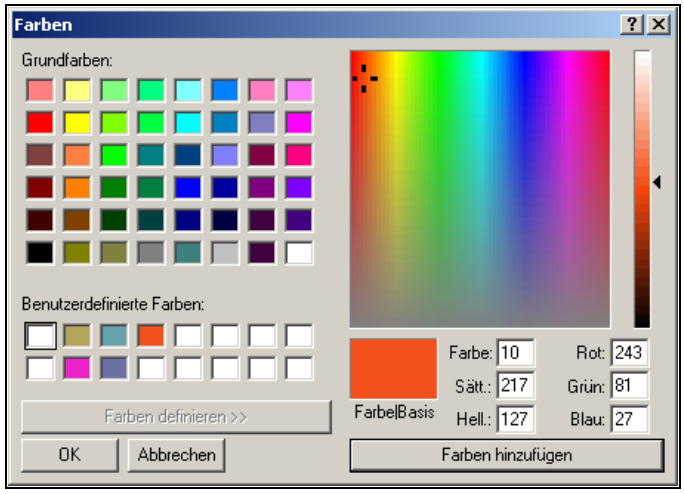


Abb. 5.13
Der Farbdialog

Verwenden der eingestellten Daten

Was noch fehlt, ist der tatsächliche Einsatz der im Dokument gespeicherten Informationen. Um diese einzusetzen, muss die *OnDraw*-Methode der *CApfelmaennchenView*-Klasse erneut umgeschrieben werden:

Passende Erweiterung der Ansichtsklasse um die Verwendung der gespeicherten Daten

Listing 5.15
Farbige Apfelmännchen

```
void CApfelmaennchenView::OnDraw(CDC* pDC)
{
   CApfelmaennchenDoc* pDoc = GetDocument();
   ASSERT_VALID(pDoc);

   // Größe der Ausgabefläche festlegen
   CRect ClientRect;
   GetClientRect(ClientRect);
   int nAufloesung = ClientRect.Width() >
ClientRect.Height() ?
ClientRect.Width() :
ClientRect.Height();

   // Anzahl der Gitterpunkte bestimmen
   int nGitterpunkte = nAufloesung + 1;

   // maximale Iterationstiefe bestimmen
   int nMaxIter = pDoc->GetMaxIter();

   // Startposition für die Ausgabe festlegen
   POINT Position;
   Position.x = 0;
   Position.y = 0;
```

```
   // Bereich, in dem das Apfelmännchen berechnet
// werden soll
   double dXStart, dYStart, dXEnd, dYEnd;
   dXStart = pDoc->GetStartX();
   dYStart = pDoc->GetStartY();
   dXEnd   = pDoc->GetEndX();
   dYEnd   = pDoc->GetEndY();

   // Farbverlauf vorberechnen
   int nRed = (pDoc->GetStartRed() -
pDoc->GetEndRed()) / pDoc->GetMaxIter();
   int nGreen = (pDoc->GetStartGreen() -
pDoc->GetEndGreen()) / pDoc->GetMaxIter();
   int nBlue = (pDoc->GetStartBlue() -
pDoc->GetEndBlue()) / pDoc->GetMaxIter();

   // momentane Position in der Mandelbrotmenge
   double dX, dY;
   dX = dXStart;
   dY = dYStart;

   // Schrittweite in der Mandelbrotmenge
   double dSX, dSY;
   dSX = (dXEnd – dXStart) / nAufloesung;
   dSY = (dYEnd – dYStart) / nAufloesung;

   // alle Zeilen berechnen
   while (Position.y < nGitterpunkte)
   {
      // neue Zeile, x-Wert zurücksetzen
      Position.x = 0;
      dX = dXStart;

      // jeden Pixel der Zeile berechnen
      while (Position.x < nGitterpunkte)
      {
         // Iterationswert bestimmen
         int nIterations =
 GetNumberOfIterations(dX, dY, nMaxIter);

         // schwarzen Pixel zeichnen, wenn
// maximale Iterationstiefe
// erreicht oder
         // überschritten
         if (nIterations >= nMaxIter)
         {
            pDC->SetPixel(Position,
```

```
 RGB(0,0,0));
         }
         else
         {
            // befinden wir uns im
// Farbmodus?
            if ((pDoc->IsColored()) &&
(nIterations > 0))
            {
               // Farbwerte berechnen
               int nR, nG, nB;

               nR = pDoc
->GetStartRed()
- (nIterations * nRed);

nG = pDoc
->GetStartGreen()
-(nIterations *
nGreen);

nB = pDoc
->GetStartBlue()
-(nIterations * nBlue);

               // Farbwerte im
// gültigen Bereich
// halten
// Sicherheits-
// vorkehrung
               if (nR < 0) nR = 0;
               if (nR > 255) nR = 255;

               if (nG < 0) nG = 0;
               if (nG > 255) nG = 255;

               if (nB < 0) nB = 0;
               if (nB > 255) nB = 255;

               // farbigen Pixel
// setzen
               pDC->SetPixel(Position,
RGB(nR, nG, nB));
            }
         }

         // zum nächsten Bildpunkt
// fortschreiten
```

```
            Position.x++;
            dX += dSX;
        }

        // nächste Zeile
        Position.y++;
        dY += dSY;
    }
}
```

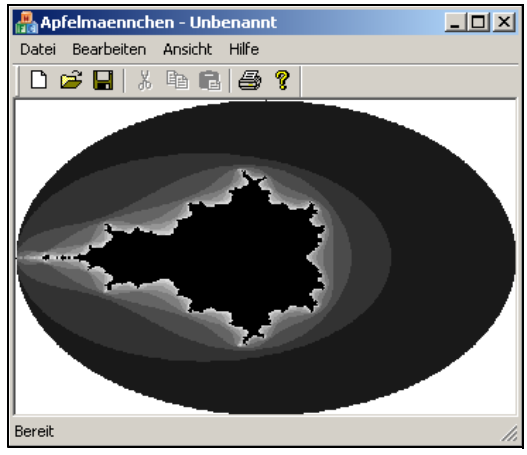
Weitere Änderungen in *OnDraw*

Hier wurde im Wesentlichen der Zugriff auf die Dokumentdaten ergänzt. Weiterhin entscheidet die Methode darüber, ob farbige oder schwarz-weiße Apfelmännchen gezeichnet werden soll.

Berechnung des Farbverlaufs

Der Farbverlauf wird durch einfaches Verrechnen der Start- und Endfarben ermittelt, wodurch sich ein linearer Farbverlauf zwischen diesen beiden Werten ergibt. Starten Sie das Programm, erhalten Sie nun die folgende Ausgabe:

Abb. 5.14
Zwischenstand



Testen Sie nun einmal die Auswirkungen von anderen Einstellungen innerhalb des Optionsfelds aus. Verändern Sie den angezeigten Bildausschnitt, den Farbverlauf, oder die Grundeinstellung, ob farbige oder schwarz-weiße Apfelmännchen zu zeichnen sind. Probieren Sie auch höhere und niedrigere Werte für die maximale Iterationstiefe aus.

Auf den letzten Seiten dieses Kapitels wird es nun noch um einige abrundende Veränderungen und Erweiterungen gehen.

Ergänzen der Statusleisteninformationen

Eine beliebte Stelle, um wichtige Informationen innerhalb eines Programms schnell verfügbar zu machen, ist die *Statuszeile*, die sich für gewöhnlich am unteren Rand eines Fensters befindet.

Definition der Statuszeile

Sie ist selbstverständlich auch innerhalb unseres Programms definiert und vielleicht erinnern Sie sich an die Deklaration des *indicators* Arrays, die hier zur Vereinfachung noch einmal wiedergegeben ist:

```
static UINT indicators[] =
{
   ID_SEPARATOR,
   ID_INDICATOR_CAPS,
   ID_INDICATOR_NUM,
   ID_INDICATOR_SCRL,

};
```

Die IDs, die hier angegeben werden, beschreiben die Inhalte, die sich in der Statuszeile befinden sollen. Schauen Sie sich die bisherige Statuszeile einmal an – Sie werden sofort erkennen, welche IDs welchen Komponenten zugeordnet sind:

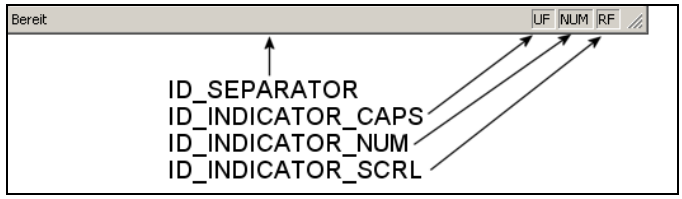


Abb. 5.15 **Elemente der Statuszeile**

Hinzufügen neuer Informationen in die Statuszeile

Offensichtlich ist es also ein erster wichtiger Schritt, dieses Array um geeignete IDs zu ergänzen, die Platzhalter für die neu einzufügenden Informationen sind.

Welche Angaben wären hier interessant? Der Benutzer kann einen Bereich des Apfelmännchens auswählen und darstellen lassen. Dieses geht durch Eingabe passender Werte in das *Options*-Dialogfeld.

Da wäre es doch nett, wenn bei einer Mausbewegung über dem Apfelmännchen die aktuelle Koordinate, über der sich der Mauszeiger befindet, angezeigt würde. So könnte man bequem interessante Stelle anfahren, die Werte notieren und in das Optionsfeld eintragen.

Wenn wir schon dabei sind, ergänzen wir die Statuszeile gleichzeitig noch um die tatsächliche aktuelle Mauszeigerposition, relativ zur linken oberen Ecke, gerechnet in Pixeln.

Ergänzen des *indicator*-Arrays

Die IDs, die in das Array eingefügt werden sollen, können frei gewählt werden. Ergänzen Sie das Array wie folgt:

Listing 5.16 **Neue Einträge für die Statusleiste**

```
static UINT indicators[] =
{
   ID_SEPARATOR,           // Statusleistenanzeige
   ID_INDICATOR_XPOSCLIENT,
   ID_INDICATOR_YPOSCLIENT,
   ID_INDICATOR_XPOSAPFEL,
   ID_INDICATOR_YPOSAPFEL,
   ID_INDICATOR_CAPS,
   ID_INDICATOR_NUM,
   ID_INDICATOR_SCRL,
};
```

ID_INDICATOR_XPOSCLIENT und *ID_INDICATOR_YPOSCLIENT* sollen die aktuelle Pixelposition relativ zur linken oberen Ecke des Client-Bereichs ausgeben, *ID_INDICATOR_XPOSAPFEL* und *ID_INDICATOR_YPOSAPFEL* die umgerechnete Position im Koordinaten system.

Die neuen Informationen werden dabei hinter dem Separator aber vor den Standardausgaben angezeigt.

Doch woher weiß das Programm nun, wie diese Werte aussehen sollen?

Die Antwort ist so einfach wie erwartet: überhaupt nicht. Der Programmierer muss selbstständig für eine Aktualisierung der Werte sorgen. Dazu dienen die IDs, denen ein geeigneter Anfangswert zugewiesen wird und über die später die gewünschten Informationen in die Statuszeile eingetragen werden können.

Stringtabellen

Eintragen von Startwerten in die neuen Felder der Statuszeile

Bei der Arbeit mit Statuszeilen und ihren Anzeigebereichen ist es wichtig zu wissen, dass die Größe der einzelnen Felder (genauer: ihre Breite) bereits bei der Initialisierung festgelegt wird.

Vereinfacht gesprochen nimmt man einen Starttext, der ausgegeben werden soll, übergibt ihn an die Statusleiste und lässt sie dann selbstständig berechnen, wie groß das zugehörige Feld sein muss, damit der komplette Text aufgenommen werden kann.

Es ergibt sich von selbst, dass es zweckmäßig ist, hier bereits einen Starttext anzugeben, der die maximal auftretende Textlänge impliziert und somit direkt während der Initialisierung für ein ausreichend großes Statusleistenelement sorgt.

Dieser Text wird in der Stringtabelle der Anwendung eingetragen, die zu den Ressourcen unserer Applikation gehört. Öffnen Sie die Ressourcenansicht und wählen Sie den *String Table*-Ordner aus. Dort klicken Sie auf die einzige vorhandene *String Table*:

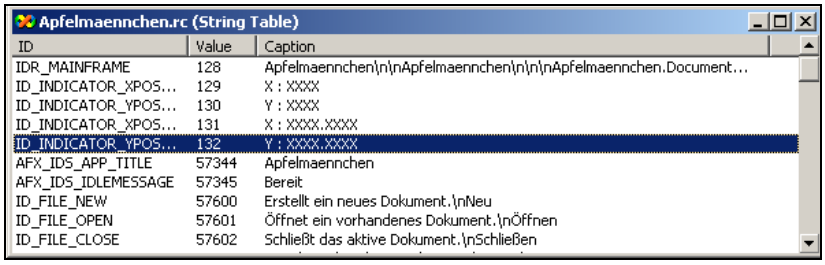


Abb. 5.16
Die Zeichenketten-Ressource

Sie finden hier bereits Texte für die drei Standardstatuszeilen-Informationen, die sich hinter den IDs *ID_INDICATOR_CAPS*, *ID_INDICATOR_NUM* und *ID_INDICATOR_SCRL* verbergen.

Eintragen neuer Strings

Am unteren Ende der Liste ist ein Platzhalter für neue IDs zu finden. Klicken Sie in dieses Feld und tragen Sie die nachstehenden Werte ein:

ID	Caption
ID_INDICATOR_XPOSCLIENT	X : XXXX
ID_INDICATOR_YPOSCLIENT	Y : XXXX
ID_INDICATOR_XPOSAPFEL	X : XXXX.XXXX
ID_INDICATOR_YPOSAPFEL	Y : XXXX.XXXX

IDs für die neuen Strings

Die automatisch erscheinenden Eintragungen im *Value*-Feld verändern Sie nicht – hier wird automatisch eine noch nicht vergebene ID ausgewählt und eingesetzt.

Kompilieren und starten Sie das Programm, Sie haben nun die folgende Statuszeile vorliegen:

Abb. 5.17
Die vorläufige Statuszeile

Wie kommt es zu den hier gewählten Starttexten? Wie eingangs erwähnt wurde, wird die Breite eines Statuszeilenelements durch den initial eingetragenen Text bestimmt.

Dieses ist gerade der Inhalt, der in der String Table unter der spezifizierten ID gefunden werden kann.

Gründe für die gewählten Startwerte

Wenn Sie mit einem nicht-größenfixierten Zeichensatz arbeiten (bei dem alle Zeichen dieselbe Größe belegen), werden Sie bemerkt haben, dass sämtliche Ziffern, die als Angabe für die aktuelle Mauszeigerposition in Frage kommen, unterschiedlich breit sind.

Hätten Sie als Starttext zum Beispiel *1111* angegeben, wäre die Breite des Felds bei einer Ausgabe von *4444* beispielsweise gesprengt worden. Man verwendet daher den Trick, statt einer konkreten Ziffernfolge so viele X-Zeichen anzugeben, wie Ziffern zu erwarten sind. Das X-Zeichen ist nämlich breiter als jede mögliche Ziffer und sorgt somit für ein ausreichend breites Ausgabefeld.

Weitere Einsatzgebiete für String Tables

Lokalisierung

String Tables werden nicht nur verwendet, um Statuszeilen-Elemente zu definieren, sondern allgemein immer, wenn Texte an irgendwelchen Stellen oder in irgendwelchen Situationen – beispielsweise als Reaktion auf eine Benutzereingabe – ausgegeben werden sollen.

Der Vorteil liegt auf der Hand: Die ausgegebenen Texte sind nicht über den kompletten Quelltext verstreut, sondern an einer zentralen Stelle zusammengefasst.

Das spart zum einen Speicherplatz (manche Texte müssen gegebenenfalls an zahlreichen Stellen ausgegeben werden), macht es zum anderen aber auch leichter, Änderungen an den Strings vorzunehmen (beispielsweise um eine Lokalisierung in eine andere Sprache durchzuführen).

Machen Sie viel Gebrauch von String Tables, ihre Programme werden es Ihnen durch bessere Wartbarkeit und Übersichtlichkeit danken.

Verändern der angezeigten Werte in der Statuszeile

Nachdem nun die gewünschten Felder in die Statusleiste eingefügt wurden, müssen noch sinnvolle Werte in diese eingetragen werden. Das sollte immer dann geschehen, wenn sich die Position des Mauszeigers verändert, was wiederum in einer Behandlungsmethode auf die *WM_MOUSEMOVE*-Nachricht innerhalb von *CApfelmaennchenView* geschehen sollte.

Zugriff auf die Statusleiste

Sie benötigen also innerhalb dieser Klasse Zugriff auf die Statuszeile. Da bietet es sich an, einen Zeiger auf die Statuszeile direkt beim Programmstart (beziehungsweise beim Initialisieren der Ansichtsklasse) zu speichern.

Fügen Sie hierzu die folgende Zeile in die Klassendeklaration von *CApfelmaennchenView* in der Datei *ApfelmaennchenView.h* ein:

```
private:
   CStatusBar *m_pStatusBar;
```

CStatusBar ist gerade die Klasse, die eine Statusleiste beschreibt. Um einen Zeiger auf die Statuszeile des Hauptfensters zu erhalten, muss das Hauptfenster natürlich zunächst einmal existieren.

Der vielleicht bei Ihnen auftauchende Ausgangsgedanke, die Initialisierung von *m_pStatusBar* in den Konstruktor von *CApfelmaennchenView* zu verfrachten, scheitert an genau dieser Problematik: Während des Konstruktoraufrufs von *CApfelmaennchenView* existiert das fertige Hauptfenster mit seiner Statusleiste noch nicht.

Konstruktoren und Initialisierungen

Überhaupt ist es aus diesem Grund keine gute Idee, Initialisierungen, die das Vorhandensein von anderen Fensterkomponenten voraussetzen, in Konstruktoren zu legen.

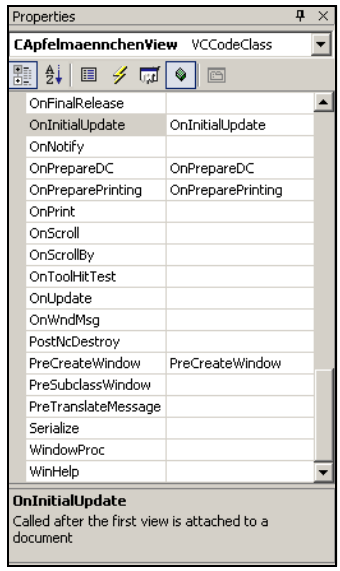
Die MFC bietet für die meisten Klassen allerdings bereits Initialisierungsmethoden an, die in den Basisklassen definiert und in Ihren abgeleiteten Klassen überschrieben werden können) – ein typischer Name für diese Methoden ist OnInitialUpdate.

Zugriff ermöglichen

Wo fügt man diese Initialisierungen dann ein? Wie dem Kasten zu entnehmen ist, existiert für eine Ansichtsklasse eine Methode namens *OnInitialUpdate*, die gerade zu diesem Zweck dient: Initialisierungen durchzuführen, die darauf vertrauen, dass sie umschließende Konstrukte – wie eben das Hauptfenster und seine Statusleiste – bereits existieren.

Öffnen Sie die Klassenansicht für *CApfelmaennchenView* und wählen Sie die überschreibbaren Methoden im Eigenschaftsfenster aus. Suchen sie hier die Methode *OnInitialUpdate* und fügen eine überschreibende Methode in die Klasse ein:

Abb. 5.18 **Hinzufügen der *OnInitialUpdate*-Methode**



Zugriff auf die Statusleiste erhalten

Speichern des Zeigers auf die Statuszeile

Um in *OnInitialUpdate* nun auf das Hauptfenster zugreifen zu können, das die Statusleiste enthält, muss zunächst eine noch fehlende Include-Zeile in die Datei *ApfelmaennchenView.cpp* eingefügt werden:

```
#include "MainFrm.h"
```

Nun können Sie *OnInitialUpdate* wie folgt implementieren:

Listing 5.17 ***OnInitialUpdate*-Implementation**

```
void CApfelmaennchenView::OnInitialUpdate()
{
   CView::OnInitialUpdate();

   // TODO: Add your specialized code here and/or
// call the base class
   CMainFrame *pFrame= (CMainFrame *) AfxGetApp()
->m_pMainWnd;
   m_pStatusBar= (CStatusBar *) & (pFrame
->m_wndStatusBar);
}
```

Die Methode beschafft sich zunächst durch einen Aufruf von *AfxGetApp*-Zugriff auf das Applikationsobjekt, das als Membervariable einen Zeiger auf das Hauptfenster der Anwendung enthält.

Dieses Hauptfenster ist bei einer durch den Assistenten erzeugten Anwendung immer vom Typ *CMainFrame* und enthält seinerseits ein Objekt vom Typ *CSta-*

tusBar namens *m_wndStatusBar*, das gerade die Statusleiste beschreibt, auf die Sie Zugriff benötigen.

OnInitialUpdate speichert nun einen Zeiger auf dieses Objekt in der zuvor definierten *CStatusBar* Variablen *m_pStatusBar* und erlaubt so auch später den Zugriff auf diese für uns so wichtige Zeile.

Reaktion auf Mausbewegungen

Nachdem Sie den Zugriff auf die Statuszeile erlangt haben, müssen Sie noch eine Entscheidung treffen, wann deren Informationen zu verändern sind.

Aktualisieren der angezeigten Werte

Grundlegend wurde die Wahl bereits auf eine die *WM_MOUSEMOVE*-Nachricht behandelnde Funktion festgelegt.

Fügen Sie also eine entsprechende Nachrichtenmethode in die Klasse *CApfelmaennchenView* ein und füllen ihren Funktionsrumpf entsprechend der nachstehenden Zeilen:

***Listing 5.18* Anzeige der Mausposition**

```
void CApfelmaennchenView::OnMouseMove(UINT nFlags, CPoint point)
{
   // Ausgabevariable vorbereiten
   CString strText;
   int nIndex;

   // Zugriff auf Statuszeilenobjekte
   nIndex = m_pStatusBar
->CommandToIndex(ID_INDICATOR_XPOSCLIENT);

   // Text vorbereiten
   strText.Format("X : %4d", point.x);

   // Text in Statuszeile schreiben
   m_pStatusBar->SetPaneText(nIndex, strText);

   // y-Koordinate editieren
   nIndex = m_pStatusBar
->CommandToIndex(ID_INDICATOR_YPOSCLIENT);
   strText.Format("Y : %4d", point.y);
   m_pStatusBar->SetPaneText(nIndex, strText);

   // Zugriff auf Dokument erlangen
   CApfelmaennchenDoc *pDoc = GetDocument();
   ASSERT_VALID(pDoc);

   // Größe der ClientArea bestimmen
   CRect ClientRect;
   GetClientRect(ClientRect);
```

```
   // Breite und Höhe auslesen
   int nClientWidth= ClientRect.Width();
   int nClientHeight= ClientRect.Height();

   // Breite und Höhe der angezeigten Auswahl
// bestimmen
   double dSelectionWidth= pDoc->GetEndX() -
 pDoc->GetStartX();
   double dSelectionHeight = pDoc->GetEndY() -
 pDoc->GetStartY();

   // Breite und Höhe eines Pixels (übertragen in's
// Mandelbrotsystem)
   // berechnen
   double dPixelWidth= dSelectionWidth /
nClientWidth;
   double dPixelHeight = dSelectionHeight /
nClientHeight;

   // Tatsächliche Position des Mauszeigers in der
// Mandelbrotmenge bestimmen
   double dApfelX, dApfelY;
   dApfelX = point.x * dPixelWidth + pDoc
->GetStartX();
   dApfelY = point.y * dPixelHeight + pDoc
->GetStartY();

   // Texte in Statusleiste schreiben
   nIndex = m_pStatusBar
->CommandToIndex(ID_INDICATOR_XPOSAPFEL);
   strText.Format("X : %4.4f", dApfelX);
   m_pStatusBar->SetPaneText(nIndex, strText);

   nIndex = m_pStatusBar
->CommandToIndex(ID_INDICATOR_YPOSAPFEL);
   strText.Format("Y : %4.4f", dApfelY);
   m_pStatusBar->SetPaneText(nIndex, strText);

   // Standardbehandlung aufrufen
   CView::OnMouseMove(nFlags, point);
}
```

Erläuterung zu *OnMouseMove*

Obwohl in der *OnMouseMove*-Methode einige Rechnungen durchgeführt werden, handelt es sich im Großen und Ganzen doch um eine sehr geradlinig arbeitende Funktion.

Vorgehen beim Aktualisieren der Statuszeile

Die vier Statuszeilenelemente, die innerhalb dieser Methode angepasst werden müssen, werden jeweils nach einem einfachen Schema editiert:

- Index des Elements in der Statuszeile bestimmen
- Einzutragenden Wert ermitteln
- Wert formatieren
- Wert in Statuszeile eintragen

Der Index der Statuszeilenelemente muss bestimmt werden, da die texteintragende Methode *SetPaneText* den bei 0 beginnenden Index des Elements übergeben bekommt, dessen Inhalt verändert werden soll.

Die Funktion *CommandToIndex* übernimmt gerade die ID des gewünschten Elements und gibt diesen Index zurück.

Das Formatieren der Ausgabestrings wird mithilfe von *CString*-Objekten realisiert, die bereits im letzten Kapitel angeschnitten wurden. Die Format-Methode funktioniert ähnlich der sicherlich bekannten *printf*-Funktionsfamilie – bestimmte Formatierungssymbole wie *%s* oder *%i* dienen hier als Platzhalter für Daten aus Variablen und sorgen für einen formatierten Ausgabestring.

Umrechnen der Mauszeigerposition

Während die Pixelpositionen direkt aus dem Eingabeparameter *point* ausgelesen werden können, muss die Position innerhalb der Mandelbrot-Menge berechnet werden. Hier kommen die gleichen Formeln zum Einsatz wie bereits in *OnDraw*, sodass nicht weiter auf die Kalkulationen eingegangen werden soll.

Das Ergebnis der Bemühungen sehen Sie, wenn Sie das Programm kompilieren, starten und dann die Maus über der Ausgabefläche hin- und herbewegen:

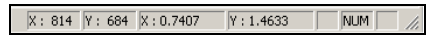


Abb. 5.19
Beispielwerte

Auswahl eines Mandelbrot-Ausschnitts mit der Maus

Jetzt, wo die aktuellen Mauspositionen angezeigt werden, wäre es doch auch nett, wenn man mithilfe eines so genannten *Gummibands* einen Bereich markieren könnte, der dann vergrößert dargestellt wird.

Erweiterte Einflussnahme durch den Benutzer

Gummibänder kennt man unter Windows vor allem aus den zahlreichen Zeichenprogrammen. Sie arbeiten prinzipiell so, dass der Benutzer die Maustaste an einer beliebigen Stelle drückt und gedrückt hält und dann durch weitere Mausbewegungen ein gestricheltes Rechteck aufspannen kann. Lässt er die Maustaste wieder los, wird dieses Rechteck als Markierung aufgefasst und kann weiter bearbeitet werden.

Im Rahmen des Apfelmännchen-Projekts soll der Benutzer die Möglichkeit haben, durch eben diese Gummibandtechnik einen Bereich zu definieren, der dann vergrößert dargestellt wird – er wird also als neuer Ansichtsbereich angenommen, neu berechnet und gezeichnet.

Damit dieses funktionieren kann, muss allerdings zunächst die linke Maustaste wieder „funktionslos" gemacht werden – derzeit löst sie ja noch die Anzeige des Optionsdialogs aus.

Der übliche Weg, Dialoge anzuzeigen, ist entweder einen Toolbar-Button oder einen Menüpunkt für diese Aktion einzurichten. An dieser Stelle sollen beide Möglichkeiten beschrieben werden.

Ein neuer Toolbar-Button

Das Erzeugen eines neuen *Toolbar*-Buttons ist denkbar einfach: Öffnen Sie die Ressourcenansicht des Projekts und suchen Sie dort den Ordner *Toolbar*. Doppelklicken Sie auf das einzige Element *IDR_MAINFRAME*.

Editieren der Werkzeugleiste

Sie finden sich im *Toolbar*-Editor wieder, der neben der kompletten derzeit vorhandenen Werkzeugleiste zwei weitere Fenster mit dem aktiven *Toolbar*-Button und einer vergrößerten Ansicht desselben enthält.

Am rechten Rand der Werkzeugleiste finden Sie ein noch nicht belegtes Feld. Klicken Sie darauf, wird es als neuer Button zur Toolbar hinzugefügt, direkt daneben erscheint ein neuer Platzhalter für weitere zusätzliche Buttons.

Benutzen Sie nun die Zeichenwerkzeuge des Toolbar-Editors, um ein Ihnen genehmes Aussehen des neuen Buttons zur Anzeige des Toolbar-Editors zu zeichnen. Das fertige Ergebnis könnte so aussehen wie im nachstehenden Screenshot:

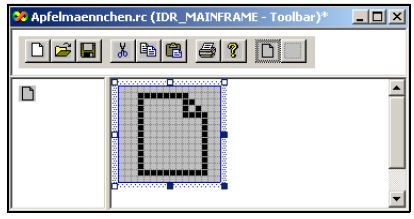


Abb. 5.20 **Die neue Ressource erhält die ID *ID_OPTIONEN_-EINSTELLUNGEN***

Geben Sie dem neuen Button im *Eigenschaften*-Fenster die ID *ID_SETTINGS*.

Ein neues Menü

Bevor wir uns um die Behandlung dieses Knopfes kümmern, soll zunächst noch ein passendes Menü erzeugt werden, dass die gleiche Aufgabe übernimmt – die Darstellung des Optionsdialogfelds.

Neues Menü erzeugen und mit Einträgen füllen

Öffnen Sie dazu in der Ressourcenansicht das Menü *IDR_MAINFRAME* aus dem *Menu*-Ordner. Hier finden Sie, ähnlich dem *Toolbar Editor*, das derzeitige Hauptfenstermenü sowie einen Platzhalter auf der rechten Seite, in den Sie eine neue Menübeschriftung eintippen können – tragen Sie hier *Optionen* ein.

Als Unterpunkt enthält dieses neue Menü dann ebenfalls einen Platzhalter, in den Sie Einstellungen eintippen. Das gesamte Menü können Sie durch Drag & Drop an eine andere Position schieben, zum Beispiel links neben das Hilfemenü. Es ergibt sich das folgende Bild:

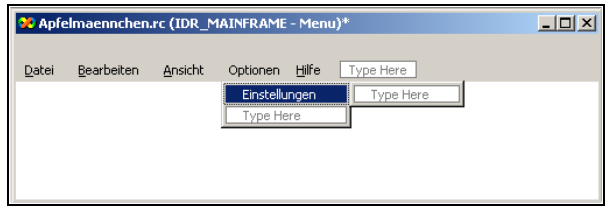


Abb. 5.21
Das neue Menü

Im *Eigenschaften*-Fenster tragen Sie für den Untermenüpunkt als ID ebenfalls *ID_OPTIONEN_EINSTELLUNGEN* ein und tippen bei *Prompt* die Zeile *„Einstellungen für das Apfelmännchen Programm“* ein. Dieser Text erscheint in der Statuszeile, sobald sich der Mauszeiger über dem betreffenden Punkt befindet.

Die übrigen Einstellungen bleiben unverändert, es sei aber angeraten, dass Sie sich kurz die Zeit nehmen, die Möglichkeiten anzutesten, um einen Überblick über die zur Verfügung stehenden Optionen zu haben.

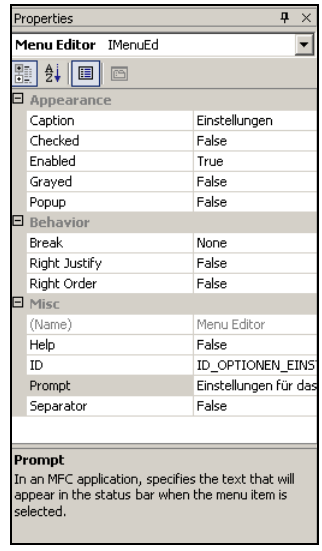


Abb. 5.22
Menüeigenschaften

Behandlung von *ID_OPTIONEN_EINSTELLUNGEN*

Öffnen Sie die Klassenansicht und wählen Sie die Klasse *CApfelmaennchenView* aus. Fügen Sie dort unter *Events* eine *COMMAND*-Behandlungsmethode für *ID_OPTIONEN_EINSTELLUNGEN* hinzu.

Durchreichen der Fensternachrichten

Dieses mag Ihnen etwas seltsam vorkommen, denn sowohl Toolbar als auch Menü gehören ja zum Hauptfenster, nicht zur Ansichtsklasse. Die Ansichtsklasse ist allerdings ein Kindfenster des Hauptfensters und kann somit auch eine behandelnde Funktion für die Nachrichten annehmen.

Das macht umso mehr Sinn, als dass wir, wie Sie gleich sehen werden, einige Operationen bei der Nachrichtenbearbeitung durchführen, die Daten der Ansichtsklasse benötigen.

Entschärfen der *OnLButtonDown*-Methode

Doch gehen wir Schritt für Schritt vor: Zunächst muss die *OnLButtonDown*-Methode von ihrer bisherigen Aufgabe befreit werden. Schneiden Sie dazu ihren Inhalt aus (möglichst über eine *Bearbeiten* > *Ausschneiden* Operation), sodass nur der folgende Rumpf erhalten bleibt:

```
void CApfelmaennchenView::OnLButtonDown(UINT nFlags,
 CPoint point)
{
   // Standardbehandlung aufrufen
   CView::OnLButtonDown(nFlags, point);
}
```

Aktivieren von *OnOptionenEinstellungen*

Fügen Sie dann den, hoffentlich noch vorhandenen, Inhalt der Zwischenablage (gerade den alten Inhalt von *OnLButtonDown*) in die Methode *OnOptionenEinstellungen* ein:

***Listing 5.19* Behandlung des neuen Menüeintrags**

```
void CApfelmaennchenView::OnOptionenEinstellungen()
{
   // Dialoginstanz anlegen
   CApfelmaennchenEinstellungen dlg;

   // Werte in Dialog kopieren
   CApfelmaennchenDoc* pDoc = GetDocument();

   dlg.m_dStartX        = pDoc->GetStartX();
   dlg.m_dStartY        = pDoc->GetStartY();
   dlg.m_dEndX          = pDoc->GetEndX();
   dlg.m_dEndY          = pDoc->GetEndY();
   dlg.m_nMaxIter       = pDoc->GetMaxIter();
   dlg.m_bColor         = pDoc->IsColored();
   dlg.m_nStartRed      = pDoc->GetStartRed();
   dlg.m_nStartG        = pDoc
```

```
->GetStartGreen();
   dlg.m_nStartB        = pDoc->GetStartBlue();
   dlg.m_nEndR          = pDoc->GetEndRed();
   dlg.m_nEndG          = pDoc->GetEndGreen();
   dlg.m_nEndB          = pDoc->GetEndBlue();


// Dialog darstellen und bei Beenden über den OK
// Button Daten
   // kopieren
   if (IDOK == dlg.DoModal())
   {
      pDoc->SetStartX(dlg.m_dStartX);
      pDoc->SetStartY(dlg.m_dStartY);
      pDoc->SetEndX(dlg.m_dEndX);
      pDoc->SetEndY(dlg.m_dEndY);
      pDoc->SetMaxIter(dlg.m_nMaxIter);
      pDoc->SetColored(dlg.m_bColor != 0);
      pDoc->SetStartRed(dlg.m_nStartRed);
      pDoc->SetStartGreen(dlg.m_nStartG);
      pDoc->SetStartBlue(dlg.m_nStartB);
      pDoc->SetEndRed(dlg.m_nEndR);
      pDoc->SetEndGreen(dlg.m_nEndG);
      pDoc->SetEndBlue(dlg.m_nEndB);

      // Grafik aktualisieren
      InvalidateRect(NULL);
   }
}
```

Testen Sie das Programm aus: Das Optionsfeld ist nun über das neue Menü und den neuen *Toolbar*-Knopf zu erreichen, die linke Maustaste hat keine Funktion mehr.

Implementieren der Gummibandauswahl

CRectTracker zur Realisierung einer Gummibandauswahl

Das Einfügen einer Gummibandauswahl ist einfacher, als man es auf den ersten Blick annehmen sollte. Praktischerweise gibt es unter den MFC nämlich bereits eine Klasse *CRectTracker*, die zwar ursprünglich zur Auswahl von OLE-Elementen gedacht ist, aber für unsere Zwecke ebenso hervorragende Dienste leisten kann.

Am besten erklärt man den Einsatz direkt an einem Beispiel, daher sofort ohne Umschweife die neue *OnLButtonDown*-Methode für das Apfelmännchen-Projekt:

Listing 5.20
Gummibandauswahl

```
void CApfelmaennchenView::OnLButtonDown(UINT nFlags, CPoint point)
{
   // Standardbehandlung aufrufen
   CView::OnLButtonDown(nFlags, point);

   // Gummiband initialisieren
   m_Tracker.TrackRubberBand(this, point, true);

   // Bereich ermitteln
   RECT Rect;
   m_Tracker.GetTrueRect(&Rect);

   // Berechnen des ausgewählten Ausschnitts
   // Größe der ClientArea bestimmen
   CRect ClientRect;
   GetClientRect(ClientRect);

   // Breite und Höhe auslesen
   int nClientWidth= ClientRect.Width();
   int nClientHeight= ClientRect.Height();

   // Breite und Höhe der angezeigten Auswahl
// bestimmen
   CApfelmaennchenDoc *pDoc = GetDocument();
   double dSelectionWidth= pDoc->GetEndX()
- pDoc->GetStartX();
   double dSelectionHeight = pDoc->GetEndY()
- pDoc->GetStartY();

   // Breite und Höhe eines Pixels (übertragen in's
// Mandelbrotsystem)
   // berechnen
   double dPixelWidth= dSelectionWidth /
nClientWidth;
   double dPixelHeight = dSelectionHeight /
nClientHeight;

   // Tatsächliche Position des Mauszeigers in der
// Mandelbrotmenge bestimmen
   double dApfelXStart, dApfelYStart, dApfelXEnd,
dApfelYEnd;
   dApfelXStart = Rect.left * dPixelWidth
+ pDoc->GetStartX();
   dApfelYStart = Rect.top * dPixelHeight
+ pDoc->GetStartY();
   dApfelXEnd = Rect.right * dPixelWidth
+ pDoc->GetStartX();
   dApfelYEnd = Rect.bottom * dPixelHeight
+ pDoc->GetStartY();
```

```
   // neue Werte setzen
   pDoc->SetStartX(dApfelXStart);
   pDoc->SetStartY(dApfelYStart);
   pDoc->SetEndX(dApfelXEnd);
   pDoc->SetEndY(dApfelYEnd);

   // neu zeichnen
   InvalidateRect(NULL);
}
```

Der interessante Teil dieser Methode beschränkt sich auf die ersten Zeilen. Dort wird zunächst ein neues *CRectTracker*-Objekt angelegt, das bereits sämtliche Methoden enthält, um eine Gummibandauswahl einzuleiten.

TrackRubberBand

Das Einleiten selbst geschieht durch den Aufruf von *TrackRubberBand*, dessen Prototyp wie folgt aussieht:

```
BOOL TrackRubberBand(
   CWnd* pWnd,
   CPoint point,
   BOOL bAllowInvert = TRUE
);
```

Tabelle 5.4
Parameter

Parameter	Beschreibung
pWnd	Zeiger auf das Fenster, in dem der Benutzer die Auswahl durchführen kann.
point	Relative Mausposition des Mauszeigers im Auswahlbereich.
bAllowInvert	Ist dieser Parameter *true*, können invertierte Auswahlen getätigt werden (d.h., das Rechteck kann von der Startposition aus nach links oben aufgespannt werden).
Rückgabewert	Ein Wert ungleich 0, wenn eine gültige Auswahl getroffen wurde, sonst 0.

Hier sieht man gleich den Vorteil, die Behandlungsmethode direkt in der Ansichtsklasse implementiert zu haben: beim Aufruf von *TrackRubberBand* muss das Fenster übergeben werden, in dem die Auswahl erfolgen soll (und auf das sie dann folgerichtig beschränkt ist).

Da die Ansichtsklasse selbst von *CView* abgeleitet ist, kann einfach der *this*-Zeiger übergeben werden. Die aktuelle Mausposition wird aus dem *OnLButton-*

Down Parameter *point* ausgelesen und negative Rechtecke sollen ebenfalls erlaubt werden (das heißt, dass Rechtecke auch von rechts unten nach links oben aufgezogen werden können, nicht ausschließlich umgekehrt, wie es bei einigen anderen Programmen der Fall ist).

GetTrueRect

Nach der Auswahl

TrackRubberBand unterbricht die Programmabarbeitung, bis der Benutzer die Maustaste wieder losgelassen hat. Danach wird mit *GetTrueRect* das ausgewählte Rechteck ausgelesen:

```
void GetTrueRect( LPRECT lpTrueRect ) const;
```

Parameter	Beschreibung
lpTrueRect	Zeiger auf eine Rechteckstruktur, die die Auswahlkoordinaten des *CRectTracker*-Objekts erhält.

Die ermittelten Koordinaten können dann, wie auch schon in *OnDraw* gesehen, wieder in die Mandelbrot-Koordinaten umgerechnet werden, die dann im Dokument abgelegt werden.

Letztendlich wird die Ansicht zum Neuzeichnen aufgefordert, was in der Folge die Darstellung des gerade gewählten Ausschnitts ergibt.

Alles zurück auf Anfang

Der Benutzer kann jetzt in das Mandelbrot hineinzoomen – nur heraus geht es derzeit noch nicht (außer natürlich über den unbequemen Weg über das Options-Dialogfeld).

Zurücksetzen des Dokuments auf die Ausgangsansicht

Was liegt also näher, als eine Reset-Operation auf die rechte Maustaste zu legen.

Überlegen Sie zunächst einmal selbst, welche Schritte dazu notwendig sind, wenn Folgendes erfüllt sein soll:

- Einleiten des Rücksetzvorgangs durch Drücken der rechten Maustaste in der Client Area des Fensters.
- Zurücksetzen des Ausschnitts auf den Bereich (-2.0, -2.0) bis (2.0, 2.0). Farb- und Iterationseinstellungen sollen unverändert bleiben.
- Neuzeichnen des Apfelmännchens.

Versuchen Sie, diese Schritte zu implementieren. Als kleine Hilfe sei gesagt, dass Sie hierfür nur eine einzige Methode benötigen, deren Rumpf mit maximal zehn Zeilen echtem Code auskommen sollte.

Falls Sie nicht weiterkommen, ist der Lösungsweg im Folgenden angegeben, probieren Sie aber wirklich zunächst eine eigene Lösung aus – wenn diese funktioniert, haben Sie schon einen Großteil des für die MFC-Programmierung notwendigen Wissens erfolgreich verinnerlicht, auch wenn die Aufgabe und ihre Lösung vergleichsweise trivial erscheinen.

Lösung zur Zurücksetzaufgabe

Hier nun der fertige Ansatz: Erzeugen Sie eine Behandlungsmethode für die Nachricht *WM_RBUTTONDOWN* in der *CApfelmaennchenView*-Klasse und schreiben Sie die folgenden Zeilen in den Funktionsrumpf:

Listing 5.21 **Zurücksetzen der Startwerte**

```
void CApfelmaennchenView::OnRButtonDown(UINT nFlags, CPoint point)
{
   // Darstellungsbereich auf Initialwerte
// zurücksetzen
   CApfelmaennchenDoc *pDoc = GetDocument();
   pDoc->SetStartX(-2.0f);
   pDoc->SetStartY(-2.0f);
   pDoc->SetEndX(2.0f);
   pDoc->SetEndY(2.0f);

   // neu zeichnen
   InvalidateRect(NULL);

   // Standardbehandlung aufrufen
   CView::OnRButtonDown(nFlags, point);
}
```

Was passiert? Zunächst wird ein Zeiger auf das Dokument geholt, da einige der Dokumentwerte zurückgesetzt werden sollen.

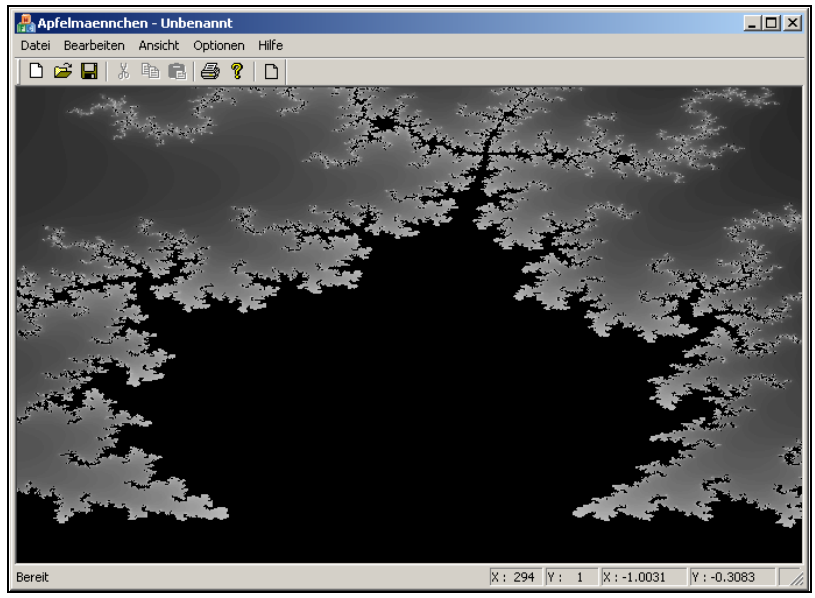
Über diesen Zeiger reinitialisieren Sie dann den Ausgabebereich auf (-2.0, -2.0) bis (2.0, 2.0).

Neuzeichnen des Fensters

Abschließend wird das Fenster durch einen Aufruf von *InvalidateRect* zum Neuzeichnen aufgefordert – fertig ist das Zurücksetzen der Darstellung auf die Ausgangswerte.

Testen Sie nun das Programm aus und wählen Sie verschiedene Bereiche mit dem Gummiband aus – wird die Darstellung zu grob, erhöhen Sie die maximale Iterationstiefe für feiner aufgelöste Ergebnisse:

Abb. 5.23
Apfelmännchen in der Vergrößerung



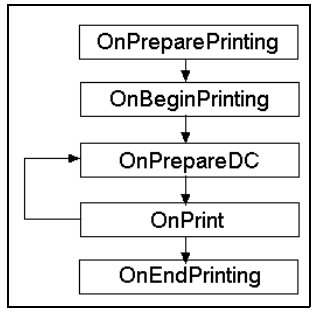
Drucken von Dokumenten

Bislang wurden Ausgaben nur in den Fensterbereich einer Anwendung getätigt. Interessanterweise ist damit aber auch schon die gesamte Grundlage geschaffen worden, die notwendig ist, um Ausgaben auf den Drucker umzuleiten.

Der Ablauf beim Drucken von Dokumenten

Einige Kleinigkeiten müssen aber dennoch eingestellt werden, daher ist es sinnvoll zu wissen, wie der tatsächliche Ablauf im Framework während eines Druckvorgangs aussieht (sprich: die Reihenfolge, in der druckrelevante Funktionen aufgerufen werden, sobald aus der Anwendung heraus beispielsweise der Menüpunkt *Datei* > *Drucken* aufgerufen wird):

Abb. 5.24
Ablauf beim Drucken



Dabei haben diese Methoden folgende Bedeutung:

- *OnPreparePrinting*: Festlegen des zu bedruckenden Seitenbereichs.
- *OnBeginPrinting*: Vorbereiten des Druckvorgangs. Beispielsweise Anlegen von GDI-Objekten, die während des gesamten Druckvorgangs benötigt werden (Brushes, Fonts etc.).
- *OnPrepareDC*: Vorbereiten des Gerätekontexts, insbesondere Einstellen des Abbildungsmodus.
- *OnPrint*: Der eigentliche Druckvorgang, in der Regel ein Aufruf von *OnDraw*. In kleineren Applikationen übernimmt *OnPrint* aber auch die Rolle von *OnPrepareDC* für Druckoperationen.
- *OnEndPrinting*: Nachbereitende Funktion, zum Beispiel Abräumen der in *OnBeginPrinting* angelegten GDI-Objekte etc.

Zyklen beim Drucken

Wichtig zu wissen ist noch, dass der Zyklus *OnPrepareDC->OnPrint->OnPrepareDC* so lange durchlaufen wird, wie neue Seiten vorliegen. *OnPrepareDC->OnPrint* wird also einmal pro auszudruckender Seite aufgerufen.

Die einzelnen Methoden haben gerade in größeren Projekten durchaus ihre Daseinsberechtigung und ermöglichen das effiziente Ausdrucken von umfangreichen Dokumenten.

Es würde den Rahmen dieses Buches sprengen, sämtliche Szenarien zu skizzieren, wie die Methoden eingesetzt werden können, aber ein kleines Beispiel soll trotz allem nicht fehlen.

Ausdrucken von Apfelmännchen

Integrieren der notwendigen Einstellungen für eine Druckausgabe in das Programm

Im Falle unseres Apfelmännchen-Projekts reicht es, die notwendigen Gerätekontexteinstellungen direkt in der *OnPrint*-Methode unterzubringen – in *OnPrepareDC* hatten wir aus Demonstrationsgründen extra abgefragt, ob ein Druckvorgang durchgeführt werden soll und in diesem Fall keine weiteren Einstellungen getätigt.

Editieren Sie den Funktionsrumpf wie folgt:

Listing 5.22
Vollenden des Programms durch Hinzufügen einer Druckfunktion

```
void CApfelmaennchenView::OnPrint(CDC* pDC, CPrintInfo* pInfo)
{
   // bedruckbare Seitengröße auslesen
   CRect ClientRect = pInfo->m_rectDraw;
   int nAufloesung;

   // Clientarea des Fensters auslesen
   CRect WindowRect;
   GetClientRect(&WindowRect);
   nAufloesung = (WindowRect.Width() >
```

```
WindowRect.Height()) ? WindowRect.Width()
: WindowRect.Height();

   // mapping mode setzen
   pDC->SetMapMode(MM_ANISOTROPIC);

   // Clientbereich ...
   pDC->SetWindowOrg(0,0);
   pDC->SetWindowExt(nAufloesung, nAufloesung);

   // ... in Druckbereich mappen
   pDC->SetViewportExt(ClientRect.Width(),
ClientRect.Height());
   pDC->SetViewportOrg(0,0);

   // Standardbehandlung aufrufen
   CView::OnPrint(pDC, pInfo);
}
```

Analog zu *OnPrepareDC* wird hier der bedruckbare Bereich über Fenster- und Viewport-Größenangaben bestimmt. Die Seitengröße stammt dabei aus der übergebenen *PrintInfo*-Struktur, deren genaue Komponenten Sie der Online-Hilfe entnehmen.

Durch das Aufrufen der Standardbehandlung in *CView::OnPrint* wird indirekt auch das Aufrufen von *OnDraw* eingeleitet. Diese Methode ist hinlänglich bekannt und kann hervorragend das Ausdrucken des Apfelmännchens übernehmen.

Laden und Speichern

Serialisierung von Apfelmännchen

Das Laden und Speichern von Dokumentdaten ist vielleicht sogar noch trivialer als das Drucken – sobald man einmal die grundlegenden Prinzipien verstanden hat.

Zunächst sei gesagt, dass bei der Windows-Programmierung häufig der Begriff der *Serialisierung* Verwendung findet, wobei allerdings zumeist nichts anderes gemeint ist, als das Laden und Speichern von Dokumentinhalten.

Dokumentinhalte selbst bestehen häufig aus einer Vielzahl von Informationen, die in der Regel als Objekte abgelegt sind – ein beliebtes Schlagwort ist dabei das der persistenten Objekte, womit solche Objekte gemeint sind, die gespeichert und bei einem Neuaufruf des Dokuments wieder eingelesen werden können.

Die Dokumentenklasse *CApfelmaennchenDoc* des in diesem Kapitel behandelten Projekts enthält bereits eine mit *Serialize* bezeichnete Methode, die den kompletten Ablauf des Ladens und Speicherns übernimmt.

Übergeben bekommt *Serialize* eine Referenz auf ein *CArchive*-Objekt, das die eigentliche Hauptarbeit beim Durchführen der Serialisierung übernimmt.

Das *CArchive*-Objekte kennt die überladenen Operatoren << und >>, die beispielsweise auch bei *cout* und *cin* Verwendung finden. Mit ihrer Hilfe können einfache Datentypen problemlos in Dateien geschrieben werden.

Serialisierung und Dateinamen

Der Serialize-*Methode wird eine* Carchive-*Objektreferenz übergeben. Hieran ist bereits zu sehen, dass offensichtlich ein Archiv geöffnet bzw. angelegt wurde.*

In der Tat finden die dafür notwendigen Operationen nicht erst in Serialize *statt, sondern bereits in vorangegangenen, vom Framework durchgeführten Aufrufen. In der Regel wurde ein Lade- oder Speicherprozess bereits durch eine ganze Reihe von Methoden hindurchgereicht, unter anderem beispielsweise durch die Funktionen* OnOpenDocument *und* OnSaveDocument.

Diese Methoden brauchen nur in den seltensten Fällen vom Programmierer selbst bearbeitet zu werden, normalerweise reichen die vom MFC Framework vorgegebenen Varianten aus. Nehmen Sie also einfach an, dass das Archiv in der Serialize-*Methode bereits Zugriff auf die zugrunde liegende Datei bietet und arbeiten Sie mit ihm wie mit einem offenen Stream.*

Serialize-Methoden für umfangreiche Datentypen

Komplexe Datentypen (also benutzerdefinierte Strukturen und dergleichen mehr) werden für die Serialisierung für gewöhnlich um eine eigene *Serialize*-Methode ergänzt, die dann von der Dokumentenklassen-*Serialize*-Funktion aufgerufen wird – erwünschtes Ziel dabei ist es, dass sich jedes persistente Objekt des Dokuments selbstständig serialisieren kann.

Serialisierung von Apfelmännchen

Im Apfelmännchenprojekt sind solche selbst serialisierenden Objekte nicht notwendig, es können vielmehr einfach die im Dokument befindlichen Variablen direkt in das Archiv geschrieben werden.

CArchive enthält neben den beschriebenen überladenen Streamoperatoren noch eine Reihe weiteres nützlicher Methoden, beispielsweise eine zum direkten Aus- beziehungsweise Einlesen von Strings und, für die Serialisierungsfunktion unumgänglich, eine zum Feststellen, ob gerade geladen oder gespeichert werden soll.

Der Name dieser zuletzt genannten Methode ist *IsStoring*, die *true* zurückliefert, wenn das Dokument (oder besser: das gerade zu serialisierende Objekt) gespeichert werden soll, oder *false*, falls Daten einzulesen sind.

Entwerfen einer *Serialize*-Methode

Erweitern Sie nun also das Apfelmännchenprojekt um die Möglichkeit, Mandelbrot-Ausschnitte – beziehungsweise deren Kenndaten – zu serialisieren:

Listing 5.23 **Laden und Speichern des Dokuments**

```
void CApfelmaennchenDoc::Serialize(CArchive& ar)
{
   if (ar.IsStoring())
   {
      // alle relevanten Daten speichern
      ar << m_dStartX;
      ar << m_dStartY;
      ar << m_dEndX;
      ar << m_dEndY;
      ar << m_nMaxIter;
      ar << m_nStartRed;
      ar << m_nStartGreen;
      ar << m_nStartBlue;
      ar << m_nEndRed;
      ar << m_nEndGreen;
      ar << m_nEndBlue;
      ar << m_bColor;
   }
   else
   {
      // Daten auslesen
      ar >> m_dStartX;
      ar >> m_dStartY;
      ar >> m_dEndX;
      ar >> m_dEndY;
      ar >> m_nMaxIter;
      ar >> m_nStartRed;
      ar >> m_nStartGreen;
      ar >> m_nStartBlue;
      ar >> m_nEndRed;
      ar >> m_nEndGreen;
      ar >> m_nEndBlue;
      ar >> m_bColor;
   }
}
```

Die Methode ist sehr geradlinig geraten und sollte keinerlei Verständnisschwierigkeiten bereiten.

Sie können nun also Dokumentdaten lesen und schreiben, Apfelmännchen drucken und mit der Maus interessante Bereiche vergrößern. Weiterhin haben Sie einiges über die unter Windows zur Verfügung stehenden Abbildungsmodi gelernt.

Abschließend soll es um ein kleines Zusatzfeature gehen, das in zahlreichen Windows-Programmen leider fehlt und doch sehr zur Bedienfreundlichkeit einer Anwendung beiträgt.

Tooltips in Dialogfeldern

Tooltips sind in Windows-Anwendungen an vielen Stellen gang und gäbe und sie werden auch vom MFC Framework standardmäßig unterstützt – Sie können das zum Beispiel bei den Symbolen der Werkzeugleiste sehen, wenn Sie mit dem Mauszeiger darauf zeigen.

Tooltips im *Options*-Dialogfeld

Aber vielleicht ist Ihnen schon aufgefallen, dass es keine Tooltips im *Options*-Dialogfeld der Apfelmännchen-Anwendung gibt.

Das liegt schlicht daran, dass diese Funktionalität nicht direkt von den MFC-Klassen vorgesehen ist, sondern erst freigeschaltet werden muss. Das ist zwar denkbar einfach, stellt einen anfangenden Programmierer aber vermutlich doch vor ein schwer zu überwindende Hürde.

Benachrichtigungsfunktionen für Tooltips

Für Tooltips gilt dasselbe Prinzip wie auch für alle sonstigen unter Windows verfügbaren Komponenten – sie werden erst nach Eingang eines passenden Ereignisses aktiviert, oder im Falle der Tooltips angezeigt.

Aktivierung von Tooltips in benutzerdefinierten Dialogen

Der erste Schritt, um das *Options*-Dialogfeld um Tooltips zu erweitern, muss es daher sein, diese Ereignisse als behandlungswürdig in die Message-Map der *CApfelmaennchenEinstellungen*-Klasse einzubinden.

Editieren Sie die Nachrichtentabelle in *ApfelmaennchenEinstellungen.cpp* daher wie folgt:

```
BEGIN_MESSAGE_MAP(CApfelmaennchenEinstellungen,
CDialog)
   ON_BN_CLICKED(IDC_BUTTON1, OnBnClickedButton1)
   ON_BN_CLICKED(IDC_BUTTON2, OnBnClickedButton2)
   ON_NOTIFY_EX_RANGE(TTN_NEEDTEXTW, 0, 0xFFFF,
 OnToolTipNotify)
   ON_NOTIFY_EX_RANGE(TTN_NEEDTEXTA, 0, 0xFFFF,
OnToolTipNotify)
END_MESSAGE_MAP()
```

Die beiden neuen Zeilen machen inhaltlich das Gleiche, fordern nämlich für Tooltips Texte an, die dargestellt werden sollen, sobald sich der Mauszeiger über einem interessierenden Objekt befindet.

TTN_NEEDTEXTA sorgt für ein Auslesen der ASCII Variante eines Texts (für Win95 Betriebssysteme), *TTN_NEEDTEXTW* (für WinNT Betriebssysteme) für

das Auslesen eines UNICODE-Texts. In diesem einfachen Projekt sollen beide Anfragen durch die gleiche Methode, *OnToolTipNotify*, die im Folgenden noch benutzerdefiniert dargestellt wird, beantwortet werden.

Einträge in den Message Maps

Innerhalb der Message Maps befinden sich eine Reihe von Makros, die näher beschreiben, auf welche Ereignisse eine Klasse reagieren soll.

ON_BN_CLICKED *beispielsweise sagt aus, welche Funktion aufgerufen werden soll, wenn eine Schaltfläche mit der in Klammern spezifizierten ID angeklickt wird.*

ON_NOTIFY_EX_RANGE, *als weiteres Beispiel, übernimmt die Notifizierungs-ID (hier* TTN_NEEDTEXTW *oder* TTN_NEEDTEXTA*), die angibt, welche Anfrage gestellt werden soll, den ID-Bereich, also die Menge der Komponenten, für die dieser Message Map-Eintrag gültig ist sowie die aufzurufende Funktion, die den tatsächlichen Text bereitstellen soll.*

Sie benötigen kein extensives Wissen über diese einzelnen Nachrichteneintragemöglichkeiten, können bei Bedarf aber nähere Informationen aus der Online-Hilfe auslesen – hin und wieder ist es halt notwendig, manuelle Eintragungen vorzunehmen.

OnToolTipNotify-Implementierung

Manuelles Editieren der Message Map

Durch das Eintragen der neuen Zeilen in die Message Map weiß die Anwendung, wo die Texte für die im Optionsfeld vorliegenden Steuerelemente herzuholen sind, nämlich gerade aus der Funktion *OnToolTipNotify*, deren Deklaration in die Datei *ApfelmaennchenEinstellungen.h* gehört:

```
BOOL OnToolTipNotify(UINT id, NMHDR* pNMHDR, LRESULT* pResult);
```

Der Prototyp dieser Funktion ist übrigens vom Framework vorgegeben.

Tücken bei der Behandlung von nicht-standardkonformen Nachrichten

Wenn gesagt wird, dass der Prototyp der Funktionen vom Framework vorgegeben wird, muss allerdings doch gleichzeitig darauf hingewiesen werden, dass die Makros in den Message Maps keinerlei Prüfung dahingehend durchführen, ob die spezifizierten Behandlungsmethoden gültige Signaturen (das heißt Rückgabe- und Parameterwerte) enthalten.

Geben Sie eine inkorrekte – und somit inkompatible – Funktion an, wird es unweigerlich zu Abstürzen kommen, da durch die ungültige Funktion Fehler auf dem Stack verursacht werden.

Schauen Sie also jeweils in der Online-Hilfe nach, welches Ereignis welchen Funktionsprototypen voraussetzt, um unnötige Probleme dieser Art zu vermeiden.

Die Implementierung der *OnToolTipNotify*-Methode ist im Folgenden wiedergegeben, sie gehört in die Datei *ApfelmaennchenEinstellungen.cpp*:

```
BOOL
CApfelmaennchenEinstellungen::OnToolTipNotify(UINT id,
NMHDR* pNMHDR, LRESULT* pResult)
{
// Zeiger auf Tooltiptext holen
    TOOLTIPTEXT* pTTT = (TOOLTIPTEXT*)pNMHDR;
    UINT nID =pNMHDR->idFrom;

   // Tooltip für's Fenster
if (pTTT->uFlags & TTF_IDISHWND)
    {
      // ID der Komponente auslesen
        nID = ::GetDlgCtrlID((HWND)nID);

      // String vorbereiten
      CString s;

      // String holen
      s.LoadString(nID);

      // String in Tooltip schreiben
      strcpy(pTTT->lpszText, s);
    }

   // Tooltip geschrieben
   return(FALSE);
}
```

Beschreibung der Implementierung

Was passiert hier? Zunächst wird der Zeiger auf die an die Funktion übergebene Tooltip-Struktur passend konvertiert, sodass im weiteren Verlauf der Funktion praktisch auf die uns dort interessierenden Komponenten zugegriffen werden kann.

Zugriff auf die wesentlichen Komponenten

Weiterhin wird das auslösende Handle (gerade das Handle des Steuerelements, über dem sich zur Zeit der Mauszeiger befindet) ausgelesen.

Bei Tooltips ist es in der Regel so, dass es sich in der Tat um ein Handle, nicht um eine konkrete ID eines Steuerelements handelt, die an die Notifizierungsmethode übergeben wird. Daher ist es nötig, dieses Handle in eine ID umzurechnen.

Nach einer vorangehenden Sicherheitsüberprüfung dahingehend, ob die in der Struktur gespeicherte ID tatsächlich ein Handle ist, wird die echte ID der Komponente über die Funktion *GetDlgCtrlID* ausgelesen.

Auslesen von Tooltip-Texten aus der Zeichenkettenressource

Es ist jetzt ein Leichtes, diese ID aus der String Table, mit der ja auch zuvor schon gearbeitet wurde, in eine *Cstring*-Variable zu schreiben und diese dann in die Tooltip-Struktur zu kopieren.

Der Rückgabewert *false* teilt dem Aufrufer mit, dass der Tooltip-Text korrekt in die Struktur eingetragen werden konnte – das Tooltip wird dargestellt.

Sie müssen natürlich für die einzelnen IDs passende Texte in die Zeichenkettentabelle ablegen, sonst können die Texte nicht gefunden werden.

Fügen Sie also die folgenden Zeilen in die String Table ein:

Tabelle 5.5
Texte für die Tooltips

ID	Text
IDC_XSTART	X-Start-Komponente
IDC_YSTART	Y-Start-Komponente
IDC_XEND	X-Ende-Komponente
IDC_YEND	Y-Ende-Komponente
IDC_MAXITER	Maximale Iterationstiefe
IDC_COLOR	Farbige Apfelmännchen
IDC_STARTRED	Anfänglicher Rotwert für Farbverlauf
IDC_STARTGREEN	Anfänglicher Grünwert für Farbverlauf
IDC_STARTBLUE	Anfänglicher Blauwert für Farbverlauf
IDC_ENDRED	Abschließender Rotwert für Farbverlauf
IDC_ENDGREEN	Abschließender Grünwert für Farbverlauf
IDC_ENDBLUE	Abschließender Blauwert für Farbverlauf
IDC_BUTTON1	Anfangswert für Farbverlauf festlegen
IDC_BUTTON2	Endwert für Farbverlauf festlegen

Aktivieren der Tooltips

Wenn Sie das Programm jetzt testweise laufen lassen, werden Sie feststellen, dass die Tooltips immer noch nicht angezeigt werden.

Das liegt daran, dass, wie ja schon oben gesagt, Tooltips in Dialogfeldern standardmäßig nicht aktiviert sind.

Sie können dieses aber leicht erreichen, indem Sie die *OnInitDialog*-Methode der *CApfelmaennchenEinstellungen*-Klasse überschreiben und wie folgt editieren:

EnableToolTips

Listing 5.24
Aktivieren der Tooltips

```
BOOL CApfelmaennchenEinstellungen::OnInitDialog()
{
   CDialog::OnInitDialog();

   // TODO:  Add extra initialization here
   EnableToolTips(TRUE);

   return TRUE;  // return TRUE unless you set the
// focus to a control
   // EXCEPTION: OCX Property Pages should return
//FALSE
}
```

Diese Zeilen verändern das initiale Erzeugen des Dialogs nicht, sorgen aber durch die *EnableToolTips*-Zeile dafür, dass Tooltips innerhalb des Dialogs angezeigt werden können.

Dadurch werden nämlich die Textanforderungsnachrichten erst an das Dialogfeld geschickt, der restliche Ablauf ist Ihnen aus der *OnToolTipNotify*-Methode bereits bekannt.

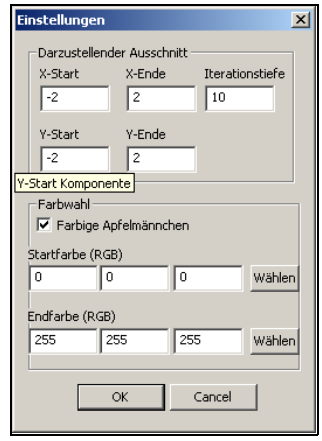


Abb. 5.25
Tooltip im Dialogfenster

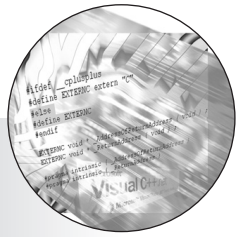
Zusammenfassung

In diesem Kapitel haben Sie gelernt, SDI-Anwendungen zu entwickeln, um Menüs und neue Toolbar-Buttons zu ergänzen, sowie Bedienoberflächen durch das Hinzufügen von selbst definierten und Windows-Standard Dialogfeldern inklusive Tooltips komfortabler zu gestalten.

Sie haben die zahlreichen Abbildungsmodi von Windows kennen gelernt und erfahren, wie einfach eine Gummibandauswahl innerhalb eines Fensters zu realisieren ist.

Abgerundet wurde das Kapitel durch eine Einführung in das Drucken von Dokumenten und die Serialisierung von Dokumentdaten.

Ausblick Im nächsten Kapitel wird es um das Erzeugen von MDI-Applikationen gehen.



MDI-Applikationen

6

MDI-Applikationen

Unterschiede zwischen SDI und MDI

MDI-Anwendungen (Multiple Document Interface) unterscheiden sich von ihren kleineren SDI-Verwandten dadurch, dass es dem Benutzer ermöglicht wird, mehrere Dokumente zur gleichen Zeit geöffnet zu halten, wobei diese in einem gemeinsamen Hauptrahmenfenster zusammen gehalten werden.

Ein gutes Beispiel für diese Art von Anwendung ist auch das Visual Studio selbst, hier können beliebig viele Quelltextfenster nebeneinander existieren. Bei SDI-Anwendungen, auf der anderen Seite, ist immer nur ein Dokument zur Zeit aktiv, das Neuanlegen eines neuen Dokuments oder das Öffnen von gespeicherten Daten, führt zu einem Schließen der derzeit bearbeiteten Datei (einhergehend mit einer Sicherheitsabfrage, falls Veränderungen an dem alten Dokument noch nicht gespeichert wurden.

Der vorliegende Abschnitt beschäftigt sich nun mit MDI-Applikationen und vertieft das Wissen über Serialisierung und Abbildungsmodi. Weiterhin wird aufgezeigt, wie mehrere Ansichtsklassen mit einem Dokument verknüpft werden können, um Dokumentdaten auf verschiedene Weisen darstellen zu können.

Quelltextunterschiede zwischen SDI-und MDI-Applikationen

CChildFrame

Im Wesentlichen gibt es nur eine weitreichende Änderung zwischen SDI-und MDI-Applikationen, sofern der vom Anwendungsassistenten erzeugte Projektcode betrachtet wird: neben der Klasse *CMainFrame* für das Hauptrahmenfenster der Anwendungen existiert noch eine weitere Klasse *CChildFrame*, die das Aussehen und Verhalten der einzelnen Dokumentfenster beschreibt.

Bei SDI-Anwendungen war diese Unterscheidung nicht erforderlich, da die Dokumentansichten der SDI-Anwendungen ohnehin immer den kompletten Client-Bereich des Hauptfensters ausfüllten und somit kein eigenes Fenster mit Rahmen benötigten.

Weitere Unterschiede beziehen sich auf die Art, wie die *OnNewDocument*-Methode eingesetzt wird, aber auch auf die Anbindung von Ansichtsklassen und das Erzeugen des Hauptrahmenfensters.

Auf diese wird zu gegebener Zeit eingegangen, sobald die betreffenden Quelltextpassagen besprochen werden.

Das MDI-Projekt „Bezierkurven“

Nachdem wir uns bereits im letzten Kapitel mit mathematischen Gebilden beschäftigt hatten, soll es in diesem Abschnitt um die Darstellung von Bezierkurven gehen.

Einführung in Bezierkurven

Eine Bezierkurve besteht prinzipiell aus vier einzelnen Punkten, von den zwei den Anfangs- und Endpunkt der Kurve beschreiben, während die anderen beiden für die Kurvenführung selbst verwendet werden.

Eine Kurve bewegt sich dabei vom Startpunkt aus jeweils auf den ersten Kontrollpunkt zu, und verändert ihren Lauf dann derart, dass sie, vom zweiten Kontrollpunkt ausgehend, auf den Endpunkt zulaufen kann.

Am besten erklärt sich dieses Verhalten durch einige Beispielkurven. In der folgenden Grafik sind die eckigen Punkte jeweils Start- bzw. Endpunkt, während die runden Punkte die Kontrollpunkte darstellen.

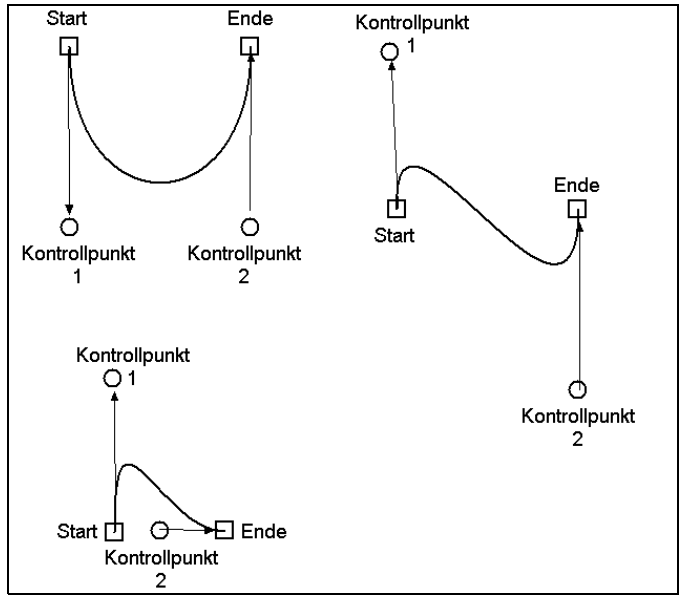


Abb. 6.1
Beispiele für Bezierkurven

Das zu entwickelnde Programm soll nun den Benutzer befähigen, solche Kurven selbstständig durch einfache Drag-&-Drop-Operationen anzulegen. Weiterhin sollen die aktuellen Positionen der Knoten- und Kontrollpunkte in einem zweiten Fenster optional eingeblendet werden können – dieses wird durch eine weitere Ansichtsklasse realisiert.

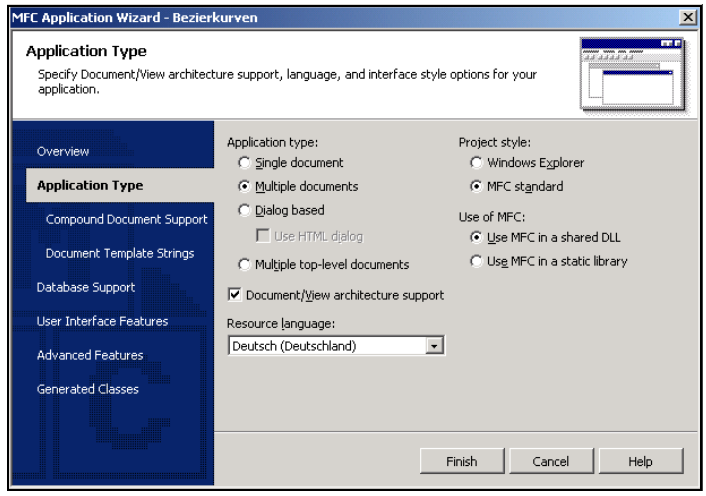
Anlegen des Projekts

Festlegen eines Applikationstyps

Der Prozess des Anlegens des Projekts entspricht dem der vorhergehenden Kapitel. Geben Sie dem Projekt den Namen *Bezierkurven* (oder wählen Sie das bereits fertige Projekt von der Buch-CD aus dem Verzeichnis *\Kapitel6\Bezierkurven*).

Abb. 6.2
Auswahl des Applikationstyps

Legen Sie als Applikationstyp *Multiple Documents* fest:



Austesten der MDI-Anwendung

Der Anwendungsassistent hat nun, wie Sie es mittlerweile gewohnt sind, eine komplett lauffähige Applikation erzeugt.

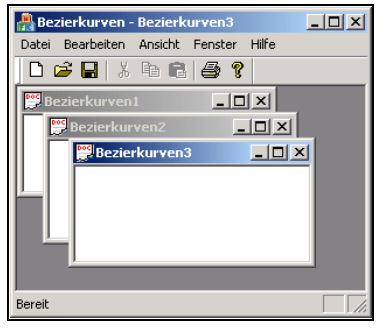
Austesten der generierten Anwendung

Kompilieren Sie daher als Erstes die entstandenden Sources und testen Sie das Grundgerüst auf seine Funktionalität hin aus. Wie Sie sehen, besteht die Anwendung neben dem Hauptfenster mit Menü und Werkzeugleiste aus einer freien Fläche, in der sich die einzelnen Dokumentfenster frei positionieren lassen.

Erzeugen Sie einige neue Dokumente durch Anwahl des Menüpunkts *Datei* > *Neu* beziehungsweise durch wiederholtes Anklicken des Symbols *Neues Dokument* in der Werkzeugleiste des Programms.

Sie erhalten eine Ansicht ähnlich der folgenden:

Abb. 6.3
Weitere Dokumente



Die neuen Dateien

Bevor es um die Erweiterung des derzeit noch recht langweilig wirkenden Programms gehen soll, ist es zweckmäßig, einen Blick auf die neuen Dateien zu werfen, die Ihnen aus den SDI- und dialogfeldbasierten Anwendungen bislang noch nicht vertraut sind.

childfrm.h und *childfrm.cpp*

Zu nennen sind hier die beiden Files *childfrm.h* und *childfrm.cpp*, die im Folgenden abgedruckt und erläutert sind. Begonnen wird mit der Headerdatei *childfrm.h*:

Listing 6.1 **Headerdatei für Kindfenster**

```
#pragma once

class CChildFrame : public CMDIChildWnd
{
   DECLARE_DYNCREATE(CChildFrame)
public:
   CChildFrame();

// Attribute
public:

// Operationen
public:

// Überschreibungen
   virtual BOOL PreCreateWindow(CREATESTRUCT& cs);

// Implementierung
public:
   virtual ~CChildFrame();
#ifdef _DEBUG
   virtual void AssertValid() const;
   virtual void Dump(CDumpContext& dc) const;
#endif

// Generierte Funktionen für die Meldungstabellen
protected:
   DECLARE_MESSAGE_MAP()
};
```

Diese Datei entspricht inhaltlich, bis auf fehlende Status- und Werkzeugleistenkomponenten, der *MainFrm.h* aus dem SDI-Kapitel. Das ist bei näherer Überlegung auch einsichtig klar, denn MDI-Dokumentfenster sind, wie Hauptrahmenfenster auch, einfache Fenster und als solche funktionsgleich mit dem übergeordneten Elternfenstern.

Wiederverwendung und Analogie von MFC-Klassen

Ein großer Vorteil bei der Arbeit mit den MFC ist es, dass die in einer Hierarchie befindlichen oder nur anderweitig ähnlichen Klassen stets auf dieselbe Art und Weise aufgebaut sind.

Egal, ob nun ein Hauptrahmen- oder ein Kindfenster als Klasse definiert wird, die MFC-Entwickler haben darauf geachtet, dass die grundlegenden Elemente bei beiden Varianten dieselben sind.

Auf diese Weise ist es einem Entwickler möglich, sich schnell in neue Klassen, mit denen er zuvor noch nicht gearbeitet hat, einzuarbeiten.

Sicherlich vergeht einige Zeit, bis Sie die Gemeinsamkeiten ausgemacht und zu Ihrem Vorteil nutzen können, doch sind Sie daraufhin in der Lage, sehr effizient mit den MFC-Klassen zu arbeiten.

Ein weiteres Beispiel für die Wiederverwendbarkeit von bereits bestehenden MFC-Klassen wird noch im weiteren Verlauf dieses Kapitels aufgezeigt.

Die Implementationsdatei *CChildFrm*

Die Implementationsdatei der *CChildFrm*-Klasse enthält im Vergleich zur SDI-*CMainFrame*-Klasse ebenfalls nur geringfügige Unterschiede:

Listing 6.2 CChildFrm.cpp

```
#include "stdafx.h"
#include "Bezierkurven.h"

#include "ChildFrm.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif


// CChildFrame

IMPLEMENT_DYNCREATE(CChildFrame, CMDIChildWnd)

BEGIN_MESSAGE_MAP(CChildFrame, CMDIChildWnd)
END_MESSAGE_MAP()


// CChildFrame Erstellung/Zerstörung

CChildFrame::CChildFrame()
```

```
{
   // TODO: Hier Code für die Memberinitialisierung einfügen
}

CChildFrame::~CChildFrame()
{
}


BOOL CChildFrame::PreCreateWindow(CREATESTRUCT& cs)
{
   // TODO: Ändern Sie die Fensterklasse oder die
// Stile hier, indem Sie CREATESTRUCT ändern
   if( !CMDIChildWnd::PreCreateWindow(cs) )
      return FALSE;

   return TRUE;
}


// CChildFrame Diagnose

#ifdef _DEBUG
void CChildFrame::AssertValid() const
{
   CMDIChildWnd::AssertValid();
}

void CChildFrame::Dump(CDumpContext& dc) const
{
   CMDIChildWnd::Dump(dc);
}

#endif //_DEBUG
```

Am auffälligsten ist sicherlich, dass die *CchildFrame*-Klasse von *CMDIChildWnd* abgeleitet ist und entsprechend deren Basisklassenmethoden verwendet und aufruft.

CMDIChildWnd

CMDIChildWnd stellt eine komplett vollständige MDI-Kindfensterklasse dar, also eine innerhalb eines Hauptrahmenfensters frei verschiebbare, minimierbare Fenstervariante, die kein eigenen Menü besitzt, sondern vielmehr das Menü des Elternfensters mit verwendet.

Es besteht allerdings die Möglichkeit, das Menü entsprechend des gerade aktiven Dokuments zu verändern, was natürlich insbesondere bei mehreren vorhandenen Ansichtsklassen interessant sein könnte.

Ausgabe von Bezierkurven

Zeichnen von Bezierkurven

Damit die MDI-Fenster nicht so leer aussehen, wie sie es derzeit tun, soll jetzt eine erste Implementation der Bezierkurven folgen, die zunächst noch mit statischen Positionen für die Knoten- und Kontrollpunkte arbeitet.

Als Erstes sollte dazu aber ein passender Abbildungsmodus gewählt werden, sodass die Bezierkurven verzerrungsfrei gezeichnet werden.

Überschreiben Sie dazu die Methode *OnPrepareDC* der *CbezierkurvenView*-Klasse, wie Sie es im letzten Kapitel bereits getan haben.

Füllen Sie daraufhin den Funktionsrumpf mit den nachstehenden Zeilen:

Listing 6.3 **Einstellen des Abbildungsmodus**

```
void CBezierkurvenView::OnPrepareDC(CDC* pDC,
CPrintInfo* pInfo)
{
   // TODO: Add your specialized code here and/or
// call the base class
   RECT Client;
   GetClientRect(&Client);

   // Abbildungsmodus setzen
   pDC->SetMapMode(MM_ISOTROPIC);

   // Kontextausmaße setzen
   pDC->SetWindowExt(300, 300);
   pDC->SetViewportExt(Client.right,
Client.bottom);

   CView::OnPrepareDC(pDC, pInfo);
}
```

Der Abbildungsmodus wird hier, da eine verzerrungsfreie Ansicht gewünscht ist, auf *MM_ISOTROPIC* gesetzt, weiterhin wird der Ausgabebereich in 300*300 logische Einheiten aufgeteilt.

Man könnte hier auch eine höhere Auflösung wählen, insbesondere um eine feinere Positionierung der Knoten- und Kontrollpunkte zu erlauben, doch zu Demonstrationszwecken ist diese Granularitätsstufe vollkommen ausreichend.

Zeichnen der Kurven

Die neue *OnDraw*-Methode

Gezeichnet werden die Bezierkurven in der *OnDraw*-Methode, die Sie nun um folgende Zeilen ergänzen:

```
void CBezierkurvenView::OnDraw(CDC* pDC)
{
   CBezierkurvenDoc* pDoc = GetDocument();
```

```
   ASSERT_VALID(pDoc);

   // Festlegen der Bezierkurven Knoten-
   // und kontrollpunkte
   POINT Bezier[4];

   Bezier[0].x = 100;
   Bezier[0].y = 100;

   Bezier[1].x = 100;
   Bezier[1].y = 200;

   Bezier[2].x = 200;
   Bezier[2].y = 200;

   Bezier[3].x = 200;
   Bezier[3].y = 100;

   // Zeichnen der Knotenpunkte
   pDC->Rectangle(Bezier[0].x – 5,
Bezier[0].y – 5,
            Bezier[0].x + 5,
Bezier[0].y + 5);

   pDC->Rectangle(Bezier[3].x – 5,
Bezier[3].y – 5,
            Bezier[3].x + 5,
Bezier[3].y + 5);

   // Zeichnen der Kontrollpunkte
   pDC->Ellipse(Bezier[1].x – 5,
Bezier[1].y – 5,
            Bezier[1].x + 5,
Bezier[1].y + 5);

   pDC->Ellipse(Bezier[2].x – 5,
Bezier[2].y – 5,
               Bezier[2].x + 5,
Bezier[2].y + 5);

   pDC->PolyBezier(Bezier, 4);
}
```

Zeichenfunktionen

Nachdem die Koordinaten für Knoten- und Kontrollpunkte festgelegt wurden, werden die Rechtecke für die Knoten- und Ellipsen für die Kreise gezeichnet. Dabei kommen die Gerätekontextmethoden *Rectangle* und *Ellipse* zum Einsatz, deren Prototypen im Folgenden abgedruckt sind:

```
BOOL Rectangle(
   int x1,
   int y1,
   int x2,
   int y2
);
```

Parameter	Beschreibung
x1	X-Koordinate der linken oberen Ecke des zu zeichnenden Rechtecks in logischen Koordinaten.
y1	Y-Koordinate der linken oberen Ecke des zu zeichnenden Rechtecks in logischen Koordinaten.
x2	X-Koordinate der rechten unteren Ecke des zu zeichnenden Rechtecks in logischen Koordinaten.
y2	Y-Koordinate der rechten unteren Ecke des zu zeichnenden Rechtecks in logischen Koordinaten.
Rückgabewert	Wert ungleich 0 bei Erfolg, sonst 0

```
BOOL Ellipse(
   int x1,
   int y1,
   int x2,
   int y2
);
```

Parameter	Beschreibung
x1	X-Koordinate der linken oberen Ecke der Boundingbox der zu zeichnenden Ellipse in logischen Koordinaten.
y1	Y-Koordinate der linken oberen Ecke der Boundingbox der zu zeichnenden Ellipse in logischen Koordinaten.
x2	X-Koordinate der rechten unteren Ecke der Boundingbox der zu zeichnenden Ellipse in logischen Koordinaten.
y2	Y-Koordinate der rechten unteren Ecke der Boundingbox der zu zeichnenden Ellipse in logischen Koordinaten.
Rückgabewert	Wert ungleich 0 bei Erfolg, sonst 0

Anpassen der Koordinaten

Die Koordinaten werden jeweils um den Wert 5 erhöht beziehungsweise vermindert, um eine ausreichend große Region um den spezifizierten Punkt zu erzeugen, und somit das gezeichnete Objekte (das Rechteck beziehungsweise die Ellipse) deutlich sichtbar im Fenster zu repräsentieren.

Darstellung von Objekten in den verschiedenen Abbildungsmodi

Es ist bei der Arbeit mit den unterschiedlichen Abbildungsmodi darauf zu achten, dass Objekte, die immer eine durchgehend gleiche Größe behalten sollen, auch noch einmal mit den logischen Koordinaten verrechnet werden.

Angenommen, ein Rechteck soll immer 10 Pixel breit und hoch sein, so müsste bei einer Fensterbreite von 1.000 Pixeln und 1.000 eingestellten logischen Einheiten im Abbildungsmodus ein Rechteck mit einer Breite von 10 logischen Einheiten gezeichnet werden, während bei einer Fensterbreite von 500 Pixeln und gleichen logischen Einheiten ein Rechteck von 20 logischen Einheiten gezeichnet werden muss.

Dieses Verfahren bietet sich zum Beispiel dann an, wenn, wie hier, Objekte zur Benutzerinteraktion verwendet werden, die mit der Maus angewählt und verschoben werden können.

Als allgemeine Formel gilt:

ZLE = DP / (FB / LFE)

wobei ZLE die zu zeichnenden logischen Einheiten, DP die darzustellende Pixelgröße, FB die Fensterbreite und LFE die Zahl der logischen Einheiten im Fenster ist.

Beachten Sie, dass diese Rechnungen für Höhe und Breite gegebenenfalls separat durchzuführen sind, wenn ein anisotropischer Abbildungsmodus eingestellt ist, oder sich die eingestellten logischen Einheiten im Fenster in Breite und Höhe unterscheiden.

Zeichnen von Bezierkurven

PolyBezier

Die Darstellung von Bezierkurven ist denkbar einfach, da es unter den Gerätekontextmethoden bereits eine dafür vorgesehene Funktion namens *PolyBezier* gibt:

```
BOOL PolyBezier(
   const POINT* lpPoints,
   int nCount
);
```

Parameter	Beschreibung
lpPoints	Zeiger auf ein Array von POINT-Strukturen, die die einzelnen Koordinaten der Kontroll- und Knotenpunkte enthalten. Die Reihenfolge dabei ist jeweils: Startpunkt, Kontrollpunk1, Kontrollpunkt2, Endpunkt. Bei mehr als einer Kurve dient der Endpunkt einer Kurve gleichzeitig als Startpunkt der nächsten Kurve, sodass für n Kurven ((2 * n) + 1) POINT-Strukturen benötigt werden.
nCount	Anzahl der POINT-Strukturen ab Adresse *lpPoints*.
Rückgabe-wert	Wert ungleich 0 bei Erfolg, sonst 1.

Sie können sich das Ergebnis nach einem Neukompilieren des Projekts direkt ansehen:

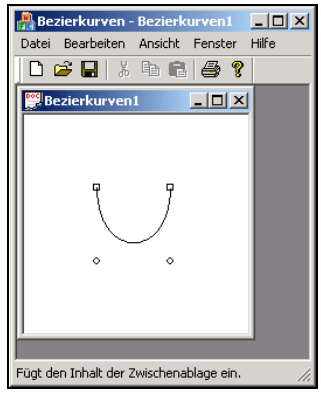


Abb. 6.4
Eine erste Bezierkurve

Wenn Sie jetzt weitere neue Dokumente erzeugen, werden zusätzliche Kindfenster geöffnet, die jeweils die gleiche Kurvendarstellung zeigen.

Der nächste Schritt besteht nun darin, die bislang statischen Knoten- und Kontrollpunkt-Koordinaten in die Dokumentenklasse zu verlagern.

Erweitern der Dokumentenklasse

Die Dokumentenklasse findet sich in den Dateien *BezierkurvenDoc.h* und *BezierkurvenDoc.cpp* – darin wird eine Klasse *CBezierkurvenDoc* deklariert und definiert, deren Aufbau exakt dem der SDI-Dokumentenklasse des letzten Kapitels entspricht.

Anlegen passender Datentypen

Im Bezierkurven-Projekt sollen die Koordinaten der darzustellenden Kurve im Dokument abgelegt werden. Um zu zeigen, wie komplexe Datentypen serialisiert werden können, werden die Daten in diesem Beispiel in einer Struktur abgelegt, die nachfolgend beschrieben ist.

Struktur zur Aufnahme der Bezierkurven-Werte

Fügen Sie die Zeilen der Strukturdeklaration in die Datei *BezierkurvenDoc.h* ein:

Listing 6.4 **Struktur zur Aufnahme der Bezierkurvendaten**

```
struct SBezierKurve
{
   POINT m_StartKnoten;
   POINT m_EndKnoten;
   POINT m_KontrollPunkt1;
   POINT m_KontrollPunkt2;

   void Serialize(CArchive &ar);
};
```

Die vier relevanten Punkte einer Bezierkurve werden jeweils über eine *POINT* Struktur gesichert. Das erlaubt eine übersichtliche Objekt-Struktur und einfachen Zugriff auf die tatsächlich interessierenden X-/Y-Koordinaten.

Zusätzlich wird eine Funktion *Serialize* in die Struktur aufgenommen, die in *CBezierkurvenDoc::Serialize* aufgerufen werden soll und die Hauptarbeit beim Serialisieren der Bezierkurvendaten übernimmt.

Änderungen an *CBezierkurvenDoc*

Die Klassendeklaration von *CBezierkurvenDoc* muss nun um die Aufnahme der Datenstruktur sowie geeignete Zugriffsmethoden erweitert werden:

Listing 6.5 **Erweiterte Klassendeklaration**

```
class CBezierkurvenDoc : public CDocument
{
protected: // Nur aus Serialisierung erstellen
   CBezierkurvenDoc();
   DECLARE_DYNCREATE(CBezierkurvenDoc)

// Attribute
public:

// Operationen
public:

// Überschreibungen
   public:
   virtual BOOL OnNewDocument();
   virtual void Serialize(CArchive& ar);
```

```
// Implementierung
public:
   virtual ~CBezierkurvenDoc();
#ifdef _DEBUG
   virtual void AssertValid() const;
   virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generierte Funktionen für die Meldungstabellen
protected:
   DECLARE_MESSAGE_MAP()

private:
   SBezierKurve m_BezierKurve;
public:
   POINT GetStartNode();
   POINT GetEndNode();
   POINT GetControlPoint1();
   POINT GetControlPoint2();

   void SetStartNode(POINT in_Point);
   void SetEndNode(POINT in_Point);
   void SetControlPoint1(POINT in_Point);
   void SetControlPoint2(POINT in_Point);
};
```

Diese Änderungen sind leicht verständlich und analog zu denen des letzten Kapitels. Es ist zu erkennen, dass der Aufbau eines MDI-Anwendungsdokuments exakt dem eines SDI-Anwendungsdokuments entspricht – Sie können sich also auch hier auf bereits bekannte Strukturen und Formalitäten verlassen.

Das Serialisieren der *SBezierKurve*-Struktur

Sie haben im letzten Kapitel gesehen, wie mithilfe der *Serialize*-Methode der Dokumentenklasse die Daten eines Dokuments gespeichert werden können.

Delegation der Serialisierung

Im neuen Projekt wird die tatsächliche Serialisierungsarbeit an ein Strukturobjekt delegiert. Die Implementation der zugehörigen Serialisierungsmethode für dieses Objekt gehört in die Datei *BezierkurvenDoc.cpp* und hat nachstehenden Aufbau:

***Listing 6.6* Serialisieren der Struktur**

```
// Serialize Methode für SBezierKurve
void SBezierKurve::Serialize(CArchive &ar)
{
   if (ar.IsStoring())
   {
```

```
      // Knoten- und Kontrollpunkte in Archiv
      // schreiben
      ar << m_StartKnoten;
      ar << m_KontrollPunkt1;
      ar << m_KontrollPunkt2;
      ar << m_EndKnoten;
   }
   else
   {
      // Knoten- und Kontrollpunkte aus Archiv
      // lesen
      ar >> m_StartKnoten;
      ar >> m_KontrollPunkt1;
      ar >> m_KontrollPunkt2;
      ar >> m_EndKnoten;
   }
}
```

Die Methode tut das, was man erwarten sollte: Nachdem geprüft wird, ob ein Speicher- oder Ladezugriff vorliegt, werden die Daten der Punkte elementweise in das Archiv geschrieben beziehungsweise aus diesem ausgelesen.

POINT-Struktur können dabei dankenswerterweise in einem Schritt transfertiert werden, sodass hier keine weitere Aufsplittung in X- und Y-Koordinaten notwendig ist.

Serialisierung des Dokuments

Die Serialisierung des Dokuments, die von der *CBezierkurvenDoc::Serialize*-Methode ausgeht, ist nun recht trivial, da außer einer *SBezierKurve*-Struktur keine weiteren Dokumentdaten existieren.

Editieren Sie die Serializemethode daher wie folgt:

Listing 6.7
Die neue *Serialize*-Methode

```
void CBezierkurvenDoc::Serialize(CArchive& ar)
{
   m_BezierKurve.Serialize(ar);
}
```

Da die Hauptarbeit ohnehin von *SBezierKurve::Serialize* erledigt wird, reicht hier eine Delegation an diese Methode aus. Man spricht hierbei häufig von Relaisfunktionen, die nur deshalb notwendig sind, weil sie im Rahmen eines globalen Anwendungsgerüsts unabdingbar aufgerufen werden.

Implementationen der Zugriffsmethoden

Damit die Ansichtsklasse der Anwendung auch auf die Dokumentdaten zugreifen kann, müssen passende Zugriffsmethoden bereitgestellt werden, die

jeweils *POINT*-Struktur übergeben beziehungsweise im Fall von setzenden Methoden als Parameter übernehmen.

Zugriffsmethoden

Die Implementationen gehören ebenfalls in die Datei *BezierkurvenDoc.cpp* und bedürfen keiner weiteren Erläuterungen, weshalb sie hier einfach in ihrer Quelltextform abgedruckt werden sollen:

```
POINT CBezierkurvenDoc::GetStartNode()
{
   return (m_BezierKurve.m_StartKnoten);
}

POINT CBezierkurvenDoc::GetEndNode()
{
   return (m_BezierKurve.m_EndKnoten);
}

POINT CBezierkurvenDoc::GetControlPoint1()
{
   return (m_BezierKurve.m_KontrollPunkt1);
}

POINT CBezierkurvenDoc::GetControlPoint2()
{
   return (m_BezierKurve.m_KontrollPunkt2);
}

void CBezierkurvenDoc::SetStartNode (POINT in_Point)
{
   m_BezierKurve.m_StartKnoten = in_Point;
}

void CBezierkurvenDoc::SetEndNode (POINT in_Point)
{
   m_BezierKurve.m_EndKnoten = in_Point;
}

void CBezierkurvenDoc::SetControlPoint1 (POINT in_Point)
{
   m_BezierKurve.m_KontrollPunkt1 = in_Point;
}

void CBezierkurvenDoc::SetControlPoint2 (POINT in_Point)
{
   m_BezierKurve.m_KontrollPunkt2 = in_Point;
}
```

Initialisieren des Dokuments

Im Gegensatz zu SDI-Anwendungen, bei denen die *OnNewDocument*-Methode jedes Mal beim Erzeugen eines neuen Dokuments aufgerufen wurde, werden bei MDI-Applikationen jeweils neue Dokument-Objekte erzeugt – was in der Quintessenz zu einem ständigen Aufrufen der *OnNewDocument*-Methode führt, sobald ein Dokument neu erschaffen wird.

Koordinaten initialisieren

Trotzdem bietet es sich an, sämtliche Initialisierungen für ein Objekt in der Form innerhalb dieser Methode vorzunehmen, dass ein Dokument komplett durch einen einfachen Zugriff auf diese Funktion auf einen definierten Startzustand gesetzt wird – die Verwendung von Konstruktoren zu diesem Zweck ist, wie Sie in den vorangegangenen Abschnitten bereits mehrfach gesehen haben, eher als verpönt anzusehen.

Die Bezierkurven-Dokumente benötigen ledlich eine Initialisierung der Koordinaten von Kontroll- und Knotenpunkten, die eingangs auf die gleichen Werte gesetzt werden sollen, die bereits in der ersten Version der *OnDraw*-Methode verwendet wurden.

Sorgen Sie für diesen Umstand durch Hinzufügen einiger Zeilen in die *OnNewDocument*-Methode:

```
BOOL CBezierkurvenDoc::OnNewDocument()
{
   if (!CDocument::OnNewDocument())
      return FALSE;

   // TODO: Hier Code zur Reinitialisierung
// einfügen
   // (SDI-Dokumente verwenden dieses Dokument)
   m_BezierKurve.m_StartKnoten.x = 100;
   m_BezierKurve.m_StartKnoten.y = 100;

   m_BezierKurve.m_EndKnoten.x = 200;
   m_BezierKurve.m_EndKnoten.y = 100;

   m_BezierKurve.m_KontrollPunkt1.x = 100;
   m_BezierKurve.m_KontrollPunkt1.y = 200;

   m_BezierKurve.m_KontrollPunkt2.x = 200;
   m_BezierKurve.m_KontrollPunkt2.y = 200;

   return TRUE;
}
```

Darstellen der Dokumentdaten

Adaptieren der Ansichtsklasse

Nachdem die notwendigen Daten nun im Dokument gespeichert und auch bereits auf vordefinierte Werte gesetzt werden, ist es höchste Zeit, die Ansichtsklasse dahingehend anzupassen, dass sie die gesicherten Informationen visualisiert.

Zunächst ist ihnen in der bisherigen *OnDraw*-Methode vielleicht aufgefallen, dass die Größe der Rechtecke und Ellipsen mit dem Zahlwert 5 fix gesetzt wurde. Dieses ist eine programmiertechnische Unzulänglichkeit, die in jedem Fall behoben werden sollte.

Magische Zahlen

Immer, wenn konstante Zahlen in Quelltexten eingesetzt werden, wie es hier im Falle der Größenangaben für Rechtecke und Ellipsen vorkommt, sollte sich der Programmierer Gedanken darüber machen, für diese Werte eine Konstante einzuführen und statt mit dem Skalar mit diesem nicht veränderlichen Wertespeicher zu arbeiten.

Die Vorteile liegen auf der Hand: abgesehen davon, dass Quelltexte, die mit aussagekräftigen Konstantennamen arbeiten, deutlich einfacher zu lesen sind (die 5 könnte in unserem Fall ja auch für ganz andere Sachverhalte stehen – im englischen spricht man hier gern von einem worst- case-Szenario, wenn aus dem Kontext überhaupt nicht hervorgeht, wofür der Wert steht. Diese Zahlen bezeichnet man daher auch als magische Zahlen, da ihre mögliche Bedeutung nur durch die Vorstellungskraft des Quelltextlesers beschränkt wird), wird die Wartung deutlich vereinfacht.

Angenommen, Sie stellen fest, dass die Größe der Rechtecke nicht adäquat zu ihren Vorstellungen ist, können Sie bei Verwendung einer Konstanten einfach deren Wert geeignet anpassen und müssen nicht womöglich hunderte von Quelltextstellen editieren – man stelle sich nur vor, wie leicht ein Vorkommen des Werts übersehen werden kann, was dann in der Folge zu unvorgesehenen Ergebnissen führen mag.

Als Faustregel gilt, dass für jeden Wert außer 0 und 1 eine Konstante eingeführt werden sollte, sogar wenn absehbar ist – oder gerade dann erst recht – dass der Wert nur einmal innerhalb des Programms verwendet werden wird. Ausnahmen bilden hier lediglich Indizes von Arrays.

Fügen Sie die Konstante in die Daten *BezierkurvenView.cpp* ein:

```
const c_iKnotenRadius = 5;
```

Eine neue *OnDraw*-Methode

Nun ist es an der Zeit, die veränderte Methode *OnDraw* der Klasse *CBezierkurvenView* zu betrachten:

```
void CBezierkurvenView::OnDraw(CDC* pDC)
{
   CBezierkurvenDoc* pDoc = GetDocument();
   ASSERT_VALID(pDoc);

   // Festlegen der Bezierkurven Knoten-
   // und kontrollpunkte
   POINT Bezier[4];

   Bezier[0] = pDoc->GetStartNode();
   Bezier[1] = pDoc->GetControlPoint1();
   Bezier[2] = pDoc->GetControlPoint2();
   Bezier[3] = pDoc->GetEndNode();

   // Zeichnen der Knotenpunkte
   pDC->Rectangle(Bezier[0].x – c_iKnotenRadius,
         Bezier[0].y – c_iKnotenRadius,
         Bezier[0].x + c_iKnotenRadius,
         Bezier[0].y + c_iKnotenRadius);

   pDC->Rectangle(Bezier[3].x – c_iKnotenRadius,
         Bezier[3].y – c_iKnotenRadius,
         Bezier[3].x + c_iKnotenRadius,
         Bezier[3].y + c_iKnotenRadius);

   // Zeichnen der Kontrollpunkte
   pDC->Ellipse(Bezier[1].x – c_iKnotenRadius,
         Bezier[1].y – c_iKnotenRadius,
         Bezier[1].x + c_iKnotenRadius,
         Bezier[1].y + c_iKnotenRadius);

   pDC->Ellipse(Bezier[2].x – c_iKnotenRadius,
         Bezier[2].y – c_iKnotenRadius,
         Bezier[2].x + c_iKnotenRadius,
         Bezier[2].y + c_iKnotenRadius);

   pDC->PolyBezier(Bezier, 4);
}
```

Beachten Sie bei dieser neuen Variante vor allem auch, dass sie durch die Verwendung der *c_iKnotenRadius*-Konstante deutlich besser lesbar ist.

Das Austesten dieser neuen Version wird zu keinen großartigen Unterschieden im Vergleich zum Vorgänger führen, da hier zwar einiges umgestellt wurde, die Ausgangswerte der Knoten- und Kontrollpunkte jedoch unverändert geblieben sind.

Der nächste Schritt muss also darin bestehen, die Benutzerinteraktion zu integrieren, die ein Verschieben der einzelnen Punkte gestattet.

Integration von Benutzerinteraktionen

Das Bedieninterface soll einfach strukturiert sein: Durch das Anklicken eines Punkts kann ein Benutzer, bei gedrückt gehaltener Maustaste, den jeweils gerade angewählten Punkt verschieben. Der momentan bewegte Punkt soll weiterhin mit einer anderen Farbe und in einer stärken Liniendicke dargestellt werden.

Statusvariablen

Aktuelle Bewegungen

Damit das Programm jederzeit darüber informiert ist, welcher Punkt gerade bewegt wird, werden in der Ansichtsklasse *CBezierkurvenView* vier boolesche Variablen angelegt, die jeweils für eine der vier (nur unabhängig voneinander durchführbaren) Bewegungswünsche vorgesehen sind.

Fügen Sie die Variablen in die Klassendeklaration in die Datei *BezierkurvenView.h* ein:

```
private:
   bool m_bMovingStartNode;
   bool m_bMovingEndNode;
   bool m_bMovingControlPoint1;
   bool m_bMovingControlPoint2;
```

Initialisierung der Statusvariablen

Konstruktoreinsatz

Die Statusvariablen müssen pro geöffneter Ansicht nur einmal initialisiert werden. Dieses ist also einer der wenigen Fälle, in denen der Konstruktor zur Initialisierung durchaus herangezogen werdend darf.

Editieren Sie daher den Konstruktor der *CBezierkurvenView*-Klasse wie folgt:

Listing 6.8
Der neue Konstruktor

```
CBezierkurvenView::CBezierkurvenView()
{
   m_bMovingStartNode = false;
   m_bMovingEndNode = false;
   m_bMovingControlPoint1 = false;
   m_bMovingControlPoint2 = false;
}
```

Anwählen von Punkten

Der Benutzer soll Knoten- und Kontrollpunkte mit der Maus anwählen können, genauer gesagt immer dann, wenn er einen Klick mit der linken Maustaste durchführt.

Behandlungsmethode für die linke Maustaste

Wie Sie mittlerweile wissen, gehören solche Aktionen in eine Behandlungsmethode für die linke Maustaste, weshalb Sie jetzt eine passende Nachrichtenverarbeitungsfunktion für die *WM_LBUTTONDOWN*-Botschaft anlegen und dann deren Funktionsrumpf wie folgt bearbeiten:

***Listing 6.9* Behandlungsmethode für die linke Maustaste**

```
void CBezierkurvenView::OnLButtonDown(UINT nFlags,
CPoint point)
{
   // Mausposition in logische Koordinaten
// umrechnen
   CPoint MausPosition = point;
   CClientDC dc(this);

   // Gerätekontext zum Umrechnen der Werte
// vorbereiten
   RECT Client;
   GetClientRect(&Client);

   // Abbildungsmodus setzen
   dc.SetMapMode(MM_ISOTROPIC);

   // Kontextausmaße setzen
   dc.SetWindowExt(300, 300);
   dc.SetViewportExt(Client.right, Client.bottom);

   // Koordinaten umrechnen
   dc.DPtoLP(&MausPosition);

   // prüfen, ob einer der Knoten- oder
// Kontrollpunkte
   // angewählt wurde
   CBezierkurvenDoc* pDoc = GetDocument();
   ASSERT_VALID(pDoc);

   if (IsHit(MausPosition, pDoc->GetStartNode()))
   {
      m_bMovingStartNode = true;
      SetCapture();
      InvalidateRect(NULL);
   }
   else
   {
```

```
      if (IsHit(MausPosition,
pDoc->GetEndNode()))
      {
         m_bMovingEndNode = true;
         SetCapture();
         InvalidateRect(NULL);
      }
      else
      {
         if (IsHit(MausPosition,
pDoc->GetControlPoint1()))
         {
            m_bMovingControlPoint1 =
 true;
            SetCapture();
            InvalidateRect(NULL);
         }
         else
         {
            if (IsHit(MausPosition,
pDoc->GetControlPoint2()))
            {
               m_bMovingControlPoint2
= true;
               SetCapture();
               InvalidateRect(NULL);
            }
         }
      }
   }

   CView::OnLButtonDown(nFlags, point);
}
```

Erklärungen zu *OnLButtonDown*

Die *OnLButtonDown*-Methode gliedert sich in zwei wesentliche Abschnitte.

Umrechnung von Koordinaten

Zunächst wird festgelegt, welcher Mauspunkt tatsächlich getroffen wurde. Da an *OnLButtonDown* nur Pixelkoordinaten, die relativ zur linken oberen Ecke der Client Area des betreffenden Fensters angegeben sind, übermittelt werden, ist es notwendig, eine geeignete Umrechnung dieser Werte in logische Koordinaten durchzuführen.

Alternativ könnte man die Punktkoordinaten der Knoten- und Kontrollpunkte in Pixel zurückrechnen, was allerdings schon allein aus Performancegründen nötig ist.

Praktischerweise stellen die MFC zwei Methoden zum Umrechnen zwischen logischen und Gerätekoordinaten zur Verfügung: *LPtoDP* (Logic Point to Device Point) zum Umrechnen von logischen in Gerätekoordinaten und *DPtoLP* (Device Point to Logic Point) für den umgekehrten Weg.

Diese Methoden gehören zur *CDC*-Klasse, es muss also zunächst ein Gerätekontext zur Verwendung der Funktionen vorliegen. Diesen erzeugt man, wie in diesem Beispiel zu sehen, durch das Anlegen eines neuen *CClientDC*-Objekts, der einen Zeiger auf das zur Ansichtsklasse gehörende Fenster erhält.

Die Prototypen von *LPtoDP* und *DPtoLP* sehen so aus:

LPtoDP

```
void LPtoDP(
LPPOINT lpPoints,
      int nCount = 1 ) const;

void LPtoDP( LPRECT lpRect ) const;

void LPtoDP( LPSIZE lpSize ) const;
```

Parameter	Beschreibung
lpPoints	Zeiger auf ein Array von umzuwandelnden Punkten
nCount	Anzahl der Punkt im Array
lpRect	Zeiger auf ein umzuwandelndes Rechteck
lpSize	Zeiger auf ein umzuwandelndes Größenobjekt (*SIZE* oder *CSize*).

DPtoLP

```
void DPtoLP(
   LPPOINT lpPoints,
   int nCount = 1 ) const;
void DPtoLP(
   LPRECT lpRect ) const;
void DPtoLP(
   LPSIZE lpSize ) const;
```

Parameter	Beschreibung
lpPoints	Zeiger auf ein Array von umzuwandelnden Punkten
nCount	Anzahl der Punkt im Array

Parameter	Beschreibung
lpRect	Zeiger auf ein umzuwandelndes Rechteck
lpSize	Zeiger auf ein umzuwandelndes Größenobjekt (SIZE oder CSize).

Beide Methoden arbeiten dabei nach dem Prinzip, dass eine übergebene Koordinate in das jeweils andere Format umgerechnet wird, wobei die ursprünglichen Werte überschrieben werden.

Originalkoordinaten kopieren

Es bietet sich allgemein also an, eine Kopie der ursprünglichen Daten anzulegen, wie es im *OnLButtonDown*-Fall des Bezierkurven-Projekts auch gehandhabt wird.

Umrechnungstücken

Beachten Sie in jedem Fall, dass die Gerätekontexte in Methoden wie OnLButtonDown *nicht eine vorhergehende Abarbeitung von* OnPrepareDC *durchlaufen.*

Das heißt insbesondere, dass als Gerätekontext-Abbildungsmodus für den Device-Kontext, der hier ja auch erst angelegt wirkt, MM_TEXT *angenommen wird. Das führt natürlich zu Umrechnungsfehlern beziehungsweise zu einer Nichtveränderung der Werte – wir erinnern uns, dass bei* MM_TEXT *eine logische Einheit gerade einem Pixel entspricht.*

Obwohl es möglich wäre, OnPrepareDC *direkt zu verwenden, wird in diesem Projekt eine separate Umstellung des Abbildungsmodus für den neu erzeugten Gerätekontext verwendet, um die Problematik besser aufzuzeigen.*

Mit diesen Überlegungen im Hinterkopf ist die restliche Umrechnung trivial, es kann jetzt daran gegangen werden zu prüfen, ob ein relevanter Punkt (nämlich gerade ein Knoten- oder ein Kontrollpunkt) getroffen wurde.

Prüfen, ob ein Kontroll- oder Knotenpunkt ausgewählt wurde

Prüffunktion *IsHi*t

Um Schreibarbeit zu sparen, wird zur Bestimmung, ob ein Punkt des Dokuments getroffen wurde, eine neue Funktion *IsHit* eingeführt, die als Parameter den zu testenden Punkt (nämlich gerade die Position des Mausklicks in logischen Koordinaten) sowie den Dokumentpunkt, gegen den geprüft werden soll, erhält.

Deklariert wird die Funktion natürlich in *BezierkurvenView.h* innerhalb der Klassendeklaration von *CBezierkurvenView*:

```
   bool IsHit(POINT in_TestPoint,
POINT in_RegionCenter);
```

Die Implementation landet wieder in in *BezierkurvenView.cpp*:

Listing 6.10
Testen, ob ein Punkt in einer Region liegt

```
bool CBezierkurvenView::IsHit(POINT in_TestPoint,
 POINT in_RegionCenter)
{
   return (
      (in_TestPoint.x >= in_RegionCenter.x
- c_iKnotenRadius) &&
      (in_TestPoint.y >= in_RegionCenter.y -
 c_iKnotenRadius) &&
      (in_TestPoint.x <= in_RegionCenter.x +
c_iKnotenRadius) &&
      (in_TestPoint.y <= in_RegionCenter.y +
c_iKnotenRadius)
      );
}
```

Hier wird ein einfacher Bereichstest durchgeführt, um zu prüfen, ob ein Punkt getroffen wurde oder nicht. Die nachstehende Grafik veranschaulicht die dabei verwendeten Koordinaten:

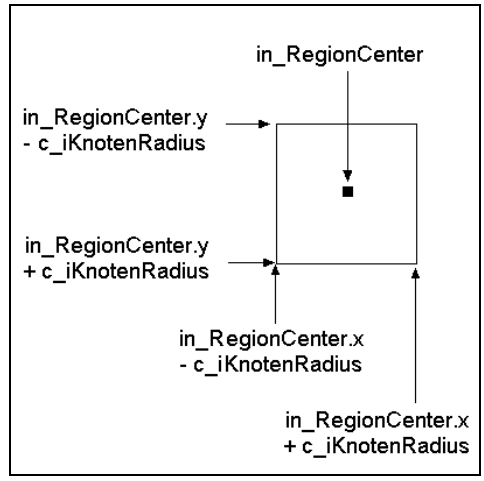


Abb. 6.5
Die Berechnung der Bereichskoordinaten

An dieser Stelle zeigt sich auch wieder der Vorteil, eine Konstante anstelle einer magischen Zahl für die Größe der jeweiligen Punkte verwendet zu haben – neben der besseren Lesbarkeit brauchen diese Zeilen nicht weiter editiert zu

werden, sollte eine Größenveränderung der Knoten- und Kontrollpunkte geplant sein.

IsHit liefert *true* zurück, wenn eine erfolgreiche Bereichsprüfung stattgefunden hat. In diesem Fall wird in *OnLButtonDown* die zugehörige Bewegungsstatus Variable auf *true* gesetzt, die Maus auf das Dokumentfenster fixiert – mittels *SetCapture* – und der komplette Bereich zum Neuzeichnen aufgefordert.

Neuzeichnen des Fensterinhalts

Das Neuzeichnen dient zum Darstellen des nun aktiven Elements, das ja rot gefärbt erscheinen soll. Es würde reichen, nur den tatsächlichen von dem Knotenpunkt eingenommenen Bereich neu zu zeichnen, da wir es hier aber ohnehin nur mit insgesamt fünf Zeichenoperationen zu tun haben, wäre die dadurch entstehende Einsparung nur als minimal zu bezeichnen.

Beenden einer Bewegung

Eine Bewegung wird dadurch beendet, dass der Benutzer die Maustaste loslässt. Für das Programm bedeutet dieses ein Zurücksetzen der booleschen Bewegungsstatusvariablen auf den Wert *false*.

Implementieren Sie die Funktion *OnLButtonUp* als Behandlungsmethode für *WM_LBUTTONUP*:

***Listing 6.11* Beenden einer Bewegung**

```
void CBezierkurvenView::OnLButtonUp(UINT nFlags, CPoint point)
{
   // alle Bewegungswerte zurücksetzen
   m_bMovingStartNode = false;
   m_bMovingEndNode = false;
   m_bMovingControlPoint1 = false;
   m_bMovingControlPoint2 = false;

   InvalidateRect(NULL);
   ReleaseCapture();

   CView::OnLButtonUp(nFlags, point);
}
```

Beim Beenden einer Bewegung soll außerdem die fixierte Maus wieder freigesetzt und der Bildbereich neu gezeichnet werden, um die normale Färbung des Knotenpunkts wieder herzustellen – diese Aufgabe übernehmen die Zeilen *InvalidateRect* und *ReleaseCapture*.

Zeichnen von markierten Knotenpunkten

Es wurde bereits viel über die rot markierten, derzeit ausgewählten Knotenpunkte gesagt. Bislang ist zwar das Grundgerüst dafür fertig gestellt worden, doch würde das Programm im jetzigen Status noch keine Färbung selbstständig durchführen.

Aufgabe zur Übung

Hierfür ist zunächst eine erneute Überarbeitung der *OnDraw*-Methode notwendig. Sie haben an dieser Stelle erneut die Möglichkeit sich selbst zu testen, und die fehlenden Zeilen in dieser Methode selbst einzusetzen – das ist eine gute Gedächtnisübung, um das zu wiederholen, was Sie über *GDI-Objekte* und dort insbesondere das *CPen*-Objekt gelernt haben.

Noch einmal die konkrete Aufgabenstellung:

- Markierte Objekte sollen rot dargestellt werden. Die Strichdicke soll zwei Pixel betragen.
- Nicht markierte Objekte sollen schwarz dargestellt werden. Die Strichdicke soll einen Pixel betragen.
- Beachten Sie, dass der ursprünglich eingestellte Stift am Ende der Methode wieder hergestellt werden muss.

Mögliche Lösung der Aufgabe

Im Folgenden steht eine mögliche Lösung für die gestellten Aufgabenpunkte, ich empfehle Ihnen jedoch, diese erst anzusehen, sobald Sie Ihre eigene Methode geeignet erweitert haben oder wirklich nicht weiter wissen:

```
void CBezierkurvenView::OnDraw(CDC* pDC)
{
   CBezierkurvenDoc* pDoc = GetDocument();
   ASSERT_VALID(pDoc);

   // Festlegen der Bezierkurven Knoten-
   // und kontrollpunkte
   POINT Bezier[4];

   Bezier[0] = pDoc->GetStartNode();
   Bezier[1] = pDoc->GetControlPoint1();
   Bezier[2] = pDoc->GetControlPoint2();
   Bezier[3] = pDoc->GetEndNode();

   // Stifte vorbereiten
   CPen BlackPen(PS_SOLID, 1, RGB(0, 0, 0));
   CPen RedPen(PS_SOLID, 2, RGB(255, 0, 0));

   // alten Stift speichern
   CPen *pOldPen = pDC->SelectObject(&BlackPen);

   // Zeichnen der Knotenpunkte
   if (m_bMovingStartNode)
pDC->SelectObject(&RedPen);

   pDC->Rectangle(Bezier[0].x – c_iKnotenRadius, Bezier[0].y –
c_iKnotenRadius,
         Bezier[0].x + c_iKnotenRadius,
```

```
            Bezier[0].y + c_iKnotenRadius);

      if (m_bMovingStartNode)
pDC->SelectObject(&BlackPen);

      if (m_bMovingEndNode)
pDC->SelectObject(&RedPen);

      pDC->Rectangle(Bezier[3].x – c_iKnotenRadius,
            Bezier[3].y – c_iKnotenRadius,
            Bezier[3].x + c_iKnotenRadius,
            Bezier[3].y + c_iKnotenRadius);

      if (m_bMovingEndNode)
pDC->SelectObject(&BlackPen);

      // Zeichnen der Kontrollpunkte
      if (m_bMovingControlPoint1)
pDC->SelectObject(&RedPen);

      pDC->Ellipse(Bezier[1].x – c_iKnotenRadius,
            Bezier[1].y – c_iKnotenRadius,
            Bezier[1].x + c_iKnotenRadius,
            Bezier[1].y + c_iKnotenRadius);

      if (m_bMovingControlPoint1)
pDC->SelectObject(&BlackPen);

      if (m_bMovingControlPoint2)
pDC->SelectObject(&RedPen);

      pDC->Ellipse(Bezier[2].x – c_iKnotenRadius,
            Bezier[2].y – c_iKnotenRadius,
            Bezier[2].x + c_iKnotenRadius,
            Bezier[2].y + c_iKnotenRadius);

      if (m_bMovingControlPoint2)
pDC->SelectObject(&BlackPen);

      pDC->PolyBezier(Bezier, 4);

      // Zurücksetzen des ursprünglichen Stifts
      pDC->SelectObject(pOldPen);
}
```

Erklärung zur Aufgabenlösung

Die Erweiterung der *OnDraw*-Methode beginnt damit, zwei neue Stifte zu definieren, von denen einer schwarz mit einer Dicke von einem Pixel (*BlackPen*), der andere rot mit einer Dicke von zwei Pixeln (*RedPen*) ist. Weiterhin wird der ursprünglich eingestellte Stift des Gerätekontexts gespeichert.

Nun wird bei jedem Zeichenvorgang geprüft, ob ein markiertes Element vorliegt und gegebenenfalls auf den roten Stift gewechselt. Nach der Operation wird der Stift gegebenenfalls auf den schwarzen Stift zurückgesetzt.

Das Ende der Methode stellt den ursprünglichen Zustand wieder her, indem der alte Stift wieder eingesetzt wird.

Bewegen der Kontroll- und Knotenpunkte

Ein kurzes Austesten des Programms zeigt, dass die Knoten- und Kontrollpunkte nun rot gefärbt werden, solange der Benutzer sie mit dem Mauszeiger anwählt und die linke Maustaste gedrückt hält. Das Loslassen der Maustaste setzt die ursprüngliche Farbe wieder ein.

Weitere Benutzerinteraktion

Was noch fehlt, ist das eigentliche Bewegen der Objekte – die Behandlung dieser Aktion gehört in eine Nachrichtenbearbeitungsmethode zu *WM_MOUSEMOVE*.

Fügen Sie eine passende Methode in die *CBezierkurvenView*-Klasse ein und ergänzen Sie sie gemäß den nachstehenden Zeilen:

```
void CBezierkurvenView::OnMouseMove(UINT nFlags,
CPoint point)
{
   // findet gerade eine Bewegung statt?
   if ( m_bMovingStartNode || m_bMovingEndNode ||
         m_bMovingControlPoint1 ||
m_bMovingControlPoint2 )
   {
      // Zugriff auf Dokument vorbereiten
      CBezierkurvenDoc* pDoc = GetDocument();
      ASSERT_VALID(pDoc);

      // aktuelle Mausposition ermitteln
      // Mausposition in logische Koordinaten
// umrechnen
      CPoint MausPosition = point;
      CClientDC dc(this);

      // Gerätekontext zum Umrechnen der Werte
// vorbereiten
      RECT Client;
      GetClientRect(&Client);
```

```
      // Abbildungsmodus setzen
      dc.SetMapMode(MM_ISOTROPIC);

      // Kontextausmaße setzen
      dc.SetWindowExt(300, 300);
      dc.SetViewportExt(Client.right,
 Client.bottom);

      // Koordinaten umrechnen
      dc.DPtoLP(&MausPosition);

      // Mausposition begrenzen
      if (MausPosition.x < 0)
MausPosition.x = 0;

      if (MausPosition.x > 300)
MausPosition.x = 300;

      if (MausPosition.y < 0)
MausPosition.y = 0;

      if (MausPosition.y > 300)
MausPosition.y = 300;

      // nun ermitteln, welche Bewegung
// durchgeführt wird
      if (m_bMovingStartNode)
      {
         // Startknoten auf neue Position
// setzen
         pDoc->SetStartNode(MausPosition);
      }

      if (m_bMovingEndNode)
      {
         // Startknoten auf neue Position
// setzen
         pDoc->SetEndNode(MausPosition);
      }

      if (m_bMovingControlPoint1)
      {
         // Startknoten auf neue Position
// setzen
         pDoc
->SetControlPoint1(MausPosition);
      }
```

```
      if (m_bMovingControlPoint2)
      {
         // Startknoten auf neue Position
// setzen
         pDoc
->SetControlPoint2(MausPosition);
      }

      // neu zeichnen
      InvalidateRect(NULL);
   }

   // Standardbehandlung aufrufen
   CView::OnMouseMove(nFlags, point);
}
```

Erklärungen zu *OnMouseMove*

Den Anfang von *OnMouseMove* bildet eine Abfrage, ob überhaupt derzeit eine Knoten- oder Kontrollpunktbewegung vorliegt. Ist das nicht der Fall, kann die Funktion mit einem Aufruf der Standardbehandlung verlassen werden.

Anderenfalls folgt eine Umrechnung der aktuellen Mausposition in logische Koordinaten, wie sie es bereits in *OnLButtonDown* gesehen haben.

Zuschneiden von berechneten Koordinaten auf gültigen Wertebereich

Der ermittelte Wert wird dann noch auf den gültigen Wertebereich beschnitten, da die Maus ja den Fensterbereich verlassen und dann entsprechend kleinere oder größere Argumente zurückliefern würde.

Je nachdem, welcher Knoten- oder Kontrollpunkt nun gerade bewegt wird, setzt eine passende Dokumentzugriffsfunktion die neuen Werte für gerade dieses Objekt.

Den Abschluss bildet dann ein Aufruf von *InvalidateRect*, der zum Neuzeichnen der Ansicht auffordert und somit sofort die vollzogene Veränderung visualisiert.

Kompilieren Sie die neue Version und testen Sie das Programm aus.

Austesten des bisherigen Programms

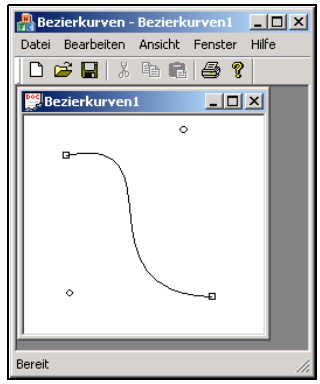
Probieren Sie, die Kontroll- und Knotenpunkte anzuwählen und verschieben Sie die Objekte innerhalb des Fensters. Beobachten Sie, wie sich die Kurve in ihrer Form verändert und beachten sie auch, dass die Rechtecke und Ellipsen am Fensterrand festgehalten werden.

Verändern Sie die Größe des Ansichtsfensters und prüfen Sie, ob die Darstellung verzerrungsfrei bleibt. An dieser Stelle ist auch schön zu sehen, dass nur je nach Fenstergröße nur ein relativ kleiner Bereich für die tatsächliche verzerrungsfreie Abbildung genutzt wird: ziehen Sie das Fenster relativ niedrig aber

recht breit auf und versuchen Sie, einen der Knoten- oder Kontrollpunkte nach rechts zu verschieben: Sie werden recht bald an eine unsichtbare Grenze stoßen.

Bei Ihren Versuchen ergibt sich dann eine Abbildung ähnlich der folgenden, womit die Applikation prinzipiell schon fertig ist:

Abb. 6.6
Eine veränderte Bezierkurve



Hinzufügen einer neuen Ansichtsklasse

Wenn ich sage „prinzipiell schon fertig", ist damit gemeint, dass in der Einführung ja davon die Rede war, eine weitere Ansichtsklasse für die Dokumentansicht zu implementieren.

Anlegen einer neuen Ansichtsklasse

Die letzten Seiten dieses Kapitels sollen sich nun mit diesem Vorhaben beschäftigen. Bevor es allerdings um die Details der tatsächlichen Implementierung geht – die hier und da etwas kniffelig sind – sollen zunächst vorbereitende Maßnahmen getätigt werden.

Als Erstes ist hier das Anlegen einer neuen Klasse zu nennen, die später als Ansichtsklasse für die Applikation dienen soll.

Wählen Sie dazu aus dem Menü *Projekt* den Punkt *Neue Klasse*. Es öffnet sich ein Dialogfeld, in dem Sie eintragen können, was für eine Art von Klasse in das Bezierkurven-Projekt eingefügt werden soll.

Suchen Sie aus dem Ordner *Visual C++* den Unterordner *MFC* und dort den Typ *MFC Class* heraus:

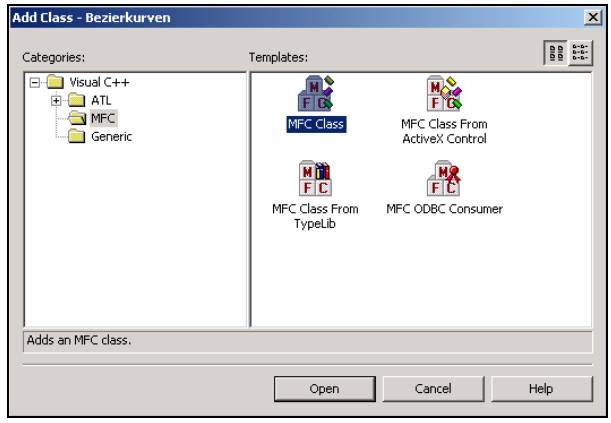


Abb. 6.7
Hinzufügen einer neuen Klasse

Bestätigen Sie die Auswahl mit einem Klick auf *Open*. Es öffnet sich ein weiteres Dialogfeld, in das Sie als Klassenname *CBezierkurvenInfoAnsicht* eintragen und als Basisklasse *CListView* auswählen:

Klasse anlegen via Assistent

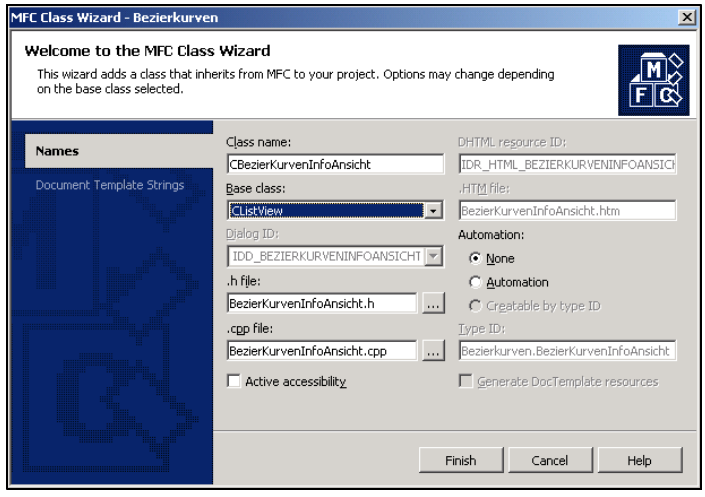


Abb. 6.8
Einstellen der Klassenattribute

Die neue Klasse

Die gerade eben erzeugte Klasse soll dazu dienen, eine weitere Ansicht der Bezierkurven-Dokumente zu ermöglichen.

Da bereits eine grafische Darstellung existiert, wäre es sicherlich interessant, die aktuellen Koordinaten der einzelnen Punkte übersichtlich präsentiert ablesen zu können. Hierzu bietet es sich eine, eine tabellarische Übersicht zu wählen.

Fähigkeiten der neuen Klasse

Ansichtsklassen sind, wie Sie wissen, von *CView* abgeleitet. Es gibt allerdings einige etwas fortschrittlichere Klassen, die schon im MFC Framework existieren und ebenfalls von *CView* abgeleitet sind.

Hierzu gehört beispielsweise *CListView*, das gerade eine Listenansicht zur Verfügung stellt, wie sie in diesem konkreten Fall für die Bezierkurvendaten nützlich sein könnte.

Nützliche Klassen im MFC Framework

Um herauszufinden, welche Arten von Ansichtsklassen (oder sonstigen Klassen) im MFC Framework bereits existieren, ist es unumgänglich, dass Sie sich einmal die Zeit nehmen, die einzelnen Klassen der MFC Stück für Stück in der Online-Hilfe anzuschauen, um einen Überblick darüber zu gewinnen, was alles schon existiert und was erst noch selbst entwickelt werden muss.

Häufig findet man bei diesem Vorgehen eine Reihe von wertvollen Klassen, denen nur kleine Details fehlen, um Ihrem Verwendungszweck zu entsprechen.

Leiten Sie eine Klasse davon ab, ergänzen Sie sie um die fehlenden Methoden und schreiben Sie sich auf diese Weise mit der Zeit selbst eine kleine Bibliothek an häufig verwendeten Klassen – der Aufwand wird sich lohnen.

Bekanntmachen der neuen Ansichtsklasse

Bevor es um die Darstellung der Dokumentinformationen durch die neue Ansichtsklasse gehen soll, wollen wir zunächst betrachten, wie die Anwendung Notiz davon erhält, dass eine weitere Ansichtsklasse zur Verfügung steht.

Bekannte Ansichts- und Dokumentenklassen

Nun, vereinfacht gesprochen, wird der Anwendung bei der Initialisierung gesagt, welche Arten von Dokumenten verwendet werden, mit welcher Ansicht sie verknüpft sind und in welcher Art von Fenster sie darzustellen sind.

Für den bisherigen Fall der *CBezierkurvenView*-Ansichtsklasse, müssten also die nachstehenden Informationen zusammengetragen werden:

- Verwendetes Dokument: *CBezierkurvenDoc*
- Verwendete Ansichtsklasse: *CBezierkurvenView*
- Verwendetes Fenster: *CChildFrame*

Beim Erzeugen einer Applikation durch den Anwendungsassistenten werden diese drei Klassen automatisch zu einem Paket zusammengeschnürt und sozusagen als Team bekannt gemacht.

Wenn Sie nun ein neues Dokument vom Typ *CBezierkurvenDoc* öffnen, wird ein *CChildFrame* Fenster erzeugt, in dessen Client-Bereich die *CBezierkurvenView*-Ansichtsklasse ihre Arbeit verrichtet.

Möglichkeit, mehrere Dokumentenklassen zu verwenden

Sie haben es vielleicht schon zwischen den Zeilen gelesen: Es ist in MDI-Anwendungen nicht nur möglich, ein Dokument mit verschiedenen Ansichtsklassen zu verknüpfen, Sie können auch mehrere Dokumenttypen in einer MDI-Anwendung definieren und verwenden. Dieser Fall soll im Rahmen dieses Buch aber nicht weiter besprochen werden.

Zurückkehrend zum gewünschten Ziel, eine zweite Ansichtsklasse mit dem Dokument zu koppeln, muss ein zweites Team, bestehend aus *CBezierkurvenDoc*, *CBezierkurvenInfoAnsicht* und *CChildFrame* zusammengestellt werden – im Rahmen der MFC wird dieses Team dann als Vorlage bezeichnet.

Eine neue Vorlage

Eine Vorlage selbst ist ein Objekt der Klasse *CMultiDocTemplate*, sodass es sich zunächst anbietet, eine entsprechende Variable in die Anwendungsklasse der Applikation, *CBezierkurvenApp*, einzubinden. Fügen Sie die nachstehende Zeile in die Klassendeklaration in der Datei *BezierkurvenApp.h* ein:

```
CMultiDocTemplate *m_pTemplateInfo;
```

Bevor die Vorlage definiert werden kann, muss zunächst die neue Ansichtsklassen-Headerdatei per *Include*-Anweisung in das Implementationsdatei von *CBezierkurvenApp* eingefügt werden.

Ergänzen Sie die Datei also um die Zeile

```
#include "BezierKurvenInfoAnsicht.h"
```

Bekanntmachen der Vorlage

Notwendige Änderungen in *InitInstance*

Nun geht es ans Eingemachte. Das Bekanntmachen – oder genauer genommen Vorbereiten – der neuen Ansichtsklasse wird in der *InitInstance*-Methode von *CBezierkurvenApp* durchgeführt.

Wir haben diese Methode bislang noch nicht näher untersucht, höchste Zeit also, sich mit ihr vertraut zu machen (die von Ihnen einzufügenden Zeilen sind fett hervorgehoben):

```
BOOL CBezierkurvenApp::InitInstance()
{
   // InitCommonControls() ist für Windows XP
// erforderlich, wenn ein Anwendungsmanifest
   // die Verwendung von ComCtl32.dll Version 6
// oder höher zum Aktivieren
   // von visuellen Stilen angibt. Ansonsten treten
```

```
// beim Erstellen von Fenstern Fehler auf.
   InitCommonControls();

   CWinApp::InitInstance();

   // OLE-Bibliotheken initialisieren
   if (!AfxOleInit())
   {
      AfxMessageBox(IDP_OLE_INIT_FAILED);
      return FALSE;
   }

AfxEnableControlContainer();

// Standardinitialisierung
   // Wenn Sie diese Features nicht verwenden und
// die Größe
// der ausführbaren Datei verringern möchten,
// entfernen Sie
   // die nicht erforderlichen
// Initialisierungsroutinen.
   // Ändern Sie den Registrierungsschlüssel unter
// dem Ihre Einstellungen gespeichert sind.
   // TODO: Ändern Sie diese Zeichenfolge
// entsprechend,
   // z.B. zum Namen Ihrer Firma oder Organisation.
   SetRegistryKey(_T("Vom lokalen Anwendungs-
Assistenten generierte Anwendungen"));

   LoadStdProfileSettings(4);  // Standard INI-
//Dateioptionen laden (einschließlich MRU)
   // Dokumentvorlagen der Anwendung registrieren.
// Dokumentvorlagen
//  dienen als Verbindung zwischen Dokumenten,
// Rahmenfenstern und Ansichten.
   CMultiDocTemplate* pDocTemplate;
   pDocTemplate = new
CMultiDocTemplate(IDR_BezierkurvenTYPE,
      RUNTIME_CLASS(CBezierkurvenDoc),
      RUNTIME_CLASS(CChildFrame),
// Benutzerspezifischer MDI-Child-Rahmen
      RUNTIME_CLASS(CBezierkurvenView));
   AddDocTemplate(pDocTemplate);

   m_pTemplateInfo = new CMultiDocTemplate
      (IDR_BezierkurvenTYPE,
      RUNTIME_CLASS(CBezierkurvenDoc),
      RUNTIME_CLASS(CChildFrame),
      RUNTIME_CLASS(CBezierKurvenInfoAnsicht));
```

```
   // Haupt-MDI-Rahmenfenster erstellen
   CMainFrame* pMainFrame = new CMainFrame;
   if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
      return FALSE;

   m_pMainWnd = pMainFrame;

   // Rufen Sie DragAcceptFiles nur auf, wenn eine
// Suffix vorhanden ist.
   //  In einer MDI-Anwendung ist dies unmittelbar
// nach dem Festlegen von m_pMainWnd
// erforderlich
   // Befehlszeile parsen, um zu prüfen auf
// Standardumgebungsbefehle DDE, Datei offen
   CCommandLineInfo cmdInfo;
   ParseCommandLine(cmdInfo);

// Verteilung der in der Befehlszeile
// angegebenen Befehle. Es wird FALSE
// zurückgegeben, wenn
   // die Anwendung mit /RegServer, /Register,
// /Unregserver oder /Unregister gestartet
// wurde.
   if (!ProcessShellCommand(cmdInfo))
      return FALSE;

   // Das Hauptfenster ist initialisiert und kann
// jetzt angezeigt und aktualisiert werden.
   pMainFrame->ShowWindow(m_nCmdShow);
   pMainFrame->UpdateWindow();
   return TRUE;
}
```

Erläuterungen zu *InitInstance*

Die ersten Zeilen der Methode beziehen sich auf Besonderheiten unter Windows XP, die Initialisierung von OLE-Containern sowie das Einladen der letztbekannten Einstellungen aus der Registry (insbesondere die Namen der vier zuletzt geöffneten Dokumente) – allesamt Themenbereiche, die den Rahmen dieses Buches sprengen würden und daher leider nicht weiter behandelt werden können. Es sei daher auf die entsprechenden Funktionsbeschreibungen in der Online-Hilfe verwiesen.

Bekanntmachen der neuen Ansichtsklasse in *InitInstance*

Der interessante Teil folgt allerdings auch erst nach diesen einleitenden Zeilen, nämlich gerade die Initialisierung der Standardvorlage gefolgt von unserer eigenen, zweiten Vorlage.

Das Anlegen derselben geschieht über eine *new*-Operation mit vier übergebenen Parametern. Neben der ID des zu verwendenden Menüs (Sie erinnern sich, dass einzelne Dokumentfenster eigenständige Menüs in der Menüleiste des Mainframe-Fensters anzeigen können – hier wird die betreffende ID eingetragen) wird der Name der Dokumentenklasse, der der Kindfensterklasse und abschließend der Ansichtsklassenname.

Die vom Anwendungsassistenten bereits integrierte Vorlage wird direkt an die Anwendung angehängt und fortan von ihr verwaltet. Sie braucht daher nicht explizit gelöscht zu werden.

Separate Verwaltung der Ansichtsklasse

Unsere eigene neue Ansichtsklasse soll separat verwaltet werden und wird daher an den in der Klasse eigens dafür angelegten Zeiger übergeben.

Sie halten jetzt also ein Vorlagenobjekt in Händen, das beim Erzeugen eines neuen Fensters als Grundlage herangezogen werden kann.

Abräumen der neuen Vorlage

Da die neue Vorlage manuell verwaltet wird, muss sie zu einem bestimmten Zeitpunkt auch wieder zerstört werden. Das tut man zweckmäßigerweise in der *InitInstance* entgegengesetzten Methode *ExitInstance*.

Fügen Sie eine überschreibende Variante die Methode in die Klasse *CBezierkurvenApp* ein:

```
int CBezierkurvenApp::ExitInstance()
{
   // TODO: Add your specialized code here and/or
// call the base class
   delete (m_pTemplateInfo);
   return CWinApp::ExitInstance();
}
```

Öffnen eines Fensters mit der neuen Vorlage

Die neue Ansichtsklasse für neue Fenster verwenden

Es wurde nun zwar eine neue Vorlage angelegt, doch kann diese bislang noch nicht zum Vorschein gebracht werden: Hierzu fehlt noch ein passender Aufruf, denn standardmäßig wird die erste via *AddDocTemplate* an die Anwendung angehängte Vorlage zum Darstellen von Dokumenten verwendet. Das gilt übrigens auch, wenn Sie weitere Ansichten des Dokuments über *Fenster* > *Neues Fenster* öffnen.

Es wäre also zweckmäßig, einen weiteren Menüpunkt in das Fenster Menü der MDI-Anwendung einzufügen, das etwa *Neues Info Fenster* heißen könnte. Öffnen Sie dazu den Ressourceneditor und editieren dort das Menü *IDR_BezierkurvenTYPE*. Fügen Sie einen Menüpunkt namens *Neues Info Fenster* hinzu.

Behalten Sie dessen zugewiesene ID bei und fügen Sie eine Behandlungsmethode für den neuen Punkt in die *CMainFrame*-Klasse ein:

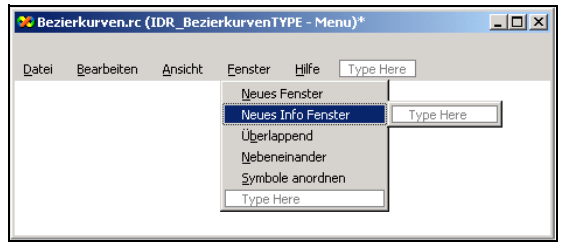


Abb. 6.9
Der neue Menüpunkt für das Bezierkurven-Programm

Behandlungsmethode für den neuen Menüpunkt

Kopieren und Anpassen von Behandlungsmethoden

Der Inhalt der Behandlungsmethode ist aus *CMDIFrameWnd::OnNewWindow* kopiert und den Bedürfnissen der neuen Klasse angepasst. Es ist oftmals hilfreich, die Sources der bestehenden MFC-Klassen zu durchstöbern, um herauszufinden, wie bestimmte Details realisiert wurden.

In diesem Fall ist es durch die Vorgabe der *CMDIFrameWnd*-Klasse ein Leichtes, zum gewünschten Ergebnis zu kommen:

```
void CMainFrame::OnFensterInfofenster()
{
   // aktives Kindfenster ermitteln
   CMDIChildWnd *pActiveChild = MDIGetActive();
   CDocument* pDocument;

   // prüfen, ob derzeit ein Fenster aktiv ist
   if (pActiveChild == NULL ||
      (pDocument = pActiveChild
->GetActiveDocument()) == NULL)
   {
      AfxMessageBox("Kein Kindfenster aktiv.");
      return;
   }

   // neues Fenster erzeugen
   CDocTemplate* pTemplate =
      ((CBezierkurvenApp*) AfxGetApp())
->m_pTemplateInfo;

   CFrameWnd* pFrame = pTemplate
->CreateNewFrame(pDocument,
      pActiveChild);
```

```
   if (pFrame == NULL)
   {
      AfxMessageBox("Konnte neues Fenster nicht
 erzeugen!");
      return;
   }

   pTemplate->InitialUpdateFrame(pFrame,
 pDocument);
}
```

Im ersten Teil der Methode wird geprüft, ob derzeit überhaupt ein Kindfenster (sprich ein Dokument) aktiv ist. Ist dem nicht so, kann natürlich auch kein zugehöriges Infofenster geöffnet werden.

Anderenfalls wird mittels der *CreateNewFrame*-Methode ein neues Fenster erzeugt, das auf Basis der vorhin erzeugten Vorlage fußt:

```
CFrameWnd* CreateNewFrame(
   CDocument* pDoc,
   CFrameWnd* pOther
);
```

Parameter	Beschreibung
pDoc	Dokument, mit dem das neue Fenster verknüpft sein soll.
pOther	Fenster, auf dem das neue Fenster basieren soll. Es wird im Titel durch eine vorangestellte Nummer gekennzeichnet.
Rückgabewert	Zeiger auf das neu erzeugte Fenster, oder NULL bei Auftreten eines Fehlers.

Ein abschließender Aufruf von *InitialUpdateFrame* initialisiert das Fenster entsprechend.

Erneutes Austesten

Wenn Sie das Programm in der jetzigen Form ausführen, werden Sie feststellen, dass die neue Ansichtsklasse noch nicht sehr viel tut – allerdings ist dieses ein gutes Zeichen, denn offensichtlich wird hier ein Fenster mit einer neuen Ansichtsklasse geöffnet, was ja genau das ist, was wir möchten.

Implementation der alternativen Ansichtsklasse

Was jetzt noch fehlt ist eine Implementation der neuen Ansichtsklassendarstellung.

Wie schon gesagt, basiert diese auf einer *CListView*-Klasse, die zur Darstellung von Tabellen geeignet ist.

Initialisierung von Tabellenansichten

Bevor mit einer solchen Tabellendarstellung gearbeitet werden kann, ist es erforderlich, dass die Tabelle geeignet angelegt wird – so ist ihr beispielsweise mitzuteilen, welche Arten von Spalten sie hat und wie breit diese sein sollen.

Solche initialisierenden Angaben gehören in die *OnInitialUpdate*-Methode der *CBezierKurvenInfoAnsicht*-Klasse in der Datei *BezierkurvenInfoAnsicht.cpp*:

Listing 6.12 **Initialisierung der Listenansicht**

```
void CBezierKurvenInfoAnsicht::OnInitialUpdate()
{
   CListView::OnInitialUpdate();

   // Zugriff auf Liste holen
   CListCtrl &refCtrl = GetListCtrl();

   // Tabellenkopf vorbereiten
   refCtrl.InsertColumn(0, "Element");
   refCtrl.InsertColumn(1, "X");
   refCtrl.InsertColumn(2, "Y");

   // Spaltenbreiten festlegen
   for (int i=0;i<3;i++)
   {
      refCtrl.SetColumnWidth(i, 100);
   }

   // Elemente in Liste einfügen
   AddItemToList(refCtrl, 0, 0, "Startknoten");
   AddItemToList(refCtrl, 1, 0, "Kontrollpunkt 1");
   AddItemToList(refCtrl, 2, 0, "Kontrollpunkt 2");
   AddItemToList(refCtrl, 3, 0, "Endknoten");

   // Werte für die Elemente setzen
   ShowNewValues();
}
```

Vorbereiten der Liste

Zunächst wird eine Referenz auf ein Listenkontrollobjekt erzeugt. Das Listenkontrollobjekt repräsentiert die Liste, die innerhalb der Ansicht darzustellen ist. Es ermöglicht den Zugriff auf die Listenstruktur (Typ der Spalten und so weiter) und die in die Liste eingetragenen Elemente.

Ergänzen der Liste um passende Spalten für den vorliegenden Anwendungsfall

Um die Liste geeignet vorzubereiten, müssen Spalten in die Liste eingefügt werden, die in ihrer Ausgangsform leer ist.

Die Methode *InsertColumn* hat den folgenden Prototyp:

```
int InsertColumn(
   int nCol,
   LPCTSTR lpszColumnHeading,
   int nFormat = LVCFMT_LEFT,
   int nWidth = -1,
   int nSubItem = -1
);
```

Parameter	Beschreibung
nCol	Index (bei 0 beginnend) der neuen Spalte
lpszColumnHeading	Überschrift der Spalte
nFormat	Ausrichtung der Spalte. Gültige Werte sind *LVCFMT_LEFT* für linksbündige Ausgabe, *LVCFMT_CENTER* für zentrierte Ausgabe und *LVCFMT_RIGHT* für rechtsbündige Ausgabe.
nWidth	Breite der Spalte in Pixeln
nSubItem	Index des Unterpunkts innerhalb dieser Spalte
Rückgabewert	Index der neuen Spalte oder -1 beim Auftreten eines Fehlers

Es werden also drei Spalten mit den Überschriften (von links nach rechts) Element, X und Y in die Liste eingetragen.

Breite der Spalten verändern

Im nächsten Schritt werden die Breiten der einzelnen Spalten festgelegt. Hierbei kommt die *CListCtrl*-Methode *SetColumnWidth* zum Einsatz:

```
BOOL SetColumnWidth(
   int nCol,
   int cx
);
```

Parameter	Beschreibung
nCol	Index der Spalte (beginnend bei 0), deren Breite zu setzen ist.
cx	Zu setzende Breite der Spalte in Pixeln.

Parameter	Beschreibung
Rückgabewert	Wert ungleich 0 bei Erfolg, sonst 0.

Die beiden Methoden *AddItemToList* und *ShowNewValues* sind selbstgeschriebene Methoden, deren Beschreibung aber zunächst noch zurückgestellt werden soll – fürs Erste sei gesagt, dass sie die linke Spalte der Tabelle (die mit Element überschrieben ist) sowie die derzeitigen Werte der Knoten- und Kontrollpunkte des aktiven Elements in die Liste eintragen.

Weitere Initialisierungen

Listentyp festlegen

Bevor es um die konkreten Implementationen der selbst geschriebenen Methoden geht, muss noch die Initialisierung der Liste vollendet werden. Die restlichen benötigten Zeilen gehören in die Behandlungsmethode für die Fensternachricht *WM_CREATE*, die Sie an dieser Stelle in die Klasse *CBezierKurvenInfoAnsicht* einfügen:

Listing 6.13 **Weitere Initialisierungen**

```
int CBezierKurvenInfoAnsicht::OnCreate(LPCREATESTRUCT
lpCreateStruct)
{
   // Stil anpassen
   lpCreateStruct->style |= LVS_REPORT |
 LVS_SINGLESEL |
                LVS_NOSORTHEADER |
 LVS_SHOWSELALWAYS;

   // ListView erzeugen
   if (CListView::OnCreate(lpCreateStruct) == -1)
      return -1;

   // alles ok
   return 0;
}
```

Hier wird die Art und Weise festgelegt, in der die Tabelle angezeigt werden soll. Die einzelnen möglichen Flags für *lpCreateStruct->style* entnehmen Sie der Online-Hilfe, an dieser Stelle sei nur gesagt, dass die Einstellungen dafür sorgen, dass die Tabelle mit einer Überschrift in nicht sortierter Reihenfolge dargestellt wird, was in unserem Fall prinzipiell egal ist, da die Überschriften Element, X und Y ohnehin schon alphabetisch korrekt sortiert sind.

Implementationen für die Listenelementaktualisierung

Es fehlen noch drei wichtige Methoden zum Aufbauen und Aktualisieren der Liste, die allesamt selbst geschrieben werden. Fügen Sie daher als Erstes die folgenden Methodendeklarationen in der Datei *BezierkurvenInfoAnsicht.h* in die Klasse *CBezierkurvenInfoAnsicht* ein:

Listing 6.14
Neue Methoden

```
void AddItemToList(CListCtrl &in_Ctrl, int in_nLine,
 int in_nColumn, char *in_lpszText);

void SetItem(CListCtrl &in_Ctrl, int in_nLine, int
in_nColumn, int in_Value);

void ShowNewValues();
```

AddItemToList

AddItemToList soll dazu dienen, neue Elemente in die Liste aufzunehmen. Das sind insbesondere die vier Beschriftungselemente in der linken Spalte (dort soll indizierend stehen: Startknoten, Kontrollpunkt 1, Kontrollpunkt 2 und Endknoten). Mit dem Einfügen dieser Elemente wird gleichzeitig jeweils eine neue Listenzeile in die Liste aufgenommen.

Daten in die Liste einfügen

Die Implementation ist im Folgenden abgedruckt:

Listing 6.15
AddItemToList

```
void CBezierKurvenInfoAnsicht::AddItemToList(CListCtrl
&in_Ctrl, int in_nLine, int in_nColumn, char
*in_lpszText)
{
   LV_ITEM lvItem;
   lvItem.mask = LVIF_TEXT;
   lvItem.iItem = in_nLine;
   lvItem.iSubItem = in_nColumn;
   lvItem.pszText = in_lpszText;
   in_Ctrl.InsertItem(&lvItem);
}
```

Die Funktion erhält als Parameter eine Referenz auf das Listenkontrollobjekt, die Zeile, in die das neue Element einzufügen ist, die passende Spalte sowie den einzutragenden Text.

Diese Informationen werden in eine *LV_ITEM*-Struktur eingetragen, deren genauen Inhalt Sie wegen ihres Umfangs der Online-Hilfe entnehmen.

Die abschließende *InsertItem*-Zeile fügt das neue Feld in die Liste ein.

ShowNewValues

Angezeigte Werte aktualisieren

ShowNewValues aktualisiert die Anzeige dahingehend, dass die neuen Werte von Kontroll- und Knotenpunkten aus dem Dokument ausgelesen und in der Liste dargestellt werden:

Listing 6.16 ShowNewValues

```
void CBezierKurvenInfoAnsicht::ShowNewValues()
{
   // Zugriff auf Liste holen
   CListCtrl &refCtrl = GetListCtrl();

   // Werte für die Elemente setzen
   POINT CurrentPoint = GetDocument()
->GetStartNode();
   SetItem(refCtrl, 0, 1, CurrentPoint.x);
   SetItem(refCtrl, 0, 2, CurrentPoint.y);

   CurrentPoint = GetDocument()
->GetControlPoint1();
   SetItem(refCtrl, 1, 1, CurrentPoint.x);
   SetItem(refCtrl, 1, 2, CurrentPoint.y);

   CurrentPoint = GetDocument()
->GetControlPoint2();
   SetItem(refCtrl, 2, 1, CurrentPoint.x);
   SetItem(refCtrl, 2, 2, CurrentPoint.y);

   CurrentPoint = GetDocument()->GetEndNode();
   SetItem(refCtrl, 3, 1, CurrentPoint.x);
   SetItem(refCtrl, 3, 2, CurrentPoint.y);
}
```

Das Eintragen der neuen Werte in die Liste wird über die dritte und letzte der selbst geschriebenen Funktionen realisiert: *SetItem*.

SetItem

Einen spezifischen Wert der Tabelle ändern

SetItem aktualisiert den Inhalt eines Listenelements. Es erhält als Parameter eine Referenz auf ein Listenkontrollobjekt, die Position (Zeile und Spalte) des zu aktualisierenden Felds sowie einen Integerwert, der in unserem Fall gerade die neue Koordinate des Kontroll- beziehungsweise Knotenpunkts angibt:

Listing 6.17 SetItem

```
void CBezierKurvenInfoAnsicht::SetItem(CListCtrl
&in_Ctrl, int in_nLine, int in_nColumn, int in_nValue)
{
   LV_ITEM lvItem;
   char lpszConvert[10];
   lvItem.mask = LVIF_TEXT;
```

```
   lvItem.iItem = in_nLine;
   lvItem.iSubItem = in_nColumn;
   itoa(in_nValue, lpszConvert, 10);
   lvItem.pszText = lpszConvert;

   in_Ctrl.SetItem(&lvItem);

}
```

Der Ablauf ist analog zu *AddItemToList*, der Unterschied besteht lediglich darin, dass zum Schluss statt *InsertItem* die Methode *SetItem* aufgerufen wird, die kein neues Element in die Liste einfügt, sondern ein bestehendes verändert.

Weitere Änderungen in der neuen Ansichtsklasse

Zugriff auf das Dokument von der neuen Ansichtsklasse aus

Ein Problem gibt es freilich noch: Die neue Ansichtsklasse hat derzeit noch keinen Zugriff auf das Dokument. Das liegt daran, dass der Klassenassistent, der die neue Klasse angelegt hat, nicht wissen konnte, zu welchem Dokumententyp *CBezierkurvenInfoAnsicht* gehören soll – das ist eine weitere Grundregel bei der Arbeit mit den MFC: Wenn nicht klar ist, welche Objekte eine Klasse kennen muss, kennt sie im Zweifelsfall gar keine Objekte. Das ist auch sinnvoll, da somit unnütze Daten aus den Klassen herausgehalten werden.

Um dem Problem der unbekannten Dokumentenklasse auf den Leib zu rücken, wird jetzt die Deklarationsdatei *BezierkurvenInfoAnsicht.h* noch einmal erweitert, nämlich zunächst um eine hinzugefügte *Include*-Zeile:

```
#include "BezierkurvenDoc.h"
```

Des Weiteren ist eine *GetDocument*-Methode wie in der regulären *CBezierkurvenView*-Klasse einzuflechten:

```
CBezierkurvenDoc* GetDocument() const;
```

Abschließend fehlt noch die Unterscheidung zwischen Debug und Release Version der *GetDocument*-Methode. Das ist zwar nicht zwingend erforderlich, bleibt aber konform mit den übrigen Klassen.

Sie erinnern sich vielleicht, dass die Releaseversion der *GetDocument*-Methode inline definiert wurde. So soll es auch hier der Fall sein:

```
#ifndef _DEBUG  // Debugversion in
   // BezierkurvenInfoAnsicht.cpp

inline CBezierkurvenDoc*
CBezierkurvenView::GetDocument() const
   { return
 reinterpret_cast<CBezierkurvenDoc*>(m_pDocument); }
#endif
```

Die Debug-Version hingegen gehört in die Datei *BezierkurvenInfoAnsicht.cpp*:

```
CBezierkurvenDoc*
CBezierKurvenInfoAnsicht::GetDocument() const
// Nicht-Debugversion ist inline
{
   ASSERT(m_pDocument
->IsKindOf(RUNTIME_CLASS(CBezierkurvenDoc)));
   return (CBezierkurvenDoc*)m_pDocument;
}
```

Ansichten aktualisieren

Jetzt ist das Programm fast fertig. Wenn Sie es nun kompilieren und starten, können Sie über den Menüpunkt *Fenster* > *Neues Info Fenster* das soeben erzeugte Listenfenster anzeigen lassen.

Sie werden jedoch feststellen, dass sich die Werte derzeit noch nicht ändern, wenn die Punkte bewegt werden.

Das Gleiche gilt übrigens auch, wenn Sie mit *Fenster* > *Neues Fenster* eine zweite Kurvenansicht öffnen – bewegen Sie nun einen Punkt im ersten Fenster, werden Sie feststellen, dass die zweite Kurvenansicht unverändert bleibt, bis sie mit der Maus in das zweite Fenster klicken.

Das ist bei bestimmten Anwendungen, die grafisch sehr aufwendig sind vielleicht sogar sinnvoll, in unserem Fall wirkt es hingegen eher unschön.

Erzwingen einer Fensteraktualisierung

Es gibt jedoch eine Methode *UpdateAllViews*, die alle Ansichten eines Dokuments zwingt, sich neu zu zeichnen:

```
void UpdateAllViews(
   CView* pSender,
   LPARAM lHint = 0L,
   CObject* pHint = NULL
);
```

Parameter	Beschreibung
pSender	Ansichtsklasse, die das Dokument verändert hat. Diese Klasse wird nicht zum Neuzeichnen aufgefordert, da sie ja ohnehin schon die aktuellste Version des Dokuments anzeigt. Geben Sie hier NULL an, um sämtliche Ansichten neu zu zeichnen.

Parameter	Beschreibung
lHint	Benutzerdefiniert. Kann weitere Angaben über die Modifizierung enthalten. Nützlich, um redundante Aktualisierungen zu vermeiden.
pHint	Zeiger auf einen benutzerdefinierten Kontext. Kann weitere Angaben über die Modifizierung enthalten. Nützlich, um redundante Aktualisierungen zu vermeiden.

Der Aufruf von *UpdateAllViews* gehört in die Dokumentenklasse, und zwar an diejenigen Stellen, an denen das Dokument verändert und eine Aktualisierung der Ansichten erforderlich ist.

Begeben Sie sich also in die Datei *BezierkurvenDoc.cpp* und aktualisieren Sie die nachstehenden Methoden:

Listing 6.18 **Neue Dokumentzugriffsmethoden**

```
void CBezierkurvenDoc::SetStartNode (POINT in_Point)
{
   m_BezierKurve.m_StartKnoten = in_Point;
   UpdateAllViews(NULL);
}

void CBezierkurvenDoc::SetEndNode (POINT in_Point)
{
   m_BezierKurve.m_EndKnoten = in_Point;
   UpdateAllViews(NULL);
}

void CBezierkurvenDoc::SetControlPoint1 (POINT
 in_Point)
{
   m_BezierKurve.m_KontrollPunkt1 = in_Point;
   UpdateAllViews(NULL);
}

void CBezierkurvenDoc::SetControlPoint2 (POINT
 in_Point)
{
   m_BezierKurve.m_KontrollPunkt2 = in_Point;
   UpdateAllViews(NULL);
}
```

Aktualisierung der neuen Ansichtsklasse

Während die reguläre Ansichtsklasse, die mit der Kurvendarstellung betraut ist, nun korrekt aktualisiert wird, ändern sich die Werte in der Listenansicht immer noch nicht.

Aktualisierung des Fensterinhalts einleiten

Das ist aber auch kaum verwunderlich, denn es wurde noch an keiner Stelle gesagt was passieren soll, wenn eine Veränderung der Dokumentdaten eintritt.

Das Aufrufen von *UpdateAllViews* sorgt für ein Aufrufen der *CView*-Methode *OnUpdate*. Überschreiben Sie diese Methode in der *CBezierKurvenInfoAnsicht*-Klasse wie folgt:

```
void CBezierKurvenInfoAnsicht::OnUpdate(CView*
/*pSender*/, LPARAM /*lHint*/, CObject* /*pHint*/)
{
   // TODO: Add your specialized code here and/or
// call the base class
   ShowNewValues();
}
```

Immer, wenn ein *UpdateAllViews*-Methodenaufruf stattfindet, wird nun ebenfalls indirekt *OnUpdate* der *CBezierKurvenInfoAnsicht* aufgerufen, was in der Folge über *ShowNewValues* zum Aktualisieren der Listenelementdaten führt.

Benutzerdefinierte Parameter

In der Funktion OnUpdate *können theoretisch die benutzerdefinierten Parameter verarbeitet werden, die im* UpdateAllViews*-Aufruf übergeben werden können.*

In der Windows-Welt werden Sie an vielen Stellen auf diese benutzerdefinierten Argumente stoßen, die Ihnen die Freiheit lassen, beliebige Daten während solcher Aufrufe mit anderen Methoden auszutauschen.

Im Falle von OnUpdate *könnte* UpdateAllViews *beispielsweise im* lHint*-Parameter einen Veränderungskoeffizienten übergeben, der bestimmt, wie stark die Veränderung des Dokuments im Vergleich zur letzten Version war.*

Angenommen, Sie arbeiten mit einem 3-D-Grafikprogramm und haben soeben einen Punkt um 0.001 Einheiten in einer Zoomansicht verschoben. Es gibt eine weitere Ansicht aus einem anderen Blickwinkel, die zugehörige Kamera befindet sich aber deutlich weiter vom Objekt entfernt.

Die Folge davon ist, dass die Veränderung in der zweiten Ansicht überhaupt nicht sichtbar wäre. Wenn Sie nun die 0.001 als lHint*-geeignet übergeben, könnte die* OnUpdate*-Methode der zweiten Ansichtsklasse diesen Umstand feststellen, und vollständig von einer Aktualisierung, die möglicherweise aufgrund der komplexen 3-D-Darstellung einige Zeit in Anspruch nehmen würde, absehen.*

Zusammenfassung

Nachdem das Programm nun endlich seinen kompletten angedachten Dienst versieht, wird es Zeit zusammen zu fassen, was Sie in diesem Kapitel gelernt haben.

Der wesentlichste Punkt war es sicherlich, eine MDI-Anwendung im Vergleich zur SDI-Anwendung des letzten Kapitels zu schreiben und eine zweite Ansicht an die Dokumentenklasse anzuhängen.

Weiterhin wurden zusätzliche Übungen im Bereich der Abbildungsmodi und GDI-Objekte durchgeführt sowie aufgezeigt, wie andere als die Standard-*CView*-Basisklasse verwendet werden können.

Die Arbeit mit Listen bildete dann den Abschluss dieses Kapitels.

Ausblick

Der nächste Abschnitt beschäftigt sich mit den Möglichkeiten, C++ im Rahmen des .NET Frameworks sinnvoll einzusetzen und insbesondere selbst geschriebene Bibliotheken und Funktionen weiterzuverwenden.

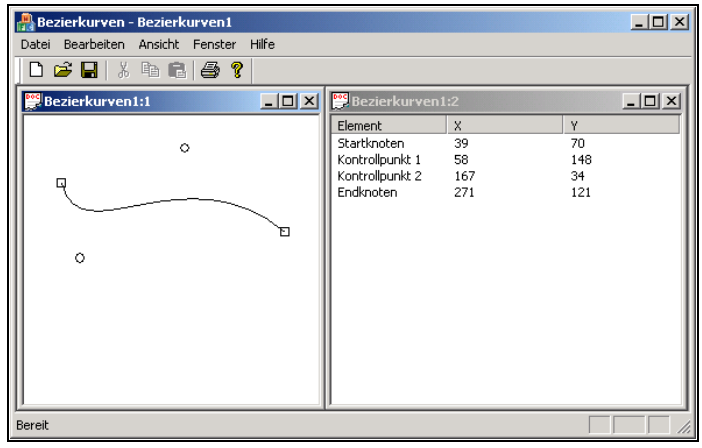
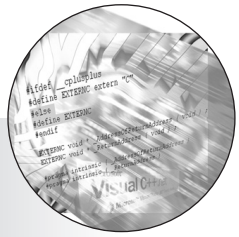


Abb. 6.10
Das fertige Programm zur Darstellung von Bezierkurven



VC++.net

7

Der Einsatz von VC++.net im Rahmen des .NET Frameworks

Das neue .NET

Die neue Version des Visual Studio zeichnet sich nicht nur durch eine komplett überholte Struktur der einzelnen Bedienelemente und des Hauptfensters sowie zahlreiche Detailverbesserungen in der Bedienung der Online-Hilfe, der Designer oder des Debuggers aus, sondern enthält in seinem Lieferumfang auch gleichzeitig noch ein komplett neues Framework namens .NET (sprich: Dotnet), das laut Microsoft neue Maßstäbe in der Entwicklung von Applikationen und hier insbesondere dem Erstellen von Webanwendungen setzen soll.

.NET vs. C++

Während es dem C++-Entwickler durchaus möglich ist, .NET-Anwendungen zu schreiben, muss doch klar gesagt werden, dass der Trend deutlich in Richtung von C# (oder Visual Basic) als Sprachplattform für das Erzeugen solcher Programme geht – das zeigt sich allein daran, dass es keine Designtools für Dialoge und Ähnliches für den C++-Anwender gibt, während diese für C# in der gewohnt umfangreichen Manier zur Verfügung stehen.

Auch existieren keine vom Anwendungsassistenten anlegbaren WinForm- oder WebForm-Vorlagen, sodass ein Entwickler viele der notwendigen Schritte per Hand übernehmen muss.

Dieses kurze Kapitel beschäftigt sich rudimentär mit dem neuen .NET Framework, zeigt beispielhaft auf, wie trotzdem eine WinForm-Anwendung erzeugt werden kann und gibt dann eine Anleitung, wie bestehende C++-Anwendungen (die im Kontext des neuen .NET-Anwendungsgerüsts als „unmanaged code“ – also nicht verwalteter Code – bezeichnet werden) in auf .NET basierenden Applikationen weiterverwendet werden können.

Für weitere Informationen zu .NET und insbesondere zu den darauf speziell ausgerichteten Sprachen wie C# haben wir als zusätzlichen Bonus zu diesem Buch eine Reihe weiterer Werke auf der beiliegenden CD zu Ihrem Nutzen abgelegt. Wenn Sie also näheres zu den Möglichkeiten des .Net Frameworks erfahren wollen, schauen Sie in den bereit gestellten Dateien nach.

Manuelles Schreiben einer WinForm-Anwendung

WinForms

WinForms entsprechen in etwa dem, was Sie von dialogfeldbasierenden Applikationen aus dem Kontext der MFC gewohnt sind.

Einsatz von WinForms in einem Beispielprojekt

Sie können die Größe dieser Felder beliebig wählen und Steuerelemente auf ihnen platzieren, die dann über Ereignisbehandlungsmethoden selbst definierte Aktionen ausführen können.

Die beispielhafte Verwendung einer solchen Form sei im Folgenden kurz skizziert.

Anlegen des neuen Projekts

Um Forms einsetzen zu können, müssen Sie zunächst Ihr Projekt auf eine Verwendung des .NET Frameworks einstellen. Erzeugen Sie hierzu ein neues Projekt und stellen als Typ *Managed C++ Application* ein. Nennen Sie die Anwendung *ManagedProject* (oder verwenden Sie das fertige Projekt von der Buch-CD aus dem Verzeichnis *\Kapitel7\ManagedProject*).

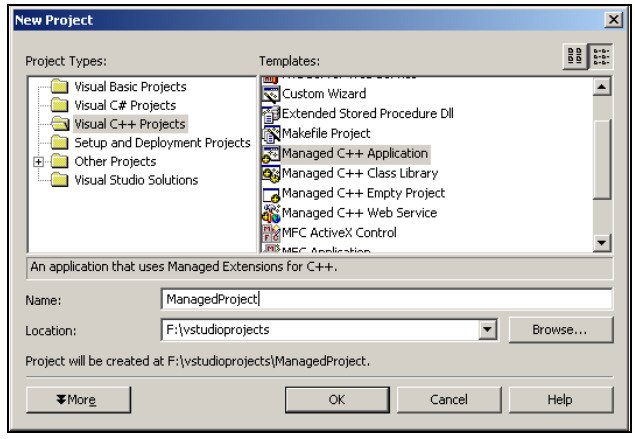


Abb. 7.1 **Projekteinstellungen**

Textausgabe in die Konsole

Öffnen Sie die entstandende Datei *ManagedProject.cpp* und editieren Sie sie wie folgt:

Listing 7.1 ***ManagedProject.cpp***

```
// This is the main project file for VC++ application
// project
// generated using an Application Wizard.

#include "stdafx.h"

#using <mscorlib.dll>
#include <tchar.h>

using namespace System;

// This is the entry point for this application
int _tmain(void)
{
    // TODO: Please replace the sample code below with
    // your own.
    Console::WriteLine(S"Eine Ausgabe durch das .NET
Framework");
    return 0;
}
```

Dieses ist die Hauptanwendungsdatei ihrer Managed C++-Anwendung. die *#using*-Anweisung sorgt dafür, dass Ihre Applikation Zugriff auf die zu .NET gehörenden Komponenten hat.

Namespaces in der .NET-Architektur

Namespaces

Durch das Verwenden des Namespaces *System* haben Sie weiterhin einen einfachen Zugriff auf die zum Framework gehörenden Klassen und Objekte.

Die Hierarchie hierbei ist an Java angelehnt. Beispielsweise befindet sich die Klasse zur Handhabung der von uns gewünschten Forms unter *System.Windows.Forms*.

Textausgabe in Konsole

Im ersten Beispielprogramm soll allerdings nur eine Textausgabe in die Konsole durchgeführt werden. Hierzu kann die Klasse *System::Console* verwendet werden, die jedwede Ein- oder Ausgaben an die Konsole kapselt.

Eine ihrer (statischen) Methoden ist *WriteLine* zum Ausgeben einer Zeile mit anschließendem Zeilenumbruch.

Intermediate Language

Eine Zwischensprache

Kompilieren Sie das Projekt, wird es – als managed application, also verwaltete Applikation – zunächst in eine *Zwischensprache* übersetzt (*intermediate language*, kurz *IL*), die dann von den Laufzeitkomponenten des Frameworks beim Ausführen des Programms in Maschinencode transferiert werden.

Das hat den Vorteil, dass vom System abhängige Optimierungen durchgeführt werden können, was bei einer maschinensprachigen Vorkompilierung nicht möglich ist – hier richtet sich das Ergebnis immer nach dem System, auf dem eine Applikation entwickelt wurde.

Warmlaufen von Applikationen

Der Nachteil ist, dass diese Kompilierung immer erst dann ausgeführt wird, wenn bestimmte Programmteile aufgerufen werden – es kommt also zu Verzögerungen bei der anfänglichen Arbeit mit Anwendungen. Je länger Sie mit einer Applikation arbeiten, desto schneller läuft sie ab. Das ist in gewisser Weise mit einem Neuwagen zu vergleichen, der auch erst eingefahren werden will, bevor er seine komplette Motorenleistung an den Tag legt.

Ergebnis des ersten .NET-Programms

Teststart

Starten Sie das Projekt von der Kommandozeile aus (anderenfalls würde die Ausgabe sofort wieder verschwinden). Hierbei fällt Ihnen vielleicht auf, dass das Ergebnis eine gute Weile auf sich warten lässt, was mit den bereits angesprochenen nachträglichen Übersetzungsvorgängen zusammenhängt.

Das – unspektakuläre – Ergebnis ist eine Ausgabe der im Quelltext eingetragenen Zeile:

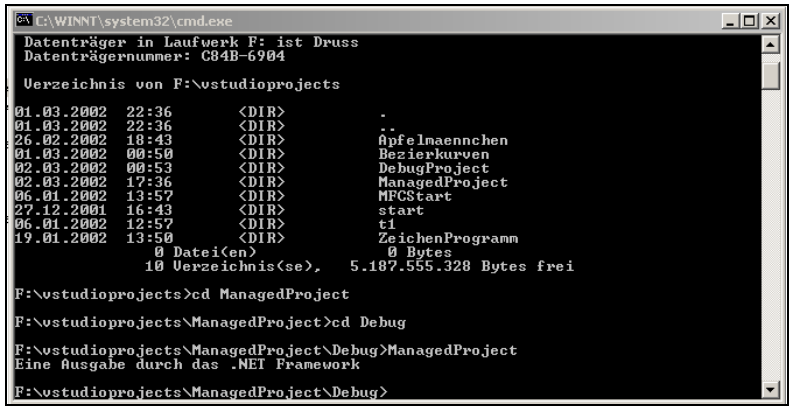


Abb. 7.2
Textausgabe in die Konsole

Erzeugen einer WinForm

Anlegen eines Fensters

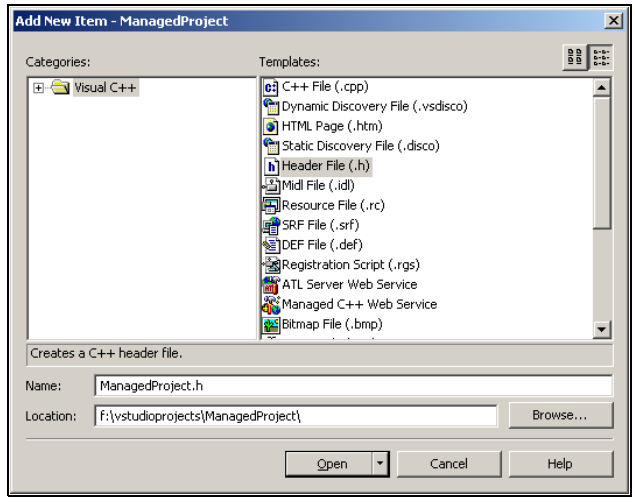
Der nächste Schritt soll darin bestehen, eine Form auf den Bildschirm zu bringen, die neben einem eigenen Fenster einen Knopf enthält, der eine Aktion ausführen kann. Der Einfachheit halber beschränken wir uns hierbei auf die Ausgabe eines Texts in die während der Programmausführung offen stehenbleibenden Konsole.

Wie bei den MFC werden Forms durch eine eigene Klasse repräsentiert, sodass es sich für den Entwickler anbietet, hieraus eine eigene Klasse abzuleiten und nach eigenen Wünschen anzupassen.

Da im bisherigen Projekt nur Quellcodedateien existieren, öffnen Sie im Arbeitsbereich das Kontextmenü des *ManagedProject*-Projekts und wählen dort den Punkt *Add > Add New Item*.

Es öffnet sich ein Dialogfeld ähnlich dem folgenden, wählen Sie dort den Typ *Header File (.h)* aus und geben der neu zu erzeugenden Datei den Namen *ManagedProject.h*:

Abb. 7.3
Hinzufügen einer neuen Klasse



Anlegen einer abgeleiteten Formklasse

Öffnen Sie die neu entstandene Datei – die derzeit komplett ohne Inhalt ist – und fügen Sie die folgenden Zeilen ein:

Listing 7.2
Die neue Formklasse

```
#using <mscorlib.dll>
#using <System.Windows.Forms.dll>
#using <System.dll>

using namespace System;
using namespace System::Windows::Forms;

__gc class NewWinForm : public Form
{
public:
   NewWinForm();
   void OnButtonClicked(Object *sender,
EventArgs *e);
};
```

Hier werden zunächst mittels *using* einige dlls spezifiziert, die zur Verwendung der Formklasse notwendig sind. Die danach angegeben Namespaces dienen zur einfacheren Arbeit mit den Klassen des .NET Frameworks.

Form ist die Basisklasse für WinForm-Applikationen, sodass eine Klasse hiervon abgeleitet werden soll. Der Modifizierer *__gc* sorgt dafür, dass wir es mit einer verwalteten Klasse zu tun haben.

Managed Applications

„Verwaltet“ bedeutet hier beispielsweise, dass Sie sich nicht selbst um Fragen des Speichermanagements kümmern müssen. Legen Sie ein Objekt mittels *new* auf dem Heap an, braucht es nicht mittels *delete* gelöscht zu werden – .NET erkennt automatisch, wann das Objekt nicht mehr verwendet wird und löscht es zu gegebener Zeit selbstständig.

Neben einem Konstruktor enthält die neue Klasse noch eine Behandlungsmethode für die Schaltfläche, die, wie bereits angekündigt, auf der Form vorhanden sein soll. Nähere Details hierzu gibt es, wenn es um die Implementation der Klasse geht.

Implementation und Anwendung der Klasse

Die Implementation selbst gehört wieder in die Datei *ManagedProject.cpp* – es wird hier eine weitere Aufteilung in weitere Dateien aus Gründen der Übersichtlichkeit eingespart:

Listing 7.3
Neue *ManagedProject.cpp*

```
#include "stdafx.h"
#include „ManagedProject.h"

#using <mscorlib.dll>
#include <tchar.h>

using namespace System;
using namespace System::Windows::Forms;

NewWinForm::NewWinForm() : Form()
{
   // Größe der Form festlegen
   Width = 150;
   Height = 60;

   // neuen Knopf anlegen
   Button *pButton = new Button();

   // Knopfattribute setzen
   pButton->Text = „Test";
   pButton->Left = 40;
   pButton->Top = 5;

   // Eventhandling delegieren
   pButton->Click += new EventHandler(this,
 OnButtonClicked);

   // Knopf zur WinForm hinzufügen
   Controls->Add(pButton);
}
```

```
void NewWinForm::OnButtonClicked(Object *sender, EventArgs *e)
{
   // Zeile ausgeben
   Console::WriteLine(„Knopf gedrueckt!");
}

// Hauptprogramm
int _tmain(void)
{
Application::Run(new NewWinForm());
    return 0;
}
```

Erklärungen zur Implementation

Wie schon bei den MFC kann ein Großteil der Funktionalitäten aus der Basisklasse übernommen werden. Genaugenommen kümmert sich das Beispielprogramm nur um einige wenige Attribute der Form, wie beispielsweise die Größe, indem diese Attribute, die in der *System.Windows.Form*-Klasse bereits enthalten sind, geeignet anpasst.

Eine Breite von 150 und eine Höhe von 60 Pixel sorgt für ein relativ kleines Ausgabefenster, das für dieses einfache Beispiel aber vollkommen ausreichend ist.

Auf der Form soll weiterhin eine Schaltfläche angeordnet werden. Eine für ein solches Steuerelement passende Klasse findet sich mit *System.Windows.Form.Button*.

Das Anlegen geschieht unspektakulär über eine new Anweisung, das Setzen der Attribute ist aus dem Kontext heraus verständlich – hier wird die Beschriftung sowie die Position auf der Form festgelegt.

Delegierte

Behandlungsmethode für die Schaltfläche

Wesentlich interessanter ist jetzt die Angabe einer Behandlungsmethode für den Fall, dass der Knopf gedrückt wird.

Sie wissen aus der MFC-Welt, dass solche Ereignisfunktionen auch in dem umschließenden Fenster definiert werden können. Das ist aber genaugenommen ein Bruch mit der objektorientierten Programmierung – schließlich sollte der Knopf selbst für seine Behandlung notwendig sein.

Ergo wäre es sinnvoll, eine Klasse vom Steuerelement abzuleiten und die Nachrichtenfunktion dort einzusetzen.

Obwohl dieses der saubere Weg wäre, ist er doch recht umständlich wenn man bedenkt, dass ein Dialogfeld unter Umständen mehrere Dutzend Kontrollen enthalten kann, was dann in der Folge das Anlegen von ebenso vielen abgeleiteten Steuerelementklassen bedeuten würde.

Delegierte

Im Rahmen von .NET besteht nun die Möglichkeit, die Behandlungen tatsächlich an genaugenommen jeder beliebigen Stelle stattfinden zu lassen. Hierzu wird für das betreffende Ereignis eine Delegierter bestimmt (eine Behandlungsfunktion), die dem Steuerelement (oder einem sonstigen Objekt, das Ereignisse erzeugen beziehungsweise erhalten kann) mitgeteilt wird.

Immer, wenn das betreffende Ereignis auftaucht, wird der Delegierte aufgefordert, seine Arbeit zu verrichten.

Im obigen Beispiel wird dem Knopf mitgeteilt, dass im Fall eines Anklickens die Funktion *OnButtonClicked* der Klasse *NewWinForm* (über den *this*-Zeiger spezifiziert) aufgerufen werden soll.

Es wird an dieser Stelle nicht ein konkreter Eventhandler zugewiesen, sondern mit dem Operator += lediglich der Delegierte an die Liste der möglicherweise schon für dieses Ereignis festgelegten anderen Delegierten angehängt.

Es ist somit möglich, dass das Anklicken der Schaltfläche tatsächlich eine ganze Reihe von Behandlungsmethodenaufrufen auslöst.

Abschließend wird der Knopf an die erzeugte Form angehängt – sie hat hierzu ein eigenes Controls-Objekt, das eine Liste sämtlicher auf der Form enthaltener Steuerelemente enthält.

Die *Main*-Funktion

Startablauf der Applikation

In der *Main*-Funktion wird die neue Form über einen Methodenaufruf von *Application::Run* gestartet. *Application* ist dabei wiederum eine Klasse von *System.Windows.Form* und übernimmt als Parameter ein Form-Objekt.

Es initiiert daraufhin das Abarbeiten der Nachrichtenfunktion für dieses Objekt. Dieser Vorgang ist mit dem analogen Startvorgang in der MFC vergleichbar.

Testen Sie die Applikation nach erfolgter Kompilierung nun erneut aus:

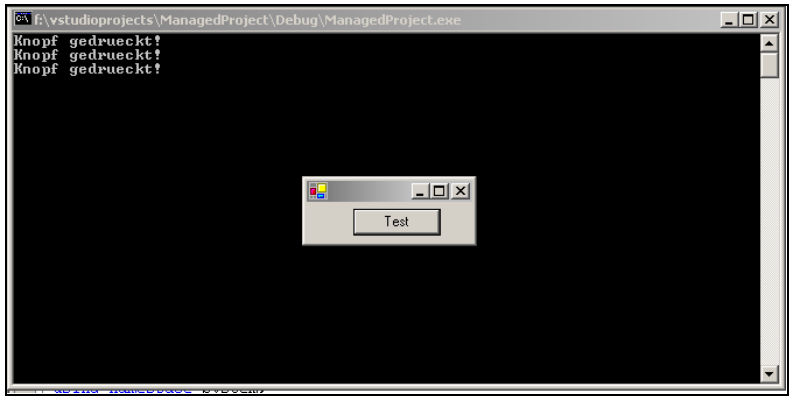


Abb. 7.4
Die Form

Die Arbeit mit dem .NET Framework entspricht von den Grundsätzen her also dem, was Sie im Rahmen der MFC bereits gesehen haben – der große Unterschied beschränkt sich auf das verwendete Klassenframework.

An dieser Stelle sollen nun keine weiteren Erklärungen hierzu folgen, es sei auf einschlägige Literatur und insbesondere die auf der Buch-CD befindlichen weiteren digitalen Bücher aus dem Sybex Verlag verwiesen.

Weiterverwenden von alten C++-Klassen

Wiederverwendung alter Bibliotheken

Eine große Sorge, die viele C++-Entwickler bei der Ankündigung von .NET hegten war die, ob es überhaupt möglich sein würde, mit alten C++-Klassenbibliotheken weiterzuarbeiten.

Die Antwort auf diese Frage ist klar mit Ja zu beantworten.

Allerdings können diese nicht verwalteten Klassen nicht ohne eine kleine Ergänzung eingesetzt werden.

Wie Sie trotzdem Ihre vielleicht schon umfangreichen Bibliotheken weiterverwenden können, sei im folgenden Beispiel grob veranschaulicht.

Eine nicht verwaltete Klasse

Als Beispiel wird zunächst eine nicht verwaltete (*unmanaged*) C++-Klasse geschrieben, die inhaltlich nicht viel unternimmt, außer eine Zufallszahl aus dem Bereich 0 bis 5 zurückzuliefern.

Die Einfachheit der gewählten Klasse erlaubt es, die wesentlichen Umstellungsvorgänge ins richtige Licht zu rücken.

Neue Dateien anlegen

Erzeugen Sie zunächst zwei neue Dateien namens *UnmanagedClass.h* und *UnmanagedClass.cpp* wie oben beschrieben (stellen Sie weiterhin die Verwendung von vorkompilierten Headern in den Projekteinstellungen aus, damit es nicht zu Problemen beim Übersetzen kommt – die vorkompilierten Header müssten von allen CPP-Dateien genutzt werden, doch ist dieses bei alten Klassen oft nicht der Fall).

Schreiben Sie dann die nachstehende Klassendeklaration in die Datei *UnmanagedClass.h*:

Listing 7.4 UnmanagedClass.h

```
class CUnmanagedClass
{
   public:
      CUnmanagedClass();
      ~CUnmanagedClass();

      int GetElement(int in_nPosition);
```

```
   private:
      int *m_pIntArray;
};
```

Die Implementation gehört nach *UnmanagedClass.cpp*:

Listing 7.5 UnmanagedClass.cpp

```
#include "UnmanagedClass.h"

CUnmanagedClass::CUnmanagedClass()
{
   m_pIntArray = new int[5];

   for (int i=0;i<5;i++)
   {
      m_pIntArray[i] = i;
   }
}

CUnmanagedClass::~CUnmanagedClass()
{
   delete [] m_pIntArray;
}

int CUnmanagedClass::GetElement(int in_nPosition)
{
   if ((in_nPosition >= 0) && (in_nPosition <= 5))
   {
      return (m_pIntArray[in_nPosition]);
   }
   else
   {
      return (-1);
   }
}
```

Sie sehen, inhaltlich geschieht hier nicht viel. Die Klasse erzeugt bei Instantiierung ein Integer Array bestehend aus fünf Elementen, die, bei 0 beginnend, sequenziell mit Werten beschrieben werden.

Die Funktion *GetElement* übernimmt einen Indexparameter und liefert den zugehörigen Wert zurück. Ein fehlerhafter Parameter sorgt für das Zurückgeben der Zahl -1.

Versuch, die Klasse in einer verwalteten Klasse zu verwenden

Probleme beim Einsatz der alten Klasse

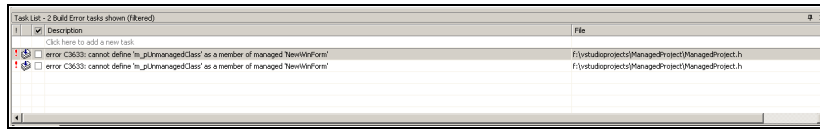
Der triviale Ansatz, die nicht verwaltete Klasse einfach in die bestehende *NewWinForm*-Klasse zu integrieren scheitert. Testen Sie dieses aus, indem Sie die Datei *ManagedProject.h* editieren:

Listing 7.6 ManagedProject.h

```
#using <mscorlib.dll>
#using <System.Windows.Forms.dll>
#using <System.dll>

using namespace System;
using namespace System::Windows::Forms;
using namespace System::Collections;

#include „UnmanagedClass.h"

__gc class NewWinForm : public Form
{
public:
   NewWinForm();
   void OnButtonClicked(Object *sender,
EventArgs *e);

private:
   CUnmanagedClass m_UnmanagedClass;
};
```

Die Kompiliierung ergibt eindeutige Fehler, die besagen, dass die nicht verwaltete Klasse nicht verwendet werden kann:

Abb. 7.5 Fehler bei der Kompilierung



Hinzufügen einer Proxyklasse

Die Lösung besteht darin, eine zwischengeschaltete Klasse zu entwickeln, die quasi als Kapselung von *CUnmangedClass* dient.

Anlegen einer zwischengeschalteten Klasse

Eine solche Kapselung enthält als Schnittstelle sämtliche Funktionen, die von der nicht verwalteten Klasse nach außen gegeben werden sollen sowie einen Zeiger auf ein Objekt der betreffenden Klasse.

Fügen Sie nun zwei weitere Dateien namens *Proxyclass.h* und *Proxyclass.cpp* in das Projekt ein.

Füllen Sie die Datei *Proxyclass.h* wie folgt auf:

Listing 7.7 Proxyclass.h

```
#include "UnmanagedClass.h"

#using <mscorlib.dll>

__gc class CProxyClass
{
public:
   CProxyClass();
   ~CProxyClass();

   int GetElement (int in_nElement);

private:
   CUnmanagedClass *m_pUnmanagedClass;
};
```

Die neue verwaltete Klasse, enthält einen Zeiger auf ein *CUnmanagedClass*-Objekt sowie eine Methode *GetElement*, die als Relaisfunktion dienen wird.

Implementieren der Klasse

Die Implementation der *CProxyClass*-Klasse in der Datei *Proxyclass.cpp* ist nachfolgend abgedruckt:

Listing 7.8 Proxyclass.cpp

```
#include "ProxyClass.h"

CProxyClass::CProxyClass()
{
   m_pUnmanagedClass = new CUnmanagedClass;
}

CProxyClass::~CProxyClass()
{
   delete (m_pUnmanagedClass);
}

int CProxyClass::GetElement(int in_nElement)
{
   return (m_pUnmanagedClass
->GetElement(in_nElement));
}
```

Die Klasse erzeugt also ein neues Objekt mithilfe des *new*-Operators und löscht es explizit über *delete*. Damit ist sichergestellt, dass durch das nicht verwaltete Objekt keine Speicherprobleme auf dem Heap entstehen können.

Die *GetElement*-Methode ist tatsächlich nur eine Relaisfunktion, die den Aufruf an das *CUnmanagedClass*-Objekt weiterleitet und deren Rückgabewert zurückliefert.

Was durchschleifen – und was nicht?

Als Faustregel gilt, dass sämtliche Methoden der nicht verwalteten Klasse, die von einer .NET-Applikation aus zugänglich sein sollen, als Relaismethoden in die Proxyklasse einzubetten sind.

Private Methoden, auf der anderen Seite, gehören hier nicht hinein.

Unter Umständen ist es notwendig, ein so genanntes Datenmarshalling zu betreiben, etwa dann, wenn eine Konvertierung von Datentypen zwischen .NET und der alten C++-Klasse notwendig ist.

In solchen Fällen ist die Relaisfunktion derart aufzubauen, dass Sie den .NET Datentyp übernimmt, ihn intern in einen für die alte C++-Klasse verständlichen umkonvertiert, diesen dann an die alte Methode übergibt und den Rückgabewert wieder für .NET passend aufbereitet.

Einsatz der Proxyklasse

Die fertige Proxyklasse kann nun in die *NewWinForm*-Klasse integriert werden:

Listing 7.9 ManagedProject.h

```
#using <mscorlib.dll>
#using <System.Windows.Forms.dll>
#using <System.dll>

using namespace System;
using namespace System::Windows::Forms;
using namespace System::Collections;

#include "ProxyClass.h"

__gc class NewWinForm : public Form
{
public:
   NewWinForm();
   void OnButtonClicked(Object *sender,
EventArgs *e);

private:
   CProxyClass *m_ProxyClass;
};
```

Zum Austesten der Funktionalität soll die *OnButtonClicked*-Methode überarbeitet werden. Editieren Sie den Funktionsrumpf daher entsprechend der nachstehenden Zeilen:

Listing 7.10
Neue *OnButtonClicked*-Methode

```
void NewWinForm::OnButtonClicked(Object *sender,
EventArgs *e)
{
   // Instanz der Proxyklasse anlegen
   m_ProxyClass = new CProxyClass;

   // Zufallszahl ermitteln
   Random *Zufallszahlen = new Random;
   int nRandom = Zufallszahlen->Next(0, 4);

   // Element über Proxyklasse auslesen
   int nElement = m_ProxyClass
->GetElement(nRandom);

   // Element auslesen
   Console::WriteLine("x is {0}", __box(nElement));
}
```

Hier wird ein neues Objekt der *CProxyClass*-Klasse angelegt, eine Zufallszahl ermittelt und dann über die Relaismethode transparent mit der dahinter liegenden alten C++-Klasse gearbeitet.

An dieser Stelle zeigt sich auch noch einmal der Sinn der Proxyklasse: das neu angelegte Objekt wird zwar mit *new* erzeugt, allerdings nicht wieder freigegeben. Dieses geschieht automatisch durch das .NET Framework, da *CProxyClass* eine verwaltete Klasse ist.

Die nicht verwaltete Klasse *CUnmanagedClass* hingegen wäre an dieser Stelle nicht freigegeben worden und hätte, bei zahlreichen Aufrufen, mit der Zeit den Heap gefüllt – das klassische Speicherleck hätte zugeschlagen.

Ausgabe von Daten via *WriteLine*

Die zurückgelieferte Zahl soll in die Konsole geschrieben werden, die Verwendung der *__box*-Funktion sorgt dafür, dass die Daten in einer für *WriteLine* angemessenen Art und Weise aufbereitet werden.

Abschließender Testlauf

Kompilieren Sie das Programm und starten es. Testen Sie aus, ob das Drücken der Schaltfläche die gewünschte Wirkung erzielt.

Zeitersparnis bei der Verwendung alter Bibliotheken

Sie sehen, das Übernehmen von alten C++-Funktionen ist ohne große Schwierigkeiten möglich. Sie können also bei der Entwicklung von .NET-Anwendungen zunächst ihre alten Bibliotheken beibehalten und diese dann Schritt für Schritt umstellen.

Durch die Proxyklassen hat dieser Schritt aber prinzipiell soviel Zeit, bis Sie in der Entwicklung Freiraum für die Umstellung haben.

Ein weiterer Vorteil, der sich durch die Proxyklassen automatisch bietet ist, dass Sie die alten C++-Klassen auch mit anderen .NET Sprachen sofort verwenden können – sämtliche .NET-Programme werden ja zunächst in die *IL* übersetzt, sodass sich nach der Kompilierung theoretisch kein Unterschied mehr zwischen einem ursprünglich unter C++.net oder C#.net entwickelten Programm finden lässt.

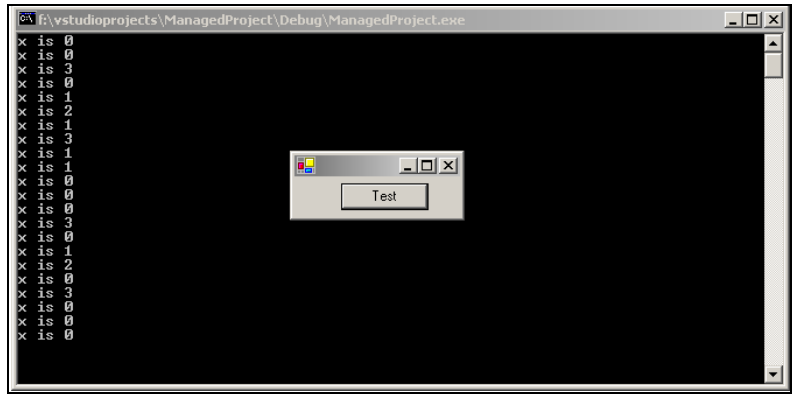
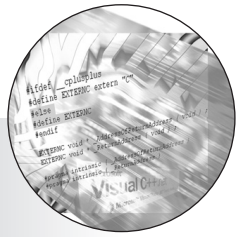


Abb. 7.6
Die Verwendung der alten C++-Klasse



Debugging

8

Die Notwendigkeit eines guten Debuggers

Den Abschluss dieses Buches – von den umfangreichen Anhängen einmal abgesehen – bildet ein eigenes Kapitel zur Verwendung des Visual Studio Debuggers im Rahmen der MFC-Programmierung.

Gründe für gutes Debugging

Es mag merkwürdig erscheinen, einen solchen Abschnitt in einem Programmierbuch vorzufinden, doch bedenken Sie, dass es hier ja konkret um die Arbeit mit Visual C++.net geht, das nahezu untrennbar mit dem Visual Studio verbunden ist.

Dieses wiederum bietet eine ganze Reihe von Hilfsmitteln, um Fehler in Programmen aufzuspüren und zu beheben.

Denn seien wir einmal ehrlich: Wie viel Zeit vergeht mit der eigentlichen Programmierung und wie viel mit der Fehlersuche? Wer hat nicht schon einmal stundenlang seine Quelltexte durchforstet, um am Ende herauszufinden, dass lediglich an irgendeiner Stelle ein '+' zuviel war oder eine Variable nicht oder falsch initialisiert wurde?

Debugging im Wandel der Zeit

Rapid Development

Um die Zeit beim Debugging so niedrig wie möglich zu halten und damit den als *Rapid Development* bezeichneten Idealzustand bei der Applikationsentwicklung erreichen zu können, sind gute Tools unabdingbar.

Wer einmal auf älteren Computersystemen gearbeitet hat, weiß, was ich meine: Obwohl die Hardware in ihrer Komplexität mit der heutiger Rechner nicht zu vergleichen war und Probleme wie Multitasking oder Multiusermanagement noch in den Sternen standen, gestaltete sich die Entwicklung von Programmen um einiges schwieriger als heutzutage.

Debugging im Wandel der Zeit

Gerade Assemblerprogrammierer können ein Lied davon singen, wie mühsam es häufig war, Quelltexte zu debuggen. Umständlich zu bedienenden Monitorprogramme ermöglichten zwar den Zugriff auf aktuelle Registerinhalte oder Speicherdumps, doch konnte das Untersuchen einer längeren Schleife durchaus zur Qual werden, wenn ein Fehler erst gegen Ende des Durchlaufs eintrat und man zunächst möglicherweise hunderte von vorhergehenden, ordnungsgemäß arbeitenden Durchgängen per Hand überprüfen musste.

Dazu kamen dann noch zwar recht interessante Neuentwicklungen zur Unterstützung bei der Fehlersuche, doch stellte sich in leidvollen Erfahrungen häufig heraus, dass diese nur neue, ursprünglich vorhandene Fehler in den Debug-Prozess einführten, oder, weitaus schlimmer, Fehler verbargen, die mithilfe der neuen Tools einfach nicht mehr auftreten wollten – diese kamen dann meistens erst beim Endanwender wieder zum Vorschein.

Heutzutage sieht die Situation deutlich besser aus. Entwicklungsumgebungen wie das Visual Studio stellen mächtige Instrumente zur Verfügung, von denen die Programmierer vergangener Zeiten nur träumen konnten.

Der aktuelle Stand der Dinge

Sämtliche relevanten Daten eines Programmablaufs sind ständig im Blickfeld und auch die Anwendungsgerüste, mit denen Applikationen erzeugt werden, stellen eigene Funktionalitäten in Form von Funktionen und Makros zur Verfügung.

Grund genug also, sich ausgiebig mit diesen Möglichkeiten zu beschäftigen – Ihre Programme und vor allem Ihre vermutlich ohnehin schon kostbare Zeit werden es Ihnen danken.

Möglichkeiten des Debuggings

Im Wesentlichen unterscheidet man drei Arten des Debuggings (obwohl viele Experten hier eine weitaus detailliertere Unterscheidung treffen, sind dieses wohl die drei Kategorien, die als Oberbegriffe genannt werden müssen):

- Debugging mithilfe von Testroutinen oder -makros, die im fehlerfreien Fall still arbeiten, bei auftretenden Problemen aber eine Meldung auf den Schirm oder ein sonstiges Medium bringen. Hierzu zählen auch Logfiles, die nach der Abarbeitung eines Programms evaluiert werden können. Gemeinsam haben diese Methoden, dass nur Daten geprüft, beziehungsweise ausgegeben werden, die vom Programmierer selbstständig eingearbeitet werden.

  Testroutinen

- Fehlermeldungen, die durch den Debugger einer Entwicklungsumgebung gemeldet werden. Die meisten Entwicklungsumgebungen stellen einen Debugger zur Verfügung, unter dem Programme ausgeführt werden können. Spezielle Versionen, so genannte Debug-Builds, des Programms enthalten zusätzliche Informationen, die implizit auf Gültigkeit geprüft werden. Beispielsweise kann ein moderner Debugger feststellen, ob eine uninitialisierte Variable innerhalb einer Rechnung verwendet wird und gibt in diesem Fall eine entsprechende Fehlermeldung aus.

  Debugger-Meldungen

- Manuelles Debugging von einzelnen Funktionen oder ganzer Programme. Der Entwickler verlässt sich hierbei nicht auf eine der beiden oben genannten Möglichkeiten, beziehungsweise hat über diese keine Option, einen Fehler zu finden und zu beseitigen. Diese Methode kommt meistens dann zum Einsatz, wenn ein Fehler zwar bekanntermaßen auftritt, aber keine ungültigen Informationen erzeugt, die durch den Debugger oder Testroutinen abgefragt werden können. Der Entwickler bedient sich der Tools des Debuggers, um schritt- oder blockweise durch das Programm hindurchzugehen, die Abarbeitung an bestimmten Stellen abzubrechen oder einfach nur Variablen-, Register- und Speicherwerte zu beliebigen Zeiten zu untersuchen.

  Breakpoints und Step-by-Step Debugging

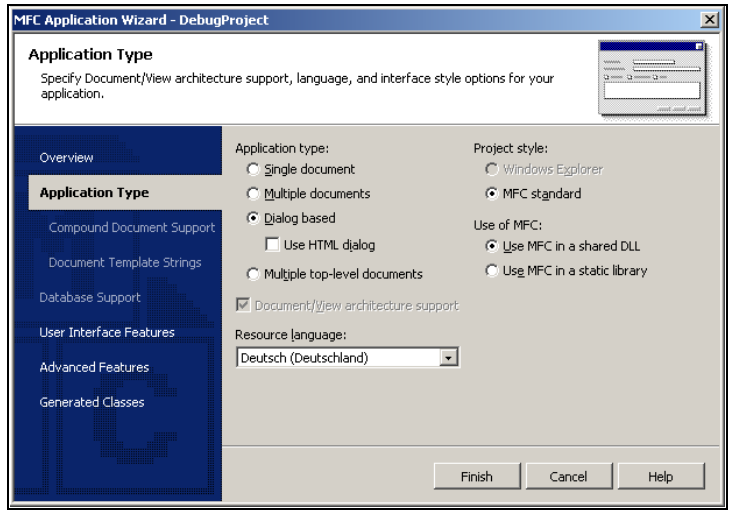
Im Folgenden werden alle drei Arten von Debugging im Rahmen des Visual Studio und den MFC genauer dargestellt.

Anlegen eines Debug-Projekts

Beispielprojekt anlegen

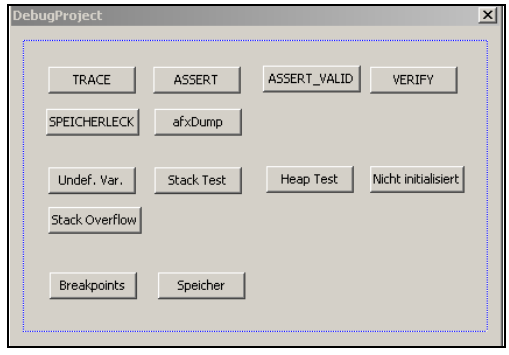
Um die einzelnen Debugging-Vorgehen an konkreten Fehlerstudien zu demonstrieren zu können, soll ein passendes Testprojekt angelegt werden, das auf Basis einer dialogfeldbasierenden Applikation zu erzeugen ist. Nennen Sie das Projekt *DebugProject* oder nehmen Sie das fertige Projekt aus dem Verzeichnis *\Kapitel8\DebugProject* der beiliegenden Buch-CD:

Abb. 8.1
Das dialogfeldbasierende Debug-Programm



Öffnen Sie in der Ressourcenansicht die Dialogfeldressource *IDD_DEBUGPROJECT_DIALOG* und fügen Sie dreizehn Schaltflächen hinzu:

Abb. 8.2
Die Schaltflächen für das Debug-Projekt



Verteilen Sie ferner die IDs und Beschriftungen der nächsten Tabelle und legen für jede Schaltfläche eine eigene Behandlungsmethode an. Darin werden dann auf den folgenden Seiten einzelne Debug-Szenarien implementiert werden.

Dialog mit Steuerelementen und IDs versehen

Tabelle 8.1
IDs für das *Debug-Project*-Programm

ID	Beschriftung
IDC_TRACETEST	TRACE
IDC_ASSERTTEXT	ASSERT
IDC_ASSERTVALIDTEST	ASSERT_VALID
IDC_VERIFYTEST	VERIFY
IDC_SPEICHERLECKTEST	Speicherleck
IDC_AFXDUMPTEST	afxDump
IDC_UNDEFVARIABLETEST	Undef. Variable
IDC_STACKCORRUPTIONTEST	Stack Test
IDC_HEAPTEST	Heap Test
IDC_NICHTINITIALISIERTTEST	Nicht initialisiert
IDC_STACKTEST	Stack Overflow
IDC_BREAKPOINTTEST	Breakpoints
IDC_SPEICHERTEST	Speicher

Fehlersuche mit den MFC

Begonnen werden soll mit dem Suchen von Fehlern unter Zuhilfenahme der MFC-eigenen Debug-Makros und -methoden.

Hiervon gibt es im Wesentlichen fünf Varianten:

MFC-Debug-Hilfen

- *TRACE* zur Ausgabe von Text
- *ASSERT* zur Prüfung von Zusicherungen
- *ASSERT_VALID* zur Prüfung von Objekt-Zusicherungen
- *VERIFY* zur Prüfung von Zusicherungen in Debugversionen
- *afxDump* zur Ausgabe von Objektinformationen

Einsatz von Fehleraufspürmethoden

Sie sollten ausgiebigen Gebrauch von allen diesen Möglichkeiten machen, um bereits bei der Entwicklung ein Höchstmaß an Fehlern aufdecken zu können.

Allgemein gilt, dass Fehler, die erst spät in einem Programm entdeckt werden, wesentlich mehr Zeit zur Behebung erfordern als solche, die frühzeitig auftauchen.

Alle fünf der oben stehenden Makros und Methoden erlauben es, bereits frühzeitig aufzudecken, ob fehlerhafte Parameter an Funktionen übergeben werden und dergleichen mehr – nutzen Sie diesen Vorteil.

TRACE

Ausgabe von Logzeilen

Eine von DOS-Entwicklern gerne eingesetzte Möglichkeit zur Aufspürung von Fehlern besteht darin, den Quelltext mit Ausgabezeilen zu spicken, die zwischen wichtigen Anweisungen platziert sind und entweder als reine Positionsanzeiger dienen oder auch Werte von Variablen ausgeben.

Das könnte beispielsweise so aussehen:

```
cout << "EineFunktion wird ausgefuehrt" << endl;
EineFunktion();
cout << "EineFunktion wurde erfolgreich ausgefuehrt"
   << endl;
int a = 27;
int b = NochEineFunktion (a);
cout << "NochEineFunktion wurde ausgefuehrt und gab
den Wert " << b << " zurueck" << endl;
```

Ein mögliches Ergebnis dieser Vorgehensweise ist der *Abbildung 8.3* zu entnehmen.

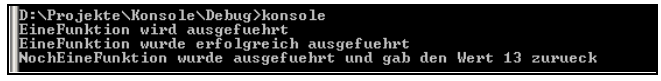


Abb. 8.3
Debug-Ausgaben in der Konsole

Ein wesentlicher Nachteil dieser Herangehensweise bei der Windows-Programmierung ist der, dass in der Regel keine Konsole für derartige Ausgaben zur Verfügung steht.

Allerdings enthält das Visual Studio ein eigenes Fenster namens Ausgabe, dass für eine gleichartige Funktionalität eingesetzt werden kann.

Mithilfe des Makros *TRACE* können Sie nämlich Ausgaben in das Ausgabefenster schreiben, wobei auch auf eine komfortable Formatierung ähnlich den bekannten *sprintf*-Anweisungen nicht verzichtet werden muss.

Ausgabe von Werten in das Ausgabe-Fenster

Editieren Sie die Methode *OnBnClickedTracetest*, um das *TRACE*-Makro auszutesten:

```
void CDebugProjectDlg::OnBnClickedTracetest()
{
   // Text definieren
   CString strText(_T("Ein Text"));

   // Zahl definieren
   int nZahl = 23;

   // Debug-Trace Ausgabe vornehmen
   TRACE(_T("\n%s und eine Zahl: %d.\n"),
 LPCTSTR(strText), nZahl);
}
```

Listing 8.1
Beispiel für das *TRACE*-Makro

Zwei Besonderheiten sind bei der Verwendung des *TRACE*-Makros zu beachten:

- Um einen auszugebenden String in ein für *TRACE* verständliches Format zu bringen, ist die Zeichenkette an das Makro *_T* zu übergeben.
- Für *%s*-Parameter müssen C-kompatible Strings analog zu *sprintf* übergeben werden. Da *strText* im obigen Beispiel vom Objekttyp *CString* ist, muss zunächst eine Typkonvertierung mittels *LPCTSTR* stattfinden, die gerade einen passenden Zeiger auf den im *CString*-Objekt befindlichen String bereitstellt.

Sie erhalten das in *Abbildung 8.4* zu sehende Ergebnis, wenn Sie die Schaltfläche *TRACE* im Debug-Modus (Starten mit F5) beziehungsweise über das Menü *Debug > Start*) betätigen.

Ist das Ausgabefenster nicht zu sehen, können Sie es mit Strg+Alt+O in den Vordergrund bringen.

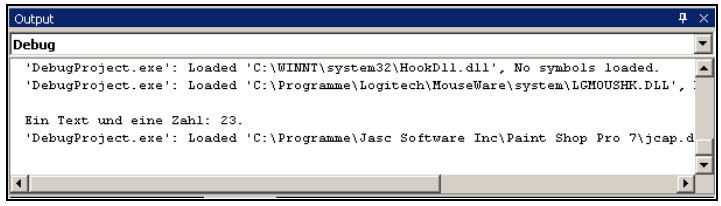


Abb. 8.4
Ausgabe von *TRACE*

Integrierte Ausgaben

Über beziehungsweise unter der *TRACE*-Ausgabe finden Sie weitere Meldungen, die automatisch eingefügt werden – auch zeigt sich hier die Begrenzung der Möglichkeiten von *TRACE*: Bei vielen Ausgaben wird diese Vorgehensweise schnell unübersichtlich.

TRACE-Makros werden in Release-Builds einer Anwendung automatisch entfernt.

ASSERT

Zusicherungen

Häufig haben Sie es in Ihren Programmen mit Funktionen oder Strukturen zu tun, bei denen Sie im Voraus abschätzen können, in welchem Wertebereich sich Parameter oder Komponenten bewegen dürfen – entweder direkt durch Angabe absoluter Werte oder indirekt durch einen Vergleich zweier oder mehrerer Daten (beispielsweise wissen Sie, dass Parameter A in einer Funktion nie größer als Parameter B sein darf).

Für gewöhnlich bereitet man entsprechende Funktionen so auf, dass sie gerade dann arbeiten, wenn die Parameter in einem zulässigen Bereich liegen – was häufig gespart wird, ist allerdings eine explizite Abfrage, ob Parameter tatsächlich gültige Werte enthalten.

Die Folge sind schwer zu lokalisierende Fehler, wenn ein Parameter aus irgendwelchen Gründen einen ungültigen Wert enthält, die Berechnung innerhalb der Funktion ungültig macht und somit die Weiterverarbeitung im Rest des Programms ebenfalls inkorrekt verläuft.

An solchen Stellen bietet es sich an, so genannte Zusicherungen zu treffen.

Eine Zusicherung ist eine Aussage, die gültig sein muss, sobald das Programm an einer bestimmten Stelle ankommt. Man unterscheidet in der Informatik zwischen punktuellen und Gültigkeitsbereichs-Zusicherungen.

Während erstere tatsächlich nur bei ihrer Überprüfung gültig sein müssen, haben die globaleren Versionen während eines gesamten Blocks ihren Wahrheitsgehalt zu behalten.

Das *ASSERT*-Makro der MFC stellen eine Implementation der punktuellen Zusicherungen dar. Mit seiner Hilfe können Sie beispielsweise beim Eintritt in eine Funktion prüfen, ob alle Parameter in einem gültigen Bereich liegen oder ob während einer Berechnung nicht erlaubte Werte entstanden sind.

Anwendungsbeispiel

Ein Beispiel hierfür implementieren Sie jetzt in der Methode *OnBnClickedAssertTest*:

```
void CDebugProjectDlg::OnBnClickedAsserttest()
{
   // Variablen vorbereiten
   int nEineZahl = 10;
   int nNochEineZahl = 25;
```

```
    // Zwischenrechnung durchführen
    int nErgebnis = nNochEineZahl – nEineZahl – 15;

    // Zusicherung prüfen
    ASSERT (nErgebnis != 0);

    // "gefährliche Rechnung" durchführen
    int nEndergebnis = 333 / nErgebnis;


}
```

Hier wird eine fehlerhafte Berechnung erzwungen, *nErgebnis* erhält den Wert 0, was für die nachfolgende Division fatale Folgen hat: Eine Divison durch 0 ist nicht erlaubt.

Das eingefügte *ASSERT*-Makro prüft, ob dieser ungültige Divisor vorliegt.

Beim Austesten des entsprechenden Buttons erhalten Sie die folgende Fehlermeldung:

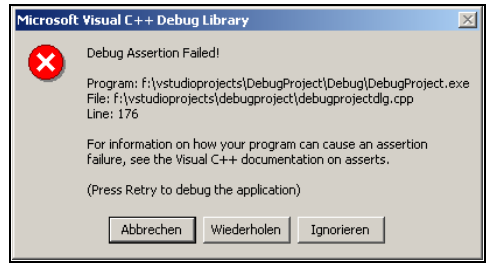


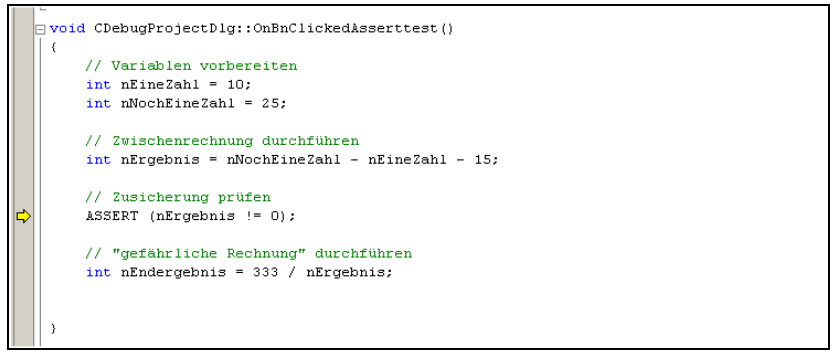
Abb. 8.5
Eine Assertion konnte nicht eingehalten werden

Das Fenster sagt aus, dass eine Zusicherung nicht eingehalten werden konnte. Das Dialogfeld enthält ferner nähere Angaben darüber, an welcher Stelle dieses Problem aufgetaucht ist. Wesentlich aussagekräftiger ist aber das Resultat, wenn Sie auf die *Wiederholen*-Schaltfläche drücken, die es Ihnen gestattet, die Anwendung zu debuggen – da Sie die Applikation ohnehin schon im Debugger gestartet hatten, werden Sie nur an die betreffende Quelltextzeile gebracht, die den Fehler auslöste, ansonsten würde nun eine neue Visual Studio-Instanz geöffnet werden.

Ergebnis des Testlaufs

Die fehlerhafte Quelltextzeile ist durch einen gelben Pfeil markiert:

Abb. 8.6
Die gescheiterte Zusicherung



Der zeigt Pfeil an, welche Zeile gerade ausgeführt wurde – gelbe Pfeile begegnen Ihnen vor allem auch noch beim manuellen Debuggen, zu dem wir später kommen werden.

Das ist doch schon recht aussagekräftig: *nErgebnis* hat den Wert 0, also darf ab hier die weitere Programmausführung nicht fortgesetzt werden. Es liegt jetzt am Entwickler herauszufinden, wie dieser ungültige Wert zustande gekommen ist.

ASSERT-Makros werden, wie die *TRACE*-Makros, automatisch aus Releaseversionen entfernt.

ASSERT_VALID und *AssertValid*

Erweiterung der *ASSERT*-Möglichkeiten

Während *ASSERT*-Makros gute Dienste Versehen, wenn es um das Überprüfen einzelner Variablenwerte oder sonstiger Programmzustände geht, wird schnell klar, dass das Überprüfen komplexer Datentypen mit ihnen deutlich unschöner zu handhaben ist.

Diesen Umstand haben auch die MSVC++ Entwickler erkannt und ein weiteres Makro in die Debug-Makro-Familie eingeführt: *ASSERT_VALID*.

ASSERT_VALID ist prinzipiell nur ein Relaismakro, denn es übernimmt einen Zeiger auf ein Objekt und ruft eine Methode namens *AssertValid* innerhalb dieses Objekts auf.

AssertValid ist eine Basisklassenmethode von *CObject* aus dem MFC-Anwendungsgerüst, sodass Klassen, die dieses Feature nutzen wollen, ebenfalls von *CObject* abgeleitet sein müssen.

Es wird dann die *AssertValid*-Methode geeignet überschrieben.

Ein Beispiel: fügen Sie die nachstehende Klassendeklaration in die Datei *DebugProjectDlg.h* ein – die Klasse wird auf den folgenden Seiten noch hin und wieder weitergehend erweitert werden, um andere Debugmechanismen zu erläutern.

Listing 8.2
Ein Testobjekt für Debug-Demonstrationen

```
class CTestObjekt : public CObject
{
private:
   int m_nEineZahl;
   char m_lpszEinString[MAX_PATH];

public:
   CTestObjekt();
   void AssertValid() const;

};
```

Diese Klasse enthält zwei Membervariablen sowie eine *AssertValid*-Methode, die als *const* zu deklarieren ist.

Die Implementation der Klasse fügen Sie der Einfachheit halber in die Datei *DebugProjectDlg.cpp* ein:

Listing 8.3
Implementationen des Testobjekts

```
CTestObjekt::CTestObjekt()
{
   // Variablenwerte initialisieren
   m_nEineZahl = 13;
   strcpy(m_lpszEinString, "Ein Teststring");
}
void CTestObjekt::AssertValid() const
{
   // AssertValid der Basisklasse aufrufen
   CObject::AssertValid();

   // Zusicherungen prüfen
   ASSERT (m_nEineZahl == 13);
   ASSERT (!strcmp(m_lpszEinString, "Ein anderer
Teststring"));
}
```

Der Konstruktor weist den Membervariablen Ausgangswerte zu, in *AssertValid* kommt erneut das schon bekannte *ASSERT*-Makro zum Zuge, das hier die einzelnen Membervariablen auf gültige Werte überprüft – es ist leicht zu sehen, dass bereits die Initialisierung innerhalb des Konstruktors fehlerhaft war, denn die beiden verglichenen Strings sind nicht identisch.

Beliebige Überprüfungen

Sie können hier natürlich jede beliebige Art von Überprüfungen durchführen, beispielsweise auch das Aufrufen von Membermethoden des Objekts, um festzustellen, ob diese gültige Werte zurückliefern und dergleichen mehr.

Zum Austesten der Funktionalität ergänzen Sie jetzt die Methode *OnBnClicked-Assertvalidtest* der *CDebugProjectDlg*-Klasse:

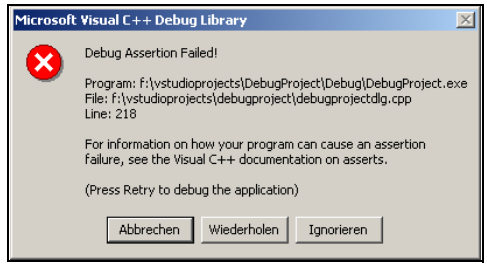
```
void CDebugProjectDlg::OnBnClickedAssertvalidtest()
{
   // zu prüfendes Objekt erzeugen
   CTestObjekt TestObjekt;

   // Objekt auf Gültigkeit überprüfen
   ASSERT_VALID(&TestObjekt);
}
```

Dem *ASSERT_VALID*-Makro übergibt man ein Zeiger auf das zu überprüfende Objekt. Interessanterweise wird hierbei gleichzeitig verifiziert, dass der Zeiger nicht *NULL* ist. Das ist insbesondere dann interessant, wenn mit dynamischen Objekten gearbeitet wird, und erspart eine weitere Prüfzeile.

Das *ASSERT_VALID*-Makro führt daraufhin die *AssertValid*-Methode des übergebenen Objekts aus.

Starten Sie jetzt das Debug-Testprogramm und wählen Sie den *ASSERT_VALID* Test aus:

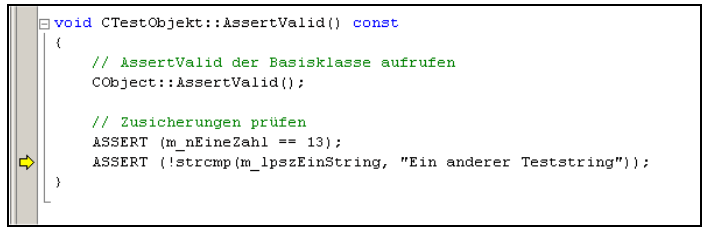
Abb. 8.7
Das Ergebnis des *ASSERT_VALID*-Tests



Ergebnis von *ASSERT_VALID*

Das Ergebnis entspricht dem von *ASSERT* – was auch leicht nachzuvollziehen ist, da *AssertValid* auf *ASSERT*-Makros aufbaut. Bei einem Betätigen der *Wiederholen*-Schaltfläche landen Sie daher praktischerweise auch nicht direkt beim *ASSERT_VALID*-Aufruf, sondern tatsächlich bei der gescheiterten Zusicherung:

Abb. 8.8
Die gescheiterte Zusicherung



ASSERT_VALID ist, wie *ASSERT*, nur in Debug-Builds vorhandenen und wird in Releaseversionen automatisch entfernt.

VERIFY

Zusicherungen im Debug-Modus

Eine etwas eigenartige Stellung übernimmt das *VERIFY*-Makro. In der Debug-Version eines Programms legt es das gleiche Verhalten an den Tag wie ASSERT, liefert also eine *Assertion Failure*-Dialogbox, falls eine Zusicherung nicht gültig war.

In der Releaseversion hingegen wird zwar das umschließende *VERIFY*-Makro entfernt, nicht aber die zu prüfende Zusicherung.

Dieses erklärt sich am besten anhand eines Beispiels. Editieren Sie dazu die Methode *OnBnClickedVerify*:

Listing 8.4
Beispiel für das *VERIFY*-Makro

```
void CDebugProjectDlg::OnBnClickedVerifytest()
{
   // Variablen initialisieren
   int m_nZahl = 0;
   int m_nZweiteZahl;

   // Prüfung durchführen (m_nZahl darf nicht 0
   // sein)
   VERIFY(m_nZweiteZahl = m_nZahl);
}
```

Wenn Sie das Programm in der Debug-Version ausführen, erhalten Sie die bekannte Meldung, da der Wert einer Zuweisung immer der Wert der rechten Zuweisungsseite – und somit in diesem Fall 0 – ist:

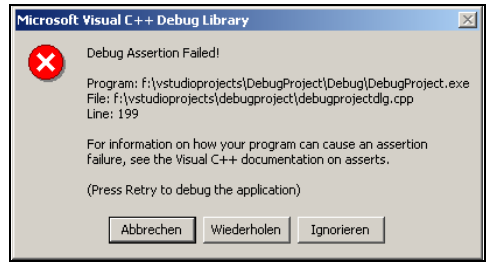
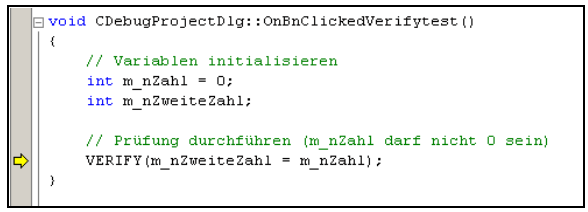


Abb. 8.9
Die bekannte *Assertion Failure*-Box

Die *Wiederholen*-Schaltfläche bringt Sie entsprechend an die Fehler auslösende Zeile:

Abb. 8.10
Ungültige Zusicherung



Auswirkungen in der Releaseversion

Der Unterschied zeigt sich erst, wenn das Programm in der Releaseversion erzeugt wird. Dann nämlich stehen im Quelltext anstelle der obigen Variante die folgenden Zeilen:

Listing 8.5
Die Releaseversion von *VERIFY*

```
void CDebugProjectDlg::OnBnClickedVerifytest()
{
   // Variablen initialisieren
   int m_nZahl = 0;
   int m_nZweiteZahl;

   // Prüfung durchführen (m_nZahl darf nicht 0
   // sein)
   m_nZweiteZahl = m_nZahl;
}
```

Die ursprüngliche Zusicherung selbst bleibt also erhalten. In der Regel benutzt man das *VERIFY*-Makro tatsächlich für Zuweisungen, bei denen nicht 0 oder NULL zugewiesen werden darf.

Speicherlecks

Eins der größten Ärgernisse beim Entwickeln von Applikationen sind auftretende Speicherlecks (engl. Memory Leaks), die immer dann entstehen, wenn ein Objekt dynamisch mit *new* angelegt, danach aber nicht wieder gelöscht wird.

Unerkannte Speicherprobleme

Das größte Problem bei diesem Speicherverschnitt ist es, dass er zunächst gar nicht erst erkannt wird – möglicherweise auch nicht, wenn ein Produkt bereits ausgeliefert wurde. Wer frühe Versionen von Blizzards Diablo II gespielt hat, kennt sicherlich das Phänomen, dass das Spiel im Laufe der Zeit immer langsamer wurde. Das hing gerade mit einem solchen massiven Speicherleck zusammen, dass anfangs keine Probleme bereitete, dann aber immer stärker die Performance schrumpfen ließ.

Zum Glück erkennt das Visual Studio bereits selbstständig, wenn Speicherlecks in MFC-Anwendungen entstehen und gibt entsprechende Meldungen mit Angaben der auslösenden Zeilen aus.

Zum Testen muss die Methode *OnBnClickedSpeicherlecktest* überarbeitet werden:

```
void CDebugProjectDlg::OnBnClickedSpeicherlecktest()
{
   // Speicherleck erzeugen
   for (int i=0;i<10;i++)
   {
      int *pZahl = new int;
   }
}
```

Listing 8.6
Erzwingen eines Speicherlecks

Hier wird ein Speicherleck erzwungen, indem zehn Integervariablen auf dem Heap angelegt, aber nicht wieder gelöscht werden.

Testen Sie das Programm aus. Drücken Sie die Speicherlecktaste und beenden Sie die Applikation dann wieder), erhalten Sie im Ausgabefenster Zeilen ähnlich der in *Abbildung 8.11*:

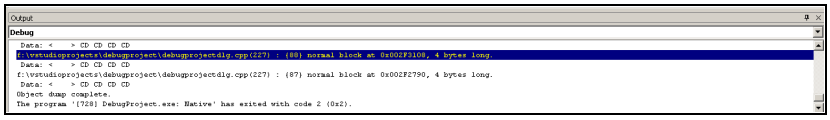


Abb. 8.11
Speicherlecks werden angezeigt

Ein Anklicken einer solchen Zeile (in *Abbildung 8.11* ist eine Meldung exemplarisch markiert), bringt Sie sogleich in den betreffenden Quelltextbereich:

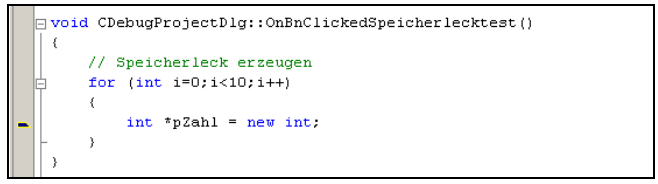


Abb. 8.12
Die verursachende Zeile

Da das Programm zu diesem Zeitpunkt nicht mehr läuft, wird auch kein gelber Pfeil angegeben, sondern ein gelb umrandeter schwarzer Marker, der gerade die verursachende Zeile darstellt.

Trotz dieses Hilfsmittels ist es manchmal recht schwer festzustellen, warum das Objekt nun nicht freigeräumt wird, aber das Erkennen der Problematik als solcher ist allein schon Gold wert.

afxDump

Erweiterung des *TRACE*-Makros

Wie beim *ASSERT*-Makro in Form des *ASSERT_VALID*-Makros gibt es auch für *TRACE* eine mächtigere Variante, die nicht nur einzelne Zeilen, sondern Statusinformationen über komplette Objekte mit einem einzigen Aufruf darstellen kann: *afxDump*.

Das *afxDump*-Objekt

afxDump ist ein Objekt, dass ähnlich *cout* verwendet werden kann, und mittels der <<-Operators übergebene Daten in den Ausgabebereich von Visual Studio .NET transferieren kann.

So können Sie einfache Strings an *afxDump* übergeben, wie zum Beispiel:

```
afxDump << "Eine Textzeile\n";
```

Das ist allerdings noch nicht sehr spannend, interessanter ist es, selbst definierte Objekte auszugeben. Die auszugebenden Objekte müssen eine Methode *Dump* enthalten, deren Prototyp Sie in der erweiterten Klassendeklaration zu *CTestObjekt* wiederfinden (Datei *DebugProjectDlg.h*)

Listing 8.7 **Erweiterung der *CTest-Objekt*-Klassendeklaration**

```
class CTestObjekt : public CObject
{
private:
   int m_nEineZahl;
   char m_lpszEinString[MAX_PATH];

public:
   CTestObjekt();
   void AssertValid() const;
   void Dump(CDumpContext &_dumpContext) const;

};
```

Die *Dump*-Methode erhält als Parameter eine Referenz auf ein *CDumpContext*-Objekt – dieser ist für Sie aber nicht weiter relevant, arbeiten Sie einfach mit dem *afxDump*-Objekt innerhalb der Implementation der Dumpmethode, wie es in der Beispielimplementation von *CTestObjekt::Dump* aufgezeigt wird (diese gehört in die Datei *DebugProjectDlg.cpp*):

Listing 8.8 **Die *Dump*-Methode**

```
void CTestObjekt::Dump(CDumpContext &_dumpContext)
const
{
   // Ausgabe der internen Daten
   afxDump << "\nDaten des CTestObjekts:\n";
   afxDump << "------------------------\n";
   afxDump << "m_nEineZahl= "
<< m_nEineZahl << "\n";
   afxDump << "m_lpszEinString= "
<< m_lpszEinString << "\n";
}
```

Wie Sie sehen können, werden hier einfach die Daten der Reihe nach über den <<-Operator an das *afxDump*-Objekt übertragen.

Inwieweit Sie sich hier um eine sinnvolle Formatierung kümmern, ist Ihnen selbst überlassen, empfohlen wird aber, zunächst eine Objekttitelzeile auszu-

geben und dann pro Zeile möglichst nur einen Wert, oder aber eine Reihe von zusammengehörenden Werten (beispielsweise X- und Y-Koordinaten eines Punkts) auszugeben .

Sie können die Ausgabe leicht forcieren, indem Sie die *OnBnClickedAfxdumptest*-Methode geeignet ergänzen:

```
void CDebugProjectDlg::OnBnClickedAfxdumptest()
{
   // Objekt anlegen
   CTestObjekt TestObjekt;

   // Ausgabe der Inhalte im Debugmode
#ifdef _DEBUG
   afxDump << TestObjekt;
#endif
}
```

Bedingte Verwendung von *afxDump*

Hier fällt auf, dass die Ausgabe nur dann ausgeführt wird, wenn das *_DEBUG*-Symbol definiert ist, was in der Regel nur bei Debug-Versionen der Fall ist.

Der Grund dafür ist einfach der, dass das *afxDump*-Objekt in Releasebuilds nicht zur Verfügung steht, weshalb die Release-Kompilierung an dieser Stelle fehlschlagen würde.

Sorgen Sie also immer dafür, dass *afxDump*-Operationen von einem *#ifdef ... #endif*-Konstrukt umschlossen sind, so laufen Sie nicht Gefahr, im Nachhinein eine Unmenge von Kompilierfehlern bei der Übersetzung einer Releaseversion beheben zu müssen.

Starten Sie das Programm nun im Debug-Modus und klicken Sie die *afxDump*-Schaltfläche an. Das Ergebnis entspricht der *Abbildung 8.13*.

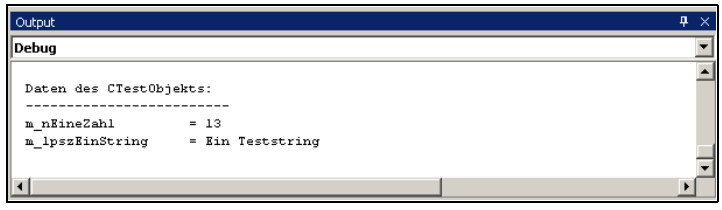


Abb. 8.13 **Ausgabe mittels *afxDump***

Damit haben Sie den ersten Teil des Debugging-Abschnitts erfolgreich hinter sich gebracht.

Es sei angeraten, dass Sie sich die einzelnen Möglichkeiten verinnerlichen und auch ausgiebig einsetzen. Jeder Entwickler macht Fehler, umso wichtiger ist es, diese auch als solche zu erkennen und vor allem frühzeitig über sie in Kenntnis gesetzt zu werden.

Es ist allemal als Vorteil anzusehen, jeden Tag mit diesen Mitteln zehn Fehler zu erkennen, als nach der Auslieferung eines Produkts zehn Anrufe von empörten Anwendern zu erhalten, die den Fehler vor Ihnen gefunden haben.

Debuggerinterne Überprüfungen

Der zweite große Abschnitt beim Finden von Fehlern wird Ihnen praktischerweise ohne Kosten in Form eigener Arbeitszeit geliefert: Debuggerinterne Laufzeitüberprüfungen, die in Debug-Versionen von Applikationen automatisch durchgeführt werden.

Debug-Versionen

Der Preis, den Sie hierfür zahlen müssen, ist eine geringfügig größere Ausgabedatei (in Form von EXE- beziehungsweise DLL-Dateien) und eine leicht niedrigere Performance, doch sind dieses Opfer, denen Sie sich grundsätzlich während der Entwicklungszeit aussetzen sollten.

Gerade in der neuen Version bietet das Studio durch eine Überprüfung, ob Variablen vor einer Verwendung definiert wurden, einen weiteren Grund an, überhaupt nicht mit Releaseversionen zu arbeiten, solange nicht der Großteil an Funktionalität erfolgreich implementiert wurde.

In früheren Zeiten haben Entwickler gerne hin und wieder Releasebuilds erzeugt, um auf diesem Wege herauszufinden, ob an irgendwelchen Stellen Variablen undefiniert verwendet wurden – was sich meistens in Abstürzen während der Programmausführung äußerte.

Undefinierte Variablen

Mit diesem Problemfall soll dann auch direkt eingestiegen werden. Editieren Sie die Methode *OnBnClickedUndefvariabletest* in der Datei *DebugProjectDlg.cpp*:

Listing 8.9 **Verwenden einer nicht definierten Variable**

```
void CDebugProjectDlg::OnBnClickedUndefvariabletest()
{
   // Variable deklarieren
   int nEineZahl;

   // Variable NICHT setzen
   if (1 < 0)
   {
      nEineZahl = 13;
   }

   // Undefinierte Variable verwenden
   int nRechnung = 5 / nEineZahl;
}
```

Hier wird die Integervariable *nEineZahl* nicht definiert, da die Bedingung im *if*-Konstrukt nicht erfüllt ist. Die Folge ist, dass die Berechnung hinter dem Block ein undefiniertes Ergebnis erhalten würde.

Starten Sie die Debuganwendung und wählen die Sie *Undef. Var.*-Schaltfläche aus – Sie erhalten umgehend eine Meldung des Debuggers:

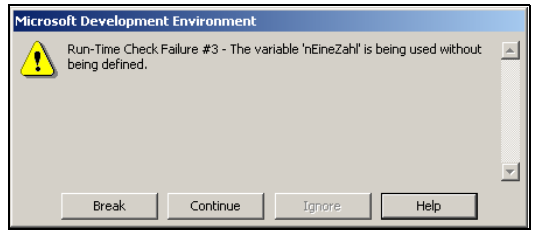


Abb. 8.14
Eine nicht definierte Variable wurde entdeckt

Der Debugger moniert hier die Verwendung einer nicht definierten Variable namens *nEineZahl*. Das sind schon hilfreiche Informationen, die durch das Anklicken der *Break*-Schaltfläche noch sinnvoll ergänzt werden können:

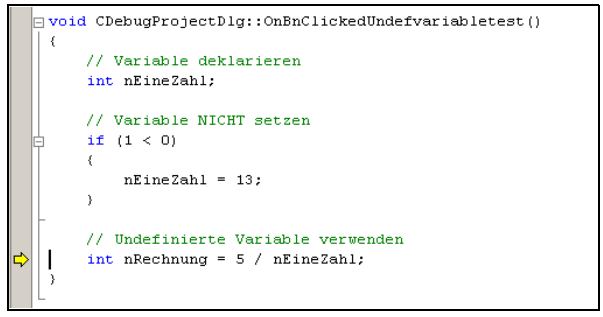


Abb. 8.15
Die fehlerhafte Zeile

Sie landen direkt im Quelltexteditor an der Stelle, an der die undefinierte Variable eingesetzt wurde. Es ist nun in der Regel einfach festzustellen, wo der Wertespeicher deklariert aber nicht initial definiert wurde.

Undefinierte Variablen belegen

Das Hauptproblem bei der Behebung dieses Fehlers besteht nun darin, einen geeigneten Initialisierungswert zu finden, doch sollte sich dieser im Normalfall in den meisten Fällen von selbst ergeben.

Überlegen Sie jedoch einmal, wie lange man einen solchen Fehler ohne dieses nützliche Feature vergeblich suchen könnte.

Korruption des Laufzeitstacks

Eine weitere potenzielle Fehlerquelle tut sich bei der Arbeit mit dem Laufzeitstack auf. Der Laufzeitstack, den in früheren Jahren noch jeder Entwickler in und auswendig kannte, ist heutzutage nur noch wenigen Programmierern ein Begriff.

Wenn dann auch noch Programmfehler auftauchen, die in irgendeiner Form mit diesem seltsamen Konstrukt zu tun haben, sorgt das häufig für extreme Konfusion und Verwirrung.

Bedeutung des Laufzeitstacks

Der Laufzeitstack eines Programms ist im Wesentlichen für zwei Dinge zuständig: Zum einen werden auf ihm sämtliche lokalen Variablen angelegt, zum anderen führt er Buch darüber, welche Parameter an eine Funktion übergeben werden und wohin eine aufgerufene Routine nach Beendigung derselben zurückspringen soll.

Das Laufzeitstack arbeitet dabei auf Basis eines so genannten FILO-Stapels. FILO steht für First In, Last Out (zuerst rein, als Letztes heraus) und beschreibt damit das Verhalten des Stacks, wenn Daten auf ihm abgelegt und wieder heruntergeholt werden.

Stellen Sie sich den Stack einfach als eine Sammlung von Blättern vor, die auf einem Schreibtisch liegen. Immer, wenn Sie ein Datum auf den Stack legen, wächst der Stapel. Das erste Blatt, das Sie auf ihm platzieren, wird aber erst als Letztes wieder heruntergenommen, wenn beispielsweise zehn Werte auf dem Stack abgelegt und danach wieder ausgelesen werden.

Die genauen Details der Stacks sind an dieser Stelle nicht wichtig, wichtig ist aber zu wissen, dass Sie die ganze Zeit mit einem solchen Anwendungsstapel arbeiten, wenn lokale Variablen innerhalb einer Funktion angelegt werden.

Beispiel eines Stackfehlers

Dabei kann es zu Stackfehlern kommen, die in der Methode *OnBnClickedStackcorruptiontest*, die Sie jetzt entsprechend auffüllen, demonstriert werden:

***Listing 8.10* Beispiel eines Stapelfehlers**

```
void
 CDebugProjectDlg::OnBnClickedStackcorruptiontest()
{
   // Stringvariable anlegen
   char lpszText[5];

   // Bereich überschreiben
   strcpy(lpszText, "Ein zu langer Text");
}
```

Hier wird ein Chararray von fünf Zeichen Länge angelegt – und zwar auf dem Stack, wie wir gerade eben gelernt haben.

Die nächste Zeile kopiert eine deutlich längeren String in dieses Array und zerstört damit eben diesen Stack.

Der Debugger moniert dieses Problem aber sofort mit einer zutreffenden Aussage, wenn Sie das Programm jetzt starten und die passende Schaltfläche auswählen:

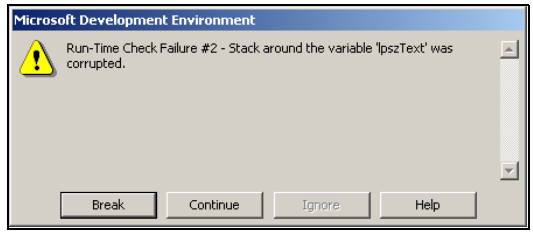


Abb. 8.16 **Anzeigen des Stackfehlers**

Neben der generellen Aussage, dass der Laufzeitstack beschädigt wurde, wird auch gleich noch gesagt, im Bereich welcher Variablen dieses Problem aufgetreten ist – in diesem Fall gerade zufälligerweise beim Chararray *lpszText*.

Auch hier führt Sie das Anklicken der *Break*-Schaltfläche an die passende Quelltextzeile:

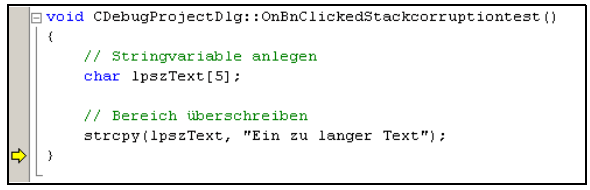


Abb. 8.17 **Der Fehler im Quelltext**

Hier wird allerdings interessanterweise nicht die Zuweisung des Strings oder besser der entsprechende Kopiervorgang angezeigt, sondern die letzte Zeile der Methode, die gerade aus der schließenden geschweiften Klammer besteht.

Abräumen des Stacks

Auf dem Stack angelegte lokale Variablen werden beim Verlassen einer Funktion wieder vom Stack geräumt – das ist vergleichbar mit dem Entnehmen der obersten Blätter vom weiter oben beschriebenen Blattstapel.

Erst hier kann der Debugger bemerken, dass eine Korruption in Form von überschriebenem Stackbereich durchgeführt wurde.

Lassen Sie sich also nicht von der seltsam anmutenden Quelltextposition irritieren – irgendwo innerhalb dieser Funktion wurde nicht ordnungsgemäß auf dem Stack gearbeitet, was in 95 Prozent aller Fälle auf eine unsachgemäße Stringkopieraktion zurückzuführen sein dürfte.

Beschädigung des Heaps

Was ist der Heap?

Wenn Probleme auf dem Stack auftauchen können, so gilt dieses auch für den Heap. Der Heap (zu deutsch etwa Haufen) ist der große Rest des Speichers, der von Applikationen genutzt werden kann, um Daten abzulegen.

Hier landen sämtliche Daten, die beispielsweise mittels *new* angelegt werden – ein Problem des Stacks ist es nämlich, dass er nicht unendlich groß ist, sondern eine fest vorgegebene Größe hat (die allerdings über geeignete Compilereinstellungen verändert werden kann).

Beliebte Fehler, die bei der Arbeit mit dem Heap auftreten bestehen darin, dort erzeugte Objekte mehrfach zu löschen. Aber auch dieses Problem wird vom Visual Studio Debugger abgefangen.

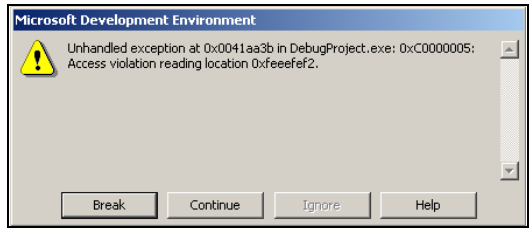
Dazu ein weiteres Beispiel, das dieses Mal in die Methode *OnBnClickedHeaptest* gehört:

***Listing 8.11* Beschädigen des Heaps**

```
void CDebugProjectDlg::OnBnClickedHeaptest()
{
   // Objekt auf Heap erzeugen
   CTestObjekt *pNewObject = new CTestObjekt;

   // Objekt abräumen
   delete (pNewObject);

   // Objekt erneut abräumen
   delete (pNewObject);
}
```

Die Methode legt ein Objekt mittels *new* auf dem Heap an und löscht es danach zweimal. Dieses führt, nach Kompilieren und Starten der neuen Programmversion, zu der Benachrichtigung über eine unbehandelte Ausnahme durch den Debugger:

***Abb. 8.18* Probleme auf dem Heap**



Diese Fehlermeldung ist für den Laien vielleicht nur schwer lesbar, deutet aber doch darauf hin, dass eine nicht korrekte Aktion im Speicher durchgeführt wurde.

Zum Glück wird diese Annahme durch einen Blick auf die Quelltextzeile, zu der Sie durch Anklicken der Break-Schaltfläche geführt werden, bestätigt:

```
void CDebugProjectDlg::OnBnClickedCasttest()
{
    // Objekt auf Heap erzeugen
    CTestObjekt *pNewObject = new CTestObjekt;

    // Objekt abräumen
    delete (pNewObject);

    // Objekt erneut abräumen
    delete (pNewObject);
}
```

Abb. 8.19
Das zweite Löschen ist ungültig

Speicherkorruption durch Löschvorgänge

Es ist nun klar ersichtlich, dass aufgrund des zweiten Löschvorgangs offensichtlich eine Beschädigung des Heaps aufgetreten ist.

Solche Aktionen können ganz unvorhergesehene Folgen habe: durch den zusätzlichen Löschvorgang werden Operationen in Gang gesetzt, die den Heap in gewissen Bereichen zerstören.

Im besten Fall merken Sie hiervon nichts, doch kann es so leicht geschehen, dass die Verwaltungstabellen, die Buch darüber führen, welche Speicherbereiche verwendet werden und welche noch frei sind, ebenfalls in Mitleidenschaft gezogen werden.

Die Folge davon kann sein, dass das Neuanlegen von weiteren dynamischen Objekten fehlschlägt. Sie erkennen dieses in der Regel daran, dass ein Aufruf von *new* NULL zurückliefert – nur in den wenigsten Fällen hängt dieses damit zusammen, dass tatsächlich kein Speicher mehr zur Verfügung steht.

Die Arbeit mit dem Heap, oder besser: Das Aufspüren von mit dem Heap direkt zusammenhängenden Fehlern gehört zu den zeitaufwendigsten Debug-Tätigkeiten eines jeden Programmierers.

Ordnung bei der Arbeit mit dem Heap halten

Achten Sie stets darauf, immer sorgfältig Buch zu führen, welche Objekte erzeugt wurden und welche bereits zerstört sind – ein laxes Umgehen mit dieser Thematik ist definitiv fehl am Platze.

Ebenso mit Vorsicht zu genießen sind die Worte eines unbekannten Programmierers, der zu dieser Thematik einmal sagte: „Im Zweifelsfall ist ein Speicherleck alle mal einem Heap-Fehler vorzuziehen. Das Auslassen von *delete*-Operationen kann hier manchmal Wunder wirken“.

Lassen Sie sich nicht zu einer solchen Programmiertechnik verleiten, sorgen Sie jedoch auf der anderen Seite auch dafür, dass stets geflissentlich mit dynamischen Objekten umgegangen wird.

Nicht initialisierte Zeiger und Standardwerte

Wir hatten vorhin bereits erlebt was passiert, wenn der Debugger über eine nicht initialisierte Variable läuft, die innerhalb eines Ausdrucks verwendet werden soll: Es wird eine entsprechend Fehlermeldung ausgegeben und die Programmausführung unterbrochen.

Standardwerte für nicht initialisierte Variablen

Bei nicht initialisierten Variablen – und hier insbesondere bei Zeigern – gibt es aber noch eine weitere Nettigkeit, die durch den Debugger des Visual Studios realisiert wird: die Initialisierung mit Standardwerten.

Im Debug-Modus eines Programms werden nämlich sämtliche Elemente mit vermeintlich sinnlosen Daten initialisiert. Das hat den Vorteil, dass Sie zum Beispiel bei der Arbeit mit Zeigervariablen sehr schnell erkennen können, wenn Sie es mit uninitialisierten Werten zu tun haben.

Bei der Releaseversion eines Programms ist dieses nicht der Fall, dort stehen beliebige Werte in den jeweiligen Variablen, insbesondere nämlich die, die gerade an der Stelle im Speicher standen, an der die neue Variable gerade angelegt wurde.

Zur Demonstrierung der Zeigerinitialisierung ergänzen Sie den Funktionsrumpf der Methode *OnBnClickedNichtinitialisiert* gemäß dem folgenden Listing:

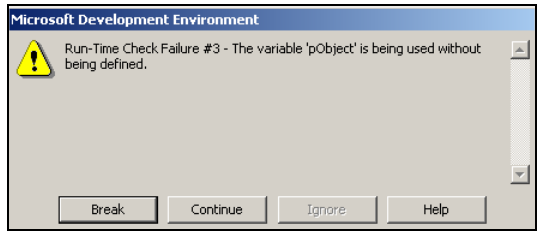
Listing 8.12
Verwendung eines nichtinitialisierten Zeigers

```
void
CDebugProjectDlg::OnBnClickedNichtinitialisierttest()
{
   // Zeiger anlegen
   CTestObjekt *pObject;

   // Funktion aufrufen
   pObject->AssertValid();
}
```

In dieser Methode soll ein nicht initialisierter Zeiger verwendet werden, um mittels der schon bekannten Methode *AssertValid* zu prüfen, ob ein Objekt gültige Werte enthält.

Kompilieren und starten Sie das Programm, erhalten Sie die schon bekannte Ausgabe bezüglich einer nicht initialisierten Variablen, die verwendet werden soll:

Abb. 8.20
Nicht initialisierter Zeiger wird verwendet



Anzeige der relevanten Variablen

Brechen Sie die Programmausführung durch Anwahl des Punkts *Break* ab. Dieses Mal ist jedoch nicht die auslösende Zeile interessant, sondern vielmehr die Informationen, die Sie in einem weiteren Fenster der Debug-Umgebung finden: die Anzeige der gerade aktiven Variablen.

Sie können hier umstellen zwischen sämtlichen lokalen Variablen sowie einer automatischen Ansicht, die alle bislang verwendet Variablen auflistet, die im lokalen Kontext genutzt wurden oder noch in Benutzung sind.

Schalten Sie die Ansicht auf die Anzeige aller lokalen Variablen um, Sie finden dort eine Ausgabe ähnlich der folgenden:

Abb. 8.21 **Ausgabe der Variableninhalte**

Sie finden hier einen Eintrag namens *pObject*, dem ein Inhalt von *oxccccccccc* zugeordnet ist. Das es sich hierbei um einen hexadezimalen Wert handelt, ist nebensächlich, interessant ist die Art des Werts, der kaum zufällig entstanden sein kann, sondern eher den Eindruck erweckt, künstlich erzeugt worden zu sein.

Tatsächlich ist dieses eine Standardinitialisierung von Zeiger, die automatisch beim Anlegen einer Debug-Version erzeugt wird.

Die Standardwerte für Variablen

Es gibt weitere Werte dieser Art, wie zum Beispiel *oxcdcdcdcd* oder *oxefefefefef*, auf die Sie bei der Arbeit mit nicht initialisierten Daten stoßen werden.

Immer, wenn Sie es mit solchen offensichtlich künstlichen Werten zu tun haben, wissen Sie also, dass Sie keine eigene geeignete Initialisierung für die Daten spezifiziert haben – ein Umstand, der auf jeden Fall behoben werden sollte.

Stack Overflow

Größenbeschränkung des Stacks

Weiter oben wurde angedeutet, dass die Größe des Stacks nicht unbegrenzt ist. Obwohl sie zur Zeit der Kompilierung (man spricht hier von Compile Time) verändert werden darf, ist die sie doch stets einer oberen Grenze unterworfen.

Das bedeutet im Klartext, dass es nicht möglich ist, beliebig viele – oder besser: beliebig große – Objekte auf dem Stack zu erzeugen.

Ebenso heißt es aber, dass die Anzahl der Funktionsaufrufe, die sukzessiv ohne Rücksprünge durchgeführt werden können, begrenzt ist.

Wann immer der Laufzeitstack in seiner Größe gesprengt wird, gibt der Visual Studio .NET Debugger eine passende Fehlermeldung aus, die im Folgenden erzwungen werden soll.

Editieren Sie dazu zunächst in der Datei *DebugProjectDlg.h* die Klassendeklaration von *CTestObjekt* wie folgt:

Listing 8.13
Eine neue Klassendeklaration

```
class CTestObjekt : public CObject
{
private:
   int m_nEineZahl;
   char m_lpszEinString[MAX_PATH];

public:
   CTestObjekt();
   void AssertValid() const;
   void Dump(CDumpContext &_dumpContext) const;
   void NichtTerminieren(int in_nZahl);
};
```

Die neu eingefügte Methode soll sich selbst kontinuierlich aufrufen. Wie Sie inzwischen wissen, werden für jeden Funktionsaufruf die Parameter und die Rücksprungadresse auf dem Stack abgelegt.

Das heißt in der Folge, dass durch das sukzessive Aufrufen einer Funktion, die sich selbst ständig aufruft, der Stack ständig erweitert wird, bis er schließlich seine maximale Größe sprengt.

Implementieren einer nicht terminierenden Funktion

Die Implementation der *NichtTerminieren*-Methode soll nun dargestellt werden, sie gehört in die Datei *DebugProjectDlg.cpp*:

Listing 8.14
Implementation von *NichtTerminieren*

```
void CTestObjekt::NichtTerminieren(int in_nZahl)
{
   NichtTerminieren(in_nZahl + 1);
}
```

Diese Methode macht nichts anderes, als sich selbst aufzurufen, und den ihr dabei übergebenen Parameter um eins zu erhöhen.

Das initiale Aufrufen der Methode gehört in ebenfalls in die Datei *DebugProjectDlg.cpp* und ist in der Behandlungsmethode *OnBnClickedStacktest* abzulegen:

Listing 8.15
Aufrufen der nicht terminierenden Funktion

```
void CDebugProjectDlg::OnBnClickedStacktest()
{
   // Testobjekt anlegen
   CTestObjekt NewObjekt;

   // nicht terminierende Methode aufrufen
   NewObjekt.NichtTerminieren(0);
}
```

Die Methode erzeugt zunächst ein neues Objekt vom Typ *CTestObjekt* auf dem Stack – es handelt sich ja um eine lokale Variable – und ruft dann ihre Methode *NichtTerminieren* mit einem Startwert von 0 auf.

Es ist nun interessant zu beobachten, welchen Wert der letzte Funktionsaufruf haben wird, denn daraus lässt sich direkt die Größe des Stacks ablesen und vielleicht auch erahnen, in welcher Geschwindigkeit Funktionsaufrufe über den Stack abgearbeitet werden können.

Hilfreiche Meldungen beim Kompilieren

In diesem trivialen Fall bräuchte man den Debugger des Visual Studio vermutlich nicht einmal bemühen, denn bereits beim Kompilieren wird eine Warnung ausgegeben, dass die Methode NichtTerminieren *nur sich selbst aufruft und keinerlei alternativen Abarbeitungspfade zum Verlassen der Methode besitzt – das Beispiel wurde in diesem Fall nur zur Veranschaulichung gewählt.*

In anderen, nicht trivialen Fällen, kann der Compiler die Problematik nicht erkennen, beispielsweise wenn sich zwei Methoden wechselseitig aufrufen, oder zwar eine Abbruchbedingung gegeben ist, diese aber niemals erreicht wird.

Kompilieren und starten Sie das Debug-Programm erneut und wählen Sie den Knopf zum Auslösen des *Stack-Overflow*-Fehlers.

Dieser präsentiert sich in einem gewohnt schlichten Fenster:

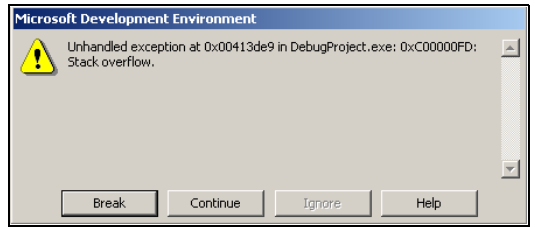


Abb. 8.22
Ein Stack Overflow ist eingetreten

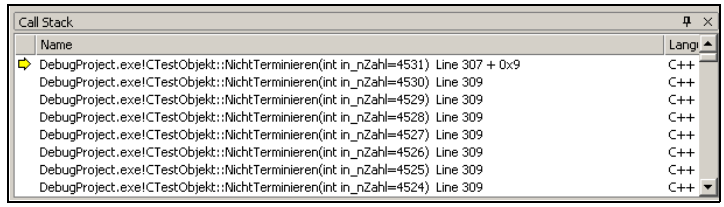
Der Stacküberlauf wurde erkannt, drücken Sie die Break-Schaltfläche und werfen einen Blick in den so genannten Call Stack. Der Call Stack enthält eine Übersicht über die letzten Aufrufe, die bis hin zu diesem Fehler geführt haben.

Anzeige eines Stacküberlaufs

Im Falle der nicht terminierenden Funktion ist zu sehen, dass sie offensichtlich mehrfach hintereinander aufgerufen wurde, bis es zum Stacküberlauf gekommen ist.

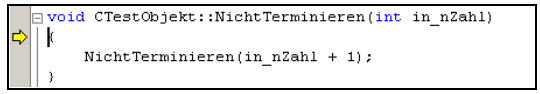
Insbesondere wurde die Methode über 4.500 Mal aufgerufen – eine gewaltige Zahl, wenn Sie bedenken, dass der Fehler quasi sofort ausgegeben wurde:

Abb. 8.23
Call Stack im Überblick



Interessant ist es weiterhin zu schauen, an welcher Stelle der Abbruch stattgefunden hat, dieses können Sie am gelben Pfeil innerhalb des Quelltextfensters gut erkennen:

Abb. 8.24
Abbruchstelle



Der Abbruch hat offensichtlich beim Eintreten in die Funktion stattgefunden. Das ist auch nicht anders zu erwarten, denn beim Aufruf der Funktion werden ja Daten auf dem Stack abgelegt – insbesondere der Parameter und die Rücksprungadresse.

Dieses sukzessive Vorgehen führte irgendwann zum Abbruch aufgrund eines Stack Overflow.

Zusammenfassung

Damit ist der zweite Abschnitt des Debugging-Kapitels abgeschlossen. Sie haben wiederum einige hilfereiche Funktionen des Debuggers kennen gelernt und erfahren, dass eine Reihe von Fehlern automatisch ohne Ihr Zutun erkannt werden.

Außerdem haben Sie zwei wichtige Fenster entdeckt, nämlich den Callstack und das Variablenausgabefenster. Sie werden diese beiden Fenster im Laufe der Entwicklungsarbeit mit dem Visual Studio .NET noch sehr zu schätzen lernen, genauso wie die anderen Möglichkeiten, denen sich der nächste Abschnitt widmen will.

Manuelle Fehlersuche

In gewissen Situationen helfen die besten Tools nichts oder nur wenig: Fehler, die nur in bestimmten Situationen eintreten oder nicht durch ungültige Vari-

ablenwerte hervorgerufen werden, sind mit konventionellen Mitteln nicht zu fassen – das Allheilmittel für das Debugging von fehlerverseuchten Programmen gibt es leider bislang noch nicht.

Mühsames Debugging per Hand

Hier muss man sich selbst behelfen und manuell durch die vermutlich fehlerhaften Portionen eines Quelltexts hindurcharbeiten – das kann im Rahmen von Windows-Anwendungen schon recht mühsam werden, Sie haben ja bereits festgestellt, dass einige Applikationen relativ umfangreich sind, insbesondere wenn man bedenkt, welche Quellcodeanteile die Anwendungsassistenten automatisch bereitstellen und wie viele Zeilen sich erst noch in den Basisklassen des MFC Frameworks verbergen.

Doch auch in solchen Situation kann der Visual Studio Debugger wertvolle Hilfe leisten.

Im Normalfall muss ja nicht ein komplettes Programm untersucht werden, sondern nur bestimmte Teilbereiche hieraus. Angenommen, Sie haben eine Funktion, die Sie für fehlerhaft empfinden, würde es ja reichen, die Programmausführung beim Eintritt in diese Funktion anzuhalten und dann schrittweise ausführen zu können.

Breakpoints

Zu diesem Zweck wurden vor langer Zeit die so genannten Breakpoints in die meisten Debugger integriert, die zu genau diesem Zweck die dienen. Mit ihnen beschäftigt sich ein Großteil dieses dritten und letzten Abschnitts.

Breakpoints

Ein Breakpoint (zu deutsch etwa: Abbruchpunkt) kann auf jede beliebige gültige Quellcodezeile gesetzt werden. Gültig heißt hierbei, dass die entsprechende Zeile auch noch eine gültige Repräsentation im kompilierten Programm aufweisen können muss.

Kommentarzeilen beispielsweise fallen nicht unter diesen Gesichtspunkt und auch die Deklaration von lokalen Variablen ohne zugehörige Initialisierung wird im Maschinencode sicherlich an den Einstieg einer Funktion gelegt und nicht wie in Ihren Quelltexten irgendwo mitten in der Methodenabarbeitung angesiedelt sein.

Wann treten Breakpoints in Aktion?

Wenn Sie ein Programm mit dem Debugger starten, läuft es solange in höchster Geschwindigkeit durch, bis es über einen Breakpoint läuft. Das heißt für den Debugger zunächst zu prüfen, ob er es mit einem bedingten oder einem unbedingten Abbruchpunkt zu tun hat.

Bedingte Abbruchpunkte unterliegen noch einer bestimmten Bedingung, die besagt, ob die Programmausführung unterbrochen werden soll oder nicht – beispielsweise könnte zunächst geprüft werden, ob eine Variable einen bestimmten Wert erreicht hat oder die entsprechende Quelltextzeile schon zum vierten Mal erreicht wurde.

Mit bedingten Breakpoints befassen wir uns ein wenig später.

Um überhaupt mit den Abbruchpunkten arbeiten zu können, soll zunächst eine kleine Testroutine geschrieben werden, anhand derer die wesentlichen Möglichkeiten beim manuellen Debuggen aufgezeigt werden sollen.

Eine Testmethode für die Abbruchpunkte

Editieren Sie also die Methode *OnBnClickedBreakpointtest* wie folgt:

Listing 8.16 **Die Methode zum Austesten der Breakpoints**

```
void CDebugProjectDlg::OnBnClickedBreakpointtest()
{
   // Variablendefinitionen
   char lpszTemp[10];
   strcpy(lpszTemp, "Hallo");

   int a = 10;

   CTestObjekt *pNewObject;

   // eine Schleife
   for (int i=0;i<20;i++)
   {
      a = a + 5;
      TRACE(_T("%d\n"), a);
   }

   // neues Objekt anlegen
   pNewObject = new CTestObjekt;

   if (NULL == pNewObject)
   {
      AfxMessageBox("Konnte Objekt nicht
anlegen");
      return;
   }

   // Objekt abräumen
   delete pNewObject;
}
```

Diese Methode ist relativ übersichtlich aufgebaut, in ihrer Funktion aber höchst ominös. Nach dem Initialisieren zweier Variablen, wird in einer Schleife ein Wertespeicher kontinuierlich um 5 erhöht.

Danach wird ein neues Objekt der Klasse *CTestObjekt* erzeugt, geprüft, ob das Anlegen erfolgreich war und direkt danach wieder gelöscht.

Egal, ob sinnvoll oder nicht, die Methode ist hervorragend dafür geeignet, die wichtigsten Möglichkeiten beim manuellen Debuggen zu erklären.

Als Erstes ist es notwendig, Breakpoints an interessante Stellen innerhalb der Methode zu setzen.

Setzen der Breakpoints

Einfügen von Breakpoints innerhalb des Visual Studio

Im Rahmen des Visual Studio können Sie Breakpoints setzen, indem Sie den kleinen grauen vertikalen Streifen am linken Rand jedes Quelltextfensters mit der linken Maustaste einmal anklicken – ein zweites Anklicken entfernt den Breakpoint wieder.

Setzen Sie einen Abbruchpunkt an eine ungültige Stelle, wird er spätestens beim nächsten Debugger_Start auf die nächste gültige Zeile verschoben. Markieren Sie nun beispielsweise die Zeilen

```
char lpszTemp[10];
...
a = a + 5;
...
pNewObject = new CTestObjekt;
```

```
void CDebugProjectDlg::OnBnClickedBreakpointtest()
{
    // Variablendefinitionen
    char lpszTemp[10];
    strcpy(lpszTemp, "Hallo");

    int a = 10;

    CTestObjekt *pNewObject;

    // eine Schleife
    for (int i=0;i<20;i++)
    {
        a = a + 5;
        TRACE(_T("%d\n"), a);
    }

    // neues Objekt anlegen
    pNewObject = new CTestObjekt;

    if (NULL == pNewObject)
    {
        AfxMessageBox("Konnte Objekt nicht anlegen");
        return;
    }
```

Abb. 8.25
Eingefügte Breakpoints in der Beispielmethode

Sie sehen das Ergebnis des Einfügens in der *Abbildung 8.25*.

Erreichen eines Breakpoints

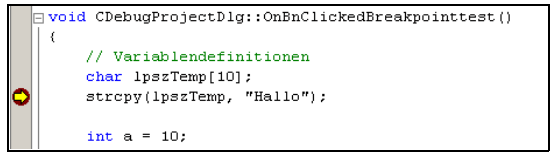
Kompilieren und starten Sie das Beispielprogramm. Wählen Sie aus dem Dialogfeld die Schaltfläche zum Austesten der Breakpoints aus.

Erreichen eines Breakpoints

Die Ausführung des Programms wird mehr oder weniger sofort beendet – Sie landen im Quelltexteditor, der erste von Ihnen gesetzte Breakpoint wurde scheinbar eine Zeile nach unten verschoben, liegt somit nicht mehr auf einer Variablendeklarationszeile, und enthält nun einen gelben Pfeil.

Dieser gelbe Pfeil gibt, wie bereits mehrfach angesprochen, die Zeile an, die als Nächstes ausgeführt werden soll. Eine erste Regel bei der Arbeit mit Abbruchpunkten ist also die, dass eine Zeile, die einen Breakpoint enthält nicht mehr abgearbeitet wird, bevor das Programm angehalten wird.

Abb. 8.26
Die Programmausführung wurde abgebrochen



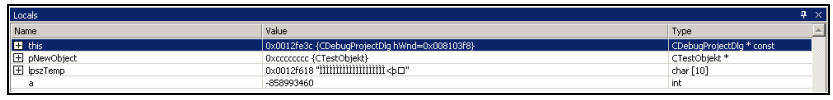
Werfen Sie einen Blick in das Anzeigefenster mit den Variablen, Sie finden dort die derzeitig aktuellen Werte. Dabei ist zunächst eine automatische Übersicht aktiviert:

Abb. 8.27
Automatische Variablenauswahl

Häufig ist diese Liste nicht ausreichend, beziehungsweise nicht erschöpfend genug. In diesem Fall ist es ratsam, auf eine komplette Liste der lokalen Variablen umzuschalten. Sie können dieses mit dem entsprechenden Reiter unterhalb der Anzeigefensters tun.

Sie erhalten daraufhin eine Ausgabe ähnlich der folgenden:

Abb. 8.28
Die lokalen Variablen



Betrachten Sie nun die Inhalte der einzelnen Variablen. Wie nicht anders zu erwarten, sind sie allesamt nicht initialisiert und tragen vom Debugger vorgegebene Werte. Beispielsweise findet sich für *pNewObject* der bekannte Wert *0xcccccccc* wieder.

Steuerung nach Erreichen eines Breakpoints

Sobald ein Abbruchpunkt erreicht wurde, liegt es in ihrer Hand, wie mit der Programmausführung fortzufahren ist.

Die wichtigsten Kommandos sind im Folgenden aufgeführt:

Fortsetzen des Programmablaufs

- F5: Setzt den Programmablauf fort. Dieses entspricht dem Starten des Programms, nur dass dieses Mal die Ausführung an der Stelle weitergeführt wird, an der sie soeben abgebrochen wurden. Das Programm wird weiterlaufen, bis es terminiert oder der nächste Breakpoint erreicht wurde.

Einzelschrittausführung

- F10: Führt die nächste Anweisung komplett aus und bricht die Programmausführung danach ab. Hat den gleichen Effekt, als wäre in der nächsten Zeile wieder ein Breakpoint gesetzt und die Programmabarbeitung würde mit F5 fortgesetzt werden. Mithilfe diesen Kommandos ist es möglich, Schritt für Schritt durch ein Programm hindurch zu gehen (man bezeichnet diesen Vorgang häufig mit steppen).

Einzelschrittausführung mit Betreten von aufgerufenen Funktionen

- F11: Führt die nächste Anweisung aus und springt gegebenenfalls in hinein. Diese Ausführungsart ist vor allem dann interessant, wenn die nächste auszuführende Zeile einen Funktionsaufruf enthält. In diesem Fall führt der Debugger nur den eigentlichen Aufruf aus und stoppt dann auf der ersten gültigen Zeile innerhalb der aufgerufenen Funktion.

Abbrechen eines Programms

- ⇧+F5: Bricht die Programmabarbeitung ab.

Spielen Sie ein wenig mit diesen Kommandos herum. Beobachten Sie, wie sich der Callstack verändert und wie die Variablenwerte in den entsprechenden Anzeigefenster aktualisiert werden.

Bedingte Abbruchpunkte

Bedingungen für Breakpoints definieren

Bislang sind sämtliche Breakpoints unbedingte Abbruchpunkte. Jeder Breakpoint, den Sie neu in einem Programm setzen, zählt zu dieser Kategorie.

Sie können sich übrigens im Breakpoint-Fenster eine Übersicht aller derzeit im Programm aktiven Abbruchpunkte anzeigen lassen. Ist dieses Fenster derzeit nicht aktiv, können Sie es mit der Tastenfolge Strg+Alt+B in den Vordergrund holen:

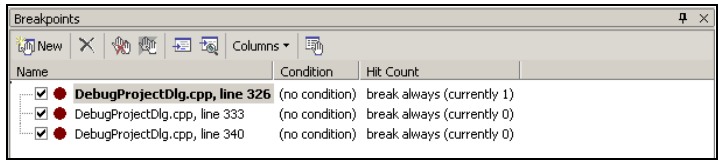
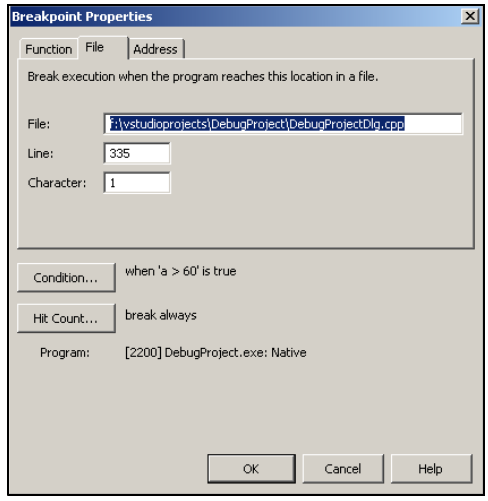


Abb. 8.29 **Übersicht aller Breakpoints**

Der Kasten links neben den Abbruchpunkten (die mit Datei und Zeilennummer tituliert sind) gibt an, ob der Breakpoint gerade aktiv ist oder ignoriert werden soll. Die weiteren Angaben beziehen sich darauf, ob ein bedingter oder unbedingter Abbruchpunkt vorliegt.

Die Eigenschaften eines Abbruchpunkts können Sie verändern, indem sie einen Breakpoint anklicken und aus seinem Kontextmenü den Punkt *Eigenschaften* auswählen:

Abb. 8.30
Eigenschaften eines Breakpoints



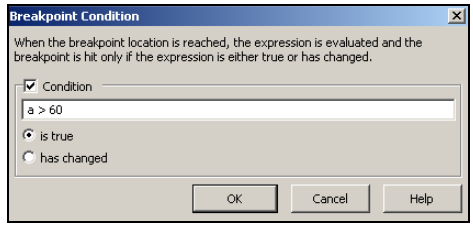
Die Eigenschaften beschreiben die Position des Breakpoints, beispielsweise die Datei, in der er vorkommt und die Zeilennummer, auf die er platziert wurde.

Weiterhin können Sie hier Bedingungen angeben, die beschreiben, wann ein Abbruchpunkt greifen soll und wann der Debugger ihn zu ignorieren hat.

Beispielhaftes Setzen einer Breakpoint-Bedingung

Wählen Sie aus dem Quelltext den zweiten gesetzten Breakpoint aus – den im *for*-Konstrukt – und öffnen durch Anklicken der Bedingungsschaltfläche einen weiteren Konfigurationsdialog:

Abb. 8.31
Bedingungen für einen Breakpoint



In diesem können Sie spezifizieren, ob eine Bedingung für den Breakpoint festgelegt werden soll, und wie diese auszusehen hat. Im Wesentlichen haben Sie für die Beschreibung einer Bedingung sämtliche Möglichkeiten, die Ihnen die C++-Syntax auch bietet.

Schauen Sie für die Beschreibung einiger Ausnahmen und Ergänzungen in Ihre Online-Hilfe.

Wahre und geänderte Bedingungen

Möglichkeiten der Bedingungsüberprüfung

Neben der Bedingung selbst können Sie angeben, ob die Bedingung lediglich auf ihren Wahrheitsgehalt geprüft werden soll (wenn die Bedingung wahr ist, hält der Debugger die Programmausführung an) oder ob sich der Wahrheitsgehalt der Bedingung geändert hat.

In letzterem Fall speichert der Debugger das letzte Ergebnis der Abfrage und vergleicht es mit dem aktuellen. Hat sich hier eine Änderung ergeben, wird das Programm angehalten.

Beim ersten Erreichen des Breakpoints wird das Programm entsprechend nie angehalten, da keine Vergleichsmöglichkeiten zu vorhergehenden Durchläufen besteht.

Setzen Sie als Bedingung für den zweiten Breakpoint die Zeile

```
a > 60
```

ein und als Abbruchkriterium die Option `Ist Wahr`.

Führen Sie dann die Programmausführung fort – wenn Sie sich nicht mehr auf dem ersten Breakpoint befinden, starten Sie das Programm beziehungsweise die Behandlung der *OnBnClickedBreakpointtest*-Methode – neu.

Drücken Sie gegebenenfalls mehrfach auf F5, bis Sie am zweiten Breakpunkt stoppen. Untersuchen Sie nun die aktuell gültigen Variablenwerte:

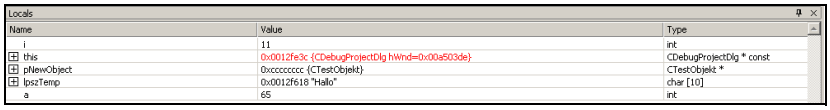


Abb. 8.32
Die Bedingung ist erfüllt

Wie zu erwarten wurde der Breakpunkt erst aktiviert, nachdem die Variable a einen Wert größer als 60 angenommen hatte. Drücken Sie erneut F5, hält die Programmausführung wieder an der gleichen Stelle an, a hat dann den Wert 70 und so weiter.

Anzahl der Breakpoint-Aufrufe als Abbruchkriterium

Eine andere Möglichkeit, den Programmablauf zu kontrollieren, ist der, Breakpoints nur dann aktiv werden zu lassen, sobald die betreffenden Abbruchpunkte eine gewisse Anzahl von Durchläufen bereits angesprungen wurden.

Den betreffenden Punkt erreichen Sie ebenfalls über das *Eigenschaften*-Dialogfeld für Breakpoints.

Für einen kleinen Test wählen Sie nun den dritten Abbruchpunkt aus, öffnen seinen *Eigenschaften*-Dialog und wählen den Punkt *Hit Count*.

Hier haben Sie die Möglichkeit festzulegen, wann der Breakpoint aktiv sein soll – als Auswahl bieten sich hier die folgenden Möglichkeiten:

Immer abbrechen

- *immer abbrechen*: der Debugger hält den Programmablauf jedes Mal an, wenn der Breakpoint erreicht wird.

Zahl der Aufrufe ist gleich x

- *ist gleich*: Sie können einen Wert angeben, der spezifiziert, nach wie vielen Anläufe der Breakpoint aktiv sein soll. Tragen Sie beispielsweise den Wert 3 ein, stoppt der Debugger das ablaufende Programm erst, sobald der Abbruchpunkt exakt zum dritten Mal erreicht wird. Erreicht das Programm den Abbruchpunkt zum vierten oder einem weiteren Mal, wird das Programm nicht mehr angehalten.

Zahl der Aufrufe ist ein Vielfaches von x

- *ist ein Vielfaches von*: Sie können einen Wert angeben, der spezifiziert, nach wie vielen Durchläufen der Breakpoint aktiv sein muss. Der aktuelle Zählstand wird dabei daraufhin geprüft, ob er ein Vielfaches des eingetragenen Werts ist. Haben Sie als Zahl 3 eingetragen, stoppt der Debugger das Programm beim dritten Erreichen des Breakpoints, beim sechsten Mal, beim neunten Mal und so weiter.

Zahl der Aufrufe ist gleich oder größer als x

- *ist gleich oder größer als*: Sie können einen Wert angeben, der spezifiziert, nach wie vielen Durchläufen der Breakpoint aktiv sein soll. Der aktuelle Zählstand wird beim Erreichen des Breakpoints daraufhin geprüft, ob er größer oder gleich dem eingetragenen Wert ist. Haben Sie hier zum Beispiel fünf eingestellt, wird der Debugger den Programmablauf ab dem fünften Erreichen des Breakpoints jedes Mal stoppen, sobald er wieder erreicht wird.

Tragen Sie für den dritten Breakpoint als Abbruchbedingung *ist gleich* ein, als Wert 3:

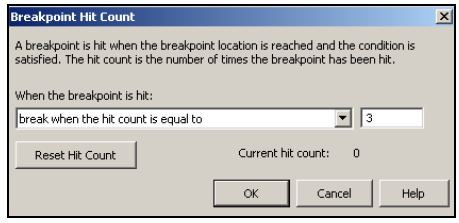


Abb. 8.33
Bedingter Breakpoint

Testen Sie das Verhalten aus. Prüfen Sie, ob der Breakpoint in den ersten Methodendurchläufen ignoriert wird, was im dritten und was in den darauf folgenden Durchläufen geschieht.

Austesten des Breakpoint-Verhaltens

Sie werden bemerken, dass der Breakpoint das erste Mal tatsächlich erst im dritten Durchlauf greift:

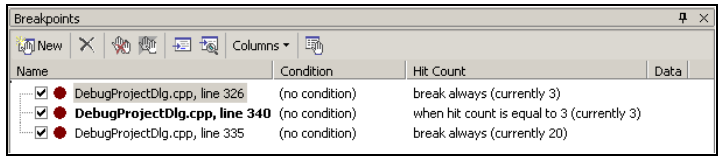


Abb. 8.34
Der Breakpoint wird im dritten Durchlauf das erste und einzige Mal aktiviert

Verändern von Variablenwerten

Während Sie gerade beim dritten Breakpoint angelegt sind, können Sie eine weitere Nettigkeit des Debuggers antesten: Das Verändern von Variablenwerten zur Laufzeit des Programms.

Die durch den Abbruchpunkt angegebene Zeile ist gerade die zum Anlegen eines neuen Objekts:

```
    // neues Objekt anlegen
    pNewObject = new CTestObjekt;

    if (NULL == pNewObject)
    {
        AfxMessageBox("Konnte Objekt nicht anlegen");
        return;
    }

    // Objekt abräumen
    delete pNewObject;
}
```

Abb. 8.35
Anlegen eines neuen Objekts

Drücken Sie einmal auf F10, um die aktuelle Zeile auszuführen und beobachten Sie, was mit der lokalen Variable *pNewObject* im Variablenanzeigefenster geschieht.

Sie werden sehen, dass ihr eine Speicheradresse zugewiesen wird, die ungleich NULL (also ungleich *oxoooooooo*) ist.

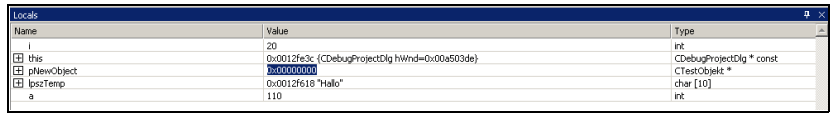
Daraus ergibt sich sofort, dass das nachfolgende *if*-Konstrukt nicht greifen sondern überspringen wird.

Doppelklicken Sie jetzt im Variablenanzeigefenster auf den Wert des *pNewObject*-Objekts. Es erscheint ein Eingabefeld, in das Sie einen neuen Wert eintragen können.

Verändern eines Werts zur Laufzeit

Stellen Sie hier den Wert *oxooooooooo* ein, wie es die folgende Abbildung zeigt:

Abb. 8.36
Verändern eines Werts zur Laufzeit



Das dieser Wert übernommen wird, können Sie sehen, wenn Sie nun erneut mit F10 die Programmausführung schrittweise fortsetzen: Die Bedingung im *if*-Konstrukt wird als wahr ausgewertet, die Fehlermeldung wird ausgegeben.

Es ist in vielen Situationen sinnvoll, Variablenwerte zur Laufzeit manuell zu verändern, beispielsweise um längere Schleifen vorzeitig abbrechen zu können, aber auch, um auszutesten, wie Funktionen auf bestimmte Werte reagieren.

Überprüfen des Speicherinhalts

Als letzter Punkt in diesem Abschnitt soll die Überprüfung von Speicherinhalten genannt werden.

Anzeige des Speicherinhalts

Der Speicher (insbesondere der Heap-Bereich des Speichers) ist zunächst nichts anderes als eine nur durch den physikalisch vorhandenen Speicher begrenzte Größe, bestehend aus zahllosen Bits und Bytes, die irgend einer Bedeutung haben – oder auch nicht. Es sei an dieser Stelle nicht darauf eingegangen, dass Windows den zur Verfügung stehenden Speicher durch die Verwendung einer Auslagerungsdatei enorm erweitern kann.

Die Bedeutung der einzelnen Speicherstellen hängt von den Programmen ab, die mit diesen Speicherbereichen arbeiten. So sind die einzelnen Segmente vielleicht mit sinnvollen Daten belegt, vielleicht aber auch Überbleibsel längst beendeter Applikationen.

Während zu früheren Zeiten noch mehr oder weniger beliebig im Speicher gearbeitet werden konnte, unterbinden moderne Betriebssysteme wie Windows 2000 oder Windows XP dieses Vorgehen und gestatten nur Zugriffe auf den zu einem Prozess gehörenden Speicherbereich.

In der Regel werden Speicherbereiche im Rahmen der Programmentwicklung nach Möglichkeit nur als abstrakte Objekte angesehen, die irgendwie zur Verfügung stehen und zur Arbeit erst angefordert werden müssen.

Tatsächlich ist es in den meisten Fällen nicht notwendig, einen Blick auf den physikalisch tatsächlich verwendeten Speicher zu werfen.

Hin und wieder ist es jedoch trotzdem ratsam, und einmal gesehen zu haben, wie Daten in den Speicher geschrieben werden, ist in jedem Fall für jeden Softwareentwickler sinnvoll.

Ein Programmbeispiel zur Speicheransicht

Es soll nun eine kleine Methode geschrieben werden, die sechzehn Zahlen in einen Speicherbereich schreibt. Fügen Sie die zugehörigen Zeilen in der Datei *DebugProjectDlg.cpp* in die Behandlungsmethode *OnBnClickedSpeichertest* ein:

Listing 8.17
Belegen von Speicherbereichen

```
void CDebugProjectDlg::OnBnClickedSpeichertest()
{
   // Speicher erbitten
   char *pChars = new char[16];

   // Speicher beschreiben
   for (int i=0;i<16;i++)
   {
      pChars[i] = i;
   }

   // Speicher freigeben
   delete [] pChars;
}
```

Diese Methode erzeugt ein Chararray mit sechzehn Einträgen und füllt diese nacheinander mit den Zahlen 0 bis 15. Danach wird der angeforderte Speicher wieder freigegeben.

Setzen Sie einen Breakpoint auf die erste Zeile der Methode und starten Sie das Programm dann über den Debugger. Wählen Sie den Punkt Speichertest aus dem Dialogfeld aus – sie finden sich jetzt im Quelltext wieder, gerade auf der Zeile, die den new Operator enthält.

Speicheransichtsfenster öffnen

Drücken Sie einmal auf die F10-Taste, Sie sehen, dass der Variablen jetzt eine Speicheradresse zugewiesen wurde.

Öffnen Sie eines der Speicheransichtsfenster (Strg+Alt+M gefolgt von einer der Tasten 1 bis 4 für Speicheransicht 1 bis 4) und tragen Sie als Basisadresse die Adresse des neu erzeugten Objekts ein. Sie erhalten eine Ansicht ähnlich der folgenden:

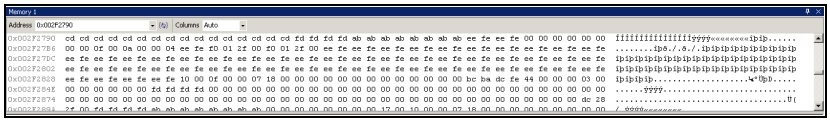
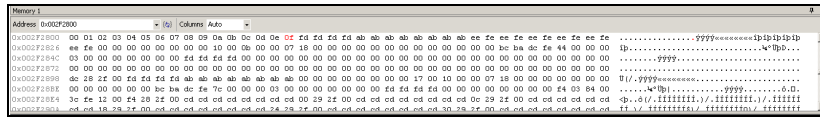


Abb. 8.37
Der neue Speicherbereich

Es ist sofort zu erkennen, dass der Speicherbereich aufgrund der vorliegenden Debug-Version praktischerweise mit *oxcd*-Werten initialisiert wurde.

Steppen Sie nun weiter mit der Taste F10 durch das Programm: Die Speicherstellen werden der Reihe nach beschrieben, wobei sich die gerade veränderten Zellen rot färben und so den neuen Wert deutlich anzeigen:

Abb. 8.38
Die veränderten Speicherstellen



Zusammenfassung

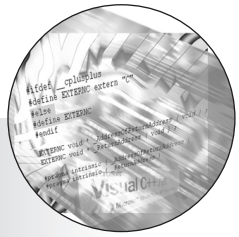
Sie sind jetzt am Ende dieses Kapitels angelangt und haben einiges über die Arbeit mit dem Debugger und den zur Verfügung stehenden Hilfsmitteln gelernt.

Es sei wärmstens empfohlen, die hier aufgezeigten Techniken extensiv einzusetzen: Jede Minute der Fehlersuche und insbesondere der Fehlerprävention ist eine gewonnene Minute!

Zum Abschluss

Abschließend hoffen wir, dass Sie Gefallen an dem vorliegenden Buch gefunden und einen für Ihre Bedürfnisse guten ersten Einblick in die Entwicklung unter Windows mit den MFC und dem neuen Visual Studio .NET erhalten haben.

Auf Ihrem weiteren Weg als Windows-Entwickler wünschen ich und der gesamte Sybex Verlag Ihnen viel Spass und allseits guten Erfolg!



Fensterstrukturen und zugehörige Funktionen

A

Windows-Messages

Die folgende Tabelle gibt einen Überblick über die verfügbaren Windows-Messages (Nachrichten), die Sie innerhalb Ihrer Applikationen verwenden können.

Sie sind alphabetisch sortiert, die Tabelle umfasst die Namen der Botschaften, sowie den Prototyp ihrer Behandlungsmethoden und eine kurze Beschreibung, wann die Nachricht versendet wird.

Die Bedeutung der einzelnen Parameter entnehmen Sie der MSVC++ beiliegenden Online-Hilfe.

Tabelle A.1
Die Windows-Nachrichten im Überblick

Nachricht	Behandlungsmethode
WM_ACTIVATE	afx_msg void OnActivate (UINT, CWnd*, BOOL);
Eine *WM_ACTIVATE*-Nachricht wird gesendet, sobald ein neues Fenster den Fokus erhält (zum Beispiel durch einen Mausklick des Benutzers). Die Botschaft geht sowohl an das deaktivierte, wie auch an das aktivierte Fenster.	
WM_ACTIVATEAPP	afx_msg void OnActivateApp (BOOL, HANDLE);
Ähnlich wie die *WM_ACTIVATE*-Nachricht wird *WM_ACTIVATEAPP* versendet, sobald ein neues Fenster den Fokus erhält. Der Unterschied besteht darin, dass *WM_ACTIVATEAPP* nur dann verschickt wird, wenn zwei unterschiedliche Applikationen durch diesen Fensterwechsel betroffen sind. Wechselt der Benutzer beispielsweise zwischen zwei Fenstern innerhalb einer MDI-Applikation, kommt diese Botschaft nicht zum Tragen.	
WM_ASKCBFORMATNAME	afx_msg void OnAskCbFormatName (UINT, LPSTR);
Diese Nachricht wird von einem die Zwischenablage benutzenden Fenster an den ursprünglichen Beschreiber der Zwischenablage gesendet, um den Namen des *CF_OWNERDISPLAYS* zu erhalten.	
WM_CANCELMODE	afx_msg void OnCancelMode ();
WM_CANCELMODE wird verschickt, um das Abbrechen einiger Zustandsmodi zu veranlassen, beispielsweise eines *MouseCaptures*. Ein weiteres Beispiel ist das Senden dieser Nachricht an ein aktives Fenster, sobald eine Messagebox geöffnet wird.	
WM_CAPTURECHANGED	afx_msg void OnCaptureChanged (CWnd*);
Diese Nachricht wird an ein Fenster gesendet, das gerade im Begriff ist, den *MouseCapture* zu verlieren.	

Nachricht	Behandlungsmethode
WM_CHANGECBCHAIN	afx_msg void OnChangeCbChain (HWND, HWND);
Zwischenablagefenster (also solche, die den Inhalt einer Zwischenablage anzeigen) sind untereinander verknüpft. Wird eines der Fenster geschossen, erhält das erste Fenster in dieser Liste die Botschaft *WM_CHANGECBCHAIN*. Es wird empfohlen, diese Nachricht per *SendMessage* an das nächste Fenster in der Liste weiterzuleiten.	
WM_CHAR	afx_msg void OnChar (UINT, UINT, UINT);
Die *WM_CHAR*-Nachricht wird versendet, sobald ein Zeichen durch *TranslateMessage* übersetzt wurde. Der Code dieses Zeichens wird als Nachrichtenparameter mit übermittelt.	
WM_CHARTOITEM	afx_msg int OnCharToItem (UINT, CWnd*, UINT);
Diese Nachricht wird von einer Listbox, die das *LBS_WANTKEYBOARDINPUT*-Flag besitzt an ihr Elternfenster in Antwort auf eine *WM_CHAR*-Nachricht gesendet.	
WM_CHILDACTIVATE	afx_msg void OnChildActivate ();
Diese Botschaft wird an Kindfenster einer MDI-Applikation versandt, sobald diese aktiviert, bewegt oder in der Größe verändert werden.	
WM_CLOSE	afx_msg void OnClose ();
WM_CLOSE signalisiert das Schließen eines Fensters oder einer Applikation.	
WM_COMPACTING	afx_msg void OnCompacting (UINT);
Diese Mitteilung wird an alle Hauptrahmenfenster versandt, wenn das Windows-System feststellt, dass der Systemspeicher knapp wird.	
WM_COMPAREITEM	afx_msg int OnCompareItem (LPCOMPAREITEMSTRUCT);
WM_COMPAREITEM ermittelt die relative Position eines neuen Elements in einer Combo- oder Listbox. Bei jedem Hinzufügen eines neuen Elements wird die Mitteilung an das Elternfenster dieser Steuerelemente gesandt, wenn diese mit dem *CBS_SORT* oder *LBS_SORT*-Flag erzeugt wurden.	
WM_CONTEXTMENU	afx_msg void OnContextMenu (CWnd*, CPoint);
Wird versandt, wenn der Benutzer innerhalb eines Fensters die rechte Maustaste gedrückt hat.	

Nachricht	Behandlungsmethode
WM_COPYDATA	afx_msg BOOL OnCopyData (CWnd* pWnd, COPYDATASTRUCT* pCopyDataStruct);
WM_COPYDATA signalisiert, dass eine Applikation Daten an eine andere transferiert. Um diese Nachricht selbst zu versenden, muss die *SendMessage*-Funktion verwendet werden, nicht etwa *PostMessage*.	
WM_CREATE	afx_msg int OnCreate (LPCREATESTRUCT);
WM_CREATE wird versendet, sobald eine Applikation ein neues Fenster mittels *CreateWindowEx* oder *CreateWindow* erzeugen will.	
WM_CTLCOLOR	afx_msg HBRUSH OnCtlColor (CDC*, CWnd*, UINT);
Wird vom Anwendungsgerüst versendet, wenn ein Steuerelement gezeichnet werden soll. *WM_CTLCOLOR* wird für gewöhnlich verwendet, um dem Gerätekontext die richtigen Farben für die entsprechende Kontrolle mitzuteilen.	
WM_DEADCHAR	afx_msg void OnDeadChar (UINT, UINT, UINT);
Diese Nachricht wird versandt, sobald ein so genanntes totes Zeichen gedrückt und durch *TranslateMessage* übersetzt wurde. Ein Beispiel hierfür ist das ^, das erst durch Drücken einer weiteren Taste zu einem Zeichen zusammengesetzt wird. Die Kombination ^ und e ergibt beispielsweise ê.	
WM_DELETEITEM	afx_msg void OnDeleteItem (LPDELETEITEMSTRUCT);
WM_DELETEITEM signalisiert das Löschen eines Elements aus einer List- oder Combobox, bzw. das Zerstören der Box selbst.	
WM_DESTROY	afx_msg void OnDestroy ();
Teilt einem Fenster mit, dass es zerstört wird. In diesem Augenblick ist es bereits vom Desktop verschwunden.	
WM_DESTROYCLIPBOARD	afx_msg void OnDestroyClipboard ();
Teilt dem Besitzer der Zwischenablage mit, dass diese durch einen Aufruf von *EmptyClipboard* gelöscht wurde.	
WM_DEVICECHANGE	afx_msg void OnDeviceChange (UINT, DWORD);

Nachricht	Behandlungsmethode
Benachrichtigt eine Applikation oder einen Gerätetreiber darüber, dass eine Veränderung an den Hardwareeinstellungen dieses Geräts oder des Computers stattgefunden hat.	
WM_DEVMODECHANGE	afx_msg void OnDevModeChange (LPSTR);
Diese Nachricht wird an alle Hauptrahmenfenster versendet, sobald Änderungen an den Einstellungen eines Geräts getätigt werden.	
WM_DRAWCLIPBOARD	afx_msg void OnDrawClipboard ();
Diese Nachricht wird an das erste Fenster in der Liste der Zwischenablagenanzeigenden Fenster gesendet, um es darüber zu informieren, dass eine Veränderung derselben stattgefunden hat. Diese Nachricht sollte per *SendMessage* an das nächste Fenster in der Liste weitergeleitet werden.	
WM_DRAWITEM	afx_msg void OnDrawItem (LPDRAWITEMSTRUCT);
Die Mitteilung wird an ein Fenster geschickt, wenn eins seiner untergeordneten Steuerelemente (insbesondere Buttons, Comboboxes, Listboxes oder Menüs) verändert wurde.	
WM_DROPFILES	afx_msg void OnDropFiles (HDROP);
Wird an ein Fenster verschickt, wenn der Benutzer ein File per Drag & Drop auf dieses Fenster zieht. Es muss als Fenster eingetragen sein, dass solche Aktionen unterstützt.	
WM_ENABLE	afx_msg void OnEnable (BOOL);
Aktiviert oder deaktiviert eine Applikation ein ihr zugehöriges Fenster, wird diese Nachricht verschickt. Der übergebene boolesche Parameter zeigt an, ob das entsprechende Fenster deaktiviert oder aktiviert werden soll.	
WM_ENDSESSION	afx_msg void OnEndSession (BOOL);
Die Nachricht wird an eine Applikation verschickt, nachdem das System das Ergebnis einer *WM_QUERYENDSESSION*-Nachricht ermittelt hat. *WM_ENDSESSION* informiert die Applikation darüber, ob die Sitzung beendet wird.	
WM_ENTERIDLE	afx_msg void OnEnterIdle (UINT, CWnd*);
Diese Nachricht wird an ein Elternfenster eines modalen Dialogs oder eines Menüs gesendet, sobald dieses seine gesamte Nachrichten abgearbeitet hat.	

Nachricht	Behandlungsmethode
WM_ERASEBKGND	afx_msg BOOL OnEraseBkgnd (CDC *);
WM_ERASEBKGND signalisiert, dass der Hintergrund eines Fensters gelöscht werden muss, um das Neuzeichnen seines Inhalts zu ermöglichen.	
WM_FONTCHANGE	afx_msg void OnFontChange ();
WM_FONTCHANGE wird an alle Hauptrahmenfenster verschickt, und signalisiert, dass eine Veränderung in den auf dem System installierten Zeichensätze stattgefunden hat.	
WM_GETDLGCODE	afx_msg UINT OnGetDlgCode ();
WM_GETDLGCODE wird an Fenster versendet, die ein Steuerelement enthalten. Auf diese Weise kann die Standardeinstellung umgangen werden, nach der Windows alle Nachrichten für ein Steuerelement verwaltet und relevante Eingaben des Benutzers (zum Beispiel die ⇆-Taste zum Navigieren zwischen einzelnen Elementen, oder Hotkeys zur Ansteuerung einzelner Dialogfeldelemente) selbstständig interpretiert. Durch das Reagieren auf die *WM_GETDLGCODE*-Nachrichten, kann jedes Steuerelement individuell auf die Eingaben des Anwenders reagieren.	
WM_GETMINMAXINFO	afx_msg void OnGetMinMaxInfo (LPPOINT);
Diese Nachricht wird an ein Fenster verschickt, dessen Größe oder Position geändert wird. Die übergebene Struktur enthält Informationen über erlaubte maximale und minimale Größe bzw. die Position des Fensters. Durch Anpassen der Strukturkomponenten können eigene Werte für die folgende Vergrößerungs- bzw. Repositioierungsaktion gültig erklärt werden.	
WM_HELP	nicht verfügbar
Wird verschickt, wenn ein Benutzer in einem mit dem *WS_EX_CONTEXTHELP*-Flag versehenen Fenster den Fragezeichen-Mauszeiger benutzt, um Informationen über ein Fensterelement zu erhalten.	
WM_HELPINFO	afx_msg BOOL OnHelpInfo (HELPINFO*);
Wird beim Drücken der Taste F1 verschickt.	
WM_HSCROLL	afx_msg void OnHScroll (UINT, UINT, CWnd*);
Die *WM_HSCROL*L-Botschaft wird beim Benutzen einer horizontalen Bildlaufleiste verschickt.	

Nachricht	Behandlungsmethode
WM_HSCROLLCLIPBOARD	afx_msg void OnHScrollClipboard CWnd*, UINT, UINT);
Wird an den Besitzer der Zwischenablage geschickt, sobald in einem der diese anzeigenden Fenster eine horizontale Bildlaufleiste benutzt wird.	
WM_ICONERASEBKGND	afx_msg void OnIconEraseBkgnd (CDC *);
Wird an ein minimierte Fenster geschickt, wenn der Hintergrund eines Icons gefüllt werden muss, bevor dass eigentliche Icon gezeichnet werden darf.	
WM_INITMENU	afx_msg void OnInitMenu (CMenu *);
WM_INITMENU wird verschickt, bevor ein Menü aktiviert wird (also wenn der Benutzer gerade auf einen Eintrag in der Menüleiste geklickt hat). Das Abfangen dieser Nachricht wird benutzt, um das Erscheinungsbild eines Menüs zu verändern, bevor es dargestellt wird.	
WM_INITMENUPOPUP	afx_msg void OnInitMenuPopup (CMenu *, UINT, BOOL);
Diese Nachricht entspricht *WM_INITMENU*, wird jedoch nur dann verschickt, wenn ein Popup-Menü geöffnet werden soll.	
WM_KEYDOWN	afx_msg void OnKeyDown (UINT, UINT, UINT);
Die Botschaft wird immer dann versendet, wenn eine Taste gedrückt wird, die keine Systemtaste ist.	
WM_KEYUP	afx_msg void OnKeyUp (UINT, UINT, UINT);
Wird im Gegensatz zu *WM_KEYDOWN* verschickt, wenn eine Taste losgelassen wird, die keine Systemtaste ist.	
WM_KILLFOCUS	afx_msg void OnKillFocus (CWnd*);
Diese Nachricht wird an ein Fenster geschickt, bevor es den Eingabefokus verliert.	
WM_LBUTTONDBLCLK	afx_msg void OnLButtonDblClk (UINT, CPoint);
WM_LBUTTONDBCLK wird verschickt, wenn der Benutzer einen Doppelklick ausgeführt hat.	
WM_LBUTTONDOWN	afx_msg void OnLButtonDown (UINT, CPoint);

Nachricht	Behandlungsmethode
WM_LBUTTONDOWN signalisiert das Drücken der linken Maustaste.	
WM_LBUTTONUP	afx_msg void OnLButtonUp (UINT, CPoint);
WM_LBUTTONUP signalisiert das Loslassen der linken Maustaste.	
WM_MBUTTONDBLCLK	afx_msg void OnMButtonDblClk (UINT, CPoint);
WM_MBUTTONDBLCK signalisiert analog zu *WM_LBUTTONDBLCK* einen Doppelklick der mittleren Maustaste.	
WM_MBUTTONDOWN	afx_msg void OnMButtonDown (UINT, CPoint);
WM_MBUTTONDOWN signalisiert analog zu *WM_LBUTTONDOWN* das Drücken der linken Maustaste.	
WM_MBUTTONUP	afx_msg void OnMButtonUp (UINT, CPoint);
WM_MBUTTONUP signalisiert analog zu *WM_LBUTTONUP* das Loslassen der linken Maustaste.	
WM_MDIACTIVATE	afx_msg void OnMDIActivate (BOOL, CWnd*, CWnd*);
WM_MDIACTIVATE wird durch eine Applikation an ein MDI-Hauptrahmenfenster verschickt, um dieses aufzufordern, ein bestimmtes Kindfenster zu aktivieren. Dabei wird die Botschaft sowohl an das zu aktivierende als auch an das aus diesem Grund zu deaktivierende Fenster weitergeleitet.	
WM_MEASUREITEM	afx_msg void OnMeasureItem (LPMEASUREITEMSTRUCT);
Diese Nachricht wird an ein Fenster verschickt, in dem ein Button, eine Combo- oder Listbox oder ein Menüeintrag erzeugt werden soll.	
WM_MENUCHAR	afx_msg LONG OnMenuChar (UINT, UINT, CMenu *);
Drückt der Benutzer in einem aktivierten Menü eine nicht vorhergesehene Taste (insbesondere eine, die keinen Shortcut auf einen Menüpunkt darstellt), wird das eingetippte Zeichen an das Elternfenster des Menüs weitergeleitet, was durch die *WM_MENUCHAR*-Botschaft geschieht.	
WM_MENUSELECT	afx_msg void OnMenuSelect (UINT, UINT, HMENU);

Nachricht	Behandlungsmethode
Die Nachricht wird an das Elternfenster eines Menüs geschickt, sobald der Benutzer einen Menüeintrag ausgewählt hat.	
WM_MOUSEACTIVATE	afx_msg int OnMouseActivate (CWnd*, UINT, UINT);
WM_MOUSEACTIVATE wird an ein inaktives Fenster versandt, wenn eine Maustaste gedrückt wird, während sich der Mauszeiger über diesem Fenster befindet.	
WM_MOUSEMOVE	afx_msg void OnMouseMove (UINT, CPoint);
Das Bewegen der Maus sorgt für ein Erzeugen dieser Nachricht, anhand dessen die Applikation die aktuelle Mauszeigerposition ermitteln kann.	
WM_MOUSEWHEEL	afx_msg BOOL OnMouseWheel (UINT, short, CPoint);
Das Bewegen des Mausrads sorgt für das Verschicken einer *WM_MOUSEWHEEL*-Botschaft an das den Eingabefokus innehabenden Fensters.	
WM_MOVE	afx_msg void OnMove (int, int);
WM_MOVE signalisiert, dass ein Fenster verschoben wurde.	
WM_MOVING	afx_msg void OnMoving (UINT, LPRECT);
WM_MOVING wird an ein Fenster geschickt, das gerade bewegt wird.	
WM_NCACTIVATE	afx_msg BOOL OnNcActivate (BOOL);
WM_NCACTIVATE (das NC steht wie bei allen Botschaften dieser Kategorie für Non-Client, bezeichnet als den Bereich eines Fensters ohne dessen Client Area) wird verschickt, wenn die Non-Client Area eines Fensters verändert werden muss.	
WM_NCCALCSIZE	afx_msg void OnNcCalcSize (BOOL, NCCALCSIZE_PARAMS FAR*);
Diese Nachricht wird generiert, wenn die Größe und Position der Client Area eines Fenster neu berechnet werden muss.	
WM_NCCREATE	afx_msg BOOL OnNcCreate(LPCREATESTRUCT);
Diese Nachricht wird unmittelbar vor WM_CREATE verschickt, wenn ein neues Fenster erzeugt wird.	

Nachricht	Behandlungsmethode
WM_NCDESTROY	afx_msg void OnNcDestroy ();
Diese Nachricht signalisiert, dass eine Non-Client Area zerstört werden soll. Sie wird in der Regel nach der *WM_DESTROY*-Botschaft verschickt.	
WM_NCHITTEST	afx_msg UINT OnNcHitTest (CPoint);
WM_NCHITTEST wird verschickt, wenn eine Mausbewegung stattgefunden hat oder eine der Maustasten gedrückt wurde. Hiermit kann geprüft wird, welcher Non-Client Bereich eines Fensters angewählt wurde (beispielsweise die Ränder oder eins der Größenveränderungssymbole).	
WM_NCLBUTTONDBLCLK	afx_msg void OnNcLButtonDblClk (UINT, CPoint);
Wird analog zu *WM_LBUTTONDBLCLK* beim Doppelklick der linken Maustaste versandt, dieses Mal jedoch nur, wenn sich der Mauszeiger über einer Non-Client Area befindet.	
WM_NCLBUTTONDOWN	afx_msg void OnNcLButtonDblClk (UINT, CPoint);
Wird analog zu *WM_LBUTTONDOWN* beim Drücken der linken Maustaste versandt, dieses Mal jedoch nur, wenn sich der Mauszeiger über einer Non-Client Area befindet.	
WM_NCLBUTTONUP	afx_msg void OnNcLButtonUp (UINT, CPoint);
Wird analog zu *WM_LBUTTONUP* beim Loslassen der linken Maustaste versandt, dieses Mal jedoch nur, wenn sich der Mauszeiger über einer Non-Client Area befindet.	
WM_NCMBUTTONDBLCLK	afx_msg void OnNcMButtonDblClk (UINT, CPoint);
Wird analog zu *WM_MBUTTONDBLCLK* beim Doppelklick der mittleren Maustaste versandt, jedoch nur, wenn sich der Mauszeiger über einer Non-Client Area befindet.	
WM_NCMBUTTONDOWN	afx_msg void OnNcMButtonDown (UINT, CPoint);
Wird analog zu *WM_MBUTTONDOWN* beim Drücken der mittleren Maustaste versandt, jedoch nur, wenn sich der Mauszeiger über einer Non-Client Area befindet.	
WM_NCMBUTTONUP	afx_msg void OnNcMButtonUp (UINT, CPoint);

Nachricht	Behandlungsmethode
Wird analog zu *WM_MBUTTONUP* beim Loslassen der mittleren Maustaste versandt, dieses Mal jedoch nur, wenn sich der Mauszeiger über einer Non-Client Area befindet.	
WM_NCMOUSEMOVE	afx_msg void OnNcMouseMove (UINT, CPoint);
Wird analog zur *WM_MOUSEMOVE* beim Bewegen der Maus verschickt, dieses Mal jedoch nur, wenn sich der Mauszeiger über einen Non-Client Area befindet.	
WM_NCPAINT	afx_msg void OnNcPaint ();
WM_NCPAINT wird verschickt, wenn die Non-Client Area eines Fenster neu gezeichnet werden muss.	
WM_NCRBUTTONDBLCLK	afx_msg void OnNcRButtonDblClk (UINT, CPoint);
Wird analog zu *WM_RBUTTONDBLCLK* beim Doppelklick der rechten Maustaste versandt, allerdings nur, wenn sich der Mauszeiger über einer Non-Client Area befindet.	
WM_NCRBUTTONDOWN	afx_msg void OnNcRButtonDown (UINT, CPoint);
Wird analog zu *WM_RBUTTONDBLCLK* beim Drücken der rechten Maustaste versandt, dieses Mal jedoch nur, wenn sich der Mauszeiger über einer Non-Client Area befindet.	
WM_NCRBUTTONUP	afx_msg void OnNcRButtonUp (UINT, CPoint);
Wird analog zu *WM_RBUTTONDBLCLK* beim Loslassen der rechten Maustaste versandt, dieses Mal jedoch nur, wenn sich der Mauszeiger über einer Non-Client Area befindet.	
WM_PAINT	afx_msg void OnPaint ();
WM_PAINT signalisiert, dass die Client Area eines Fenster neu gezeichnet werden muss.	
WM_PAINTCLIPBOARD	afx_msg void OnPaintClipboard (CWnd*, HANDLE);
Wird an ein die Zwischenablage anzeigendes Fenster verschickt, wenn aufgrund einer Änderung in der Zwischenablage ein Neuzeichnen desselben erforderlich geworden ist.	

Nachricht	Behandlungsmethode
WM_PALETTECHANGED	afx_msg void OnPaletteChanged (CWnd*);
WM_PALETTECHANGED wird an alle Hauptrahmenfenster und Overlapped-Fenster verschickt, nachdem das Fenster mit dem Eingabefokus seine logische Palette eingestellt hat, was ein Verändern der Systempalette zur Folge hat.	
WM_PALETTEISCHANGING	afx_msg void OnPaletteIsChanging (CWnd*);
WM_PALETTEISCHANGING wird verschickt, während die Palette verändert wird.	
WM_PARENTNOTIFY	afx_msg void OnParentNotify (UINT, LONG);
WM_PARENTNOTIFY wird an ein Elternfenster geschickt, wenn eine Maustaste gedrückt wird, während sich der Zeiger über einem Kindfenster befindet, oder sobald ein Kindfenster erzeugt oder zerstört wird.	
WM_QUERYDRAGICON	afx_msg HCURSOR OnQueryDragIcon ();
Wenn ein minimiertes Fenster, das einer Fensterklasse ohne Klassenicon entspringt, vom Benutzer bewegt wird, sendet das System diese Nachricht an die zugehörige Applikation, um ein individuelles Icon für den Drag-&-Drop-Vorgang bereitzustellen.	
WM_QUERYENDSESSION	afx_msg BOOL OnQueryEndSession);
WM_QUERYENDSESSION wird verschickt, sobald ein Anwender eine Sitzung beenden möchte. Gibt eine der angesprochenen Applikationen eine o zurück, wird der Vorgang unterbrochen.	
WM_QUERYNEWPALETTE	afx_msg BOOL OnQueryNewPalette ();
Diese Nachricht teilt einem Fenster mit, dass es gleich den Eingabefokus erhalten wird. Es kann daraufhin seine Palette vorbereiten.	
WM_QUERYOPEN	afx_msg BOOL OnQueryOpen ();
WM_QUERYICON wird an ein minimiertes Fenster geschickt, wenn es wieder in seine ursprüngliche (nicht-minimierte) Form gebracht werden soll, beispielsweise durch das Anklicken durch den Benutzer.	
WM_RBUTTONDBLCLK	afx_msg void OnRButtonDblClk (UINT, CPoint);

Nachricht	Behandlungsmethode
Analog zu *WM_LBUTTONDBLCK* wird diese Nachricht verschickt, sobald ein Doppelklick der rechten Maustaste ausgeführt wurde.	
WM_RBUTTONDOWN	afx_msg void OnRButtonDown (UINT, CPoint);
Analog zu *WM_LBUTTONDOWN* wird diese Nachricht verschickt, sobald die rechte Maustaste gedrückt wurde.	
WM_RBUTTONUP	afx_msg void OnRButtonUp (UINT, CPoint);
Analog zu *WM_LBUTTONUP* wird diese Nachricht verschickt, sobald die rechte Maustaste losgelassen wurde.	
WM_RENDERALLFORMATS	afx_msg void OnRenderAllFormats ();
Diese Nachricht wird an den Besitzer der Zwischenablage geschickt, bevor er zerstört wird. Damit der Inhalt der Zwischenablage für andere Applikationen erreichbar bleibt, muss der Besitzer vor seiner Beendigung die relevanten Daten in allen ihm möglichen Formaten in die Zwischenablage kopieren.	
WM_RENDERFORMAT	afx_msg void OnRenderFormat (UINT);
Ruft ein die Zwischenablagende benutzendes Programm eine Information aus dieser ab, muss der Besitzer der Zwischenablage die angeforderten Informationen im korrekten Format in die Zwischenablage kopieren. Dieser Vorgang wird durch die *WM_RENDERFORMAT*-Nachricht eingeleitet.	
WM_SETCURSOR	afx_msg BOOL OnSetCursor (CWnd*, UINT, UINT);
Sorgt dafür, dass ein bestimmter Mauszeiger für ein Fenster dargestellt wird, wenn sich der Cursor darüber befindet.	
WM_SETFOCUS	afx_msg void OnSetFocus (CWnd*);
WM_SETFOCUS wird an ein Fenster geschickt, sobald dieses den Eingabefokus erlangt hat.	
WM_SHOWWINDOW	afx_msg void OnShowWindow (BOOL, UINT);
Diese Nachricht wird an ein Fenster verschickt, das angezeigt oder verdeckt werden soll.	
WM_SIZE	afx_msg void OnSize (UINT, int, int);

Nachricht	Behandlungsmethode
WM_SIZE wird an ein Fenster geschickt, nachdem dessen Größe verändert wurde.	
WM_SIZECLIPBOARD	afx_msg void OnSizeClipboard (CWnd*, HANDLE);
WM_SIZECLIPBOARD wird an den Besitzer der Zwischenablage geschickt, wenn sich die Größe eines der die Zwischenablage anzeigenden Fensters verändert hat und die Daten in der Zwischenablage im *CF_OWNERDISPLAY*-Format abgelegt sind.	
WM_SIZING	afx_msg void OnSizing (UINT, LPRECT);
WM_SIZING wird versendet, während eine Größenveränderung eines Fensters durchgeführt wird.	
WM_SPOOLERSTATUS	afx_msg void OnSpoolerStatus (UINT, UINT);
WM_SPOOLERSTATUS wird vom Druckermanager verschickt, wenn ein Auftrag hinzugefügt oder abgearbeitet wurde.	
WM_STYLECHANGED	afx_msg void OnStyleChanged (int, LPSTYLESTRUCT);
Diese Nachricht wird an ein Fenster geschickt, nachdem ein oder mehrere seiner Stilflags verändert wurden. Dias geschieht normalerweise durch Aufruf der Funktion *SetWindowLong*.	
WM_STYLECHANGING	afx_msg void OnStyleChanging (int, LPSTYLESTRUCT);
Diese Nachricht wird verschickt, bevor die Funktion SetWindowLong Fensterstilflags verändert.	
WM_SYSCHAR	afx_msg void OnSysChar (UINT, UINT, UINT);
WM_SYSCHAR wird an ein Fenster verschickt, wenn eine *WM_SYSKEYDOWN*-Nachricht durch TranslateMessage bearbeitet wird. Dabei werden nur Systemtasten behandelt. (Eine Systemtaste ist eine Kombination aus der Alt-Taste und einer beliebigen weiteren Taste).	
WM_SYSCOLORCHANGE	afx_msg void OnSysColorChange ();
Diese Nachricht wird an alle Hauptrahmenfenster verschickt, wenn eine Veränderung an den Systemfarbeinstellungen vorgenommen wurde.	
WM_SYSCOMMAND	afx_msg void OnSysCommand (UINT, LONG);

Nachricht	Behandlungsmethode
WM_SYSCOMMAND kennzeichnet das Ausführen eines Befehls aus dem Systemmenü.	
WM_SYSDEADCHAR	afx_msg void OnSysDeadChar (UINT, UINT, UINT);
Im Gegensatz zu *WM_SYSCHAR* wird diese Nachricht verschickt, sobald eine tote Taste wie ^ zusammen mit der Alt-Taste gedrückt wird.	
WM_SYSKEYDOWN	afx_msg void OnSysKeyDown (UINT, UINT, UINT);
Die Nachricht wird durch das Drücken der Taste F10 oder einer Kombination aus Alt und einer weiteren Taste erzeugt.	
WM_SYSKEYUP	afx_msg void OnSysKeyUp (UINT, UINT, UINT);
WM_SYSKEYUP wird beim Loslassen einer der unter *WM_SYSKEYDOWN* beschriebenen Tasten generiert.	
WM_TIMECHANGE	afx_msg void OnTimeChange ();
Diese Nachricht signalisiert das Verändern der Systemzeit.	
WM_TIMER	afx_msg void OnTimer (UINT);
Die *WM_TIMER*-Botschaft wird immer dann ausgelöst, wenn ein durch *SetTimer* gesetzter Timer feuert.	
WM_VKEYTOITEM	afx_msg int OnVKeyToItem (UINT, CWnd*, UINT);
Diese Nachricht wird von einer Listbox mit dem *LBS_WANTKEYBOARDINPUT*-Flag als Reaktion auf eine *WM_KEYDOWN*-Nachricht an ihr Elternfenster geschickt.	
WM_VSCROLL	afx_msg void OnVScroll (UINT, UINT, CWnd*);
Entspricht *WM_HSCROL*L, bezieht sich jedoch auf die Benutzung einer vertikalen Bildlaufleiste.	
WM_VSCROLLCLIPBOARD	afx_msg void OnVScrollClipboard (CWnd*, UINT, UINT);
Entspricht *WM_HSCROLLCLIPBOARD*, bezieht sich jedoch auf die Benutzung einer vertikalen Bildlaufleiste.	

Nachricht	Behandlungsmethode
WM_WINDOWPOSCHANGED	afx_msg void OnWindowPosChanged (WINDOWPOS* lpwndpos);
Diese Nachricht wird an ein Fenster geschickt, dessen Größe, Position, oder Anordnung auf dem Schirm (vor und hinter anderen Fenstern) sich geändert hat.	
WM_WINDOWPOSCHANGING	afx_msg void OnWindowPosChanging (WINDOWPOS* lpwndpos);
Diese Nachricht wird an ein Fenster geschickt, dessen Größe, Position oder Anordnung auf dem Schirm gleich verändert wird.	

Die WNDCLASSEX-Struktur

Um ein Fenster unter der reinen Win32-API-Programmierung zu öffnen, muss zunächst eine dazu passende Fensterklasse registriert werden. Die hierzu auszufüllende WNDCLASSEX Struktur besitzt die folgenden Komponenten:

Komponente	Bedeutung
cbSize	Die Größe der *WNDCLASSEX*-Struktur. Wird normalerweise direkt per *sizeof(...)* gesetzt.
style	Legt den Stil der Fensterklasse durch Stilflags fest. Siehe auch folgende Tabelle Class-Styles.
lpfnWndProc	Bestimmt, welche Funktion die Verarbeitung von eingehenden Botschaften übernehmen soll. Sie wird für gewöhnlich als Nachrichtenfunktion bezeichnet.
cbClsExtra	Durch Angabe eines Werts größer als 0 kann man hier zusätzliche, frei verwendbare Bytes allokieren. Diese werden einmal pro Fensterklasse reserviert. Normalerweise wird *cbClsExtra* auf 0 gesetzt.
cbWndExtra	Durch Angabe eines Werts größer als 0 kann man hier zusätzliche, frei verwendbare Bytes allokieren, die dann für jedes der aus dieser Fensterklasse erzeugten Fensters zur Verfügung stehen. Wird wie *cbClsExtra* ebenfalls normalerweise auf 0 gesetzt.

Komponente	Bedeutung
hInstance	Um zu verhindern, dass mehrere Instanzen redundante Kopien derselben Fensterklasse anlegen, ordnet man hiermit der Klasse direkt das Handle der erzeugenden Instanz zu. Später aufgerufene Programme können so ein mehrfaches Registrieren einer Klasse vermeiden.
hIcon	Handle zum Symbol, das bei einem minimierten Fenster angezeigt wird, wenn *hIconSm* ungleich NULL ist. Ansonsten ist es nur das Symbol, dass beispielsweise im Explorer neben dem Namen der Anwendung steht. Icons befinden sich meistens in einer Ressource-Datei, aus der sie per *LoadIcon()* ausgelesen werden können.
hCursor	Handle zum Mauszeiger, der angezeigt werden soll, sobald sich der Cursor über dem Fenster befindet. Kann per *LoadCursor()* aus der Ressource-Datei geladen werden.
hbrBackground	Handle zu dem Brush („Pinsel“), mit dem der Fensterhintergrund ausgefüllt werden soll. Ein normales Fenster mit weißem Hintergrund erhält man durch Verwendung eines standardisierten weißen Pinsels, auf den durch *GetStockObject* Zugriff erlangt werden kann.
lpszMenuName	Gibt den Namen des für das Fenster gültigen Menüs an.
lpszClassName	Der Name der zu erzeugenden Fensterklasse. Da keine zwei Klassen mit demselben Namen angelegt werden dürfen, muss es sich hierbei um eine eindeutige Bezeichnung handeln.
hIconSm	Handle zu einem Icon, das die Fensterklasse repäsentiert. Es taucht in der linken oberen Ecke des Hauptrahmenfensters und bei Minimierung eines Fensters in der Windows-Startleiste auf.

Stilflags für WNDCLASSEX.style

Flag	Bedeutung
CS_CLASSDC	Legt einen Gerätekontext für alle Fenster derselben Klasse an. Es wird jedoch immer nur einem Fenster zur Zeit der Zugriff gewährt.
CS_DBLCLKS	Führt der Anwender einen Doppelklick im Fenster aus, wird eine entsprechende Botschaft an dieses Fenster abgesetzt.
CS_GLOBALCLASS	Definiert die Klasse als global.
CS_HREDRAW	Ein Fenster mit diesem Stilbit wird neu gezeichnet, sobald eine eine horizontale Größenveränderung stattgefunden hat, das Fenster also in der Breite verändert wurde.
CS_NOCLOSE	Deaktiviert den *Schließen*-Befehl im System Menü.
CS_OWNDC	Legt für jedes Fenster einen eigenen Gerätekontext an.
CS_PARENTDC	Hiermit wird einem Kindfenster gestattet, auch im Bereich des Elternfensters zu zeichnen. Dieses Fenster erhält einen eigenen Gerätekontext und benutzt nicht den des übergeordneten Fensters.
CS_SAVEBITS	Speichert in einer Bitmap den Bereich des Bildschirms, der von einem Fenster überdeckt wird. Mit diesen Daten kann dann der unterliegende Bereich wiederhergestellt werden, sobald das Fenster verschoben, minimiert oder geschlossen wird. Eventuell verdeckte Fenster erhalten in diesem Fall auch keine *WM_PAINT*-Nachricht. *CS_SAVEBITS* findet hauptsächlich bei Dialogboxen Anwendung.
CS_VREDRAW	Analog zu *CS_HREDRAW* wird ein Fenster mit diesem Stilbit komplett neu gezeichnet, sobald eine vertikale Größenänderung stattgefunden hat, es also in der Höhe verändert wurde.

CreateWindow(Ex)

```
HWND CreateWindowEx
(
DWORD dwExStyle
LPCTSTR lpClassName,
LPCTSTR lpWindowName,
DWORD dwStyle,
int x,
int y,
int nWidth,
int nHeight,
HWND hWndParent,
HMENU hMenu,
HANDLE hInstance,
LPVOID lpParam
);
```

Parameter	Bedeutung
dwExStyle	Erweiterte Stilelemente (siehe unten)
lpClassName	Hier wird der Name der zugrunde liegenden Fensterklasse angegeben.
lpWindowName	Der Text, der in der Titelzeile des Fensters ausgegeben wird.
dwStyle	Legt den Fensterstil fest (siehe Tabelle Fensterstile)
x, y	Die anfängliche linke obere Ecke des Fensters. Für ein Hauptfenster ist diese Angabe absolut, für ein Kindfenster relativ zur linken oberen Ecke des Hauptfensters.
nWidth, nHeight	Die anfängliche Breite und Höhe des Fensters.
hWndParent	Gibt an, ob das Fenster ein Kindfenster ist. In diesem Fall ist ein gültiges Handle anzugeben, für ein Hauptfenster wird NULL eingetragen.
hMenu	Handle zu einem evtl. vorhandenen Menü.
hInstance	Handle zur Instanz, die das Fenster generiert hat.
lpParam	Erweiterter Parameter, der an die Nachricht *WM_CREATE* weitergereicht wird.

Standardstile beim Erzeugen von Fenstern

Die folgenden Stile können für den *style* Parameter von *CreateWindowEx* eingesetzt werden:

Stilflag	Bedeutung
WS_BORDER	Das Fenster hat einen dünnen Rand.
WS_CAPTION	Das Fenster hat eine Titelzeile.
WS_CHILD	Es handelt sich um ein Kindfenster, das keine Menüzeile haben kann. *WS_CHILD* darf nicht zusammen mit *WS_POPUP* verwendet werden.
WS_EX_TOPMOST	Dieses Fenster steht immer über allen anderen als nicht *WS_EX_TOPMOST* deklarierten Fenstern. Dieses Flag wird von Fenstern genutzt, die immer im Vordergrund stehen sollen. Dies können zum Beispiel Dialogfenster sein, die eine Fortschrittsanzeige einer Berechnung darstellen, aber auch die von Windows her bekannte Task-Leiste mit dem Startbutton, die immer über allen anderen Fenstern steht, nutzt dieses Flag.
WS_HSCROLL	Das Fenster hat eine horizontale Scrolleiste.
WS_MAXIMIZE	Das Fenster wird nach der Erzeugung maximiert dargestellt.
WS_MINIMIZE	Analog zu *WS_MAXIMIZE* wird das Fenster nach der Erzeugung minimiert dargestellt.
WS_OVERLAPPED	Erzeugt ein Standardfenster, das eine Titelzeile, einen dünnen Rand und keine Kontrollelemente enthält.
WS_OVERLAPPED-WINDOW	Erzeugt ein Standardfenster, das eine Titelzeile und einen dicken Rand besitzt.
WS_POPUP	Erzeugt ein Popup-Fenster.
WS_VISIBLE	Das Fenster ist von Anfang an sichtbar.
WS_VSCROLL	Analog zu *WS_HSCROLL* erhält das Fenster eine vertikale Scrolleiste.

Erweiterte Stile beim Erzeugen von Fenstern

Ein wesentliches Element bei der Erzeugung eines Fenster (siehe auch *CFrameWnd::Create* oder *CreateWindowEx*) ist der *dwExStyle* Parameter, der die folgenden Werte annehmen kann:

Stil	Bedeutung
WS_EX_ACCEPTFILES	Gibt an, dass das Fenster Dateien über Drag & Drop akzeptiert.
WS_EX_CLIENTEDGE	Gibt dem Fenster einen versenkten Rahmen (wie zum Beispiel bei herkömmlichen *Suchen/Ersetzen*-Dialogfenstern).
WS_EX_CONTEXTHELP	Fenster mit diesem Stil besitzen ein Fragezeichensymbol in der Titelzeile des Fensters. Wird auf dieses Fragezeichen geklickt, wechselt der Mauszeiger ebenfalls in eins. Beim Klicken in eins der Kindfenster wird eine WM_HELP Nachricht an dieses verschickt.
WS_EX_CONTROLPARENT	Gestattet das Verwenden der ⇆-Taste, um zwischen den einzelnen Kindfenstern hin und her zu springen.
WS_EX_DLGMODALFRAME	Erzeugt ein Fenster mit einem doppelten Rand.
WS_EX_LEFT	Wenn es um das Ausrichten von Elementen innerhalb Fensters geht, wird immer von der linken Seite ausgegangen. Dieses ist die Standardeinstellung.
WS_EX_LEFTSCROLLBAR	Dieser Stil wird nur dann akzeptiert, wenn eine Sprache wie Hebräisch oder Arabisch verwendet wird.
WS_EX_LTRREADING	Der im Fenster ausgegebene Text wird von links nach rechts ausgerichtet. Dieses ist die Standardeinstellung.
WS_EX_MDICHILD	Erzeugt ein MDI-Kindfenster.

Stil	Bedeutung
WS_EX_NOPARENTNOTIFY	Ein Kindfenster, das mit diesem Flag erzeugt wird, sendet keine *WM_PARENTNOTIFY*-Nachricht an sein Elternfenster, wenn es erzeugt oder zerstört wird.
WS_EX_OVERLAPPEDWINDOW	Kombiniert die Stile *WS_EX_CLIENTEDGE* und *WS_EX_WINDOWEDGE*.
WS_EX_PALETTEWINDOW	Kombiniert die Stile *WS_EX_WINDOWEDGE*, *WS_EX_TOOLWINDOW* und *WS_EX_TOPMOST*.
WS_EX_RIGHT	Im Gegensatz zu *WS_EX_LEFT* werden Elemente standardmäßig nach rechts ausgerichtet. Wird nur akzeptiert, wenn wie bei *WS_EX_LEFTSCROLLBAR* eine Sprache wie Hebräisch oder Arabisch eingestellt ist.
WS_EX_RIGHTSCROLLBAR	Eine möglicherweise verwendete vertikale Bildlaufleiste wird an der rechten Seite des Fensters positioniert (Standardeinstellung).
WS_EX_RTLREADING	Bildet das Gegenteil von *WS_EX_LTRREADING*, nämlich von rechts nach links ausgericheteten Text und wird nur bei Sprachen wie Arabisch oder Hebräisch akzeptiert.
WS_EX_STATICEDGE	Erzeugt ein Fenster mit einem dreidimensionalen Rand. Wird in der Regel für Fenster verwendet, die keine Benutzereingaben zulassen.
WS_EX_TOOLWINDOW	Erzeugt eine frei bewegliche Werkzeugleiste.
WS_EX_TOPMOST	Ein Fenster mit diesem Stilbit befindet sich immer über allen anderen nicht als *WS_EX_TOPMOST* gekennzeichneten Fenstern, auch wenn es deakitivert ist.

Stil	Bedeutung
WS_EX_TRANSPARENT	Dieses Flag bestimmt, dass der Fensterinhalt dieses Fensters erst dann neu gezeichnet wird, nachdem die hinter ihm angeordneten Geschwisterfenster neu gezeichnet wurden.
WS_EX_WINDOWEDGE	Gibt an, dass dieses Fenster einen angehobenen Rand haben soll.

CFrameWnd::Create

Erzeugt ein neues Rahmenfenster. Dient zum Initialisieren eines Objekts der *CFrameWnd* oder einer davon abgeleiteten Klasse.

```
virtual BOOL Create
(
   LPCTSTR lpszClassName,
   LPCTSTR lpszWindowName,
   DWORD dwStyle = WS_OVERLAPPEDWINDOW,
   const RECT& rect = rectDefault,
   CWnd* pParentWnd = NULL,
   LPCTSTR lpszMenuName = NULL,
   DWORD dwExStyle = 0,
   CCreateContext* pContext = NULL
);
```

Die Parameter entscheiden dabei darüber, welches Aussehen das entstehende Fenster haben wird. Ihre Bedeutung im Einzelnen:

Parameter	Bedeutung
lpszClassName	Der Klassenname für die zu verwendende, registrierte Fensterklasse (siehe auch *RegisterClass*). Wird kein Name angegeben, werden die *CFrameWnd*-Standardattribute für das Erzeugen des Fensters verwendet.
lpszWindowName	Name der zu erzeugenden Fensters, wird in der Titelleiste ausgegeben
dwStyle	Spezifziert die Windows-Style-Attribute. Für eine Übersicht über die zur Verfügung stehenden Attribute schauen Sie weiter oben in der zugehörigen Tabelle nach.

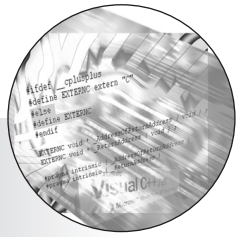
Parameter	Bedeutung
rect	Rechteck, das die Größe und Position des Fensters spezifiziert. Wird rectDefault angegeben, gestatten Sie Windows, diese Daten selbstständig zu setzen.
pParentWnd	Gibt das Elternfenster für das zu erzeugende Fenster an. Sollte bei Toplevel-Fenstern auf NULL gesetzt werden.
lpszMenuName	Gibt den Name der zu verwendenden Menü-Ressource an.
dwExStyle	Spezifiziert die erweiterten Stilattribute für dieses Fenster. Für eine Übersicht der zur Verfügung stehenden Attribute schauen Sie in den Anhang dieses Buches.
pContext	Zeiger auf eine *CCreateContext*-Struktur, darf auf NULL gesetzt werden. Eine Übersicht über diese Struktur finden Sie in der folgenden Tabelle.

CCreateContext

Fortgeschrittene Struktur, die zur Verknüpfung zwischen Dokument und Ansichten verwendet wird. Braucht nur spezifiziert zu werden, falls eigene Initialisierungsprozeduren eingesetzt werden.

Die Komponenten der Struktur im Einzelnen:

Komponente	Bedeutung
m_pNewViewClass	Zeiger auf *CRuntimeClass*-Objekt der neuen Ansicht.
m_pCurrentDoc	Dokument, das mit der Ansicht verbunden werden soll. Das Dokument muss existent sein.
m_pNewDocTemplate	Dokumentvorlage im Rahmen von MDI-Dokumenten.
m_pLastView	Zeiger auf die ursprüngliche Ansicht für das Dokument.
m_pCurrentFrame	Zeiger auf das aktuelle Framewindow, wird benötigt, wenn weitere Framewindows angebunden werden sollen.



Werkzeugklassen

B

MFC-Werkzeugklassen

Innerhalb der MFC stehen die folgenden Werkzeugklassen zur Verfügung:

- *CPoint* – Die Struktur speichert in zwei Komponenten x und y die Koordinaten eines Punkts.
- *CSize* – In den zwei Komponenten *cx* und *cy* kann je eine Größenangabe gespeichert werden, z. B. die Seitenlängen eines Rechtecks.
- *CRect* – Die vier Elementvariablen *top*, *bottom*, *left* und *right* speichern die Eckpunkt-Koordinaten eines Rechtecks.
- *CTime* – Eine auf der von C++ bekannten *time_t*-Struktur basierende Klasse, mit deren Hilfe die aktuelle Zeit und das aktuelle Datum bestimmt werden können.
- *CTimeSpan* – Diese Klasse kapselt in Sekundenschritten eine Zeitspanne, mit der dann zum Beispiel geprüft werden kann, wie viele Sekunden seit dem 1.1.1970 verstrichen sind. Der Wert wird in einem Langwort gespeichert und arbeitet somit bis zum 18.03.2038 korrekt.
- *CString* – Eine sehr interessante Klasse zur Arbeit mit Strings. Es werden Vergleichsoperatoren, Zuweisungen usw. angeboten, außerdem ist eine dynamische Größenveränderung von Strings möglich. Die von C++ her bekannten Befehle wie *strcpy* und *strcmp* werden nicht weiter benötigt. Außerdem kann ein *CString*-Objekt problemlos als *const char** oder *LPCTSTR*-Funktionsargument verwendet werden.

Die nachfolgenden Sourcecodezeilen geben einen kurzen Überblick über die mögliche praktische Verwendung dieser Klassen:

```
   // CPoint Struktur erzeugen. Das Objekt
// repräsentiert dann eine Koordinate,
   // nämlich den Punkt (20, 40).
   CPoint point(20,40);

   CPoint point2;// Alternative
   point2.x = 20;
   point2.y = 40;

   // Eine CSize Struktur füllen. Wir verwenden sie
// im Anschluß als
   // Längenangaben für eine CRect Struktur
   CSize size(100,50);

   CSize size2;
   size2.cx = 100;// Alternative
   size2.cy = 50;
```

```
   // Eine CRect Struktur, diese kann zum Beispiel
// für den
   // InvalidateRect Aufruf verwendet werden, oder
// zum Zeichnen eines
   // Rechtecks ins Fenster. Wir verwenden die
// vorher definierten CPoint und
   // CSize zum Initialisieren der Struktur.
   CRect rect(point, size);

   CRect rect2;// Alternative
   rect2.left    = point.x;
   rect2.right   = rect2.left + size.cx;
   rect2.top     = point.y;
   rect2.bottom  = rect2.top + size.cy;

   // Wir speichern die aktuelle Zeit in einer
// CTime Struktur
   CTime time = CTime::GetCurrentTime();

   // Alternativ geben wir selbst die gewünschte
// Zeit an
   CTime time2 (2002, 5, 5, 12, 27, 33);
// 5. Mai 2002, 12:27Uhr, 33 Sekunden
   time2.GetDay();    // liefert 5
   time2.GetHour();   // liefert 12
   time2.GetSecond(); // liefert 33

   // Die Verwendung von CTimeSpan gestaltet sich
// ähnlich einfach. Entweder
   // initialisieren wir durch Angabe von
// verstrichenen Tagen, Stunden, Minuten
   // und Sekunden:
   CTimeSpan timespan (3, 1, 5, 12);
// 3 Tage, 1 Stunde, 5 Minuten, 12 Sekunden

   // oder nur durch die Angabe von Sekunden
   CTimeSpan timespan2(263112);
   timespan2.GetTotalHours();// Ergibt 73
   timespan2.GetTotalMinutes();// Ergibt 4385

   // Auch die Verwendung von CString gestaltet
// sich sehr komfortabel:
   CString string1 ("Teststring1 ");
   CString string2 ("Teststring2");

   CString string3;
   string3 = string1 + string2; // ergibt
// 'Teststring1 Teststring2'
```

```
if (!(string3 == string1))
{
   string3 = string1.Left(4); // ergibt Test'
   string3.MakeUpper();// ergibt 'TEST'
}
```

Die folgenden Tabellen geben einen Überblick über die Elementmethoden der einzelnen Werkzeugklassen und sollen zur schnellen Referenz bei der täglichen Arbeit dienen:

CPoint

Methode	Beschreibung
Offset	Addiert die übergebene X-/Y-Komponente zum *CPoint*-Objekt hinzu.
==, !=	Prüft zwei *CPoint*-Objekte auf Gleichheit, bzw. Ungleichheit.
+=, -=	Addiert (subtrahiert) zu einem *CPoint*-Objekt ein weiteres *CPoint*-Objekt oder ein *CSize*-Objekt hinzu.
+	Gibt die Summe eines *CPoint*-Objekts mit einer Größe (Typ *size*) oder einem Punkt zurück. (Kann *CPoint* oder *CRect*-Werte zurückliefern).
-	Gibt die Differenz zwischen einem *CPoint*-Objekt und eine Größe (Typ *size*) zurück. (Kann *CPoint*, *CSize* oder *CRect*-Werte zurückliefern.)

CRect

Methode	Beschreibung
Width	Berechnet die Breite eines *CRect*-Objekts.
Height	Berechnet die Höhe eines *CRect*-Objekts.
Size	Berechnet die Größe eines *CRect*-Objekts.
TopLeft	Gibt den oberen linken Punkt eines *CRect*-Objekts zurück.
BottomRight	Gibt den unteren rechten Punkt eines *CRect*-Objekts zurück.

Methode	Beschreibung
CenterPoint	Gibt den Mittelpunkt eines *CRect*-Objekts zurück.
IsRectEmpty	Gibt an, ob das *CRect*-Objekt leer ist. Das ist der Fall, wenn Breite und Höhe gleich 0 sind.
IsRectNull	Gibt an, ob alle vier Werte für linken, rechten, oberen und unteren Rand gleich 0 sind.
PtInRect	Gibt an, ob ein übergebener Punkt innerhalb des *CRect*-Objekts liegt.
SetRectEmpty	Setzt alle Koordinaten des Rechtecks auf 0.
CopyRect	Kopiert die Koordinaten eines Quellrechtecks in das *CRect*-Objekt.
EqualRect	Bestimmt, ob das übergebene Rechteck mit dem *CRect*-Objekt übereinstimmt.
InflateRect	Erhöht Breite und Höhe des *CRect*-Objekts.
DeflateRect	Verringert Breite und Höhe des *CRect*-Objekts.
NormalizeRect	Normalisiert Höhe und Breite des *CRect*-Objekts.
OffSetRect	Verschiebt das *CRect*-Objekt um die übergebenen Koordinaten.
SubtractRect	Subtrahiert zwei Rechtecke voneinander.
IntersectRect	Setzt das *CRect*-Objekt auf die Schnittmenge zweier Rechtecke.
UnionRect	Setzt das *CRect*-Objekt auf die Vereinigung zweier Rechtecke.
LPCRECT	Konvertiert *CRect* in ein *LPCRECT*.
LPRECT	Konvertiert *CRect* in ein *LPRECT*.
=	Kopiert ein Rechteck ins *CRect*-Objekt.
==, !=	Prüft, ob das *CRect*-Objekt mit dem übergebenen Rechteck (nicht) übereinstimmt.
+=, -=	Addiert (subtrahiert) die übergebenen Koordinaten zum Rechteck.
&=	Setzt das *CRect*-Objekt auf die Schnittmenge zwischen dem *CRect*-Objekt und einem übergebenen Rechteck.

Methode	Beschreibung
|=	Setzt das *CRect*-Objekt auf die Vereinigung zwischen dem *CRect*-Objekt und einem übergebenen Rechteck.
+, -	Gibt die Summe (Differenz) vom *CRect*-Objekt mit zwei übergebenen Koordinaten zurück.
&, |	Gibt das Schnittmengen-Rechteck eines *CRect*-Objekts mit einem weiteren Rechteck zurück.

CSize

Methode	Beschreibung
==, !=	Prüft zwei *CSize*-Objekte auf Gleichheit bzw. Ungleichheit.
+=, -=	Addiert (subtrahiert) eine Größe (Typ *size*) von einem *CSize*-Objekt.
+, -	Addiert (subtrahiert) zwei Größen (Type *size*) voneinander.

CString

Methode	Beschreibung
GetLength	Gibt die Anzahl der Zeichen in einem *CString*-Objekt zurück.
IsEmpty	Prüft, ob das *CString*-Objekt leer ist.
Empty	Setzt die Länge eines *CString*-Objekts auf 0.
GetAt	Gibt ein Zeichen an der angegebenen Position zurück
[]	Gibt ein Zeichen an der angegebenen Position zurück
SetAt	Setzt ein Zeichen an der angegebenen Position.
LPCTSTR	Konvertiert das *CString*-Objekt in einen C-String.
=	Weist einem *CString*-Objekt eine neue Zeichenkette zu.
+	Fügt zwei Zeichenketten zusammen.
+=	Hängt eine Zeichenkette an das *CString*-Objekt an.

Methode	Beschreibung
==,<, >, <=, >=, !=	Vergleichsoperationen zwischen zwei Zeichenketten.
Compare	Vergleicht zwei Zeichenketten.
CompareNo-Case	Vergleicht zwei Zeichenketten (ohne Beachtung der Groß-/Kleinschreibung)
Mid	Extrahiert den mittleren Teil einer Zeichenkette.
Left	Extrahiert den linken Teil einer Zeichenkette.
Right	Extrahiert den rechten Teil einer Zeichenkette.
SpanIncluding	Gibt einen Teilstring zurück, der nur die in übergebenen Zeichen enthält.
SpanExcluding	Gibt einen Teilstring zurück, der nur Zeichen enthält, die der Methode nicht übergeben wurden.
MakeUpper	Wandelt alle Zeichen in Großbuchstaben um.
MakeLower	Wandelt alle Zeichen in Kleinbuchstaben um.
MakeReverse	Dreht die Zeichenkette um (Beispiel: *Hallo -> ollaH*).
Replace	Ersetzt angegebene Zeichen durch andere Zeichen.
Remove	Entfernt angegebene Zeichen aus einem String.
Insert	Fügt eine Zeichenkette an einer angegebenen Position in den String ein.
Delete	Entfernt eine Zeichenkette aus einem String.
Format	Formatiert den String kompatibel zu *sprintf*.
FormatV	Formatiert den String kompatibel zu *vsprintf*.
TrimLeft	Entfernt führende Leerzeichen aus der Zeichenkette.
TrimRight	Entfernt nachfolgende Leerzeichen aus der Zeichenkette.
Find	Sucht eine Zeichenkette innerhalb des Strings.
ReverseFind	Sucht eine Zeichenkette innerhalbs des Strings und beginnt dabei von hinten.
FindOneOf	Findet aus einer Reihe von übergebenen Zeichen das erste, das auch im String vorkommt.

Methode	Beschreibung
<<, >>	Ein- und Ausgabe eines *CString*-Objekts in oder aus einem Archiv.
GetBuffer	Gibt einen Zeiger auf die eigentliche Zeichenkette zurück.
ReleaseBuffer	Gibt den durch GetBuffer erlangten Zugriff auf die Zeichenkette wieder frei.
FreeExtra	Entfernt mittlerweile redundanten Speicher, der durch vorangeganene Operationen auf dem *CString*-Objekt angelegt wurde.
LoadString	Lädt eine Zeichenkette aus einer Windows-Ressource.
AnsiToOem	Konvertiert einen String von ANSI nach OEM.
OemToAnsi	Konvertiert einen String von OEM nach ANSI.

CTime

Methode	Beschreibung
GetCurrent-Time	Erzeugt ein *CTime*-Objekt, das die augenblickliche Zeit enthält.
GetTime	Gibt eine *time_t*-Struktur zurück, die die Werte des *CTime*-Objekts enthält.
GetYear	Gibt das Jahr zurück.
GetMonth	Gibt den Monat zurück.
GetDay	Gibt den Tag zurück.
GetHour	Gibt die Stunde zurück.
GetMinute	Gibt die Minute zurück.
GetSecond	Gibt die Sekunde zurück.
GetDayOfWeek	Gibt den Wochentag zurück (als numerischen Wert: 1 = Sonntag, 2 = Montag etc.).
Format	Konvertiert ein *CTime*-Objekt in einen formatierten String.
=	Weist einen neuen Zeitwert zu.

Methode	Beschreibung
+, -	Addiert (subtrahiert) *CTimeSpan* und *CTime*-Objekte.
+=, -=	Addiert (subtrahiert) ein *CTimeSpan*-Objekt vom *CTime*-Objekt.
==,<, >, <=, >=, !=	Vergleichsoperationen zwischen zwei *CTime*-Objekten.
<<, >>	Ein- und Ausgabe eine *CTime*-Objekts in oder aus einem *CArchive*.

CTimeSpan

Methode	Beschreibung
GetDays	Gibt die Zahl der Tage im *CTimeSpan*-Objekt zurück.
GetHours	Gibt die Anzahl der Stunden am aktuellen Tag zurück.
GetTotalHours	Gibt die Zahl der Stunden im *CTimeSpan*-Objekt zurück.
GetMinutes	Gibt die Anzahl der Minuten in der aktuellen Stunde zurück.
GetTotalMinutes	Gibt die Zahl der Minuten im *CTimeSpan*-Objekt zurück.
GetSeconds	Gibt die Anzahl der Sekunden in der aktuellen Minute zurück.
GetTotalSeconds	Gibt die Zahl der Sekunden im *CTimeSpan*-Objekt zurück.
Format	Konvertiert ein *CTimeSpan*-Objekt in einen formatierten String.
=	Weist einen neuen Zeitwert zu.
+, -	Addiert (subtrahiert) *CTimeSpan*-Objekte.
+=, -=	Addiert (subtrahiert) ein *CTimeSpan*-Objekt von diesem *CTimeSpan*-Objekt.
==,<, >, <=, >=, !=	Vergleichsoperationen zwischen zwei *CTimeSpan*-Objekten.
<<, >>	Ein- und Ausgabe eines *CTimeSpan*-Objekts in oder aus einem *CArchive*.

MFC-Containerklassen

Die MFC stellen drei Containerklassen zur Verfügung, die zur Verwaltung von eigenen Daten dienen. Die Methoden, die jeweils verwendet werden können, sind im Folgenden aufgeführt.

CList

CList entspricht der klassischen doppelt verketteten Liste mit entsprechenden Zugriffsfunktionen. Der Funktionsprototyp sieht so aus:

```
template< class TYPE, class ARG_TYPE = const TYPE& >
class CList : public CObject
```

Dabei ist *TYPE* der Typ der aufzunehmenden Elemente, *ARG_TYPE* der Typ der verwendet werden soll, wenn Elemente dieser Liste per Parameter übergeben werden. In der Regel ist *ARG_TYPE* eine Referenz auf den *TYPE*-Typ.

Methode	Beschreibung
GetHead	Gibt das erste Element der Liste zurück (Liste darf nicht leer sein).
GetTail	Gibt das letzte Element der Liste zurück (Liste darf nicht leer sein).
AddHead	Fügt ein Element am Anfang der Liste ein. Dieses Element kann selbst wieder eine *CList* mit Inhalten des gleichen Typs sein.
AddTail	Fügt ein Element am Ende der Liste ein. Dieses Element kann selbst wieder eine *CList* mit Inhalten des gleichen Typs sein.
RemoveAll	Entfernt sämtliche Elemente aus der Liste.
RemoveHead	Entfernt das erste Element aus der Liste.
RemoveTail	Entfernt das letzte Element aus der Liste.
GetHeadPosition	Gibt das *POSITIONs*-Objekt für das erste Element der Liste zurück.
GetNext	Gibt das von einem *POSITIONs*-Objekt abhängige Element zurück und setzt das *POSITIONs*-Objekt auf das nächste Element der Liste.
GetPrev	Gibt das von einem *POSITIONs*-Objekt abhängige Element zurück und setzt das *POSITIONs*-Objekt auf das vorige Element der Liste.

Methode	Beschreibung
GetTailPosition	Gibt das *POSITIONs*-Objekt für das letzte Element der Liste zurück.
GetAt	Gibt das Element an der spezifizierten Position zurück.
RemoveAt	Entfernt das Element an der spezifizierten Position.
SetAt	Setzt das Element an der spezifizierten Position mit neuen Werten.
InsertAfter	Fügt ein Element hinter einer angegebenen Position ein.
InsertBefore	Fügt ein Element vor einer angegebenen Position ein.
Find	Gibt das *POSITIONs*-Objekt eines zu suchenden Elements zurück oder NULL, falls das Element nicht gefunden wurde.
FindIndex	Gibt das *POSITIONs*-Objekt eines gesuchten Elements zurück. Dabei wird die Position des Elements in der sequentiellen Liste numerisch angegeben, 0 entspricht also dem ersten, 1 dem zweiten Element und so weiter. Gibt NULL zurück, wenn die gesuchte Position nicht existiert.
GetCount, GetSize	Gibt die Anzahl der Elemente in der Liste zurück.
IsEmpty	Gibt *true* zurück, falls die Liste leer ist, sonst *false*.

CMap

Ein *CMap*-Objekt speichert Elemente, die jeweils aus einem Schlüsselwert und dem eigentlichen Inhalt bestehen. Mögliche Paare sind also beispielsweise (1, „Hallo"), (57, 1) oder (113, 1.567), falls der Schlüssel jeweils als Integertyp deklariert ist.

Der Template-Prototyp sieht wie folgt aus:

```
template< class KEY, class ARG_KEY, class VALUE, class ARG_VALUE
>class CMap : public CObject
```

Dabei sind *KEY* und *ARG_KEY* der Datentyp, der als Schlüsselwert bzw. als Schlüsselwert in Parameterübergaben verwendet wird (in der Regel ist *ARG_KEY* eine Referenz vom *KEY*-Typ) und *VALUE* bzw. *ARG_VALUE* das eigentliche Element, wobei *ARG_VALUE* in der Regel eine Referenz vom *VALUE*-Typ ist.

Die Wertepaare werden in einer Hashtable abgelegt, es ist möglich, die Größe dieser Hashtable anzugeben, wie aus der folgenden Methodenübersicht hervorgeht:

Methode	Beschreibung
GetHashTableSize	Gibt die Größe der Hashtable zurück.
GetNextAssoc	Gibt das nächste Element zu Iteration zurück. Es ist nicht sichergestellt, in welcher Reihenfolge die Elemente zurückgeliefert werden, insbesondere in der Regel nicht nach Schlüsselwerten sortiert!
PGetNextAssoc	Gibt einen Zeiger auf das nächste Element zur Iteration zurück. Es ist nicht sichergestellt, in welcher Reihenfolge die Elemente zurückgeliefert werden, insbesondere in der Regel nicht nach Schlüsselwerten sortiert!
GetStartPosition	Gibt ein *POSITIONs*-Objekt auf das erste Element zurück.
PGetFirstAssoc	Gibt einen Zeiger auf das erste Element zurück.
InitHashTable	Initialisiert die Hashtable und spezifiziert ihre Größe.
Lookup	Liest einen Wert aus, dessen Schlüssel an die Funktion übergeben wird.
PLookup	Gibt einen Zeiger auf einen Wert zurück, dessen Schlüssel an die Funktion übergeben wird.
RemoveAll	Entfernt sämtliche Elemente aus der Liste.
RemoveKey	Entfernt einen Wert aus der Liste, der zu einem übergebenen Schlüssel gehört.
SetAt	Fügt einen Wert an einer spezifizierten Position in die Hashtable ein und überschreibt Werte, die sich an derselben Position befinden.
GetCount, GetSize	Liefert die Anzahl der Elemente in der Hashtable zurück.
IsEmpty	Liefert *true* zurück, wenn die Hashtable leer ist, sonst *false*.

CArray

Die *CArray*-Klasse ist analog zu herkömmlichen Arrays zu betrachten, kann ihre Größe jedoch dynamisch während der Laufzeit eines Programms an die jeweilige Situation anpassen.

Der Template-Prototyp sieht wie folgt aus:

```
template < class TYPE, class ARG_TYPE = const TYPE& >
class CArray : public CObject
```

Dabei ist *TYPE* der Typ der aufzunehmenden Elemente, *ARG_TYPE* der Typ der verwendet werden soll, wenn Elemente dieser Liste per Parameter übergeben werden. In der Regel ist *ARG_TYPE* eine Referenz auf den *TYPE*-Typ.

Methode	Beschreibung
GetCount, GetSize	Gibt die Anzahl an Elementen im Array zurück.
GetUpperBound	Gibt den größten gültigen Arrayindex zurück.
IsEmpty	Gibt *true* zurück, falls das Array leer ist, sonst *false*.
SetSize	Legt die Anzahl der Elemente fest, die in das Array eingeordnet werden können.
FreeExtra	Gibt den Speicher frei, der für nicht belegte Elemente oberhalb des letzten gültigen Elements reserviert ist.
RemoveAll	Entfernt sämtliche Elemente aus dem Array.
ElementAt	Gibt einen Zeiger auf das Element an einer bestimmten Position zurück.
GetAt	Gibt den Wert des Elements an einer bestimmten Position im Array zurück.
GetData	Erlaubt direkten Zugriff auf Elemente innerhalb eines Arrays (dient zur Veränderung von Werten).
SetAt	Setzt den Wert eines Elements an einer bestimmten Stelle im Array. Die Arraygröße wird hierdurch nicht beeinflusst.
Add	Hängt ein Element an das Ende des Arrays an, vergrößert das Array bei Bedarf.
Append	Hängt ein zweites Array an das bestehende Array an, vergrößert das Array bei Bedarf.
Copy	Kopiert ein anderes Array in das bestehende Array, vergrößert das Array bei Bedarf.

Methode	Beschreibung
SetAtGrow	Wie *SetAt*, vergrößert das Array jedoch, falls die übergebene Position zu groß ist.
InsertAt	Fügt ein Element an einer spezifizierten Position ein.
RemoveAt	Entfernt ein Element an einer spezifzierten Position.

GDI-Objekte

Windows stellt eine kleine Zahl an nützlichen Zeichenwerkzeugen zur Verfügung, die im Rahmen von Gerätekontexten zum Einsatz kommen können.

Im Einzelnen sind dies:

- *CBitmap* zur Darstellung von Bitmaps
- *CBrush* zum Füllen von Flächeninhalten
- *CFont* zum Schreiben von Text
- *CPen* zum Zeichen von Linien
- *CPalette* zur Realisierung einer Windows-Farbpalette
- *CRgn,* um Regionen innerhalb eines Fensters zu spezifizieren.

Die folgenden Seiten zeigen die wichtigsten Methode der jeweiligen Objekte auf.

CBitmap

CBitmap repräsentiert eine einfache Bitmap-Grafik.

Methode	Beschreibung
CreateBitmap	Erzeugt ein *CBitmap*-Objekt unter Verwendung einer im Speicher befindlichen Bitmapgrafik mit festgelegten Höhe, Breite und Bitpattern.
CreateBitmap-Indirect	Erzeugt ein *CBitmap*-Objekt unter Verwendung einer im Speicher befindlichen Bitmapgrafik. Breite, Höhe und Bitpattern werden gesondert angegeben.
CreateCompatible-Bitmap	Initialisiert ein *CBitmap*-Objekt mit einer Bitmap und macht es kompatibel zu einem vorgegebenen Gerät.

Methode	Beschreibung
CreateDiscardable-Bitmap	Erzeugt ein *CBitmap*-Objekt, das von Windows selbstständig abgeräumt werden kann, wenn es nicht in Benutzung ist. Wird versucht, eine abgeräumte Bitmap in einen Gerätekontext einzubinden, gibt *SelectObject* NULL zurück.
LoadBitmap	Erzeugt ein *CBitmap*-Objekt und hängt eine Bitmap aus den Ressourcen der Applikation an.
LoadMapped-Bitmap	Lädt eine Bitmapgrafik ein und biegt die Farben auf die derzeit eingestellten Systemfarben um.
LoadOEMBitmap	Lädt eine Windows-eigene Bitmap ein und hängt diese an das *CBitmap*-Objekt an. Eine Auflistung der verfügbaren Bitmaps ist im Anschluss an diese Tabelle zu finden.
GetBitmap	Schreibt die Attribute des *CBitmap*-Objekts in eine *BITMAP*-Struktur (siehe unten).
FromHandle	Nimmt ein *HBITMAP*-Handle entgegen und gibt einen Zeiger auf ein *CBitmap*-Objekt zurück.
GetBitmapBits	Transferiert die Daten der Bitmap in einen angegebenen Buffer zur weiteren Manipulation.
GetBitmapDimension	Gibt Breite und Höhe der Bitmap zurück. Muss vorher durch die *SetBitmapDimension*-Methode gesetzt werden.
SetBitmapBits	Kopiert Daten in die Bitmap.
SetBitmapDimension	Legt die Breite und Höhe der Bitmap fest.

Mögliche Werte für die windows-eigenen Bitmaps (siehe *LoadOEMBitmap*) sind: *OBM_BTNCORNERS, OBM_OLD_RESTORE, OBM_BTSIZE, OBM_OLD_RGARROW, OBM_CHECK, OBM_OLD_UPARROW, OBM_CHECKBOXES, OBM_OLD_ZOOM,OBM_CLOSE, OBM_REDUCE, OBM_COMBO, OBM_REDUCED, OBM_DNARROW, OBM_RESTORE, OBM_DNARROWD, OBM_RESTORED, OBM_DNARROWI, OBM_RGARROW, OBM_LFARROW, OBM_RGARROWD, OBM_LFARROWD, OBM_RGARROWI, OBM_LFARROWI, OBM_SIZE, OBM_MNARROW, OBM_UPARROW, OBM_OLD_CLOSE, OBM_UPARROWD, OBM_OLD_DNARROW, OBM_UPARROW, OBM_OLD_LFARROW, OBM_ZOOM, OBM_OLD_REDUCE und OBM_ZOOMD.*

Konstanten, die mit *OBM_OLD* beginnen, stellen Grafiken da, die vor der Windows-Version 3.0 verwendet wurden.

Zur Verwendung dieser OEM-Ressourcen, muss die Konstante *OEMRESOURCE* definiert sein, bevor die Include-Datei *windows.h* eingebunden wird.

BITMAP-Struktur

Die Windows-*BITMAP*-Struktur enthält Informationen über den Aufbau einer Bitmap. Sie können diese durch die *CBitmap*-Methode *GetBitmap* auslesen.

```
typedef struct tagBITMAP {
   int bmType;
   int bmWidth;
   int bmHeight;
   int bmWidthBytes;
   BYTE bmPlanes;
   BYTE bmBitsPixel;
   LPVOID bmBits;
} BITMAP;
```

Komponente	Bedeutung
bmType	Gibt den Typ der Bitmap an. Normalerweise 0.
bmWidth	Breite der Bitmap. Muss größer als 0 sein.
bmHeight	Höhe der Bitmap. Muss größer als 0 sein.
bmWidthBytes	Anzahl der Bytes in einer Bildzeile.
bmPlanes	Anzahl der Farbplanes in der Bitmap.
bmBitsPixel	Anzahl der Bits pro Pixel.
bmBits	Zeiger auf den Beginn der Bitmap-Bildinformationen.

CBrush

Ein *CBrush* ist eine Art Stempel, mit dem Flächeninhalte von geometrischen Figuren ausgefüllt werden können. Ein Brush kann dabei eine einfache Farbe, ein Muster, oder auch eine Bitmap sein.

Methode	Beschreibung
CreateBrush-Indirect	Initialisiert das *CBrush*-Objekt mit einem Stil, einer Farbe und einem Muster. Diese Daten sind in einer *LOGBRUSH*-Struktur angegeben (siehe unten).

Methode	Beschreibung
CreateDIBPattern-Brush	Initialisiert das *CBrush*-Objekt mit einem Muster, das durch eine DIB (device-independent Bitmap) angegeben wird.
CreateHatchBrush	Initialisiert das *CBrush*-Objekt mit einer angegebenen Farbe und einem angegebenen Muster.
CreatePattern-Brush	Initialisiert das *CBrush*-Objekt mit einem durch eine Bitmap angegebenen Muster.
CreateSolidBrush	Erzeugt einen einfarbigen *CBrush* unter Verwendung einer angegebenen Farbe.
CreateSysColor-Brush	Erzeugt ein *CBrush*-Objekt mit der momentan eingestellten Systemfarbe.
FromHandle	Nimmt ein Windows-*HBRUSH*-Handle entgegen und gibt einen passenden Zeiger auf ein *CBrush*-Objekt zurück.
GetLogBrush	Füllt eine *LOGBRUSH*-Struktur mit den Informationen des *CBrush*-Objekts.

LOGBRUSH

Ein *CBrush*-Objekt definiert sich über eine Struktur namens *LOGBRUSH*, die weitere Informationen über den Stil des Brushes enthält:

```
typedef struct tagLOGBRUSH {
  UINT     lbStyle;
  COLORREF lbColor;
  LONG     lbHatch;
} LOGBRUSH, *PLOGBRUSH;
```

Komponente	Beschreibung
lbStyle	Gibt den Stil des Brushes an. Für mögliche Stile, siehe unten.
lbColor	Farbe des Brushs. Wird ignoriert, wenn *lbStyle* auf *BS_HOLLOW* oder *BS_PATTERN* gesetzt ist.
lbHatch	Schraffur des Brushes. Siehe unten.

Mögliche Werte für LOGBRUSH::lbStyle

Wert	Beschreibung
BS_DIBPATTERN, BS_DIBPATTERN8X8	Ein Brush, der durch dein DIB-Grafik spezifiziert wird. *lbHatch* enthält ein Handle auf eine gepackte DIB.
BS_DIBPATTERNPT	Ein Brush, der durch dein DIB-Grafik spezifiziert wird. *lbHatch* enthält einen Zeiger auf eine gepackte DIB.
BS_HATCHED	Schraffierter Brush
BS_HOLLOW, BS_NULL	Leerer Brush
BS_PATTERN, BS_PATTERN8X8	Brush ist durch eine im Speicher befindliche Grafik definiert.
BS_SOLID	Einfarbiger Brush

Mögliche Werte für LOGBRUSH::lbColor

Abhängig von *lbStyle* kann *lbColor* verschiedenartige Werte annehmen. Ist *lbStyle* auf BS_SOLID oder BS_HATCHED gesetzt, enthält *lbColor* einen *COLORREF*-Wert der gewünschten Farbe.

Ist *lbStyle* auf *BS_DIBPATTERN* oder *BS_DIBPATTERNPT* gesetzt, kann *lbColor* einen der beiden folgenden Werte enthalten:

Werte	Beschreibung
DIB_PAL_COLORS	Brush verwendet die derzeit aktive Farbpalette.
DIB_RGB_COLORS	Brush verwendet echte RGB-Werte.

Ist *lbStyle* auf *BS_HOLLOW* oder *BS_PATTERN* gesetzt, wird der *lbColor*-Wert ignoriert.

Mögliche Werte für LOGBRUSH::lbHatch

Die Bedeutung von *lbHatch* ist abhängig von *lbStyle*.

Ist *lbStyle* auf *BS_DIBPATTERN* gesetzt, enthält *lbHatch* ein Handle auf eine gepackte DIB.

Ist *lbStyle* auf *BS_DIBPATTERNPT* gesetzt, enthält *lbHatch* einen Zeiger auf eine gepackte DIB.

Ist *lbStyle* auf *BS_PATTERN* gesetzt, enthält *lbHatch* ein Handle auf die zugrunde liegende Bitmap.

Ist *lbStyle* auf *BS_SOLID* oder *BS_HOLLOW* gesetzt, wird *lbHatch* ignoriert.

Ist *lbStyle* auf *BS_HATCHED* gesetzt, kann *lbHatch* einen der folgenden Werte annehmen:

Werte	Beschreibung
HS_BDIAGONAL	Brush wird mit Linien schraffiert, die sich im 45-Grad-Winkel von links unten nach rechts oben ziehen.
HS_CROSS	Brush wird mit horizontalen und vertikalen Linien schraffiert, die ein Karomuster bilden.
HS_DIAGCROSS	Brush wird mit diagonal gekreuzten Linien schraffiert.
HS_FDIAGONAL	Brush wird mit Linien schraffiert, die sich im 45-Grad-Winkel von von links oben nach rechts unten ziehen.
HS_HORIZONTAL	Brush wird mit horizontalen Linien schraffiert.
HS_VERTICAL	Brush wird mit vertikalen Linien schraffiert.

CFont

Ein *CFont*-Objekt repräsentiert einen Windows-Font, der über geeignete Methoden manipuliert werden kann:

Methode	Beschreibung
CreateFont	Erzeugt ein *CFont*-Objekt mit übergebenen Spezifikationen.
CreateFontIndirect	Erzeugt ein *CFont*-Objekt und verwendet die in einer *LOGFONT*-Struktur (siehe unten) übergebenen Spezifikationen.
CreatePointFont	Initialisiert ein *CFont*-Objekt mit einer in 1/10 Punkten angegebenen Höhe und einer Schriftart.

Methode	Beschreibung
CreatePointFontIndirect	Erzeugt ein *CFont*-Objekt und verwendet die in einer *LOGFONT*-Struktur (siehe unten) übergebenen Spezifikationen. Die Höhe des Fonts wird dabei in 1/10 Punkten interpretiert.
FromHandle	Übernimmt einen Windows-*HFONT*-Handle und gibt einen passenden *CFont*-Zeiger zurück.
GetlogFont	Füllt eine *LOGFONT*-Struktur mit den zum *CFont*-Objekt gehörenden Werten.

LOGFONT-Struktur

Die *LOGFONT*-Struktur beschreibt einen Zeichensatz:

```
typedef struct tagLOGFONT {
  LONG lfHeight;
  LONG lfWidth;
  LONG lfEscapement;
  LONG lfOrientation;
  LONG lfWeight;
  BYTE lfItalic;
  BYTE lfUnderline;
  BYTE lfStrikeOut;
  BYTE lfCharSet;
  BYTE lfOutPrecision;
  BYTE lfClipPrecision;
  BYTE lfQuality;
  BYTE lfPitchAndFamily;
  TCHAR lfFaceName[LF_FACESIZE];
} LOGFONT, *PLOGFONT;
```

Komponente	Beschreibung
lfHeight	Gibt die Höhe in logischen Einheiten an. Der Fontmapper interpretiert positve Angaben als tatsächliche Geräteeinheiten, negative Angaben werden als absolute Werte gelesen, ein Wert von 0 lässt den Fontmapper eine Standardgröße wählen.
lfWidth	Gibt die durchschnittliche Breite eines Zeichens an. Steht hier eine 0, vergleicht der Fontmapper das aktuelle Seitenverhältnis mit dem Seitenverhältnis der verfügbaren Zeichensätze.

Komponente	Beschreibung
lfEscapement	Gibt die Ausrichtung des Zeichensatz in 1/10 Grad zur horizontalen X-Achse an. Sollte den gleichen Wert haben wie *lfOrientation*.
lfOrientation	Gibt den Winkel in 1/10 Grad zwischen der untersten gedachten Linie jedes Zeichens und der horizontalen X-Achse an.
lfWeight	Gibt die Gewichtung des Zeichensatzes an und muss zwischen 0 und 1.000 liegen. Im Anschluss an diese Tabelle finden Sie eine Übersicht über einige bereits vordefinierte Konstanten für diese Komponente.
lfItalic	*true*, wenn dies ein kursiver Zeichensatz ist, sonst *false*.
lfUnderline	*true*, wenn dies ein unterstrichener Zeichensatz ist, sonst *false*.
lfStrikeOut	*true*, wenn dies ein durchgestrichener Zeichensatz ist, sonst *false*.
lfCharSet	Gibt die verfügbaren Zeichen des Fonts an, siehe unten.
lfOutPrecision	Gibt an, wie exakt der ausgegebene Zeichensatz mit dem beschriebenen Zeichensatz übereinstimmen soll. Für mögliche Werte siehe unten.
lfClipPrecision	Gibt an, wie präzise Zeichen abgeschnitten werden sollen, die sich nur teilweise auf einer darstellbaren Fläche befinden. Für mögliche Werte, siehe unten.
lfQuality	Gibt die Ausgabequalität für diesen Zeichensatz an. Für mögliche Werte, siehe unten.
lfPitchAndFamily	Gibt Steigung und Familie des Zeichensatzes an, siehe unten.
lfFaceName	Gibt den Schriftart Namen des Zeichensatzes an.

Mögliche Werte für LOGFONT::lfWeight

lfWeight gibt die Gewichtung eines Zeichensatzes an. Der Wert muss zwischen 0 und 1.000 liegen. Einige vordefinierte Konstanten:

Wert	Entspricht einem *lfWeight*-Wert von
FW_DONTCARE	0
FW_THIN	100
FW_EXTRALIGHT	200
FW_ULTRALIGHT	200
FW_LIGHT	300
FW_NORMAL	400
FW_REGULAR	400
FW_MEDIUM	500
FW_SEMIBOLD	600
FW_DEMIBOLD	600
FW_BOLD	700
FW_EXTRABOLD	800
FW_ULTRABOLD	800
FW_HEAVY	900
FW_BLACK	900

Mögliche Werte für LOGFONT::lfCharSet

Die verfügbaren Zeichen in einem Zeichensatz werden über die *lfCharSet* Komponente der *LOGFONT*-Struktur spezifiziert. Mögliche Werte sind *ANSI_CHARSET, BALTIC_CHARSET, CHINESEBIG5_CHARSET, DEFAULT_CHARSET, EASTEUROPE_CHARSET, GB2312_CHARSET, GREEK_CHARSET, HANGUL_CHARSET, MAC_CHARSET, OEM_CHARSET, RUSSIAN_CHARSET, SHIFTJIS_CHARSET, SYMBOL_CHARSET, TURKISH_CHARSET, VIETNAMESE_CHARSET*. Dazu kommen bei koreanischen Sprachversionen von Windows noch *JOHAB_CHARSET*, bei Sprachversionen des mittleren Ostens *ARABIC_CHARSET* und *HEBREW_CHARSET* und bei thailändischen Sprachversionen *THAI_CHARSET*.

Mögliche Werte für LOGFONT::lfOutPrecision

Wert	Beschreibung
OUT_CHARACTER_PRECIS	Wird nicht verwendet.

Wert	Beschreibung
OUT_DEFAULT_PRECIS	Standardpräzision des Fontmappers.
OUT_DEVICE_PRECIS	Lässt den Fontmapper bei mehreren Möglichkeiten einen Gerätezeichensatz wählen.
OUT_OUTLINE_PRECIS	Lässt den Fontmanager aus TrueType-Schriftarten auswählen (nur Windows NT/2000/XP).
OUT_RASTER_PRECIS	Lässt den Fontmapper bei mehreren Möglichkeiten einen Rasterzeichensatz wählen.
OUT_STRING_PRECIS	Rückgabewert für die Enumeration von Rasterzeichensätzen, wird vom Fontmapper nicht benutzt.
OUT_STROKE_PRECIS	Rückgabewert für Enumerationen von TrueType- oder Vektorzeichensätzen, wird vom Fontmapper nicht benutzt.
OUT_TT_ONLY_PRECIS	Lässt den Fontmanager nur aus den verfügbaren TrueType-Zeichensätzen auswählen. Wenn keine TrueType-Zeichensätze vorhanden sind, wählt der Fontmanager aber auch aus den übrigen aus.

Mögliche Werte für LOGFONT::lfClipPrecision

Wert	Beschreibung
CLIP_DEFAULT_PRECIS	Aktiviert Standardclipping.
CLIP_CHARACTER_PRECIS	Nicht verwendet.
CLIP_STROKE_PRECIS	Wird aus Kompatibilitätsgründen zu früheren Windows-Versionen bei der Enumeration von Zeichensätzen zurückgegeben.
CLIP_MASK	Nicht verwendet.
CLIP_EMBEDDED	Muss verwendet werden, wenn eingebettete Zeichensätze verwendet werden, die nur ausgelesen werden können.

Wert	Beschreibung
CLIP_LH_ANGLES	Gibt die Drehrichtung des Zeichensatzes an. Wird dieser angegeben, hängt die Drehrichtung davon ab, ob mit einem links- oder rechtshändigen Koordinatensystem gearbeitet wird. Ohne diesen Wert werden Zeichensätze stets im Uhrzeigersinn gedreht.
CLIP_TT_ALWAYS	Nicht verwendet.

Mögliche Werte für LOGFONT::lfQuality

Wert	Beschreibung
ANTIALIASED_QUALITY	Windows NT 4.0 und später: Zeichensatz wird immer geglättet dargestellt, so lange es der Zeichensatz erlaubt und er nicht zu groß oder zu klein ist. Windows 95 Plus, Windows 98, Windows ME: Damit der Zeichensatz geglättet werden kann, muss die Bildschirmeinstellung eine höhere Farbtiefe als 8 Bit aufweisen und nicht auf Palettenbasis arbeiten.
CLEARTYPE_QUALITY	Nur Windows XP: Zeichen werden nach der *ClearType*-Glättungsfunktion geglättet.
DEFAULT_QUALITY	Qualität des Zeichensatzes spielt keine Rolle.
DRAFT_QUALITY	Qualität des Zeichensatzes ist wichtiger als bei *DEFAULT_QUALITY,* aber niedriger als bei *PROOF_QUALITY*. Attribute wie kursiver oder fett gedruckter Text werden künstlich erzeugt.
NONANTIALIASED_QUALITY	Windows 95/98/ME, NT4.0 und später: Zeichensatz wird nie geglättet.
PROOF_QUALITY	Beste Qualität. Güte der Darstellung wird der exakten Übereinstimmung zwischen gewünschten und tatsächlichen Zeichensatzeigenschaften der *LOGFONT*-Struktur vorgezogen.

Mögliche Werte für lfPitchAndFamily

Wert	Beschreibung
FF_DECORATIVE	Verzierte Schriften. Beispiel: Old English.
FF_DONTCARE	Schriftfamilie ist unbekannt oder nicht relevant.
FF_MODERN	Beschreibt Zeichensätze mit konstanter Zeichenbreite und ohne Haarstriche, zum Beispiel Courier New.
FF_ROMAN	Zeichensätze mit veränderlicher Zeichenbreite und mit Haarstrichen. Zum Beispiel MS Serif.
FF_SCRIPT	Zeichensatz, der tatsächliche Handschrift angenähert ist. Zum Beispiel Script.
FF_SWISS	Zeichensatz mit veränderlichen Zeichenbreite und ohne Haarstriche, zum Beispiel MS Sans Serif.

CPen

Ein *CPen* ist im Kontext der GDC als Stift zu verstehen, mit dem beispielsweise Linien gezeichnet werden können. Er wird auch als Umrandungsinstrument für Rechtecke und andere geometrische Figuren benutzt.

Methode	Beschreibung
CreatePen	Erzeugt einen neuen Stift mit spezifiziertem Stil (siehe unten), Breite und Brush-Attributen.
CreatePenIndirect	Erzeugt einen neuen Stift unter Verwendung einer beschreibenden *LOGPEN*-Struktur (siehe unten).
FromHandle	Nimmt ein Standard-Windows-*HPEN*-Handle entgegen und gibt einen passen *Cpen*-Zeiger auf diesen zurück.
GetExtLogPen	Extrahiert die Beschreibungsdaten des Stifts in eine *EXTLOPEN*-Struktur (siehe unten).
GetLogPen	Extrahier die Beschreibungsdaten des Stifts in eine *LOGPEN*-Struktur (siehe unten).

Mögliche Stile für die CPen-Erzeugung

Stil	Beschreibung
PS_SOLID	Stift zieht eine ununterbrochene durchgezogene Linie.
PS_DASH	Stift zieht eine gestrichelte Linie.
PS_DOT	Stift zieht eine gepunktete Linie.
PS_DASHDOT	Stift zieht eine Linie die abwechselnd aus kurzen Strichen und Punkten besteht.
PS_DASHDOTDOT	Stift zieht eine Linie, die aus einer Folge „Strich-Punkt-Punkt“ zusammengesetzt wird.

Die LOGPEN-Struktur

Die *LOGPEN*-Struktur enthält Informationen über den Stil, Breite und Farbe eines Stifts und genügt dem folgenden Prototyp:

```
typedef struct tagLOGPEN {
  UINT     lopnStyle;
  POINT    lopnWidth;
  COLORREF lopnColor;
} LOGPEN, *PLOGPEN;
```

Komponente	Beschreibung
lopnStyle	Stil des Stifts, siehe oben.
lopnWidth	*POINT*-Struktur, deren X-Komponente die Breite des Stifts angibt – die Y-Komponente wird nicht verwendet. Ist dieser Zeiger NULL, hat der Stift eine Breite von einem Pixel.
lopnColor	*COLORREF*-Wert, der die Farbe des Stifts angibt.

Die EXTLOGPEN-Struktur

Die *EXTLOGPEN*-Struktur enthält im Vergleich zu *LOGPEN* eine ganze Reihe erweiterte Informationen:

```
typedef struct tagEXTLOGPEN {
  DWORD     elpPenStyle;
  DWORD     elpWidth;
  UINT      elpBrushStyle;
  COLORREF  elpColor;
```

```
  ULONG_PTR elpHatch;
  DWORD     elpNumEntries;
  DWORD     elpStyleEntry[1];
} EXTLOGPEN, *PEXTLOGPEN;
```

Komponente	Beschreibung
elpPenStyle	Enthält die Informationen zum Stiftstil. Nähere Informationen hierzu finden Sie unter dieser Tabelle.
elpWidth	Breite des Stifts. Ist *elpPenStyle PS_GEOMETRIC*, gibt dieser Werte die Breite des Stifts in logischen Einheiten an, ansonsten ist die Breite 1 Pixel.
elpBrushStyle	Nähere Informationen zum Stempelstil des Stifts. Siehe unten.
elpColor	Nähere Informationen zur Farbe des Stifts. Siehe unten.
elpHatch	Nähere Informationen zur Schraffierung des Stifts. Siehe unten.
elpNum-Entries	Gibt die Anzahl der Elemente im benutzerdefinierten Stilarray an. Hiermit ist es möglich, bei durchgezogenen Strichen zu bestimmen, in welcher Reihenfolge Striche und Punkte gezeichnet werden sollen, wie lang sie jeweils sind und wie groß die Lücken zwischen ihnen sein sollen.
elpStyleEntry	Das benutzerdefinierte Stilarray. Das erste Element des Arrays gibt die Länge des ersten zu zeichnenden Teilstrichs an, das zweite Element die Pause bis zum nächsten Strich. Entsprechend finden sich im dritten Element die Werte zur Länge des zweiten Strichs und im vierten der Abstand zum dritten und so weiter. Wurde das Ende des Arrays erreicht (dessen Größe über *elpNumEntries* definiert ist), beginnt das Auswerten wieder von vorn. Ist der Stiftstil *PS_GEOMETRIC*, werden die Längenangaben in logischen Einheiten interpretiert, sonst in Geräteeinheiten.

Mögliche Werte für EXTLOGPEN::elpBrushStyle

Wert	Beschreibung
BS_DIBPATTERN	Ein Brush, der durch dein DIB-Grafik spezifiziert wird. *elpHatch* enthält ein Handle auf eine gepackte DIB.

Wert	Beschreibung
BS_DIBPATTERNPT	Ein Brush, der durch dein DIB-Grafik spezifiziert wird. *elpHatch* enthält einen Zeiger auf eine gepackte DIB.
BS_HATCHED	Schraffierter Brush
BS_HOLLOW, BS_NULL	Leerer Brush
BS_PATTERN	Brush ist durch eine im Speicher befindliche Grafik definiert.
BS_SOLID	Einfarbiger Brush

Mögliche Werte für EXTLOGPEN::elpColor

Abhängig von *elpBrushStyle* kann *elpColor* verschiedenartige Werte annehmen. Ist *elpBrushStyle* auf *BS_SOLID* oder *BS_HATCHED* gesetzt, enthält *elpColor* einen *COLORREF*-Wert der gewünschten Farbe.

Ist *elpBrushStyle* auf *BS_DIBPATTERN* oder *BS_DIBPATTERNPT* gesetzt, kann *elpColor* einen der beiden folgenden Werte enthalten:

Werte	Beschreibung
DIB_PAL_COLORS	Brush verwendet die derzeit aktive Farbpalette.
DIB_RGB_COLORS	Brush verwendet echte RGB-Werte.

Ist *elpBrushStyle* auf *BS_HOLLOW* oder *BS_PATTERN* gesetzt, wird der *elpColor* Wert ignoriert.

Mögliche Werte für EXTLOGPEN:: elpHatch

Die Bedeutung von *elpHatch* ist abhängig von *elpBrushStyle*.

Ist *elpBrushStyle* auf *BS_DIBPATTERN* gesetzt, enthält *elpHatch* ein Handle auf eine gepackte DIB.

Ist *elpBrushStyle* auf *BS_DIBPATTERNPT* gesetzt, enthält *elpHatch* einen Zeiger auf eine gepackte DIB.

Ist *elpBrushStyle* auf *BS_PATTERN* gesetzt, enthält *elpHatch* ein Handle auf die zugrunde liegende Bitmap.

Ist *elpBrushStyle* auf *BS_SOLID* oder *BS_HOLLOW* gesetzt, wird *elpHatch* ignoriert.

Ist *elpBrushStyle* auf *BS_HATCHED* gesetzt, kann *elpHatch* einen der folgenden Werte annehmen:

Werte	Beschreibung
HS_BDIAGONAL	Brush wird mit Linien schraffiert, die sich im 45-Grad-Winkel von links unten nach rechts oben ziehen.
HS_CROSS	Brush wird mit horizontalen und vertikalen Linien schraffiert, die ein Karomuster bilden.
HS_DIAGCROSS	Brush wird mit diagonal gekreuzten Linien schraffiert.
HS_FDIAGONAL	Brush wird mit Linien schraffiert, die sich im 45-Grad-Winkel von von links oben nach rechts unten ziehen.
HS_HORIZONTAL	Brush wird mit horizontalen Linien schraffiert.
HS_VERTICAL	Brush wird mit vertikalen Linien schraffiert.

CPalette

CPalette stellt eine Windows-Farbpalette dar, die als Schnittstelle zwischen einer Applikation und dem Ausgabemedium dient. So ist es möglich, Farben, die in der Anwendung vorkommen, möglichst originalgetreu auf anderen Geräten abzubilden.

Methode	Beschreibung
CreateHalftonePalette	Erzeugt eine Halbton-Palette für den zugrunde liegenden Gerätekontext.
CreatePalette	Erzeugt eine Windows-Farbpalette.
AnimatePalette	Tauscht Farbwerte innerhalb einer logischen Palette aus.
FromHandle	Nimmt ein Handle auf ein Windows-Palettenobjekt entgegen und gibt einen passenden *CPalette*-Zeiger zurück.
GetNearestPaletteIndex	Nimmt einen *COLORREF*-Farbwert entgegen und gibt den Eintrag der Palette zurück, der am ehesten mit dem gewünschten Farbwert übereinstimmt.
ResizePalette	Verändert die Größe der Farbpalette (d.h. die Anzahl ihrer Einträge).

Methode	Beschreibung
GetEntryCount	Gibt die Anzahl der Einträge der Farbpalette zurück.
GetPaletteEntries	Gibt einen Ausschnitt einer Farbpalette zurück. Der Programmierer muss ein entsprechend großes Array an *PALETTEENTRY*-Strukturen bereithalten (siehe unten).
SetPaletteEntries	Setzt eine Reihe von Einträgen in der Farbpalette. Der Entwickler muss hierfür ein Array von *PALETTEENTRY*-Struktur bereitstellen (siehe unten).

Die PALETTEENTRY-Struktur

Die *PALETTEENTRY*-Stuktur beschreibt gerade einen Eintrag in der Farbtabelle:

```
typedef struct tagPALETTEENTRY {
  BYTE peRed;
  BYTE peGreen;
  BYTE peBlue;
  BYTE peFlags;
} PALETTEENTRY;
```

Methode	Beschreibung
peRed	Rotanteil der Farbe
peGreen	Grünanteil der Farbe
peBlue	Blauanteil der Farbe
peFlags	Gibt an, wie die übrigen Werte der Struktur zu verwenden sind. Ist *peFlags* gleich *PC_EXPLICIT*, handelt es sich bei diesem Eintrag um einen Hardwarepalettenindex. *PC_NOCOLLAPSE* gibt an, dass dieser Eintrag beim Einsortieren in eine bestehende Liste nicht gegen andere Farbwerte verglichen, sondern direkt eingefügt werden soll. Sind keine freien Plätze für den Eintrag vorhanden, wird die Farbe normal gegen die bereits vorhandenen Farben verglichen. *PC_RESERVED* markiert einen Farbeintrag als einen, gegen den Farben nicht verglichen werden sollen, zum Beispiel weil der Eintrag im Rahmen einer Palettenrotation ohnehin ständig verändert wird und somit zu Farbfehlern führen könnte.

CRgn

Ein *CRgn*-Objekt beschreibt einen elliptischen oder polygonalen Bereich innerhalb einer GDI Region.

Methode	Beschreibung
CombineRgn	Fügt zwei *CRgn*-Objekte zu einem zusammen, bildet also die Vereinigung der beiden Bereiche.
CopyRgn	Kopiert die Daten eines *CRgn*-Objekts in ein anderes.
CreateEllipticRgn	Initialisiert ein neues *CRgn*-Objekt mit einer elliptischen Region.
CreateEllipticRgn-Indirect	Initialisiert ein neues *CRgn*-Objekt mit einer elliptischen Region auf Basis einer *RECT*-Struktur.
CreateFromData	Initialisiert ein *CRgn*-Objekt aus einer gegebenen Region und passenden Transformationsdaten.
CreateFromPath	Erzeugt eine Region auf Basis eines Gerätekontextpfads.
CreatePolygonRgn	Erzeugt eine polygonale Region, deren Eckpunkte in einem Array übergeben werden.
CreatePolyPolygonRgn	Erzeugt ein *CRgn*-Objekt bestehend aus mehreren einzelnen Polygonen, deren Eckpunkte in einem Array übergeben werden.
CreateRectRgn	Initialisiert ein *CRgn*-Objekt mit einer rechteckigen Region.
CreateRectRgnIndirect	Initialisiert ein *CRgn*-Objekt mit einer rechteckigen Region, die durch eine *RECT*-Struktur initialisiert wird.
CreateRoundRectRgn	Initialisiert ein *CRgn*-Objekt mit einer rechteckigen Region, deren Ecken abgerundet sind.
EqualRgn	Vergleicht zwei *CRgn*-Objekte und gibt zurück, ob die von ihnen beschriebenen Regionen identisch sind.
FromHandle	Nimmt ein Handle auf eine Windows-Region an und gibt einen Zeiger auf ein *CRgn*-Objekt zurück.

Methode	Beschreibung
GetRegionData	Liest die zu einer Region gehörenden Daten aus.
GetRgnBox	Gibt die Boundingbox der Region eines *CRgn*-Objekts zurück.
OffsetRgn	Verschiebt eine Region um einen spezifizierten Offset.
PtInRegion	Prüft, ob sich ein Punkt in der *CRgn*-Region befindet.
RectInRegion	Prüft, ob sich ein Rechteck in der Region befindet.
SetRectRgn	Setzt die Region, die ein *CRgn*-Objekt beschreibt auf den übergebenen rechteckigen Bereich.

Standarddialoge

Sie haben im Verlauf des Buchs bereits mit einigen Standarddialogen gearbeitet. Die folgenden Seiten fassen die wichtigsten Informationen dieser kleinen praktischen Klassen zusammen.

Es gibt sechs verschiedene Standarddialogklassen:

- *CColorDialog* – Eine Farbtabelle wird präsentiert, aus der eine Farbe ausgewählt werden kann.
- *CFileDialog* – Ein Dialogfeld zum Laden oder Speichern von Dateien.
- *CFindReplaceDialog* – Ein *Suchen/Ersetzen*-Dialogfeld.
- *CFontDialog* – Ein Dialogfeld zur Auswahl einer Schrift.
- *CPageSetupDialog* – Dialogfeld zum Einrichten von Seiten.
- *CPrintDialog* – Dialogfeld zur Druckereinrichtung.

Die wichtigsten Methoden dieser Klassen stehen in der nachfolgenden Tabellen.

CColorDialog

Methode	Beschreibung
DoModal	Stellt den Dialog modal dar.

Methode	Beschreibung
GetColor	Gibt eine *COLORREF*-Struktur zurück, welche die Farbwerte der ausgewählten Farbe enthält.
GetSavedCustomColors	Entnimmt die vom Benutzer angelegten, frei definierten Farben.
SetCurrentColor	Setzt die aktuelle Farbe in der Auswahl auf einen vorgegebenen Wert.
OnColorOK	(Kann überschrieben werden) Prüft die ausgewählte Farbe auf Gültigkeit.

CFileDialog

Methode	Beschreibung
DoModal	Stellt den Dialog modal dar.
GetPathName	Gibt den Pfadnamen der ausgewählten Datei zurück.
GetFileName	Gibt den Dateinamen der ausgewählten Datei zurück.
GetFileExt	Gibt das Suffix der ausgewählten Datei zurück.
GetFileTitle	Gibt den Namen der ausgewählten Datei zurück.
GetNextPathName	Gibt den Pfad der nächsten ausgewählten Datei zurück.
GetReadOnlyPref	Gibt den Status des Schreibgeschützt-Flags der ausgewählten Datei zurück.
GetStartPosition	Gibt die Position der ersten ausgewählten Datei in der Dateinamenliste zurück.
OnShareViolation	Kann überschrieben werden. Wird aufgerufen, wenn die Datei bereits benutzt wird.
OnFileNameOK	Kann überschrieben werden. Prüft den ausgewählten Dateinamen auf Gültigkeit.
OnLBSelChangedNotify	Kann überschrieben werden. Wird aufgerufen, wenn sich die Auswahl in der Dateinamen-Listbox ändert.

Methode	Beschreibung
OnInitDone	Kann überschrieben werden. Behandlungsmethode für *WM_NOTIFY CDC_INITDONE*-Nachrichten.
OnFileNameChange	Kann überschrieben werden. Behandlungsmethode für *WM_NOTIFY CDC_SELCHANGE*-Nachrichten.
OnFolderChange	Kann überschrieben werden. Behandlungsmethode für *WM_NOTIIFY CDC_FOLDERCHANGE*-Nachrichten.
OnTypeChange	Kann überschrieben werden. Behandlungsmethode für *WM_NOTIFY CDC_TYPECHANGE*-Nachrichten.

CFindReplaceDialog

Methode	Beschreibung
Create	Erzeugt ein *CFindReplaceDialog*-Objekt und stellt es dar.
FindNext	Gibt an, ob der Benutzer nach dem nächsten Auftreten einer Zeichenkette suchen möchte.
GetFindString	Gibt den eingegebenen Suchstring zurück.
GetReplaceString	Gibt den eingegebenen Ersetzungsstring zurück.
IsTerminating	Gibt an, ob die Dialogbox geschlossen wird.
MatchCase	Gibt an, ob auf der Benutzer wünscht, dass Groß- und Kleinschreibung geachtet werden soll.
MatchWholeWord	Gibt an, ob der Benutzer wünscht, dass nur übereinstimmende ganze Worte gefunden werden.
ReplaceAll	Gibt an, ob der Benutzer wünscht, dass jedes Auftreten des Suchstrings ersetzt wird.
ReplaceCurrent	Gibt an, ob der Benutzer wünscht, dass das aktuell gefundene Wort ersetzt wird.

Methode	Beschreibung
SearchDown	Gibt an, ob der Benutzer die Datei von oben nach unten durchsuchen möchte.

CFontDialog

Methode	Beschreibung
DoModal	Stellt den Dialog modal dar.
GetCurrentFont	Gibt den Namen des aktuell selektierten Zeichensatzes zurück.
GetFaceName	Gibt den Facename des aktuell selektierten Zeichensatzes zurück.
GetStyleName	Gibt den Stilnamen des aktuell selektierten Zeichensatzes zurück.
GetSize	Gibt die Größe (in Punkt) des aktuell selektierten Zeichensatzes zurück.
GetColor	Gibt die Farbe des aktuell selektierten Zeichensatzes zurück
GetWeight	Gibt die Gewichtung des aktuell selektierten Zeichensatzes zurück.
IsStrikeOut	Gibt an, ob der Zeichensatz durchgestrichen dargestellt werden soll.
IsUnderline	Gibt an, ob der Zeichensatz unterstrichen dargestellt werden soll.
IsBold	Gibt an, ob der Zeichensatz fett gedruckt dargestellt werden soll.
IsItallic	Gibt an, ob der Zeichensatz kursiv gedruckt dargestellt werden soll.

CPageSetupDialog

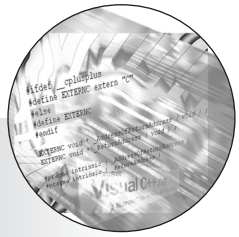
Methode	Beschreibung
DoModal	Stellt den Dialog modal dar.

Methode	Beschreibung
CreatePrinterDC	Erzeugt einen Gerätekontext für den Drucker.
GetDeviceName	Gibt den Gerätenamen des Druckers zurück.
GetDevMode	Gibt den *DEVMODE* des Druckers zurück.
GetDriveName	Gibt den vom Drucker verwendeten Treiber zurück.
GetMargins	Gibt die Seitenrandeinstellungen des Drucks zurück.
GetPortName	Gibt den Ausgabeport zurück.
GetPaperSize	Gibt die Papiergröße des Druckers zurück.
OnDrawPage	Kann überschrieben werden. Wird vom Anwendungsgerüst aufgerufen, um eine auszudruckende Seite auf dem Bildschirm darzustellen.
PreDrawPage	Kann überschrieben werden. Wird vom Anwendungsgerüst vor *OnDrawPage* aufgerufen.

CPrintDialog

Methode	Beschreibung
DoModal	Stellt den Dialog modal dar.
CreatePrinterDC	Erzeugt einen Gerätekontext für den Drucker ohne den *Drucken*-Dialog anzuzeigen.
GetCopies	Gibt die Anzahl der gewünschten Kopien zurück.
GetDefaults	Gibt die Gerätestandardeinstellungen zurück, ohne einen Dialog einzublenden.
GetDeviceName	Gibt den Namen des aktuell ausgewählten Druckers zurück.
GetDevMode	Gibt die *DEVMODE*-Struktur zurück.
GetDriverName	Gibt den Namen des aktuell gewählten Druckertreibers zurück.
GetFromPage	Gibt die gewünschte Startseite zurück.
GetToPage	Gibt die gewünschte Endseite zurück.

Methode	Beschreibung
GetPortName	Gibt den Name des aktuell ausgewählten Druckeranschlusses zurück.
GetPrinterDC	Gibt einen Handle auf den Drucker-Gerätekontext zurück.
PrintAll	Gibt an, ob alle Seiten gedruckt werden sollen.
PrintRange	Gibt an, ob nur ein bestimmter Seitenbereich gedruckt werden soll.
PrintSelection	Gibt an, ob nur ein markierter Bereich gedruckt werden soll.



Alphabetische Übersicht der im Buch verwendeten MFC-Funktionen

C

AfxGetApp

Gibt einen Zeiger auf das globale *CwinApp*-Objekt zurück.

```
CWinApp* AFXAPI AfxGetApp( );
```

Parameter	Beschreibung
Rückgabewert	Zeiger auf das globale *CWinApp*-Objekt.

CommandToIndex [CStatusBar, CToolBar, CToolbarCtrl]

Liefert den Index zu einer übergebenen Steuerelement-ID zurück. Wird immer dann verwendet, wenn zwar die ID, nicht aber die Position (zum Beispiel auf der Statusleiste) eines Elements bekannt ist.

```
int CommandToIndex( UINT nIDFind ) const;
```

Parameter	Beschreibung
nIDFind	ID der gewünschten Komponente.
Rückgabewert	Lineare Position (bei 0 beginnend) des Elements, oder -1, wenn das Element nicht gefunden werden konnte.

Create [CFrameWnd]

Erzeugt ein Objekt der *CFrameWnd*-Klasse.

```
BOOL Create(
LPCTSTR lpszClassName,
         LPCTSTR lpszWindowName,
         DWORD dwStyle =
WS_OVERLAPPEDWINDOW,
         const RECT& rect = rectDefault,
         CWnd* pParentWnd = NULL,
         LPCTSTR lpszMenuName = NULL,
         DWORD dwExStyle = 0,
         CCreateContext* pContext = NULL
);
```

Parameter	Beschreibung
lpszClassName	Name der Fensterklasse. NULL sorgt für das Verwenden der Standardattribute von *CFrameWnd*.
lpszWindowName	Name des Fensters, der in der Titelzeile erscheint.

Parameter	Beschreibung
dwStyle	Fensterstil. *FWS_ADDTOTITLE* mit angeben, wenn der Name des aktuellen Dokuments in der Titelzeile auftauchen soll.
rect	Größe und Position des Fensters.
pParentWnd	Gibt das Elternfenster dieses Fensters an, NULL für Hauptrahmenfenster.
lpszMenuName	Name der zu verwendenden Menü-Ressource. Kann auf NULL gesetzt werden.
dwExStyle	Erweiterte Fensterattribute.
pContext	Zeiger auf eine *CCreateContext*-Struktur. Kann auf NULL gesetzt werden.

Create [CStatusBar]

Erzeugt ein neues Statuszeilenobjekt.

```
virtual BOOL Create(
   CWnd* pParentWnd,
   DWORD dwStyle = WS_CHILD | WS_VISIBLE |
CBRS_BOTTOM,
   UINT nID = AFX_IDW_STATUS_BAR
);
```

Parameter	Beschreibung
pParentWnd	Zeiger auf das Elternfenster.
dwStyle	Stil der Statuszeile. Zusätzlich zu den Standard-Fensterflags sind folgende Stile möglich: *CBRS_TOP*: Leiste ist am oberen Bildrand, *CBRS_BOTTOM*: Leiste ist am unteren Bildrand, *CBRS_NOALIGN*: Leiste wird nicht verschoben, wenn sich die Elternfenstergröße ändert.
nID	ID des Fensters, normalerweise *AFX_IDW_STATUS_BAR*.
Rückgabewert	Ein Wert ungleich 0 bei Erfolg, sonst 0.

CreateEllipticRgnIndirect [CRgn]

Erzeugt eine elliptische Region, die durch eine übergebene Rechteckstruktur verwendet wird. Durch weitere Methoden wie *PtInRegion* kann dann beispielsweise geprüft werden, ob sich ein Punkt innerhalb dieser Region befindet.

```
BOOL CreateEllipticRgnIndirect( LPCRECT lpRect );
```

Parameter	Beschreibung
lpRect	Zeiger auf eine Rechteckstruktur, die das die Ellipse umgebende Rechteck repräsentiert.
Rückgabewert	1, falls die Methode erfolgreich war, sonst 0.

CreateEx [CToolBar]

Erzeugt eine neue Werkzeugleiste.

```
BOOL CreateEx(
   CWnd* pParentWnd,
   DWORD dwCtrlStyle = TBSTYLE_FLAT,
   DWORD dwStyle = WS_CHILD | WS_VISIBLE |
 CBRS_ALIGN_TOP,
   CRect rcBorders = CRect( 0, 0, 0, 0 ),
   UINT nID = AFX_IDW_TOOLBAR
);
```

Parameter	Beschreibung
pParentWnd	Zeiger auf Elternfenster.
dwCtrlStyle	Für eine komplette Liste siehe *Toolbar Control And Button Styles* in der Online-Hilfe.
dwStyle	Für eine komplette Liste siehe *Toolbar Control And Button Styles* in der Online-Hilfe.
rcBorders	Gibt die Größe der Ränder der Werkzeugleiste an. (0, 0, 0, 0) erzeugt eine Toolbar ohne Ränder.
nID	ID der Toolbar. Normalerweise *AFX_IDW_TOOLBAR*.
Rückgabewert	Ein Wert ungleich 0 bei Erfolg, sonst 0.

CreateFontIndirect [CFont]

Initialisiert ein *CFont*-Objekt mit den durch eine *LOGFONT*-Struktur übergebenen Charakteristika.

```
BOOL CreateFontIndirect(const LOGFONT* lpLogFont );
```

Parameter	Beschreibung
lpLogFont	Zeiger auf eine *LOGFONT*-Struktur, die die Charakteristika des zu erzeugenden Zeichensatzes enthält.
Rückgabewert	1, falls die Methode erfolgreich war, sonst 0.

CreateNewFrame [CDocTemplate]

Erzeugt ein neues Fenster, verknüpft mit einem anderen Fenster und einem Dokument.

```
CFrameWnd* CreateNewFrame(
   CDocument* pDoc,
   CFrameWnd* pOther
);
```

Parameter	Beschreibung
pDoc	Dokument, mit dem das neue Fenster verknüpft sein soll.
pOther	Fenster, auf dem das neue Fenster basieren soll. Es wird im Titel durch eine vorangestellte Nummer gekennzeichnet.
Rückgabewert	Zeiger auf das neu erzeugte Fenster, oder NULL bei Auftreten eines Fehlers.

DPtoLP

Rechnet Gerätekoordinaten in logische Koordinaten um.

```
void DPtoLP(
   LPPOINT lpPoints,
   int nCount = 1 ) const;
void DPtoLP(
   LPRECT lpRect ) const;
void DPtoLP(
   LPSIZE lpSize ) const;
```

Parameter	Beschreibung
lpPoints	Zeiger auf ein Array von umzuwandelnden Punkten.
nCount	Anzahl der Punkt im Array.
lpRect	Zeiger auf ein umzuwandelndes Rechteck.
lpSize	Zeiger auf ein umzuwandelndes Größenobjekt (*SIZE* oder *CSize*).

DrawText [CDC]

Druckt einen Text im Gerätekontext aus.

```
virtual int DrawText(
LPCTSTR lpszString,
         int nCount,
         LPRECT lpRect,
         UINT nFormat );
```

Parameter	Beschreibung
lpszString	Der auszugebende Text.
nCount	Die Anzahl der auszugebenden Zeichen. Ist dieser Wert -1, wird die von *lpszString* automatisch berechnet. *lpszString* muss dann eine null-terminierte Zeichenkette sein.
lpRect	Gibt den rechteckigen Bereich in logischen Koordinaten an, in dem der Text gezeichnet werden soll.
nFormat	Gibt die Art an, in welcher der Text formatiert werden soll. Siehe unten.
Rückgabewert	Die Höhe des Texts.

Mögliche Werte für *nFormat*:

Flag	Beschreibung
DT_BOTTOM	Text wird nach unten ausgerichtet. Muss zusammen mit *DT_SINGLELINE* gesetzt sein.
DT_CALCRECT	Bestimmt die Breite und Höhe des zu verwendenden Rechtecks, ohne den Text zu zeichnen.

Flag	Beschreibung
DT_CENTER	Text wird horizontal zentriert.
DT_EXPANDTABS	Erweitert TAB-Zeichen. Standardeinstellung hierfür ist 8.
DT_LEFT	Richtet den Text nach links aus.
DT_NOPREFIX	Ignoriert Präfix-Symbole wie & (unterstreicht das folgende Zeichen) und && (kaufmännisches UND).
DT_RIGHT	Richtet den Text nach rechts aus.
DT_SINGLELINE	Zeilenumbrüche innerhalb des übergebenen Texts werden ignoriert.
DT_TABSTOP	Setzt die Anzahl von Leerzeichen pro TAB-Zeichen. Dieser Wert steht im oberen Byte von nFormat. Standardeinstellung ist 8.
DT_TOP	Richtet den Text nach oben aus.
DT_VCENTER	Zentriert den Text vertikal (nur in Verbindung mit *DT_SINGLELINE*).
DT_WORDBREAK	Schaltet Zeilenumbruch nach Worten ein, die nicht mehr in dieselbe Zeile passen würden.

Delete [CRecordset]

Löscht einen Datensatz aus einem Recordset

```
virtual void Delete( );
```

DockControlBar [CFrameWnd]

Dockt eine Toolbar an einer Fensterseite an.

```
void DockControlBar(
   CControlBar* pBar,
   UINT nDockBarID = 0,
   LPCRECT lpRect = NULL
);
```

Parameter	Beschreibung
pBar	Zeiger auf die anzudockende Toolbar

Parameter	Beschreibung
nDockBarID	Gibt an, an welche Seite die Toolbar angedockt werden soll. Mögliche Werte: *AFX_IDW_DOCKBAR_TOP*: an den oberen Rand andocken, *AFX_IDW_DOCKBAR_BOTTOM*: an den unteren Rand andocken, *AFX_IDW_DOCKBAR_LEFT*: an den linken Rand andocken, *AFX_IDW_DOCKBAR_RIGHT*: an den rechten Rand andocken, *0*: gleichgültig, welcher Rand verwendet wird.
lpRect	Gibt die Position im Nicht-Client-Bereich an, an dem die Toolbar angedockt werden soll, oder NULL, falls das egal ist.

Ellipse [CDC]

Zeichnet eine Ellipse im Gerätekontext.

```
BOOL Ellipse( int x1,
         int y1,
         int x2,
         int y2 );
```

Parameter	Beschreibung
x1	X-Koordinate der linken oberen Ecke des die Ellipse umgebenden Rechtecks.
y1	Y-Koordinate der linken oberen Ecke des die Ellipse umgebenden Rechtecks.
x2	X-Koordinate der rechten unteren Ecke des die Ellipse umgebenden Rechtecks.
y2	Y-Koordinate der rechten unteren Ecke des die Ellipse umgebenden Rechtecks.
Rückgabewert	1, falls die Methode erfolgreich war, sonst 0.

EnableDocking [CFrameWnd]

Aktiviert andockbare Toolbars innerhalb eines Hauptfensters.

```
void EnableDocking( DWORD dwDockStyle );
```

Parameter	Beschreibung
dwDockStyle	Mögliche Stilflags: *CBRS_ALIGN_TOP*: erlaubt Andocken am oberen Fensterrand, *CBRS_ALIGN_BOTTOM*: erlaubt Andocken am unteren Fensterrand, *CBRS_ALIGN_LEFT*: erlaubt Andocken am linken Fensterrand, *CBRS_ALIGN_RIGHT*: erlaubt Andocken am rechten Fensterrand, *CBRS_ALIGN_ANY*: erlaubt Andocken an jedem Fensterrand.

GetClientRect [CWnd]

Kopiert die Rechteckstruktur der Client Area des Fensters (insbesondere somit auch dessen Größe) in ein als Parameter übergebenes Rechteck.

```
void GetClientRect( LPRECT lpRect ) const;
```

Parameter	Beschreibung
lpRect	Zeiger auf eine Rechteckstruktur, die die Reckteckstruktur der Client Area aufnehmen soll.

GetRValue [Makro]

Liest den Rotanteil aus einer *COLORREF*-Struktur aus.

```
GetRValue(COLORREF Color);
```

Parameter	Beschreibung
Color	*COLORREF*-Struktur, aus der der Farbanteil extrahiert werden soll.
Rückgabewert	Rotanteil der Farbe

GetGValue [Makro]

Liest den Grünanteil aus einer *COLORREF*-Struktur aus.

```
GetGValue(COLORREF Color);
```

Parameter	Beschreibung
Color	*COLORREF*-Struktur, aus der der Farbanteil extrahiert werden soll.

Parameter	Beschreibung
Rückgabewert	Grünanteil der Farbe

GetBValue [Makro]

Liest den Blauanteil aus einer *COLORREF*-Struktur aus.

```
GetBValue(COLORREF Color);
```

Parameter	Beschreibung
Color	*COLORREF*-Struktur, aus der der Farbanteil extrahiert werden soll.
Rückgabewert	Blauanteil der Farbe

GetTrueRect [CRectTracker]

Gibt die tatsächliche Auswahl eines *CRectTracker*-Objekts zurück.

```
void GetTrueRect( LPRECT lpTrueRect ) const;
```

Parameter	Beschreibung
lpTrueRect	Zeiger auf eine Rechteckstruktur, die die Auswahlkoordinaten des *CRectTracker*-Objekts erhält.

InsertColumn [CListCtrl]

Fügt eine Spalte in eine Liste ein.

```
int InsertColumn(
   int nCol,
   LPCTSTR lpszColumnHeading,
   int nFormat = LVCFMT_LEFT,
   int nWidth = -1,
   int nSubItem = -1
);
```

Parameter	Beschreibung
nCol	Index (bei 0 beginnend) der neuen Spalte

Parameter	Beschreibung
lpszColumn-Heading	Überschrift der Spalte
nFormat	Ausrichtung der Spalte. Gültige Werte sind *LVCFMT_LEFT* für linksbündige Ausgabe, *LVCFMT_CENTER* für zentrierte Ausgabe und *LVCFMT_RIGHT* für rechtsbündige Ausgabe.
nWidth	Breite der Spalte in Pixeln
nSubItem	Index des Unterpunkts innerhalb dieser Spalte
Rückgabewert	Index der neuen Spalte oder -1 beim Auftreten eines Fehlers

InvalidateRect [CWnd]

Erklärt einen rechteckigen Bereich in der Client Area für ungültig und sorgt somit für ein Neuzeichnen desselbigen.

```
void InvalidateRect(
LPCRECT lpRect,
         BOOL bErase = TRUE );
```

Parameter	Beschreibung
lpRect	Zeiger auf die Rechteckstruktur, die die Koordinaten des als ungültig zu erklärenden Rechtecks enthält. NULL erklärt die gesamte Client Area für ungültig.
bErase	Gibt an, ob der Bereich der ungültigen Rechtecks vor dem Neuzeichnen gelöscht werden soll.

LineTo [CDC]

Zeichnet eine Linie von der aktuellen Stiftposition zu den angegebenen Koordinaten unter Verwendung des aktuell eingestellten Stifts.

```
BOOL LineTo( int x,
         int y );
```

Parameter	Beschreibung
x	X-Koordinate des Zielpunkts

Parameter	Beschreibung
y	Y-Koordinate des Zielpunkts
Rückgabewert	1, falls die Methode erfolgreich war, sonst 0

LPtoDP [CDC]

Wandelt logische Einheiten in Geräteeinheiten um. Es existieren drei Prototypen für diese Methode:

```
void LPtoDP(
LPPOINT lpPoints,
      int nCount = 1 ) const;

void LPtoDP( LPRECT lpRect ) const;

void LPtoDP( LPSIZE lpSize ) const;
```

Parameter	Beschreibung
lpPoints	Zeiger auf ein Array von umzuwandelnden Punkten
nCount	Anzahl der Punkt im Array
lpRect	Zeiger auf ein umzuwandelndes Rechteck
lpSize	Zeiger auf ein umzuwandelndes Größenobjekt (*SIZE* oder *CSize*)

MessageBox [CWnd]

Öffnet eine Messagebox, die ein Icon und einen Text enthalten kann.

```
int MessageBox( LPCTSTR lpszText,
         LPCTSTR lpszCaption = NULL,
         UINT nType = MB_OK );
```

Parameter	Beschreibung
lpszText	Text, der in der Messagebox ausgegeben werden soll.
lpszCaption	Text, der in der Titelzeile der Messagebox ausgegeben werden soll. NULL sorgt für eine Ausgabe des Texts *ERROR*.

Parameter	Beschreibung
nType	Gibt den Typ der Box an, was sich in erster Linie auf das verwendete Icon bezieht (siehe nachfolgende Tabelle).
Rückgabewert	0 signalisiert einen Fehler.

Die folgenden Werte sind für nType zulässig:

Parameter	Beschreibung
MB_ICONHAND, MB_ICONSTOP, MB_ICONERROR	Ein Hand-Symbol
MB_ICONQUESTION	Ein Fragezeichen-Symbol
MB_ICONEXCLAMATION, MB_ICONWARNING	Ein Ausrufezeichen-Symbol
MB_ICONASTERISK, MB_ICONINFORMATION	Ein Informations-Symbol (für gewöhnlich ein 'i')

MoveFirst [CRecordset]

Setzt den ersten Eintrag eines Datensatzsets als aktuellen Datensatz.

```
void MoveFirst( );
```

MoveLast [CRecordset]

Setzt den letzten Eintrag eines Datensatzsets als aktuellen Datensatz.

```
void MoveLast( );
```

MoveNext [CRecordset]

Bewegt sich im Datensatzset um einen Satz vorwärts.

```
void MoveNext( );
```

MovePrev [CRecordset]

Bewegt sich im Datensatzset um einen Satz zurück.

```
void MovePrev( );
```

MoveTo [CDC]

Setzt die aktuelle Position des Stifts auf die übergebenen Parameterangaben.

```
CPoint MoveTo( int x,
         int y );
```

Parameter	Beschreibung
x	X-Koordinate des Zielpunkts
y	Y-Koordinate des Zielpunkts
Rückgabewert	Der Punkt, der vor der *MoveTo*-Operation gesetzt war

OnBeginPrinting [CView, Behandlungsmethode]

Wird vom Framework direkt nach *OnPreparePrinting* aufgerufen, bevor ein Druckvorgang oder eine Seitenansicht gestartet wird.

```
virtual void OnBeginPrinting(
CDC* pDC,
   CPrintInfo* pInfo
);
```

Parameter	Beschreibung
pDC	Zeiger auf den Druckkontext
pInfo	Zeiger auf eine *CPrintInfo*-Struktur

OnDraw [CView, Behandlungsmethode]

Führt die zu einer Ansichtsklasse gehörenden Zeichenoperationen auf Basis eines Dokuments durch.

```
void OnDraw( CDC* pDC ) ;
```

Parameter	Beschreibung
pDC	Zeiger auf den Gerätekontext, der für die Zeichenoperationen verwendet werden soll

OnEndPrinting [CView, Behandlungsmethode]

Wird vom Framework direkt nach einem Druckvorgang oder einer Seitenvoransicht aufgerufen.

```
virtual void OnEndPrinting(
CDC* pDC,
   CPrintInfo* pInfo
);
```

Parameter	Beschreibung
pDC	Zeiger auf den Druckkontext
pInfo	Zeiger auf eine *CPrintInfo*-Struktur

OnLButtonDown [CWnd, Behandlungsmethode]

Wird ausgeführt, wenn die linke Maustaste über einem Fenster gedrückt wurde.

```
afx_msg void OnLButtonDown(
   UINT nFlags,
   CPoint point
);
```

Parameter	Beschreibung
nFlags	Gibt an, ob das Mausereignis von zusätzlichen Tastendrücken begleitet war. Mögliche Flags: *MK_CONTROL* für die Strg-Taste, *MK_LBUTTON* für die linke Maustaste, *MK_MBUTTON* für die mittlere Maustaste, *MK_RBUTTON* für die rechte Maustaste und *MK_SHIFT* für die ⇧-Taste.
point	*Cpoint*-Struktur, die die aktuelle Position des Mauszeigers enthält.

OnLButtonUp [CWnd, Behandlungsmethode]

Wird ausgeführt, wenn die linke Maustaste über einem Fenster losgelassen wurde.

```
afx_msg void OnLButtonUp(
   UINT nFlags,
   CPoint point
);
```

Parameter	Beschreibung
nFlags	Gibt an, ob das Mausereignis von zusätzlichen Tastendrücken begleitet war. Mögliche Flags: *MK_CONTROL* für die Strg-Taste, *MK_LBUTTON* für die linke Maustaste, *MK_MBUTTON* für die mittlere Maustaste, *MK_RBUTTON* für die rechte Maustaste und *MK_SHIFT* für die ⇧-Taste.
point	*CPoint*-Struktur, welche die aktuelle Position des Mauszeigers enthält.

OnMouseMove [CWnd, Behandlungsmethode]

Wird ausgeführt, wenn eine Mausbewegung über einem Fenster stattgefunden hat.

```
afx_msg void OnMouseMove(
   UINT nFlags,
   CPoint point
);
```

Parameter	Beschreibung
nFlags	Gibt an, ob das Mausereignis von zusätzlichen Tastendrücken begleitet war. Mögliche Flags: *MK_CONTROL* für die Strg-Taste, *MK_LBUTTON* für die linke Maustaste, *MK_MBUTTON* für die mittlere Maustaste, *MK_RBUTTON* für die rechte Maustaste und *MK_SHIFT* für die ⇧-Taste.
point	*CPoint*-Struktur, welche die aktuelle Position des Mauszeigers enthält.

OnNewDocument [CDocument, Behandlungsmethode]

Wird aufgerufen, sobald ein neues Dokument erzeugt werden soll.

```
virtual BOOL OnNewDocument( );
```

Parameter	Beschreibung
Rückgabewert	Ein Wert ungleich 0, wenn das Dokument erfolgreich initialisiert wurde, sonst 0.

OnPrepareDC [CView, Behandlungsmethode]

Wird vom Framework vor Aufrufen von OnDraw bzw. OnPrint ausgeführt und erlaubt es, den zu verwendenden Gerätekontext zweckmäßig vorzubereiten.

```
void OnPrepareDC(CDC* pDC,
CPrintInfo* pInfo = NULL );
```

Parameter	Beschreibung
pDC	Zeiger auf den Gerätekontext, der für die Zeichenoperationen verwendet werden soll.
pInfo	Zeiger auf eine *CPrintInfo*-Struktur, falls ein Druckvorgang eingeleitet wird. Siehe Online-Hilfe für weitere Informationen über die einzelnen Strukturkomponenten.

OnPreparePrinting [CView, Behandlungsmethode]

Wird vom Framework aufgerufen, bevor ein Druckvorgang oder eine Seitenansicht aufgerufen wird.

```
virtual BOOL OnPreparePrinting( CPrintInfo* pInfo );
```

Parameter	Beschreibung
pInfo	Zeiger auf eine *CPrintInfo*-Struktur
Rückgabewert	Ein Wert ungleich 0, um den Druckvorgang, bzw. Seitenansichtsaufbau, zu starten, 0, um den Vorgang abzubrechen.

OnRButtonDown [CWnd, Behandlungsmethode]

Wird ausgeführt, wenn die rechte Maustaste über einem Fenster gedrückt wurde.

```
afx_msg void OnRButtonDown(
   UINT nFlags,
   CPoint point
);
```

Parameter	Beschreibung
nFlags	Gibt an, ob das Mausereignis von zusätzlichen Tastendrücken begleitet war. Mögliche Flags: *MK_CONTROL* für die Strg-Taste, *MK_LBUTTON* für die linke Maustaste, *MK_MBUTTON* für die mittlere Maustaste, *MK_RBUTTON* für die rechte Maustaste und *MK_SHIFT* für die ⇧-Taste..
point	*CPoint*-Struktur, die die aktuelle Position des Mauszeigers enthält.

OnRButtonUp [CWnd, Behandlungsmethode]

Wird ausgeführt, wenn die rechte Maustaste über einem Fenster losgelassen wurde.

```
afx_msg void OnRButtonUp(
   UINT nFlags,
   CPoint point
);
```

Parameter	Beschreibung
nFlags	Gibt an, ob das Mausereignis von zusätzlichen Tastendrücken begleitet war. Mögliche Flags: *MK_CONTROL* für die Strg-Taste, *MK_LBUTTON* für die linke Maustaste, *MK_MBUTTON* für die mittlere Maustaste, *MK_RBUTTON* für die rechte Maustaste und *MK_SHIFT* für die ⇧-Taste.
point	*CPoint*-Struktur, welche die aktuelle Position des Mauszeigers enthält.

OnInitDialog [CDialog]

Enthält Initialisierungen für ein Dialogfeld.

```
virtual  BOOL OnInitDialog( );
```

Parameter	Beschreibung
Rückgabewert	Wird ein Wert ungleich 0 zurückgegeben, setzt das Framework den Fokus automatisch auf das erste Element des Dialogfelds, ansonsten wird der Fokus nicht selbstständig verändert/gesetzt.

OnPrint [CView]

Wird vom Framework aufgerufen, um eine Seite zu drucken oder eine Druckvoransicht zu erzeugen.

```
void OnPrint(
   CDC* pDC,
   CPrintInfo* pInfo
);
```

PolyBezier [CDC]

Zeichnet eine oder mehrere Bezierkurven im Gerätekontext.

```
BOOL PolyBezier(
   const POINT* lpPoints,
   int nCount
);
```

Parameter	Beschreibung
lpPoints	Zeiger auf ein Array von *POINT*-Strukturen, welche die einzelnen Koordinaten der Kontroll- und Knotenpunkte enthalten. Die Reihenfolge dabei ist jeweils: Startpunkt, Kontrollpunk1, Kontrollpunkt2, Endpunkt. Bei mehr als einer Kurve, dient der Endpunkt einer Kurve gleichzeitig als Startpunkt der nächsten Kurve, sodass für n Kurven ((2 * n) + 1) *POINT*-Strukturen benötigt werden.
nCount	Anzahl der *POINT*-Strukturen ab Adresse *lpPoints*
Rückgabewert	Wert ungleich 0 bei Erfolg, sonst 1

PreCreateWindow [CWnd]

Wird vom Anwendungsgerüst vor der tatsächlichen Erzeugung eines Fensters aufgerufen.

```
virtual BOOL PreCreateWindow(
   CREATESTRUCT& cs
);
```

Parameter	Beschreibung
cs	Ein *CREATESTRUCT*-Objekt. Siehe Online-Hilfe für Details.

Parameter	Beschreibung
Rückgabewert	Ein Wert ungleich 0, wenn die Fenstererzeugung fortgesetzt werden soll, sonst 0.

PtInRegion [CRgn]

Prüft, ob sich ein Punkt innerhalb der Region befindet.

```
BOOL PtInRegion( int x,
         int y ) const;
```

Parameter	Beschreibung
x	X-Koordinate des Punkts
y	Y-Koordinate des Punkts
Rückgabewert	1, wenn sich der Punkt innerhalb der Region befindet, sonst 0

Rectangle [CDC]

Zeichnet ein Rechteck in das Gerätekontext-Objekt.

```
BOOL Rectangle(
   int x1,
   int y1,
   int x2,
   int y2
);
```

Parameter	Beschreibung
x1	X-Koordinate der linken oberen Ecke des zu zeichnenden Rechtecks in logischen Koordinaten.
y1	Y-Koordinate der linken oberen Ecke des zu zeichnenden Rechtecks in logischen Koordinaten.
x2	X-Koordinate der rechten unteren Ecke des zu zeichnenden Rechtecks in logischen Koordinaten.
y2	Y-Koordinate der rechten unteren Ecke des zu zeichnenden Rechtecks in logischen Koordinaten.
Rückgabewert	Wert ungleich 0 bei Erfolg, sonst 0

ReleaseCapture

Gibt die Maus wieder frei, nachdem sie mit *SetCapture* fixiert wurde.

```
BOOL ReleaseCapture ( );
```

Parameter	Beschreibung
Rückgabewert	*true* bei Erfolg, sonst *false*

SelectObject [CDC]

Wählt ein Objekt (Stift, Pinsel, Bitmap, Region oder Zeichensatz) in den Gerätekontext. Zu diesem Zweck gibt es eine Reihe von verschiedenen Funktionsprototypen. Beachten Sie, dass am Ende von Zeichenoperationen die ursprünglich eingestellten Objekte wieder eingerichtet werden sollten.

```
CPen* SelectObject( CPen* pPen );

CBrush* SelectObject( CBrush* pBrush );

virtual CFont* SelectObject( CFont* pFont );

CBitmap* SelectObject( CBitmap* pBitmap );

int SelectObject( CRgn* pRgn );
```

Parameter	Beschreibung
pPen	Zeiger auf das auszuwählende *CPen*-Objekt
pBrush	Zeiger auf das auszuwählende *CBrush*-Objekt
pFont	Zeiger auf das auszuwählende *CFont*-Objekt
pBitmap	Zeiger auf das auszuwählende *CBitmap*-Objekt
pRgn	Zeiger auf das auszuwählende *CRgn*-Objekt
Rückgabewert	Zeiger auf das alte, jetzt deselektierte Objekt

SelectStockObject [CDC]

Wählt ein Standardobjekt, das nicht erst erzeugt werden muss, in den Gerätekontext.

```
CGdiObject* SelectStockObject( int nIndex );
```

Parameter	Beschreibung
BLACK_BRUSH	Ein schwarzer Pinsel
DKGRAY_BRUSH	Ein dunkelgrauer Pinsel
GRAY_BRUSH	Ein grauer Pinsel
HOLLOW_BRUSH	Ein hohler Pinsel
LTGRAY_BRUSH	Ein hellgrauer Pinsel
NULL_BRUSH	NULL-Pinsel
WHITE_BRUSH	Ein weißer Pinsel
BLACK_PEN	Ein schwarzer Stift
NULL_PEN	NULL-Stift
WHITE_PEN	Ein weißer Stift
ANSI_FIXED_FONT	Fester ANSI-System-Zeichensatz
ANSI_VAR_FONT	Variabler ANSI System-Zeichensatz
DEVICE_-DEFAULT_FONT	Geräteabhängiger Standardzeichensatz
OEM_FIXED_FONT	Fester OEM-System-Zeichensatz
SYSTEM_FONT	Der System-Zeichensatz, der auch standardmäßig für Textausgabe in Menüs, Dialogboxen etc. verwendet wird
SYSTEM_-FIXED_FONT	Redundanter Zeichensatz, der in alten Windows-Versionen verwendet wurde
DEFAULT_PALETTE	Standard-Farbpalette, bestehend aus 20 Farben
Rückgabewert	Zeiger auf das abgewählte GDI-Objekt

Serialize [CObject]

Methode zur Serialisierung (Laden/Speichern) von Dokumenten und Daten.

```
virtual void Serialize( CArchive& ar );
```

Parameter	Beschreibung
ar	*CArchive*-Objekt, von dem / in das serialisiert werden soll

SetCapture [CWnd]

Fixiert die Maus auf ein Fenster und stellt sicher, dass auch Mausnachrichten an das Fenster geleitet werden, wenn sich der Mauszeiger außerhalb des Fensters befindet.

```
CWnd* SetCapture( );
```

Parameter	Beschreibung
Rückgabewert	Zeiger auf das Fensterobjekt, das zuvor die Mausnachrichten entgegengenommen hat

SetColumnWidth [CListCtrl]

Setzt die Breite einer Spalte einer Liste in Pixeln.

```
BOOL SetColumnWidth(
   int nCol,
   int cx
);
```

Parameter	Beschreibung
nCol	Index der Spalte (beginnend bei 0), deren Breite zu setzen ist
cx	Zu setzende Breite der Spalte in Pixeln
Rückgabewert	Wert ungleich 0 bei Erfolg, sonst 0

SetCursor

Verändert den Mauszeiger und setzt ihn auf einen übergebenen Cursor.

```
HCURSOR SetCursor (HCURSOR hCursor );
```

Parameter	Beschreibung
hCursor	Handle zu einem Mauszeiger
Rückgabewert	Handle zum alten Mauszeiger

SetMapMode [CDC]

Setzt den Abbildungsmodus.

```
int SetMapMode( int nMapMode );
```

Parameter	Beschreibung
nMapMode	Der einzustellende Abbildungsmodus
Rückgabewert	Der zuvor eingestellte Abbildungsmodus

SetPaneText [CStatusBar]

Setzt den Inhalt eines Elements in der Statuszeile auf einen neuen Wert.

```
BOOL SetPaneText(
   int nIndex,
   LPCTSTR lpszNewText,
   BOOL bUpdate = TRUE
);
```

Parameter	Beschreibung
nIndex	Lineare Position des zu verändernden Elements
lpszNewText	Neuer Text für das Statuszeilenelement
bUpdate	Legt fest, ob die Anzeige sofort aktualisiert werden soll
Rückgabewert	Ein Wert ungleich 0 bei Erfolg, sonst 0

SetPixel [CDC]

Setzt einen Bildpunkt an die angegebene Stelle und in der angegebenen Farbe in den Gerätekontext.

```
COLORREF SetPixel(
int x,
          int y,
          COLORREF crColor );
```

Parameter	Beschreibung
x	X-Koordinate des Zielpunkts in logischen Koordinaten

Parameter	Beschreibung
y	Y-Koordinate des Zielpunkts in logischen Koordinaten
crColor	Die zu benutzende Farbe
Rückgabewert	–1, wenn die Funktion fehlschlägt, sonst die beim Zeichnen verwendete Farbe

SetScrollSizes [CScrollView]

Aktualisiert die Größe der Bildlaufleisten. Wird verwendet, wenn sich die Größe des Fensters oder der Umfang des Dokuments verändert hat.

```
void SetScrollSizes(
int nMapMode,
         SIZE sizeTotal,
         const SIZE& sizePage = sizeDefault,
         const SIZE& sizeLine = sizeDefault );
```

Parameter	Beschreibung
nMapMode	Der zu verwendende Abbildungsmodus.
sizeTotal	Die komplette Größe der ScrollView.
sizePage	Die Schrittweite um die gescrollt wird, wenn in die Bildlaufleiste geklickt wird.
sizeLine	Die Schrittweite um die gescrollt wird, wenn einer der Scrollpfeile betätigt wird.

SetTextColor [CDC]

Setzt die Farbe für auszugebenden Text.

```
COLORREF SetTextColor( COLORREF crColor );
```

Parameter	Beschreibung
crColor	Die zu verwendende Farbe.
Rückgabewert	Die zuvor eingestellte Farbe.

SetTimer [CWnd]

Richtet einen Timer ein.

```
UINT SetTimer( UINT nIDEvent,
         UINT nElapse,
         void (CALLBACK EXPORT* lpfnTimer));
```

Parameter	Beschreibung
nIDEvent	Eine selbst gewählte ID ungleich 0, anhand der man mehrere gestartete Timer voneinander unterscheiden kann. In der Behandlungsroutine für Timer wird diese ID als Parameter mit übergeben.
nElapse	Gibt die Millisekunden an, nach denen der Timer jeweils aufgerufen werden soll.
lpfnTimer	Die Behandlungsmethode, die auf Timer-Events reagieren soll. Diese Callback-Methode wird äquivalent zur Nachrichtenfunktion in der Win32-API-Programmierung verwendet. Ist dieser Parameter NULL, wird keine Behandlungsroutine ausgelöst, sondern eine *WM_TIMER*-Botschaft verschickt.
Rückgabewert	Die ID des erzeugten Timer-Events oder 0 im Fall eines Fehlers.

SetViewportExt [CDC]

Setzt Breite und Höhe des Viewports des Gerätekontexts.

```
CSize SetViewportExt( int cx,
            int cy );
```

Parameter	Beschreibung
cx	Breite des Viewports
cy	Höhe des Viewports
Rückgabewert	Die alten Einstellungen des Viewports

SetViewportOrg [CDC]

Setzt den Ursprung des Viewports.

```
CPoint SetViewportOrg( int x,
            int y );
```

Parameter	Beschreibung
x	x-Koordinate des Ursprungs des Viewports
y	y-Koordinate des Ursprungs des Viewports
Rückgabewert	Der alte Ursprung des Viewports

SetWindowExt [CDC]

Setzt Breite und Höhe des Fensters in logischen Einheiten.

```
CSize SetWindowExt(
int cx,
          int cy );
```

Parameter	Beschreibung
cx	Breite des Fensters (in logischen Einheiten)
cy	Höhe des Fensters (in logischen Einheiten)
Rückgabewert	Die alten Ausmaße des Fensters

SetWindowOrg [CDC]

Setzt den Fensterursprung.

```
CPoint SetWindowOrg( int x,
             int y );
```

Parameter	Beschreibung
x	x-Koordinate des Ursprungs des Fensters
y	y-Koordinate des Ursprungs des Fensters
Rückgabewert	Der alte Ursprung des Fensters

ShowWindow [CWnd]

Verändert die Anzeigeart eines Fensters.

```
ShowWindow( int nCmdShow );
```

Parameter	Beschreibung
nCmdShow	Gibt an, wie das Fenster dargestellt werden soll. (Für eine komplette Übersicht der möglichen Werte schauen Sie in Anhang A.)
Rückgabewert	0, falls das Fenster zuvor verdeckt war, sonst 1.

TrackRubberBand [CRectTracker]

Erlaubt dem Benutzer eine Gummibandauswahl durchzuführen. Funktion kehrt erst zurück, sobald der Benutzer die Aktion beendet (Loslassen der linken Maustaste) oder abbricht (Drücken der Esc-Taste).

```
BOOL TrackRubberBand(
   CWnd* pWnd,
   CPoint point,
   BOOL bAllowInvert = TRUE
);
```

Parameter	Beschreibung
pWnd	Zeiger auf das Fenster, in dem der Benutzer die Auswahl durchführen kann
point	Relative Mausposition des Mauszeigers im Auswahlbereich
bAllowInvert	Ist dieser Parameter *true*, können invertierte Auswahlen getätigt werden (d.h., das Rechteck kann von der Startposition aus nach links oben aufgespannt werden).
Rückgabewert	Ein Wert ungleich 0, wenn eine gültige Auswahl getroffen wurde, sonst 0

UpdateAllViews

Aktualisiert alle geöffneten Ansichten eines Dokuments.

```
void UpdateAllViews(
   CView* pSender,
   LPARAM lHint = 0L,
   CObject* pHint = NULL
);
```

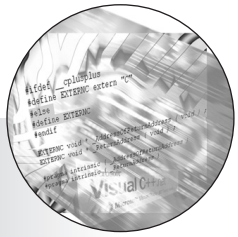
Parameter	Beschreibung
pSender	Ansichtsklasse, die das Dokument verändert hat. Diese Klasse wird nicht zum Neuzeichnen aufgefordert, da sie ja ohnehin schon die aktuellste Version des Dokuments anzeigt. Geben Sie hier NULL an, um sämtliche Ansichten neu zu zeichnen.
lHint	Benutzerdefiniert. Kann weitere Angaben über die Modifizierung enthalten. Nützlich, um redundante Aktualisierungen zu vermeiden.
pHint	Zeiger auf einen benutzerdefinierten Kontext. Kann weitere Angaben über die Modifizierung enthalten. Nützlich, um redundante Aktualisierungen zu vermeiden.

UpdateData [CWnd]

Aktualisiert Daten in Dialogfeldern oder zugehörigen Membervariablen.

```
BOOL UpdateData( BOOL bSaveAndValidate = TRUE );
```

Parameter	Beschreibung
bSaveAndValidate	Gibt an, ob Daten aus Dialogfeldern ausgelesen (*true*) oder in Dialogfelder geschrieben werden sollen (*false*).
Rückgabewert	0, wenn die Operation fehlgeschlagen ist, sonst 1.



INDEX

!

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

Z