1 Die Entwicklungsumgebung

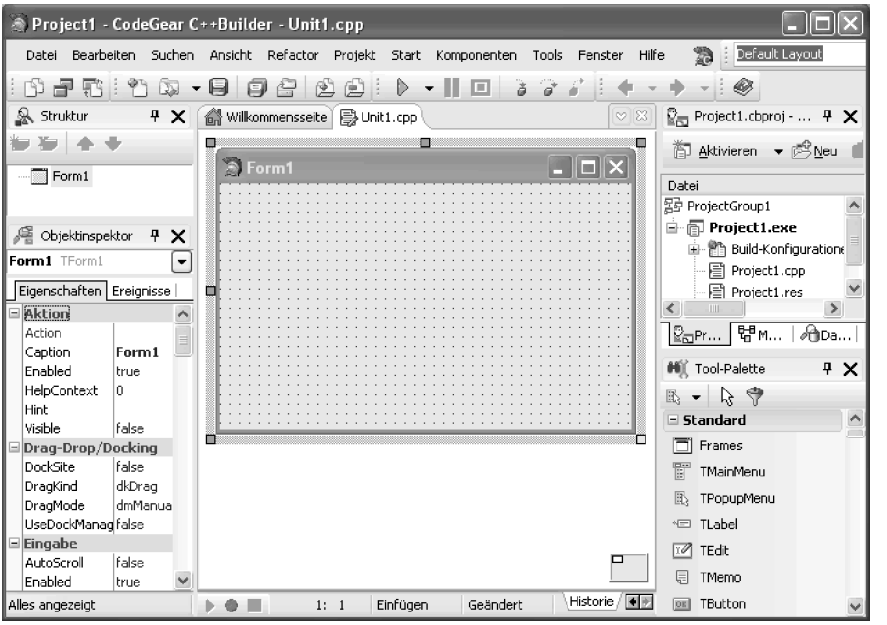
Der C++Builder besteht aus verschiedenen Werkzeugen (Tools), die einen Programmierer bei der Entwicklung von Software unterstützen. Eine solche Zusammenstellung von Werkzeugen zur Softwareentwicklung bezeichnet man auch als Programmier- oder **Entwicklungsumgebung**.

Einfache Entwicklungsumgebungen bestehen nur aus einem Editor und einem Compiler. Für eine effiziente Entwicklung von komplexeren Anwendungen (dazu gehören viele Windows-Anwendungen) sind aber oft weitere Werkzeuge notwendig. Wenn diese wie im C++Builder in einem einzigen Programm integriert sind, spricht man auch von einer **integrierten Entwicklungsumgebung (**engl.: „integrated development environment“, **IDE)**.

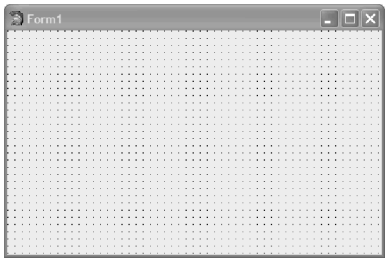
In diesem Kapitel wird zunächst an einfachen Beispielen gezeigt, wie man mit dem C++Builder Windows-Programme mit einer grafischen Benutzeroberfläche entwickeln kann. Anschließend (ab Abschnitt 1.3) werden dann die wichtigsten Werkzeuge des C++Builders ausführlicher vorgestellt. Für viele einfache Anwendungen (wie z.B. die Übungsaufgaben) reichen die Abschnitte bis 1.7. Die folgenden Abschnitte sind nur für anspruchsvollere oder spezielle Anwendungen notwendig. Sie sind deshalb mit dem Zeichen Θ (siehe Seite xxiii) gekennzeichnet und können übergangen werden. Weitere Elemente der Entwicklungsumgebung werden später beschrieben, wenn sie dann auch eingesetzt werden können.

1.1 Visuelle Programmierung: Ein erstes kleines Programm

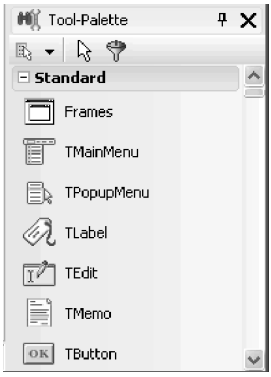
Im C++Builder kann man mit ***Datei|Neu*** Projekte für verschiedene Arten von Anwendungen anlegen. Ein Projekt für ein Windowsprogramm mit einer grafischen Benutzeroberfläche erhält man mit ***VCL-Formularanwendunganwendung – C++Builder***. Anschließend wird die Entwicklungsumgebung mit einigen ihrer Tools angezeigt:



Das **Formular** ist der Ausgangspunkt für alle Windows-Anwendungen, die mit dem C++Builder entwickelt werden. Es entspricht dem Fenster, das beim Start der Anwendung angezeigt wird:

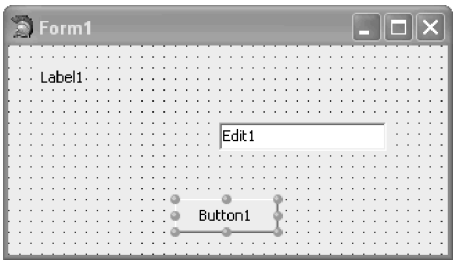


Ein Formular kann mit den in der **Tool-Palette** verfügbaren **Steuerelementen** (**Controls**) gestaltet werden. Die Tool-Palette zeigt praktisch alle der unter Windows üblichen Steuerelemente an, wenn das Formular angezeigt wird. Sie sind auf verschiedene Gruppen (*Standard*, *Zusätzlich* usw.) verteilt, die über die Icons + und – auf- und zugeklappt werden können. Ein Teil dieser Komponenten (wie z.B. ein Button) entspricht Steuerelementen, die im laufenden Programm angezeigt werden. Andere, wie der *Timer* von der Seite *System*, sind im laufenden Programm nicht sichtbar.



Um eine Komponente aus der Tool-Palette auf das Formular zu setzen, klickt man sie zuerst mit der Maus an (sie wird dann als markiert dargestellt). Anschließend klickt man mit dem Mauszeiger auf die Stelle im Formular, an die die linke obere Ecke der Komponente kommen soll.

Beispiel: Nachdem man ein Label (die Komponente mit dem Namen *TLabel*), ein Edit-Fenster (Name *TEdit*) und einen Button (Name *TButton*, mit der Aufschrift *OK*) auf das Formular gesetzt hat, sieht es etwa folgendermaßen aus:



Durch diese Spielereien haben Sie **schon ein richtiges Windows-Programm** erstellt – zwar kein besonders nützliches, aber immerhin. Sie können es folgendermaßen starten:

- mit *Start|Start* von der Menüleiste, oder
- mit *F9* von einem beliebigen Fenster im C++Builder oder
- durch den Aufruf der vom Compiler erzeugten Exe-Datei.

Dieses Programm hat schon viele Eigenschaften, die man von einem Windows-Programm erwartet: Man kann es mit der Maus verschieben, vergrößern, verkleinern und schließen.

Bemerkenswert an diesem Programm ist vor allem der im Vergleich zu einem nichtvisuellen Entwicklungssystem **geringe Aufwand**, mit dem es erstellt wurde. So braucht Petzold in seinem Klassiker „Programmierung unter Windows“ (Petzold 1992, S. 33) ca. 80 Zeilen nichttriviale C-Anweisungen, um den Text „Hello Windows“ wie in einem Label in ein Fenster zu schreiben. Und in jeder dieser 80 Zeilen kann man einiges falsch machen.

Vergessen Sie nicht, Ihr **Programm** zu **beenden**, bevor Sie es weiterbearbeiten. Sie können den Compiler nicht erneut starten, solange das Programm noch läuft.

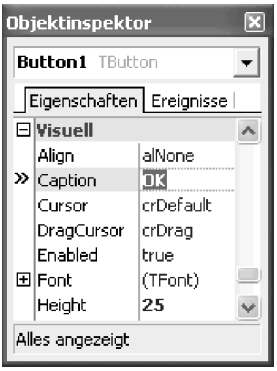
Diese Art der Programmierung bezeichnet man als **visuelle Programmierung**. Während man bei der konventionellen Programmierung ein Programm ausschließlich durch das Schreiben von Anweisungen (Text) in einer Programmier-

sprache entwickelt, wird es bei der visuellen Programmierung ganz oder teilweise aus vorgefertigten grafischen Komponenten zusammengesetzt.

Mit dem C++Builder kann die Benutzeroberfläche eines Programms visuell gestaltet werden. Damit sieht man bereits beim Entwurf des Programms, wie es später zur Laufzeit aussehen wird. Die Anweisungen, die als Reaktionen auf Benutzereingaben (Mausklicks usw.) erfolgen sollen, werden dagegen konventionell in der Programmiersprache **C++** geschrieben.

Die zuletzt auf einem Formular (bzw. im Pull-down-Menü des Objektinspektors) angeklickte Komponente wird als die **aktuell ausgewählte Komponente** bezeichnet. Man erkennt sie an den 8 kleinen blauen Punkten an ihrem Rand. An ihnen kann man mit der Maus ziehen und so die Größe der Komponente verändern. Ein **Formular** wird dadurch zur **aktuell ausgewählten Komponente**, indem man mit der Maus eine freie Stelle im Formular anklickt.

Beispiel: Im letzten Beispiel ist *Button1* die aktuell ausgewählte Komponente.



Der **Objektinspektor** zeigt die Eigenschaften (properties) der aktuell ausgewählten Komponente an. In der linken Spalte stehen die **Namen** und in der rechten die **Werte** der Eigenschaften. Mit der Taste *F1* erhält man eine Beschreibung der Eigenschaft.

Den Wert einer Eigenschaft kann man über die rechte Spalte verändern. Bei manchen Eigenschaften kann man den neuen Wert über die Tastatur eintippen. Bei anderen wird nach dem Anklicken der rechten Spalten ein kleines Dreieck für ein Pull-down-Menü angezeigt, über das ein Wert ausgewählt werden kann. Oder es wird ein Symbol mit drei Punkten „…“ angezeigt, über das man Werte eingeben kann.

Beispiel: Bei der Eigenschaft ***Caption*** kann man mit der Tastatur einen Text eingeben. Bei einem Button ist dieser Text die Aufschrift auf dem Button (z.B. „OK“), und bei einem Formular die Titelzeile (z.B. „Mein erstes C++-Programm“).
Bei der Eigenschaft ***Color*** (z.B. bei einer Edit-Komponente) kann man über ein Pull-down-Menü die **Hintergrundfarbe** auswählen (z.B. ein wunderschönes *clLime*). Bei der Eigenschaft ***Cursor*** kann man über ein Pull-down-Menü die Form des Cursors auswählen, die zur Laufzeit angezeigt wird, wenn der Cursor über dem Steuerelement ist.
Klickt man die rechte Spalte der Eigenschaft ***Font*** und dann das Symbol „…“ an, kann man die **Schriftart** der Eigenschaft *Caption* auswählen.

Eine Komponente auf dem Formular wird nicht nur an ihre Eigenschaften im Objektinspektor angepasst, sondern auch umgekehrt: Wenn man die Größe einer

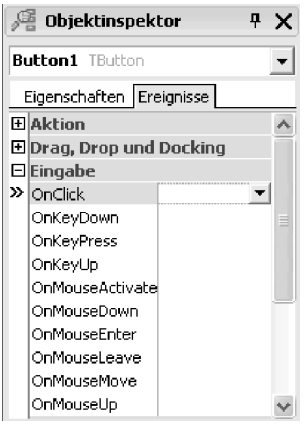
Komponente durch Ziehen an den Ziehquadraten verändert, werden die Werte der entsprechenden Eigenschaften (*Left*, *Top*, *Height* oder *Width*) im Objektinspektor automatisch aktualisiert.

1.2 Erste Schritte in C++

Als nächstes soll das Programm aus dem letzten Abschnitt so erweitert werden, dass als Reaktion auf Benutzereingaben (z.B. beim Anklicken eines Buttons) Anweisungen ausgeführt werden.

Windows-Programme können Benutzereingaben in Form von Mausklicks oder Tastatureingaben entgegennehmen. Im Unterschied zu einfachen Konsolen-Programmen (z.B. DOS-Programmen) muss man in einem Windows-Programm aber keine speziellen Funktionen (wie *Readln* in Pascal oder *scanf* in C) aufrufen, die auf solche Eingaben warten. Stattdessen werden alle Eingaben von Windows zentral entgegengenommen und als sogenannte Botschaften (Messages) an das entsprechende Programm weitergegeben. Dadurch wird in diesem Programm ein sogenanntes **Ereignis** ausgelöst.

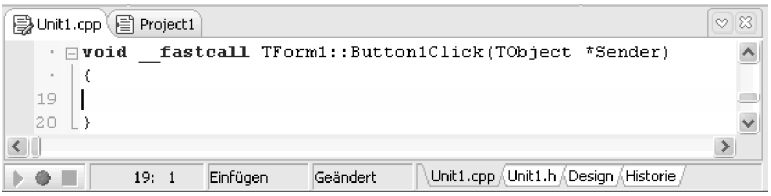
Die Ereignisse, die für die aktuell ausgewählte Komponente eintreten können, zeigt der Objektinspektor an, wenn man das **Register Ereignisse** anklickt.



Die Abbildung rechts zeigt einige Ereignisse für einen Button. Dabei steht *OnClick* für das Ereignis, das beim Anklicken des Buttons eintritt. Offensichtlich kann ein Button nicht nur auf das Anklicken reagieren, sondern auch noch auf zahlreiche andere Ereignisse.

Einem solchen Ereignis kann eine Funktion zugeordnet werden, die dann aufgerufen wird, wenn das Ereignis eintritt. Diese Funktion wird auch als **Ereignisbehandlungsroutine** (engl. **event handler**) bezeichnet. Sie wird vom C++Builder durch einen Doppelklick auf die rechte Spalte des Ereignisses erzeugt und im **Quelltexteditor** angezeigt. Der Cursor steht dann am Anfang der Funktion.

Vorläufig soll unser Programm allerdings nur auf das Anklicken eines Buttons reagieren. Die bei diesem Ereignis aufgerufene Funktion erhält man am einfachsten durch einen Doppelklick auf den Button im Formular. Dadurch erzeugt der C++Builder die folgende Funktion:



Zwischen die geschweiften Klammern „{“ und „}“ schreibt man dann die **Anweisungen**, die ausgeführt werden sollen, wenn das Ereignis *OnClick* eintritt.

Welche Anweisungen hier möglich sind und wie diese aufgebaut werden müssen, ist der Hauptgegenstand dieses Buches und wird ab Kapitel 3 ausführlich beschrieben. Im Rahmen dieses einführenden Kapitels sollen nur einige wenige Anweisungen vorgestellt werden und diese auch nur so weit, wie das zum Grundverständnis des C++Builders notwendig ist. Falls Ihnen Begriffe wie „Variablen“ usw. neu sind, lesen Sie trotzdem weiter – aus dem Zusammenhang erhalten Sie sicherlich eine intuitive Vorstellung, die zunächst ausreicht. Später werden diese Begriffe dann genauer erklärt.

Eine beim Programmieren häufig verwendete Anweisung ist die **Zuweisung** (mit dem Operator „=“), mit der man einer Variablen einen Wert zuweisen kann. Als Variablen sollen zunächst nur solche Eigenschaften von Komponenten verwendet werden, die auch im Objektinspektor angezeigt werden. Diesen Variablen können dann die Werte zugewiesen werden, die auch im Objektinspektor in der rechten Spalte der Eigenschaften vorgesehen sind.

In der Abbildung rechts sieht man einige zulässige Werte für die Eigenschaft *Color* (Abschnitt *Visuell*). Sie werden nach dem Aufklappen des Pull-down-Menüs angezeigt.

Schreibt man jetzt die Anweisung

```
Edit1->Color = clLime;
```

zwischen die geschweiften Klammern

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
Edit1->Color = clLime;
}
```

erhält die Eigenschaft *Color* von *Edit1* beim Anklicken von *Button1* während der Ausführung des Programms den Wert *clLime*, der für die Farbe Limonengrün steht. Wenn Sie das Programm jetzt mit *F9* starten und dann *Button1* anklicken, erhält das Edit-Fenster tatsächlich diese Farbe.

Auch wenn dieses Programm noch nicht viel sinnvoller ist als das erste, haben Sie doch gesehen, wie mit dem C++Builder Windows-Anwendungen entwickelt werden. Dieser **Entwicklungsprozess** besteht immer aus den folgenden Aktivitäten:

1. Man gestaltet die Benutzeroberfläche, indem man Komponenten aus der Tool-Palette auf das Formular setzt (drag and drop) und ihre Eigenschaften im Objektinspektor oder das Layout mit der Maus anpasst (visuelle Programmierung).
2. Man schreibt in C++ die Anweisungen, die als Reaktion auf Benutzereingaben erfolgen sollen (nichtvisuelle Programmierung).
3. Man startet das Programm und testet, ob es sich auch wirklich so verhält, wie es sich verhalten soll.

Der Zeitraum der Programmentwicklung (Aktivitäten 1. und 2.) wird auch als **Entwurfszeit** bezeichnet. Im Unterschied dazu bezeichnet man die Zeit, während der ein Programm läuft, als **Laufzeit** eines Programms.

1.3 Der Quelltexteditor

Der Quelltexteditor (kurz: **Editor**) ist das Werkzeug, mit dem die Quelltexte geschrieben werden. Er ist in die Entwicklungsumgebung integriert und kann auf verschiedene Arten aufgerufen werden, wie z.B.

- durch Anklicken eines Registers, wie z.B. 

oder 

- über *Ansicht|Units* oder *F12*
- durch einen Doppelklick auf die rechte Spalte eines Ereignisses im Objektinspektor. Der Cursor befindet sich dann in der Ereignisbehandlungsroutine für dieses Ereignis.
- durch einen Doppelklick auf eine Komponente in einem Formular. Der Cursor befindet sich dann in einer bestimmten Ereignisbehandlungsroutine dieser Komponente.

Da die letzten beiden Arten den Cursor in eine bestimmte Ereignisbehandlungsroutine platzieren, bieten sie eine einfache Möglichkeit, diese Funktion zu finden, ohne sie im Editor suchen zu müssen.

Der Editor enthält über Tastenkombinationen zahlreiche Funktionen, mit denen sich nahezu alle Aufgaben effektiv durchführen lassen, die beim Schreiben von Programmen auftreten. In der ersten der nächsten beiden Tabellen sind einige der Funktionen zusammengestellt, die man auch in vielen anderen Editoren findet.

Tastenkürzel	Aktion oder Befehl
Strg+F	wie *Suchen|Suchen*
F3	wie *Suchen|Suche wiederholen*
Strg+R	wie *Suchen|Ersetzen*
Strg+S	wie *Datei|Speichern*
Strg+Entf	löscht das Wort ab der Cursorposition
Strg+Y	löscht die gesamte Zeile
Strg+Rücktaste	löscht das Wort links vom Cursor
Strg+Umschalt+Y	löscht die Zeile ab dem Cursor bis zum Ende
Alt+Rücktaste oder *Strg+Z*	wie *Bearbeiten|Rückgängig*. Damit können Editor-Aktionen rückgängig gemacht werden
Alt+Umschalt+ Rücktaste oder *Strg+Umschalt+Z*	wie *Bearbeiten|Wiederherstellen*
Pos1 bzw. *Ende*	an den Anfang bzw. das Ende der Zeile springen
Strg+Pos1 bzw. *Strg+Ende*	an den Anfang bzw. das Ende der Datei springen
Strg+← bzw. *Strg+*→	um ein Wort nach links bzw. rechts springen
Strg+Bild↑ bzw. *Strg+Bild*↓	an den Anfang bzw. das Ende der Seite springen
Strg+↑ bzw. *Strg+*↓	Text um eine Zeile nach oben bzw. unten verschieben; die Position des Cursors im Text bleibt gleich
Einfg	schaltet zwischen Einfügen und Überschreiben um

Dazu kommen noch die üblichen **Tastenkombinationen unter Windows,** wie Markieren eines Textteils mit gedrückter Umschalt-Taste und gleichzeitigem Bewegen des Cursors bzw. der Maus bei gedrückter linker Maustaste. Ein markierter Bereich kann mit *Strg+X* ausgeschnitten, mit *Strg+C* in die Zwischenablage kopiert und mit *Entf* gelöscht werden. *Strg+V* fügt den Inhalt der Zwischenablage ein.

Die nächste Tabelle enthält Funktionen, die vor allem beim Programmieren nützlich sind, und die man in einer allgemeinen Textverarbeitung nur selten findet. Einige dieser Optionen werden auch in einer Symbolleiste (*Ansicht|Symbolleisten*) angezeigt:

Tastenkürzel	Aktion oder Befehl
F9 oder 	kompilieren und starten, wie *Start*|*Start*
Strg+F9	kompilieren, aber nicht starten
Strg+F2 oder 	Laufendes Programm beenden, wie *Start*|*Programm abbrechen*. Damit können oft auch Programme beendet werden, die mit  nicht beendet werden können. Versuchen Sie immer zuerst diese Option wenn Sie meinen, Sie müssten den C++Builder mit dem Windows Task Manager beenden.
F1 bzw. *Strg+F1*	wie *Hilfe*|*Borland-Hilfe*, oft kontextsensitiv
Strg+Enter	falls der Text unter dem Cursor einen Dateinamen darstellt, wird diese Datei geöffnet
Strg+Umschalt+I bzw. *Strg+Umschalt+U*	rückt den als Block markierten Text eine Spalte nach links bzw. rechts (z.B. zum Aus- und Einrücken von {}-Blöcken)
Alt+[bzw. *Alt+]* bei einer amerikanischen Tastatur und *Strg+Q+Ü* bei einer deutschen	setzt den Cursor vor die zugehörige Klammer, wenn er vor einer Klammer (z.B. (), {}, [ ] oder <>) steht
Strg+#	einen markierten Block mit // auskommentieren bzw. die Auskommentierung entfernen
rechte Maustaste, *Umgeben*	einen markierten Block in einen /*...*/-Kommentar oder eine Anweisung einfügen
rechte Maustaste, *Einblenden/Ausblenden*	ganze Funktionen, Klassen usw. auf- oder zuklappen
F11	wie *Ansicht*|*Objektinspektor*
F12	wie *Ansicht*|*Umschalten Formular/Unit*
Alt+0	zeigt Liste offenen Fenster, wie *Ansicht*|*Fensterliste*

Alt+Maus bewegen bzw. *Alt+Umschalt+* Pfeiltaste *(←, →, ↑* oder *↓)*	zum Markieren von Spalten , z.B.
	`/// <summary>`
	`/// Clean up any`
	`/// </summary>`
	`/// <param name=`

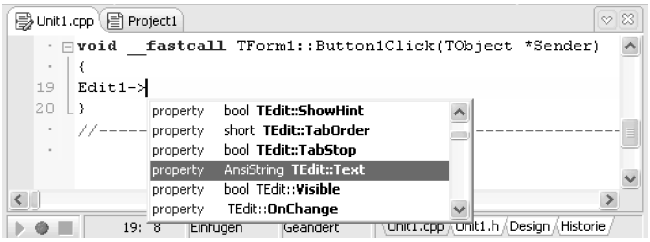
Strg+K+n (n eine Ziffer)	setzt oder löscht die Positionsmarke n (wie die Positionsmarken-Befehle im Kontextmenü)
Strg+n	springt zur Positionsmarke n
Strg+Tab bzw. *Strg+Umschalt+Tab*	zeigt das nächste bzw. das vorherige Editor-Fenster an

Eine ausführlichere Beschreibung der Tastaturbelegung findet man in der Online-Hilfe (*Hilfe*|*Borland Hilfe*) unter *Hilfe*|*Inhalt*|*Borland Hilfe*|*Developer Studio 2006 (Allgemein)*|*Referenz*|*Tastenzuordnungen*|*Standard-Tastaturvorlage*.

Die folgenden Programmierhilfen beruhen auf einer Analyse des aktuellen Programms. Sie werden zusammen mit einigen weiteren sowohl unter dem Oberbegriff **Code Insight** als auch unter dem Oberbegriff **Programmierhilfe** zusammengefasst:

- **Code-Vervollständigung**: Nachdem man den Namen einer Komponente (genauer: eines Klassenobjekts bzw. eines Zeigers auf ein Klassenobjekt) und den zugehörigen Operator („.“ oder „->“) eingetippt hat, wird eine Liste mit allen Elementen der Klasse angezeigt. Aus dieser Liste kann man mit der Enter-Taste ein Element auswählen.
- **Code-Parameter**: Zeigt nach dem Eintippen eines Funktionsnamens und einer öffnenden Klammer die Parameter der Funktion an
- **Symbolinformation durch Kurzhinweis**: Wenn man mit der Maus über einen Namen für ein zuvor definiertes Symbol fährt, werden Informationen über die Deklaration angezeigt.
- Wenn die Option *Tools|Optionen|Editor-Optionen|Programmierhilfe|Quelltext Template Vervollständigung* aktiviert ist, wird nach dem Eintippen eines Wortes aus der mit *Ansicht|Templates* angezeigten Liste und einem Tab- bzw. Leerzeichen der Code entsprechend vervollständigt.

Beispiel: Wenn das Formular eine Edit-Komponente *Edit1* enthält, wird nach dem Eintippen von „Edit1->“ eine Liste mit allen Elementen von *Edit1* angezeigt:



Tippt man weitere Buchstaben ein, werden nur die Elemente mit diesen Anfangsbuchstaben angezeigt.

Falls Sie einen langsamen Rechner und große Programme haben, können diese Programmierhilfen den C++Builder unangenehm langsam machen. Dann kann man sie unter *Tools|Optionen|Editor-Optionen|Programmierhilfe* abschalten.

Der linke Rand im Editor enthält Zeilennummern und grüne und gelbe Linien. Sie bedeuten, dass der Text in der aktuellen Sitzung geschrieben bzw. noch nicht gespeichert ist. Vor ausführbaren Anweisungen stehen blaue Punkte.

Wenn man einen Block markiert, der mindestens einen Bezeichner mehrfach enthält, wird das **Sync-Bearbeitungsmodus** Symbol  angezeigt. Klickt man es

an, werden alle solchen Bezeichner hervorgehoben. Klickt man dann einen dieser hervorgehobenen Bezeichner an, werden Änderungen an diesem Bezeichner auch mit allen anderen durchgeführt.

Unter *Tools|Optionen|Editor-Optionen* gibt es zahlreiche Möglichkeiten, den Editor individuell anzupassen. Insbesondere kann die Tastaturbelegung von einigen verbreiteten Editoren eingestellt werden.

Falls in den Editoroptionen *Sicherungsdateien erstellen* markiert ist, speichert der Editor die letzten 10 (bzw. die unter *Anzahl Dateisicherungen* eingetragene Anzahl) Versionen. Diese können dann im History-Fenster angezeigt und miteinander verglichen werden.

Aufgabe 1.3

Schreiben Sie einen kleinen Text im Editor und probieren Sie die Tastenkombinationen aus, die Sie nicht schon kennen. Insbesondere sollten Sie zumindest einmal gesehen haben, wie man

a) Änderungen rückgängig machen kann,
b) rückgängig gemachte Änderungen wiederherstellen kann,
c) einen markierten Block ein- und ausrücken kann,
d) nach dem Eintippen von „Memo1->“ aus der Liste der Elemente „Lines“ auswählen kann und dann nach dem Eintippen von „->“ auch noch „Add“. Wenn der Cursor dann hinter der Klammer „(“ steht, sollte der Parametertyp *AnsiString* angezeigt werden. Sie brauchen im Moment noch nicht zu verstehen, was das alles bedeutet. Sie benötigen aber ein Formular mit einem Memo *Memo1* und müssen das alles in einer Funktion wie *Button1Click* eingeben.
e) mit *Strg+#* einen Block auskommentieren und dies wieder rückgängig machen kann,
f) mit *F11* zwischen den verschiedenen Fenstern wechseln kann und
g) mit *Strg+Eingabe* eine Datei „c:\test.txt“ öffnen kann, wenn sich der Cursor über diesem Text befindet. Dazu müssen Sie zuvor eine Datei mit diesem Namen anlegen, z.B. mit *notepad*.

1.4 Kontextmenüs und Symbolleisten (Toolbars)

Einige der häufiger gebrauchten Menüoptionen stehen auch über Kontextmenüs und Symbolleisten zur Verfügung. Damit kann man diese Optionen etwas schneller auswählen als über ein Menü.

Eine Symbolleiste (Toolbar) ist eine Leiste mit grafischen Symbolen (Icons), die unterhalb der Menüleiste angezeigt wird. Diese Symbole stehen für Programm-

optionen, die auch über die Menüleiste verfügbar sind. Durch das Anklicken eines Symbols kann man sie mit einem einzigen Mausklick auswählen. Das ist etwas schneller als die Auswahl über ein Menü, die mindestes zwei Mausklicks erfordert. Symbolleisten können außerdem zur Übersichtlichkeit beitragen, da sie Optionen zusammenfassen, die inhaltlich zusammengehören.

Der C++Builder enthält einige vordefinierte Symbolleisten. Mit *Ansicht|Symbolleisten* kann man diejenigen auswählen, die man gerade braucht, sowie eigene Symbolleisten konfigurieren. Einige Optionen der *Standard* Symbolleiste wurden schon im Zusammenhang mit dem Editor vorgestellt:

Standard Toolbar

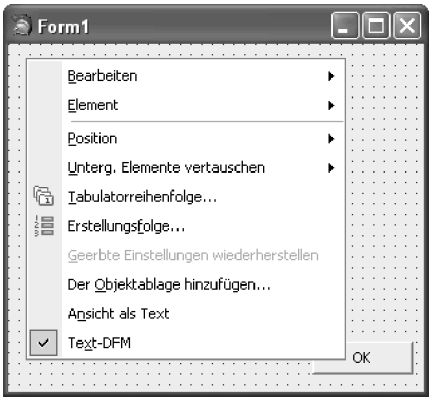
Debug Toolbar

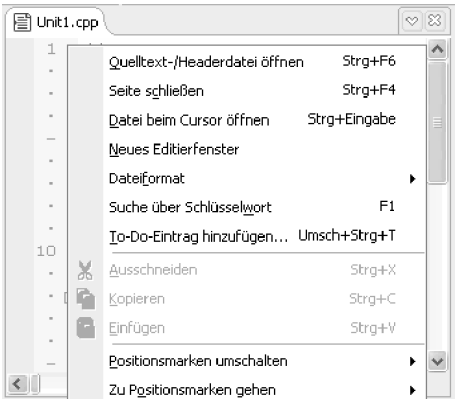
Falls Ihnen die relativ kleinen Symbole nicht viel sagen, lassen Sie den Mauszeiger kurz auf einer Schaltfläche stehen. Dann wird die entsprechende Option in einem kleinen Fenster beschrieben.

Die Symbolleisten können über *Ansicht|Symbolleisten|Anpassen* angepasst werden: Die unter *Anweisungen* angebotenen Optionen kann man auf eine Symbolleiste ziehen, und durch Ziehen an einer Option auf einer Symbolleiste kann man sie von der Symbolleiste entfernen. Falls man eine Option versehentlich entfernt, kann man die Symbolleiste mit *Zurücksetzen* wieder in den ursprünglichen Zustand versetzen.

Über die rechte Maustaste erhält man in den meisten Fenstern des C++Builders ein sogenanntes **Kontextmenü** (auch die Bezeichnung „**lokales Menü**“ ist verbreitet), das eine Reihe gebräuchlicher Optionen für dieses Fenster anbietet.

Beispiele: Links das Kontextmenü in einem Formular und rechts das im Editor:





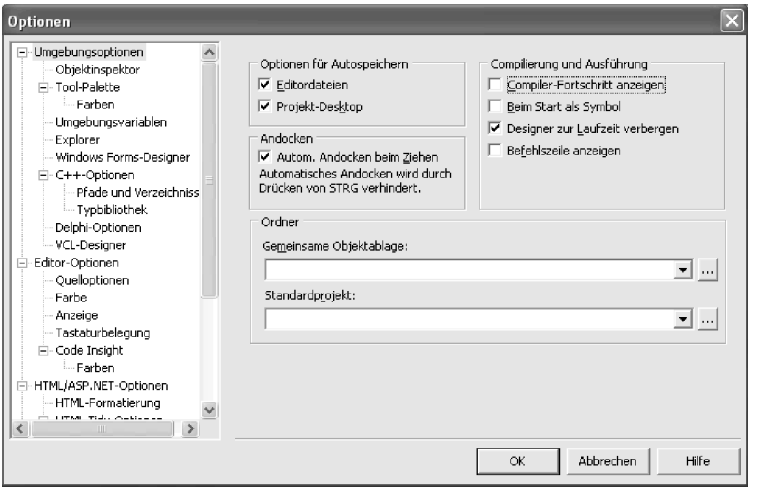
Über die Option „Ansicht als Text“ kann man ein Formular auch als Text darstellen. Da nur die Abweichungen von den Voreinstellungen angezeigt werden, erhält man damit leicht einen Überblick über die geänderten Eigenschaften.

1.5 Projekte, Projektdateien und Projektoptionen

Es empfiehlt sich, ein **Projekt** oder zumindest die gerade bearbeiteten Dateien **regelmäßig** zu **speichern**. Man kann nie ausschließen, dass sich Windows aufhängt oder ein Stromausfall die Arbeit seit dem letzten Speichern zunichte macht.

Das ist z.B. mit *Datei|Alles speichern* bzw.  möglich. Falls dieses Symbol oder die entsprechende Menüoption nicht aktiviert ist, wurden seit dem letzten Speichern keine Dateien verändert.

Da man aber meist nur dann an das Speichern denkt, wenn es schon zu spät ist, empfiehlt sich die Verwendung der unter *Tools|Optionen|Umgebungsoptionen* angebotenen „Optionen für Autospeichern“:



Markiert man hier „Editordateien“, werden vor jedem Start des Programms (z.B. mit *F9*) alle zum Projekt gehörenden Dateien gespeichert. Markiert man außerdem noch die Option „Projekt-Desktop“, werden beim nächsten Start des C++Builders wieder alle die Dateien geöffnet, die beim letzten Beenden geöffnet waren.

In diesem Zusammenhang wird außerdem empfohlen, unter *Tools|Editor-Optionen|Editor* die Option „Rückgängig nach Speichern“ zu markieren. Sonst können nach dem Speichern keine Änderungen mit *Bearbeiten|Rückgängig* wieder rückgängig gemacht machen.

Beim erstmaligen Speichern fragt der C++Builder zuerst nach einem Namen für alle zum Projekt gehörenden Units und dann nach einem Namen für das Projekt.

– Für jedes Formular erzeugt der C++Builder einige Dateien, deren Namen sich aus dem für die **Unit** eingegebenen Namen und den folgenden Endungen zusammensetzt:

  .cpp In diese Datei schreibt der C++Builder die Ereignisbehandlungsroutinen wie *Button1Click*. Sie wird vom Programmierer durch weitere Anweisungen ergänzt.
  .h Eine Header-Datei mit der Klassendefinition des Formulars.
  .obj Eine sogenannte Object-Datei, die vom Compiler aus der cpp- und h-Datei erzeugt wird.
  .dfm Diese Textdatei enthält eine Beschreibung des Formulars mit allen visuellen Komponenten und ihren Eigenschaften, die z.B. im Objektinspektor gesetzt wurden. Aus diesen Informationen wird das Formular beim Start des Programms erzeugt.

– Der für das **Projekt** eingegebene Name wird für Dateien mit diesen Endungen verwendet:

  .cpp Das sogenannte **Hauptprogramm** mit der ***WinMain*** Funktion, das automatisch vom C++Builder angelegt und verwaltet wird. Es sollte normalerweise nicht manuell verändert werden.
  .bdsproj Die Projekt-Datei mit den Projekteinstellungen.
  .res Die sogenannte Ressourcen-Datei.
  .obj Die vom Compiler aus dem Hauptprogramm erzeugte Object-Datei.
  .exe Das vom **Linker** aus den Object-Dateien des Projekts und den „lib“-Bibliotheken erzeugte **ausführbare Programm**.
  Der Linker ist wie der Compiler ein in die Entwicklungsumgebung integriertes Programm, das automatisch mit *Start|Start* aufgerufen wird. Bei vielen C++Builder-Projekten braucht man allerdings nicht einmal zu wissen, dass der Linker überhaupt existiert. Normalerweise ist es kein Fehler, wenn man sich vorstellt, dass das ausführbare Programm allein vom Compiler erzeugt wird.

– In Abhängigkeit von der Build Konfiguration (*Projekt|Build-Konfigurationen*) werden einige dieser Dateien beim C++Builder 2006 in den Unterverzeichnissen *Debug_Build* oder *Release_Build* bzw. *Debug* und *Release* im C++Builder 2007 angelegt.

– Für Projekt-Verzeichnisse sollten Pfade mit den Zeichen „–“ oder „+“ vermieden werden (wie z.B. „c:\C++-Programme“). Solche Namen können seltsame Fehlermeldungen des Linkers verursachen, die keinen Hinweis auf die Ursache geben.

Angesichts der relativ großen Anzahl von Dateien, die zu einem Projekt gehören, liegt es nahe, jedes Projekt in einem eigenen Verzeichnis zu speichern. Das ist bei größeren Projekten mit mehreren Units auch meist empfehlenswert.

Falls man aber viele kleinere Projekte hat (wie z.B. die später folgenden Aufgaben), ist es meist einfacher, mehrere Projekte in einem gemeinsamen Verzeichnis zu speichern. Dazu kann man folgendermaßen vorgehen:

- Da sowohl zum Projektnamen als auch zum Namen der Unit eine Datei mit der Endung „.cpp“ angelegt wird, müssen für **das Projekt und die Units verschiedene Namen** gewählt werden.
- Damit man alle Dateien eines Projekts dann auch einfach im Windows Explorer kopieren kann (um z.B. auf einem anderen Rechner daran weiterzuarbeiten), wählt man **am einfachsten** Namen, die mit derselben Zeichenfolge beginnen. Damit die Namen des Projekts und der Unit verschieden sind, reicht es aus, wenn sie sich im letzten Buchstaben unterscheiden (z.B. durch ein zusätzliches „U“ für die Unit). Diese Konventionen wurden auch für die meisten Lösungen auf der Buch-CD gewählt.

Zum Speichern von Dateien gibt es außerdem noch die folgenden Optionen:

- *Datei|Projekt speichern unter* speichert die Dateien mit dem Projektnamen unter einem neuen Namen. Diese Option bedeutet nicht, dass alle Dateien eines Projekts gespeichert werden. Wenn man mit dieser Option ein komplettes Projekt kopieren will, um auf einem anderen Rechner daran weiterzuarbeiten, wird man feststellen, dass die Dateien mit den Units fehlen.
- *Datei|Speichern* bzw. *Strg+S* bzw.  speichert die derzeit im Editor angezeigte Unit einschließlich der zugehörigen Header- und Formulardatei.

Beim Aufruf von *Projekt|Projekt compilieren* (*Strg+F9*) bzw. *Start|Start* (*F9*) übersetzt der Compiler alle seit dem letzten Aufruf geänderten Quelltextdateien usw. neu. Dann wird der Linker aufgerufen, der aus den dabei erzeugten Object-Dateien eine ausführbare Exe-Datei erzeugt.

Dabei werden jedes Mal Dateien mit den Endungen „.tds“ und „.obj“ neu erzeugt, die recht groß werden können. Diese Dateien **kann man löschen** (im C++Builder 2007 mit *Projekt|Bereinigen*). Bei einer Kopie des Projekts sind sie nicht notwendig. Im C++Builder 2006 sind sie in den Unterverzeichnissen *Debug_Build* und *Release_Build*. In älteren Versionen des C++Builders befinden sie sich im Projektverzeichnis.

Unter *Projekt|Optionen* kann man zahlreiche Einstellungen für den Compiler, den Linker, die Anwendung usw. vornehmen. Die voreingestellten Werte sollte man aber nur dann ändern, wenn man sich über deren Konsequenzen im Klaren ist. Die wichtigsten Konfigurationen sind die **Debug**- und **Release Konfiguration**, die beide aus einem Satz von Optionen bestehen. **Eigene Konfigurationen** kann man

mit *Projekt|Build-Konfiguration|Neu* bzw. *Kopieren* anlegen. Für jede solche Konfiguration kann man dann eigene Projektoptionen setzen. Damit kann man einfach zwischen verschiedenen Konfigurationen umschalten, ohne die Standard-Konfigurationen verändern zu müssen.

Anmerkungen für Delphi-Programmierer: Der Projekt-Datei mit der Endung „.cpp“ entspricht in Delphi die „.dpr“-Datei des Projekts. Der Header-Datei und der cpp-Datei einer Unit entsprechen in Delphi der Interface-Teil der Implementationsteil einer “.pas”-Unit.

1.6 Einige Tipps zur Arbeit mit Projekten

Die Arbeit mit dem C++Builder ist meist einfach, wenn man alles richtig macht. Es gibt allerdings einige typische Fehler, über die Anfänger immer wieder stolpern. Die ersten fünf der folgenden Tipps sollen helfen, diese Fehler zu vermeiden. Die übrigen sind einfach oft nützlich.

1) Ein **neues Projekt** wird mit *Datei|Neu|VCL-Formularanwendung* angelegt und nicht mit *Datei|Neu|Formular*.

   Mit *Datei|Neu|Formular* wird dem aktuellen Projekt ein neues Formular hinzugefügt. Dieses Formular kann man wie das Hauptformular (das erste Formular) des Projekts gestalten, so dass man diesen Fehler zunächst gar nicht bemerkt. Es wird aber beim Start des Programms nicht automatisch angezeigt. Das hat dann die verzweifelte Frage zur Folge „Wo ist mein neues Programm, bei mir startet immer nur das alte“.

2) Es ist meist empfehlenswert, unter *Tools|Optionen|Umgebungsoptionen|Optionen für Autospeichern* die Option „**Projekt-Desktop**“ zu **markieren**. Dann wird beim nächsten Start des C++Builders automatisch das Projekt geöffnet, das beim letzten Beenden des C++Builders geöffnet war. Alle Fenster sind genau so angeordnet wie beim letzten Mal, und der Cursor blinkt an derselben Stelle. So kann man sofort da weiterarbeiten, wo man aufgehört hat.

   Falls man *Autospeichern Projekt Desktop* markiert hat, sollte man aber beim Beenden des C++Builders keinen übertriebenen Ordnungssinn an den Tag legen und den Schreibtisch aufräumen, indem man alle Fenster mit den Formularen und Quelltextdateien schließt. Dann werden sie nämlich beim nächsten Start des C++Builders nicht angezeigt, was zu dem entsetzten Aufschrei „Meine Dateien sind weg“ führen kann. Mit *Ansicht|Formulare* werden sie wieder angezeigt.

3) Zum **Öffnen** eines früher angelegten **Projekts** werden vor allem die Optionen *Datei|Zuletzt verwendet* und *Datei|Projekt öffnen* empfohlen. Die erste zeigt im

oberen Teil des Untermenüs die zuletzt geöffneten Projekte an, und mit der zweiten kann man auf den Laufwerken nach einem Projekt suchen.

Mit *Datei|Öffnen* kann man dagegen sowohl Projekte als auch andere Dateien öffnen. Da in der Voreinstellung viele verschiedene Dateitypen angezeigt werden und die Symbole für Projekte und andere Dateitypen leicht verwechselt werden können, wird mit dieser Option gelegentlich auch eine Unit anstelle der Projektdatei geöffnet. Das hat dann wie unter 1). zur Folge, dass sich Änderungen in der Unit nach dem Kompilieren nicht auf das laufende Programm auswirken.

4) Solange man noch nicht weiß, welche Ereignisse es gibt und wann diese eintreten (siehe z.B. Abschnitt 2.6), sollte man nur Ereignisbehandlungsroutinen verwenden, die wie

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
}
```

auf das **Anklicken** eines **Buttons** reagieren. Eine Funktion wie

```
void __fastcall TForm1::Edit1Change(TObject *Sender)
{
}
```

die man durch einen Doppelklick auf ein Eingabefeld *Edit1* erhält, wird zur Laufzeit des Programms bei jeder Änderung des Textes im Edit-Fenster aufgerufen. Das ist aber meist nicht beabsichtigt.

5) Um eine vom **C++Builder erzeugte Funktion** zu löschen, sollte man sie **nicht manuell löschen**. Vielmehr muss man nur den Text **zwischen den geschweiften Klammern { } löschen. Dann entfernt der C++Builder die Funktion automatisch beim nächsten Kompilieren.**

Die Nichtbeachtung dieser Regel kann eine Menge Ärger nach sich ziehen. Falls z.B. versehentlich durch einen Doppelklick auf ein Edit-Fenster die Funktion

```
void __fastcall TForm1::Edit1Change(TObject *Sender)
{
}
```

erzeugt wurde und (nachdem man festgestellt hat, dass sie nicht den beabsichtigten Effekt hat) diese dann manuell aus dem Quelltext gelöscht wird, erhält man die Fehlermeldung

```
LinkerFehler Unresolved external 'TForm1::Edit1Change(..
```

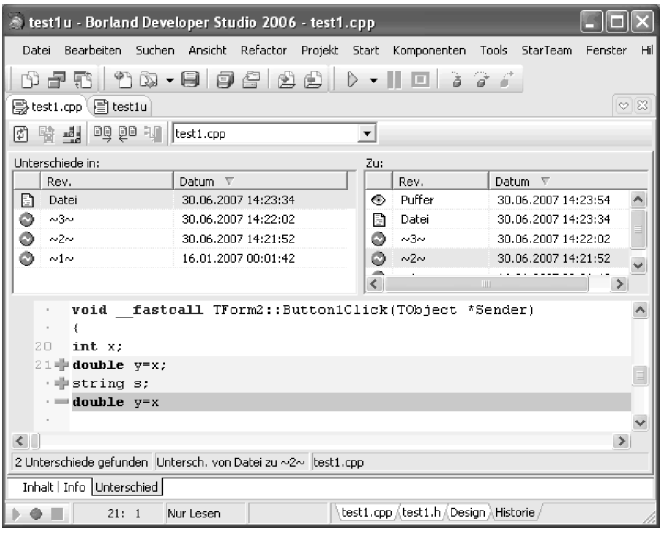
Nach einem solchen Fehler macht man am einfachsten alle bisherigen Eingaben mit *Alt+Rücktaste* wieder rückgängig, bis die Funktion wieder angezeigt wird,

oder man fängt das gesamte Projekt nochmals neu an. Falls das nicht möglich oder zu aufwendig ist, kann man auch in der Header-Datei zur Unit (die man mit *Strg+F6* im Editor erhält) die folgende Zeile durch die beiden Zeichen „//“ auskommentieren:

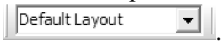
```
// void __fastcall Edit1Change(TObject *Sender);
```

Ansonsten wird aber von jeder Änderung dieser Datei abgeraten, solange man sich über ihre Bedeutung nicht klar ist.

6) Nach dem Anklicken des *Historie*-Registers im Editor kann man sich die Unterschiede des aktuellen Textes und einer der 10 letzten Versionen anzeigen lassen. Diese werden beim Speichern automatisch im versteckten Unterverzeichnis *__history* angelegt.



7) Das manchmal doch recht lästige **automatische Andocken** von Fenstern kann man durch Markieren von „Andocken“ unter *Tools|Optionen|Umgebungsoptionen* unterbinden.

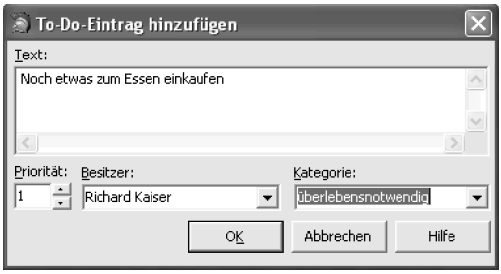
8) Mit *Ansicht|Desktops* kann man eine Desktop-Einstellung mit einer bestimmten Auswahl und Anordnung von IDE-Fenstern speichern und laden. Während der Laufzeit eines Programms im Debugger wird der mit *Ansicht|Desktops|Debug-Desktop einstellen* gesetzte Debug-Desktop angezeigt. Nach dem Ende des Debuggers wird der zuvor verwendete Desktop wieder angezeigt. Den Desktop der Voreinstellung erhält man mit 

.

9) Falls Sie schon mit früheren Versionen des C++Builders gearbeitet haben und Ihnen der ältere Desktop besser gefällt, können Sie diesen mit den folgenden beiden Einstellungen herstellen:

- *Ansicht|Desktops|Classic Undocked* und die Markierung bei
- *Tools|Optionen|VCL Designer|Eingebetteter Designer* entfernen

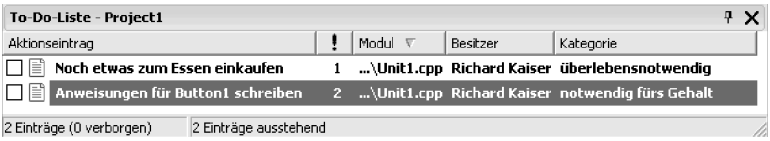
10) Mit *To-Do-Eintrag hinzufügen* kann man der **To-Do-Liste** einen Eintrag hinzufügen und damit einige Merkzettel sparen:



Dieser Eintrag wird dann an der aktuellen Cursor-Position als Kommentar (siehe Abschnitt 3.16) in den Quelltext eingefügt:

```
/* TODO 1 -oRichard Kaiser -cüberlebensnotwendig : Noch
etwas zum Essen einkaufen */
/* TODO 2 -oRichard Kaiser -cnotwendig fürs Gehalt :
Anweisungen für Button1 schreiben */
```

Die Einträge der To-Do-Liste werden dann mit *Ansicht|To-Do-Liste* angezeigt:



11) Dieser Tipp ist nur dann von Bedeutung, wenn man ein Projekt mit **verschiedenen Versionen des C++Builders** bearbeiten will (z.B. an der Uni mit dem C++Builder 5 und zuhause mit dem C++Builder 2006).

Eine neuere Version des C++Builders kann die Projektdateien einer älteren Version lesen und konvertiert sie in das Format der neueren Version. Da das aber dann von der älteren Version nicht mehr gelesen werden kann, kann man das Projekt anschließend nicht mehr mit der älteren Version bearbeiten.

Falls die Anweisungen in den Units von beiden Versionen des C++Builders übersetzt werden können (das ist oft möglich), kann man dieses Problem dadurch umgehen, dass man für jede Version des C++Builders ein eigenes Projekt anlegt und in beiden Projekten dieselben Units verwendet. Dazu kann man folgendermaßen vorgehen:

1. Für das zuerst angelegte Projekt wählt man mit *Datei|Projekt speichern unter* einen Namen, der z.B. die Versionsnummer des C++Builders als letztes Zeichen enthält (z.B. *AufgP5* beim C++Builder 5).
2. Für die andere Version des C++Builders legt man mit *Datei|Neu|VCL-Formularanwendung* ein neues Projekt mit einem entsprechenden Namen an. Aus diesem Projekt entfernt man dann die vom C++Builder automatisch erzeugte Unit mit *Projekt|Aus dem Projekt entfernen* und nimmt anschließend mit *Projekt|Dem Projekt hinzufügen* die Unit aus dem zuerst angelegten Projekt in dieses Projekt auf.

1.7 Die Online-Hilfe

Da sich kaum jemand die vielen Einzelheiten des C++Builders und von C++ merken kann, ist es für eine effektive Arbeit unerlässlich, die Online-Hilfe nutzen zu können.

Am einfachsten ist oft die kontextbezogene Hilfe mit ***F1***: In den meisten Fenstern der Entwicklungsumgebung erhält man mit *F1* Informationen, wie z.B.

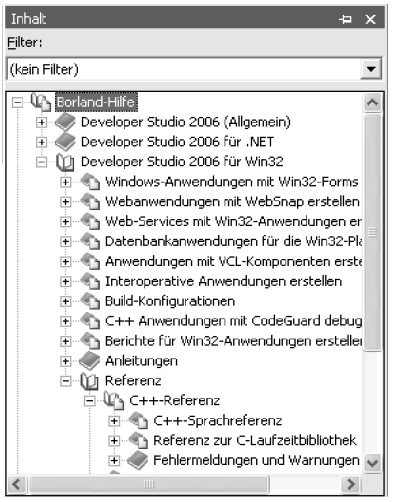
– im Editor zum Wort unter dem Cursor
– im Objektinspektor zur angewählten Eigenschaft
– auf einem Formular zum angeklickten Steuerelement, usw.

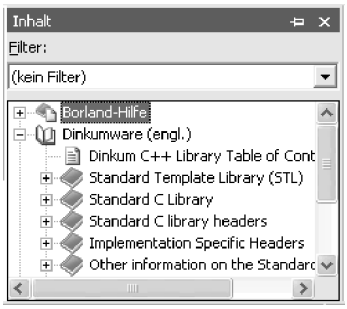
Das Borland Developer Studio verwendet den Microsoft Document Explorer zur Anzeige der Online-Hilfe. Er wird mit *Hilfe|Borland-Hilfe* gestartet und bietet über sein *Hilfe*-Menü zahlreiche Optionen zur Anzeige von Informationen.

Falls man das Wort kennt, zu dem man weitere Informationen sucht, kann man mit ***Hilfe|Index*** die Online-Hilfe dazu aufrufen. Beachten Sie, dass die Namen der meisten **VCL-Klassen mit** dem Buchstaben **„T“ beginnen**, d.h. dass Sie die Informationen zu einem Button, Label usw. unter *TButton*, *TLabel* usw. finden.

Oft kennt man aber den entsprechenden Indexeintrag nicht. Für diesen Fall bietet das *Hilfe* Menü einige Optionen an, mit denen man dann hoffentlich weiterkommt.

Über ***Hilfe|Inhalt*** kann man in thematisch geordneten Büchern, Anleitungen, Referenzen usw. suchen. Die Online-Hilfe zu den **C- und C++-Standardbibliotheken**, auf die später öfter verwiesen wird, findet man unter *Dinkumware*:





Unter *Inhalt|Borland Hilfe|Developer Studio 2006 für Win32|Referenz|VCL für Win32 (C++)* findet man die Beschreibung der **VCL-Klassen**. Diese Informationen werden in Abschnitt 2.1 noch ausführlicher beschrieben.

Die Microsoft Dokumentation zu Win32 findet man sowohl unter *Inhalt|Microsoft Platform SDK*, als auch in einer anderen Gliederung nach einem Klick auf *Inhalt|Microsoft Platform SDK* im rechten Fenster (Windows Server 2003 Family) unter *Contents*. Hier findet man auch die **Win32-API** unter *Windows API*.

Einige weitere Optionen des Document Explorers:

- bzw. *Hilfe|Inhalt synchronisieren* synchronisiert das Inhaltsverzeichnis mit der angezeigten Seite.
- Mit einem Filter kann man die angezeigten Suchergebnisse reduzieren.
- *Hilfe|Suchen* (Volltextsuche) zeigt Seiten an, die den Suchbegriff enthalten.
- Mit „Zu Favoriten hinzufügen“ im Kontextmenü einer Hilfeseite kann man **Lesezeichen** setzen.

Einen Teil dieser Informationen findet man auch in den beiden pdf-Dateien im Help-Verzeichnis, die auf der Willkommensseite (*Ansicht|Willkommens-Seite*) als Benutzerhandbuch und Sprachreferenz angeboten werden. Auf dieser Seite findet man unter ***Dokumentation|Anleitungen*** auch einige nützliche Anleitungen (z.B. unter *Anleitung|Einführung* oder *Anleitungen|Anleitungen für Win32*).

Die Online-Hilfe des C++Builders 2006 hat einige Schwächen. Deshalb hat Borland auch noch die Online-Hilfe zum C++Builder 6 unter *http://dn.codegear.com/article/34064* zur Verfügung gestellt. Im C++Builder 2007 ist die Online-Hilfe besser.

Der Zugriff auf spezielle Inhalte der Online-Hilfe ist in nahezu jeder Version des C++Builders anders. Für die Leser, die mit einer älteren Version arbeiten, hier deshalb kurz die wichtigsten Zugriffspfade:

Win32-API:	*Hilfe|Windows SDK* (C++Builder 5/6)
C++ Standardbibliothek:	*Hilfe|STLport-Hilfe* (C++Builder 6)
	Start|Programme|Borland C++Builder 5|Hilfe|-Standard C++ Bibliothek (C++Builder 5)
Weitere Hilfedateien:	*Start|Programme|Borland C++Builder 5/6|Hilfe*

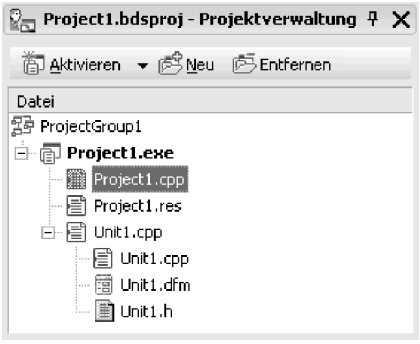
Aufgabe 1.7

Mit diesen Übungen sollen Sie lediglich die Möglichkeiten der Online-Hilfe kennen lernen. Sie brauchen die angezeigten Informationen nicht zu verstehen.

a) Rufen Sie mit *F1* die Online-Hilfe auf
   – für das Wort „int“ im Editor
   – für ein Edit-Feld auf einem Formular; und
   – im Objektinspektor für die Eigenschaft *Text* eines Edit-Feld.
b) Suchen Sie in *Inhalt* unter *Borland Hilfe|Developer Studio 2006 für Win32|-Referenz|C++-Referenz|C++-Sprachreferenz|Sprachstruktur|Deklarationssyntax|Grundlegende Typen|*“ nach einer Übersicht über „Grundlegende Typen“.

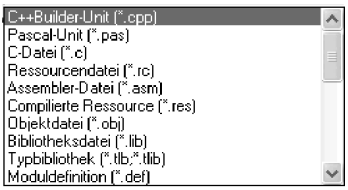
1.8 Projektgruppen und die Projektverwaltung Θ

Die Projektverwaltung (*Ansicht|Projektverwaltung*) bietet zahlreiche Optionen zur Verwaltung und Konfiguration von Projekten. Sie zeigt alle Dateien an, die zu einem Projekt gehören, und enthält verschiedene Kontextmenüs, je nachdem, ob man ein Projekt, eine Quelltextdatei usw. anklickt.



Das Kontextmenü eines Projekts enthält unter anderem diese beiden Optionen:

– Mit *Hinzufügen* (wie *Projekt|Dem Projekt hinzufügen*) kann man einem Projekt eine Datei hinzufügen. Der C++Builder bietet hier die folgenden Dateitypen an:



Falls diese Datei eine

- **Quelltextdatei** ist (z.B. mit der Endung .cpp, .pas, .c), wird sie beim Kompilieren des Projekts mitkompiliert und die dabei erzeugte Object-Datei zum Projekt gelinkt.
- **Bibliothek** ist (z.B. mit der Endung .lib, obj), wird sie zum Projekt gelinkt.

Diese Option darf nicht mit einer *#include*-Anweisung (siehe Abschnitt 3.22.1) verwechselt werden: Eine Quelltextdatei oder eine Bibliothek wird normalerweise einem Projekt hinzugefügt, während die zugehörige Header-Datei mit einer *#include*-Anweisung in eine der Quelltextdateien des Projekts aufgenommen wird.

- Unter *Build-Ereignisse* kann man Anweisungen festlegen, die vor oder nach dem Kompilieren bzw. vor dem Linken ausgeführt werden sollen. Diese Anweisungen können Befehle für die Eingabeaufforderung sein.

Falls man an verschiedenen Projekten arbeitet, die von einander abhängig sind, ist es meist bequemer, sie alle gemeinsam zu öffnen, zu kompilieren und zu schließen. Das ist mit einer sogenannten **Projektgruppe** möglich.

Eine Projektgruppe fasst ein oder mehrere Projekte zusammen. Sie wird entweder mit *Datei|Neu|Weitere|Andere Dateien|Projektgruppe* angelegt, oder indem man in der Projektverwaltung eines Projekts die Projektgruppe über das Kontextmenü speichert. Über das Kontextmenü kann man ihr ein existierendes oder ein neues Projekt hinzufügen bzw. Projekte aus ihr entfernen.

Das Projekt, das mit dem nächsten *Start|Start (F9)* erzeugt und ausgeführt wird, heißt das **aktive Projekt**. Man kann es in der Projektverwaltung mit einem Doppelklick oder mit *Aktivieren* aus dem Kontextmenü festlegen. Die nächsten beiden Optionen des *Projekt* Menüs erzeugen das aktive Projekt:

- *Projekt erzeugen* berücksichtigt alle Dateien, unabhängig davon, ob sie geändert wurden oder nicht.
- *Projekt compilieren* berücksichtigt alle Dateien, die seit dem letzten Kompilieren verändert wurden. Diese Option ist normalerweise ausreichend, obwohl es auch Situationen gibt, in denen *Projekt erzeugen* notwendig ist.

Mit den folgenden Optionen des *Projekt* Menüs werden alle Projekte einer Projektgruppe erzeugt:

- *Alle Projekte erstellen*: wie *Projekt erzeugen* für alle Projekte
- *Alle Projekte aktualisieren*: wie *Projekt compilieren* für alle Projekte

Aufgabe 1.8

Erzeugen Sie eine Projektgruppe mit dem Namen „MeineProjektgruppe“ und eine VCL Formularanwendung „MeinProjekt1“. Es ist nicht notwendig, irgendwelche Komponenten auf das Formular zu setzen. Damit man die Anwendungen später unterscheiden kann, soll die Eigenschaft *Caption* des Formulars auf „Projekt 1“ gesetzt werden. Verschaffen Sie sich mit dem Windows-Explorer nach jeder der folgenden Teilaufgaben einen Überblick über die in den verschiedenen Verzeichnissen erzeugten Dateien.

a) Führen Sie *Start*|*Start* (*F9*) aus.
b) Ändern Sie die Konfiguration von *Debug* zu *Release* und führen Sie *Start*|*Start* (*F9*) aus.
c) Fügen Sie *MeineProjektgruppe* mit der Projektverwaltung (*Ansicht*|*Projektverwaltung*) ein weiteres Projekt mit dem Namen *MeinProjekt2* hinzu. Setzen Sie die Eigenschaft *Caption* des Formulars auf „Projekt 2“.
d) Wechseln Sie zwischen den aktiven Projekten und führen Sie danach jeweils *Start*|*Start* (*F9*) aus. Danach wie d).

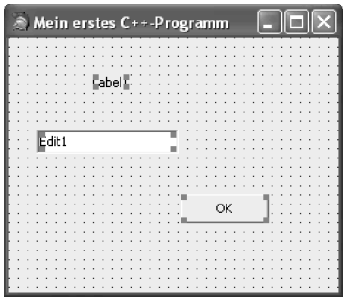
1.9 Hilfsmittel zur Gestaltung von Formularen Θ

Für die Gestaltung von Formularen stehen über das Menü *Bearbeiten*, das Kontextmenü im Formular sowie die Symbolleisten **Ausrichten** und **Abstand** zahlreiche Optionen zur Verfügung.


Symbolleiste *Ausrichten*


Symbolleiste *Abstand*

Die meisten dieser Optionen können auf eine Gruppe von markierten Steuerelementen angewandt werden. Dazu klickt man auf eine freie Stelle im Formular und fasst sie durch Ziehen mit der gedrückten linken Maustaste zusammen.



Die Reihenfolge, in der die Steuerelemente des Formulars während der Ausführung des Programms mit der Tab-Taste angesprungen werden (Tab-Ordnung), kann man über die entsprechende Option im Kontextmenü des Formulars mit den Pfeiltasten einstellen:

Aufgabe 1.9

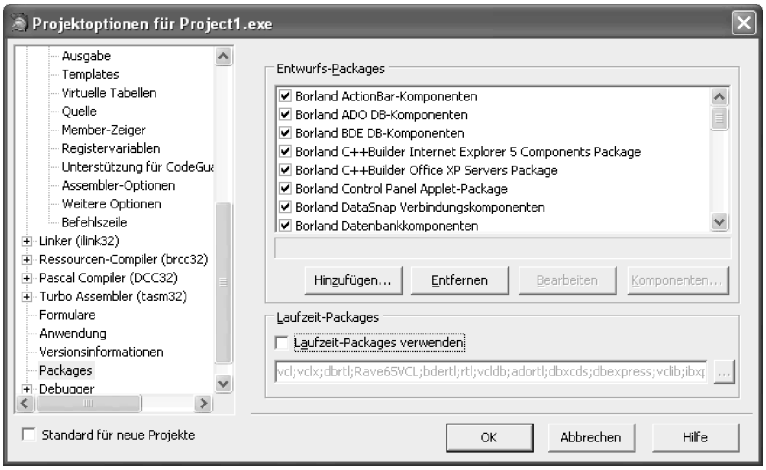
Setzen Sie einige Komponenten (z.B. zwei Buttons und ein Label) in unregelmäßiger Anordnung auf ein Formular. Bringen Sie sie vor jeder neuen Teilaufgabe wieder in eine unregelmäßige Anordnung.

a) Ordnen Sie alle Komponenten an einer gemeinsamen linken Linie aus.
b) Geben Sie allen Komponenten dieselbe Breite.
c) Verändern Sie die Tabulator-Ordnung.

1.10 Packages und eigenständig ausführbare Programme Θ

Viele Programme, die mit dem C++Builder entwickelt werden, verwenden gemeinsame Funktionen und Komponenten wie z.B. ein Formular oder einen Button. Wenn man ihren Code in jede Exe-Datei aufnimmt, wird sie relativ groß. Deswegen fasst man häufig benutzte Komponenten oder Funktionen oft in Bibliotheken (meist sogenannte DLLs) zusammen. Der C++Builder verwendet spezielle DLLs, die als **Packages** bezeichnet werden.

Wenn unter *Projekt*|*Optionen*|*Packages* die Option „Mit Laufzeit-Packages aktualisieren“ markiert ist, verwendet das vom C++Builder erzeugte Programm die angegebenen Laufzeit-Packages.



In der Voreinstellung ist diese Option markiert. Ein einfaches Programm mit einem Button ist dann ca. 20 KB groß, im Unterschied zu etwa 200 KB ohne Packages.

Damit man ein Programm auf einem Rechner ausführen kann, auf dem der C++-Builder nicht installiert ist, erzeugt man es meist am einfachsten so, dass es keine Packages und keine DLLs für die dynamische RTL (CC3270MT.DLL und BorlndMM.DLL) verwendet. Dann reicht zum Start des Programms die Exe-Datei aus, und man braucht die relativ großen DLLs nicht (z.B. 1,6 MB für die immer benötigte VCL100.BPL). Auf einem Rechner mit dem C++Builder sind die Packages vorhanden, da sie bei der Installation des C++Builders in das System-Verzeichnis von Windows kopiert werden.

Fassen wir zusammen:

- Wenn man ein Programm nur auf dem Rechner ausführen will, auf dem man es entwickelt, kann man mit Packages Speicherplatz sparen. Das ist die Voreinstellung. Sie ist für die zahlreichen Übungsaufgaben in diesem Buch normalerweise am besten.
- Wenn man ein Programm dagegen auf einem Rechner ausführen will, auf dem der C++Builder nicht installiert ist, muss man entweder alle notwendigen Bibliotheken zur Verfügung stellen und dabei darauf achten, dass man keine vergisst. Oder man erstellt das Programm ohne Laufzeit-Packages und ohne dynamische Laufzeitbibliothek, indem man die nächsten beiden Optionen **nicht markiert:**

  1. *Projekt*|*Optionen*|*Packages*|***Mit Laufzeit-Packages aktualisieren*** und
  2. *Projekt*|*Optionen*|*Linker*|*Linken*|***Dynamische RTL verwenden***

1.11 Win32-API und Konsolen-Anwendungen Θ

Mit dem C++Builder kann man nicht nur Windows-Programme schreiben, sondern auch sogenannte Konsolen-Anwendungen und Windows-Programme auf der Basis der Win32 API. Obwohl auf solche Programme in diesem Buch nicht weiter eingegangen wird, soll hier kurz skizziert werden, wie man solche Anwendungen entwickelt.

1.11.1 Konsolen-Anwendungen Θ

Eine Konsolen-Anwendung verwendet wie ein DOS-Programm ein Textfenster für Ein- und Ausgaben. Im Unterschied zu einem Programm für eine grafische Benutzeroberfläche erfolgen Ein- und Ausgaben vor allem zeichenweise über die Tastatur und den Bildschirm. Solche Programme werden meist von der Kommandozeile aus gestartet. Obwohl eine Konsolen-Anwendung wie ein DOS-Programm aussieht, ist es nur unter Win32 und nicht unter MS-DOS lauffähig.

Mit dem C++Builder erhält man ein Projekt für eine solche Anwendung mit *Datei|Neu|Weitere|C++Builder-Projekte|Konsolenanwendung*. Daraufhin wird eine Datei „Unit.cpp“ angelegt, die eine Funktion mit dem Namen *main* enthält:

```
int main(int argc, char* argv[])
{
   return 0;
}
```

Sie wird beim Start eines Konsolen-Programms aufgerufen. Die Anweisungen, die vom Programm ausgeführt werden sollen, werden dann vor *return* eingefügt.

Ein- und Ausgaben erfolgen bei einem Konsolen-Programm vor allem über die in <iostream> vordefinierten Streams

cin // for input from the keyboard
cout // for output to the screen

mit den Ein- und Ausgabe-Operatoren „<<“ und „>>“:

```
#include <iostream> // für cin und cout notwendig
using namespace std;
int main(int argc, char* argv[])
{
  int x,y;
  cout<<"x="; // der Anwender soll einen Wert eingeben
  cin>>x;     // den Wert einlesen
  cout<<"y=";
  cin>>y;
  cout<<"x+y="<<(x+y);
  return 0;
}
```

Dieses einfache Beispiel zeigt bereits einen wesentlichen Unterschied zu den bisher entwickelten Programmen für eine grafische Benutzeroberfläche wie Windows. Hier erfolgen alle Ein- und Ausgaben sequenziell: Ein Wert für y kann erst eingegeben werden, nachdem ein Wert für x eingegeben wurde. Damit ein Anwender Fehleingaben korrigieren kann, sind relativ aufwendige Programmkonstruktionen notwendig, die durch weitere sequenzielle Eingaben realisiert werden müssen. Sowohl die Programmierung als auch die Bedienung von Konsolenprogrammen ist oft ziemlich aufwendig. Im Unterschied dazu kann ein Programm für eine grafische Benutzeroberfläche mehrere Eingabefelder gleichzeitig anzeigen. Mit der Maus kann dann jedes direkt adressiert werden.

Zur Formatierung der Ausgabe kann man die Manipulatoren und Funktionen verwenden, die in Abschnitt 4.3.5 vorgestellt werden:

```
for (int i=0; i<10; i++)
  cout <<setw(10)<<i<<setw(10)<<i*i<<endl;
```

Funktionen wie *printf* aus der Programmiersprache C können auch verwendet werden:

```
for (int i=0; i<10; i++)
  printf("%d   %d \n",i,i*i);
```

Ein Windows-Programm besitzt im Unterschied zu einem Textfenster-Programm keine vordefinierten Streams zur Ein- und Ausgabe. Verwendet man *cin* oder *cout* in einem Windows-Programm, bleibt das deshalb ohne sichtbares Ergebnis.

Aufgabe 1.11

Schreiben Sie eine einfache Konsolen-Anwendung, die „Hello world“ am Bildschirm ausgibt.

1.11.2 Der Start des Compilers von der Kommandozeile Θ

Der Compiler des C++Builders kann unter dem Namen *bcc32* auch von einer Kommandozeile aus gestartet werden:

```
c:\CBuilder\bin\bcc32 test.cpp –DTEST –IC:\MyIncludes
```

Die zahlreichen Parameter erhält man mit der Option –h (z.B. –D für Präprozessor-Makros und –I für *#include*-Suchpfade).

1.11.3 Win32-API Anwendungen Θ

Mit dem C++Builder kann man auch Windows-Anwendungen auf der Basis der Win32 API (Application Programmers Interface) entwickeln. Die erste Version dieser Bibliothek wurde ca. 1992 mit Windows NT veröffentlicht und basiert auf

der Programmiersprache C. Wer damit auch heute noch programmieren will, sei auf die alten Ausgaben der Bücher von Charles Petzold „Programming Windows“ verwiesen.

Normalerweise gibt es heutzutage keinen Grund, neue Projekte auf dieser Basis zu beginnen. Es ist aber gelegentlich notwendig, alte Projekte, die auf dieser Bibliothek basieren, weiterzuentwickeln. Das ist mit dem C++Builder – zumindest **im Prinzip** – möglich: Man muss lediglich das Hauptprogramm eines C++Builder-Programms (z.B. Project1.cpp) durch das Programm mit der *WinMain*-Funktion ersetzen. Da moderne C/C++-Compiler aber sehr viel schärfere Prüfungen als ältere Compiler durchführen, ist die Wahrscheinlichkeit groß, dass alte Programme nicht mehr akzeptiert werden, obwohl sie von älteren Compilern anstandslos kompiliert werden.

1.12 Windows-Programme und Units Θ

Ein C++-**Windowsprogramm** unterscheidet sich von einem C++-**Konsolenprogramm** unter anderem dadurch, dass bei seinem Start die Funktion ***WinMain*** und nicht ***main*** aufgerufen wird. Sie befindet sich in einer Datei, die der C++Builder für jedes Projekt anlegt. Ihr Name setzt sich aus dem Namen, der für das beim Speichern Projekt gewählt wird, und der Namenserweiterung „.cpp“ zusammen mit.

Zum Beispiel hat der C++Builder nach dem Speichern eines Projekts unter dem Namen „Project1“ die folgende Datei „Project1.cpp“ erzeugt:

```
#include <vcl.h>
#pragma hdrstop
//---------------------------------------------------------
USEFORM("Unit1.cpp", Form1);
//---------------------------------------------------------
```

```
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
   try
   {
      Application->Initialize();
      Application->CreateForm(__classid(TForm1), &Form1);
      Application->Run();
   }
   catch (Exception &exception)
   {
      Application->ShowException(&exception);
   }
   catch (...)
   {
      try
      {
         throw Exception("");
      }
      catch (Exception &exception)
      {
         Application->ShowException(&exception);
      }
   }
   return 0;
}
```

Beim Start des Programms werden dann die folgenden Anweisungen der Reihe nach ausgeführt:

- Zuerst wird ***Application->Initialize*** aufgerufen.
- Durch den Aufruf von ***Application->CreateForm*** wird das **Formular** *Form1* so **erzeugt**, wie es mit der Entwicklungsumgebung gestaltet wurde.
- Der Aufruf von ***Application->Run*** integriert das Programm in das Botschaftensystem von Windows: Es kann dann **auf die Ereignisse reagieren**, für die entsprechende Ereignisbehandlungsroutinen definiert sind. Das sind entweder **vom C++Builder vordefinierte Funktionen** oder aber **Funktionen** wie *Button1Click***,** die **in den Units** definiert wurden.

Das Programm läuft dann so lange, bis es durch ein entsprechendes Ereignis beendet wird (z.B. durch Anklicken der Option *Schließen* im Systemmenü oder durch die Tastenkombination *Alt-F4*). Diese beiden Reaktionen gehören zu den vom C++Builder vordefinierten Reaktionen.

Normalerweise hat ein Programmierer, der mit dem C++Builder arbeitet, allerdings nichts in der Datei mit der Funktion *WinMain* verloren. Diese Datei wird vom C++Builder erzeugt und verwaltet und sollte normalerweise nicht verändert werden. Stattdessen betätigt man sich bei der Arbeit mit dem C++Builder meist in einer **Unit**. Dabei kann es sich um eine Unit handeln, die vom C++Builder für ein Formular erzeugt wurde, oder eine, die der Programmierer selbst angelegt hat.